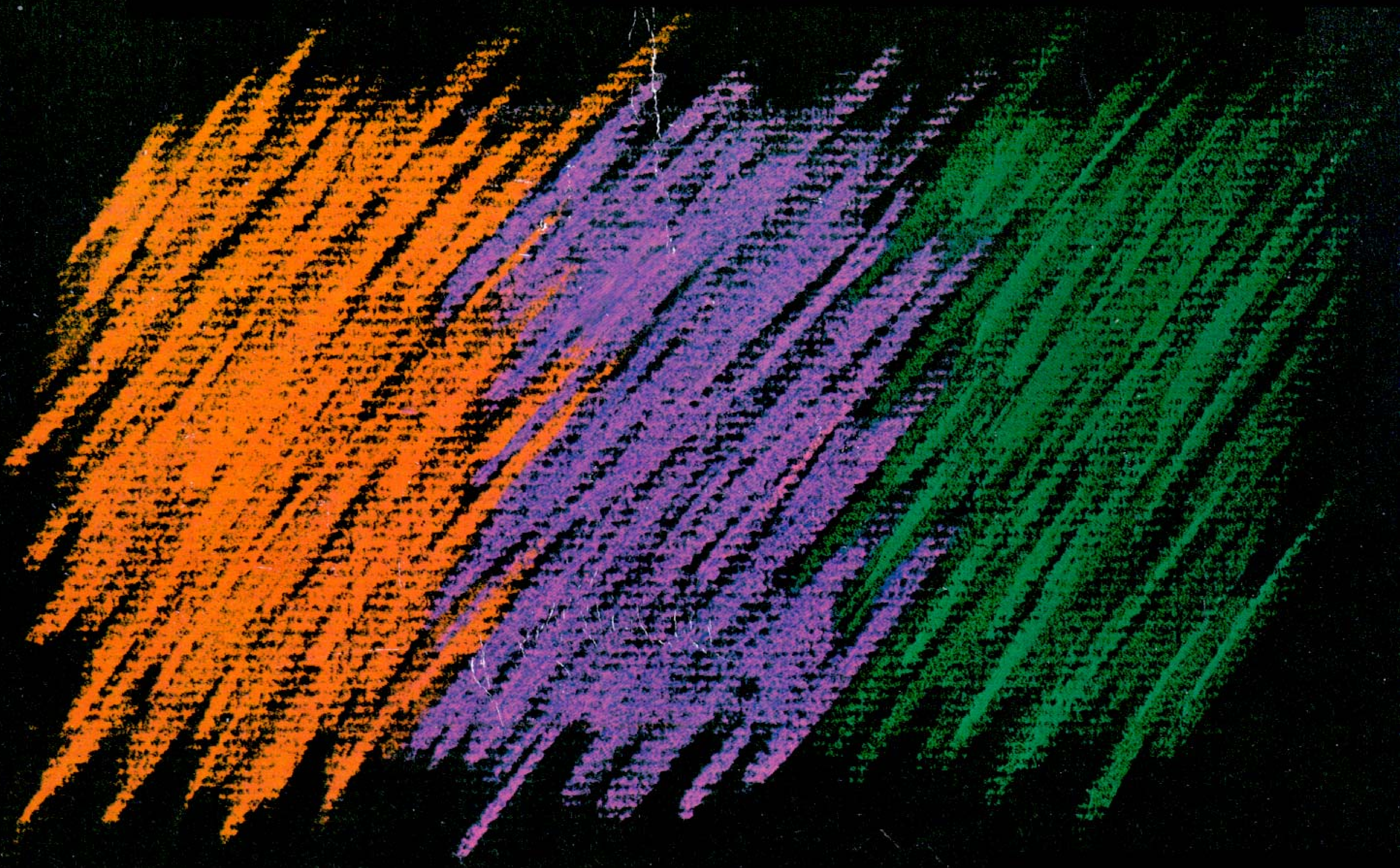
Osborne/McGraw-Hill



Z80® ASSEMBLY LANGUAGE SUBROUTINES

Lance A. Leventhal
Winthrop Saville

Z80®
Assembly Language Subroutines

Z80®
Assembly Language Subroutines

Lance A. Leventhal
Winthrop Saville

Osborne/McGraw-Hill
Berkeley, California

Published by
Osborne/McGraw-Hill
2600 Tenth Street
Berkeley, California 94710
U.S.A.

For information on translations and book distributors outside of the U.S.A., please write to Osborne/ McGraw-Hill at the above address.

Z80® ASSEMBLY LANGUAGE SUBROUTINES

234567890 DODO 8987654

ISBN 0-931988-91-8

Cover by Jean Lake

Text design by Paul Butzler

Contents

Preface

This book is intended to serve as a source and a reference for the assembly language programmer. It contains an overview of assembly language programming for a particular microprocessor and a collection of useful subroutines. In the subroutines, a standard format, documentation package, and parameter passing techniques were used. The rules of the most popular assemblers have been followed, and the purpose, procedure, parameters, results, execution time, and memory usage of each routine have been described.

The overview sections summarize assembly language programming for those who do not have the time or need for a complete textbook; the Assembly Language Programming series provides more extensive discussions. Chapter 1 introduces assembly language programming for the particular processor and summarizes the major features that make this processor different from other microprocessors and minicomputers. Chapter 2 shows how to implement instructions and addressing modes that are not explicitly available. Chapter 3 describes common programming errors.

The collection of subroutines emphasizes common tasks that occur in many applications. These tasks include code conversion, array manipulation, arithmetic, bit manipulation, shifting functions, string manipulation, sorting, and searching. We have also provided examples of I/O routines, interrupt service routines, and initialization routines for common family chips such as parallel interfaces, serial interfaces, and timers. You should be able to use these programs as subroutines in actual applications and as starting points for more complex programs.

This book is intended for the person who wants to use assembly language immediately, rather than just learn about it. The reader could be

- An engineer, technician, or programmer who must write assembly language programs for a design project.
- A microcomputer user who wants to write an I/O driver, a diagnostic program, a utility, or a systems program in assembly language.

- An experienced assembly language programmer who needs a quick review of techniques for a particular microprocessor.
- A systems designer who needs a specific routine or technique for immediate use.
- A high-level language programmer who must debug or optimize programs at the assembly level or must link a program written in a high-level language to one written in assembly language.
- A maintenance programmer who must understand quickly how specific assembly language programs work.
- A microcomputer owner who wants to understand the operating system for a particular computer or who wants to modify standard I/O routines or systems programs.
- A student, hobbyist, or teacher who wants to see examples of working assembly language programs.

This book can also serve as a supplement for students of the Assembly Language Programming series.

This book should save the reader time and effort. The reader should not have to write, debug, test, or optimize standard routines or search through a textbook for particular examples. The reader should instead be able to obtain easily the specific information, technique, or routine that he or she needs. This book has been organized and indexed for rapid use and reference.

Obviously, a book with such an aim demands feedback from its readers. Although all the programs have been thoroughly tested and carefully documented, please inform the publisher if you find any errors. If you have suggestions for better methods or for additional topics, routines, programming hints, or index entries, please tell us about them. We have used our programming experience to develop this book, but your help is needed to improve it. We would greatly appreciate your comments, criticisms, and suggestions.

NOMENCLATURE

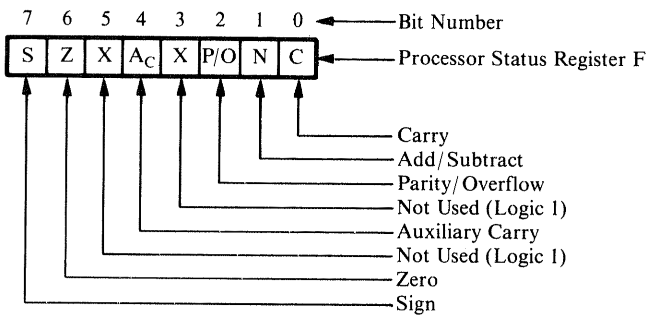
We have used the following nomenclature in this book to describe the architecture of the Z80 processor, to specify operands, and to represent general values of numbers and addresses.

Z80 Architecture

Byte-length registers include

A (accumulator)	R (refresh)
B	A′
C	B′
D	C′
E	D′
H	E′
L	H′
F (flags)	L′
I (interrupt vector)	F′

Of these, the primary user registers are the first seven: A, B, C, D, E, H, and L. The I (interrupt vector) register contains the more significant byte (page number) of interrupt service addresses in Interrupt Mode 2. The R (refresh) register contains a memory refresh counter. The F (flag) register consists of a set of bits with independent functions and meanings, organized as shown in the following diagram:



Register pairs and word-length registers include

AF	(Accumulator and flags, accumulator most significant)
AF′	(Registers A′ and F′, A′ most significant)
BC	(Registers B and C, B most significant)
BC′	(Registers B′ and C′, B′ most significant)
DE	(Registers D and E, D most significant)
DE′	(Registers D′ and E′, D′ most significant)
HL	(Registers H and L, H most significant)
HL′	(Registers H′ and L′, H′ most significant)
IX	(Index register X or IX)
IY	(Index register Y or IY)
PC	(Program counter)
SP	(Stack pointer)

Flags include

Add/Subtract (N)
Carry (C)
Auxiliary Carry (A_C)

Parity/Overflow (P/O or P/V)
Sign (S)
Zero (Z)

These flags are arranged in the F register as shown previously.

Miscellaneous facilities include

Interrupt Flip-flop 1 (IFF1)
Interrupt Flip-flop 2 (IFF2)

Z80 Assembler

Delimiters include

:	After a label, except for EQU, DEFL, and MACRO, which require a space
space	After an operation code
,	Between operands in the operand (address) field
;	Before a comment
(,)	Around memory references

All operands are treated as data unless they are enclosed in parentheses.

Pseudo-Operations include

DB or DEFB	Define byte; place byte-length data in memory.
DEFL	Define label (may be redefined later).
DEFM	Define string; place ASCII data in memory.
DS or DEFS	Define storage; allocate bytes of memory.
DW or DEFW	Define word; place word-length data in memory.
END	End of program.
EQU	Equate; define the attached label.
ORG	Set origin; place subsequent object code starting at the specified address.

Designations include

Number systems:

B (suffix)	Binary
D (suffix)	Decimal
H (suffix)	Hexadecimal
Q (suffix)	Octal

The default mode is decimal; hexadecimal numbers must start with a digit (you must add a leading zero if the number starts with a letter).

Others:

' ' or " "ASCII (characters surrounded by single or double quotation marks)
$ Current value of location (program) counter

General Nomenclature

ADDR	A 16-bit address in data memory
ADDR1	A 16-bit address in data memory
ADDR2	A 16-bit address in data memory
BASE	A constant 16-bit address in data memory
BICON	An 8-bit data item in binary format
CONST	A constant 8-bit data item
DEST	A 16-bit address in program memory, the destination for a jump instruction
HIGH	A 16-bit data item
INDIR	A 16-bit address in data memory, the starting address for an indirect address. The indirect address is stored in memory locations INDIR and INDIR+1.
LOW	A 16-bit data item
MASK	An 8-bit number used for masking
n	A bit position in a byte; possible values are 0 through 7
NPARAM	A 16-bit data item
NEXT	A 16-bit address in program memory
NRESLT	A 16-bit data item
NTIMES	An 8-bit data item
NTIML	An 8-bit data item
NTIMM	An 8-bit data item
NUM	A 16-bit data item
NUM1	A 16-bit address in data memory
NUM2	A 16-bit address in data memory
OFF	An 8-bit fixed offset
OFFSET	An 8-bit fixed offset
oper	An 8-bit data item, a register, (HL), or an indexed address
OPER	A 16-bit address in data memory
OPER1	A 16-bit address in data memory
OPER2	A 16-bit address in data memory
reg	A primary user register (A, B, C, D, E, H, or L)
reg1	A primary user register
RETPT	A 16-bit address in program memory
rp	A primary register pair (BC, DE, or HL)
rph	The more significant byte of rp
rpl	The less significant byte of rp
rp1	A primary register pair
rp1h	The more significant byte of rp1
rp1l	The less significant byte of rp1
rp2	Another primary register pair, not the same as rp1
rp2h	The more significant byte of rp2
rp2l	The less significant byte of rp2
SPTR	A 16-bit address in data memory
STRNG	A 16-bit address in data memory
SUM	A 16-bit address in data memory
TEMP	A 16-bit address in data memory
VAL16	A 16-bit data item
VAL16H	The more significant byte of VAL16
VAL16L	The less significant byte of VAL16
VALUE	An 8-bit data item
xy	An index register, either IX or IY

Chapter 1 General Programming Methods

Some general methods for writing assembly language programs for the Z80 microprocessor are presented in this chapter. In addition, techniques for performing the following operations are explained:

- Loading and saving registers
- Storing data in memory
- Arithmetic and logical functions
- Bit manipulation and testing
- Testing for specific values
- Numerical comparisons
- Looping (repeating sequences of operations)
- Array processing and manipulation
- Table lookup
- Character code manipulation
- Code conversion
- Multiple-precision arithmetic
- Multiplication and division
- List processing
- Processing of data structures.

Also included in this chapter are special sections that describe passing parameters to subroutines, general methods for writing I/O drivers and interrupt service routines, and ways of making programs run faster or use less memory.

The operations described are required in such applications as instrumentation, test equipment, computer peripherals, communications equipment, industrial control, process control, business equipment, aerospace and military systems, and consumer products. Microcomputer users will employ these operations in writing I/O drivers, utility programs, diagnostics, and systems software, and in understanding, debugging, and improving programs written in high-level languages. This chapter provides a brief guide to Z80 assembly language programming for those who have an immediate application in mind.

SUMMARY FOR EXPERIENCED PROGRAMMERS

For those who are familiar with assembly language programming on other computers, we provide here a brief review of the peculiarities of the Z80. Being aware of these unusual features can save a lot of time and trouble.

1. Arithmetic and logical operations are allowed only between the accumulator and a byte of immediate data, the contents of a general-purpose register, the contents of the address in register pair HL, or the contents of an indexed address. Arithmetic and logical instructions do *not* allow direct addressing.

For example, the alternatives for the OR instruction are OR CONST, where CONST is a fixed data byte; OR reg, where reg is an 8-bit general-purpose register; OR (HL); and OR (xy+OFF). The third alternative logically ORs the accumulator with the data byte located at the address in HL. The fourth alternative logically ORs the accumulator with the data byte located at an indexed address; the processor determines the address by adding the 8-bit offset OFF to a 16-bit index register.

2. The accumulator and register pair HL are special. The accumulator is the only byte-length register that can be loaded or stored directly. The accumulator is also the only register that can be complemented, negated, shifted with a single-byte instruction, loaded indirectly from the addresses in register pairs BC or DE, stored indirectly at the addresses in register pairs BC or DE, or used in IN and OUT instructions with direct addressing.

HL is the only register pair that can serve as an indirect address in arithmetic or logical instructions or in loading or storing registers other than the accumulator. HL is also the only register pair that can be transferred to the program counter or stack pointer. Furthermore, HL serves as a double-length accumulator in 16-bit addition and subtraction. Register pair DE is also special because the instruction EX DE,HL can exchange it with HL. Thus, the Z80's registers are highly asymmetric, and the programmer must carefully choose which data and addresses go in which registers.

3. There are often several names for the same physical register. The registers A, B, C, D, E, H, and L are all available as 8-bit registers. The register pairs BC (B more significant), DE (D more significant), and HL (H more significant) are also available as 16-bit register pairs in many instructions. The terms "register pair B," "registers B and C," and "register pair BC" all have the same meaning, and there are similar variations for registers D and E and H and L. Note that the register pair and the two single registers are physically identical and cannot be used for different purposes at the same time.

In fact, H and L are almost always used to hold an indirect address because of the availability of instructions that access the data at that address as well as special instructions like LD SP,HL; JP (HL); EX (SP),HL; and EX DE,HL. Register pair DE is used for a second address when one is needed because of the EX DE,HL instruction. Registers B and C are generally used as separate 8-bit registers for temporary data storage and counters.

4. The effects of instructions on flags are extremely inconsistent. Some particularly unusual effects are (a) logical instructions clear the Carry, (b) one-byte accumulator rotate instructions affect no flags other than the Carry, (c) load, store, transfer, increment register pair or index register, and decrement register pair or index register instructions affect no flags at all, and (d) 16-bit addition (ADD HL or ADD xy) affects only the Carry flag. Table A-1 in Appendix A can be used as an aid in determining how an instruction affects the flags.

5. There is no indirect addressing through memory locations. The lack of indirect addressing is overcome by loading the indirect address into register pair HL. Thus, indirect addressing is a two-step process. The indirect address can also be loaded into registers pair BC or DE, but it can then only be used to load or store the accumulator.

6. The Z80's indexing allows only an 8-bit fixed offset in the instruction. Its main purpose is to implement postindexing and to allow offsets in data structures. A more general form of indexed addressing requires an explicit 16-bit addition of register pairs using HL as a 16-bit accumulator. Thus, indexing usually requires several steps: The index must be loaded into one register pair, the base address must be loaded into another register pair (one pair must be HL), the two must be added explicitly (using ADD HL,rp), and the sum must be used as an indirect address (by referring to (HL)). Generalized indexing on the Z80 is a long, awkward process.

7. There is a combined Parity/Overflow indicator. This flag indicates even parity after all instructions that affect it except addition and subtraction. Then it indicates the occurrence of two's complement overflow.

8. Many common instructions are missing but can easily be simulated with register operations. Some examples are clearing the accumulator (use SUB A or XOR A), clearing the Carry flag (use AND A or OR A), and logically shifting the accumulator left (use ADD A,A). Either AND A or OR A clears the Carry flag and sets the other flags according to the contents of the accumulator. But remember, loading a register does not affect any flags.

9. There are both relative and absolute branches (using the operation codes JR and JP, respectively). Both addressing methods are allowed for unconditional branches. The sets of conditional branches differ; relative branches exist only for the Carry and Zero flags, whereas absolute branches exist for the Carry, Sign, Parity/Overflow, and Zero flags. What is interesting here is that the relative branches occupy less memory

than the corresponding absolute branches (2 bytes rather than 3) but execute more slowly if the branch is taken (12 cycles rather than 10).

10. Increment and decrement instructions behave differently, depending on whether they are applied to 8-bit or 16-bit operands. Decrementing or incrementing an 8-bit register affects all flags except the Carry. Decrementing or incrementing a 16-bit register pair or index register does not affect any flags at all. A 16-bit register pair can be used as a counter, but the only way to test the pair for zero is to logically OR the two bytes together in the accumulator. The 16-bit instructions are intended primarily for address calculations, not for data manipulation.

11. Instructions that are additions to the original 8080 instruction set occupy more memory and execute more slowly than other instructions with similar functions and addressing modes. Among them are bit manipulation, arithmetic shift, logical shift, shifts of registers other than the accumulator, and some loads. These instructions execute more slowly because they require a prefix byte that tells the processor the instruction is not an original 8080 instruction and the next byte is the real operation code. Weller makes it easier to recognize the secondary instructions by using mnemonics derived from the 8080 instruction set.[1]

12. Certain registers and facilities are clearly secondary in importance. The programmer should employ them only when the primary registers and facilities are already in use or too inconvenient to use. The secondary facilities, like the secondary instructions, represent additions to the underlying 8080 microprocessor. The most important additions are index registers IX and IY; many instructions use these registers, but they take more memory and much more time than instructions that use the other register pairs. Another addition is the primed register set. Only two instructions (EX 'AF,AF' and EXX) allow access to the primed set, and for this reason programmers generally reserve it for functions such as fast interrupt response.

13. Operations that can be done directly to a general-purpose register are shift it, transfer it to or from another register, load it with a constant, increment it by 1, or decrement it by 1. These operations can also be performed indirectly on the memory address in HL or on a memory location addressed via indexing.

14. Only register pairs or index registers can be moved to or from the stack. One pair is AF, which consists of the accumulator (more significant byte) and the flags (less significant byte). The CALL and RET instructions transfer addresses to or from the stack; there are conditional calls and returns but they are seldom used.

15. The Z80 has a readable interrupt enable flag. One can determine its value by executing LD A,I or LD A,R. Either instruction moves the Interrupt flip-flop to the Parity/Overflow flag. That flag then reflects the state of the interrupt system at a particular time, and thus can be used to restore the state after the processor executes code that must run with interrupts disabled.

16. The Z80 uses the following common conventions:

· The 16-bit addresses are stored with the less significant byte first (that is, at the lower address). The order of the bytes in an address is the same as in the 8080, 8085, and 6502 microprocessors, but the opposite of that used in the 6800 and 6809.

· The stack pointer contains the lowest address actually occupied by the stack. This convention is also used in the 8080, 8085, and 6809 microprocessors, but the obvious alternative (next available address) is used in the 6502 and 6800. Z80 instructions store data in the stack using predecrementing (they subtract 1 from the stack pointer before storing a byte) and load data from the stack using postincrementing (they add 1 to the stack pointer after loading a byte).

· The interrupt (enable) flag is 1 to allow interrupts and 0 to disallow them. This convention is the same as in the 8080 and 8085, but the opposite of that used in the 6502, 6800, and 6809.

REGISTER SET

Z80 assembly language programming is complicated by the asymmetry of the processor's instruction set. Many instructions apply only to particular registers, register pairs, or sets of registers. Almost every register has its own unique features, and almost every instruction has its own peculiarities. Table 1-1 lists the 8-bit registers and the instructions that use them. Table 1-2 lists the 16-bit registers and the instructions that use them (of course, all instructions change the program counter implicitly). Table 1-3 lists the indirect addresses contained in on-board register pairs and the instructions that use them. Table 1-4 lists the instructions that apply only to the accumulator, and Table 1-5 lists the instructions that apply only to particular 16-bit registers. Table 1-6 lists the instructions that apply to the stack.

The general uses of the registers are as follows:

· The accumulator, the center of data processing, is the source of one operand and destination of the result for most arithmetic, logical, and other processing operations.

· Register pair HL is the primary memory address register. Instructions can often refer to the data at the address in HL, that is, (HL).

· Register pair DE is the secondary memory address register because the programmer can exchange its contents with HL using EX DE,HL.

· Registers B and C (register pair BC) are general-purpose registers used mainly for counters and temporary data storage. Register B is often used as a loop counter because of its special usage in the DJNZ instruction.

· Index registers IX and IY are used when the programmer is referring to memory addresses by means of fixed offsets from a variable base. These registers also serve as backups to HL when that register pair is occupied.

Table 1-1. Eight-Bit Registers and Applicable Instructions

8-Bit Register	Instructions
A only	CPL, DAA; IN A,(port); LD (ADDR),LD (BC or DE), NEG; OUT (port),A; RLA, RLCA, RLD, RRA, RRCA, RRD.
A,B,C,D,E,H,L	ADC A; ADD A; AND, CP, DEC; IN reg,(C); INC, LD, OR; OUT (C),reg; RL, RLC, RR, RRC, SBC A; SLA, SRA, SRL, SUB, XOR
B only	DJNZ, IND, INDR, INI, INIR, OTDR, OTIR, OUTD, OUTI
C only	IN reg,(C); OUT (C),reg; IND, INDR, INI, INIR, OTDR, OTIR, OUTD, OUTI
F (flags)	CCF, SCF (see also AF register pair)
I (interrupt vector)	LD I,A; LD A,I
R (refresh)	LD R,A; LD A,R

Table 1-2. Sixteen-Bit Registers and Applicable Instructions

16-Bit Register	Instructions
AF	POP; PUSH; EX AF,AF′
AF′	EX AF,AF′
BC	ADC HL, ADD xy, ADD HL, CPD, CPDR, CPI, CPIR, DEC, EXX, INC, LD, LDD, LDDR, LDI, LDIR, POP, PUSH, SBC HL
BC′	EXX
DE	ADC HL, ADD xy, ADD HL, DEC; EX DE,HL; EXX, INC, LD, LDD, LDDR, LDI, LDIR, POP, PUSH, SBC HL
DE′	EXX
HL	ADC HL, ADD HL, CPD, CPDR, CPI, CPIR, DEC; EX DE,HL; EX (SP),HL; EXX, INC, IND, INDR, INI, INIR, LD, LDD, LDDR, LDI, LDIR, OTDR, OTIR, OUTD, OUTI, POP, PUSH, SBC HL
HL′	EXX
IX	ADD IX, LD, POP, PUSH; EX (SP),IX
IY	ADD IY, LD, POP, PUSH; EX (SP),IY
Program Counter	CALL instructions, JP, JR, RETURN instructions, RETI, RETN, RST
Stack Pointer	CALL instructions, ADD HL, DEC, INC, LD, POP, PUSH, RETURN instructions, RST

Table 1-3. Indirect Addresses and Applicable Instructions

Location of Address	Instructions
Register pair BC	LD A,(BC); LD (BC),A
Register pair DE	LD A,(DE); LD (DE),A
Register pair HL*	ADC A; ADD A; AND, CP, DEC, INC, JP, LD, OR, SBC A; SUB, XOR
Stack Pointer	CALL instructions, POP, PUSH, RETURN instructions, RST
Index register X or Y	JP

* Index register X or Y can also be used as an indirect address for the same instructions as HL by specifying indexed addressing with a fixed offset of zero.

Table 1-4. Instructions That Apply Only to the Accumulator

Instruction	Function
ADC A	Add with carry
ADD A	Add
AND	Logical AND immediate
CPL	One's complement
CP	Compare
DAA	Decimal adjust (decimal correction)
IN A,(port)	Input direct
LD A,(ADDR)	Load direct
LD A,(rp)	Load indirect
NEG	Two's complement (negate)
OR	Logical OR
OUT (port),A	Output direct
RLA	Rotate accumulator left through carry
RLCA	Rotate accumulator left
RRA	Rotate accumulator right through carry
RRCA	Rotate accumulator right
SBC A	Subtract with borrow
SUB	Subtract
XOR	Logical EXCLUSIVE OR

Table 1-5. Instructions That Apply Only to One or Two 16-Bit Registers

Instruction	16-Bit Registers	Function
EX AF,AF′	AF,AF′	Exchange program status with alternate program status
EX DE,HL	DE,HL	Exchange HL with DE
EX (SP),HL	HL	Exchange HL with top of stack
EX (SP),xy	IX or IY	Exchange index register with top of stack
LD SP,HL	HL,SP	Load stack pointer from HL
LD SP,xy	IX or IY,SP	Load stack pointer from index register

Table 1-6. Instructions That Use the Stack

Instruction	Function
Call instructions	Jump and save program counter in stack (including conditionals)
EX (SP),HL	Exchange HL with top of stack
EX (SP),xy	Exchange index register with top of stack
POP	Load register pair from stack
PUSH	Store register pair in stack
RETURN instructions	Load program counter from stack (including conditionals)
RST	Jump to vector address and save program counter in stack

We may describe the special features of particular registers as follows:

· **Accumulator.** Only single register that can be loaded or stored directly. Only 8-bit register that can be shifted with a one-byte instruction. Only register that can be complemented, decimal adjusted, or negated with a single instruction. Only register that can be loaded or stored using the addresses in register pairs BC or DE. Only register that can be stored in an output port or loaded from an input port using direct addressing. Source and destination for all 8-bit arithmetic and logical instructions except DEC and INC. Only register that can be transferred to or from the interrupt vector (I) or refresh (R) register.

· **Register pair HL.** Only register pair that can be used indirectly in the instructions ADC, ADD, AND, CMP, DEC, INC, OR, SBC, SUB, and XOR. Source and destination for the instructions ADC HL, ADD HL, and SBC HL. Only register pair that can be exchanged with register pair DE or with the top of the stack. Only register pair that can have its contents moved to the stack pointer (LD SP,HL) or the program counter (JP (HL)). Only register pair that can be shifted with a single instruction (ADD HL,HL). Automatically used as a source address register in block move, block compare, and block output instructions. Automatically used as a destination address register in block input instructions.

- **Register pair DE.** Only register pair that can exchanged with HL (EX DE,HL). Automatically used as a destination address register in block move instructions.

- **Register pair BC.** Automatically used as a counter in block move and block compare instructions.

- **Register B.** Automatically used as a counter in the DJNZ instruction and in block input and output instructions.

- **Register C.** Only register that can be used as an indirect port address for input and output. Automatically used as a port address in block input and output instructions.

- **Index registers IX and IY.** Only address registers that allow an indexed offset. Used as source and destination in ADD xy instruction. Can be exchanged with the top of the stack, moved to the stack pointer or program counter, or shifted with ADD xy,xy.

- **Stack pointer.** Automatically postincremented by instructions that load data from the stack and predecremented by instructions that store data in the stack. Only address register that can be used to transfer other register pairs to or from memory (PUSH and POP) or to transfer the program counter to or from memory (CALL instructions and RETURN instructions).

Note the following:

- The A register is the only 8-bit register that can be loaded from memory or stored in memory using direct addressing.

- Only the address in register pair HL or an address obtained via indexing can be used in operations other than loading and storing the accumulator. That is, only the data at the address in HL or at an indexed address can be moved to or from a user register, decremented, incremented, or used in arithmetic and logical operations.

- Only DEC reg and INC reg perform 8-bit arithmetic operations without involving the accumulator (of course, DEC and INC may be applied to the accumulator).

- Only index registers IX and IY allow an offset from a base address. The data at the indexed address can be used like the data at the address in HL.

- The index registers IX and IY make useful backups to HL because of the availability of the 16-bit instructions ADD xy; EX (SP),xy; JP (xy); and LD SP,xy.

Register Transfers

The LD instruction can transfer any 8-bit general-purpose register (A, B, C, D, E, H, or L) to any other 8-bit general-purpose register. The flag (F) register can only be transferred to or from the stack along with the accumulator (PUSH AF and POP AF). Register pairs DE and HL can be exchanged using EX DE,HL.

The common transfer instructions are

- LD A,reg transfers the contents of reg to the accumulator
- LD reg,A transfers the contents of the accumulator to reg
- LD reg,(HL) loads reg with the contents of the memory address in register pair HL
- LD (HL),reg stores reg at the memory address in register pair HL
- EX DE,HL exchanges register pair DE with HL.

The destination always comes first in the operand field of LD. That is, LD reg1,reg2 transfers the contents of reg2 to reg1, the opposite of the convention proposed in IEEE Standard 694 for assembly language instructions.[2] The LD changes the destination, but leaves the source as it was. Note that EX DE,HL changes all four registers (D, E, H, and L); it is thus equivalent to four LDs plus some intermediate steps that save one byte of data while transferring another.

LOADING REGISTERS FROM MEMORY

The Z80 microprocessor has five addressing modes that can be used to load registers from memory. These addressing modes are: Direct (from a specific memory address), Immediate (with a specific value), Indirect (from an address stored in a register pair), Indexed (from an address obtained by adding a fixed offset to an index register), and Stack (from the top of the stack).[3]

Direct Loading of Registers

The accumulator, a primary register pair (BC, DE, or HL), the stack pointer, or an index register can be loaded from memory using direct addressing.

Examples

1. LD A,(2050H)

This instruction loads the accumulator (register A) from memory location 2050_{16}.

2. LD HL,(0A000H)

This instruction loads register L from memory location $A000_{16}$ and register H from memory location. $A001_{16}$ Note the standard Z80 practice of storing 16-bit numbers with the less significant byte at the lower address, followed by the more significant byte.

3. LD SP,(9A12H)

This instruction loads the stack pointer from memory locations $9A12_{16}$ (less significant byte) and $9A13_{16}$ (more significant byte).

Immediate Loading of Registers

Immediate addressing can be used to load any register, register pair, or index register with a specific value. The register pairs include the stack pointer.

Examples

1. LD C,6

This instruction loads register C with the number 6. The 6 is an 8-bit data item, not a 16-bit address. Do not confuse the number 6 with the address 0006_{16}.

2. LD DE,15E3H

This instruction loads register D with 15_{16} and register E with $E3_{16}$.

3. LD IY,0B7EEH

This instruction loads index register IY with $B7EE_{16}$.

Indirect Loading of Registers

The instruction LD reg,(HL) can load any register from the address in register pair HL. The instruction LD A,(rp) can load the accumulator using the address in a register pair (BC, DE, or HL). Note that there is no instruction that loads a register pair indirectly.

Examples

1. LD D,(HL)

This instruction loads register D from the memory address in register pair HL. The assembly language instruction takes the form "LD destination register, source register"; the order of the operands is the opposite of that proposed for IEEE Standard 694.[4]

2. LD A,(BC)

This instruction loads the accumulator from the memory address in register pair BC. Note that you cannot load any register except A using BC or DE indirectly.

Indexed Loading of Registers

The instruction LD A,(xy+OFFSET) loads the accumulator from the indexed address obtained by adding the 8-bit number OFFSET to the contents of an index register. Note that OFFSET is a fixed 8-bit number (its value is part of the program), while the index register contains a 16-bit address that can be changed.[5] If OFFSET = 0, indexing is equivalent to indirection, but it is slower since the processor still must perform the address addition.

Stack Loading of Registers

The instruction POP rp or POP xy loads a register pair or an index register from the top of the stack and adjusts the stack pointer appropriately. One register pair for POP rp is AF, which consists of the accumulator (more significant byte) and the flags (less significant byte). No instructions load 8-bit registers from the stack or use the stack pointer indirectly without changing it (although EX (SP),HL and EX (SP),xy have no net effect on the stack pointer since they transfer data both to and from the stack).

Examples

1. POP DE

This instruction loads register pair DE from the top of the stack and increments the stack pointer by 2. Register E is loaded first.

2. POP IY

This instruction loads index register IY from the top of the stack and increments the stack pointer by 2. The less significant byte of IY is loaded first.

The stack has the following special features:

- The stack pointer contains the address of the most recently occupied location. The stack can be anywhere in memory.
- Data is stored in the stack using predecrementing—the instructions decrement the stack pointer by 1 *before* storing each byte. Data is loaded from the stack using postincrementing—the instructions increment the stack pointer by 1 *after* loading each byte.
- As is typical with microprocessors, there are no overflow or underflow indicators.

STORING REGISTERS IN MEMORY

The Z80 has four addressing modes that can be used to store registers in memory. These modes are: Direct (at a specific memory address), Indirect (at an address stored in a register pair), Indexed (at an address calculated by adding an 8-bit offset to the contents of an index register), and Stack (at the top of the stack).

Direct Storage of Registers

Direct addressing can be used to store the accumulator, a register pair (BC, DE, or HL), the stack pointer, or an index register.

Examples

1. **LD (35C8H),A**

This instruction stores the accumulator in memory location $35C8_{16}$.

2. **LD (203AH),HL**

This instruction stores register L in memory location $203A_{16}$ and register H in memory location $203B_{16}$.

3. **LD (0A57BH),SP**

This instruction stores the stack pointer in memory locations $A57B_{16}$ (less significant byte) and $A57C_{16}$ (more significant byte).

Indirect Storage of Registers

The instruction LD (HL),reg can store any register at the address in register pair HL. The instruction LD (rp),A can store the accumulator at the address in a register pair (BC, DE, or HL). Note that there is no instruction that stores a register pair indirectly.

Examples

1. **LD (HL),C**

This instruction stores register C at the address in register pair HL. The form is "move to address in HL from C."

2. **LD (DE),A**

This instruction stores the accumulator at the memory address in register pair DE. Note that you cannot store any register except A using BC or DE indirectly.

Indexed Storage of Registers

The instruction LD (xy+OFFSET),A stores the accumulator at the indexed address obtained by adding the 8-bit number OFFSET to the contents of an index register. If OFFSET = 0, the indexed address is simply the contents of the index register, and indexing is reduced to a slow version of indirect addressing.

Stack Storage of Registers

The instruction PUSH rp or PUSH xy stores a register pair or an index register at the top of the stack and adjusts the stack pointer appropriately. One register pair is AF, consisting of the accumulator (more significant byte) and the flags (less significant byte). There is no instruction that stores an 8-bit register in the stack.

Examples

1. PUSH BC

This instruction stores register pair BC at the top of the stack and decrements the stack pointer by 2. Note that B is stored first, so C ends up at the top of the stack.

2. PUSH IX

This instruction stores index register IX at the top of the stack and decrements the stack pointer by 2. Note that the less significant byte of IX is stored last, and thus it ends up at the top of the stack.

OTHER LOADING AND STORING OPERATIONS

Other loading and storing operations require more than one instruction. Some typical examples are

1. Direct loading of a register other than A.

```
        LD    A,(ADDR)
        LD    reg,A
```

An alternative is

```
        LD    HL,ADDR
        LD    reg,(HL)
```

The second approach leaves A unchanged, but makes HL an indirect addressing pair. Of course, the address in HL would then be available for later use.

2. Indirect loading of a register (from the address in memory locations INDIR and INDIR+1).

```
LD    HL,(INDIR)       ;GET INDIRECT ADDRESS
LD    reg,(HL)         ;LOAD DATA INDIRECTLY
```

3. Direct storage of a register other than A.

```
LD    A,reg
LD    (ADDR),A
```

An alternative is

```
LD    HL,ADDR
LD    (HL),reg
```

4. Indirect storage of a register (at the address in memory locations INDIR and INDIR+1).

```
LD    HL,(INDIR)       ;GET THE INDIRECT ADDRESS
LD    (HL),reg         ;STORE DATA THERE
```

STORING VALUES IN RAM

The usual ways to initialize RAM locations are (1) through the accumulator, (2) using register pair HL directly or indirectly, and (3) using an index register with a fixed offset.

Examples

1. Store an 8-bit item (VALUE) in address ADDR.

```
LD    A,VALUE
LD    (ADDR),A
```

or

```
LD    HL,ADDR
LD    (HL),VALUE
```

If VALUE = 0, we could use SUB A or XOR A instead of LD A, 0. Note, however, that SUB A or XOR A affects the flags, whereas LD A,0 does not.

2. Store a 16-bit item (VAL16) in addresses ADDR and ADDR+1 (MSB in ADDR+1).

```
LD    HL,VAL16
LD    (ADDR),HL
```

3. Store an 8-bit item (VALUE) at the address in memory locations INDIR and INDIR+1.

```
        LD      HL,(INDIR)      ;GET INDIRECT ADDRESS
        LD      (HL),VALUE      ;STORE DATA INDIRECTLY
```

4. Store an 8-bit item (VALUE) nine bytes beyond the address in memory locations INDIR and INDIR+1.

```
        LD      A,VALUE
        LD      xy,(INDIR)      ;GET BASE ADDRESS
        LD      (xy+9),A        ;STORE DATA 9 BYTES BEYOND BASE
```

Here the indirect address is the base address of an array or other data structure.

ARITHMETIC AND LOGICAL OPERATIONS

Most arithmetic and logical operations (addition, subtraction, AND, OR, EXCLUSIVE OR, and comparison) can be performed only between the accumulator and an 8-bit register, a byte of immediate data, or a byte of data in memory addressed through register pair HL or via indexing. Note that arithmetic and logical instructions do *not* allow direct addressing. If a result is produced (comparison does not produce any), it replaces the operand in the accumulator.

Examples

1. Logically OR the accumulator with register C.

```
        OR      C
```

OR C logically ORs register C with the accumulator and places the result in the accumulator. The programmer only has to specify one operand; the other operand and the destination of the result are always the accumulator.

2. Add register B to the accumulator.

```
        ADD     A,B
```

ADD A,B adds register B to the accumulator (register A) and places the result in the accumulator. In the instructions ADC, ADD, and SBC, the programmer must specify both operands. The reason is that the Z80 also has the instructions ADC HL (add register pair to HL with carry), ADD HL (add register pair to HL), ADD xy (add register pair or index register to index register), and SBC HL (subtract register pair from HL with borrow). Note the inconsistency here: Both operands must be specified in ADC, ADD, and SBC, but only one operand in SUB; furthermore, the Z80 has an ADD xy instruction, but no ADC xy or SBC xy instruction. Since the 16-bit arithmetic instructions are mainly intended for addressing, we will discuss them later.

3. Logically AND the accumulator with the binary constant BICON.

```
        AND   BICON
```

Immediate addressing is the default mode; no special operation code or designation is necessary.

4. Logically OR the accumulator with the data at the address in register pair HL.

```
        OR    (HL)
```

Parentheses indicate a reference to the contents of a memory address.

Other operations require more than one instruction. Some typical examples are:

· Add memory locations OPER1 and OPER2, place sum in memory location SUM.

```
        LD    A,(OPER1)       ;GET FIRST OPERAND
        LD    B,A
        LD    A,(OPER2)       ;GET SECOND OPERAND
        ADD   A,B
        LD    (SUM),A         ;SAVE SUM
```

or

```
        LD    HL,OPER1
        LD    A,(HL)          ;GET FIRST OPERAND
        LD    HL,OPER2
        ADD   A,(HL)          ;ADD SECOND OPERAND
        LD    HL,SUM
        LD    (HL),A          ;SAVE SUM
```

We can shorten the second alternative considerably if the operands and the sum occupy consecutive memory addresses. For example, if OPER2 = OPER1 + 1 and SUM = OPER2 + 1, we have

```
        LD    HL,OPER1
        LD    A,(HL)          ;GET FIRST OPERAND
        INC   HL
        ADD   A,(HL)          ;ADD SECOND OPERAND
        INC   HL
        LD    (HL),A          ;SAVE SUM
```

· Add a constant (VALUE) to memory location OPER.

```
        LD    A,(OPER)
        ADD   A,VALUE
        LD    (OPER),A
```

or

```
        LD    HL,OPER
        LD    A,(HL)
        ADD   A,VALUE
        LD    (HL),A
```

If VALUE = 1, we can shorten the second alternative to

```
LD   HL,OPER
INC  (HL)
```

You can use DEC (HL) similarly without changing the accumulator, but both DEC (HL) and INC (HL) affect all the flags except Carry.

BIT MANIPULATION

The Z80 has specific instructions for setting, clearing, or testing a single bit in a register or memory location. Other bit operations require a series of single-bit instructions or logical instructions with appropriate masks. Complementing (CPL) applies only to the accumulator. Chapter 7 contains additional examples of bit manipulation.

The specific bit manipulation instructions are

```
SET n,reg
RES n,reg
BIT n,reg
```

- Sets bit n of register reg
- Clears bit n of register reg
- Tests bit n of register reg, setting the Zero flag if that bit is 0 and clearing the Zero flag if it is 1.

All three instructions can also be applied to (HL) or to an indexed address. Note that the bit position is not a variable; it is part of the instruction.[6]

Other bit operations can be implemented by applying logical instructions to the accumulator as follows:

- Set bits to 1 by logically ORing them with 1's in the appropriate positions.
- Clear bits by logically ANDing them with 0's in the appropriate positions.
- Invert (complement) bits by logically EXCLUSIVE ORing them with 1's in the appropriate positions.
- Test bits (for all 0's) by logically ANDing them with 1's in the appropriate positions.

This approach is inconvenient since the logical instructions can only be applied to the accumulator. It does, however, allow the programmer to invert bits and change several bits at the same time.

Examples

1. Set bit 6 of the accumulator.

```
SET  6,A
```

or

```
        OR    01000000B         ;SET BIT 6 BY ORING WITH 1
```

Logically ORing a bit with 0 leaves it unchanged.

2. Clear bit 3 of the accumulator.

```
        RES   3,A
```

or

```
        AND   11110111B         ;CLEAR BIT 3 BY ANDING WITH 0
```

Logically ANDing a bit with 1 leaves it unchanged.

3. Invert (complement) bit 2 of the accumulator.

```
        XOR   00000100B         ;INVERT BIT 2 BY XORING WITH 1
```

Logically EXCLUSIVE ORing a bit with 0 leaves it unchanged. Here there is no special bit manipulation instruction. Fortunately, setting and clearing bits are much more common operations than complementing bits.

4. Test bit 5 of the accumulator. In other words, clear the Zero flag if bit 5 is 1, and set it if bit 5 is 0.

```
        BIT   5,A
```

or

```
        AND   00100000B         ;TEST BIT 5 BY ANDING WITH 1
```

Note the inversion here in either alternative: The Zero flag is set to 1 if the bit is 0, and to 0 if the bit is 1.

5. Set bit 4 of register D.

```
        SET   4,D
```

To use a logical function, we would have to load the data into the accumulator and load the result back into register D.

6. Invert (complement) bit 7 of memory location ADDR.

```
        LD    A,(ADDR)          ;GET DATA
        XOR   10000000B         ;COMPLEMENT BIT 7
        LD    (ADDR),A          ;RETURN RESULT TO MEMORY
```

7. Set bit 0 of the memory location five bytes beyond the address in INDIR and INDIR+1.

```
        LD    xy,(ADDR)         ;GET INDIRECT ADDRESS
        SET   0,(xy+5)          ;SET BIT 0 OF BYTE 5
```

You can change more than one bit at a time by using a series of bit manipulation instructions or by using the logical functions with appropriate masks.

8. Set bits 4 and 5 of the accumulator.

```
        OR    00110000B        ;SET BITS 4 AND 5 BY ORING WITH 1
```

or

```
        SET   4,A              ;SET BIT 4 FIRST
        SET   5,A              ;AND THEN SET BIT 5
```

9. Invert (complement) bits 0 and 7 of the accumulator.

```
        XOR   10000001B        ;INVERT BITS 0 AND 7 BY XORING WITH 1
```

A handy shortcut to change bit 0 of a register or memory location is to use INC to set it (if you know that it is 0) and DEC to clear it (if you know that it is 1). You can also use either INC or DEC to complement bit 0 if you are not using the other bits of a register or memory location. These shortcuts are useful when you are storing a single 1-bit flag in a register or memory location.

SHIFT OPERATIONS

The Z80 has shift instructions that operate on any register or memory location. Special instructions apply only to the accumulator, register pair HL, or an index register. Chapter 7 contains further examples of shift operations.

The instructions RL and RR rotate a register or memory location and the Carry flag as if they formed a 9-bit register. Figures 1-1 and 1-2 show the effects of RL and RR. The instructions RLC and RRC rotate the register or memory location alone as shown in Figures 1-3 and 1-4. The bit shifted off the end still appears in the Carry flag as well as in the bit position at the other end. The instructions SLA and SRL perform logical shifts (as shown in Figures 1-5 and 1-6) which fill the bit at the far right or left with a 0. SRA performs an arithmetic shift (see Figure 1-7) which preserves the sign bit by extending (copying) it to the right. Note that RL and RR preserve the old Carry flag (in either bit 0 or bit 7), whereas the other shift instructions destroy it.

Certain special instructions are shorter and faster than the regular shifts in specific situations. One-byte circular shifts (RLA, RLCA, RRA, RRCA) apply only to the accumulator. Adding a register to itself (ADD A,A; ADD HL,HL; ADD xy,xy) is equivalent to a logical left shift, while adding a register to itself with Carry (ADC A,A or ADC HL,HL) is equivalent to a left rotate through Carry.

Examples

1. Rotate accumulator right two positions without the Carry.

```
        RRCA
        RRCA
```

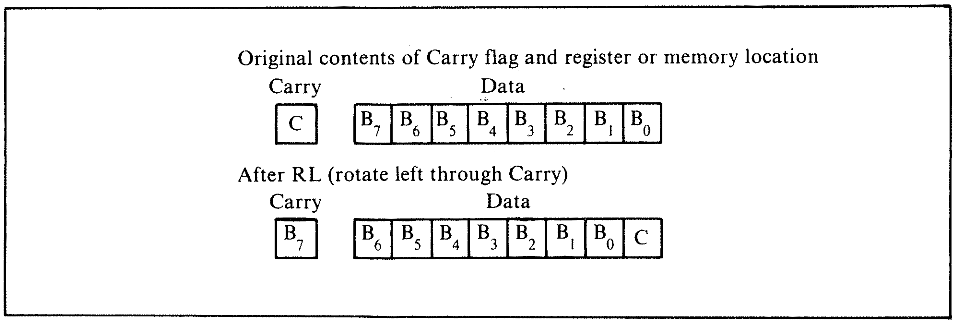

Original contents of Carry flag and register or memory location
Carry
Data
C
B_7 B_6 B_5 B_4 B_3 B_2 B_1 B_0
After RL (rotate left through Carry)
Carry
Data
B_7
B_6 B_5 B_4 B_3 B_2 B_1 B_0 C

Figure 1-1. The RL (rotate left through Carry) instruction

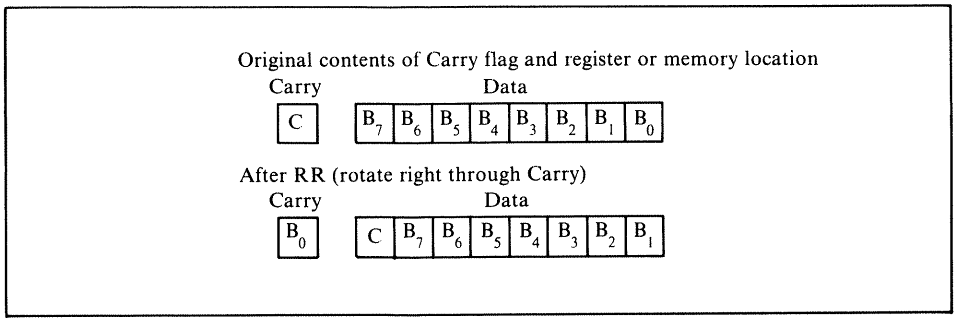

Original contents of Carry flag and register or memory location
Carry
Data
C
B_7 B_6 B_5 B_4 B_3 B_2 B_1 B_0
After RR (rotate right through Carry)
Carry
Data
B_0
C B_7 B_6 B_5 B_4 B_3 B_2 B_1

Figure 1-2. The RR (rotate right through Carry) instruction

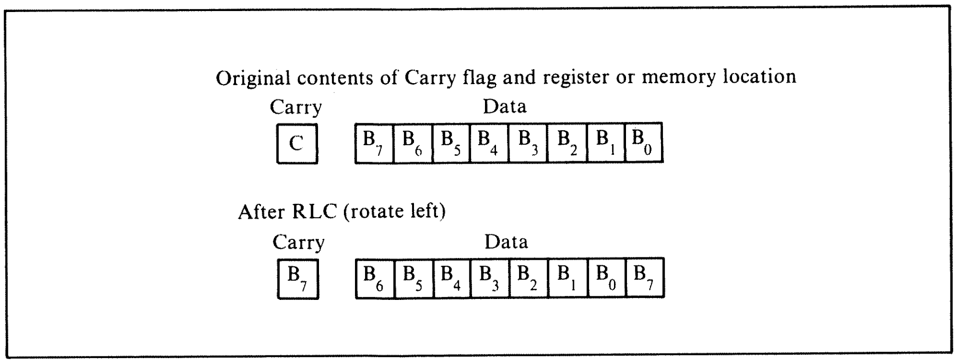

Original contents of Carry flag and register or memory location
Carry
Data
C
B_7 B_6 B_5 B_4 B_3 B_2 B_1 B_0
After RLC (rotate left)
Carry
Data
B_7
B_6 B_5 B_4 B_3 B_2 B_1 B_0 B_7

Figure 1-3. The RLC (rotate left) instruction

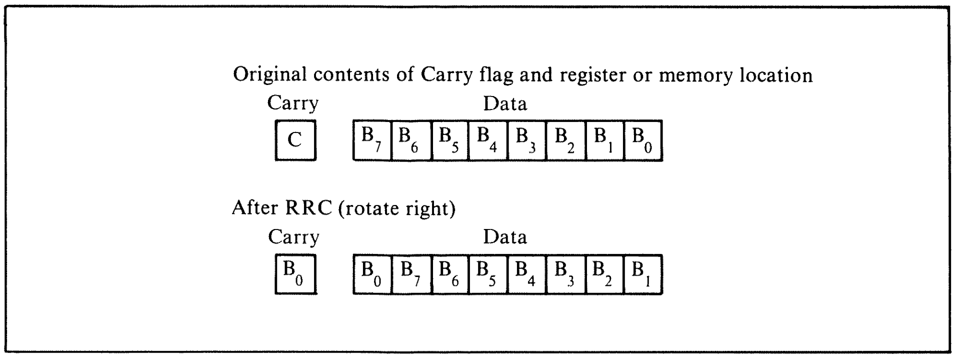

Original contents of Carry flag and register or memory location
Carry
Data
C
B_7 B_6 B_5 B_4 B_3 B_2 B_1 B_0
After RRC (rotate right)
Carry
Data
B_0
B_0 B_7 B_6 B_5 B_4 B_3 B_2 B_1

Figure 1-4. The RRC (rotate right) instruction

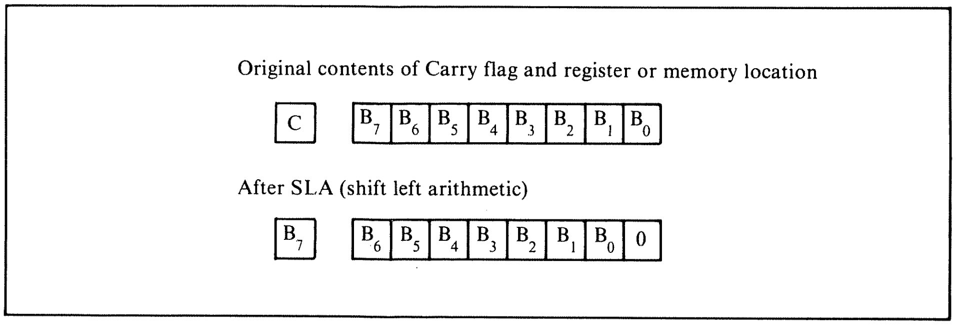

Original contents of Carry flag and register or memory location
C
B_7 B_6 B_5 B_4 B_3 B_2 B_1 B_0
After SLA (shift left arithmetic)
B_7
B_6 B_5 B_4 B_3 B_2 B_1 B_0 0

Figure 1-5. The SLA (shift left arithmetic) instruction

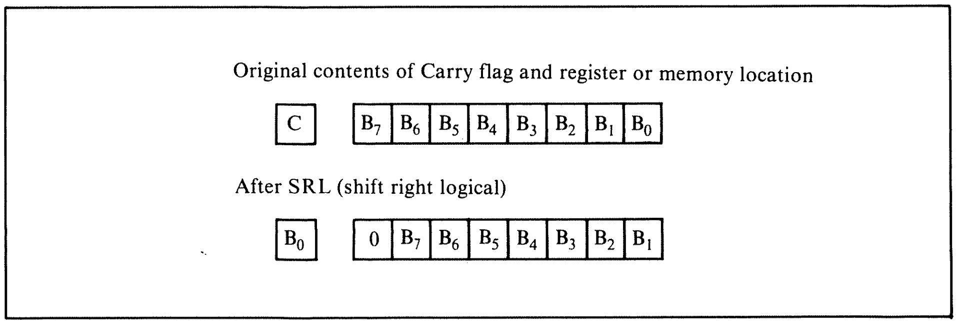

Original contents of Carry flag and register or memory location
C
B_7 B_6 B_5 B_4 B_3 B_2 B_1 B_0
After SRL (shift right logical)
B_0
0 B_7 B_6 B_5 B_4 B_3 B_2 B_1

Figure 1-6. The SRL (shift right logical) instruction

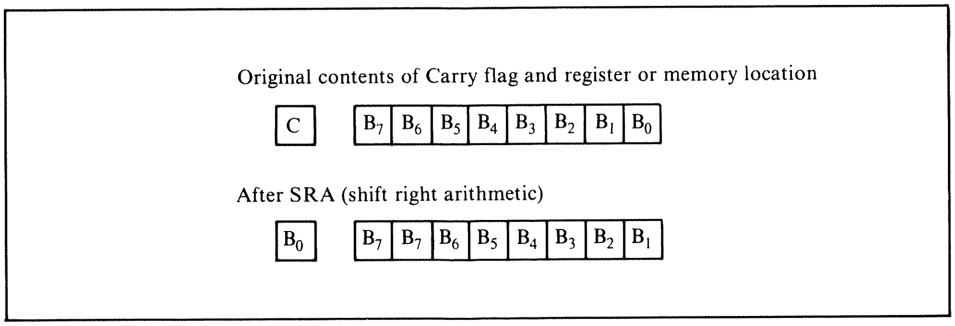

Original contents of Carry flag and register or memory location
C
B_7 B_6 B_5 B_4 B_3 B_2 B_1 B_0
After SRA (shift right arithmetic)
B_0
B_7 B_7 B_6 B_5 B_4 B_3 B_2 B_1

Figure 1-7. The SRA (shift right arithmetic) instruction

Note the special form for the accumulator.

2. Shift accumulator left logically two positions.

```
SLA   A
SLA   A
```

A shorter, faster alternative is

```
ADD   A,A
ADD   A,A
```

The instruction ADD A,A is equivalent to a logical left shift of A. Note that ADD A,A is a one-byte instruction, whereas SLA is always at least a two-byte instruction since it is an addition to the 8080 instruction set.

3. Shift register C right logically one position.

```
SRL   C
```

4. Shift register pair HL left logically two positions.

```
ADD   HL,HL
ADD   HL,HL
```

ADD HL, HL is a one-byte logical left shift of HL.

Shift instructions can also be applied to memory locations addressed either through register pair HL or through indexing from IX or IY.

5. Shift memory location ADDR right one position, preserving the sign bit (bit 7).

```
LD    HL,ADDR
SRA   (HL)
```

Shifting while preserving the sign bit is called *sign extension.* A shift that operates in this manner is called an *arithmetic shift,* since it preserves the sign of a two's complement number. It can therefore be used to divide or normalize signed numbers.

6. Rotate right the memory location eight bytes beyond the address in INDIR and INDIR+1.

```
        LD    xy,(INDIR)        ;GET INDIRECT ADDRESS
        RR    (xy+8)            ;ROTATE BYTE 8 RIGHT
```

MAKING DECISIONS

In this section procedures are presented for making the following three types of decisions:

- Branching if a bit is set or cleared
- Branching if two values are equal or not equal
- Branching if one value is greater or less than another.

The first type of decision allows the processor to sense the value of a flag, switch, status line, or other binary (ON/OFF) input. The second type of decision allows the processor to determine whether an input or a result has a specific value (an input is a specific command character or terminator, or a result is 0). The third type of decision allows the processor to determine whether a value is above or below a numerical threshold (a value is valid or invalid, or is above or below a warning level or setpoint). Assuming that the primary value is in the accumulator and the secondary value (if needed) is at address ADDR, the procedures are as follows.

Branching Set or Cleared Bit

Determine if a bit is set or cleared with the BIT instruction. The operands are the bit position and the register or memory address (either the one in HL or one accessed via indexing). The Zero flag reflects the bit value and can be used for branching.

Examples

1. Branch to DEST if bit 5 of the accumulator is 1.

```
BIT   5,A
JR    NZ,DEST
```

JP (absolute addressing) can be used instead of JR (relative addressing). The Zero flag is set to 1 if and only if bit 5 of A is 0.

2. Branch to DEST if bit 2 of register C is 0.

```
BIT   2,C
JR    Z,DEST
```

3. Branch to DEST if bit 6 of memory location ADDR is 1.

```
LD    HL,ADDR
BIT   6,(HL)
JR    NZ,DEST
```

4. Branch to DEST if bit 3 of the memory location seven bytes beyond the address in INDIR and INDIR+1 is 0.

```
LD    xy,(INDIR)
BIT   3,(xy+7)
JR    Z,DEST
```

There are shortcuts for bits 0, 6, and 7 of the accumulator.

5. Branch to DEST if bit 7 of the accumulator is 1.

```
AND   A                 ;ESTABLISH SIGN FLAG
JP    M,DEST
```

There is no relative jump based on the Sign flag.

6. Branch to DEST if bit 6 of the accumulator is 0.

```
ADD   A,A               ;ESTABLISH SIGN FLAG FROM BIT 6
JP    P,DEST
```

7. Branch to DEST if bit 0 of the accumulator is 1.

```
        RRA                     ;MOVE BIT 0 TO CARRY
        JR      C,DEST
```

Here we have the choice of either a relative or an absolute jump.

Branching Based on Equality

Determine if the value in the accumulator is equal to another value by subtraction. The Zero flag is set to 1 if the values are equal. Compare instructions (CP) are more useful than subtract instructions (SBC or SUB) because compares preserve the value in the accumulator for later operations. Note, however, that the Z80 has a 16-bit subtract with borrow instruction (SBC HL), but no 16-bit compare or subtract instruction.

Examples

1. Branch to DEST if the accumulator contains the number VALUE.

```
        CP      VALUE           ;DOES A CONTAIN VALUE?
        JR      Z,DEST          ;YES, BRANCH
```

2. Branch to DEST if the contents of the accumulator are not equal to the contents of memory location ADDR.

```
        LD      HL,ADDR
        CP      (HL)            ;IS A THE SAME AS DATA IN MEMORY?
        JR      NZ,DEST         ;NO, BRANCH
```

There are shortcuts if VALUE is 0, 1, or FF_{16}.

3. Branch to DEST if the accumulator contains 0.

```
        AND     A               ;ESTABLISH ZERO FLAG
        JR      Z,DEST          ;BRANCH IF A CONTAINS ZERO
```

4. Branch to DEST if the accumulator does not contain FF_{16}.

```
        INC     A               ;ESTABLISH ZERO FLAG
        JR      NZ,DEST         ;BRANCH IF A WAS NOT FF
```

This procedure can be applied to any 8-bit register or to a memory location addressed through HL or via indexing.

5. Branch to DEST if the accumulator contains 1.

```
        DEC     A               ;ESTABLISH ZERO FLAG
        JR      Z,DEST          ;BRANCH IF A WAS 1
```

6. Branch to DEST if memory location ADDR contains 0.

```
        LD     HL,ADDR
        INC    (HL)            ;ESTABLISH ZERO FLAG IN TWO STEPS
        DEC    (HL)
        JR     Z,DEST          ;BRANCH IF ADDR CONTAINS ZERO
```

This procedure will also work on data at an indexed address or in registers B, C, D, E, H, or L.

7. Branch to DEST if register pair HL contains VAL16.

```
        AND    A               ;CLEAR CARRY, DON'T CHANGE A
        LD     rp,VAL16
        SBC    HL,rp           ;DOES HL CONTAIN VAL16?
        JR     Z,DEST          ;YES, BRANCH
```

The 16-bit subtraction instruction always includes the Carry and is available only for HL and another register pair (BC, DE, or SP).

Branching Based on Magnitude Comparisons

Determine if the value in the accumulator is greater than or less than some other value by subtraction. If, as is typical, the values are unsigned, the Carry flag indicates which is larger. In general,

- Carry = 1 if the value subtracted is larger than the value in the accumulator (that is, if a borrow is required).
- Carry = 0 if the value in the accumulator is larger or if the two values are equal.

Since subtracting equal values makes the Carry 0, the alternatives (considering the accumulator as the primary operand) are

- Primary operand less than secondary operand (Carry set)
- Primary operand greater than or equal to secondary operand (Carry cleared).

If the alternatives you need are "less than or equal to" and "greater than," you can simply exchange the primary and secondary operands (that is, from Y − X instead of X − Y).

Examples

1. Branch to DEST if the contents of the accumulator are greater than or equal to the number VALUE.

```
        CP     VALUE           ;IS A ABOVE VALUE?
        JR     NC,DEST         ;YES, BRANCH
```

2. Branch to DEST if the contents of memory address OPER1 are less than the contents of memory address OPER2.

```
LD    A,(OPER1)       ;GET FIRST OPERAND
LD    HL,OPER2
CP    (HL)            ;IS IT LESS THAN SECOND OPERAND?
JR    C,DEST          ;YES, BRANCH
```

3. Branch to DEST if the contents of memory address OPER1 are less than or equal to the contents of memory address OPER2.

```
LD    A,(OPER2)       ;GET SECOND OPERAND
LD    HL,OPER1
CP    (HL)            ;IS IT GREATER THAN OR EQUAL TO FIRST?
JR    NC,DEST         ;YES, BRANCH
```

If we loaded the accumulator with OPER1 and compared to OPER2, we could branch only on the conditions

- OPER1 greater than or equal to OPER2 (Carry cleared)
- OPER1 less than OPER2 (Carry set).

Since neither is what we want, we must reverse the order in which the operands are handled.

4. Branch to DEST if the contents of register pair HL are greater than or equal to VAL16.

```
AND   A               ;CLEAR CARRY
LD    rp,VAL16        ;IS HL ABOVE VAL16?
SBC   HL,rp
JR    NC,DEST         ;YES, BRANCH
```

If the values are signed, we must allow for the possible occurrence of two's complement overflow.[7] This is the situation in which the difference between the numbers cannot be contained in seven bits and, therefore, the sign bit is changed. For example, if one number is +7 and the other is −125, the difference is −132, which is beyond the capacity of eight bits (it is less than −128, the most negative number that eight bits can hold).

If overflow is a possibility, we can determine if it occurred by examining the Parity/Overflow flag after the addition or subtraction instruction. If that flag is 1, overflow did occur. The mnemonics here are confusing, since the Parity/Overflow flag normally indicates whether the result has even parity; the branches are therefore PE (Parity Even or Overflow Set) and PO (Parity Odd or Overflow Clear). Weller clarifies the situation by defining additional mnemonics JV and JNV.[8]

Thus, in the case of signed numbers, we must allow for the following possibilities:

- The result has the sign (positive or negative, as shown by the Sign flag) that we want, and the Parity/Overflow flag indicates that the sign is valid.

· The result does not have the sign that we want, but the Parity/Overflow flag indicates that two's complement overflow has changed the real sign.

We have to look for both a true positive (the sign we want, unaffected by overflow) or a false negative (the opposite of the sign we want, but inverted by two's complement overflow).

Examples

1. Branch to DEST if the accumulator contains a signed number greater than or equal to the number VALUE.

```
        CP   VALUE      ;PERFORM THE COMPARISON
        JP   PE,FNEG    ;DID OVERFLOW OCCUR?
        JP   P,DEST     ;NO, BRANCH IF RESULT POSITIVE
        JR   DONE
FNEG:   JP   M,DEST     ;YES, BRANCH IF RESULT NEGATIVE
DONE:   NOP
```

There are no relative jumps based on the Parity/Overflow flag.

2. Branch to DEST if the accumulator contains a signed number less than the contents of memory address ADDR.

```
        LD   HL,ADDR
        CP   (HL)       ;PERFORM THE COMPARISON
        JP   PE,FPOS    ;DID OVERFLOW OCCUR?
        JP   M,DEST     ;NO, BRANCH IF RESULT NEGATIVE
        JR   DONE
FPOS:   JP   P,DEST     ;YES, BRANCH IF RESULT POSITIVE
DONE:   NOP
```

Remember, JP PE means "jump on overflow," while JP PO means "jump on no overflow."

The programmer should also note that this is one of the few cases in which the Z80 is not fully upward-compatible with the 8080 microprocessor.[9] The 8080 has no overflow indicator and the P flag always indicates even parity.

There are some cases in which overflow cannot occur and all we must do is use the Sign flag instead of the Carry flag for branching. These cases are the following:

· The two numbers have the same sign. When this occurs, the difference is smaller in magnitude than the larger of the two numbers and overflow cannot occur. You can easily determine if two numbers have the same sign by EXCLUSIVE ORing them together and checking the Sign flag. Remember, the EXCLUSIVE OR of two bits is 1 if and only if the two bits have different values.

```
        XOR  VALUE      ;COULD OVERFLOW OCCUR?
        JP   P,NOOVF    ;NOT IF SIGNS ARE THE SAME
```

· A value is being compared with zero. In this case, the Sign flag must be set and examined.

Examples

1. Jump to DEST if the accumulator contains a signed positive number.

```
        AND   A          ;SET FLAGS FROM VALUE IN A
        JP    P,DEST
```

2. Jump to DEST if an 8-bit register contains a signed negative number.

```
        INC   reg        ;SET FLAGS FROM VALUE IN REGISTER
        DEC   reg
        JP    M,DEST
```

This sequence does not affect the accumulator or the register.

3. Jump to DEST if memory location ADDR contains a signed positive number.

```
        LD    HL,ADDR    ;POINT TO DATA IN MEMORY
        INC   (HL)
        DEC   (HL)
        JP    P,DEST     ;BRANCH IF DATA IS POSITIVE
```

This sequence does not affect the accumulator or the memory location.

Tables 1-7 and 1-8 summarize the common instruction sequences for making decisions with the Z80 microprocessor. Table 1-7 lists the sequences that depend only on the value in the accumulator; Table 1-8 lists the sequences that depend on numerical comparisons between the value in the accumulator and a specific number, the contents of a register, or the contents of a memory location (addressed through HL or an index register). Table 1-9 contains the sequences that depend only on the contents of a memory location.

LOOPING

The simplest way to implement a loop (that is, to repeat a sequence of instructions) with the Z80 microprocessor is to perform the following steps:

1. Load register B with the number of times the sequence is to be repeated.

2. Execute the sequence.

3. Use the DJNZ instruction to decrement register B and return to Step 2 if the result is not 0.

The DJNZ instruction is useful for loop control since it combines a decrement and a conditional relative branch. Note that DJNZ always operates on register B and

Table 1-7. Decision Sequences Depending on the Accumulator Alone

Condition	Flag Setting Instruction	Conditional Jump
Any bit = 0	BIT n,A	JR Z or JP Z
Any bit = 1	BIT n,A	JR NZ or JP NZ
Bit 7 = 0	RLA, RLCA, or ADD A,A	JR NC or JP NC
Bit 7 = 1	RLA, RLCA, or ADD A,A	JR C or JP C
Bit 6 = 0	ADD A,A	JP P
Bit 6 = 1	ADD A,A	JP M
Bit 0 = 0	RRA or RRCA	JR NC or JP NC
Bit 0 = 1	RRA or RRCA	JR C or JP C
Equals zero	AND A or OR A	JR Z or JP Z
Not equal to zero	AND A or OR A	JR NZ or JP NZ
Positive (MSB = 0)	AND A or OR A	JP P
Negative (MSB = 1)	AND A or OR A	JP M

Table 1-8. Decision Sequences Depending on Numerical Comparisons with the Accumulator (Using CP)

Condition	Conditional Jump
Equal	JR Z or JP Z
Not equal	JR NZ or JP NZ
Greater than or equal (unsigned)	JR NC or JP NC
Less than (unsigned)	JR C or JP C
Greater than or equal (signed)	JP P (assuming no overflow)
Less than (signed)	JP M (assuming no overflow)
Note: All conditions assume that the accumulator contains the primary operand; for example, less than means "accumulator less than other operand."	

Table 1-9. Decision Sequences Depending on a Memory Location Alone

Condition	Flag Setting Instruction(s)	Conditional Jump
Any bit = 0	BIT n, (HL) or (xy+OFFSET)	JR Z or JP Z
Any bit = 1	BIT n,(HL) or (xy+OFFSET)	JR NZ or JP NZ
= 0	INC,DEC	JR Z or JP Z
$\neq 0$	INC,DEC	JR NZ or JP NZ

branches if B is not decremented to 0—the instruction set does not provide any other combinations. However, DJNZ has limitations: It allows only an 8-bit counter and an 8-bit offset for the relative branch (the branch is thus limited to 129 bytes forward or 126 backward from the first byte of the instruction).

Typical programs look like the following:

```
         LD   B,NTIMES        ;NTIMES = NUMBER OF REPETITIONS
LOOP:    .
         .    Instructions to be repeated
         .
         .
         DJNZ LOOP
```

We could, of course, use other 8-bit registers or count up rather than counting down. These alternative approaches would require a slightly different initialization, an explicit DEC or INC instruction, and a conditional JR or JP instruction. In any case, the instructions to be repeated must not interfere with the counting of the repetitions. Note that register B is special, and most programmers reserve it as a loop counter.

The 8-bit length of register B limits this simple loop to 256 repetitions. The programmer can provide larger numbers of repetitions by nesting single-register loops or by using a register pair as illustrated in the following examples:

· Nested loops

```
         LD   C,NTIMM         ;START OUTER COUNTER
LOOPO:   LD   B,NTIML         ;START INNER COUNTER
LOOPI:   .
         .
         .     Instructions to be repeated
         .
         DJNZ LOOPI           ;DECREMENT INNER COUNTER
         DEC  C               ;DECREMENT OUTER COUNTER
         JR   NZ,LOOPO
```

The outer loop restores the inner counter (register B) to its starting value (NTIML) after each decrement of the outer counter (register C). The nesting produces a multiplicative factor—the instructions starting at LOOPI are repeated NTIMM × NTIML times. We use register B as the inner counter to take maximum advantage of DJNZ. (Clearly, the inner loop is executed many more times than the outer loop.)

· A register pair as 16-bit counter

```
         LD   BC,NTIMES       ;INITIALIZE 16-BIT COUNTER
LOOP:    .
         .
         .    Instructions to be repeated
         .
         DEC  BC
         LD   A,B             ;TEST 16-BIT COUNTER FOR ZERO
         OR   C
         JR   NZ,LOOP
```

The extra steps are necessary because DEC rp (or DEC xy) does not affect the Zero flag (so there is no way of telling if the count has reached 0). The simplest way to determine if a 16-bit register pair contains 0 is to logically OR the two registers. The result of the logical OR is 0 if and only if all bits in both registers are 0's. Check this procedure by hand if you are not sure why it works. A major drawback to this approach is its use of the accumulator, which requires saving the previous contents if they are needed in the next iteration.

ARRAY MANIPULATION

The simplest way to access a particular element of an array is to place the element's address in register pair HL. In this way, it is possible to

- Manipulate the element by referring to it indirectly, that is, as (HL).
- Access the succeeding element (at the next higher address) by using INC to increment register pair HL or access the preceding element (at the next lower address) by using DEC to decrement HL.
- Access an arbitrary element by loading another register pair with the element's offset from the address in HL and using the ADD HL instruction. If the offset is fixed, we can also use indexing from a base address in either index register.

Typical array manipulation procedures are easy to program if the array is one-dimensional and the elements each occupy one byte. Some examples are

- Add an element of an array to the accumulator. Assume that the address of the element is in register pair HL. Update HL so that it contains the address of the succeeding 8-bit element.

```
        ADD   (HL)        ;ADD CURRENT ELEMENT
        INC   HL          ;ADDRESS NEXT ELEMENT
```

- Check to see if an element of an array is 0 and add 1 to register C if it is. Assume that the element's address is in register pair HL. Update HL so that it contains the address of the preceding 8-bit element.

```
        LD    A,(HL)      ;GET CURRENT ELEMENT
        AND   A           ;IS IT ZERO?
        JR    NZ,UPDDT
        INC   C           ;YES, ADD 1 TO COUNT OF ZEROS
UPDDT:  DEC   HL          ;ADDRESS PRECEDING ELEMENT
```

- Load the accumulator with the 35th element of an array. Assume that the base address of the array is in register pair HL.

```
        LD    DE,35       ;GET OFFSET FOR REQUIRED ELEMENT
        ADD   HL,DE       ;CALCULATE ADDRESS OF ELEMENT
        LD    A,(HL)      ;OBTAIN THE ELEMENT
```

ADD HL,DE performs a 16-bit addition, using register pair HL as a 16-bit accumulator. Note that the 16-bit offset in register pair DE can be either positive or negative.

The following single instruction performs the same task if the offset is an 8-bit unsigned number and the base address is in an index register:

```
        LD    A,(xy+35)  ;OBTAIN THE ELEMENT IN ONE STEP
```

Manipulating array elements becomes more difficult if more than one element is needed during each iteration (as in a sort that requires interchanging of elements), if the elements are more than one byte long, or if the elements are themselves addresses (as in a table of starting addresses). The basic problems are the lack of indexing with a variable offset and the lack of instructions that access 16-bit items indirectly. Some examples of more general array manipulation are

· Load register pair DE with a 16-bit element of an array (stored LSB first). The starting address of the element is in register pair HL. Update HL so that it points to the next 16-bit element.

```
        LD    E,(HL)      ;GET LSB OF ELEMENT
        INC   HL
        LD    D,(HL)      ;GET MSB OF ELEMENT
        INC   HL          ;ADDRESS NEXT ELEMENT
```

· Exchange an element of an array with its successor if the two are not already in descending order. Assume that the elements are 8-bit unsigned numbers and that the address of the current element is in register pair HL. Update HL so that it contains the address of the successor element.

```
        LD    A,(HL)      ;GET CURRENT ELEMENT
        INC   (HL)
        CP    (HL)        ;IS IT LESS THAN SUCCESSOR?
        JR    NC,DONE     ;NO, NO INTERCHANGE NECESSARY
        LD    B,(HL)      ;YES, START THE INTERCHANGE
        LD    (HL),A      ;CURRENT ELEMENT TO NEW POSITION
        DEC   HL
        LD    (HL),B      ;SUCCESSOR ELEMENT TO NEW POSITION
        INC   HL
DONE:   NOP
```

This procedure is awkward because the processor can address only one element at a time using HL. Clearly, the problem would be even more serious if the two elements were more than one position apart.

An alternative approach is to use an index register; that is,

```
        LD    A,(xy+0)    ;GET CURRENT ELEMENT
        CP    (xy+1)      ;IS IT LESS THAN SUCCESSOR?
        JR    NC,DONE     ;NO, NO INTERCHANGE NECESSARY
        LD    B,(xy+0)    ;YES, START THE INTERCHANGE
        LD    (xy+1),A    ;CURRENT ELEMENT TO NEW POSITION
        LD    (xy+0),B    ;SUCCESSOR ELEMENT TO NEW POSITION
DONE:   INC   xy          ;MOVE ON TO NEXT PAIR
```

· Load the accumulator from the 12th indirect address in a table. Assume that the base address of the table is in register pair HL.

```
LD    DE,24       ;GET DOUBLED OFFSET FOR ELEMENT
ADD   HL,DE       ;CALCULATE STARTING ADDRESS OF ELEMENT
LD    E,(HL)      ;GET LSB OF INDIRECT ADDRESS
INC   HL
LD    D,(HL)      ;GET MSB OF INDIRECT ADDRESS
LD    A,(DE)      ;OBTAIN DATA FROM INDIRECT ADDRESS
```

An alternative approach using an index register is

```
LD    A,(xy+24)   ;GET LSB OF INDIRECT ADDRESS
LD    E,A
LD    A,(xy+25)   ;GET MSB OF INDIRECT ADDRESS
LD    D,A
LD    A,(DE)      ;OBTAIN DATA FROM INDIRECT ADDRESS
```

Note that in either approach you must double the index to handle tables containing addresses, since each 16-bit address occupies two bytes of memory.

Some ways to simplify array processing are

· Keep the base address of the table or array in register pair DE (or BC), so ADD HL or ADD xy does not destroy it.

· Use ADD A,A to double an index in the accumulator. The doubled index can then be used to handle arrays or tables consisting of 16-bit elements. ADD HL,HL or ADD xy,xy may be used to double 16-bit indexes.

· Use EX DE,HL to move addresses to and from register pair HL.

Chapters 5 and 9 contain further examples of array manipulation.

Block Move and Block Compare Instructions

Another way to simplify array processing is to use the Z80's block move and block compare instructions. The block move instructions not only transfer data from one memory location to another without using the accumulator, but they also update the array pointers and decrement a 16-bit loop counter. Thus, a block move instruction can replace a sequence of load, increment, and decrement instructions. Repeated block move instructions continue transferring data, updating the pointers, and decrementing the counter until the counter is decremented to zero. Block compare instructions are similar to block moves, except that only a single pointer is involved (the other operand is in the accumulator), and the repeated versions also terminate if the operands being compared are equal (this is referred to as a *true comparison*).

A further convenience of block moves and block compares is that they solve the problem of testing a 16-bit counter for 0. Both block moves and block compares clear the Parity/Overflow flag if the 16-bit counter (always in register pair BC) is decremented to zero, and set the Parity/Overflow flag otherwise. Note that the indicator is the Parity/Overflow flag, *not* the Zero flag.

The block move and compare instructions are the following:

- LDI (LDD) moves a byte of data from the address in HL to the address in DE, decrements BC, and increments (decrements) DE and HL.
- LDIR (LDDR) repeats LDI (LDD) until BC is decremented to 0.
- CPI (CPD) compares the accumulator to the data at the address in HL, decrements BC, and increments (decrements) HL. Both CPI and CPD set the Zero flag if the operands being compared are equal, and clear the Zero flag otherwise.
- CPIR (CPDR) repeats CPI (CPD) until BC is decremented to 0.

Note that block moves reserve BC, DE, and HL for special purposes, while block compares reserve only BC and HL.

Examples

1. Move a byte of data from memory location ADDR1 to memory location ADDR2.

```
LD    BC,1        ;NUMBER OF BYTES TO MOVE = 1
LD    DE,ADDR1    ;INITIALIZE SOURCE POINTER
LD    HL,ADDR2    ;INITIALIZE DESTINATION POINTER
LDI or LDD        ;MOVE A BYTE OF DATA
```

Obviously, the overhead of loading all the register pairs makes it uneconomical to use LDI or LDD to move a single byte of data.

2. Move two bytes of data from memory locations ADDR1 and ADDR1+1 to memory locations ADDR2 and ADDR2+1.

```
LD    BC,2        ;NUMBER OF BYTES TO MOVE = 2
LD    DE,ADDR1    ;INITIALIZE SOURCE POINTER
LD    HL,ADDR2    ;INITIALIZE DESTINATION POINTER
LDIR              ;MOVE TWO BYTES OF DATA
```

or

```
LD    BC,2          ;NUMBER OF BYTES TO MOVE = 2
LD    DE,ADDR1+1    ;INITIALIZE SOURCE POINTER
LD    HL,ADDR2+1    ;INITIALIZE DESTINATION POINTER
LDDR                ;MOVE TWO BYTES OF DATA
```

The block move instructions become more useful as the number of bytes to be moved increases.

3. Move ten bytes of data from memory locations starting at ADDR1 to memory locations starting at ADDR2.

```
LD    BC,10        ;NUMBER OF BYTES TO MOVE = 10
LD    DE,ADDR1     ;INITIALIZE SOURCE POINTER
LD    HL,ADDR2     ;INITIALIZE DESTINATION POINTER
LDIR               ;MOVE TEN BYTES OF DATA
```

or

```
LD    BC,10         ;NUMBER OF BYTES TO MOVE = 10
LD    DE,ADDR1+9    ;INITIALIZE SOURCE POINTER
LD    HL,ADDR2+9    ;INITIALIZE DESTINATION POINTER
LDDR                ;MOVE TEN BYTES OF DATA
```

4. Examine memory locations starting at ADDR until one is encountered that contains 0 or until 256 bytes have been examined.

```
LD    BC,100H      ;MAXIMUM LENGTH = 100 HEX = 256
LD    HL,ADDR      ;POINT TO START OF SEARCH AREA
SUB   A            ;GET ZERO FOR COMPARISON
CPIR
```

The final value of the Zero flag indicates why the program exited.

Zero flag = 1 if the program found a 0 in memory.

Zero flag = 0 if the program decremented BC to 0.

The block move and block compare instructions are convenient, but their forms are restricted and their applications are limited. The programmer must remember the following:

· BC always serves as the counter; it is decremented after each iteration. The Parity/Overflow flag (*not* the Zero flag) indicates whether BC has been decremented to 0. Be careful—the P/V flag is set to 0 if BC has been decremented to 0; the polarity is opposite of that used with the Zero flag. Thus, after a block move or block compare, the relevant conditional branches have the following meanings:

JP PE means "branch if BC has not been decremented to 0."

JP PO means "branch if BC has been decremented to 0."

· HL always serves as the source pointer in block moves and as the memory pointer in block compares. HL is incremented or decremented *after* the data is transferred or a comparison is performed.

· DE always serves as the destination pointer in block moves; it is not used in block compares. Like HL, DE is incremented or decremented *after* the data is transferred. Note also that LDI and LDIR increment both HL and DE, while LDD and LDDR decrement both pairs.

· Repeated block comparisons exit if either a true comparison occurs or BC is decremented to 0. Testing the Zero flag will determine which condition caused the exit.

TABLE LOOKUP

Although the Z80 processor has indexing, the calculations required for table lookup must be performed explicitly using the ADD HL or ADD xy instruction. This is because the Z80's indexing assumes a variable 16-bit address in an index register and a fixed 8-bit offset. As with array manipulation, table lookup is simple if the table consists of 8-bit data items; it is more complicated if the table contains longer items or addresses. The instructions EX DE,HL and JP (HL) or JP (xy) can be useful, but require the programmer to place the results in specific 16-bit registers.

Examples

1. Load the accumulator with an element from a table. Assume that the base address of the table is BASE (a constant) and the 16-bit index is in memory locations INDEX and INDEX+1 (MSB in INDEX+1).

```
LD   DE,BASE          ;GET BASE ADDRESS
LD   HL,(INDEX)       ;GET INDEX
ADD  HL,DE            ;CALCULATE ADDRESS OF ELEMENT
LD   A,(HL)           ;OBTAIN THE ELEMENT
```

Reversing the roles of DE and HL would slow down the program since LD DE,(ADDR) executes more slowly and occupies more memory than does LD HL,(ADDR). This asymmetry is caused by the fact that only LD HL,(ADDR) is an original 8080 instruction; the direct loads of other register pairs (including the stack pointer) are additions to the underlying 8080 instruction set.

2. Load the accumulator with an element from a table. Assume that the base address of the table is BASE (a constant) and the index is in the accumulator.

```
LD   L,A              ;EXTEND INDEX TO 16 BITS IN HL
LD   H,0
LD   DE,BASE          ;GET BASE ADDRESS
ADD  HL,DE            ;CALCULATE ADDRESS OF ELEMENT
LD   A,(HL)           ;OBTAIN THE ELEMENT
```

3. Load register pair DE with a 16-bit element from a table. Assume that the base address of the table is BASE (a constant) and the index is in the accumulator.

```
ADD  A,A              ;DOUBLE INDEX FOR 16-BIT ELEMENTS
LD   L,A              ;EXTEND INDEX TO 16 BITS
LD   H,0
```

```
        LD    BC,BASE          ;GET BASE ADDRESS
        ADD   HL,BC            ;CALCULATE STARTING ADDRESS
        LD    E,(HL)           ;GET LSB OF ELEMENT
        INC   HL
        LD    D,(HL)           ;GET MSB OF ELEMENT
```

You can also use the instruction ADD HL,HL to double the index; it is slower than ADD A,A but it automatically handles cases in which the doubled index is too large for 8 bits.

4. Transfer control (jump) to a 16-bit address obtained from a table. Assume that the base address of the table is BASE (a constant) and the index is in the accumulator.

```
        ADD   A,A              ;DOUBLE INDEX FOR 16-BIT ELEMENTS
        LD    L,A              ;EXTEND INDEX TO 16 BITS
        LD    H,0
        LD    BC,BASE          ;GET BASE ADDRESS
        ADD   HL,BC            ;CALCULATE STARTING ADDRESS
        LD    E,(HL)           ;GET LSB OF DESTINATION
        INC   HL
        LD    D,(HL)           ;GET MSB OF DESTINATION
        EX    DE,HL
        JP    (HL)             ;JUMP TO DESTINATION
```

The common uses of jump tables are to implement CASE statements (multi-way branches used in languages such as FORTRAN, Pascal, and PL/I), to decode commands from a keyboard, and to respond to function keys on a terminal.

CHARACTER MANIPULATION

The easiest way to manipulate characters on the Z80 processor is to treat them as unsigned 8-bit numbers. The letters and digits form ordered subsequences of the ASCII character set (for example, the ASCII version of the letter A is one less than the ASCII version of B). Appendix C contains a complete ASCII character set.

Examples

1. Branch to address DEST if the accumulator contains ASCII E.

```
        CP    'E'        ;IS DATA ASCII E?
        JR     Z,DEST    ;YES, BRANCH
```

2. Search a string starting at address STRNG until a non-blank character is found.

```
        LD    HL,STRNG   ;POINT TO START OF STRING
EXAMC:  LD    A,(HL)     ;GET NEXT CHARACTER
        CP    ' '        ;IS IT A BLANK?
        JR    NZ,DONE    ;NO, DONE
```

```
          INC   HL          ;YES, PROCEED TO NEXT CHARACTER
          JP    EXAMC
DONE:     NOP
```

or

```
          LD    HL,STRNG-1  ;POINT TO BYTE BEFORE STRING
EXAMC:    INC   HL
          LD    A,(HL)      ;GET NEXT CHARACTER
          CP    ' '         ;IS IT A BLANK?
          JR    Z,EXAMC     ;YES, KEEP LOOKING
```

We could make either version execute faster by placing the blank character in a general-purpose register (for example, register C) and comparing each character with that register (using CP C) rather than with an immediate data value.

We could also use the block compare instructions which combine the comparison and the incrementing of the pointer in HL. The CPI instruction, for example, not only compares the accumulator with the data at the address in HL, but also increments HL and decrements BC. Thus, the program using CPI is

```
          LD    HL,STRNG    ;POINT TO START OF STRING
          LD    A,' '       ;GET A BLANK FOR COMPARISON
EXAMC:    CPI               ;IS NEXT CHARACTER A BLANK?
          JR    Z,EXAMC     ;YES, KEEP LOOKING
```

The CPI instruction sets the Zero flag to 1 if the operands being compared are equal and to 0 if they are not equal. It also sets the Parity/Overflow flag to 0 if it decrements BC to 0 and to 1 if it does not, thus allowing the programmer to check easily for the termination of the string as well as for a true comparison. We cannot use CPIR here, since it would terminate as soon as a blank character (rather than a non-blank character) was found.

3. Branch to address DEST if the accumulator contains a letter between C and F, inclusive.

```
          CP    'C'         ;IS DATA BELOW C?
          JR    C,DONE      ;YES, DONE
          CP    'G'         ;IS DATA BELOW G?
          JR    C,DEST      ;YES, MUST BE BETWEEN C AND F
DONE:     NOP
```

We have taken advantage of the fact that G follows F numerically in ASCII, just as it does in the alphabet. Chapter 8 contains further examples of string manipulation.

CODE CONVERSION

You can convert data from one code to another using arithmetic or logical operations (if the relationship is simple) or lookup tables (if the relationship is complex).

Examples

1. Convert an ASCII digit to its binary-coded decimal (BCD) equivalent.

```
        SUB   '0'         ;CONVERT ASCII TO BCD
```

Since the ASCII digits form an ordered subsequence of the code, all that must be done is subtract the offset (ASCII 0).

You can also clear bits 4 and 5 with the instruction

```
        AND   11001111B ;CONVERT ASCII TO BCD
```

Either the arithmetic instruction or the logical instruction will convert ASCII 0 (30_{16}) to decimal 0 (00_{16}).

2. Convert a binary-coded-decimal (BCD) digit to its ASCII equivalent.

```
        ADD   A,'0'       ;CONVERT BCD TO ASCII
```

The inverse conversion is equally simple. Bits 4 and 5 can be set with the instruction

```
        OR    00110000B ;CONVERT BCD TO ASCII
```

Either the arithmetic instruction or the logical instruction will convert decimal 6 (06_{16}) to ASCII 6 (36_{16}).

3. Convert one 8-bit code to another using a lookup table. Assume that the lookup table starts at address NEWCD and is indexed by the value in the original code (for example, the 27th entry is the value in the new code corresponding to 27 in the original code). Assume that the data is in memory location CODE.

```
        LD    A,(CODE)    ;GET THE OLD CODE
        LD    L,A         ;EXTEND INDEX TO 16 BITS
        LD    H,0
        LD    DE,NEWCD    ;GET BASE ADDRESS
        ADD   HL,DE       ;CALCULATE ADDRESS OF ELEMENT
        LD    A,(HL)      ;GET THE ELEMENT
```

Indexed addressing cannot be used here, since memory location CODE contains a variable value.

Chapter 4 contains further examples of code conversion.

MULTIPLE-PRECISION ARITHMETIC

Multiple-precision arithmetic requires a series of 8-bit operations. They are

- Clear the Carry flag initially, since there is never a carry into or borrow from the least significant byte.

· Use the Add with Carry (ADC) or Subtract with Carry (SBC) instruction to perform an 8-bit operation and include the carry or borrow from the previous operation.

A typical 64-bit addition program is

```
          LD    B,8         ;NUMBER OF BYTES = 8
          SUB   A           ;CLEAR CARRY INITIALLY
          LD    HL,NUM1     ;POINT TO START OF NUMBERS
          LD    DE,NUM2
ADD8:     LD    A,(DE)      ;GET A BYTE OF ONE OPERAND
          ADC   A,(HL)      ;ADD A BYTE OF THE OTHER OPERAND
          LD    (HL),A      ;STORE THE 8-BIT SUM
          INC   DE          ;UPDATE POINTERS
          INC   HL
          DJNZ  ADD8        ;COUNT BYTE OPERATIONS
```

Chapter 6 contains further examples.

MULTIPLICATION AND DIVISION

There are many ways to implement multiplication. One approach is to convert multiplication by a small integer into a specific short sequence of additions and left shifts.

Examples

1. Multiply the contents of the accumulator by 2.

```
          ADD   A,A         ;DOUBLE A
```

2. Multiply the contents of the accumulator by 5.

```
          LD    B,A
          ADD   A,A         ;A TIMES 2
          ADD   A,A         ;A TIMES 4
          ADD   A,B         ;A TIMES 5
```

Both examples assume that no carries ever occur. ADD HL could be similarly used to produce a 16-bit result.

This approach is often handy in accessing elements of two-dimensional arrays. For example, assume a set of temperature readings taken at four different positions in each of three different storage tanks. Organize the readings as a two-dimensional array T(I,J), where I is the tank number (1, 2, or 3) and J identifies the position in the tank (1, 2, 3, or 4). Store the reading in the computer's memory one after another as follows, starting with the reading at position 1 of tank 1:

```
     BASE            T(1,1)    Reading at tank 1, position 1
     BASE+1          T(1,2)    Reading at tank 1, position 2
     BASE+2          T(1,3)    Reading at tank 1, position 3
```

```
BASE+3          T(1,4)     Reading at tank 1, position 4
BASE+4          T(2,1)     Reading at tank 2, position 1
BASE+5          T(2,2)     Reading at tank 2, position 2
BASE+6          T(2,3)     Reading at tank 2, position 3
BASE+7          T(2,4)     Reading at tank 2, position 4
BASE+8          T(3,1)     Reading at tank 3, position 1
BASE+9          T(3,2)     Reading at tank 3, position 2
BASE+10         T(3,3)     Reading at tank 3, position 3
BASE+11         T(3,4)     Reading at tank 3, position 4
```

Generally, the reading T(I,J) is located at address BASE + 4 * (I−1) + (J−1). If I is in the accumulator and J is in register B, the accumulator can be loaded with T(I,J) as follows:

```
DEC   A          ;OFFSET FOR TANK I
ADD   A,A        ;2 * (I-1)
ADD   A,A        ;4 * (I-1)
ADD   A,B        ;ADD OFFSET FOR POSITION J
DEC   A          ;4 * (I-1) + (J-1)
LD    L,A        ;EXTEND INDEX TO 16 BITS
LD    H,0
LD    DE,BASE    ;GET BASE ADDRESS OF READINGS
ADD   HL,DE      ;ACCESS DESIRED READING
LD    A,(HL)     ;FETCH T(I,J)
```

Extending this approach to handle arrays with more dimensions is shown in Chapter 5.

Division by a power of 2 can be implemented as a series of right logical shifts.

Example

Divide the contents of the accumulator by 4.

```
SRL   A          ;DIVIDE A BY 2
SRL   A          ;AND THEN BY 2 AGAIN
```

or

```
RRA              ;DIVIDE A BY 4 BY ROTATING IT TWICE
RRA
AND   00111111B  ;MAKE SHIFTS LOGICAL BY CLEARING MSB'S
```

The second alternative uses the one-byte instruction RRA, rather than the two-byte instruction SRL A. When multiplying or dividing signed numbers, be careful to separate the signs from the magnitudes. Replace logical shifts with arithmetic shifts that preserve the value of the sign bit.

Other approaches to multiplication and division include algorithms involving shifts and additions (multiplication) or shifts and subtractions (division) as described in Chapter 6, and lookup tables as discussed previously in this chapter.

LIST PROCESSING

Additional information on the following material can be found in an article by K.S. Shankar published in *IEEE Computer.*[10]

Lists can be processed like arrays if the elements are stored in consecutive addresses. If the elements are queued or chained, however, the limitations of the instruction set are evident because

- Indexed addressing allows only an 8-bit fixed offset.
- No indirect addressing is available, except through register pairs or index registers.
- Addresses in register pairs or index registers can be used only to retrieve or store 8-bit data.

Examples

1. Retrieve an address stored starting at the address in register pair HL. Place the retrieved address in HL.

```
LD      E,(HL)      ;GET LSB OF LINK
INC     HL
LD      D,(HL)      ;GET MSB OF LINK
EX      DE,HL       ;REPLACE CURRENT POINTER WITH LINK
```

This procedure allows you to move from one element to another in a linked list.

2. Retrieve data from the address currently in memory locations INDIR and INDIR+1 and increase that address by 1.

```
LD      HL,(INDIR);GET POINTER FROM MEMORY
LD      A,(HL)      ;GET DATA USING POINTER
INC     HL          ;UPDATE POINTER BY 1
LD      (INDIR),HL
```

This procedure allows the use of the address in memory as a pointer to the next available location in a buffer.

3. Store an address from DE starting at the address currently in register pair HL. Increment HL by 2.

```
LD      (HL),E      ;STORE LSB OF POINTER
INC     HL
LD      (HL),D      ;STORE MSB OF POINTER
INC     HL          ;COMPLETE UPDATING OF HL
```

This procedure allows building a list of addresses. Such a list could be used, for example, to write threaded code in which each routine concludes by transferring control to its successor. The list could also contain the starting addresses of a series of test procedures or tasks or the addresses of memory locations or I/O devices assigned by the operator to particular functions.

GENERAL DATA STRUCTURES

Additional information on the following material can be found in the book *Data Structures Using Pascal* by A. Tenenbaum and M. Augenstein.[11] There are several versions of this book by the same authors for different languages and computers.

More general data structures can be handled using the procedures for array manipulation, table lookup, and list processing that have been described earlier. The key limitations in the instruction set are the same ones mentioned in the discussion of list processing.

Examples

1. Queues or linked lists. Assume there is a queue header consisting of the base address of the first element in memory locations HEAD and HEAD+1. If there are no elements in the queue, HEAD and HEAD+1 both contain 0. The first two locations in each element contain the base address of the next element or 0 if there is no next element.

· Add an element to the head of the queue. Assume that the element's base address is in DE.

```
        LD      HL,HEAD         ;REPLACE HEAD, SAVING OLD VALUE
        LD      A,(HL)          ;MOVE LESS SIGNIFICANT BYTES
        LD      (HL),E
        INC     HL
        LD      B,(HL)          ;MOVE MORE SIGNIFICANT BYTES
        LD      (HL),D
        LD      (DE),A          ;NEW HEAD POINTS TO OLD HEAD
        LD      A,B             ;INCLUDING MORE SIGNIFICANT BYTES
        INC     DE
        LD      (DE),A
```

· Remove an element from the head of the queue and set the Zero flag if no element is available. Place the base address of the element (or 0 if there is no element) in DE.

```
        LD      HL,HEAD         ;OBTAIN HEAD OF QUEUE
        LD      E,(HL)          ;LESS SIGNIFICANT BYTE
        INC     HL
        LD      D,(HL)          ;MORE SIGNIFICANT BYTE
        LD      A,D
        OR      E               ;ANY ELEMENTS IN QUEUE?
        JR      Z,DONE          ;NO, DONE
        INC     DE              ;YES, MAKE NEXT ELEMENT NEW HEAD
        LD      A,(DE)
        LD      (HL),A          ;MORE SIGNIFICANT BYTE
        DEC     DE
        DEC     HL
        LD      (DE),A          ;LESS SIGNIFICANT BYTE
        LD      (HL),A
DONE:   NOP
```

Since no instruction after OR E affects any flags, the final value of the Zero flag indicates whether the queue was empty.

2. Stacks. Assume there is a stack structure consisting of 8-bit elements. The address of the next empty location is in addresses SPTR and SPTR+1. The lowest address that the stack can occupy is LOW and the highest address is HIGH. Note that this software stack grows up in memory (toward higher addresses), whereas the Z80's hardware stack grows down (toward lower addresses).

· If the stack overflows, set the Carry flag and exit. Otherwise, store the accumulator in the stack and increase the stack pointer by 1. Overflow means that the stack has expanded beyond its assigned area.

```
        LD   HL,(SPTR)      ;GET THE STACK POINTER
        EX   DE,HL
        LD   HL,-(HIGH+1)   ;CHECK FOR STACK OVERFLOW
        ADD  HL,DE          ;SET CARRY IF STACK OVERFLOWS
        JR   C,DONE         ;AND EXIT ON OVERFLOW
        EX   DE,HL          ;GET STACK POINTER BACK
        LD   (HL),A         ;STORE ACCUMULATOR IN STACK
        INC  HL             ;UPDATE STACK POINTER
        LD   (SPTR),HL
DONE:   NOP
```

· If the stack underflows, set the Carry flag and exit. Otherwise, decrease the stack pointer by 1 and load the accumulator from the stack. Underflow means that an attempt has been made to remove data from an empty stack.

```
        LD   HL,(SPTR)      ;GET THE STACK POINTER
        EX   DE,HL
        LD   HL,-(LOW+1)    ;CHECK FOR STACK UNDERFLOW
        ADD  HL,DE          ;CLEAR CARRY IF STACK UNDERFLOWS
        JR   NC,DONE        ;AND EXIT ON UNDERFLOW
        EX   DE,HL          ;GET STACK POINTER BACK
        DEC  HL             ;UPDATE STACK POINTER
        LD   A,(HL)         ;LOAD ACCUMULATOR FROM STACK
        LD   (SPTR),HL      ;RESTORE STACK POINTER
DONE:   CCF                 ;SET CARRY ON UNDERFLOW
```

Both example programs utilize the fact that ADD HL affects only the Carry flag. Remember, ADD HL does not affect the Zero flag. Note also that DEC rp and INC rp do not affect any flags.

PARAMETER PASSING TECHNIQUES

The most common ways to pass parameters on the Z80 microprocessor are

1. In registers. Seven 8-bit primary user registers (A, B, C, D, E, H, and L) are available, and the three register pairs (BC, DE, and HL) and two index registers (IX and IY) may be used readily to pass addresses. This approach is adequate in simple cases, but it lacks generality and can handle only a limited number of parameters. The programmer must remember the normal uses of the registers in assigning parameters. In other words,

- The accumulator is the obvious place to put a single 8-bit parameter.
- Register pair HL is the obvious place to put a single address-length (16-bit) parameter.
- Register pair DE is a better place to put a second address-length parameter than register pair BC, because of the EX DE,HL instruction.
- An index register (IX or IY) is the obvious place to put the base address of a data structure when elements are available at fixed offsets.

This approach is reentrant as long as the interrupt service routines save and restore all the registers.

2. **In an assigned area of memory.** There are two ways to implement this approach. One is to place the base address of the assigned area in an index register. Then particular parameters may be accessed with fixed offsets. The problem here is that the Z80's indexing is extremely time-consuming. An alternative is to place the base address in HL. Then parameters must be retrieved in consecutive order, one byte at a time.

In either alternative, the calling routine must store the parameters in memory and load the starting address into the index register or HL before transferring control to the subroutine. This approach is general and can handle any number of parameters, but it requires a lot of management. If different areas of memory are assigned for each call or each routine, a unique stack is essentially created. If a common area of memory is used, reentrancy is lost. In this method, the programmer is responsible for assigning areas of memory, avoiding interference between routines, and saving and restoring the pointers required to resume routines after subroutine calls or interrupts.

3. **In program memory immediately following the subroutine call.** If this approach is used, remember the following:

- The base address of the memory area is at the top of the stack; that is, the base address is the normal return address, the location of the instruction immediately following the call. The base address can be moved to an index register by popping the stack with

```
        POP   xy          ;RETRIEVE BASE ADDRESS OF PARAMETERS
```

Now access the parameters with fixed offsets from the index register. For example, the accumulator can be loaded with the first parameter by using the instruction

```
        LD    A,(xy+0)    ;MOVE FIRST PARAMETER TO A
```

· All parameters must be fixed for a given call, since the program memory is typically read-only.

· The subroutine must calculate the actual return address (the address immediately following the parameter area) and place it on top of the stack before executing a RET instruction.

Example

Assume that subroutine SUBR requires an 8-bit parameter and a 16-bit parameter. Show a main program that calls SUBR and contains the required parameters. Also show the initial part of the subroutine that retrieves the parameters, storing the 8-bit item in the accumulator and the 16-bit item in register pair HL, and places the correct return address at the top of the stack.

Subroutine call

```
        CALL SUBR        ;EXECUTE SUBROUTINE
        DEFB PAR8        ;8-BIT PARAMETER
        DEFW PAR16       ;16-BIT PARAMETER
        ...  next instruction  ...
```

Subroutine

```
SUBR:   POP  xy          ;POINT TO START OF PARAMETER AREA
        LD   A,(xy+1)    ;GET LSB OF 16-BIT PARAMETER
        LD   E,A
        LD   A,(xy+2)    ;GET MSB OF 16-BIT PARAMETER
        LD   D,A
        LD   A,(xy+0)    ;GET 8-BIT PARAMETER
        LD   BC,3        ;UPDATE RETURN ADDRESS
        ADD  xy,BC
        PUSH xy
        .
        .
        . . . remainder of subroutine . . .
        .
        .
        RET              ;RETURN TO NEXT INSTRUCTION
```

The initial POP xy instruction loads the index register with the return address that CALL SUBR saved at the top of the stack. In fact, the return address does not contain an instruction; instead, it contains the first parameter (PAR8). The next instructions move the parameters to their respective registers. Finally, adding 3 to the return address and saving the sum in the stack makes the final RET instruction transfer control back to the instruction following the parameters.

This approach allows parameter lists of any length. However, obtaining the parameters from memory and adjusting the return address is awkward at best; it becomes a longer and slower process as the number of parameters increases.

4. In the stack. When using this approach, remember the following:

· CALL stores the return address at the top of the stack. The parameters that the calling routine placed in the stack begin at address ssss + 2, where ssss is the contents of the stack pointer. The 16-bit return address occupies the top two locations of the stack, and the stack pointer itself always refers to the lowest occupied address, not the highest empty one.

· The subroutine can determine the value of the stack pointer (the location of the parameters) by (a) storing it in memory with LD (ADDR),SP or (b) using the sequence

```
        LD    HL,0         ;MOVE STACK POINTER TO HL
        ADD   HL,SP
```

This sequence places the stack pointer in register pair HL (the opposite of LD SP,HL). We can use an index register instead of HL if HL is reserved for other purposes.

· The calling program must place the parameters in the stack and assign space for the results before calling the subroutine. It must also remove the parameters from the stack (often referred to as *cleaning the stack*) afterward. Cleaning the stack is simple if the programmer always places the parameters above the empty area assigned to the results. Then the parameters can be removed, leaving the results at the top. The next example illustrates how this is done. An obvious alternative is for the results to replace some or all of the parameters.

· Stack locations can be allocated dynamically for results with the sequence

```
        LD    HL,-NRESLT       ;LEAVE ROOM FOR RESULTS
        ADD   HL,SP
        LD    SP,HL
```

This sequence leaves NRESLT empty locations at the top of the stack as shown in Figure 1-8. Of course, if NRESLT is small, simply executing DEC SP NRESLT times will be faster and shorter. The same approaches can be used to provide stack locations for temporary storage.

Example

Assume that subroutine SUBR requires an 8-bit parameter and a 16-bit parameter, and that it produces two 8-bit results. Show a call of SUBR, the placing of the parameters in the accumulator and register pair HL, and the cleaning of the stack after the return. Figure 1-9 shows the appearance of the stack initially, after the subroutine call, and at the end. Using the stack for parameters and results will generally keep the parameters at the top of the stack in the proper order. In this case, there is no need to save the parameters or assign space in the stack for the results (they will replace some or all of the original parameters). However, space must be assigned on the stack for temporary storage to maintain generality and reentrancy.

Calling program

```
        LD   HL,-2      ;LEAVE ROOM ON STACK FOR RESULT
        ADD  HL,SP      ;A GENERAL WAY TO ADJUST SP
        LD   SP,HL
        LD   HL,(PAR16);OBTAIN 16-BIT PARAMETER
        PUSH HL         ;MOVE 16-BIT PARAMETER TO STACK
        LD   A,(PAR8)   ;OBTAIN 8-BIT PARAMETER
        PUSH AF         ;MOVE 8-BIT PARAMETER TO STACK
        INC  SP         ;REMOVE EXTRANEOUS BYTE
        CALL SUBR       ;EXECUTE SUBROUTINE
        LD   HL,3       ;CLEAN PARAMETERS FROM STACK
        ADD  HL,SP
        LD   SP,HL      ;RESULT IS NOW AT TOP OF STACK
```

Subroutine

```
SUBR:   LD   HL,2       ;POINT TO START OF PARAMETER AREA
        ADD  HL,SP
        LD   A,(HL)     ;GET 8-BIT PARAMETER
        INC  HL
        LD   E,(HL)     ;GET 16-BIT PARAMETER
        INC  HL
        LD   D,(HL)
        INC  HL
        EX   DE,HL
        .
        . . . remainder of subroutine . . .
        .
        RET
```

The first three instructions of the calling program could be replaced with two DEC SP instructions, and the last three instructions with three INC SP instructions. Note that only 16-bit register pairs can be moved to or from the stack. Remember, AF consists of the accumulator (MSB) and the flags (LSB).

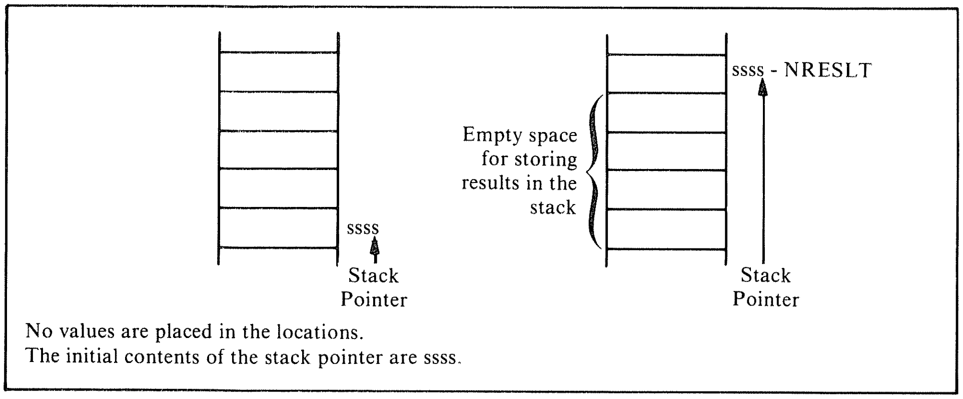

ssss
Stack Pointer
Empty space for storing results in the stack
ssss - NRESLT
Stack Pointer
No values are placed in the locations.
The initial contents of the stack pointer are ssss.

Figure 1-8. The stack before and after assigning NRESLT empty locations for results

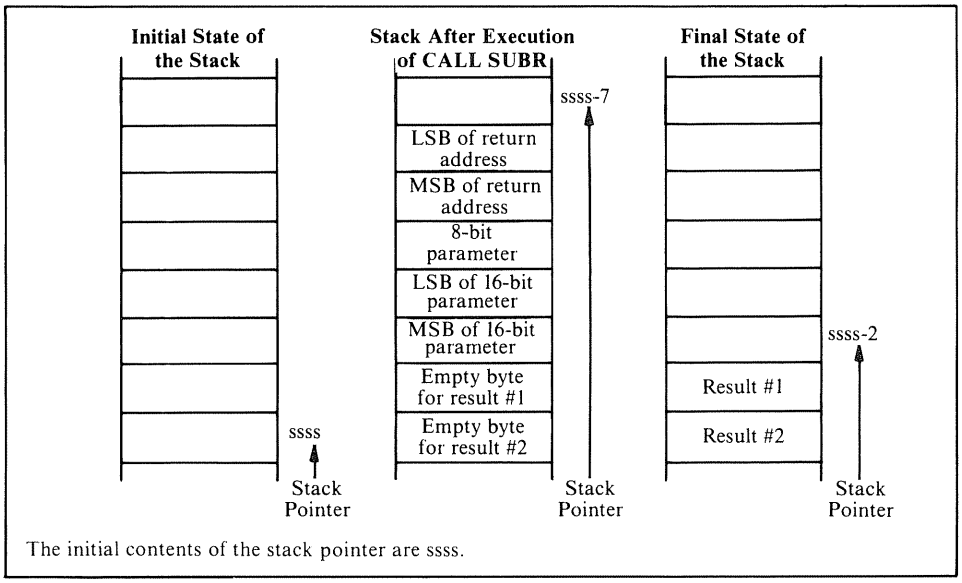


Figure 1-9. The effect of a subroutine on the stack

SIMPLE INPUT/OUTPUT

Simple input/output can be performed using either 8-bit device (port) addresses or full 16-bit memory addresses. The advantages of device addresses are that they are short and provide a separate address space for I/O ports. The disadvantages are that only a few instructions (IN, OUT, and block I/O instructions) use device addresses. If, on the other hand, I/O devices occupy memory addresses, any instruction that references memory can also perform I/O. The problems with this approach are that it is non-standard, it makes it difficult for a reader to differentiate I/O transfers from memory transfers, and it requires that some memory address space be reserved for I/O devices.

Examples

1. Load the accumulator from input port 2.

```
        IN    A,(2)       ;READ FROM PORT 2
```

or

```
        LD    C,2         ;PUT PORT ADDRESS IN C
        IN    A,(C)       ;READ FROM PORT 2
```

The second alternative is longer but more flexible. The IN reg,(C) instruction allows the data to be obtained from any port and loaded into any register. On the other hand, IN A,(port) is limited to loading the accumulator from a fixed port address. The Sign and Zero flags can be set by IN reg,(C) for later testing, whereas IN A,(port) does not affect the flags.

2. Load the accumulator from the input port addressed by the contents of memory location IPORT.

```
        LD      A,(IPORT)  ;GET DEVICE (PORT) ADDRESS
        LD      C,A
        IN      A,(C)      ;READ DATA FROM INPUT PORT
```

The port address can be readily changed (by changing RAM location IPORT) to accommodate multiple input devices attached to a single CPU or to handle different device addresses used in different models, configurations, or computers.

3. Load the accumulator from the input port assigned to the memory address in HL.

```
        LD      A,(HL)     ;READ DATA FROM INPUT PORT
```

Here the same input routine can obtain data from any memory address. Of course, that memory address is no longer available for normal use, thus reducing the actual memory capacity of the computer.

4. Store the accumulator in output port 6.

```
        OUT     (6),A      ;WRITE DATA TO PORT 6
```

or

```
        LD      C,6        ;ACCESS PORT 6
        OUT     (C),A      ;WRITE DATA TO PORT 6
```

In the second alternative, the indirect port address can be changed easily to accommodate a different set of I/O ports or variable I/O devices.

5. Store the accumulator in the output port addressed by the contents of memory location OPORT.

```
        LD      HL,OPORT   ;OBTAIN PORT ADDRESS
        LD      C,(HL)
        OUT     (C),A      ;SEND DATA TO OUTPUT PORT
```

Here the port address is a variable.

6. Store the accumulator in the output port assigned to the memory address in HL.

```
        LD      (HL),A     ;SEND DATA TO OUTPUT PORT
```

Here the same output routine can send data to any memory address.

7. Set the Zero flag if bit 5 of port $D4_{16}$ is 0.

```
        IN   A,(0D4H)           ;READ DATA FROM PORT D4
        BIT  5,A                ;TEST BIT 5
```

If the bit position to be tested is 0, 6, or 7, a shift or AND A instruction can be used to test it.

8. Load the Carry flag from bit 7 of the input port assigned to memory address $33A5_{16}$.

```
        LD   A,(33A5H)          ;OBTAIN DATA
        RLA                     ;MOVE SIGN BIT TO CARRY
```

or

```
        LD   HL,(33A5H)
        RL   (HL)               ;MOVE SIGN BIT OF INPUT DATA TO CARRY
```

RL (HL) could have unpredictable side effects, since it will attempt to store its result back in the input port. Although the port is addressed as a memory location, it may not be writable (that is, it might act like a ROM location). For example, it could be attached to a set of switches that the microprocessor obviously cannot change.

9. Set bit 5 of output port $A5_{16}$.

```
        LD   A,00100000B         ;SET BIT 5 TO 1
        OUT  (0A5H),A            ;MOVE THE BIT TO PORT A5
```

To leave the other bits of port $A5_{16}$ unchanged, a copy of the data in RAM is needed. Then the following sequence will set bit 5 to 1.

```
        LD   A,(COPY)           ;GET COPY OF DATA
        SET  5,A                ;SET BIT 5
        OUT  (0A5H),A           ;UPDATE OUTPUT DATA
        LD   (COPY),A           ;UPDATE COPY OF DATA
```

Note that the CPU cannot generally read an output port, and the input port with the same device address is not necessarily the same physical location.

10. Clear bit 3 of the output port assigned to memory address $B070_{16}$.

```
        LD   HL, 0B070H
        RES  3,(HL)             ;CLEAR BIT 3
```

Even though the output port is addressed as a memory location, it may not be readable. If it is not, the overall effect of RES 3,(HL) will be uncertain; the instruction will surely clear bit 3, but it will assign the other bits of the port the values supposedly obtained by reading from them. These values are generally arbitrary unless the port is latched and buffered. Saving a copy of the data in RAM location TEMP removes the uncertainty. Now bit 3 can be cleared with the sequence

```
        LD    HL,TEMP
        RES   3,(HL)              ;SET BIT 3 OF COPY
        LD    DE,B070H
        LDI                       ;SET BIT 3 OF OUTPUT DATA ALSO
```

Block Input and Output Instructions

The Z80 has special instructions that combine input or output with counting and updating of a memory pointer. These so-called block I/O instructions work much like the block move and block compare instructions discussed earlier. All block I/O instructions move data either from memory to an output port or from an input port to memory (without involving the accumulator), update (either increment or decrement) the memory pointer in register pair HL, and decrement the counter in register B. Note that block I/O instructions use an 8-bit byte counter in register B, whereas block move and block compare instructions use a 16-bit counter in BC. In block I/O instructions, register C always contains the device address. The only meaningful flag is the Zero flag; it is set to 1 if the instruction decrements B to 0, and to 0 otherwise.

Repeated block I/O instructions continue transferring data, updating HL, and decrementing B until B is decremented to 0. The drawback here is that continuous data transfers make sense only if the I/O device operates at the same speed as the processor. Obviously, most I/O devices operate much more slowly than the processor, and the programmer must introduce a delay between transfers. For example, the processor cannot transfer a block of data to or from a keyboard, printer, video display, or magnetic tape unit without waiting between characters. Thus, repeated block I/O instructions are useful only to transfer data to devices that operate at processor speed, such as a buffer memory or a peripheral chip.

The Z80's block I/O instructions are the following:

- INI (IND) moves a byte of data from the port address in C to the memory address in HL, increments (decrements) HL, and decrements B.
- INIR (INDR) repeats INI (IND) until B is decremented to 0.
- OUTI (OUTD) moves a byte of data from the memory address in HL to the port address in C, increments (decrements) HL, and decrements B.
- OTIR (OTDR) repeats OUTI (OUTD) until B is decremented to 0.

Note that block I/O instructions reserve B, C, and HL, but not DE. These instructions also change all the flags except Carry, although only the Zero flag is meaningful.

Examples

1. Move a byte of data from memory address ADDR to output port OPORT.

```
        LD   B,1             ;NUMBER OF BYTES = 1
        LD   C,OPORT         ;PORT ADDRESS = OPORT
        LD   HL,ADDR         ;INITIALIZE MEMORY POINTER
        OUTI                 ;MOVE A BYTE OF DATA
```

Obviously, the overhead of loading the registers makes it uneconomical to use OUTI to send a single byte of data.

2. Move two bytes of data from input port IPORT to memory addresses ADDR and ADDR+1. Use subroutine DELAY to wait before each transfer; assume that DELAY provides the proper time interval without affecting any registers.

```
        LD   B,2             ;NUMBER OF BYTES = 2
        LD   C,IPORT         ;PORT ADDRESS = IPORT
        LD   HL,ADDR         ;INITIALIZE MEMORY POINTER
INBYT:  CALL DELAY           ;WAIT BEFORE EACH INPUT BYTE
        INI                  ;READ A BYTE AND UPDATE
        JR   NZ,INBYT
```

The Zero flag indicates whether the counter in B has been decremented to 0. Not only does INI transfer the data directly into memory, but it also increments HL and decrements B.

3. Move ten bytes of data from memory addresses starting with ADDR to output port OPORT. Use subroutine DELAY to wait between bytes.

```
        LD   B,10            ;NUMBER OF BYTES = 10
        LD   C,OPORT         ;PORT ADDRESS = OPORT
        LD   HL,ADDR         ;INITIALIZE MEMORY POINTER
OUTBYT: OUTI                 ;WRITE A BYTE AND UPDATE
        CALL DELAY           ;WAIT BETWEEN BYTES
        JR   NZ,OUTBYT
```

We cannot use the repeated block output instruction OTIR, since it does not allow a delay between bytes.

4. Move 30 bytes of data from an input buffer addressed through input port IPORT to memory addresses starting with ADDR. Assume that the processor can read successive bytes of data from the buffer without waiting.

```
        LD   B,30            ;NUMBER OF BYTES = 30
        LD   C,IPORT         ;PORT ADDRESS = IPORT
        LD   HL,ADDR         ;INITIALIZE MEMORY POINTER
        INIR                 ;READ A BLOCK OF DATA
```

This sequence does not allow any programmed delay between input operations, so it makes sense only if the input device operates at the same speed as the processor.

LOGICAL AND PHYSICAL DEVICES

One way to allow references to I/O devices by number is to use an I/O device table. An I/O device table assigns the actual I/O addresses (*physical devices*) to the device numbers (*logical devices*) to which a program refers. A systems program then uses the table to convert the device numbers into actual I/O addresses.

The same applications program can be made to utilize different I/O devices by making the appropriate changes in the I/O device table. A program written in a high-level language may, for example, refer to input device #2 and output device #5. For testing purposes, an operator may assign devices #2 and #5 to be the input and output ports, respectively, of his or her console. For normal stand-alone operation, the operator may assign device #2 to be an analog input unit and device #5 the system printer. For operation by remote control, the operator may assign devices #2 and #5 to be communications units used for input and output.

This distinction between logical and physical devices can be implemented by using the instructions IN reg,(C) and OUT (C),reg. If a device table starting in address IOTBL and consisting of 8-bit device addresses is used, input and output are generalized as follows:

- Load the accumulator from a fixed device number DNUM.

```
        LD      A,(IOTBL+DNUM) ;GET DEVICE ADDRESS
        LD      C,A
        IN      A,(C)          ;OBTAIN DATA FROM DEVICE
```

- Load the accumulator from the device number in memory location DEVNO.

```
        LD      A,(DEVNO)       ;GET DEVICE NUMBER
        LD      L,A             ;MAKE DEVICE NUMBER INTO INDEX
        LD      H,0
        LD      DE,IOTBL        ;GET BASE ADDRESS OF DEVICE TABLE
        ADD     HL,DE           ;ACCESS ACTUAL DEVICE ADDRESS
        LD      C,(HL)          ;OBTAIN DEVICE ADDRESS
        IN      A,(C)           ;OBTAIN DATA FROM DEVICE
```

- Store the accumulator in a fixed device number DNUM.

```
        LD      HL,IOTBL+DNUM  ;GET DEVICE ADDRESS
        LD      C,(HL)
        OUT     (C),A          ;SEND DATA TO DEVICE
```

- Store the accumulator in the device number in memory location DEVNO.

```
        LD      B,A             ;SAVE OUTPUT DATA
        LD      A,(DEVNO)       ;GET DEVICE NUMBER
        LD      L,A             ;MAKE DEVICE NUMBER INTO INDEX
        LD      H,0
        LD      DE,IOTBL        ;GET BASE ADDRESS OF DEVICE TABLE
        ADD     HL,DE           ;ACCESS ACTUAL DEVICE ADDRESS
        LD      C,(HL)          ;OBTAIN DEVICE ADDRESS
        OUT     (C),B           ;SEND DATA TO DEVICE
```

In real applications (see Chapter 10), the device table generally contains the starting addresses of I/O subroutines (*drivers*) rather than actual device addresses.

STATUS AND CONTROL

Status and control signals can be handled like any other data. The only special problem is that the processor cannot ordinarily read output ports. To know the current contents of an output port, retain a copy in RAM of the data stored there.

Examples

1. Branch to address DEST if bit 3 of input port 6 is 1.

```
IN    A,(6)              ;READ STATUS FROM PORT 6
BIT   3,A                ;TEST BIT 3
JR    NZ,DEST            ;BRANCH IF BIT 3 IS 1
```

2. Branch to address DEST if bits 4, 5, and 6 of input port STAT are 5 (101 binary).

```
IN    A,(STAT)           ;READ STATUS
AND   01110000B          ;MASK OFF BITS 4,5,AND 6
CP    01010000B          ;IS STATUS FIELD = 5?
JR    Z,DEST             ;YES, BRANCH TO DEST
```

3. Set bit 5 of output port CNTL to 1. Assume that a copy of the data is in a table starting at address OUTP.

```
LD    HL,OUTP+CNTL       ;GET COPY OF DATA
LD    A,(HL)
OR    00100000B          ;SET BIT 5 OF PORT
OUT   (CNTL),A           ;SEND DATA TO OUTPUT PORT
LD    (HL),A             ;UPDATE COPY OF DATA
```

Update the copy every time the data is changed.

4. Set bits 2, 3, and 4 of output port CNTL to 6 (110 binary). Assume that a copy of the data is in a table starting at address OUTP.

```
LD    HL,OUTP+CNTL       ;GET COPY OF DATA
LD    A,(HL)
AND   11100011B          ;CLEAR BITS 2,3, AND 4
OR    00011000B          ;SET CONTROL FIELD TO 6
OUT   (CNTL),A           ;SEND DATA TO OUTPUT PORT
LD    (HL),A             ;UPDATE COPY OF DATA
```

Retaining copies of the data in memory (or using the values stored in a latched, buffered output port) allows changing part of the data without affecting other parts that may have unrelated meanings. For example, changing the state of one indicator light (such as a light that indicated remote operation) will not affect other indicator lights attached to the same port. Similarly, changing one control line (for example, a line that determined whether an object was moving in the positive or negative X-direction) would not affect other control lines attached to the same port.

5. Branch to address DEST if bit 7 of input port IPORT is 0.

```
LD    C,IPORT          ;ESTABLISH PORT ADDRESS
IN    A,(C)            ;READ DATA FROM PORT
JP    Z,DEST           ;BRANCH IF INPUT BIT 7 IS 0
```

The instruction IN reg,(C) affects the Sign and Zero flags, whereas IN A,(port) does not.

PERIPHERAL CHIPS

The most common peripheral chips in Z80-based computers are the PIO (Parallel Input/Output device), SIO (Serial Input/Output device), and CTC (Clock/Timer Circuit). All these devices can perform many functions, much as the microprocessor itself can. Of course, peripheral chips perform fewer different functions than processors, and the range of functions is much more limited. The idea behind programmable peripheral chips is that each chip contains many useful circuits; the designer selects the one he or she wants to use by storing arbitrary codes in control registers, much like selecting circuits from a designer's casebook by specifying arbitrary page numbers or other designations. The advantages of programmable chips are that a single board containing such devices can handle many applications, and changes or corrections can be made by changing selection codes rather than by redesigning circuit boards. The disadvantages of programmable chips are the lack of standards and the difficulty of learning and explaining how specific chips operate.

Chapter 10 contains typical initialization routines for the PIO, SIO, and CTC devices. (The PIO and CTC are discussed in detail in the *Osborne 4 & 8-Bit Microprocessor Handbook.*[12]) We will provide only a brief overview of the PIO device here, since it is the most widely used. Bas and Kaynak describe a typical industrial application using a PIO.[13]

PIO (Parallel Input/Output Device) General Description

The PIO contains two 8-bit ports, A and B. Each port contains

- An 8-bit output register.
- An 8-bit input register.

- A 2-bit mode control register, which indicates whether the port is in an output, input, bidirectional, or control mode.
- An 8-bit input/output control register, which determines whether the corresponding data pins are inputs (1) or outputs (0) in the control mode.
- Two control lines (STB and RDY) that can be used for handshaking signals (the contents of the mode control register determine how these lines operate).
- An interrupt enable bit.
- A 2-bit mask control register (used only in the control mode) that determines the active polarity of the inputs and whether they will be logically ANDed or ORed to form an interrupt signal.
- An 8-bit mask register (used only in the control mode) that determines which port lines will be monitored to form the interrupt signal.
- An 8-bit vector address register used with the interrupt system.

Here, the important points are the input and output registers, the mode control register, the input/output control register, and the control lines. The interrupt-related features of the PIO are discussed in *Z80 Assembly Language Programming.*[14]

The meanings of the bits in the various control and mask registers are related to the underlying hardware and are entirely arbitrary as far as the programmer is concerned. Tables are provided here and in Appendix B for looking them up.

Each PIO occupies four input port addresses and four output port addresses. The B/A SEL (Port B or A select) and C/D SEL (Control or Data select) lines choose one of the four ports as described in Table 1-10. Most often, designers attach address line A_0 to B/A SEL and A_1 to C/D SEL. The PIO then occupies the four consecutive port addresses given in the last column of Table 1-10.

Clearly, there are far more internal control registers than there are port addresses available. In fact, all the control registers for each port occupy one address determined

Table 1-10. PIO Addresses

Control or Data select	Port B or A Select	Register Addressed	Port Address (Starting with PIOADD)
0	0	Data Register A	PIOADD
0	1	Data Register B	PIOADD+1
1	0	Control A	PIOADD+2
1	1	Control B	PIOADD+3

The port addresses assume that C/D SEL is tied to A_1 and B/A SEL to A_0.

Table 1-11. Addressing of PIO Control Registers

Register	Addressing
Mode Control	$D_3 = D_2 = D_1 = D_0 = 1$
Input/Output Control	Next byte after port placed in mode 3
Mask Control Register	$D_3 = 0, D_2 = D_1 = D_0 = 1$
Interrupt Mask Register	Next byte after mask control register accessed with $D_4 = 1$
Interrupt Enable	$D_3 = D_2 = 0, D_1 = D_0 = 1$
Interrupt Vector	$D_0 = 1$

by the C/D SEL connection. Thus, some of the data bits sent to a control register are actually used for addressing. Note the following situations (see Table 1-11):

· If $D_0 = 0$, the remaining data bits are loaded into the interrupt vector register.

· If $D_3 = 0$ and $D_2 = D_1 = D_0 = 1$, the remaining data bits are loaded into the mask control register. If $D_4 = 1$, the next control byte is loaded into the interrupt mask register. Interrupts can be enabled ($D_7 = 1$) or disabled ($D_7 = 0$) with $D_3 = D_2 = 0$, $D_1 = D_0 = 1$.

· If D_3, D_2, D_1, and D_0 are all 1's, the remaining data bits are loaded into the mode control register. If $D_7 = D_6 = 1$ (that is, the port has been placed in the control mode), the next control byte is loaded into the input/output control register.

This sharing of an external address means

· The programmer must be careful to specify the proper addresses, data values, and order of operations. The actual destination of an OUT instruction directed to a PIO control address depends on the data value and may also depend on the OUT instruction that preceded it.

· The programmer should document the PIO initialization in detail. The device is complex, and a reader cannot be expected to understand the initializing sequence.

The control registers of the PIO are usually initialized only in an overall startup routine. Other routines typically refer only to the PIO input and output registers. Since all of its control registers share a port address, a repeated block output instruction (OTIR or OTDR) can be used to initialize a PIO. No timing problem occurs, since the PIO operates at the same speed as the CPU. Chapter 10 contains an example showing the use of repeated block output instructions to initialize PIOs and other peripheral chips.

PIO Operating Modes

A startup program selects the operating mode of a PIO port by writing a control byte to the PIO in the form shown in Figure 1-10. The lower table in Figure 1-10 describes the operating modes and their associated control bytes. Note that only bits 6 (M_0) and 7 (M_1) affect the operating mode; bits 4 and 5 are not used and bits 0 through 3 are used for addressing. When power is turned on, the PIO comes up in mode 1 (input). The modes may be summarized as follows:

- **Mode 0 — Output** (bit 7 = bit 6 = 0)

Writing data into the port's output register latches the data and causes it to appear on the port's data bus. The Ready (RDY) line goes high to indicate Data Ready; it remains high until the peripheral sends a rising edge (a 0-to-1 or low-to-high transition) on the Strobe (STB) line to indicate Data Accepted or Device Ready. The rising edge of STB causes an interrupt if the interrupt is enabled.

- **Mode 1 — Input** (bit 7 = 0, bit 6 = 1)

The peripheral latches data into the port's input register using the Strobe signal. The rising edge of STB causes an interrupt (if enabled) and deactivates RDY (makes it 0). When the CPU reads the data, RDY goes high to indicate Data Accepted or Input Register Empty. Note that the peripheral can strobe data into the register regardless of the state of RDY. The programmer is therefore responsible for guarding against overrun (new data being placed in the register before the CPU has read the old data).

Set Mode

M1	**M0**	**Mode**
0	0	Output
0	1	Input
1	0	Bidirectional
1	1	Bit Control

M1	M0	X	X	1	1	1	1

If a port is placed in mode 3. the next byte sets the I/O control register:

I/O7	I/O6	I/O5	I/O4	I/O3	I/O2	I/O1	I/O0

I/O = 1 Sets bit to Input
I/O = 0 Sets bit to Output

PIO Mode	**Meaning**	**Control Byte (Binary)**	**Control Byte (Hex)**
0	Output	00001111	0F
1	Input	01001111	4F
2	Bidirectional	10001111	8F
3	Control	11001111	CF

Note that bits 4 and 5 are not used and could have any values.

Figure 1-10. Mode control for the Z80 PIO

· **Mode 2 — Bidirectional** (bit 7 = 1, bit 6 = 0)

Since this mode uses all four handshake lines, it is allowed only on port A. The port A RDY and STB signals are used for output control and the port B RDY and STB signals are used for input control. The only difference between this mode and a combination of modes 0 and 1 is that data from the port A Output register is enabled onto the port's data bus only when A STB is active. This allows the port A bus to be used bidirectionally under the control of A STB (Output Data Request) and B STB (Input Data Available). Note that operations on input register A govern port B's control signals in this mode.

· **Mode 3 — Control** (bit 7 = 1, bit 6 = 1)

This mode does not use the RDY and STB signals. It is intended for status and control applications in which each bit has an individual meaning. When mode 3 is selected, the next control byte sent to the PIO defines the directions of the port's bus lines. A 1 in a bit position makes the corresponding bus line an input, whereas a 0 makes it an output.

Note the following features of the PIO's operating modes:

· In modes 0, 1, and 2, the peripheral indicates Data Ready, Device Ready, or Data Accepted with a rising edge on the STB line. This edge also causes an interrupt if the interrupt is enabled.

· In modes 0, 1, and 2, the PIO indicates Data Ready, Input Buffer Empty, or Data Accepted by sending RDY high. This signal remains high until the next rising edge on STB.

· The bidirectional mode (mode 2) applies only to port A, and port B must be placed in mode 3 (control) since all the handshaking lines are already committed.

· The input/output control register is used only in the control mode (mode 3). Otherwise, the entire 8-bit port is used for either input or output.

· There is no way for the processor to determine if a pulse has occurred on STB if interrupts are not being used. The PIO is designed for use in interrupt-driven systems rather than in programmed I/O systems. STB should be tied low if it is not being used.

· The processor cannot control the RDY lines directly. The RDY line on a port goes high when data is transferred to or from the port and goes low on the rising edge of STB.

· The contents of the output register can be read if the port is in the output or bidirectional mode. If the port is in the control mode, the output register data from the lines assigned as outputs can be read. The contents of control registers cannot be read. If a program needs to know their contents, it must save copies in RAM of the values stored there.

· If the RDY output is tied to the STB input on a port in the output mode, RDY will go high for one clock period after each output operation. This brief pulse can be used to multiplex displays.

PIO Initialization

When power is turned on, the PIO comes up in the input mode with all interrupts disabled and inhibited and control signals deactivated (low). The steps in initializing a PIO port are

- Select the operating mode by writing the appropriate control byte into the mode control register. Interrupt control as well as I/O mode information may have to be sent.
- If in mode 3, establish the directions of the I/O pins by writing a control byte into the input/output control register. This byte must follow the control byte that selected mode 3.

Examples

1. Make port B output.

```
LD    A,00001111B     ;MAKE PORT B OUTPUT
OUT   (PIOCRB),A
```

Bits 0 through 3 of the control byte are all 1's to address the mode control register. Bits 6 and 7 are both 0's to put the port in the output mode. Bits 4 and 5 are not used.

2. Make port A input.

```
LD    A,01001111B     ;MAKE PORT A INPUT
OUT   (PIOCRA),A
```

Bit 7 = 0 and bit 6 = 1 to put the port in the input mode.

3. Make port A bidirectional.

```
LD    A,10001111B     ;MAKE PORT A BIDIRECTIONAL
OUT   (PIOCRA),A
```

Bit 7 = 1 and bit 6 = 0 to put the port in the bidirectional mode. Remember that only port A can be operated in the bidirectional mode, and that port B must then be operated in the control mode.

4. Make port A control with all lines inputs.

```
LD    A,11001111B     ;MAKE PORT A CONTROL
OUT   (PIOCRA),A
LD    A,11111111B     ;ALL BITS INPUTS
OUT   (PIOCRA),A
```

The first OUT instruction puts port A in the control mode, since bits 6 and 7 are both 1. The second OUT operation to the same address loads a different register (the input/output control register). A 0 in a bit position of that register makes the corresponding pin an output, while a 1 makes it an input. The polarity here is arbitrary, and many bidirectional devices use the opposite convention.

5. Make port B control with all lines outputs.

```
        LD    A,11001111B      ;MAKE PORT B CONTROL
        OUT   (PIOCRB),A
        SUB   A                ;ALL BITS OUTPUTS
        OUT   (PIOCRB),A
```

The second byte is directed automatically to the input/output control register if the first byte puts the port in the control mode.

6. Make port A control with lines 1, 5, and 6 inputs and lines 0, 2, 3, 4, and 7 outputs.

```
        LD    A,11001111B      ;MAKE PORT A CONTROL
        OUT   (PIOCRA),A
        LD    A,01100010B      ;1,5,6 IN--0,2,3,4,7 OUT
```

INTERRUPT SERVICE ROUTINES

More information on material in this section can be found in the book *Practical Microcomputer Programming: The Z80* by W.J. Weller, Chapter 16.

Z80 interrupt systems may operate in any of three modes.[15] In all three modes, the processor responds to an interrupt by executing a CALL or RST instruction which transfers control to a specific memory address and saves the current program counter at the top of the stack. Table 1-12 lists the destination addresses for the RST instructions and the non-maskable interrupt. No other registers (besides the program counter) are saved automatically.

There are two common approaches to saving registers:

· If there is only a single level of interrupts, primary registers may be saved in the alternate set. The service routine begins with

```
        EX    AF,AF'           ;SAVE PRIMARY REGISTERS IN ALTERNATES
        EXX
```

The EXX instruction exchanges registers B, C, D, E, H, and L with their primed equivalents. The service routine must end by restoring the original primary registers with

```
        EXX                    ;RESTORE ORIGINAL PRIMARY REGISTERS
        EX    AF,AF'
```

This approach assumes that the alternate (primed) registers are reserved for use in interrupt service routines.

Table 1-12. Destination Addresses for RST (Restart) Instructions and the Non-Maskable Interrupt

RST Instruction (Mnemonic)	Operation Code (Hex)	Destination Address	
		(Hex)	(Decimal)
RST 0	C7	0000	0
RST 8	CF	0008	08
RST 10H	D7	0010	16
RST 18H	DF	0018	24
RST 20H	E7	0020	32
RST 28H	EF	0028	40
RST 30H	F7	0030	48
RST 38H	FF	0038	56
Non-maskable interrupt		0066	102

· If there are several levels of interrupts, each service routine must save all registers that it uses in the stack. Since the Z80 has so many registers, most programmers keep their service routines simple so that they must save only a few registers. Otherwise, the overhead involved in servicing interrupts (sometimes called the *interrupt latency*) becomes excessive. A typical sequence for saving the primary registers in the stack is

```
PUSH AF         ;SAVE REGISTERS
PUSH BC
PUSH DE
PUSH HL
```

The opposite sequence restores the primary registers.

```
POP  HL         ;RESTORE REGISTERS
POP  DE
POP  BC
POP  AF
```

Interrupts must be reenabled explicitly with EI immediately before the RET instruction that terminates the service routine. The EI instruction delays the actual enabling of interrupts for one instruction cycle to avoid unnecessary stacking of return addresses (that is, an RET instruction can remove the return address from the stack before a pending interrupt is recognized).

You must be careful to save any write-only registers that may have to be restored at the end of the routine. For example, the PIO's control registers are all write-only, and many external priority registers are also write-only. Copies of such registers must be saved in RAM and restored from the stack. A typical example is

```
PUSH AF         ;SAVE REGISTERS
PUSH BC
PUSH DE
PUSH HL
LD   A,(PRTY)   ;SAVE OLD PRIORITY
PUSH AF
LD   A,NPRTY    ;GET NEW PRIORITY
OUT  PPORT      ;PLACE IT IN EXTERNAL PRIORITY REGISTER
LD   (PRTY),A   ;SAVE COPY OF NEW PRIORITY IN RAM
```

The restoration procedure must recover the previous priority as well as the original contents of the registers.

```
POP  AF         ;RESTORE OLD PRIORITY
OUT  PPORT      ;PLACE IT IN EXTERNAL PRIORITY REGISTER
LD   (PRTY),A   ;SAVE COPY OF PRIORITY IN RAM
POP  HL         ;RESTORE REGISTERS
POP  DE
POP  BC
POP  AF
```

To achieve general reentrancy, the stack must be used for all temporary storage beyond that provided by the registers. As noted in the discussion of parameter passing, space is assigned on the stack (NPARAM bytes) with the sequence

```
LD   HL,-NPARAM      ;ASSIGN NPARAM EMPTY BYTES
ADD  HL,SP
LD   SP,HL
```

Later, of course, the temporary storage area is discarded with the sequence

```
LD   HL,NPARAM       ;REMOVE NPARAM BYTES FROM STACK
ADD  HL,SP
LD   SP,HL
```

If NPARAM is small, save execution time and memory by replacing these sequences with NPARAM DEC SP or INC SP instructions. Chapter 11 contains examples of simple interrupt service routines.

Interrupt service routines that are based on signals from Z80 peripheral chips (PIOs, SIOs, or CTCs) or that utilize the non-maskable input require special terminating instructions. These special instructions restore the program counter from the top of the stack just like the normal RET. The RETI (return from interrupt) instruction also signals the peripheral chips that the service routine has been completed, thus unblocking lower priority interrupts. The RETN (return from non-maskable interrupt) instruction also restores the interrupt enable logic, thus reenabling interrupts if (and only if) they were enabled when the non-maskable interrupt occurred.

MAKING PROGRAMS RUN FASTER

More information on material in this section can be found in an article by T. Dollhoff, "Microprocessor Software: How to Optimize Timing and Memory Usage. Part Four: Techniques for the Zilog Z80," *Digital Design,* February 1977, pp. 44-45.

In general, programs can be made to run substantially faster only by first determining where they spend their time. This requires determining which loops (other than delay routines) the processor is executing most often. Reducing the execution time of a frequently executed loop will have a major effect because of the multiplying factor. It is thus critical to determine how often instructions are being executed and to work on loops in the order of their frequency of execution.

Once it is determined which loops the processor executes most frequently, reduce their execution time wlth the following techniques:

- Eliminate redundant operations. These may include a constant that is being added during each iteration or a special case that is being tested repeatedly. Another example is a constant value or a memory address that is being fetched from memory each time rather than being stored in a register or register pair.

- Reorganize the loop to reduce the number of jump instructions. You can often eliminate branches by changing the initial conditions, inverting the order of operations, or combining operations. In particular, you may find it helpful to initialize everything one step back, thus making the first iteration the same as all the others. Inverting the order of operations can be helpful if numerical comparisons are involved, since the equality case may not have to be handled separately. Reorganization may also combine condition checking inside the loop with the overall loop control.

- Use in-line code rather than subroutines. This will save at least a CALL and RET.

- Use the stack rather than specific memory addresses for temporary storage. Remember that EX HL,(SP) exchanges the top of the stack with register pair HL and thus can serve to both restore an old value and save the current one.

- Assign registers to take maximum advantage of such specialized instructions as LD HL,(ADDR); LD (ADDR),HL; EX DE, HL; EX HL,(SP); DJNZ; and the block move, compare, and I/O instructions. Thus it is preferable to always use B or BC for a counter, HL for an indirect address, and DE for another indirect address if needed.

- Use the block move, block compare, and block I/O instructions to handle blocks of data. These instructions can replace an entire program sequence, since they combine counting and updating of pointers with the actual data manipulation or transfer operations. Note, in particular, that the block move and block I/O instructions transfer data to or from memory without using the accumulator.

- Use the 16-bit instructions whenever possible to manipulate 16-bit data. These instructions are ADC, ADD, DEC, EX, INC, LD, POP, PUSH, and SBC.

- Use instructions that operate directly on data in user registers or in memory to avoid having to save and restore the accumulator, HL, or an index register. These instructions include DEC, EX, INC, LD, POP, PUSH, and the bit manipulation and shift instructions.

- Minimize the use of the index registers, since they always require extra execution time and memory. The index registers are generally used only as backups to HL and in handling data structures that involve many fixed offsets.

- Minimize the use of special Z80 instructions that require a 2-byte operation code. These always require extra execution time and memory. Examples are BIT, RES, SET, SLA, SRA, and SRL, as well as some load instructions such as LD DE,(ADDR), LD (ADDR),BC, and LD SP,(ADDR).

- Take advantage of specialized short instructions such as the accumulator shifts (RLA, RLCA, RRA, and RRCA) and DJNZ.

- Use absolute jumps (JP) rather than relative jumps (JR). The absolute jumps take less time if a branch actually occurs.

- Organize sequences of conditional jumps to minimize average execution time. Branches that are often taken should come before ones that are seldom taken, for example, checking for a result being negative (true 50% of the time if the value is random) before checking for it to be zero (true less than 1% of the time if the value is random).

- Test for conditions under which a sequence has no effect and branch around it if the conditions hold. This will be profitable if the sequence is long, and it frequently does not change the result. A typical example is the propagation of carries through higher order bytes. If a carry seldom occurs, it will be faster on the average to test for it rather than simply propagate a 0.

A general way to reduce execution time is to replace long sequences of instructions with tables. A single table lookup can perform the same operation as a sequence of instructions if there are no special exits or program logic involved. The cost is extra memory, but that may be justified if the memory is available. If enough memory is available, a lookup table may be a reasonable approach even if many of its entries are repetitive—that is, even if many inputs produce the same output. In addition to its speed, table lookup is also general, easy to program, and easy to change.

MAKING PROGRAMS USE LESS MEMORY

Only by identifying common instruction sequences and replacing those sequences with subroutine calls can a program be made to use significantly less memory. The result is a single copy of each sequence; the cost is the extra execution time of the

CALL and RET instructions. The more instructions placed in subroutines, the more memory is saved. Of course, such subroutines are typically not general and may be difficult to understand or use. Some sequences may even be available in a monitor or other systems program. Then those sequences can be replaced with calls to the systems program as long as the return is handled properly.

Some methods that reduce execution time also reduce memory usage. In particular, eliminating redundant operations, reorganizing loops, using the stack, organizing the use of registers, using the 16-bit registers, using block instructions and short forms, operating directly on memory or registers, and minimizing the use of the index registers and special Z80 instructions reduce both memory usage and execution time. Of course, using in-line code rather than loops and subroutines reduces execution time but increases memory usage. Absolute and relative jumps represent a minor tradeoff between memory and execution time; absolute jumps are faster (if a branch occurs) but use more memory.

Lookup tables generally use extra memory but save execution time. Some ways to reduce their memory requirements are to eliminate intermediate values and interpolate the results, eliminate redundant values with special tests, and reduce the range of input values.[16,17] Often a few prior tests or restrictions will greatly reduce the size of the required table.

REFERENCES

1. Weller, W.J., *Practical Microcomputer Programming: The Z80,* Evanston, Ill.: Northern Technology Books, 1979.

2. Fisher, W.P., "Microprocessor Assembly Language Draft Standard," *IEEE Computer,* December 1979, pp. 96-109. Further discussions of the draft standard appear on pp. 79-80 of *IEEE Computer,* April 1980 and on pp. 8-9 of *IEEE Computer,* May 1981. See also Duncan, F.G., "Level-Independent Notation for Microcomputer Programs," *IEEE Micro,* May 1981, pp. 47-56.

3. Osborne, A. *An Introduction to Microcomputers: Volume 1 — Basic Concepts, 2nd ed.,* Berkeley, Calif.: Osborne/McGraw-Hill, 1980.

4. Fisher, op.cit.

5. Osborne, op. cit.

6. Weller, op.cit., p. 224.

7. Ibid., pp. 19-26.

8. Ibid.

9. Ibid., p. 69.

10. Shankar, K.S., "Data Structures and Abstractions," *IEEE Computer,* April, 1980, pp. 67-77.

11. Tenenbaum, A. and M. Augenstein, *Data Structures Using Pascal,* Englewood Cliffs, N.J.: Prentice-Hall, 1981.

12. Osborne, A. and G. Kane, *4 & 8-Bit Microprocessor Handbook,* Berkeley, Calif.: Osborne/McGraw-Hill, 1981, pp. 7-45 to 7-54 (PIO), pp. 7-54 to 7-62 (CTC).

13. Bas, S. and O. Kaynak, "Microprocessor Controlled Single Phase Cycloconverter," 1981 IECI Proceedings on Industrial Applications of Mini and Microcomputers, pp. 39-44. Available from IEEE, 445 Hoes Lane, Piscataway, N.J. 08854 (catalog no. 81CH1714-5).

14. Leventhal, L., *Z80 Assembly Language Programming,* Berkeley, Calif.: Osborne/McGraw-Hill, 1979, Chapter 12.

15. Ibid.

16. Seim, T.A., "Numerical Interpolation for Microprocessor-Based Systems," *Computer Design,* February 1978, pp. 111-116.

17. Abramovich, A. and T.R. Crawford, "An Interpolating Algorithm for Control Applications on Microprocessors," 1978 IECI Proceedings on Industrial Applications of Microprocessors, pp. 195-201. This Proceedings is available from IEEE, 445 Hoes Lane, Piscataway, N.J. 08854.

Chapter 2 Implementing Additional Instructions and Addressing Modes

This chapter shows how to implement instructions and addressing modes that are not included in the Z80 instruction set. Of course, no instruction set can ever include all possible combinations. Designers must choose a set based on how many operation codes are available, how easily an additional combination could be implemented, and how often it would be used. A description of additional instructions and addressing modes does not imply that the basic instruction set is incomplete or poorly designed.

The chapter will concentrate on additional instructions and addressing modes that are

- Obvious parallels to those included in the instruction set.
- Described in Fischer's "Microprocessor Assembly Language Standard".[1]
- Discussed in Volume 1 of *An Introduction to Microcomputers.*[2]
- Implemented on other microprocessors, especially ones that are closely related or partly compatible.[3]

This chapter should be of particular interest to those who are familiar with the assembly languages of other computers.

INSTRUCTION SET EXTENSIONS

In describing extensions to the instruction set, we follow the organization suggested in the draft standard for IEEE Task P694.[4] Instructions are divided into the following groups (listed in the order in which they are discussed): arithmetic, logical, data transfer, branch, skip, subroutine call, subroutine return, and miscellaneous. For each type of instruction, types of operands are discussed in the following order: byte (8-bit), word (16-bit), decimal, bit, nibble or digit, and multiple. In describing addressing modes, we use the following order: direct, indirect, immediate, indexed, register, autopreincrement, autopostincrement, autopredecrement, autopostdecrement, indirect preindexed (also called preindexed or indexed indirect), and indirect postindexed (also called postindexed or indirect indexed).

ARITHMETIC INSTRUCTIONS

This group includes addition, addition with Carry, subtraction, subtraction in reverse, subtraction with Carry (borrow), increment, decrement, multiplication, division, comparison, two's complement (negate), and extension. Instructions that do not clearly fall into a particular category are repeated for convenience.

Addition Instructions (Without Carry)

1. Add memory location ADDR to accumulator.

```
LD    HL,ADDR     ;POINT TO DATA
ADD   A,(HL)      ;THEN ADD IT
```

2. Add Carry to accumulator.

```
ADC   A,0         ;ACC = ACC + CARRY + 0
```

3. Decimal add Carry to accumulator.

```
ADC   A,0         ;ACC = ACC + CARRY + 0
DAA               ;  IN DECIMAL
```

4. Decimal add VALUE to accumulator.

```
ADD   A,VALUE     ;ACC = ACC + VALUE
DAA               ;  IN DECIMAL
```

5. Decimal add register to accumulator.

```
ADD   A,reg       ;ACC = ACC + REG
DAA               ;  IN DECIMAL
```

6. Add 16-bit number VAL16 to HL.

```
LD    rp,VAL16
ADD   HL,rp       ;HL = HL + VAL16
```

rp can be either BC or DE.

7. Add 16-bit number VAL16 to an index register.

```
LD    rp,VAL16
ADD   xy,rp       ;XY = XY + VAL16
```

rp can be either BC or DE.

8. Add memory locations ADDR and ADDR+1 (MSB in ADDR+1) to HL.

```
        LD    rp,(ADDR)
        ADD   HL,rp
```

The 16-bit data is stored in the usual Z80 format with the less significant byte first (at the lower address).

9. Add memory locations ADDR and ADDR+1 (MSB in ADDR+1) to an index register.

```
        LD    rp,(ADDR)
        ADD   xy,rp
```

10. Add memory locations ADDR and ADDR+1 (MSB in ADDR+1) to memory locations SUM and SUM+1 (MSB in SUM+1).

```
        LD    HL,(SUM)          ;GET CURRENT SUM
        LD    DE,(ADDR)         ;ADD ELEMENT
        ADD   HL,DE
        LD    (SUM),HL          ;SAVE UPDATED SUM
```

11. Add the 16-bit number VAL16 to memory locations ADDR and ADDR+1 (MSB in ADDR+1).

```
        LD    HL,(SUM)          ;GET CURRENT SUM
        LD    DE,VAL16          ;ADD ELEMENT
        ADD   HL,DE
        LD    (SUM),HL          ;SAVE UPDATED SUM
```

Addition Instructions (with Carry)

1. Add memory location ADDR to accumulator with Carry.

```
        LD    HL,ADDR           ;POINT TO DATA
        ADC   A,(HL)            ;THEN ADD IN DATA
```

2. Add Carry to accumulator.

```
        ADC   A,0               ;ACC = ACC + CARRY + 0
```

3. Decimal add VALUE to accumulator with Carry.

```
        ADC   A,VALUE           ;ACC = ACC + VALUE + CARRY
        DAA                     ;  IN DECIMAL
```

4. Decimal add register to accumulator with Carry.

```
        ADC   A,reg             ;ACC = ACC + REG + CARRY
        DAA                     ;  IN DECIMAL
```

5. Add 16-bit number VAL16 to HL with Carry.

```
        LD    rp,VAL16
        ADC   HL,rp              ;HL = HL + VAL16 + CARRY
```

6. Add memory locations ADDR and ADDR+1 (MSB in ADDR+1) to HL with Carry.

```
        LD    rp,(ADDR)
        ADC   HL,rp              ;HL = HL + (ADDR) + CARRY
```

Subtraction Instructions (Without Borrow)

1. Subtract memory location ADDR from accumulator.

```
        LD    HL,ADDR            ;POINT TO DATA
        SUB   (HL)               ;THEN SUBTRACT IT
```

2. Subtract borrow (Carry) from accumulator.

```
        SBC   A,0                ;ACC = ACC - CARRY
```

3. Decimal subtract VALUE from accumulator.

```
        SUB   VALUE              ;ACC = ACC - VALUE
        DAA                      ;  IN DECIMAL
```

4. Decimal subtract register from accumulator.

```
        SUB   reg                ;ACC = ACC - REG
        DAA                      ;  IN DECIMAL
```

Since the Z80 has an Add/Subtract flag, it can perform decimal subtraction directly. On the 8080 and 8085 processors, the programmer must implement decimal subtraction as the addition of a negative number.

5. Subtract register pair from HL.

```
        AND   A                  ;CLEAR CARRY
        SBC   HL,rp              ;SUBTRACT REGISTER PAIR WITH CARRY
```

The Z80 has a subtract register pair with Carry instruction, but no plain subtract register pair (without Carry).

6. Subtract 16-bit number VAL16 from HL.

```
        LD    rp,-VAL16
        ADD   HL,rp
```

or

```
        AND   A                  ;CLEAR CARRY
        LD    rp,VAL16
        SBC   HL,rp              ;SUBTRACT 16-BIT NUMBER FROM HL
```

rp can be either BC or DE. Carry is an inverted borrow in the first alternative and a true borrow in the second. The first alternative is obviously much shorter, particularly since SBC HL requires a 2-byte operation code.

7. Subtract memory locations ADDR and ADDR+1 (MSB in ADDR+1) from HL.

```
AND   A                ;CLEAR CARRY
LD    rp,(ADDR)        ;THEN SUBTRACT WITH CARRY
SBC   HL,rp
```

There is no subtract register pair (without Carry) instruction.

Subtraction in Reverse Instructions

1. Subtract accumulator from VALUE and place difference in accumulator.

```
NEG                    ;NEGATE A
ADD   A,VALUE          ;FORM - A + VALUE
```

or

```
LD    reg,A            ;CALCULATE VALUE - ACC
LD    A,VALUE
SUB   reg
```

The Carry is an inverted borrow in the first method and a true borrow in the second.

2. Subtract accumulator from register and place difference in accumulator.

```
NEG                    ;NEGATE A
ADD   A,reg            ;FORM - A + REG
```

The Carry is an inverted borrow; that is, it is 1 if the subtraction does not require a borrow.

3. Decimal subtract accumulator from VALUE and place difference in accumulator.

```
LD    reg,A            ;CALCULATE VALUE - ACC
LD    A,VALUE
SUB   reg
DAA
```

4. Decimal subtract accumulator from register and place difference in accumulator.

```
LD    reg1,A           ;CALCULATE REG - ACC
LD    A,reg
SUB   reg1
DAA                    ;IN DECIMAL
```

Subtraction with Borrow (Carry) Instructions

1. Subtract memory location ADDR from accumulator with borrow.

```
LD    HL,ADDR             ;POINT TO DATA
SBC   A,(HL)              ;THEN SUBTRACT WITH BORROW
```

2. Subtract borrow (Carry) from accumulator.

```
SBC   A,0                 ;FORM A - BORROW
```

3. Decimal subtract inverted borrow from accumulator (Carry = 1 if no borrow was generated, 0 if a borrow was generated).

```
ADC   A,99H               ;ADD 99 PLUS CARRY
DAA
```

The final Carry is 1 if the subtraction generates a borrow and 0 if it does not.

4. Decimal subtract VALUE from accumulator with borrow.

```
SBC   A,VALUE             ;A = A - VALUE - BORROW
DAA                       ;  IN DECIMAL
```

5. Decimal subtract register from accumulator with borrow.

```
SBC   A,reg               ;A = A - REG - BORROW
DAA                       ;  IN DECIMAL
```

6. Subtract 16-bit number VAL16 from HL with borrow.

```
LD    rp,VAL16
SBC   HL,rp               ;HL = HL - VAL16 - BORROW
```

Increment Instructions

1. Increment memory location ADDR.

```
LD    HL,ADDR
INC   (HL)
```

2. Increment accumulator, setting the Carry flag if the result is 0.

```
ADD   A,1
```

Remember that INC does not affect Carry, but it does affect the Zero flag.

3. Decimal increment accumulator (add 1 to A in decimal).

```
        ADD   A,1
        DAA
```

You cannot use INC, since it does not affect Carry.

4. Decimal increment register (add 1 to reg in decimal).

```
        LD    A,reg
        ADD   A,1
        DAA
        LD    reg,A
```

DAA applies only to the accumulator.

5. Increment memory locations ADDR and ADDR+1 (MSB in ADDR + 1).

```
        LD    HL,(ADDR)
        INC   HL          ;16-BIT INCREMENT
        LD    (ADDR),HL
```

or

```
        LD    HL,ADDR
        INC   (HL)        ;INCREMENT LSB
        JR    NZ,DONE
        INC   HL          ;ADD CARRY TO MSB
        INC   (HL)
        DEC   HL
DONE:   NOP
```

The second alternative leaves ADDR in HL for later use.

6. Increment register pair, setting the Zero flag if the result is 0.

```
        INC   rp          ;16-BIT INCREMENT
        LD    A,rpl       ;TEST RESULT FOR ZERO
        OR    rph
```

This sequence destroys the old contents of the accumulator and the flags. OR clears Carry.

Decrement Instructions

1. Decrement memory location ADDR.

```
        LD    HL,ADDR
        DEC   (HL)
```

2. Decrement accumulator, setting Carry flag if a borrow is generated.

```
SUB   1
```

3. Decrement accumulator, setting Carry flag if no borrow is generated.

```
ADD   A,0FFH
```

4. Decimal decrement accumulator (subtract 1 from A in decimal).

```
SUB   1
DAA
```

DEC cannot be used here, since it does not affect Carry.

5. Decimal decrement register (subtract 1 from reg in decimal).

```
LD    A,reg
SUB   1
DAA
LD    reg,A
```

DAA applies only to the accumulator.

6. Decrement memory locations ADDR and ADDR+1 (MSB in ADDR+1).

```
LD    HL,(ADDR)
DEC   HL                ;16-BIT DECREMENT
LD    (ADDR),HL
```

7. Decrement register pair, setting the Zero flag if the result is 0.

```
DEC   rp                ;16-BIT DECREMENT
LD    A,rpl             ;TEST 16-BIT RESULT FOR ZERO
OR    rph
```

This sequence destroys the old contents of the accumulator and changes the other flags. OR clears the Carry flag.

Multiplication Instructions

1. Multiply accumulator by 2.

```
ADD   A,A
```

2. Multiply accumulator by 3 (using reg for temporary storage).

```
LD    reg,A             ;SAVE A
ADD   A,A               ;2 X A
ADD   A,reg             ;3 X A
```

3. Multiply accumulator by 4.

```
ADD   A,A            ;2 X A
ADD   A,A            ;4 X A
```

We can easily extend cases 1, 2, and 3 to multiplication by other small integers.

4. Multiply register by 2.

```
SLA   reg
```

5. Multiply register by 4.

```
SLA   reg            ;MULTIPLY BY 2
SLA   reg            ;AND THEN BY 2 AGAIN
```

Since SLA is a 2-byte instruction, it eventually becomes faster to move the data to the accumulator and use the 1-byte instruction ADD A, A.

6. Multiply register pair HL by 2.

```
ADD   HL,HL
```

7. Multiply register pair HL by 3 (using rp for temporary storage).

```
LD    rph,H
LD    rpl,L
ADD   HL,HL          ;2 X HL
ADD   HL,rp          ;3 X HL
```

Note that you cannot use EX DE,HL here, since it changes HL.

8. Multiply an index register by 2.

```
ADD   xy,xy
```

9. Multiply memory locations ADDR and ADDR+1 (MSB in ADDR+1) by 2.

```
LD    HL,ADDR
SLA   (HL)           ;SHIFT LSB LEFT LOGICALLY
INC   HL
RL    (HL)           ;THEN ROTATE MSB TO PICK UP CARRY
```

or

```
LD    xy,ADDR
SLA   (xy+0)         ;SHIFT LSB LEFT LOGICALLY
RL    (xy+1)         ;THEN ROTATE MSB TO PICK UP CARRY
```

Note that you must rotate the more significant byte to pick up the Carry produced by shifting the less significant byte.

Division Instructions

1. Divide accumulator by 2 unsigned.

```
        SRL   A                  ;DIVIDE BY 2, CLEARING SIGN
```

2. Divide accumulator by 4 unsigned.

```
        SRL   A                  ;DIVIDE BY 2, CLEARING SIGN
        SRL   A                  ;THEN BY 2 AGAIN
```

or

```
        RRA                      ;ROTATE A RIGHT TWICE
        RRA
        AND   00111111B          ;THEN CLEAR 2 MSB'S
```

Since SRL is a 2-byte instruction, it eventually becomes faster to use the 1-byte instruction RRA and clear the more significant bits explicitly at the end.

3. Divide accumulator by 2 signed.

```
        SRA   A                  ;DIVIDE BY 2, EXTENDING SIGN
```

4. Divide memory locations ADDR and ADDR+1 (MSB in ADDR+1) by 2 unsigned.

```
        LD    XY,ADR
        SRL   (XY+1)             ;SHIFT MSB RIGHT LOGICALLY
        RR    (XY+0)             ;THEN ROTATE LSB RIGHT
```

Rotating the less significant byte picks up the Carry from the more significant byte.

5. Divide memory locations ADDR and ADDR+1 (MSB in ADDR+1) by 2 signed.

```
        LD    XY,ADR
        SRA   (XY+1)             ;SHIFT MSB RIGHT ARITHMETICALLY
        RR    (XY+0)             ;THEN ROTATE LSB RIGHT
```

6. Divide register pair by 2 unsigned.

```
        SRL   rph                ;SHIFT MSB RIGHT LOGICALLY
        RR    rpl                ;THEN ROTATE LSB RIGHT
```

7. Divide register pair by 2 signed.

```
        SRA   rph                ;SHIFT MSB RIGHT ARITHMETICALLY
        RR    rpl                ;THEN ROTATE LSB RIGHT
```

Comparison Instructions

1. Compare VALUE with accumulator bit by bit, setting each bit position that is different.

```
        XOR   VALUE
```

Remember, the EXCLUSIVE OR of two bits is 1 if and only if the two bits are different.

2. Compare register with accumulator bit by bit, setting each bit position that is different.

```
        XOR   reg
```

3. Compare register pairs (rp and HL). Set Carry if rp is larger (unsigned) than HL and clear Carry otherwise.

```
        AND   A                 ;CLEAR CARRY
        SBC   HL,rp
```

This sequence changes HL.

4. Compare register pair HL with 16-bit number VAL16.

```
        LD    rp,-VAL16         ;FORM HL - VAL16 BY ADDING
        ADD   HL,rp
```

or

```
        AND   A                 ;CLEAR CARRY
        LD    rp,VAL16
        SBC   HL,rp
```

Carry is an inverted borrow after the first alternative and a true borrow after the second. Both sequences change HL and rp.

5. Compare index register with 16-bit number VAL16. Clear Carry if VAL16 is greater than index register and set Carry otherwise.

```
        LD    rp,-VAL16         ;FORM INDEX REGISTER - VAL16
        ADD   xy,rp
```

Carry is an inverted borrow here, since we are subtracting by adding the two's complement.

6. Compare register pair with memory locations ADDR and ADDR+1 (MSB in ADDR+1).

```
        AND   A                 ;CLEAR CARRY
        LD    rp,(ADDR)         ;SUBTRACT REGISTER PAIR
        SBC   HL,rp
```

Carry is a true borrow.

7. Compare index register with memory locations ADDR and ADDR+1 (MSB in ADDR+1).

```
        PUSH  xy              ;MOVE INDEX REGISTER TO HL
        POP   HL
        AND   A               ;CLEAR CARRY
        LD    rp,(ADDR)       ;FORM INDEX REGISTER - OTHER OPERAND
        SBC   HL,rp
```

The Z80 has no SBC xy instruction.

8. Compare stack pointer with the 16-bit number VAL16.

```
        LD    HL,0            ;MOVE STACK POINTER TO HL
        ADD   HL,SP
        LD    rp,-VAL16
        ADD   HL,rp
```

Carry is an inverted borrow.

9. Compare stack pointer with memory locations ADDR and ADDR+1 (MSB in ADDR+1).

```
        LD    HL,0            ;MOVE STACK POINTER TO HL
        ADD   HL,SP
        LD    rp,(ADDR)
        AND   A               ;CLEAR CARRY
        SBC   HL,rp           ;FORM SP - MEMORY
```

Carry is a true borrow.

Two's Complement (Negate) Instructions

1. Negate register.

```
        SUB   A               ;FORM 0 - REG
        SUB   reg
        LD    reg,A
```

or

```
        LD    A,reg
        NEG
        LD    reg,A
```

2. Negate memory location ADDR.

```
        SUB   A
        LD    HL,ADDR
        SUB   (HL)              ;FORM 0 - (MEMORY)
        LD    (HL),A
```

or

```
        LD    HL,ADDR
        LD    A,(HL)            ;FORM -  (ADDR)
        NEG
        LD    (HL),A
```

3. Negate register pair.

```
        LD    A,rph             ;16-BIT ONE'S COMPLEMENT
        CPL
        LD    rph,A
        LD    A,rpl
        CPL
        LD    rpl,A
        INC   rp                ;ADD 1 FOR TWO'S COMPLEMENT
```

or

```
        LD    HL,0              ;FORM 0 - (RP)
        AND   A                 ;CLEAR CARRY
        SBC   HL,rp
```

The second sequence leaves the negative in HL; it can then be moved easily to another register pair.

4. Negate memory locations ADDR and ADDR+1 (MSB in ADDR+1).

```
        LD    HL,0              ;FORM 0 - (MEMORY)
        LD    rp,(ADDR)
        AND   A
        SBC   HL,rp
        LD    (ADDR),HL
```

5. Nine's complement accumulator (that is, replace (A) with 99−(A)).

```
        LD    reg,A
        LD    A,99H
        SUB   reg
```

No DAA is necessary, since 99 − (A) is always a valid BCD number if the accumulator originally contained a valid BCD number.

6. Ten's complement accumulator (that is, replace (A) with 100−(A)).

```
        NEG                     ;FORM 0 - ACCUMULATOR
        DAA                     ;THEN DECIMAL ADJUST
```

Extend Instructions

1. Extend accumulator to a 16-bit unsigned number in a register pair.

```
LD    rpl,A             ;8-BIT MOVE
LD    rph,0             ;EXTEND 8 BITS TO 16 BITS
```

This procedure allows you to use the value in the accumulator as an index. ADD HL or ADD xy will then add the index to the base.

2. Extend accumulator to a 16-bit signed number in a register pair.

```
LD    rpl,A             ;8-BIT MOVE
ADD   A,A               ;MOVE SIGN BIT TO CARRY
SBC   A,A               ;SUBTRACT SIGN BIT FROM ZERO
LD    rph,A             ;EXTEND 8 BITS TO 16 BITS SIGNED
```

SBC A,A produces 00 if Carry is 0 and FF_{16} if Carry is 1. It thus extends Carry across the entire accumulator.

3. Extend memory location ADDR to a 16-bit signed number in memory locations ADDR (LSB) and ADDR+1 (MSB).

```
LD    HL,ADDR           ;FETCH NUMBER
LD    A,(HL)
ADD   A,A               ;MOVE SIGN TO CARRY
SBC   A,A               ;FORM SIGN BYTE (00 OR FF)
INC   HL                ;STORE SIGN BYTE
LD    (HL),A
```

4. Extend bit 0 of accumulator across entire accumulator; that is, (A) = 00 if bit 0 = 0 and FF_{16} if bit 0 = 1.

```
RRA                     ;MOVE BIT 0 TO CARRY
SBC   A,A               ;FORM 0 - BIT 0
```

5. Sign function. Replace the value in the accumulator by 00 if it is positive and by FF_{16} if it is negative.

```
ADD   A,A               ;MOVE SIGN BIT TO CARRY
SBC   A,A               ;FORM 0 - SIGN BIT
```

LOGICAL INSTRUCTIONS

This group includes logical AND, logical OR, logical EXCLUSIVE OR, logical NOT (complement), shift, rotate, and test instructions. Also included are arithmetic instructions (such as adding the accumulator to itself) that perform logical functions.

Logical AND Instructions

1. Clear bits of accumulator.

```
        AND   MASK              ;CLEAR BITS BY MASKING
```

MASK has 0's in the bit positions to be cleared and 1's in the positions to be left unchanged. For example:

```
        AND   11011011B         ;CLEAR BITS 2 AND 5
```

Remember, logically ANDing a bit with 1 does not affect its value. Since RES can clear only one bit at a time, the following sequence would be needed to produce an equivalent result:

```
        RES   2,A               ;CLEAR BIT 2
        RES   5,A               ;AND THEN CLEAR BIT 5
```

2. Bit test—set the flags as if accumulator had been logically ANDed with a register or memory location, but do not change the accumulator.

```
        LD    reg,A             ;SAVE ACCUMULATOR
        LD    HL,ADDR
        AND   (HL)              ;PERFORM LOGICAL AND
        LD    A,reg             ;RESTORE ACCUMULATOR
```

LD does not affect any flags.

3. Test bits of accumulator. Set the Zero flag to 1 if all the tested bits are 0 and to 0 otherwise.

```
        AND   MASK              ;TEST BITS BY MASKING
```

MASK has 1's in the positions to be tested and 0's elsewhere. The Zero flag is set to 1 if all the tested bit positions are 0, and to 0 otherwise. Since the BIT instruction can test only one bit position at a time, AND MASK is equivalent to a sequence of BIT instructions and conditional jumps. For example:

```
        AND   01000001OB        ;TEST BITS 1 AND 6 FOR ZERO
```

is equivalent to the sequence

```
            BIT   6,A           ;TEST BIT 6 FOR ZERO
            JR    NZ,DONE       ;BRANCH IF IT IS NOT ZERO
            BIT   1,A           ;THEN TEST BIT 1 FOR ZERO
DONE:       NOP
```

4. Logical AND immediate with flags (condition codes). Logically AND a byte of immediate data with the Flag register, clearing those flags that are logically ANDed with 0's.

```
            PUSH  AF            ;MOVE AF TO A REGISTER PAIR
            POP   rp
            LD    A,MASK        ;CLEAR FLAGS
            AND   rpl
            LD    rpl,A
            PUSH  rp            ;RESTORE AF WITH FLAGS CLEARED
            POP   AF
```

This sequence changes a register pair (BC, DE, or HL).

Logical OR Instructions

1. Set bits of accumulator.

```
            OR    MASK          ;SET BITS BY MASKING
```

MASK has 1's in the bit positions to be set and 0's elsewhere. For example:

```
            OR    00010010B     ;SET BITS 1 AND 4
```

Remember, logically ORing a bit with 0 does not affect its value. Since SET can set only one bit at a time, we would need the following sequence to produce the same result:

```
            SET   1,A           ;SET BIT 1
            SET   4,A           ;AND THEN SET BIT 4
```

2. Test a register pair for 0. Set the Zero flag if both halves of a register pair are 0.

```
            LD    A,rph         ;TEST REGISTER PAIR FOR ZERO
            OR    rpl
```

The Zero flag is set if and only if both halves of register pair rp are 0. The accumulator and the other flags are also changed.

3. Logical OR immediate with flags (condition codes). Logically OR a byte of immediate data with the flag register, setting those flags that are logically ORed with 1's.

```
            PUSH  AF            ;MOVE AF TO A REGISTER PAIR
            POP   rp
```

```
        LD   A,MASK          ;SET FLAGS
        OR   rpl
        LD   rpl,A
        PUSH rp              ;RESTORE AF WITH FLAGS SET
        POP  AF
```

This sequence changes a register pair (BC, DE, or HL).

Logical EXCLUSIVE OR Instructions

1. Complement bits of accumulator.

```
        XOR  MASK            ;COMPLEMENT BITS BY MASKING
```

MASK has 1's in the bit positions to be complemented and 0's in the positions that are to be left unchanged. For example:

```
        XOR  11000000B       ;COMPLEMENT BITS 6 AND 7
```

Remember, logically EXCLUSIVE ORing a bit with 0 leaves it unchanged.

2. Complement accumulator, setting flags.

```
        XOR  11111111B       ;INVERT AND SET FLAGS
```

Logically EXCLUSIVE ORing with all 1's inverts all the bits. This instruction differs from CPL only in that it affects the flags, whereas CPL does not.

3. Compare register with accumulator bit by bit, setting each bit position that is different.

```
        XOR  reg             ;BIT BY BIT COMPARISON
```

The EXCLUSIVE OR function is the same as a "not equal" function. Note that the Sign flag is 1 if the two operands have different values in bit 7.

4. Add register to accumulator logically (that is, without any carries between bit positions).

```
        XOR  reg             ;LOGICAL ADDITION
```

The EXCLUSIVE OR function is also the same as a bit-by-bit sum with no carries. Logical sums are often used to form checksums and error-detecting or error-correcting codes.

Logical NOT Instructions

1. Complement accumulator, setting flags.

```
        XOR   11111111B         ;INVERT AND SET FLAGS
```

Logically EXCLUSIVE ORing with all 1's inverts all the bits. This instruction differs from CPL only in that it affects the flags, whereas CPL does not.

2. Complement bits of accumulator.

```
        XOR   MASK              ;COMPLEMENT BIT BY MASKING
```

MASK has 1's in the bit positions to be complemented and 0's in the positions that are to be left unchanged. For example:

```
        XOR   01010001B         ;COMPLEMENT BITS 0, 4, AND 6
```

Remember, logically EXCLUSIVE ORing a bit with 0 leaves it unchanged.

3. Complement memory location ADDR.

```
        LD    HL,ADDR
        LD    A,(HL)            ;OBTAIN DATA
        CPL                     ;COMPLEMENT
        LD    (HL),A            ;RESTORE RESULT
```

CPL applies only to the accumulator.

4. Complement bit 0 of a register.

```
        INC   reg
```

or

```
        DEC   reg
```

Either instruction may, of course, affect the other bits in the register. The final value of bit 0 will surely be 0 if it was originally 1 and if it was originally 0.

5. Complement bit 0 of a memory location.

```
        LD    HL,ADDR
        INC   (HL)
```

or

```
        LD    HL,ADDR
        DEC   (HL)
```

6. Complement digit of accumulator.

- Less significant digit

```
        XOR   00001111B         ;COMPLEMENT LESS SIGNIFICANT DIGIT
```

- More significant digit

```
        XOR  11110000B        ;COMPLEMENT MORE SIGNIFICANT DIGIT
```

These procedures are useful if the accumulator contains a decimal digit in negative logic, such as the input from a typical ten-position rotary or thumbwheel switch.

7. Complement a register pair.

```
        LD   HL,0FFFFH        ;SET HL TO ALL ONES
        AND  A                ;CLEAR CARRY
        SBC  HL,rp            ;SUBTRACT REGISTER PAIR FROM ALL ONES
```

The result ends up in HL.

Shift Instructions

1. Shift accumulator left logically.

```
        ADD  A,A              ;SHIFT A LEFT LOGICALLY
```

Adding the accumulator to itself is equivalent to a logical left shift.

2. Shift register pair HL left logically.

```
        ADD  HL,HL            ;SHIFT HL LEFT LOGICALLY
```

3. Shift index register left logically.

```
        ADD  xy,xy            ;SHIFT IX OR IY LEFT LOGICALLY
```

4. Shift register pair right logically.

```
        SRL  rph              ;SHIFT MSB RIGHT LOGICALLY
        RR   rpl              ;AND THEN ROTATE LSB RIGHT
```

The key point here is that the less significant byte must be rotated to pick up the Carry from the logical shifting of the more significant byte.

5. Shift register pair right arithmetically.

```
        SRA  rph              ;SHIFT MSB RIGHT ARITHMETICALLY
        RR   rpl              ;AND THEN ROTATE LSB RIGHT
```

The rotation of the less significant byte is the same as in the logical shift.

6. Shift memory locations ADDR and ADDR+1 (MSB in ADDR+1) left logically.

```
        LD      HL,ADDR
        SLA     (HL)                ;SHIFT LSB LEFT LOGICALLY
        INC     HL
        RL      (HL)                ;AND THEN ROTATE MSB LEFT
```

or

```
        LD      xy,ADDR
        SLA     (xy+0)              ;SHIFT LSB LEFT LOGICALLY
        RL      (xy+1)              ;AND THEN ROTATE MSB LEFT
```

To produce a 16-bit left shift, you must shift the less significant byte first and then rotate the more significant byte.

7. Shift memory locations ADDR and ADDR+1 (MSB in ADDR+1) right logically.

```
        LD      HL,ADDR+1
        SRL     (HL)                ;SHIFT MSB RIGHT LOGICALLY
        DEC     HL
        RR      (HL)                ;AND THEN ROTATE LSB RIGHT
```

or

```
        LD      xy,ADDR
        SRL     (xy+1)              ;SHIFT MSB RIGHT LOGICALLY
        RR      (xy+0)              ;AND THEN ROTATE LSB RIGHT
```

8. Digit swap accumulator. That is, exchange the four least significant bits with the four most significant bits.

```
        RLCA                        ;DIGIT SHIFT = 4 LEFT ROTATES
        RLCA
        RLCA
        RLCA
```

or

```
        RRCA                        ;DIGIT SHIFT = 4 RIGHT ROTATES
        RRCA
        RRCA
        RRCA
```

9. Normalize accumulator. That is, shift the accumulator left until its most significant bit is 1. Do not shift at all if the accumulator contains 0.

```
        AND     A                   ;TEST ACCUMULATOR
        JP      M,DONE              ;EXIT IF ALREADY NORMALIZED
        JR      Z,DONE              ;EXIT IF ZERO
SHIFT:  ADD     A,A                 ;OTHERWISE, SHIFT A LEFT 1 BIT
        JP      P,SHIFT             ;KEEP SHIFTING UNTIL NORMALIZED
DONE:   NOP
```

10. Normalize register pair HL. That is, shift the 16-bit number left until its most significant bit is 1. Do not shift the number at all if it is 0.

```
        LD    A,H           ;IS ENTIRE NUMBER 0?
        OR    L
        JR    Z,DONE        ;YES, DONE
SHIFT:  ADD   HL,HL         ;SHIFT NUMBER LEFT 1 BIT
        JR    NC,SHIFT      ;KEEP SHIFTING UNTIL CARRY IS 1
        RR    H             ;THEN SHIFT BACK ONCE
        RR    L
DONE:   NOP
```

ADD HL affects the Carry but not the Sign or Zero flag.

Rotate Instructions

1. Rotate register pair right.

```
        RRC   rpl           ;COPY BIT 0 FOR ROTATION
        RL    rpl           ;CARRY = BIT 0
        RR    rph           ;ROTATE MSB WITH BIT 0
        RR    rpl           ;THEN ROTATE LSB RIGHT
```

The RRC rpl instruction places bit 0 both in bit 7 and in the Carry flag; RL rpl then restores the register but leaves the original bit 0 in the Carry.

2. Rotate register pair left.

```
        RLC   rph           ;COPY BIT 15 FOR ROTATION
        RR    rph           ;CARRY = BIT 15
        RL    rpl           ;ROTATE LSB WITH BIT 15
        RL    rph           ;THEN ROTATE MSB LEFT
```

RLC rph places bit 7 of the more significant byte both in bit 0 and in the Carry. RR rph then restores the register but leaves the original bit 7 (bit 15 of the 16-bit register pair) in the Carry.

3. Rotate accumulator left through Carry, setting flags.

```
        ADC   A,A           ;ROTATE LEFT AND SET FLAGS
```

This instruction is the same as RLA, except that it affects all the flags whereas RLA affects only the Carry.

4. Rotate register pair right through Carry.

```
        RR    rph           ;ROTATE MSB RIGHT WITH CARRY
        RR    rpl           ;THEN ROTATE LSB RIGHT WITH CARRY
```

5. Rotate register pair left through Carry.

```
        RL    rpl              ;ROTATE LSB LEFT WITH CARRY
        RL    rph              ;ROTATE MSB LEFT WITH CARRY
```

6. Rotate memory locations ADDR and ADDR+1 (MSB in ADDR+1) right 1 bit position through Carry.

```
        LD    HL,ADDR+1
        RR    (HL)             ;ROTATE MSB RIGHT WITH CARRY
        DEC   HL
        RR    (HL)             ;THEN ROTATE LSB RIGHT WITH CARRY
```

or

```
        LD    xy,ADDR
        RR    (xy+1)           ;ROTATE MSB RIGHT WITH CARRY
        RR    (xy+0)           ;THEN ROTATE LSB RIGHT WITH CARRY
```

7. Rotate memory locations ADDR and ADDR+1 (MSB in ADDR+1) left one bit position through Carry.

```
        LD    HL,ADDR
        RL    (HL)             ;ROTATE LSB LEFT WITH CARRY
        INC   HL
        RL    (HL)             ;THEN ROTATE MSB LEFT WITH CARRY
```

or

```
        LD    xy,ADDR
        RL    (xy+0)           ;ROTATE LSB LEFT WITH CARRY
        RL    (xy+1)           ;THEN ROTATE MSB LEFT WITH CARRY
```

Test Instructions

1. **Test accumulator**. Set flags according to the value in the accumulator without changing that value.

```
        AND   A                ;TEST ACCUMULATOR
```

or

```
        OR    A                ;TEST ACCUMULATOR
```

Both alternatives clear the Carry.

2. **Test register**. Set flags according to the value in a register without changing that value.

```
        INC   reg              ;TEST REGISTER
        DEC   reg
```

This sequence does not affect the Carry or the accumulator.

3. **Test memory location**. Set flags according to the value in memory location ADDR without changing that value.

```
        LD    HL,ADDR          ;TEST MEMORY LOCATION ADDR
        INC   (HL)
        DEC   (HL)
```

This sequence does not affect the Carry or the accumulator.

4. **Test register pair**. Set the Zero flag according to the value in a register pair without changing that value.

```
        LD    A,rph            ;TEST REGISTER PAIR
        OR    rpl
```

This sequence changes the accumulator and the other flags.

5. **Test index register**. Set the Zero flag according to the value in an index register without changing that value.

```
        PUSH  xy               ;MOVE INDEX REG TO REGISTER PAIR
        POP   rp
        LD    A,rph            ;TEST REGISTER PAIR
        OR    rpl
```

This sequence changes a register pair, the accumulator, and the other flags.

6. **Test a pair of memory locations**. Set the Zero flag according to the contents of memory locations ADDR and ADDR+1.

```
        LD    HL,(ADDR)        ;TEST A MEMORY WORD
        LD    A,H
        OR    L
```

This sequence changes HL, the accumulator, and the other flags.

7. **Test bits of accumulator**. Set the Zero flag if all the tested bits are 0's and clear the Zero flag otherwise.

```
        AND   MASK             ;TEST BITS BY MASKING
```

MASK has 1's in the bit positions to be tested and 0's elsewhere. The Zero flag is set to 1 if all the tested bits are 0's and to 0 otherwise. For example:

```
        AND   10000001B        ;TEST BITS 0 AND 7
```

The Zero flag is set to 1 if bits 0 and 7 of the accumulator are both zero, and to 0 otherwise. The BIT instruction, on the other hand, can only handle one bit at a time; for example:

```
        BIT   7,A              ;TEST BIT 7
```

To duplicate the AND instruction, we would need the sequence

```
            BIT   7,A           ;TEST BIT 7
            JR    NZ,DONE       ;EXIT IF IT IS 1
            BIT   0,A           ;TEST BIT 0
DONE:       NOP
```

8. Compare register with accumulator bit by bit. Set each bit position that is different to 1.

```
            XOR   reg           ;BIT-BY-BIT COMPARISON
```

The EXCLUSIVE OR function is the same as a "not equal" function.

9. Bit test. Set flags as if the accumulator had been logically ANDed with a memory location, but do not change the accumulator.

```
            LD    reg,A         ;SAVE ACCUMULATOR
            LD    HL,ADDR
            AND   (HL)          ;PERFORM LOGICAL AND
            LD    A,reg         ;RESTORE ACCUMULATOR
```

DATA TRANSFER INSTRUCTIONS

In this group, we consider load, store, move, exchange, input, output, clear, and set instructions. We also include arithmetic instructions (such as subtracting the accumulator from itself) that move a specific value or the contents of another register to the accumulator or other destination without changing any data.

Load Instructions

1. Load register direct.

```
            LD    A,(ADDR)
            LD    reg,A
```

or

```
            LD    HL,ADDR
            LD    reg,(HL)
```

The first alternative uses the accumulator, while the second alternative uses register pair HL.

2. Load register indirect.

   · From address in HL

```
LD    reg,(HL)
```

- From address in BC or DE

```
LD    A,(rp)
LD    reg,A
```

Note that only the accumulator can be loaded indirectly via BC or DE.

- From address in an index register

```
LD    reg,(xy+0)
```

3. Load flag register with the 8-bit number VALUE.

```
LD    rpl,VALUE        ;PUT VALUE IN LSB OF REGISTER PAIR
PUSH  rp               ;MOVE TO FLAGS THROUGH STACK
POP   AF
```

The limitation of pushing and popping register pairs causes some unnecessary operations.

4. Load interrupt vector register with the 8-bit number VALUE.

```
LD    A,VALUE
LD    I,A
```

5. Load refresh register with the 8-bit number VALUE.

```
LD    A,VALUE
LD    R,A
```

6. Load flag register direct from memory location ADDR.

```
LD    HL,(ADDR)        ;LOAD L FROM ADDR
PUSH  HL               ;HL TO STACK, L ON TOP
POP   AF               ;HL TO AF WITH L TO FLAGS
```

This procedure allows a user to initialize the flag register for debugging or testing purposes. Note that it changes the accumulator and the less significant byte of a register pair.

7. Load interrupt vector register direct from memory location ADDR.

```
LD    A,(ADDR)
LD    I,A
```

8. Load refresh register direct from memory location ADDR.

```
LD    A,(ADDR)
LD    R,A
```

9. Load register pair HL indirect from address in HL.

```
LD    A,(HL)          ;LOAD LSB
INC   HL
LD    H,(HL)          ;LOAD MSB
LD    L,A
```

10. Load register pair (BC or DE) indirect from address in HL.

```
LD    rpl,(HL)        ;LOAD LSB
INC   HL
LD    rph,(HL)        ;LOAD MSB
DEC   HL              ;RESTORE HL TO ORIGINAL VALUE
```

11. Load alternate processor status (AF′) from stack.

```
POP   AF
EX    AF,AF'
```

12. Load memory locations PTR and PTR+1 (MSB in PTR+1) with ADDR.

```
LD    HL,ADDR         ;GET INDIRECT ADDRESS
LD    (PTR),HL        ;STORE INDIRECT ADDRESS IN MEMORY
```

Store Instructions

1. Store register direct.

```
LD    A,reg
LD    (ADDR),A
```

or

```
LD    HL,ADDR
LD    (HL),reg
```

The first alternative uses the accumulator, whereas the second uses register pair HL.

2. Store register indirect.

- At address in HL

```
LD    (HL),reg
```

- At address in DE or BC

```
LD    A,reg
LD    (rp),A
```

Only the accumulator can be stored at the address in BC or DE.

- At address in an index register

```
LD    (xy+0),reg
```

3. Store flag register direct.

```
PUSH AF              ;F TO TOP OF STACK
POP  HL              ;F TO L
LD   (ADDR),HL       ;F TO ADDR, DESTROY ADDR+1
```

or

```
PUSH AF              ;F TO TOP OF STACK
POP  HL              ;F TO L
LD   A,L             ;F TO A
STA  ADDR            ;F TO ADDR
```

4. Store interrupt vector register direct.

```
LD   A,I
LD   (ADDR),A
```

5. Store refresh register direct.

```
LD   A,R
LD   (ADDR),A
```

6. Store register pair (BC or DE) indirect at address in HL.

```
LD   (HL),rpl        ;STORE LSB
INC  HL
LD   (HL),rph        ;STORE MSB
DEC  HL              ;RETURN HL TO ORIGINAL VALUE
```

The register pair is stored in memory in the usual upside-down fashion.

7. Store alternate processor status (AF′) in stack.

```
EX   AF,AF'
PUSH AF
```

Move Instructions

1. Transfer accumulator to flag register.

```
LD   rpl,A
PUSH rp
POP  AF
```

The flag register is the less significant byte of register pair AF. This sequence also changes the accumulator and the less significant byte of a register pair (i.e., C, E, or L).

2. Transfer flag register to accumulator.

```
PUSH AF
POP  rp
MOV  A,rpl
```

This sequence changes register pair rp.

3. Move register pair 1 to register pair 2.

```
LD   rp2l,rp1l
LD   rp2h,rp1h
```

This sequence transfers the contents of register pair rp1 to rp2 without changing rp1. Remember, EX DE,HL exchanges register pairs DE and HL specifically.

4. Move stack pointer to HL.

```
LD   HL,0
ADD  HL,SP
```

5. Move stack pointer to an index register.

```
LD   xy,0
ADD  xy,SP
```

6. Move index register to register pair.

```
PUSH xy
POP  rp
```

7. Move register pair to index register.

```
PUSH rp
POP  xy
```

8. Move index register IX to index register IY.

```
PUSH IX
POP  IY
```

9. Move index register IY to index register IX.

```
PUSH IY
POP  IX
```

10. Move HL to program counter.

```
JP   (HL)
```

11. Move index register to program counter.

```
        JP    (xy)
```

12. Move memory locations ADDR and ADDR+1 (MSB in ADDR+1) to the program counter (an indirect jump).

```
        LD    HL,(ADDR)
        JP    (HL)
```

13. Move multiple (fill). Place the accumulator in successive memory locations starting at the address in register pair HL. The number of bytes to be filled (one or more) is in register B.

```
FILBYT:   LD    (HL),A          ;FILL A MEMORY LOCATION
          INC   HL              ;POINT TO NEXT LOCATION
          DJNZ  FILBYT          ;COUNT BYTES
```

This routine can initialize an array or buffer. If more than 256 bytes are to be filled, the repeated block move instructions become handy. The approach is to fill the first byte from the accumulator and then use a repeated block move to fill the succeeding bytes. The destination pointer is always one byte ahead of the source pointer, so the data being moved is always the same.

```
          LD    (HL),A          ;FILL THE FIRST BYTE MANUALLY
          LD    D,H             ;DESTINATION POINTER IS 1 BYTE UP
          LD    E,L
          INC   DE
          DEC   BC              ;COUNT DOWN 1 BYTE
          LDIR                  ;FILL THE REST AUTOMATICALLY
```

Exchange Instructions

1. Exchange registers using the accumulator.

```
          LD    A,reg1
          LD    reg1,reg2
          LD    reg2,A
```

2. Exchange register pairs.

- DE with HL

```
          EX    DE,HL
```

- BC with HL

```
          PUSH  BC              ;BC TO TOP OF STACK
          EX    HL,(SP)         ;BC TO HL, HL TO TOP OF STACK
          POP   BC              ;HL TO BC
```

EX HL,(SP) exchanges HL with the top of the stack.

- general, rp1 with rp2

```
PUSH rp1            ;PUT RP1, RP2 IN STACK
PUSH rp2
POP  rp1            ;EXCHANGE BY POPPING IN WRONG ORDER
POP  rp2
```

3. Exchange stack pointer with HL.

```
EX   DE,HL          ;HL TO DE
LD   HL,0           ;SP TO HL
ADD  HL,SP
EX   DE,HL          ;SP TO DE, RESTORE HL
LD   SP,HL          ;HL TO SP
EX   DE,HL          ;SP TO HL
```

This procedure can be used to differentiate between the user stack and the operating system or monitor stack.

4. Exchange index register with register pair.

```
PUSH xy             ;SAVE INDEX REG, REG PAIR IN STACK
PUSH rp
POP  xy             ;EXCHANGE BY POPPING IN WRONG ORDER
POP  rp
```

5. Exchange index registers.

```
PUSH IX             ;SAVE BOTH INDEX REGISTERS IN STACK
PUSH IY
POP  IX             ;EXCHANGE BY POPPING IN WRONG ORDER
POP  IY
```

Clear Instructions

1. Clear the accumulator.

```
SUB  A
```

or

```
XOR  A
```

or

```
LD   A,0
```

The third alternative executes more slowly and occupies more memory than the other two, but does not affect the flags.

2. Clear a register.

```
        LD    reg,0
```

3. Clear memory location ADDR.

```
        SUB   A
        LD    (ADDR),A
```

or

```
        LD    HL,ADDR
        LD    (HL),0
```

The second alternative executes more slowly than the first, but does not affect the accumulator or the flags. Of course, it does use register pair HL.

4. Clear a register pair.

```
        LD    rp,0
```

5. Clear memory locations ADDR and ADDR+1.

```
        LD    HL,0
        LD    (ADDR),HL
```

HL is faster to use here than DE or BC.

6. Clear Carry flag.

```
        AND   A
```

or

```
        OR    A
```

Any other logical instruction (except CPL) will also clear the Carry, but these two are particularly useful because they do not change the accumulator. Remember, ANDing or ORing a bit with itself does not affect its value. To clear Carry without affecting any other flags, use the sequence

```
        SCF                     ;FIRST SET CARRY FLAG
        CCF                     ;THEN CLEAR CARRY BY COMPLEMENTING
```

7. Clear bits of accumulator.

```
        AND   MASK              ;CLEAR BITS BY MASKING
```

MASK has 0's in the bit positions to be cleared and 1's in the positions that are to be left unchanged. For example:

```
        AND   10111110B         ;CLEAR BITS 0 AND 6
```

RES can clear only one bit at a time.

Set Instructions

1. Set the accumulator to FF_{16} (all 1's in binary).

```
        LD    A,0FFH
```

or

```
        SUB   A
        DEC   A
```

2. Set register to FF_{16}.

```
        LD    reg,0FFH
```

3. Set memory location ADDR to FF_{16}.

```
        LD    A,0FFH
        LD    (ADDR),A
```

or

```
        LD    HL,ADDR
        LD    (HL),0FFH
```

4. Set bits of accumulator.

```
        OR    MASK            ;SET BITS BY MASKING
```

MASK has 1's in the bit positions to be set and 0's elsewhere. For example:

```
        OR    10110000B       ;SET BITS 4, 5, AND 7
```

The SET instruction can set only one bit at a time.

BRANCH (JUMP) INSTRUCTIONS

Unconditional Branch Instructions

1. Jump indirect.
   - To address in HL

```
        JP    (HL)
```

   - To address at the top of the stack

```
        RET
```

Note that RET is just an ordinary indirect jump that obtains its destination from the top of the stack. RET can be used for purposes other than returning from a subroutine.

- To address in DE

```
        EX      DE,HL
        JP      (HL)
```

- To address in BC

```
        LD      H,B
        LD      L,C
        JP      (HL)
```

or

```
        PUSH    BC
        RET
```

The second alternative is much slower than the first (21 cycles as compared to 12 cycles), but does not change HL.

- To address in an index register

```
        JP      (xy)
```

- To address in memory locations ADDR and ADDR+1

```
        LD      HL,ADDR         ;FETCH INDIRECT ADDRESS
        JP      (HL)            ;AND BRANCH TO IT
```

2. Jump indexed, assuming that the base of the address table is in register pair HL and the index is in the accumulator.

```
        ADD     A,A             ;DOUBLE INDEX FOR 2-BYTE ENTRIES
        LD      E,A             ;EXTEND INDEX TO 16 BITS
        LD      D,0
        ADD     HL,DE           ;CALCULATE ADDRESS OF ELEMENT
        LD      E,(HL)          ;FETCH ELEMENT FROM ADDRESS TABLE
        INC     HL
        LD      D,(HL)
        EX      DE,HL           ;AND JUMP TO IT
        JP      (HL)
```

We have assumed that the address table (jump table) consists of as many as 128 2-byte entries, stored in the usual Z80 format with the less significant byte at the lowei address. A typical table would be

```
JTAB:   DW      ROUT0           ;ADDRESS ENTRY 0
        DW      ROUT1           ;ADDRESS ENTRY 1
        DW      ROUT2           ;ADDRESS ENTRY 2
        ...
```

3. Jump and link; that is, transfer control to address DEST, saving the current program counter in register pair HL.

```
         LD    HL,HERE      ;LOAD H AND L WITH LINK
HERE:    JP    DEST         ;TRANSFER CONTROL
```

This procedure can provide a subroutine capability that does not use the stack. The subroutine can return control by adjusting the link and executing JP (HL). For example, to return control to the instruction immediately following JP DEST, the subroutine would have to add 3 to HL (since JP DEST occupies 3 bytes). Of course, the link could also be changed to HERE+3.

Conditional Branch Instructions

1. Branch if 0.

   - Branch if accumulator contains 0

```
         AND   A            ;TEST ACCUMULATOR
         JR    Z,DEST
```

   - Branch if a register contains 0

```
         INC   reg          ;TEST REGISTER
         DEC   reg
         JR    Z,DEST
```

   - Branch if memory location ADDR contains 0

```
         LD    HL,ADDR      ;TEST MEMORY LOCATION
         INC   (HL)
         DEC   (HL)
         JR    Z,DEST
```

or

```
         LD    A,(ADDR)     ;TEST MEMORY LOCATION
         AND   A
         JR    Z,DEST
```

   - Branch if a register pair contains 0

```
         LD    A,rph        ;TEST REGISTER PAIR
         OR    rpl
         JR    Z,DEST
```

   - Branch if an index register contains 0

```
         PUSH  xy           ;MOVE INDEX REGISTER TO REGISTER PAIR
         POP   rp
         LD    A,rph        ;TEST REGISTER PAIR
         OR    rpl
         JR    Z,DEST
```

- Branch if memory locations ADDR and ADDR+1 both contain 0

```
LD    HL,(ADDR)        ;TEST A 16-BIT NUMBER IN MEMORY
LD    A,H
OR    L
JR    Z,DEST
```

- Branch if a bit of a register is 0

```
BIT   N,reg            ;TEST BIT N OF REGISTER
JR    Z,DEST
```

Special cases are

- Branch if bit 7 of the accumulator is 0

```
AND   A                ;TEST BIT 7 OF ACCUMULATOR
JP    P,DEST
```

or

```
RLA                    ;MOVE BIT 7 TO CARRY
JR    NC,DEST
```

The second alternative allows relative jumps, but it also changes the accumulator.

- Branch if bit 6 of the accumulator is 0

```
ADD   A,A              ;SET SIGN FROM BIT 6
JP    P,DEST           ;THEN TEST SIGN FLAG
```

- Branch if bit 0 of the accumulator is 0

```
RRA                    ;MOVE BIT 0 TO CARRY
JR    NC,DEST          ;AND TEST CARRY
```

- Branch if a bit of a memory location is 0

```
LD    HL,ADDR
BIT   N,(HL)           ;TEST BIT N OF MEMORY LOCATION ADDR
JR    Z,DEST
```

- Branch if interrupts are disabled (that is, if interrupt flip-flop IFF2 is 0)

```
LD    A,I              ;MOVE IFF2 TO P/V FLAG
JP    PO,DEST
```

The instruction LD A,I and LD A,R both move interrupt enable flip-flop IFF2 to the Parity/Overflow flag. This sequence can be used to save the current interrupt status before executing a routine that must run with interrupts disabled. That status can then be restored afterward.

2. Branch if not 0.

- Branch if accumulator does not contain 0

```
AND   A                ;TEST ACCUMULATOR
JR    NZ,DEST
```

- Branch if a register does not contain 0

```
INC   reg             ;TEST REGISTER
DEC   reg
JR    NZ,DEST
```

- Branch if memory location ADDR does not contain 0

```
LD    HL,ADDR         ;TEST MEMORY LOCATION
INC   (HL)
DEC   (HL)
JR    NZ,DEST
```

or

```
LD    A,(ADDR)        ;TEST MEMORY LOCATION
AND   A
JR    NZ,DEST
```

- Branch if register pair does not contain 0

```
LD    A,rph           ;TEST REGISTER PAIR
OR    rpl
JR    NZ,DEST
```

- Branch if index register does not contain 0

```
PUSH  xy              ;TRANSFER INDEX REGISTER TO REG PAIR
POP   rp
LD    A,rph           ;TEST REGISTER PAIR
OR    rpl
JR    NZ,DEST
```

- Branch if memory locations ADDR and ADDR+1 do not both contain 0

```
LD    HL,(ADDR)       ;TEST 16-BIT NUMBER IN MEMORY
LD    A,H
OR    L
JR    NZ,DEST
```

- Branch if a bit of a register is 1

```
BIT   N,reg           ;TEST BIT N OF REGISTER
JR    NZ,DEST
```

Special cases are

- Branch if bit 7 of the accumulator is 1

```
AND   A               ;TEST BIT 7 OF ACCUMULATOR
JP    M,DEST
```

or

```
RLA                   ;MOVE BIT 7 TO CARRY
JR    C,DEST
```

The second alternative allows relative jumps, but it also changes the accumulator.

- Branch if bit 6 of the accumulator is 1

```
ADD   A,A              ;SET SIGN FROM BIT 6
JP    M,DEST           ;THEN TEST SIGN FLAG
```

- Branch if bit 0 of the accumulator is 1

```
RRA                    ;MOVE BIT 0 TO CARRY
JR    C,DEST           ;THEN TEST CARRY
```

- Branch if a bit of a memory location is 1

```
LD    HL,ADDR
BIT   N,(HL)           ;TEST BIT N OF MEMORY LOCATION ADDR
JR    NZ,DEST
```

- Branch if interrupts are enabled (that is, if interrupt flip-flop IFF2 is 1)

```
LD    A,I              ;MOVE IFF2 TO P/V FLAG
JP    PE,DEST
```

The instructions LD A,I and LD A,R both move interrupt enable flip-flop IFF2 to the Parity/Overflow flag. This sequence can be used to save the current interrupt status before executing a routine that must run with interrupts disabled. That status can be restored afterward.

3. Branch if Equal.

- Branch if (A) = VALUE

```
CP    VALUE            ;COMPARE BY SUBTRACTING
JR    Z,DEST
```

The following special cases apply to any register or to a memory location addressed using HL or through indexing.

- Branch if (reg) = 1

```
DEC   reg              ;CHECK BY DECREMENTING
JR    Z,DEST           ;AND TESTING RESULT FOR ZERO
```

This procedure can be applied to any primary register, to the memory location addressed through HL, or to memory locations addressed via indexing.

- Branch if (reg) = FF_{16}

```
INC   reg              ;CHECK BY INCREMENTING
JR    Z,DEST           ;AND TESTING RESULT FOR ZERO
```

This procedure can be applied to any primary register, to the memory location addressed through HL, or to memory locations addressed via indexing.

- Branch if (A) = (reg)

```
CP    reg              ;COMPARE BY SUBTRACTING
JR    Z,DEST
```

- Branch if (A) = (ADDR)

```
LD    HL,ADDR          ;COMPARE BY SUBTRACTING
CP    (HL)
JR    Z,DEST
```

- Branch if (rp) = VAL16

```
LD    HL,VAL16
AND   A                ;CLEAR CARRY
SBC   HL,rp
JR    Z,DEST
```

Carry must be cleared, since the Z80 lacks a 16-bit subtract instruction without Carry. Note that the two's complement of VAL16 cannot be added using ADD HL, since that instruction does not affect the Zero flag.

- Branch if (HL) = (rp)

```
AND   A                ;CLEAR CARRY
SBC   HL,rp
JR    Z,DEST
```

Note: Do not use either of the next two sequences to test for stack overflow or underflow, since intervening operations could change the stack pointer by more than 1.

- Branch if (SP) = VAL16

```
LD    HL,VAL16
AND   A                ;CLEAR CARRY
SBC   HL,SP
JR    Z,DEST
```

- Branch if (SP) = (HL)

```
AND   A                ;CLEAR CARRY
SBC   HL,SP
JR    Z,DEST
```

- Branch if (xy) = VAL16

```
PUSH  xy               ;MOVE INDEX REGISTER TO REGISTER PAIR
POP   rp
LD    HL,VAL16         ;THEN COMPARE REGISTER PAIR, VAL16
AND   A                ;CLEAR CARRY
SBC   HL,SP
JR    Z,DEST
```

ADD xy cannot be used to add the two's complement of VAL16, since ADD xy does not affect the Zero flag.

4. Branch if Not Equal.

- Branch if (A) ≠ VALUE

```
CP    VALUE            ;COMPARE BY SUBTRACTING
JR    NZ,DEST
```

The following special cases apply to any register or to a memory location addressed using HL or through indexing.

- Branch if (reg) ≠ 1

```
DEC   reg              ;CHECK BY DECREMENTING
JR    NZ,DEST          ;AND TESTING RESULT FOR ZERO
```

- Branch if (reg) ≠ FF_{16}

```
INC   reg              ;CHECK BY INCREMENTING
JR    NZ,DEST          ;AND TESTING RESULT FOR ZERO
```

- Branch if (A) ≠ (reg)

```
CP    reg              ;COMPARE BY SUBTRACTING
JR    NZ,DEST
```

- Branch if (A) ≠ (ADDR)

```
LD    HL,ADDR          ;COMPARE BY SUBTRACTING
CP    (HL)
JR    NZ,DEST
```

- Branch if (rp) ≠ VAL16

```
LD    HL,VAL16
AND   A              ;CLEAR CARRY
SBC   HL,rp
JR    NZ,DEST
```

- Branch if (HL) ≠ (rp)

```
AND   A                ;CLEAR CARRY
SBC   HL,rp
JR    NZ,DEST
```

Note: You should not use either of the next two sequences to test for stack overflow or underflow, since intervening operations could change the stack pointer by more than 1.

- Branch if (SP) ≠ VAL16

```
LD    HL,VAL16
AND   A                ;CLEAR CARRY
SBC   HL,SP
JR    NZ,DEST
```

- Branch if (SP) ≠ (HL)

```
AND   A                ;CLEAR CARRY
SBC   HL,SP
JR    NZ,DEST
```

- Branch if (xy) ≠ VAL16

```
PUSH  xy               ;MOVE INDEX REGISTER TO REGISTER PAIR
POP   rp
```

```
        LD    HL,VAL16          ;THEN COMPARE REGISTER PAIR AND VAL16
        AND   A                 ;CLEAR CARRY
        SBC   HL,rp
        JR    NZ,DEST
```

ADD xy cannot be used to add the two's complement of VAL16, since ADD xy does not affect the Zero flag.

5. Branch if Positive.

- Branch if contents of accumulator are positive

```
        AND   A                 ;TEST ACCUMULATOR
        JP    P,DEST
```

- Branch if contents of a register are positive

```
        INC   reg               ;TEST REGISTER
        DEC   reg
        JP    P,DEST
```

- Branch if contents of memory location ADDR are positive

```
        LD    HL,ADDR           ;TEST MEMORY LOCATION
        INC   (HL)
        DEC   (HL)
        JP    P,DEST
```

or

```
        LD    A,(ADDR)          ;TEST MEMORY LOCATION
        AND   A
        JP    P,DEST
```

- Branch if contents of a register pair are positive

```
        INC   rph               ;TEST MORE SIGNIFICANT BYTE ONLY
        DEC   rph
        JP    P,DEST
```

- Branch if contents of index register are positive

```
        PUSH  xy                ;TRANSFER INDEX REGISTER TO AF
        POP   AF
        AND   A                 ;TEST MORE SIGNIFICANT BYTE ONLY
        JP    P,DEST
```

- Branch if 16-bit number in memory locations ADDR and ADDR+1 (MSB in ADDR+1) is positive

```
        LD    A,(ADDR+1)        ;TEST MORE SIGNIFICANT BYTE ONLY
        AND   A
        JP    P,DEST
```

or

```
        LD    HL,ADDR+1         ;TEST SIGN BIT OF MSB
        BIT   7,(HL)
        JR    Z,DEST
```

6. Branch if Negative.

- Branch if contents of accumulator are negative

```
        AND   A                 ;TEST ACCUMULATOR
        JP    M,DEST
```

- Branch if contents of a register are negative

```
        INC   reg               ;TEST PRIMARY REGISTER
        DEC   reg
        JP    M,DEST
```

- Branch if contents of memory location ADDR are negative

```
        LD    HL,ADDR           ;TEST MEMORY LOCATION
        INC   (HL)
        DEC   (HL)
        JP    M,DEST
```

or

```
        LD    A,(ADDR)          ;TEST MEMORY LOCATION
        AND   A
        JP    M,DEST
```

- Branch if contents of a register pair are negative

```
        INC   rph               ;TEST MORE SIGNIFICANT BYTE ONLY
        DEC   rph
        JP    M,DEST
```

- Branch if contents of an index register are negative

```
        PUSH  xy                ;MOVE INDEX REGISTER TO AF
        POP   AF
        AND   A                 ;TEST MORE SIGNIFICANT BYTE ONLY
        JP    M,DEST
```

- Branch if 16-bit number in memory locations ADDR and ADDR+1 (MSB in ADDR+1) is negative

```
        LD    A,(ADDR+1)        ;TEST MORE SIGNIFICANT BYTE ONLY
        AND   A
        JP    M,DEST
```

or

```
        LD    HL,ADDR+1         ;TEST SIGN BIT OF MSB
        BIT   7,(HL)
        JR    NZ,DEST
```

7. Signed Branches.

These sequences must allow for two's complement overflow. After a comparison, the setting of the Parity/Overflow flag indicates that overflow occurred. The branches are JP PE (Branch on Overflow) and JP PO (Branch on No Overflow). The idea then is to force a branch if the specified condition holds and overflow did not occur (a true positive), or if the condition does not hold but overflow did occur (a false negative). The operand in the initial comparison (indicated as *oper*) could be a data byte, a register, (HL), or an indexed address.

- Branch if accumulator is greater than other operand (signed)

```
        CP      oper        ;PERFORM COMPARISON
        JP      PE,CHRVS    ;BRANCH IF OVERFLOW OCCURRED
        JP      M,DONE      ;NO OVERFLOW - NO BRANCH ON NEGATIVE
        JR      NZ,DEST     ;  BRANCH IF RESULT NON-ZERO POSITIVE
        JR      DONE
CHRVS:  JP      M,DONE      ;BRANCH IF NEGATIVE BUT OVERFLOW
DONE:   NOP
```

This sequence forces a branch if the result is greater than 0 and overflow did not occur, or if the result is less than 0 but overflow did occur.

- Branch if accumulator is greater than or equal to other operand (signed)

```
        CP      oper        ;PERFORM COMPARISON
        JP      PE,CHRVS    ;BRANCH IF OVERFLOW OCCURRED
        JP      P,DEST      ;BRANCH IF NO OVERFLOW, POSITIVE
        JR      DONE
CHRVS:  JP      M,DEST      ;BRANCH IF OVERFLOW, NEGATIVE
DONE:   NOP
```

This sequence forces a branch if the result is greater than or equal to 0 and overflow did not occur, or if the result is less than 0 but overflow did occur.

- Branch if accumulator is less than other operand (signed)

```
        CP      oper        ;PERFORM COMPARISON
        JP      PE,CHRVS    ;BRANCH IF OVERFLOW OCCURRED
        JP      M,DEST      ;BRANCH IF NO OVERFLOW, NEGATIVE
        JR      DONE
CHRVS:  JP      P,DEST      ;BRANCH IF OVERFLOW, POSITIVE
DONE:   NOP
```

This sequence forces a branch if the result is less than 0 and overflow did not occur, or if the result is greater than or equal to 0 and overflow did occur.

- Branch if accumulator is less than or equal to other operand (signed)

```
        CP      oper        ;PERFORM COMPARISON
        JP      PE,CHRVS    ;BRANCH IF OVERFLOW OCCURRED
        JP      M,DEST      ;BRANCH IF NO OVERFLOW, NEGATIVE
        JR      Z,DEST      ;BRANCH IF NO OVERFLOW, ZERO
        JR      DONE
CHRVS:  JP      M,DONE
```

```
        JR    NZ,DEST     ;BRANCH IF OVERFLOW, POSITIVE
DONE:   NOP
```

This sequence forces a branch if the result is less than or equal to 0 and overflow did not occur, or if the result is greater than 0 and overflow did occur.

8. Branch if Higher (Unsigned).

Branch if the operands being compared are not equal and the comparison does not require a borrow. The special problem here is avoiding a branch when the operands are equal.

- Branch if (A) > VALUE (unsigned)

```
        CP    VALUE       ;COMPARE BY SUBTRACTING
        JR    C,DONE      ;NO BRANCH IF BORROW NEEDED
        JR    NZ,DEST     ;BRANCH IF NO BORROW, NOT EQUAL
DONE:   NOP
```

Comparing equal numbers clears Carry. An alternative approach is

```
        CP    VALUE+1     ;COMPARE BY SUBTRACTING VALUE+1
        JR    NC,DEST     ;BRANCH IF NO BORROW NEEDED
```

- Branch if (A) > (reg) (unsigned)

```
        CP    reg         ;COMPARE BY SUBTRACTING
        JR    C,DONE      ;NO BRANCH IF BORROW NEEDED
        JR    NZ,DEST     ;BRANCH IF NO BORROW, NOT EQUAL
DONE:   NOP
```

or

```
        LD    reg1,A      ;FORM REG - A
        LD    A,reg
        CP    reg1
        JR    NC,DEST     ;BRANCH IF BORROW NEEDED
```

or

```
        INC   reg         ;FORM A - REG - 1
        CP    reg
        JR    NC,DEST     ;BRANCH IF NO BORROW NEEDED
```

In the third alternative, we could replace INC reg with DEC A, thus changing the accumulator instead of the register.

- Branch if (A) > (ADDR) (unsigned)

```
        LD    HL,ADDR
        CP    (HL)        ;COMPARE BY SUBTRACTING
        JR    C,DONE      ;NO BRANCH IF BORROW NEEDED
        JR    NZ,DEST     ;BRANCH IF NO BORROW, NOT EQUAL
DONE:   NOP
```

or

```
        LD    reg,A           ;FORM (ADDR) - A
        LD    A,(ADDR)
        CP    reg
        JR    C,DEST          ;BRANCH IF BORROW NEEDED
```

- Branch if (HL) > (rp) (unsigned)

```
        SCF                   ;SET CARRY FLAG
        SBC   HL,rp
        JR    NC,DEST         ;BRANCH IF NO BORROW NEEDED
```

- Branch if (HL) > VAL16 (unsigned)

```
        LD    rp,-VAL16-1     ;FORM HL - VAL16 - 1
        ADD   HL,rp
        JR    C,DEST          ;BRANCH IF NO BORROW NEEDED
```

Carry is an inverted borrow here, since we are subtracting by adding the two's complement.

- Branch if (SP) > (HL) (unsigned)

```
        AND   A               ;CLEAR CARRY FLAG
        SBC   HL,SP
        JR    C,DEST
```

- Branch if (SP) > VAL16 (unsigned)

```
        LD    HL,-VAL16-1     ;FORM SP - VAL16 - 1
        ADD   HL,SP
        JR    C,DEST          ;BRANCH IF NO BORROW GENERATED
```

- Branch if (xy) > VAL16 (unsigned)

```
        LD    rp,-VAL16-1     ;FORM XY - VAL16 - 1
        ADD   xy,rp
        JR    C,DEST          ;BRANCH IF NO BORROW GENERATED
```

- Branch if (xy) > (HL) (unsigned)

```
        PUSH  xy              ;MOVE INDEX REGISTER TO REGISTER PAIR
        POP   rp
        AND   A               ;CLEAR CARRY FLAG
        SBC   HL,rp
        JR    C,DEST
```

9. Branch if Not Higher (Unsigned).

Branch if the operands being compared are equal or the comparison requires a borrow. The special problem here is forcing a branch if the operands are equal.

- Branch if (A) ≤ VALUE (unsigned)

```
        CP    VALUE           ;COMPARE BY SUBTRACTING
        JR    C,DEST          ;BRANCH IF BORROW NEEDED
        JR    Z,DEST          ;  OR IF EQUAL
```

or

```
        CP    VALUE+1          ;COMPARE BY SUBTRACTING VALUE+1
        JR    C,DEST           ;BRANCH IF BORROW NEEDED
```

- Branch if (A) ≤ (reg) (unsigned)

```
        CP    reg              ;COMPARE BY SUBTRACTING
        JR    C,DEST           ;BRANCH IF BORROW NEEDED
        JR    Z,DEST           ;  OR IF EQUAL
```

or

```
        LD    reg1,A           ;FORM REG - A
        LD    A,reg
        CP    reg1
        JR    NC,DEST          ;BRANCH IF NO BORROW NEEDED
```

or

```
        INC   reg              ;FORM A - REG - 1
        CP    reg
        JR    C,DEST           ;BRANCH IF BORROW NEEDED
```

In the third alternative, we could replace INC reg with DEC A, thus changing the accumulator instead of the register.

- Branch if (A) ≤ (ADDR) (unsigned)

```
        LD    HL,ADDR
        CP    (HL)             ;COMPARE BY SUBTRACTING
        JR    C,DEST           ;BRANCH IF BORROW NEEDED
        JR    Z,DEST           ;  OR IF EQUAL
```

or

```
        LD    reg,A            ;FORM (ADDR) - A
        LD    (ADDR),A
        CP    reg
        JR    NC,DEST          ;BRANCH IF NO BORROW NEEDED
```

- Branch if (HL) ≤ (rp) (unsigned)

```
        SCF                    ;SET CARRY FLAG
        SBC   HL,rp            ;FORM HL - RP - 1
        JR    C,DEST           ;BRANCH IF BORROW NEEDED
```

- Branch if (HL) ≤ VAL16 (unsigned)

```
        LD    rp,-VAL16-1
        ADD   HL,rp            ;FORM HL - VAL16 - 1
        JR    NC,DEST          ;BRANCH IF BORROW NEEDED
```

- Branch if (SP) ≤ (HL) (unsigned)

```
        AND   A                ;CLEAR CARRY
        SBC   HL,SP            ;FORM HL - SP
        JR    NC,DEST          ;BRANCH IF NO BORROW NEEDED
```

- Branch if (SP) ≤ VAL16 (unsigned)

```
LD    HL,-VAL16-1      ;FORM SP - VAL16 - 1
ADD   HL,SP
JR    NC,DEST          ;BRANCH IF BORROW NEEDED
```

- Branch if (xy) ≤ VAL16 (unsigned)

```
LD    rp,-VAL16-1      ;FORM XY - VAL16 - 1
ADD   xy,rp
JR    NC,DEST          ;BRANCH IF BORROW NEEDED
```

- Branch if (xy) ≤ (HL) (unsigned)

```
PUSH  xy               ;MOVE INDEX REGISTER TO REGISTER PAIR
POP   rp
AND   A                ;CLEAR CARRY
SBC   HL,rp            ;FORM HL - XY
JR    NC,DEST          ;BRANCH IF NO BORROW NEEDED
```

10. Branch if Lower (Unsigned). Branch if the unsigned comparison requires a borrow.

- Branch if (A) < VALUE (unsigned)

```
CP    VALUE            ;COMPARE BY SUBTRACTING
JR    C,DEST           ;BRANCH IF BORROW NEEDED
```

- Branch if (A) < (reg) (unsigned)

```
CP    reg              ;COMPARE BY SUBTRACTING
JR    C,DEST           ;BRANCH IF BORROW NEEDED
```

- Branch if (A) < (ADDR) (unsigned)

```
LD    HL,ADDR
CP    (HL)             ;COMPARE BY SUBTRACTING
JR    C,DEST
```

- Branch if (HL) < (rp) (unsigned)

```
AND   A                ;FORM HL - RP
SBC   HL,rp
JR    C,DEST           ;BRANCH IF BORROW NEEDED
```

- Branch if (HL) < VAL16 (unsigned)

```
LD    rp,-VAL16        ;FORM HL - VAL16
ADD   HL,rp
JR    NC,DEST          ;BRANCH IF BORROW NEEDED
```

- Branch if (SP) < (HL) (unsigned)

```
SCF                    ;FORM HL - SP-1
SBC   HL,SP
JR    NC,DEST          ;BRANCH IF NO BORROW NEEDED
```

- Branch if (SP) < VAL16 (unsigned)

```
LD    HL,-VAL16         ;FORM SP - VAL16
ADD   HL,SP
JR    NC,DEST           ;BRANCH IF NO BORROW NEEDED
```

- Branch if (xy) < VAL16 (unsigned)

```
LD    rp,-VAL16         ;FORM XY - VAL16
ADD   xy,rp
JR    NC,DEST           ;BRANCH IF NO BORROW NEEDED
```

- Branch if (xy) < (HL) (unsigned)

```
PUSH  xy                ;MOVE INDEX REGISTER TO REGISTER PAIR
POP   rp
SCF                     ;FORM HL - XY-1
SBC   HL,rp
JR    NC,DEST           ;BRANCH IF NO BORROW NEEDED
```

11. Branch if Not Lower (Unsigned). Branch if the unsigned comparison does not require a borrow.

- Branch if (A) ≥ VALUE (unsigned)

```
CP    VALUE             ;COMPARE BY SUBTRACTING
JR    NC,DEST           ;BRANCH IF NO BORROW NEEDED
```

- Branch if (A) ≥ (reg) (unsigned)

```
CP    reg               ;COMPARE BY SUBTRACTING
JR    NC,DEST           ;BRANCH IF NO BORROW NEEDED
```

- Branch if (A) ≥ (ADDR) (unsigned)

```
LD    HL,ADDR
CP    (HL)              ;COMPARE BY SUBTRACTING
JR    NC,DEST           ;BRANCH IF NO BORROW NEEDED
```

- Branch if (HL) ≥ (rp) (unsigned)

```
AND   A                 ;FORM HL - RP
SBC   HL,rp
JR    NC,DEST           ;BRANCH IF NO BORROW NEEDED
```

- Branch if (HL) ≥ VAL16 (unsigned)

```
LD    rp,-VAL16         ;FORM HL - VAL16
ADD   HL,rp
JR    C,DEST            ;BRANCH IF NO BORROW NEEDED
```

- Branch if (SP) ≥ (HL) (unsigned)

```
SCF                     ;FORM HL - SP-1
SBC   HL,SP
JR    C,DEST            ;BRANCH IF BORROW NEEDED
```

- Branch if (SP) ≥ VAL16 (unsigned)

```
        LD   HL,-VAL16 ;FORM SP - VAL16
        ADD  HL,rp
        JR   C,DEST    ;BRANCH IF NO BORROW NEEDED
```

- Branch if (xy) ≥ VAL16 (unsigned)

```
        LD   rp,-VAL16 ;FORM XY - VAL16
        ADD  xy,SP
        JR   C,DEST    ;BRANCH IF NO BORROW NEEDED
```

- Branch if (xy) ≥ (HL) (unsigned)

```
        PUSH xy        ;TRANSFER INDEX REG TO REGISTER PAIR
        POP  rp
        SCF            ;FORM HL - XY - 1
        SBC  HL,rp
        JR   C,DEST    ;BRANCH IF BORROW NEEDED
```

SKIP INSTRUCTIONS

Skip instructions can be implemented on the Z80 microprocessor by using jump instructions with the proper destination. That destination should be one instruction beyond the one that follows the jump sequentially. The actual number of bytes skipped will vary, since Z80 instructions vary from one to four bytes in length.

SUBROUTINE CALL INSTRUCTIONS

Unconditional Call Instructions

An indirect call on the Z80 microprocessor can be implemented by calling a routine that performs an indirect jump. An RET instruction at the end of the subroutine will then transfer control back to the original calling point. The main program performs

```
CALL TRANS
```

where subroutine TRANS transfers control to the ultimate destination. Note that TRANS ends with a jump, not with a return. Typical TRANS routines are

- To address in HL

```
TRANS:    JP    (HL)       ;ENTRY POINT IN HL
```

- To address in an index register

```
TRANS:    JP    (xy)       ;ENTRY POINT IN AN INDEX REGISTER
```

· To address in DE

```
TRANS:          EX    DE,HL      ;ENTRY POINT IN DE
                JP    (HL)
```

· To address in BC

```
TRANS:          LD    H,B        ;ENTRY POINT IN BC
                LD    L,C
                JP    (HL)
```

or

```
TRANS:          PUSH  BC         ;ENTRY POINT IN BC
                RET
```

The second alternative is longer, but leaves HL unchanged.

· To address in memory locations ADDR and ADDR+1

```
TRANS:          LD    HL,(ADDR)  ;ENTRY POINT AT ADDR
                JP    (HL)
```

· To address at the top of the stack. Here we must exchange the return address with the top of the stack. This can be done in the main program as follows:

```
                LD    HL,RETPT   ;GET RETURN POINT ADDRESS
                EX    HL,(SP)    ;PUT RETURN ADDRESS ON STACK
                JP    (HL)       ;AND JUMP TO OLD TOP OF STACK
```

The exchange can allow later resumption of a suspended program or provide a special exit to an error-handling routine.

You can implement indexed calls in the same way as indirect calls. The CALL instruction transfers control to a routine that performs an indexed jump as shown earlier. That routine ends with an ordinary jump instruction (typically JP (HL)) that does not affect the stack. An RET instruction at the end of the actual subroutine will therefore transfer control back to the original calling point.

If the main program executes CALL JMPIND with the index in the accumulator and the starting address of the jump table in register pair HL, the indexed jump routine is

```
JMPIND:         ADD   A,A        ;DOUBLE INDEX FOR 2-BYTE ENTRIES
                LD    E,A        ;EXTEND INDEX TO 16 BITS
                LD    D,0
                ADD   HL,DE      ;CALCULATE ADDRESS OF ELEMENT
                LD    E,(HL)     ;FETCH ELEMENT FROM ADDRESS TABLE
                INC   HL
                LD    D,(HL)
                EX    DE,HL      ;AND JUMP TO IT
                JP    (HL)
```

One problem with indexed and indirect calls is that the transfer routines may interfere with the subroutines. For example, the indexed jump routine JMPIND changes the accumulator, register pair DE, register pair HL, and the flags. Thus, none of these registers can be used to pass parameters to the subroutine. The programmer must always remember that the intermediate transfer routines are interposed between the main program and the actual subroutine. A similar interposition occurs when operating system routines transfer control from one task to another or from a main program to an I/O driver or an interrupt service routine.

Conditional Call Instructions

Conditional calls can be implemented on the Z80 by using the sequences shown for conditional branches. The only change is that jumps to the actual destination must be replaced with calls (for example, replace JR NZ,DEST with CALL NZ,DEST or JP P,DEST with CALL P,DEST).

SUBROUTINE RETURN INSTRUCTIONS

Unconditional Return Instructions

The RET instruction returns control automatically to the address saved at the top of the stack. If the return address is saved elsewhere (for example, in a register pair or in two fixed memory locations) you can transfer control to it by performing an indirect jump.

Conditional Return Instructions

Conditional returns can be implemented on the Z80 microprocessor by using the sequences shown earlier for conditional branches. The only change is that you must replace jumps to the actual destination with RETs (for example, replace JR NC, DEST with RET NC or JP M,DEST with RET M).

Return with Skip Instructions

· Return control to the address at the top of the stack after it has been incremented by an offset NEXT. This sequence lets you transfer control past parameters, data, and other non-executable items.

```
POP   DE              ;GET RETURN ADDRESS
LD    HL,NEXT         ;OFFSET TO NEXT EXECUTABLE INSTRUCTION
ADD   HL,DE
JP    (HL)            ;AND RETURN
```

· Change the return address to RETPT. Assume that the return address is currently stored at the top of the stack.

```
LD    HL,RETPT         ;CHANGE RETURN ADDRESS TO RETPT
EX    HL,(SP)
```

EX HL,(SP) exchanges HL with the top of the stack. This procedure allows you to force a special exit to an error routine or other exception-handling program without changing the logic of the subroutine or losing track of the original return address.

Return from Interrupt Instructions

If the initial portion of the interrupt service routine saves all the primary registers and the index registers with the sequence

```
PUSH AF                ;SAVE PRIMARY REGISTERS
PUSH BC
PUSH DE
PUSH HL
PUSH IX                ;SAVE INDEX REGISTERS
PUSH IY
```

a standard return sequence is

```
POP   IY               ;RESTORE INDEX REGISTERS
POP   IX
POP   HL               ;RESTORE PRIMARY REGISTERS
POP   DE
POP   BC
POP   AF
EI                     ;REENABLE INTERRUPTS
RETI
```

The order of restoration is the opposite of the order in which the registers were saved. The instruction EI must come immediately before RETI to avoid unnecessary stacking of return addresses.

MISCELLANEOUS INSTRUCTIONS

In this category, we include no operations, push, pop, halt, wait, trap (break or software interrupt), decimal adjust, enabling and disabling of interrupts, translation (table lookup), and other instructions that do not fall into any of the earlier categories.

1. No Operation Instructions.

Like NOP itself, any LD instruction with the same source and destination register does nothing except advance the program counter. These additional no-ops are

```
LD   A,A
LD   B,B
LD   C,C
LD   D,D
LD   E,E
LD   H,H
LD   L,L
```

2. Push Instructions.

   · Push a single register (A, B, D, or H)

```
PUSH rp              ;PUSH THE REGISTER PAIR
INC  SP              ;BUT DROP THE LESS SIGNIFICANT HALF
```

The register pair could be AF. Programmers generally prefer to combine byte-length operands or simply waste a byte of the stack rather than attempt to push a single byte.

   · Push memory location ADDR

```
LD   A,(ADDR)        ;OBTAIN DATA FROM MEMORY
PUSH AF              ;PUSH DATA, FLAGS
INC  SP              ;THEN DROP THE FLAGS
```

ADDR could be an external priority or control register (or a copy of an external register).

   · Push memory locations ADDR and ADDR+1

```
LD   HL,(ADDR)       ;PUSH A PAIR OF MEMORY LOCATIONS
PUSH HL
```

   · Push the interrupt flip-flop IFF2

```
LD   A,I             ;MOVE IFF2 TO PARITY/OVERFLOW FLAG
PUSH AF
```

This sequence allows you to save the interrupt status in the Parity/Overflow flag (bit 2 of register F) for later restoration.

3. Pop (Pull) Instructions.

   · Pop a single register (A, B, D, or H), assuming that it has been saved as shown previously

```
DEC  SP              ;BACK UP THE STACK POINTER
POP  rp              ;POP THE REGISTER PAIR
```

This sequence changes the less significant half of the register pair unpredictably.

   · Pop memory location ADDR, assuming that it has been saved at the top of the stack

```
DEC  SP              ;BACK UP THE STACK POINTER
POP  AF              ;POP ACCUMULATOR AND FLAGS
LD   (ADDR),A        ;RESTORE DATA TO MEMORY
```

This sequence changes the flags unpredictably. ADDR could be an external priority or control register (or a copy of an external register).

· Pop memory locations ADDR and ADDR+1, assuming that they were saved as shown previously

```
        POP   HL          ;RESTORE A PAIR OF MEMORY LOCATIONS
        LD    (ADDR),HL
```

Sometimes you must push and pop key memory locations and other values beside the registers.

· Restore interrupt status, assuming that it has been saved at the top of the stack.

```
        POP   AF          ;OBTAIN PREVIOUS INTERRUPT STATUS
        JP    PE,ENABLE
        DI                ;DISABLE INTERRUPTS IF PREVIOUSLY SO
        JR    DONE
ENABLE: EI                ;ENABLE INTERRUPTS IF PREVIOUSLY SO
DONE:   NOP
```

The interrupt flip-flop IFF2 is saved in the Parity/Overflow flag; interrupts were previously enabled if that flag is 1 and disabled if it is 0.

Wait Instructions

The simplest way to implement a wait on the Z80 microprocessor is to use an endless loop such as

```
HERE:   JP    HERE
```

The processor will execute JP until it is interrupted and will resume executing it after the interrupt service routine returns control. Of course, regular interrupts must have been enabled (with EI) or the processor will execute the endless loop indefinitely. The non-maskable interrupt can interrupt at any time without being enabled.

Trap Instructions

The common Z80 traps (also called breaks or software interrupts) are the RST instructions (see the list in Table 1-9). RST n calls the subroutine starting at address n. Thus, for example, RST 0 transfers control to memory address 0000 after saving the current program counter in the stack. Similarly, RST 30H transfers control to memory address 0030_{16} after saving the current program counter in the stack. The interrupt system generally uses the RST instructions, but the programmer can dedicate unused ones to common subroutines, error traps, or supervisor entry points. RST then serves as a 1-byte call.

Adjust Instructions

1. Branch if accumulator does not contain a valid decimal (BCD) number.

```
LD    reg,A          ;SAVE COPY OF ACCUMULATOR
ADD   A,0            ;THEN DECIMAL ADJUST ACCUMULATOR
DAA
CMP   reg            ;DID DECIMAL ADJUST CHANGE A?
JR    NZ,DEST        ;YES, A WAS NOT DECIMAL
```

2. Decimal increment accumulator (add 1 to A in decimal).

```
ADD   A,1            ;ADD 1 IN DECIMAL
DAA
```

3. Decimal decrement accumulator (subtract 1 from A in decimal).

```
SUB   1              ;SUBTRACT 1 IN DECIMAL
DAA
```

or

```
ADD   A,99H          ;SUBTRACT 1 BY ADDING 99
DAA
```

The second alternative is compatible with the 8080 and 8085 processors, where DAA works properly only after addition instructions.

Enable and Disable Interrupt Instructions

1. Enable interrupts but save previous value of interrupt flip-flop 2 (the interrupt status).

```
LD    A,I            ;MOVE INTERRUPT FLIP-FLOP TO P/V FLAG
PUSH  AF             ;SAVE OLD IFF2 IN STACK
EI                   ;THEN ENABLE INTERRUPTS
```

2. Disable interrupts but save previous value of interrupt flip-flop 2 (the interrupt status).

```
LD    A,I            ;MOVE INTERRUPT FLIP-FLOP TO P/V FLAG
PUSH  AF             ;SAVE OLD IFF2 IN STACK
DI                   ;THEN DISABLE INTERRUPTS
```

3. Restore interrupt status, assuming that it is currently saved in the Parity/Overflow flag at the top of the stack.

```
POP   AF             ;OBTAIN PREVIOUS INTERRUPT STATUS
JP    PE,ENABLE      ;WERE INTERRUPTS ENABLED ORIGINALLY?
```

```
            DI                  ;NO, THEN DISABLE THEM NOW
            JR    DONE
ENABLE:     EI                  ;YES, THEN ENABLE THEM NOW
DONE:       NOP
```

After LD A,I or LD A,R, JP PE means "branch if interrupts are enabled," while JP PO means "branch if interrupts are disabled."

Translate Instructions

1. Translate the accumulator into the corresponding entry in a table starting at the address in register pair HL.

```
        LD    E,A       ;EXTEND OPERAND TO 16-BIT INDEX
        LD    D,0
        ADD   HL,DE     ;USE OPERAND TO ACCESS TABLE
        LD    A,(HL)    ;REPLACE OPERAND WITH TABLE ENTRY
```

This procedure can be used to convert data from one code to another.

2. Translate the accumulator into the corresponding 16-bit entry in a table starting at the address in register pair HL. Place the entry in HL.

```
        EX    DE,HL     ;MOVE STARTING ADDRESS TO DE
        LD    L,A       ;EXTEND OPERAND TO 16-BIT INDEX
        LD    H,0
        ADD   HL,HL     ;DOUBLE INDEX FOR 2-BYTE ENTRIES
        ADD   HL,DE     ;CALCULATE INDEXED ADDRESS
        LD    E,(HL)    ;OBTAIN ENTRY
        INC   HL
        LD    D,(HL)
        EX    DE,HL     ;MOVE ENTRY TO HL
```

Using ADD HL,HL to double the operand allows it to take on any 8-bit value (using ADD A,A would limit us to values below 128).

Miscellaneous Instructions

1. Allocate space on the stack; decrease the stack pointer to provide NUM empty locations at the top.

```
        LD    HL,-NUM   ;ADD NUM EMPTY BYTES TO TOP OF STACK
        ADD   HL,SP
        LD    SP,HL     ;SP = SP - NUM
```

An alternative is a series of DEC SP instructions.

2. Deallocate space from the stack; increase the stack pointer to remove NUM temporary locations from the top.

```
        LD    HL,NUM          ;DELETE NUM BYTES FROM STACK
        ADD   HL,SP
        LD    SP,HL           ;SP = SP + NUM
```

An alternative is a series of INC SP instructions.

ADDITIONAL ADDRESSING MODES

· **Indirect Addressing**. Indirect addressing can be provided on the Z80 processor by loading the indirect address into register pair HL. Then addressing through HL provides the equivalent of true indirect addressing. This is a two-step process that generally requires HL, although BC or DE can be employed to load and store the accumulator. The index registers may also be used, although at the cost of extra execution time and memory. Note that indexed addressing with a 0 offset is simply a slow version of indirect addressing.

Examples

1. Load the accumulator indirectly from the address in memory locations ADDR and ADDR+1.

```
        LD    HL,(ADDR)       ;FETCH INDIRECT ADDRESS
        LD    A,(HL)          ;FETCH DATA INDIRECTLY
```

or

```
        LD    xy,(ADDR)       ;FETCH INDIRECT ADDRESS
        LD    A,(xy+0)        ;FETCH DATA INDIRECTLY
```

2. Store the accumulator indirectly at the address in memory locations ADDR and ADDR+1.

```
        LD    HL,(ADDR)       ;FETCH INDIRECT ADDRESS
        LD    (HL),A          ;STORE DATA INDIRECTLY
```

or

```
        LD    xy,(ADDR)       ;FETCH INDIRECT ADDRESS
        LD    (xy+0),A        ;STORE DATA INDIRECTLY
```

3. Load the accumulator indirectly from the address in register pair HL (that is, from the address stored starting at the address in HL).

```
        LD    E,(HL)          ;FETCH INDIRECT ADDRESS
        INC   HL
        LD    D,(HL)
        LD    A,(DE)          ;FETCH DATA INDIRECTLY
```

4. Load the accumulator indirectly from the address in an index register (that is, from the address stored starting at the address in an index register).

```
        LD    L,(xy+0)        ;FETCH INDIRECT ADDRESS
        LD    H,(xy+1)
        LD    A,(HL)          ;FETCH DATA INDIRECTLY
```

5. Store the accumulator indirectly at the address in register pair HL (that is, at the address stored starting at the address in HL).

```
        LD    E,(HL)          ;FETCH INDIRECT ADDRESS
        INC   HL
        LD    D,(HL)
        LD    (DE),A          ;STORE DATA INDIRECTLY
```

6. Store the accumulator indirectly at the address in an index register (that is, at the address stored starting at the address in an index register).

```
        LD    L,(xy+0)        ;FETCH INDIRECT ADDRESS
        LD    H,(xy+1)
        LD    (HL),A          ;STORE DATA INDIRECTLY
```

7. Jump indirectly to the address in memory locations ADDR and ADDR+1.

```
        LD    HL,(ADDR)       ;FETCH INDIRECT ADDRESS
        JP    (HL)            ;AND TRANSFER CONTROL TO IT
```

or

```
        LD    xy,(ADDR)       ;FETCH INDIRECT ADDRESS
        JP    (xy)            ;AND TRANSFER CONTROL TO IT
```

Indirection can be repeated indefinitely to provide multi-level indirect addressing. For example, the following routine uses the indirect address indirectly to load the accumulator:

```
        LD    E,(HL)          ;FETCH FIRST INDIRECT ADDRESS
        INC   HL
        LD    D,(HL)
        EX    DE,HL
        LD    E,(HL)          ;USE INDIRECT ADDRESS INDIRECTLY
        INC   HL
        LD    D,(HL)
        LD    A,(DE)          ;FETCH DATA INDIRECTLY
```

Indirect addresses should be stored in memory in the usual Z80 format—that is, with the less significant byte first (at the lower address).

· **Indexed Addressing.** Indexed addressing can be provided by using ADD HL to add the base and the index. Obviously, the explicit addition requires extra execution time. The index registers are useful when the index is fixed (as in a data structure) or when HL is already occupied.

Examples

1. Load the accumulator from an indexed address obtained by adding the accumulator to a fixed base address.

```
        LD    DE,BASE          ;GET BASE ADDRESS
        LD    L,A              ;EXTEND INDEX TO 16 BITS
        LD    H,0
        ADD   HL,DE            ;CALCULATE INDEXED ADDRESS
        LD    A,(HL)           ;FETCH DATA FROM INDEXED ADDRESS
```

2. Load the accumulator from an indexed address obtained by adding the accumulator to memory locations BASE and BASE+1.

```
        LD    HL,(BASE)        ;GET BASE ADDRESS
        LD    E,A              ;EXTEND INDEX TO 16 BITS
        LD    D,0
        ADD   HL,DE            ;CALCULATE INDEXED ADDRESS
        LD    A,(HL)           ;FETCH DATA FROM INDEXED ADDRESS
```

3. Load the accumulator from an indexed address obtained by adding memory locations INDEX and INDEX+1 to register pair HL.

```
        LD    DE,(INDEX)       ;GET INDEX FROM MEMORY
        ADD   HL,DE            ;CALCULATE INDEXED ADDRESS
        LD    A,(HL)           ;FETCH DATA FROM INDEXED ADDRESS
```

4. Jump indexed to a jump instruction in a list. The index is in the accumulator and the base address of the list is in register pair HL.

```
        LD    B,A              ;MULTIPLY INDEX TIMES 3
        ADD   A,A
        ADD   B,A
        LD    C,A              ;EXTEND INDEX TO 16 BITS
        LD    B,0
        ADD   HL,BC            ;CALCULATE INDEXED ADDRESS
        JP    (HL)             ;AND TRANSFER CONTROL THERE
```

The following is a typical list starting at address BASE:

```
BASE:   JP    SUB0             ;JUMP TO SUBROUTINE 0
        JP    SUB1             ;JUMP TO SUBROUTINE 1
        JP    SUB2             ;JUMP TO SUBROUTINE 2
        ...
```

Since each JP instruction occupies three bytes, we must multiply the index by 3 before adding it to the base address. If the list is more than 256 bytes long, we can use the following procedure to multiply the index by 3:

```
        EX    DE,HL            ;SAVE BASE ADDRESS IN DE
        LD    L,A              ;EXTEND INDEX TO 16 BITS
        LD    H,0
        LD    B,L              ;COPY INDEX INTO BC
```

```
LD    C,H
ADD   HL,HL            ;DOUBLE INDEX
ADD   HL,BC            ;TRIPLE INDEX
ADD   HL,DE            ;CALCULATE INDEXED ADDRESS
JP    (HL)             ;AND TRANSFER CONTROL THERE
```

· **Autopreincrementing.** In autopreincrementing, the address register is incremented automatically before it is used. Autopreincrementing can be provided on the Z80 by incrementing a register pair before using it as an address.

Examples

· Load the accumulator using autopreincrementing on register pair HL.

```
INC   HL               ;AUTOPREINCREMENT HL
LD    A,(HL)           ;FETCH DATA
```

· Store the accumulator using autopreincrementing on register pair DE.

```
INC   DE               ;AUTOPREINCREMENT DE
LD    (DE),A           ;STORE DATA
```

· Load register pair DE starting at the address two larger than the contents of HL.

```
INC   HL               ;AUTOPREINCREMENT HL BY 2
INC   HL
LD    E,(HL)           ;FETCH LSB
INC   HL
LD    D,(HL)           ;FETCH MSB
```

Autoincrementing by 2 is essential in handling arrays of addresses or 16-bit data items.

· Store the accumulator using autopreincrementing on memory locations ADDR and ADDR+1.

```
LD    HL,(ADDR)        ;AUTOPREINCREMENT INDIRECT ADDRESS
INC   HL
LD    (HL),A           ;STORE DATA
LD    (ADDR),HL        ;UPDATE INDIRECT ADDRESS
```

Autopreincrementing can be combined with indirection. Here memory locations ADDR and ADDR+1 could point to the last occupied location in a buffer.

· Transfer control to the address stored starting at an address two larger than the contents of memory locations NXTPGM and NXTPGM+1.

```
LD    HL,(NXTPGM)      ;GET POINTER
INC   HL               ;AUTOPREINCREMENT POINTER
INC   HL
LD    (NXTPGM),HL      ;UPDATE POINTER
LD    E,(HL)           ;FETCH STARTING ADDRESS
INC   HL
```

```
        LD    D,(HL)
        EX    DE,HL          ;AND TRANSFER CONTROL TO IT
        JP    (HL)
```

Here NXTPGM and NXTPGM+1 point to the starting address of the routine that the processor has just executed. Initially, NXTPGM and NXTPGM+1 would contain BASE-2, where BASE is the starting address of a table of routines. A typical table would be

```
BASE:   DW    ROUT0          ;STARTING ADDRESS FOR ROUTINE 0
        DW    ROUT1          ;STARTING ADDRESS FOR ROUTINE 1
        DW    ROUT2          ;STARTING ADDRESS FOR ROUTINE 2
        DW    ROUT3          ;STARTING ADDRESS FOR ROUTINE 3
        ....
```

· **Autopostincrementing.** In autopostincrementing, the address register is incremented after it is used. Autopostincrementing can be provided on the Z80 by incrementing a register pair after using it as an address. Note that the Z80 autopostincrements the stack pointer when it executes POP and RET.

Examples

· Load the accumulator using autopostincrementing on register pair HL.

```
        LD    A,(HL)         ;FETCH DATA
        INC   HL             ;AUTOPOSTINCREMENT HL
```

· Store the accumulator using autopostincrementing on register pair DE.

```
        LD    (DE),A         ;STORE DATA
        INC   DE             ;AUTOPOSTINCREMENT DE
```

· Load register pair DE starting at the address in HL. Then increment HL by 2.

```
        LD    E,(HL)         ;FETCH LSB
        INC   HL
        LD    D,(HL)         ;FETCH MSB
        INC   HL
```

Autoincrementing by 2 is essential in handling arrays of addresses or 16-bit data items. Note that postincrementing is generally simpler and more natural than preincrementing.

· Store the accumulator using autopostincrementing on memory locations ADDR and ADDR+1.

```
        LD    HL,(ADDR)      ;FETCH INDIRECT ADDRESS
        LD    (HL),A         ;STORE DATA
        INC   HL             ;AUTOPOSTINCREMENT INDIRECT ADDRESS
        LD    (ADDR),HL
```

· Autopostincrementing can be combined with indirection. Here memory locations ADDR and ADDR+1 could point to the next empty location in a buffer.

· Transfer control to the address stored at the address in memory locations NXTPGM and NXTPGM+1. Then increment those locations by 2.

```
        LD   HL,(NXTPGM)
        LD   E,(HL)          ;FETCH STARTING ADDRESS
        INC  HL
        LD   D,(HL)
        INC  HL              ;COMPLETE AUTOPOSTINCREMENT
        LD   (NXTPGM),HL
        EX   DE,HL           ;TRANSFER CONTROL TO START ADDRESS
        JP   (HL)
```

Here NXTPGM and NXTPGM+1 point to the starting address of the next routine the processor is to execute. Initially, NXTPGM and NXTPGM+1 would contain BASE, the starting address of a table of routines. A typical table would be

```
BASE:   DW   ROUT0           ;STARTING ADDRESS FOR ROUTINE 0
        DW   ROUT1           ;STARTING ADDRESS FOR ROUTINE 1
        DW   ROUT2           ;STARTING ADDRESS FOR ROUTINE 2
        DW   ROUT3           ;STARTING ADDRESS FOR ROUTINE 3
```

· **Autopredecrementing.** In autopredecrementing, the address register is decremented automatically before it is used. Autopredecrementing can be provided on the Z80 processor by decrementing a register pair before using it as an address. Note that the processor autopredecrements the stack pointer when it executes PUSH and CALL.

Examples

· Load the accumulator using autopredecrementing on register pair HL.

```
        DEC  HL              ;AUTOPREDECREMENT HL
        LD   A,(HL)          ;FETCH DATA
```

· Store the accumulator using autopredecrementing on register pair DE.

```
        DEC  DE              ;AUTOPREDECREMENT DE
        LD   (DE),A          ;STORE DATA
```

· Load register pair DE starting at the address two smaller than the contents of HL.

```
        DEC  HL              ;FETCH MSB
        LD   D,(HL)
        DEC  HL              ;FETCH LSB
        LD   E,(HL)
```

Autodecrementing by 2 is essential in handling arrays of addresses or 16-bit data items. Note that predecrementing is generally simpler and more natural than postdecrementing.

· Store the accumulator using autopredecrementing on memory locations ADDR and ADDR+1.

```
        LD    HL,(ADDR)        ;AUTOPREINCREMENT INDIRECT ADDRESS
        DEC   HL
        LD    (HL),A           ;STORE DATA
        LD    (ADDR),HL        ;UPDATE INDIRECT ADDRESS
```

Autodecrementing can be combined with indirection. Here memory locations ADDR and ADDR+1 could point to the last occupied location in a stack.

· Transfer control to the address stored at an address two smaller than the contents of memory locations NXTPGM and NXTPGM+1.

```
        LD    HL,(NXTPGM)      ;FETCH STARTING ADDRESS
        DEC   HL
        LD    D,(HL)
        DEC   HL
        LD    E,(HL)
        LD    (NXTPGM),HL      ;STORE AUTOPREDECREMENTED POINTER
        EX    DE,HL            ;TRANSFER CONTROL TO START ADDRESS
        JP    (HL)
```

Here NXTPGM and NXTPGM+1 point to the starting address of the most recently executed routine in a list. Initially, NXTPGM and NXTPGM+1 would contain FINAL+2, where FINAL is the address of the last entry in a table of routines. A typical table would be

```
        DW    ROUTO            ;STARTING ADDRESS FOR ROUTINE 0
        DW    ROUT1            ;STARTING ADDRESS FOR ROUTINE 1
        .
        .
        .
FINAL:  DW    ROUTL            ;STARTING ADDRESS FOR LAST ROUTINE
```

Here we work through the table backward. This approach is useful in evaluating mathematical formulas entered from a keyboard. If, for example, the computer must evaluate the expression

```
Z = LN (A x SIN (B x EXP(C x Y)))
```

it must work backward. That is, the order of operations is

· Calculate C x Y

· Calculate EXP (C x Y)

· Calculate B x EXP(C x Y)

· Calculate SIN (B x EXP(C x Y))

· Calculate A x SIN (B x EXP(C x Y))

· Calculate LN(A x SIN(B x EXP(C x Y))) .

Working backward is convenient when the computer cannot start a task until it has received an entire line or command. It must then work back to the beginning.

· **Autopostdecrementing.** In autopostdecrementing, the address register is decremented automatically after it is used. Autopostdecrementing can be implemented on the Z80 by decrementing a register pair after using it as an address.

Examples

· Load the accumulator using autopostdecrementing on register pair HL.

```
LD   A,(HL)          ;FETCH DATA
DEC  HL              ;AUTOPOSTDECREMENT HL
```

· Store the accumulator using autopostdecrementing on register pair DE.

```
LD   (DE),A          ;STORE DATA
DEC  DE              ;AUTOPOSTDECREMENT DE
```

· Load register pair DE starting at the address in HL. Afterward, decrement HL by 2.

```
INC  HL              ;FETCH MSB
LD   D,(HL)
DEC  HL              ;FETCH LSB
LD   E,(HL)
DEC  HL              ;AUTOPOSTDECREMENT HL BY 2
DEC  HL
```

Autodecrementing by 2 is essential in handling arrays of addresses or 16-bit data items.

· Store the accumulator using autopostdecrementing on memory locations ADDR and ADDR+1.

```
LD   HL,(ADDR)       ;FETCH INDIRECT ADDRESS
LD   (HL),A          ;STORE DATA
DEC  HL              ;AUTOPOSTDECREMENT INDIRECT ADDRESS
LD   (ADDR),HL
```

Autopostdecrementing can be combined with indirection. Here memory locations ADDR and ADDR+1 could point to the next empty location in a buffer.

· Transfer control to the address stored at the address in memory locations NXTPGM and NXTPGM + 1. Then decrement those locations by 2.

```
LD   HL,(NXTPGM)     ;FETCH POINTER
INC  HL              ;FETCH STARTING ADDRESS
LD   D,(HL)
DEC  HL
LD   E,(HL)
DEC  HL              ;AUTOPOSTDECREMENT POINTER
DEC  HL
LD   (NXTPGM),HL
EX   DE,HL           ;JUMP TO STARTING ADDRESS
JP   (HL)
```

Here NXTPGM and NXTPGM+1 point to the starting address of the next routine the processor is to execute. Initially, NXTPGM and NXTPGM+1 contain FINAL, the address of the last entry in a table of routines. A typical table would be

```
        DW    ROUT0           ;STARTING ADDRESS OF ROUTINE 0
        DW    ROUT1           ;STARTING ADDRESS OF ROUTINE 1
        .
        .
        .
FINAL:  DW    ROUTL           ;STARTING ADDRESS OF LAST ROUTINE
```

Here the computer works through the table backward. This approach is useful in interpreting commands entered in the normal left-to-right manner from a keyboard. For example, assume that the operator of a process controller enters the command SET TEMP(POSITION 2) = MEAN(TEMP(POSITION 1), TEMP(POSITION 3)). The controller program must execute the command working right-to-left and starting from inside the inner parentheses as follows:

1. Determine the index corresponding to POSITION 1.
2. Obtain TEMP(POSITION 1) from a table of temperature readings.
3. Determine the index corresponding to POSITION 3.
4. Obtain TEMP(POSITION 3) from a table of temperature readings.
5. Evaluate MEAN(TEMP(POSITION 1), TEMP(POSITION 3)) by executing the MEAN program with the two entries as data.
6. Determine the index corresponding to POSITION 2.
7. Execute the SET function, which presumably involves setting controls and parameters to achieve the desired value of TEMP (POSITION 2).

The operator enters the command working left to right and from outer parentheses to inner parentheses. The computer, on the other hand, must execute it inside out (starting from the inner parentheses) and right to left. Autodecrementing is obviously a handy way to implement this reversal.

· **Indirect preindexed addressing (preindexing).** In preindexing, the processor must first calculate an indexed address and then use that address indirectly. Since the indexed table must consist of 2-byte indirect addresses, the indexing must involve a multiplication by 2.

Examples

· Load the accumulator using preindexing. The base address is in an index register and the index is a constant INDEX.

```
LD    L,(xy+2*INDEX)       ;OBTAIN LSB OF ADDRESS
LD    H,(xy+2*INDEX+1)     ;OBTAIN MSB OF ADDRESS
LD    A,(HL)               ;OBTAIN DATA INDIRECTLY
```

Because of the limitations of Z80 indexing, this approach works only when INDEX is a constant.

· Load the accumulator using preindexing. The base address is in register pair HL and the index is in the accumulator.

```
        ADD   A,A                 ;DOUBLE INDEX FOR 2-BYTE ENTRIES
        LD    E,A                 ;EXTEND INDEX TO 16 BITS
        LD    D,0
        ADD   HL,DE               ;CALCULATE INDEXED ADDRESS
        LD    E,(HL)              ;OBTAIN INDIRECT ADDRESS
        INC   HL
        LD    D,(HL)
        LD    A,(DE)              ;OBTAIN DATA INDIRECTLY
```

· Store the accumulator using preindexing. The base address is in memory locations ADDR and ADDR+1 and the index is a constant INDEX.

```
        LD    xy,(ADDR)           ;OBTAIN BASE ADDRESS
        LD    L,(xy+2*INDEX)      ;OBTAIN INDIRECT ADDRESS
        LD    H,(xy+2*INDEX+1)
        LD    (HL),A              ;STORE DATA INDIRECTLY
```

· Store the accumulator using preindexing. The base address is in memory locations ADDR and ADDR+1 and the index is in memory location INDEX.

```
        LD    HL,(ADDR)           ;FETCH BASE ADDRESS
        LD    B,A                 ;SAVE DATA
        LD    A,(INDEX)           ;FETCH INDEX
        ADD   A,A                 ;DOUBLE INDEX FOR 2-BYTE ENTRIES
        LD    E,A                 ;EXTEND INDEX TO 16 BITS
        LD    D,0
        ADD   HL,DE               ;CALCULATE INDEXED ADDRESS
        LD    E,(HL)              ;OBTAIN INDIRECT ADDRESS
        INC   HL
        LD    D,(HL)
        EX    DE,HL               ;STORE DATA INDIRECTLY
        LD    (HL),B
```

· Transfer control (jump) to the address obtained indirectly from the table starting at address JTAB. The index is in the accumulator.

```
        ADD   A,A                 ;DOUBLE INDEX FOR 2-BYTE ENTRIES
        LD    E,A                 ;EXTEND INDEX TO 16 BITS
        LD    D,0
        LD    HL,JTAB             ;GET BASE ADDRESS
        ADD   HL,DE               ;CALCULATE INDEXED ADDRESS
        LD    E,(HL)              ;OBTAIN INDIRECT ADDRESS
        INC   HL
        LD    D,(HL)
        EX    DE,HL               ;JUMP TO INDIRECT ADDRESS
        JP    (HL)
```

The table starting at address JTAB would appear as follows:

```
JTAB:       DW    ROUT0            ;STARTING ADDRESS OF ROUTINE 0
            DW    ROUT1            ;STARTING ADDRESS OF ROUTINE 1
            DW    ROUT2            ;STARTING ADDRESS OF ROUTINE 2
            ....
```

· **Indirect postindexed addressing (postindexing).** In postindexing, the processor must first obtain an indirect address and then apply indexing with that address as the base. Thus the indirect address tells the processor where the table or array starts.

Examples

· Load a register using postindexing. The base address is in memory locations ADDR and ADDR+1 and the index is a constant OFFSET.

```
        LD    xy,(ADDR)        ;OBTAIN BASE ADDRESS INDIRECTLY
        LD    reg,(xy+OFFSET);OBTAIN DATA
```

This approach is useful when ADDR and ADDR+1 contain the base address of a data structure and OFFSET is the fixed distance from the base address to a particular data item.

· Load the accumulator using postindexing. The base address is in memory locations ADDR and ADDR+1 and the index is in the accumulator.

```
        LD    HL,(ADDR)        ;OBTAIN BASE ADDRESS INDIRECTLY
        LD    E,A              ;EXTEND INDEX TO 16 BITS
        LD    D,0
        ADD   HL,DE            ;CALCULATE INDEXED ADDRESS
        LD    A,(HL)           ;OBTAIN DATA
```

· Store a register using postindexing. The base address is in memory locations ADDR and ADDR+1 and the index is a constant OFFSET.

```
        LD    xy,(ADDR)        ;OBTAIN BASE ADDRESS INDIRECTLY
        LD    (xy+OFFSET),reg;STORE DATA POSTINDEXED
```

· Store the accumulator using postindexing. The base address is in memory locations ADDR and ADDR+1 and the index is in memory location INDEX.

```
        LD    HL,(ADDR)        ;OBTAIN BASE ADDRESS INDIRECTLY
        LD    B,A              ;SAVE DATA
        LD    A,(INDEX)        ;OBTAIN INDEX
        LD    E,A              ;EXTEND INDEX TO 16 BITS
        LD    D,0
        ADD   HL,DE            ;CALCULATE INDEXED ADDRESS
        LD    (HL),B           ;STORE DATA
```

By changing the contents of memory locations ADDR and ADDR+1, we can make this routine operate on many different arrays.

· Transfer control (jump) to the address obtained by indexing from the base address in memory locations ADDR and ADDR+1. The index is a constant OFFSET.

```
        LD    xy,(ADDR)         ;OBTAIN BASE ADDRESS INDIRECTLY
        LD    L,(xy+OFFSET)     ;OBTAIN LSB OF DESTINATION
        LD    H,(xy+OFFSET+1);OBTAIN MSB OF DESTINATION
        JP    (HL)              ;JUMP TO DESTINATION
```

This procedure is useful when a data structure contains the starting address of a routine at a fixed offset. The routine could, for example, be a driver for an I/O control block, an error routine for a mathematical function, or a control equation for a process loop.

· Transfer control (jump) to the address obtained by indexing from the base address in memory locations ADDR and ADDR+1. The index is in the accumulator.

```
        LD    B,A               ;TRIPLE INDEX FOR 3-BYTE ENTRIES
        ADD   A,A
        ADD   A,B
        LD    E,A               ;EXTEND INDEX TO 16 BITS
        LD    D,0
        LD    HL,(ADDR)         ;OBTAIN BASE ADDRESS INDIRECTLY
        ADD   HL,DE             ;CALCULATE INDEXED ADDRESS
        JP    (HL)              ;AND TRANSFER CONTROL TO IT
```

The table contains 3-byte JP instructions; a typical example is

```
BASE:   JP    ROUT0             ;JUMP TO ROUTINE 0
        JP    ROUT1             ;JUMP TO ROUTINE 1
        JP    ROUT2             ;JUMP TO ROUTINE 2
        ...
```

The address BASE must be placed in memory locations ADDR and ADDR+1.

REFERENCES

1. Fisher, W.P., "Microprocessor Assembly Language Draft Standard," *IEEE Computer,* December 1979, pp. 96-109. (See also Distler, R.J. and M.A. Shaver, "Trial Implementation Reveals Errors in IEEE Standard," *IEEE Computer,* July 1982, pp. 76-77.)

2. Osborne, A., *An Introduction to Microcomputers. Volume 1: Basic Concepts.* 2nd ed. Berkeley, Calif.: Osborne/McGraw-Hill, 1980.

3. Leventhal, L.A., *8080A/8085 Assembly Language Programming.* Berkeley, Calif.: Osborne/McGraw-Hill, 1978.

4. Fischer, op. cit.

Chapter 3 Common Programming Errors

This chapter describes common errors in Z80 assembly language programs. The final section describes common errors in input/output drivers and interrupt service routines. Our aims here are the following:

- To warn programmers of potential trouble spots and sources of confusion.
- To describe likely causes of programming errors.
- To emphasize the techniques and warnings presented in Chapters 1 and 2.
- To inform maintenance programmers of likely places to look for errors and misinterpretations.
- To provide the beginner with a starting point in the difficult process of locating and correcting errors.

Of course, no list of errors can be complete. Only the most common errors are emphasized—not the infrequent or subtle errors that frustrate even the experienced programmer. However, most errors are remarkably obvious once uncovered, and this discussion should help in debugging most programs.

CATEGORIZATION OF PROGRAMMING ERRORS

Common Z80 programming errors can be divided into the following categories:

- Reversing the order of operands or parts of operands. Typical errors include reversing source and destination in load instructions, inverting the format in which 16-bit quantities are stored, and inverting the direction of subtractions or comparisons.

· Using the flags improperly. Typical errors include using the wrong flag (such as Sign instead of Carry), branching after instructions that do not affect a particular flag, inverting branch conditions (particularly those involving the Zero flag), branching incorrectly in equality cases, and changing a flag accidentally before branching.

· Confusing registers and register pairs. A typical error is operating on a register instead of on a register pair.

· Confusing addresses and data. The most common error is omitting the parentheses around an address and hence accidentally using immediate addressing instead of direct addressing. Another error is confusing registers or register pairs with the memory locations addressed via register pairs.

· Using the wrong formats. Typical errors include using BCD (decimal) instead of binary, or vice versa, and using binary or hexadecimal instead of ASCII.

· Handling arrays incorrectly. The usual problem is going outside the array's boundaries.

· Ignoring implicit effects. Typical errors include using the accumulator, a register pair, the stack pointer, flags, or memory locations without considering the effects of intervening instructions. Most errors arise from instructions that have unexpected, implicit, or indirect effects.

· Failing to provide proper initial conditions for routines or for the microcomputer as a whole. Most routines require the initialization of counters, indirect addresses, base addresses, registers, flags, and temporary storage locations. The microcomputer as a whole requires the initialization of the interrupt system and all global RAM addresses. (Note particularly indirect addresses and counters.)

· Organizing the program incorrectly. Typical errors include skipping or repeating initialization routines, failing to update counters or address registers, and forgetting to save intermediate or final results.

A common source of errors that is beyond the scope of this discussion is conflict between user programs and systems programs. A simple example of conflict is for a user program to save data in memory locations that a systems program also uses. The user program's data thus changes mysteriously whenever the systems program is executed.

More complex sources of conflict include the interrupt system, input/output ports, the stack, and the flags. After all, systems programs must employ the same resources as user programs. Systems programs generally attempt to save and restore the user's environment, but they often have subtle or unexpected effects. Making an operating system transparent to the user is a problem comparable to devising a set of regulations, laws, or tax codes that have no loopholes or side effects.

REVERSING THE ORDER OF OPERANDS

The following instructions and conventions are the most common sources of errors:

· The LD D,S instruction moves the contents of S to D. Reversing the source and the destination in LD instructions is probably the single most common error in Z80 assembly language programs. The best way to avoid this problem is to use the operator notation described by Duncan.[1]

· 16-bit addresses and data items are assumed to be stored with their less significant bytes first (that is, at the lower address). This convention becomes particularly confusing in instructions that load or store register pairs or use the stack.

· The CP instruction subtracts its operand from the accumulator, not the other way around. Thus, CP sets the flags as if the processor had calculated (A) — OPER, where OPER is the operand specified in the instruction.

Examples

1. LD A,B

This instruction loads the accumulator from register B. Since it does not change B, the instruction acts like "copy B into A."

2. LD (HL),A

This instruction stores the accumulator at the memory address in register pair HL. Since it does not change the accumulator, the instruction acts like "copy A into memory addressed by HL."

3. LD (2040H),A

The address 2040_{16} occupies the two bytes of program memory immediately following the operation code; 40_{16} comes first and 20_{16} last. This order is particularly important to remember when entering or changing an address at the object code level during debugging.

4. PUSH HL

This instruction stores register pair HL in memory at the addresses immediately below the initial contents of the stack pointer (that is, at addresses S-1 and S-2 if S is the initial contents of the stack pointer). Register H is stored at address S-1 and L at S-2 in the usual upside-down format.

5. LD HL,(2050H)

This instruction loads register L from memory address 2050_{16} and H from 2051_{16}.

6. LD (3600H),HL

This instruction stores register L at memory address 3600_{16} and H at address 3601_{16}.

7. CP B

This instruction sets the flags as if register B had been subtracted from the accumulator.

8. CP 25H

This instruction sets the flags as if the number 25_{16} had been subtracted from the accumulator.

USING THE FLAGS INCORRECTLY

Z80 instructions have widely varying effects on the flags. There are few general rules, and even instructions with similar meanings may work differently. Cases that require special caution are

· Data transfer instructions such as LD and EX (except EX AF, AF') do not affect any flags. You may need an otherwise superfluous arithmetic or logical instruction (such as AND A, DEC, INC, or OR A) to set the flags.

· The Carry flag acts as a borrow after CP, SBC, or SUB instructions; that is, the Carry is set if the 8-bit unsigned subtraction requires a borrow. If, however, you implement subtraction by adding the two's or ten's complement of the subtrahend, the Carry is an inverted borrow; that is, the Carry is cleared if the 8-bit unsigned subtraction requires a borrow and set if it does not.

· After a comparison (CP), the Zero flag indicates whether the operands are equal; it is set if they are equal and cleared if they are not. There is an obvious source of confusion here—JZ means "jump if the result is 0," that is, "jump if the Zero flag is 1." JNZ, of course, has the opposite meaning.

· When comparing unsigned numbers, the Carry flag indicates which number is larger. CP sets Carry if the accumulator is less than the other operand and clears it if the accumulator is greater than or equal to the other operand. Note that the Carry is cleared if the operands are equal. If this division of cases ("greater than or equal" and "less than") is not what you want (that is, you want the division to be "greater than" and "less than or equal"), you can reverse the subtraction, subtract 1 from the accumulator, or add 1 to the other operand.

· In comparing signed numbers, the Sign flag indicates which operand is larger unless two's complement overflow occurs (see Chapter 1). CP sets the Sign flag if the accumulator is less than the other operand and clears it if the accumulator is greater than or equal to the other operand. Note that comparing equal operands clears the Sign flag. As with the unsigned numbers, you can handle the equality case in the opposite way by adjusting either operand or by reversing the subtraction. If overflow occurs (signified by the setting of the Parity/Overflow flag), the sense of the Sign flag is inverted.

· All logical instructions except CPL clear the Carry flag. AND A or OR A is, in fact, a quick, simple way to clear Carry without affecting any registers. CPL affects no flags at all (XOR 0FFH is an equivalent instruction that affects the flags).

· The common way to execute code only if a condition is truc is to branch around it if the condition is false. For example, to increment register B if Carry is 1, use the sequence

```
                JR    NC,NEXT
                INC   B
NEXT:           NOP
```

The branch occurs if Carry is 0.

· Many 16-bit arithmetic instructions have little effect on the flags. INC and DEC do not affect any flags when applied to register pairs or index registers; ADD HL and ADD xy affect only the Carry flag. The limited effects on the flags show that these instructions are intended for address arithmetic, not for the processing of 16-bit data. Note, however, that ADC HL and SBC HL affect all the flags and can be used for ordinary processing of 16-bit data.

· INC and DEC do not affect the Carry flag. This allows them to be used for counting in loops that perform multiple byte arithmetic. (The Carry is needed to transfer carries or borrows between bytes.) The 8-bit versions of INC and DEC do, however, affect the Zero and Sign flags, and you can use those effects to determine whether a carry or borrow occurred.

· The special instructions RLCA, RLA, RRCA, and RRA affect only the Carry flag.

· Special-purpose arithmetic and logical instructions such as ADD A,A (logical left shift accumulator), ADC A,A (rotate left accumulator), SUB A (clear accumulator), and AND A or OR A (test accumulator) affect all the flags.

· PUSH and POP instructions do not affect the flags, except for POP AF which changes all the flags. Remember, AF consists of the accumulator (MSB) and the flags (LSB).

Examples

1. The sequence

```
LD    A,(2040H)
JR    Z,DONE
```

has unpredictable results, since LD does not affect the flags. To produce a jump if memory location 2040_{16} contains 0, use

```
LD    A,(2040H)
AND   A                  ;TEST ACCUMULATOR
JR    Z,DONE
```

OR A may be used instead of AND A.

2. The sequence

```
LD    A,E
JP    P,DEST
```

has unpredictable results, since LD does not affect the flags. Either of the following sequences forces a jump if register E is positive:

```
LD    A,E
AND   A
JP    P,DEST
```

or

```
SUB   A
OR    E
JP    P,DEST
```

3. The instruction CP 25H sets the Carry flag as follows:

- Carry = 1 if the contents of A are between 00 and 24_{16}.
- Carry = 0 if the contents of A are between 25_{16} and FF_{16}.

The Carry flag is set if A contains an unsigned number less than the other operand and is cleared if A contains an unsigned number greater than or equal to the other operand.

If you want to set Carry if the accumulator contains 25_{16}, use CP 26H instead of CP 25H. That is, we have

```
CP    25H
JR    C,LESS             ;BRANCH IF (A) LESS THAN 25
```

or

```
CP    26H
JR    C,LESSEQ           ;BRANCH IF (A) 25 OR LESS
```

4. The sequence

```
RLA
JP    P,DONE
```

has unpredictable results, since RLA does not affect the Sign flag. The correct sequence (producing a circular shift that affects the flags) is

```
ADC   A,A                ;SHIFT CIRCULAR, SETTING FLAGS
JP    P,DONE
```

Of course, you can also use the somewhat slower

```
        RLA
        RLA
        JR    C,DONE
```

This approach allows a relative branch.

5. The sequence

```
        INC   B
        JR    C,OVRFLW
```

has unpredictable results, since INC does not affect the Carry flag. The correct sequence is

```
        INC   B
        JR    Z,OVRFLW
```

since INC does affect the Zero flag when it is applied to an 8-bit operand.

6. The sequence

```
        DEC   B
        JR    C,OVRFLW
```

has unpredictable results, since DEC does not affect the Carry flag. If B cannot contain a number larger than 80_{16} (unsigned), you can use

```
        DEC   B
        JP    M,OVRFLW
```

since DEC does affect the Sign flag (when applied to an 8-bit operand). Note, however, that you will get an erroneous branch if B initially contains 81_{16}.

A longer but more general sequence is

```
        INC   B                 ;TEST REGISTER B
        DEC   B
        JR    Z,OVRFLW          ;BRANCH IF B CONTAINS ZERO
        DEC   B
```

Note that register B will contain 0 (not FF_{16}) if the program branches to address OVRFLW.

7. The sequence

```
        DEC   BC
        JR    NZ,LOOP
```

has unpredictable results, since DEC does not affect any flags when it is applied to a 16-bit operand. The correct sequence for decrementing and testing a 16-bit counter in register pair BC is

```
        DEC   BC
        LD    A,C               ;CHECK IF BC HAS ANY 1 BITS
        OR    B
        JR    NZ,LOOP           ;BC CANNOT BE ZERO IF ANY BITS ARE 1
```

This sequence affects the accumulator and all the flags, including Carry (which OR clears).

8. AND A or OR A clears Carry without affecting any registers. To clear Carry without affecting the other flags, use the sequence

```
        SCF                     ;FIRST SET THE CARRY FLAG
        CCF                     ;THEN CLEAR IT BY COMPLEMENTING
```

9. SUB A or XOR A clears the accumulator, the Carry flag, and the Sign flag (and sets the Zero flag). To clear the accumulator without affecting the flags, use LD A,0.

10. The sequence

```
        ADD   HL,DE
        JR    Z,BNDRY
```

has unpredictable results, since ADD HL does not affect the Zero flag. To force a branch if the sum is 0, you must test HL explicitly as follows:

```
        ADD   HL,DE
        LD    A,H               ;TEST H AND L FOR ZERO
        OR    L
        JR    Z,BNDRY
```

An alternative is

```
        AND   A                 ;CLEAR CARRY
        ADC   HL,DE
        JR    Z,BNDRY
```

Unlike ADD HL, ADC HL affects the Zero flag.

CONFUSING REGISTERS AND REGISTER PAIRS

The rules to remember are

- ADC, ADD, DEC, INC, LD, and SBC can be applied to either 8-bit operands or 16-bit register pairs. ADD, DEC, INC, and LD can also be applied to index registers.
- AND, OR, SUB, and XOR can only be applied to 8-bit operands.
- EX, POP, and PUSH can only be applied to register pairs or index registers.
- (rp) refers to the byte of memory located at the address in the register pair. It does not refer to either half of the register pair itself.

One common error is that of referring to H or L instead of (HL). The use of register pairs to hold addresses means that certain transfers are uncommon. For example, LD

L,(HL) would load register L from the address in HL; HL would then contain one byte of an address (in H) and one byte of data (in L). While this is legal, it is seldom useful.

Examples

1. LD A,H

This instruction moves register H to the accumulator. It does not change register H or any memory location.

2. LD A,(BC)

This instruction loads the accumulator from the memory address in register pair BC. It does not affect either register B or register C.

3. LD H,0

This instruction places 0 in register H. It does not affect memory.

4. LD (HL),A

This instruction stores the accumulator in the memory location addressed by register pair HL. It does not affect either H or L. A sequence that loads HL with an address indirectly is

```
LD    E,(HL)          ;GET LSB OF INDIRECT ADDRESS
INC   HL
LD    D,(HL)          ;GET MSB OF INDIRECT ADDRESS
EX    DE,HL           ;PUT INDIRECT ADDRESS IN HL
```

We may limit ourselves to a single temporary register (the accumulator) by loading the more significant byte directly into H as follows:

```
LD    A,(HL)          ;GET LSB OF INDIRECT ADDRESS
INC   HL
LD    H,(HL)          ;GET MSB OF INDIRECT ADDRESS
LD    L,A             ;MOVE LSB OF ADDRESS TO L
```

This takes the same number of clock cycles as the previous sequence, but uses A instead of DE for temporary storage.

5. LD HL,2050H

This instruction loads 2050_{16} into register pair HL (20_{16} into H and 50_{16} into L).

6. ADD A,(HL)

This instruction adds the memory byte addressed via register pair HL to the accumulator. It does not affect either H or L.

7. ADD HL,HL

This instruction adds register pair HL to itself, thus shifting HL left 1 bit logically. This instruction does not affect the accumulator or access data from memory.

CONFUSING ADDRESSES AND DATA

The rules to remember are

- LD requires an address when you want to move data to or from memory. That address must be placed in parentheses.
- The standard assembler treats all operands as data unless they are enclosed in parentheses. Thus, if you omit the parentheses around an address, the assembler will treat it as a data item.
- DJNZ, JP, JR, and CALL always require addresses.

There is some confusion with addressing terminology in jump instructions. These instructions essentially treat their operands as if one level of indirection had been removed. For example, we say that JP 2040H uses direct addressing, yet we do not place the address in parentheses. Furthermore, JP 2040H loads 2040_{16} into the program counter, much as LD HL,2040H loads 2040_{16} into register pair HL. LD HL,(2040H) loads the contents of memory locations 2040_{16} and 2041_{16} into register pair HL. Note also that JP (HL) loads HL into the program counter; it does not use HL indirectly or access the memory at all.

Examples

1. LD A,40H loads the number 40_{16} into the accumulator. LD A,(40H) loads the contents of memory location 0040_{16} into the accumulator.

2. LD HL,0C00H loads $0C00_{16}$ into register pair HL ($0C_{16}$ into H and 00_{16} into L). LD HL,(0C00H) loads the contents of memory locations $0C00_{16}$ and $0C01_{16}$ into register pair HL (the contents of $0C00_{16}$ into L and the contents of $0C01_{16}$ into H).

3. JP (xy) transfers control to the address in an index register. No indexing is performed, nor is the address used to access memory.

Confusing addresses and their contents is a common error in handling data structures. For example, the queue of tasks to be executed by a piece of test equipment might consist of a block of information for each task. That block might contain

- Starting address of the test routine
- Number of seconds for which the test is to run

· Address in which the result is to be saved

· Upper and lower thresholds against which the result is to be compared

· Base address of the next block in the queue.

Thus, the block contains data, direct addresses, and indirect addresses. Typical errors that a programmer could make are

· Transferring control to the memory locations containing the starting address of the test routine, rather than to the actual starting address.

· Storing the result in the block rather than in the address specified in the block.

· Using a threshold as an address rather than as data.

· Assuming that the next block starts in the current block, rather than at the base address given in the current block.

Jump tables are another common source of errors. The following are alternative implementations:

· Form a table of jump instructions and transfer control to the correct element (for example, to the third jump instruction).

· Form a table of destination addresses and transfer control to the contents of the correct element (for example, to the address in the third element).

You will surely have problems if the processor uses jump instructions as addresses or vice versa.

FORMAT ERRORS

The rules you should remember for the standard Z80 assembler are

· An H at the end of a number indicates hexadecimal and a B indicates binary.

· The default mode for numbers is decimal; that is, the assembler assumes all numbers to be decimal unless they are specifically marked otherwise.

· All operands are treated as data unless they are enclosed in parentheses. Operands enclosed in parentheses are assumed to be memory addresses.

· A hexadecimal number that starts with a letter digit (A, B, C, D, E, or F) must be preceded by 0 (for example, 0CFH instead of CFH) for the assembler to interpret it correctly. Of course, the leading 0 does not affect the value of the number.

· All arithmetic and logical operations are binary, except DAA, which corrects the result of an 8-bit binary addition or subtraction to the proper BCD value.

You should beware of the following common errors:

- Omitting the H from a hexadecimal operand. The assembler will assume it to be decimal if it contains no letter digits and to be a name if it starts with a letter. The assembler will indicate an error only if it cannot interpret the operand as either a decimal number or a name.
- Omitting the B from a binary operand. The assembler will assume it to be decimal.
- Confusing decimal (BCD) representations with binary representations. Remember, ten is not an integral power of two, so the binary and BCD representations are not the same beyond nine. BCD constants must be designated as hexadecimal numbers, not as decimal numbers.
- Confusing binary or decimal representations with ASCII representations. An ASCII input device produces ASCII characters and an ASCII output device responds to ASCII characters.

Examples

1. LD A,(2000)

This instruction loads the accumulator from memory address 2000_{10} ($07D0_{16}$), not address 2000_{16}. The assembler will not produce an error message, since 2000 is a valid decimal number.

2. AND 00000011

This instruction logically ANDs the accumulator with the decimal number 11 (1011_2), not with the binary number 11 (3_{10}). The assembler will not produce an error message, since 00000011 is a valid decimal number despite its unusual form.

3. ADD A,40

This instruction adds the number 40_{10} to the accumulator. Note that 40_{10} is not the same as BCD 40, which is 40_{16}; $40_{10} = 28_{16}$. The assembler will not produce an error message, since 40 is a valid decimal number.

4. LD A,3

This instruction loads the accumulator with the number 3. If this value is now sent to an ASCII output device, the device will respond as if it had received the character ETX (03_{16}), not the character 3 (33_{16}). The correct version is

```
        LD    A,'3'              ;GET AN ASCII 3
```

If memory location 2040_{16} contains a single digit, the sequence

```
        LD    A,(2040H)
        OUT   (DEVCE),A
```

will not print that digit on an ASCII output device. The correct sequence is

```
        LD    A,(2040H)         ;GET DECIMAL DIGIT
        ADD   A,'0'             ;ADJUST TO ASCII
        OUT   (DEVCE),A
```

If input port INDEV contains a single ASCII decimal digit, the sequence

```
        IN    A,(INDEV)
        LD    (2040H),A
```

will not store the actual digit in memory location 2040_{16}. Instead, it will store the ASCII version, which is the actual digit plus 30_{16}. The correct sequence is

```
        IN    A,(INDEV)         ;GET ASCII DIGIT
        SUB   '0'               ;ADJUST TO DECIMAL
        LD    (2040H),A
```

Performing decimal arithmetic on the Z80 is awkward, since a DAA instruction is required after each 8-bit addition or subtraction. Chapter 6 contains programs for decimal arithmetic operations. Since DAA does not work properly after DEC or INC, the following sequences are necessary to perform decimal increment and decrement by 1:

- Add 1 to the accumulator in decimal.

```
        ADD   A,1
        DAA
```

- Subtract 1 from the accumulator in decimal.

```
        SUB   1
        DAA
```

or

```
        ADD   A,99H
        DAA
```

In the second alternative, Carry is an inverted borrow.

HANDLING ARRAYS INCORRECTLY

The most common problems here are executing an extra iteration or stopping one short. Remember, memory locations BASE through BASE+N contain N+1 bytes, not N bytes. It is easy to forget the last entry or drop the first one. On the other hand, if you have N entries, they will occupy memory locations BASE through BASE+N−1; now it is easy to find yourself working beyond the end of the array.

IMPLICIT EFFECTS

Some implicit effects you should remember are

- The clearing of Carry by all logical operations except CPL.
- The moving of the interrupt flip-flop IFF2 to the Parity/Overflow flag by LD A,I and LD A,R.
- The use of the data at the address in HL by the digit rotations RRD and RLD.
- The use of the memory address one larger than the specified one by LD rp,(ADDR), LD (ADDR),rp, LD xy,(ADDR), and LD (ADDR),xy.
- The changing of the stack pointer by POP, PUSH, CALL, RET, RETI, RETN, and RST.
- The saving of the return address in the stack by CALL and RST.
- The decrementing of register B by DJNZ.
- The implicit effects on BC, DE, and HL of the block compare, input, move, and output instructions.
- The use of the Parity/Overflow flag by LDD, LDI, CPD, CPDR, CPI, and CPIR to indicate whether the counter in BC has been decremented to 0.

Examples

1. AND 00001111B

This instruction clears the Carry, as well as performing a logical operation.

2. LD A,I

This instruction not only loads the accumulator, but also moves the interrupt flip-flop IFF2 to the Parity/Overflow flag. The interrupt status can then be saved before the computer executes a routine that must run with interrupts disabled.

3. RRD

This instruction performs a 4-bit (digit) circular shift right involving the accumulator and the memory location addressed by HL. The results are

- The 4 least significant bits of A go into the 4 most significant bits of the memory location.
- The 4 most significant bits of the memory location go into its 4 least significant bits.

· The 4 least significant bits of the memory location go into the 4 least significant bits of A.

The result is thus a 4-bit right rotation of the 12-bit number made up of the 4 LSBs of the accumulator and the memory byte.

4. LD HL,(16EFH)

This instruction loads register L from memory location $16EF_{16}$ and H from memory location $16F0_{16}$. Note the implicit use of address $16F0_{16}$.

5. POP HL

This instruction not only loads register pair HL from memory, but also increments the stack pointer by 2.

6. CALL SUBR

This instruction not only transfers control to address SUBR, but it also saves the address of the next sequential instruction in the stack. Furthermore, CALL decrements the stack pointer by 2.

7. DJNZ LOOP

This instruction decrements register B and branches to address LOOP if the result is not 0. Note that register B is implied as the counter.

8. LDD

This instruction moves data from the address in HL to the address in DE. It also decrements BC, DE, and HL by 1. The Parity/Overflow flag (*not* the Zero flag) is cleared (*not* set) if BC is decremented to 0; the Parity/Overflow flag is set otherwise.

9. CPIR

This instruction compares the accumulator with the memory byte at the address in HL. After the comparison, it increments HL by 1 and decrements BC by 1. It repeats these operations until it decrements BC to 0 (indicated by the Parity/Overflow flag being cleared) or until the comparison sets the Zero flag. Note that CPIR updates BC and HL before it tests for an exit condition.

10. OUTI

This instruction transfers data from the memory address in HL to the output port in C. It then decrements B (not BC) by 1 and increments HL by 1. OUTI sets the Zero flag to 1 if it decrements BC to 0; it clears the Zero flag otherwise.

INITIALIZATION ERRORS

Initialization routines must perform the following tasks, either for the microcomputer system as a whole or for particular routines:

· Load all RAM locations with initial values. This includes indirect addresses and other temporary storage. You cannot assume that a memory location contains 0 just because you have not used it.

· Load all registers and flags with initial values. Reset initializes the interrupt system by disabling regular interrupts and selecting Mode 0. The startup program for an interrupt-driven system must set the interrupt mode (if it is not 0), initialize the stack pointer, and load the interrupt vector register (in Mode 2).

· Initialize all counters and indirect addresses. Pay particular attention to register pairs that are used as address registers; you must initialize them before using instructions that refer to them indirectly.

ORGANIZING THE PROGRAM INCORRECTLY

The following problems are the most common:

· Accidentally reinitializing a register, register pair, flag, memory location, counter, or indirect address. Be sure that your branches do not result in the repetition of initialization instructions.

· Failing to update a counter, index register, address register, or indirect address. A problem here may be a path that branches around the updating instructions or changes values before executing those instructions.

· Forgetting to save results. It is remarkably easy to calculate a result and then load something else into the accumulator. Identifying this kind of error is frustrating and time-consuming, since all the instructions that calculate the result work properly and yet the result itself is being lost. For example, a branch may transfer control to an instruction that writes over the result.

· Forgetting to branch around instructions that should not be executed in a particular path. Remember, the computer will execute instructions consecutively unless told to do otherwise. Thus, the computer may fall through to a section of the program that you expect it to reach only via a branch. An unconditional jump instruction will force a branch around the section that should not be executed.

ERROR RECOGNITION BY ASSEMBLERS

Most assemblers will recognize some common errors immediately, such as

· Undefined operation code (usually a misspelling or the omission of a colon after a label).

· Undefined name (often a misspelling or an omitted definition).

· Illegal character (for example, a 2 in a binary number or a B in a decimal number).

· Illegal format (for example, an incorrect delimiter or the wrong operands).

· Illegal value (usually a number too large for 8 or 16 bits).

· Missing operand.

· Double definition (two different values assigned to one name).

· Illegal label (for example, a label attached to a pseudo-operation that does not allow a label).

· Missing label (for example, on an EQU pseudo-operation that requires one).

These errors are annoying but easy to correct. The only problem comes when an error (such as omitting the semicolon from a comment line) confuses the assembler completely and results in a series of meaningless error messages.

There are, however, many simple errors that assemblers will not recognize. The programmer should be aware that his or her program may contain such errors even if the assembler does not report them. Typical examples are

· Omitted lines. Obviously, the assembler cannot tell that you have omitted a line completely unless it contains a label or definition that is used elsewhere. The easiest lines to omit are ones that are repetitious or seem unnecessary. Typical repetitions are series of shifts, branches, increments, or decrements. Instructions that often appear unnecessary include AND A, DEC HL, INC HL, OR A, and SUB A.

· Omitted designations. The assembler cannot tell if you meant an operand to be hexadecimal or binary unless the omission results in an illegal character (such as C in a decimal number). Otherwise, the assembler will assume all numbers to be decimal. Problems occur with hexadecimal numbers that contain no letter digits (such as 44 or 2050) and with binary numbers (such as 00000110).

· Omitted parentheses. The assembler cannot tell if you meant to refer to a memory address unless omitting the parentheses results in an error. Many instructions, such as LD A,(40H), INC (HL), DEC (HL), and LD HL,(2050H), are also valid without parentheses.

· Misspellings that are still valid. Typical examples are typing AND or ADC instead of ADD, DI instead of EI, or D instead of E. Unless the misspelling is invalid, the assembler has no way of sensing an error. Valid misspellings are often a problem if you use names that look alike, such as XXX and XXXX, L121 and L112, or VAR1I and VARII.

· Designating instructions as comments. If you place a semicolon at the start of an instruction line, the assembler will treat the line as a comment. This can be a perplexing error, since the line appears in the listing but is not assembled into code.

Sometimes you can confuse an assembler by entering completely invalid instructions. An assembler may accept them simply because its developer never anticipated such mistakes. The results can be unpredictable, much like the result of accidentally entering your weight instead of your age or your telephone number instead of your credit card number on a form. Some cases in which a Z80 assembler can go wrong are

· If you specify a single register instead of a register pair. Some assemblers will accept instructions like LD A,(L), ADD HL,D, or LD E,2040H. They will produce meaningless object code without any indication of error.

· If you enter an invalid digit, such as X in a decimal or hexadecimal number or 7 in a binary number. Some assemblers will assign arbitrary values to such invalid digits.

· If you enter an invalid operand such as 40H in RST, AF in LD, or SP in PUSH or POP. Some assemblers will accept these and generate meaningless code.

The assembler will only recognize errors that its developer anticipated. Programmers are often able to make mistakes the developer never imagined, much as automobile drivers are often capable of getting into predicaments that no highway engineer or traffic policeman ever thought possible. Note that only a line-by-line hand checking of the program will find errors that the assembler does not recognize.

COMMON ERRORS IN I/O DRIVERS

Since most errors in I/O drivers involve both hardware and software, they are difficult to categorize. Some things you should watch for are

· Confusing input ports and output ports. Input port 20_{16} and output port 20_{16} are different in most systems. Even when the two ports are the same physically, it may still be impossible to read back output data unless the port is latched and buffered.

· Attempting to perform operations that are physically impossible. Reading data from an output device (such as a display) or sending data to an input device (such as a keyboard) makes no physical sense. However, accidentally using the wrong port number will cause no assembly errors; the port, after all, exists and the assembler has no way of knowing that certain operations cannot be performed on it. Similarly, a program may attempt to save data in an unassigned address or in a ROM.

· Forgetting implicit hardware effects. At times, transferring data to or from a port will change the status lines automatically (as in most PIO modes). Even reading or writing the port while debugging a program will change status lines. When using memory-mapped I/O, be particularly careful of instructions like comparisons and BIT that read a memory address even though they do not change any registers. Similarly, instructions like BIT, RES, SET, DEC, INC, and shifts can both read and write a memory address. Automatic port operations can save parts and simplify programs, but you must remember how they work and when they occur.

· Reading or writing without checking status. Many devices can only accept or provide data when a status line indicates they are ready. Transferring data to or from them at other times will have unpredictable results.

· Ignoring the differences between input and output. Remember that an input device normally starts out *not ready*—it has no data available although the computer is ready to accept data. On the other hand, an output device normally starts out *ready*—that is, it could accept data but the computer usually has none to send it. In many situations (particularly when using PIOs), you may have to send a null character (something that has no effect) to each output port just to change its state from *ready* to *not ready* initially.

· Failing to keep a copy of output data. Generally, you will not be able to read data back from an output port. You must save a copy in memory if it could be needed later to repeat a transmission, change some bits, or restore interrupt status (the data could, for example, be the current priority level).

· Reading data before it is stable or while it is changing. Be sure that you understand exactly when the data from the input device is guaranteed to be stable. In the case of switches that may bounce, you may want to sample them twice (more than a debouncing time apart) before taking any action. In the case of keys that may bounce, you may want to take action only when they are released rather than when they are pressed. Acting on release also forces the operator to release the key rather than holding it down. In the case of persistent data (such as in serial I/O), you should center the reception (that is, read the data near the centers of the pulses rather than at the edges where the values may be changing).

· Forgetting to reverse the polarity of data being transferred to or from devices that operate in negative logic. Many simple I/O devices, such as switches and displays, use negative logic; a logic 0 means that a switch is closed or a display is lit. Common ten-position switches or dials also often produce data in negative logic, as do many encoders. The solution is simple—complement the data using CPL after reading it or before sending it.

· Confusing actual I/O ports with registers that are inside I/O chips. Programmable I/O devices, such as the CTC, PIO, and SIO, typically have control or command registers that determine how the device operates and status registers that reflect the current state of the device or the transfer. These registers are inside the chips; they are not connected to peripherals. Transferring data to or from these registers is not the same as transferring data to or from actual I/O ports.

· Using bidirectional ports improperly. Many devices, such as the PIO, have bidirectional I/O ports that can be used either for input or output. Normally, resetting the computer makes these ports inputs in order to avoid initial transients, so the program must explicitly change them to outputs if necessary. Be particularly careful of instructions that read bits or ports that are designated as outputs or that write into bits or ports designated as inputs. The only way to determine what will happen is to read the documentation for the specific device.

· Forgetting to clear status after performing an I/O operation. Once the processor has read data from a port or written data into a port, that port should revert to the *not ready* state. Some I/O devices change the status of their ports automatically after input or output operations, but others either do not or they change status automatically only after input. Leaving the status set can result in an endless loop or erratic operation.

COMMON ERRORS IN INTERRUPT SERVICE ROUTINES

Many errors that are related to interrupts involve both hardware and software. The following are some of the more common mistakes:

· Failing to reenable interrupts. The Z80 disables interrupts automatically after accepting one, but does not reenable interrupts unless it executes EI.

· Failing to save registers. The Z80 does not automatically save any registers except the program counter, so any registers that the service routine uses must be saved explicitly in the stack.

· Saving or restoring registers in the wrong order. Registers must be restored in the opposite order from that in which they were saved.

· Enabling interrupts before initializing modes, priorities, the interrupt vector register, or other parameters of the interrupt system.

· Forgetting that the response to an interrupt includes saving the program counter at the top of the stack. The return address will thus be on top of whatever else is in the stack.

· Not disabling the interrupt during multi-byte transfers or instruction sequences that cannot be interrupted. In particular, watch for possible partial updating of data (such as time) that a service routine may use.

· Failing to reenable interrupts after a sequence that must be run with interrupts disabled. One problem here is that interrupts should not be enabled afterward if they were not enabled originally. This requirement is difficult to meet on the Z80 since its interrupt enable is not directly readable. The only way to access the interrupt flip-flop is by executing LD A,I or LD A,R; either instruction moves the interrupt flip-flop to the Parity/Overflow flag.

· Failing to clear the signal that caused the interrupt. The service routine must clear the interrupt even if no I/O operations are necessary. For example, even when the processor has no data to send to an interrupting output device, it must nonetheless either clear or disable the interrupt. Otherwise, the processor will get caught in an endless loop. Similarly, a real-time clock will typically require no servicing other than an updating of time, but the service routine still must clear the clock interrupt. This clearing may involve reading a timer register.

· Failing to communicate with the main program. The main program will not know that the interrupt has been serviced unless it is informed explicitly. The usual way to inform the main program is to have the service routine change a flag. The main program can tell from the flag's value whether the service routine has been executed. This procedure works like a postal patron raising a flag to indicate that there is mail to be picked up. The letter carrier lowers the flag after picking up the mail. Note that this simple procedure means that the main program must examine the flag often enough to avoid missing changes in its value. Of course, the programmer can always provide a buffer that can hold many data items.

· Failing to save and restore priority. The priority of an interrupt is often held in a write-only register or in a memory location. That priority must be saved just like a CPU register and restored properly at the end of the service routine. If the priority register is write-only, a copy of its contents must be saved in memory.

REFERENCES

1. Duncan, F.G., "Level-Independent Notation for Microcomputer Programs," *IEEE Micro,* May 1981, pp. 47-52.

Introduction to the Program Section

The program section contains sets of assembly language subroutines for the Z80 microprocessor. Each subroutine is documented with an introductory section and comments and is followed by at least one example of its use. The introductory material contains the following information about the purpose of the routine: its procedure and the registers that are used; the execution time, program size, and data memory required for the routine; as well as special cases, entry conditions, and exit conditions.

We have made each routine as general as possible. This is particularly difficult for the input/output (I/O) and interrupt service routines described in Chapters 10 and 11, since these routines are always computer-dependent in practice. In such cases, we have limited the computer-dependence to generalized input and output handlers and interrupt managers. We have drawn specific examples from computers based on the CP/M operating system, but the general principles are applicable to other Z80-based computers as well.

In all routines, we have used the following parameter passing techniques:

1. A single 8-bit parameter is passed in the accumulator. A second 8-bit parameter is passed in register B, and a third in register C.

2. A single 16-bit parameter is passed in register pair HL with the more significant byte in H. A second 16-bit parameter is passed in register pair DE with the more significant byte in D.

3. Large numbers of parameters are passed in the stack, either directly or indirectly. We assume that subroutines are entered via a CALL instruction that places the return address at the top of the stack, and hence on top of the parameters.

Where there has been a choice between execution time and memory usage, we have generally chosen to minimize execution time. We have therefore avoided slowly executing instructions such as stack transfers and instructions that use the index registers, even when they would make programs shorter. However, we have used relative jumps whenever possible rather than the slightly faster but longer absolute jumps to make programs easier to relocate.

We have also chosen the approach that minimizes the number of repetitive calculations. For example, in the case of array indexing, the number of bytes between the starting addresses of elements differing only by one in a particular subscript (known as the *size* of that subscript) depends only on the number of bytes per element and the bounds of the array. Thus, the sizes of the various subscripts can be calculated as soon as the bounds of the array are known; the sizes are therefore used as parameters for the indexing routines, so that they need not be calculated each time a particular array is indexed.

As for execution time, we have specified it for most short routines. For longer routines we have given an approximate execution time. The execution time of programs involving many branches will obviously depend on which path the computer follows in a particular case. This is further complicated for the Z80 because conditional jump instructions themselves require different numbers of clock cycles depending on whether the branch is taken. Thus, a precise execution time is often impossible to define. The documentation always contains at least one typical example showing an approximate or maximum execution time.

Although we have drawn examples from CP/M-based systems, we have not made our routines compatible with the 8080 or 8085 processors. Readers who need routines that can run on any of these processors should refer to the 8080/8085 version of this book. We have considered the Z80 as an independent processor and have taken advantage of such features as block moves, block compares, loop control instructions, and relative jumps.

Our philosophy on error indicators and special cases has been the following:

1. Routines should provide an easily tested indicator (such as the Carry flag) of whether any errors or exceptions have occurred.

2. Trivial cases, such as no elements in an array or strings of zero length, should result in immediate exits with minimal effect on the underlying data.

3. Incorrectly specified data (such as a maximum string length of zero or an index beyond the end of an array) should result in immediate exits with minimal effect on the underlying data.

4. The documentation should include a summary of errors and exceptions (under the heading of "Special Cases").

5. Exceptions that may actually be convenient for the user (such as deleting more characters than could possibly be left in a string rather than counting the precise number) should be handled in a reasonable way, but should still be indicated as errors.

Obviously, no method of handling errors or exceptions can ever be completely consistent or well-suited to all applications. And rather than assume that the user will always provide data in the proper form, we believe a reasonable set of subroutines must deal with this issue.

The subroutines are listed as follows:

Binary to BCD Conversion (BN2BCD) 4A

Converts one byte of binary data to two bytes of BCD data.

Procedure: The program subtracts 100 repeatedly from the original data to determine the hundreds digit, then subtracts 10 repeatedly from the remainder to determine the tens digit, and finally shifts the tens digit left four positions and combines it with the ones digit.

Registers Used: AF, C, HL
Execution Time: 497 cycles maximum; depends on the number of subtractions required to determine the tens and hundreds digits
Program Size: 27 bytes
Data Memory Required: None

Entry Conditions

Binary data in A

Exit Conditions

Hundreds digit in H
Tens and ones digits in L

Examples

1. Data:	(A) = $6E_{16}$ (110 decimal)	
Result:	(H) = 01_{16} (hundreds digit)	
	(L) = 10_{16} (tens and ones digits)	
2. Data:	(A) = $B7_{16}$ (183 decimal)	
Result:	(H) = 01_{16} (hundreds digit)	
	(L) = 83_{16} (tens and ones digits)	

```
;                                                                    ;
;                                                                    ;
;                                                                    ;
;                                                                    ;
;       Title           Binary to BCD conversion                     ;
;       Name:           BN2BCD                                       ;
;                                                                    ;
;                                                                    ;

;                                                                    ;
;       Purpose:        Convert one byte of binary data to two       ;
;                       bytes of BCD data                            ;
;                                                                    ;
;       Entry:          Register A = binary data                     ;
;                                                                    ;
;       Exit:           Register H = High byte of BCD data           ;
;                       Register L = Low  byte of BCD data           ;
;                                                                    ;
;       Registers used: AF,C,HL                                      ;
;                                                                    ;
;       Time:               497 cycles maximum                       ;
;                                                                    ;
;       Size:               Program 27 bytes                         ;
;                                                                    ;
;                                                                    ;

BN2BCD:
        ;CALCULATE 100'S DIGIT - DIVIDE BY 100
        ; H = QUOTIENT
        ; A = REMAINDER
        LD      H,0FFH          ;START QUOTIENT AT -1
D100LP:
        INC     H               ;ADD 1 TO QUOTIENT
        SUB     100             ;SUBTRACT 100
        JR      NC,D100LP       ;JUMP IF DIFFERENCE STILL POSITIVE
        ADD     A,100           ;ADD THE LAST 100 BACK

        ;CALCULATE 10'S AND 1'S DIGITS
        ;  DIVIDE REMAINDER OF THE 100'S DIGIT BY 10
        ; L = 10'S DIGIT
        ; A = 1'S DIGIT
        LD      L,0FFH          ;START QUOTIENT AT -1
D10LP:
        INC     L               ;ADD 1 TO QUOTIENT
        SUB     10              ;SUBTRACT 10
        JR      NC,D10LP        ;JUMP IF DIFFERENCE STILL POSITIVE
        ADD     A,10            ;ADD THE LAST 10 BACK

        ;COMBINE 1'S AND 10'S DIGITS
        LD      C,A             ;SAVE 1'S DIGIT IN C
        LD      A,L
        RLCA                    ;MOVE 10'S TO HIGH NIBBLE OF A
        RLCA
        RLCA
        RLCA
        OR      C               ;OR IN THE 1'S DIGIT

        ;RETURN WITH L = LOW BYTE, H = HIGH BYTE
        LD      L,A
        RET


;                                                                    ;
;                                                                    ;
;       SAMPLE EXECUTION:                                            ;
;                                                                    ;
;                                                                    ;

SC4A:
        ;CONVERT 0A HEXADECIMAL TO 10 BCD
        LD      A,0AH
        CALL    BN2BCD          ;H = 0, L = 10H

        ;CONVERT FF HEXADECIMAL TO 255 BCD
        LD      A,0FFH
        CALL    BN2BCD          ;H = 02H, L = 55H

        ;CONVERT 0 HEXADECIMAL TO 0 BCD
        LD      A,0
        CALL    BN2BCD          ;H = 0, L = 0

        JR      SC4A

        END
```

BCD to Binary Conversion (BCD2BN) 4B

Converts one byte of BCD data to one byte of binary data.

Procedure: The program masks off the more significant digit, multiplies it by 10 using shifts (10=8+2, and multiplying by 8 or by 2 is equivalent to three or one left shifts, respectively). Then the program adds the product to the less significant digit.

Registers Used: AF, BC
Execution Time: 60 cycles
Program Size: 14 bytes
Data Memory Required: None

Entry Conditions

BCD data in A

Exit Conditions

Binary data in A

Examples

1. Data: (A) = 99_{16}
   Result: (A) = $63_{16} = 99_{10}$

2. Data: (A) = 23_{16}
   Result: (A) = $17_{16} = 23_{10}$

```
;                                                                   ;
;                                                                   ;
;                                                                   ;
;                                                                   ;
;       Title           BCD to binary conversion                    ;
;       Name:           BCD2BN                                      ;
;                                                                   ;
;                                                                   ;

;                                                                   ;
;       Purpose:        Convert one byte of BCD data to one         ;
;                       byte of binary data                         ;
;                                                                   ;
;       Entry:          Register A = BCD data                       ;
;                                                                   ;
;       Exit:           Register A = Binary data                    ;
;                                                                   ;
;       Registers used: A,B,C,F                                     ;
;                                                                   ;
;       Time:           60 cycles                                   ;
;                                                                   ;
;       Size:           Program 14 bytes                                    ;
;                                                                           ;
;                                                                           ;
;                                                                           ;

BCD2BN:
        ;MULTIPLY UPPER NIBBLE BY 10 AND SAVE IT
        ; UPPER NIBBLE * 10 = UPPER NIBBLE * (8 + 2)
        LD      B,A             ;SAVE ORIGINAL BCD VALUE IN B
        AND     0F0H            ;MASK OFF UPPER NIBBLE
        RRCA                    ;SHIFT RIGHT 1 BIT
        LD      C,A             ;C = UPPER NIBBLE * 8
        RRCA                    ;SHIFT RIGHT 2 MORE TIMES
        RRCA                    ;A = UPPER NIBBLE * 2
        ADD     A,C
        LD      C,A             ;C = UPPER NIBBLE * (8+2)

        ;GET LOWER NIBBLE AND ADD IT TO THE
        ; BINARY EQUIVALENT OF THE UPPER NIBBLE
        LD      A,B             ;GET ORIGINAL VALUE BACK
        AND     0FH             ;MASK OFF UPPER NIBBLE
        ADD     A,C             ;ADD TO BINARY UPPER NIBBLE
        RET


;                                                                           ;
;                                                                           ;
;       SAMPLE EXECUTION:                                                   ;
;                                                                           ;
;                                                                           ;

SC4B:
        ;CONVERT 0 BCD TO 0 HEXADECIMAL
        LD      A,0
        CALL    BCD2BN          ;A = 0H

        ;CONVERT 99 BCD TO 63 HEXADECIMAL
        LD      A,099H
        CALL    BCD2BN          ;A=63H

        ;CONVERT 23 BCD TO 17 HEXADECIMAL
        LD      A,23H
        CALL    BCD2BN          ;A=17H

        JR      SC4B

        END
```

Binary to Hexadecimal ASCII Conversion (BN2HEX) 4C

Converts one byte of binary data to two ASCII characters corresponding to the two hexadecimal digits.

Procedure: The program masks off each hexadecimal digit separately and converts it to its ASCII equivalent. This involves a simple addition of 30_{16} if the digit is decimal. If the digit is non-decimal, an additional 7 must be added to account for the break between ASCII 9 (39_{16}) and ASCII A (41_{16}).

Registers Used: AF, B, HL
Execution Time: 162 cycles plus two extra cycles for each non-decimal digit
Program Size: 28 bytes
Data Memory Required: None

Entry Conditions

Binary data in A

Exit Conditions

ASCII version of more significant hexadecimal digit in H
ASCII version of less significant hexadecimal digit in L

Examples

1. Data: (A) = FB_{16}
   Result: (H) = 46_{16} (ASCII F)
   (L) = 42_{16} (ASCII B)

2. Data: (A) = 59_{16}
   Result: (H) = 35_{16} (ASCII 5)
   (L) = 39_{16} (ASCII 9)

```
;                                                                    ;
;                                                                    ;
;                                                                    ;
;                                                                    ;
;       Title           Binary to hex ASCII                          ;
;       Name:           BN2HEX                                       ;
;                                                                    ;
;                                                                    ;

;                                                                    ;
;       Purpose:        Convert one byte of binary data to           ;
;                       two ASCII characters                         ;
;                                                                    ;
;       Entry:          Register A = Binary data                     ;
;                                                                    ;
;       Exit:           Register H = ASCII more significant digit      ;
;                       Register L = ASCII less significant digit      ;
;                                                                      ;
;       Registers used: AF,B,HL                                        ;
;                                                                      ;
;       Time:           Approximately 162 cycles                       ;
;                                                                      ;
;       Size:           Program 28 bytes                               ;
;                                                                      ;
;                                                                      ;

BN2HEX:
        ;CONVERT HIGH NIBBLE
        LD      B,A             ;SAVE ORIGINAL BINARY VALUE
        AND     0F0H            ;GET HIGH NIBBLE
        RRCA                    ;MOVE HIGH NIBBLE TO LOW NIBBLE
        RRCA
        RRCA
        RRCA
        CALL    NASCII          ;CONVERT HIGH NIBBLE TO ASCII
        LD      H,A             ;RETURN HIGH NIBBLE IN H

        ;CONVERT LOW NIBBLE
        LD      A,B
        AND     0FH             ;GET LOW NIBBLE
        CALL    NASCII          ;CONVERT LOW NIBBLE TO ASCII
        LD      L,A             ;RETURN LOW NIBBLE IN L
        RET

        ;-----------------------------------
        ;SUBROUTINE ASCII
        ;PURPOSE:   CONVERT A HEXADECIMAL DIGIT TO ASCII
        ;ENTRY: A = BINARY DATA IN LOWER NIBBLE
        ;EXIT:  A = ASCII CHARACTER
        ;REGISTERS USED: A,F
        ;-----------------------------------
NASCII:
        CP      10
        JR      C,NAS1          ;JUMP IF HIGH NIBBLE < 10
        ADD     A,7             ;ELSE ADD 7 SO AFTER ADDING '0' THE
                                ; CHARACTER WILL BE IN 'A'..'F'
NAS1:
        ADD     A,'0'           ;ADD ASCII 0 TO MAKE A CHARACTER
        RET


;                                                                      ;
;                                                                      ;
;       SAMPLE EXECUTION:                                              ;
;                                                                      ;
;                                                                      ;

SC4C:
        ;CONVERT 0 TO '00'
        LD      A,0
```

```
CALL     BN2HEX              ;H='0'=30H, L='0'=30H

;CONVERT FF HEX TO 'FF'
LD       A,0FFH
CALL     BN2HEX              ;H='F'=46H, L='F'=46H

;CONVERT 23 HEX TO '23'
LD       A,23H
CALL     BN2HEX              ;H='2'=32H, L='3'=33H

JR       SC4C

END
```

Hexadecimal ASCII to Binary Conversion (HEX2BN) 4D

Converts two ASCII characters (representing two hexadecimal digits) to one byte of binary data.

Procedure: The program converts each ASCII character separately to a hexadecimal digit. This involves a simple subtraction of 30_{16} (ASCII 0) if the digit is decimal. If the digit is non-decimal, another 7 must be subtracted to account for the break between ASCII 9 (39_{16}) and ASCII A (41_{16}). The program then shifts the more significant digit left four bits and combines it with the less significant digit. The program does not check the validity of the ASCII characters (that is, whether they are indeed the ASCII representations of hexadecimal digits).

Registers Used: AF, B
Execution Time: 148 cycles plus two extra cycles for each non-decimal digit
Program Size: 24 bytes
Data Memory Required: None

Entry Conditions

More significant ASCII digit in H, less significant ASCII digit in L

Exit Conditions

Binary data in A

Examples

1. Data: (H) = 44_{16} (ASCII D)
   (L) = 37_{16} (ASCII 7)
   Result: (A) = $D7_{16}$

2. Data: (H) = 31_{16} (ASCII 1)
   (L) = 42_{16} (ASCII B)
   Result: (A) = $1B_{16}$

```
;                                                                    ;
;                                                                    ;
;                                                                    ;
;                                                                    ;
;       Title           Hex ASCII to binary                          ;
;       Name:           HEX2BN                                       ;
;                                                                    ;
;                                                                    ;

;                                                                    ;
;       Purpose:        Convert two ASCII characters to one          ;
;                       byte of binary data                          ;
;                                                                    ;
;       Entry:          Register H = ASCII more significant digit    ;
;                    Register L = ASCII less significant digit    ;
;                                                                  ;
;       Exit:        Register A = Binary data                      ;
;                                                                  ;
;       Registers used: AF,B                                       ;
;                                                                  ;
;       Time:        Approximately 148 cycles                      ;
;                                                                  ;
;       Size:        Program 24 bytes                              ;
;                                                                  ;
;                                                                  ;

HEX2BN:
        LD      A,L             ;GET LOW CHARACTER
        CALL    A2HEX           ;CONVERT IT TO HEXADECIMAL
        LD      B,A             ;SAVE HEX VALUE IN B
        LD      A,H             ;GET HIGH CHARACTER
        CALL    A2HEX           ;CONVERT IT TO HEXADECIMAL
        RRCA                    ;SHIFT HEX VALUE TO UPPER 4 BITS
        RRCA
        RRCA
        RRCA
        OR      B               ;OR IN LOW HEX VALUE
        RET

        ;----------------------------------------
        ;SUBROUTINE: A2HEX
        ;PURPOSE: CONVERT ASCII DIGIT TO A HEX DIGIT
        ;ENTRY: A = ASCII HEXADECIMAL DIGIT
        ;EXIT:  A = BINARY VALUE OF ASCII DIGIT
        ;REGISTERS USED: A,F
        ;----------------------------------------
A2HEX:
        SUB     '0'             ;SUBTRACT ASCII OFFSET
        CP      10
        JR      C,A2HEX1        ;BRANCH IF A IS A DECIMAL DIGIT
        SUB     7               ;ELSE SUBTRACT OFFSET FOR LETTERS
A2HEX1:
        RET

;                                                                  ;
;                                                                  ;
;       SAMPLE EXECUTION:                                          ;
;                                                                  ;
;                                                                  ;

SC4D:
        ;CONVERT 'C7' TO C7 HEXADECIMAL
        LD      H,'C'
        LD      L,'7'
        CALL    HEX2BN          ;A=C7H

        ;CONVERT '2F' TO 2F HEXADECIMAL
        LD      H,'2'
        LD      L,'F'
        CALL    HEX2BN          ;A=2FH

        ;CONVERT '2A' TO 2A HEXADECIMAL
        LD      H,'2'
        LD      L,'A'
        CALL    HEX2BN          ;A=2AH

        JR      SC4D

        END
```

Conversion of a Binary Number to Decimal ASCII (BN2DEC) 4E

Converts a 16-bit signed binary number into an ASCII string. The string consists of the length of the number in bytes, an ASCII minus sign (if needed), and the ASCII digits. Note that the length is a binary number, not an ASCII number.

Procedure: The program takes the absolute value of the number if it is negative. The program then keeps dividing the absolute value by 10 until the quotient becomes 0. It converts each digit of the quotient to ASCII by adding ASCII 0 and concatenates the digits along with an ASCII minus sign (in front) if the original number was negative.

Registers Used: AF, BC, DE, HL
Execution Time: Approximately 7200 cycles
Program Size: 107 bytes
Data Memory Required: Four bytes anywhere in memory for the buffer pointer (two bytes starting at address BUFPTR), the length of the buffer (one byte at address CURLEN), and the sign of the original value (one byte at address NGFLAG). This data memory does not include the output buffer which should be seven bytes long.

Entry Conditions

Base address of output buffer in HL
Value to convert in DE

Exit Conditions

Order in buffer:

Length of the string in bytes (a binary number)
ASCII – (if original number was negative)
ASCII digits (most significant digit first)

Examples

1. Data: Value to convert = $3EB7_{16}$

   Result (in output buffer):

   05 (number of bytes in buffer)
   31 (ASCII 1)
   36 (ASCII 6)
   30 (ASCII 0)
   35 (ASCII 5)
   35 (ASCII 5)
   That is, $3EB7_{16} = 16055_{10}$

2. Data: Value to convert = $FFC8_{16}$

   Result (in output buffer):

   03 (number of bytes in buffer)
   2D (ASCII –)
   35 (ASCII 5)
   36 (ASCII 6)
   That is, $FFC8_{16} = -56_{10}$, when considered as a signed two's complement number

```
;                                                               ;
;                                                               ;
;                                                               ;
;                                                               ;
;       Title           Binary to decimal ASCII                 ;
;       Name:           BN2DEC                                  ;
;                                                               ;
;                                                               ;

;                                                               ;
;       Purpose:        Convert a 16-bit signed binary number   ;
;                       to ASCII data                           ;
;                                                               ;
;       Entry:          Register H = High byte of output buffer address ;
;                       Register L = Low byte of output buffer address  ;
;                       Register D = High byte of value to convert      ;
;                       Register E = Low byte of value to convert       ;
;                                                               ;
;       Exit:           The first byte of the buffer is the length,     ;
;                       followed by the characters.             ;
;                                                               ;
;       Registers used: AF,BC,DE,HL                             ;
;                                                               ;
;       Time:           Approximately 7,200 cycles              ;
;                                                               ;
;       Size:           Program 107 bytes                       ;
;                       Data      4 bytes                       ;
;                                                               ;
;                                                               ;

BN2DEC:
        ;SAVE PARAMETERS
        LD      (BUFPTR),HL     ;STORE THE BUFFER POINTER
        EX      DE,HL
        LD      A,0
        LD      (CURLEN),A      ;CURRENT BUFFER LENGTH IS 0
        LD      A,H
        LD      (NGFLAG),A      ;SAVE SIGN OF VALUE
        OR      A               ;SET FLAGS FROM VALUE
        JP      P,CNVERT        ;JUMP IF VALUE IS POSITIVE
        EX      DE,HL           ;ELSE TAKE ABSOLUTE VALUE (0 - VALUE)
        LD      HL,0
        OR      A               ;CLEAR CARRY
        SBC     HL,DE           ;SUBTRACT VALUE FROM 0

        ;CONVERT VALUE TO A STRING
CNVERT:
        ;HL := HL DIV 10  (DIVIDEND,QUOTIENT)
        ;DE := HL MOD 10  (REMAINDER)
        LD      E,0             ;REMAINDER = 0
        LD      B,16            ;16 BITS IN DIVIDEND
        OR      A               ;CLEAR CARRY TO START
DVLOOP:
        ;SHIFT THE NEXT BIT OF THE QUOTIENT INTO BIT 0 OF THE DIVIDEND
        ;SHIFT NEXT MOST SIGNIFICANT BIT OF DIVIDEND INTO
        ;  LEAST SIGNIFICANT BIT OF REMAINDER
        ;HL HOLDS BOTH DIVIDEND AND QUOTIENT. QUOTIENT IS SHIFTED
        ;  IN AS THE DIVIDEND IS SHIFTED OUT.
        ;E IS THE REMAINDER.

        ;DO A 24-BIT SHIFT LEFT, SHIFTING
        ;  CARRY TO L, L TO H, H TO E
        RL      L               ;CARRY (NEXT BIT OF QUOTIENT) TO BIT 0
        RL      H               ;SHIFT HIGH BYTE
        RL      E               ;SHIFT NEXT BIT OF DIVIDEND

        ;IF REMAINDER IS 10 OR MORE, NEXT BIT OF
        ;  QUOTIENT IS 1 (THIS BIT IS PLACED IN CARRY)
        LD      A,E
        SUB     10              ;SUBTRACT 10 FROM REMAINDER
        CCF                     ;COMPLEMENT CARRY
                                ; (THIS IS NEXT BIT OF QUOTIENT)
        JR      NC,DECCNT       ;JUMP IF REMAINDER IS LESS THAN 10
        LD      E,A             ;OTHERWISE REMAINDER = DIFFERENCE
                                ; BETWEEN PREVIOUS REMAINDER AND 10

DECCNT:
        DJNZ    DVLOOP          ;CONTINUE UNTIL ALL BITS ARE DONE

        ;SHIFT LAST CARRY INTO QUOTIENT
        RL      L               ;LAST BIT OF QUOTIENT TO BIT 0
        RL      H

        ;INSERT THE NEXT CHARACTER IN ASCII
CHINS:
        LD      A,E
        ADD     A,'0'           ;CONVERT 0...9 TO ASCII '0'...'9'
        CALL    INSERT

        ;IF QUOTIENT IS NOT 0 THEN KEEP DIVIDING
        LD      A,H
        OR      L
        JR      NZ,CNVERT

EXIT:
        LD      A,(NGFLAG)
        OR      A
        JP      P,POS           ;BRANCH IF ORIGINAL VALUE WAS POSITIVE
        LD      A,'-'           ;ELSE
        CALL    INSERT          ; PUT A MINUS SIGN IN FRONT

POS:
        RET                     ;RETURN
        ;--------------------------------
        ;SUBROUTINE: INSERT
        ;PURPOSE: INSERT THE CHARACTER IN REGISTER A AT THE
        ;         FRONT OF THE BUFFER
        ;ENTRY: CURLEN = LENGTH OF BUFFER
        ;       BUFPTR = CURRENT ADDRESS OF LAST CHARACTER IN BUFFER
        ;EXIT:  REGISTER A INSERTED IMMEDIATELY AFTER LENGTH BYTE
        ;REGISTERS USED: AF,B,C,D,E
        ;--------------------------------
INSERT:
        PUSH    HL              ;SAVE HL
        PUSH    AF              ;SAVE CHARACTER TO INSERT

        ;MOVE ENTIRE BUFFER UP 1 BYTE IN MEMORY
        LD      HL,(BUFPTR)     ;GET BUFFER POINTER
        LD      D,H             ;HL = SOURCE (CURRENT END OF BUFFER)
        LD      E,L
        INC     DE              ;DE = DESTINATION (CURRENT END + 1)
        LD      (BUFPTR),DE     ;STORE NEW BUFFER POINTER
        LD      A,(CURLEN)
        OR      A               ;TEST FOR CURLEN = 0
        JR      Z,EXITMR        ;JUMP IF ZERO (NOTHING TO MOVE,
                                ; JUST STORE THE CHARACTER)
        LD      C,A             ;BC = LOOP COUNTER
        LD      B,0
        LDDR                    ;MOVE ENTIRE BUFFER UP 1 BYTE

EXITMR:
        LD      A,(CURLEN)      ;INCREMENT CURRENT LENGTH BY 1
        INC     A
        LD      (CURLEN),A
        LD      (HL),A          ;UPDATE LENGTH BYTE OF BUFFER
        EX      DE,HL           ;HL POINTS TO FIRST CHARACTER IN BUFFER
        POP     AF              ;GET CHARACTER TO INSERT
        LD      (HL),A          ;INSERT CHARACTER AT FRONT OF BUFFER
        POP     HL              ;RESTORE HL
        RET

        ;DATA
BUFPTR: DS      2               ;ADDRESS OF LAST CHARACTER IN BUFFER
CURLEN: DS      1               ;CURRENT LENGTH OF BUFFER
NGFLAG: DS      1               ;SIGN OF ORIGINAL VALUE


;                                                                ;
;                                                                ;
;       SAMPLE EXECUTION:                                        ;
;                                                                ;
;                                                                ;

SC4E:
        ;CONVERT 0 TO '0'
        LD      HL,BUFFER       ;HL = BASE ADDRESS OF BUFFER
        LD      DE,0            ;DE = 0
        CALL    BN2DEC          ;CONVERT
                                ; BUFFER SHOULD = '0'
        ;CONVERT 32767 TO '32767'
        LD      HL,BUFFER       ;HL = BASE ADDRESS OF BUFFER
        LD      DE,32767        ;DE = 32767
        CALL    BN2DEC          ;CONVERT
                                ; BUFFER SHOULD = '32767'

        ;CONVERT -32768 TO '-32768'
        LD      HL,BUFFER       ;HL = BASE ADDRESS OF BUFFER
        LD      DE,-32768       ;DE = -32768
        CALL    BN2DEC          ;CONVERT
        JR      SC4E            ; BUFFER SHOULD = '-32768'

BUFFER: DS      7               ;7-BYTE BUFFER

        END
```

Conversion of ASCII Decimal to Binary (DEC2BN) 4F

Converts an ASCII string consisting of the length of the number (in bytes), a possible ASCII − or + sign, and a series of ASCII digits to two bytes of binary data. Note that the length is an ordinary binary number, not an ASCII number.

Procedure: The program sets a flag if the first ASCII character is a minus sign and skips over a leading plus sign. It then converts each subsequent digit to decimal by subtracting ASCII 0, multiplies the previous digits by 10 (using the fact that $10 = 8 + 2$, so a multiplication by 10 can be reduced to left shifts and additions), and adds the new digit to the product. Finally, the program subtracts the result from 0 if the original number was negative. The program exits immediately, setting the Carry flag, if it finds something other than a leading sign or a decimal digit in the string.

Registers Used: AF, BC, DE, HL

Execution Time: Approximately 152 cycles per byte plus a maximum of 186 cycles overhead

Program Size: 79 bytes

Data Memory Required: One byte anywhere in RAM (address NGFLAG) for a flag indicating the sign of the number

Special Cases:

1. If the string contains something other than a leading sign or a decimal digit, the program returns with the Carry flag set to 1. The result in HL is invalid.

2. If the string contains only a leading sign (ASCII + or ASCII −), the program returns with the Carry flag set to 1 and a result of 0.

Entry Conditions

Base address of string in HL

Exit Conditions

Binary value in HL

Carry flag is 0 if the string was valid; Carry flag is 1 if the string contained an invalid character.

Note that the result is a signed two's complement 16-bit number.

Examples

1. Data: String consists of

   04 (number of bytes in string)
   31 (ASCII 1)
   32 (ASCII 2)
   33 (ASCII 3)
   34 (ASCII 4)

   That is, the number is $+1{,}234_{10}$

   Result: (H) = 04_{16} (more significant byte of binary data)
   (L) = $D2_{16}$ (less significant byte of binary data)
   That is, the number $+1{,}234_{10} = 04D2_{16}$

2. Data: String consists of

   06 (number of bytes in string)
   2D (ASCII −)
   33 (ASCII 3)
   32 (ASCII 2)
   37 (ASCII 7)
   35 (ASCII 5)
   30 (ASCII 0)

   That is, the number is $-32{,}750_{10}$

   Result: (H) = 80_{16} (more significant byte of binary data)
   (L) = 12_{16} (less significant byte of binary data)
   That is, the number $-32{,}750_{10} = 8012_{16}$

```
;                                                                      ;
;                                                                      ;
;                                                                      ;
;                                                                      ;
;       Title               Decimal ASCII to binary                    ;
;       Name:               DEC2BN                                     ;
;                                                                      ;
;                                                                      ;

;                                                                      ;
;       Purpose:            Convert ASCII characters to two bytes of binary ;
;                           data                                       ;
;                                                                      ;
;       Entry:              HL = Base address of input buffer          ;
;                                                                      ;
;       Exit:               HL = Binary value                          ;
;                           if no errors then                          ;
;                             Carry = 0                                ;
;                           else                                       ;
;                             Carry = 1                                ;
;                                                                      ;
;       Registers used: AF,BC,DE,HL                                    ;
;                                                                      ;
;       Time:               Approximately 152 cycles per byte plus     ;
;                           a maximum of 186 cycles overhead           ;
;                                                                      ;
;       Size:               Program  79 bytes                          ;
;                           Data      1 byte                           ;
;                                                                      ;
;                                                                      ;

DEC2BN:
        ;INITIALIZE - SAVE LENGTH, CLEAR SIGN AND VALUE
        LD      A,(HL)          ;SAVE LENGTH IN B
        LD      B,A
        INC     HL              ;POINT TO BYTE AFTER LENGTH
                SUB A
        LD      (NGFLAG),A      ;ASSUME NUMBER IS POSITIVE
        LD      DE,0            ;START WITH VALUE = 0

        ;CHECK FOR EMPTY BUFFER
                OR B            ;IS BUFFER LENGTH ZERO?
        JR      Z,EREXIT        ;YES, EXIT WITH VALUE = 0

        ;CHECK FOR MINUS OR PLUS SIGN IN FRONT
INIT1:
        LD      A,(HL)          ;GET FIRST CHARACTER
        CP      '-'             ;IS IT A MINUS SIGN?
        JR      NZ,PLUS         ;NO, BRANCH
        LD      A,0FFH
        LD      (NGFLAG),A      ;YES, MAKE SIGN OF NUMBER NEGATIVE
        JR      SKIP            ;SKIP OVER MINUS SIGN

PLUS:
        CP      '+'             ;IS FIRST CHARACTER A PLUS SIGN?
        JR      NZ,CHKDIG       ;NO, START CONVERSION
SKIP:   INC     HL              ;SKIP OVER THE SIGN BYTE
        DEC     B               ;DECREMENT COUNT
        JR      Z,EREXIT        ;ERROR EXIT IF ONLY A SIGN IN BUFFER

        ;CONVERSION LOOP
        ; CONTINUE UNTIL THE BUFFER IS EMPTY
        ; OR A NON-NUMERIC CHARACTER IS FOUND
CNVERT:
        LD      A,(HL)          ;GET NEXT CHARACTER
CHKDIG: SUB     '0'
        JR      C,EREXIT        ;ERROR IF < '0' (NOT A DIGIT)
        CP      9+1
        JR      NC,EREXIT       ;ERROR IF > '9' (NOT A DIGIT)
        LD      C,A             ;CHARACTER IS DIGIT, SAVE IT

        ;VALID DECIMAL DIGIT SO
        ;  VALUE := VALUE * 10
        ;   = VALUE * (8 + 2)
        ;   = (VALUE * 8) + (VALUE * 2)
        PUSH    HL              ;SAVE BUFFER POINTER
        EX      DE,HL           ;HL = VALUE
        ADD     HL,HL           ; * 2
        LD      E,L             ;SAVE TIMES 2 IN DE
        LD      D,H
        ADD     HL,HL           ; * 4
        ADD     HL,HL           ; * 8
        ADD     HL,DE           ;VALUE = VALUE * (8+2)

        ;ADD IN THE NEXT DIGIT
        ; VALUE := VALUE + DIGIT
        LD      E,C             ;MOVE NEXT DIGIT TO E
        LD      D,0             ; HIGH BYTE IS 0
        ADD     HL,DE           ;ADD DIGIT TO VALUE
        EX      DE,HL           ;DE = VALUE
        POP     HL              ;POINT TO NEXT CHARACTER
        INC     HL
        DJNZ    CNVERT          ;CONTINUE CONVERSION

        ;CONVERSION IS COMPLETE, CHECK SIGN
        EX      DE,HL           ;HL = VALUE
        LD      A,(NGFLAG)
        OR      A
        JR      Z,OKEXIT        ;JUMP IF THE VALUE WAS POSITIVE
        EX      DE,HL           ;ELSE REPLACE VALUE WITH -VALUE
        LD      HL,0
        OR      A               ;CLEAR CARRY
        SBC     HL,DE           ;SUBTRACT VALUE FROM 0

        ;NO ERRORS, EXIT WITH CARRY CLEAR
OKEXIT:
        OR      A               ;CLEAR CARRY
        RET

        ;AN ERROR, EXIT WITH CARRY SET
EREXIT:
        EX      DE,HL           ;HL = VALUE
        SCF                     ;SET CARRY TO INDICATE ERROR
        RET

        ;DATA
NGFLAG: DS      1               ;SIGN OF NUMBER


;                                                                       ;
;                                                                       ;
;       SAMPLE EXECUTION:                                               ;
;                                                                       ;
;                                                                       ;

SC4F:
        ;CONVERT '1234'
        LD      HL,S1           ;HL = BASE ADDRESS OF S1
        CALL    DEC2BN
                                ;H = 04, L = D2 HEX

        ;CONVERT '+32767'
        LD      HL,S2           ;HL = BASE ADDRESS OF S2
        CALL    DEC2BN
                                ;H = 7F, L = FF HEX

        ;CONVERT '-32768'
        LD      HL,S3           ;HL = BASE ADDRESS OF S3
        CALL    DEC2BN
                                ;H = 80 HEX, L = 00 HEX

        JR      SC4F

S1:     DB      4,'1234'
S2:     DB      6,'+32767'
S3:     DB      6,'-32768'

        END
```

Lower-Case to Upper-Case Translation (LC2UC)

Converts an ASCII lower-case letter to its upper-case equivalent.

Procedure: The program uses comparisons to determine whether the data is an ASCII lower-case letter. If it is, the program subtracts 20_{16} from it, thus converting it to its upper-case equivalent. If it is not, the program leaves it unchanged.

Registers Used: AF
Execution Time: 45 cycles if the original character is a lower-case letter, fewer cycles otherwise
Program Size: 11 bytes
Data Memory Required: None

Entry Conditions

Character in A

Exit Conditions

If an ASCII lower-case letter is present in A, then its upper-case equivalent is returned in A. In all other cases, A is unchanged.

Examples

1. Data: (A) = 62_{16} (ASCII b)
   Result: (A) = 42_{16} (ASCII B)

2. Data: (A) = 54_{16} (ASCII T)
   Result: (A) = 54_{16} (ASCII T)

```
;                                                               ;
;                                                               ;
;                                                               ;
;                                                               ;
;       Title           Lower-case to upper-case translation    ;
;       Name:           LC2UC                                   ;
;                                                               ;
;                                                               ;

;                                                               ;
;       Purpose:        Convert one ASCII character to upper case from  ;
;                       lower case if necessary                 ;
;                                                               ;
;       Entry:          Register A = Lower-case ASCII character  ;
;                                                               ;
;       Exit:           Register A = Upper-case ASCII character if A    ;
;                                    is lower case, else A is unchanged ;
;                                                               ;
;       Registers used: AF                                      ;
;                                                                          ;
;       Time:              45 cycles if A is lower case, less otherwise   ;
;                                                                          ;
;       Size:              Program 11 bytes                                ;
;                          Data       none                                 ;
;                                                                          ;
;                                                                          ;

LC2UC:
        CP      'a'
        JR      C,EXIT          ;BRANCH IF < 'a' (NOT LOWER CASE)
        CP      'z'+1
        JR      NC,EXIT         ;BRANCH IF > 'z' (NOT LOWER CASE)
        SUB     'a'-'A'         ;CHANGE 'a'..'z' into 'A'..'Z'
EXIT:
        RET


;                                                                          ;
;                                                                          ;
;       SAMPLE EXECUTION:                                                  ;
;                                                                          ;
;                                                                          ;

SC4G:
        ;CONVERT LOWER CASE E TO UPPER CASE
        LD      A,'e'
        CALL    LC2UC           ;A='E'=45H

        ;CONVERT LOWER CASE Z TO UPPER CASE
        LD      A,'z'
        CALL    LC2UC           ;A='Z'=5AH

        ;CONVERT UPPER CASE A TO UPPER CASE A
        LD      A,'A'
        CALL    LC2UC           ;A='A'=41H
        JR      SC4G

        END
```

ASCII to EBCDIC Conversion (ASC2EB)

Converts an ASCII character to its EBCDIC equivalent.

Procedure: The program uses a simple table lookup with the data as the index and address EBCDIC as the base. A printable ASCII character with no EBCDIC equivalent is translated to an EBCDIC space (40_{16}); a non-printable ASCII character with no EBCDIC equivalent is translated to an EBCDIC NUL (00_{16}).

Registers Used: AF, DE, HL
Execution Time: 55 cycles
Program Size: 11 bytes, plus 128 bytes for the conversion table
Data Memory Required: None

Entry Conditions

ASCII character in A

Exit Conditions

EBCDIC equivalent in A

Examples

1. Data: (A) = 35_{16} (ASCII 5)
   Result: (A) = $F5_{16}$ (EBCDIC 5)

2. Data: (A) = 77_{16} (ASCII w)
   Result: (A) = $A6_{16}$ (EBCDIC w)

3. Data: (A) = $2A_{16}$ (ASCII *)
   Result: (A) = $5C_{16}$ (EBCDIC *)

```
;                                                                    ;
;                                                                    ;
;                                                                    ;
;                                                                    ;
;       Title           ASCII to EBCDIC conversion                   ;
;       Name:           ASC2EB                                       ;
;                                                                    ;
;                                                                    ;

;                                                                    ;
;       Purpose:        Convert an ASCII character to its            ;
;                       corresponding EBCDIC character               ;
;                                                                    ;
;       Entry:          Register A = ASCII character                 ;
;                                                                    ;
;       Exit:           Register A = EBCDIC character                ;
;                                                                    ;
;       Registers used: AF,DE,HL                                     ;
;                                                                    ;
;       Time:           55 cycles                                    ;
;                                                                    ;
;       Size:           Program 11 bytes                             ;
;                       Data    128 bytes for the table              ;
;                                                                    ;
;                                                                    ;

ASC2EB:
        LD      HL,EBCDIC       ;GET BASE ADDRESS OF EBCDIC TABLE
        AND     01111111B       ;BE SURE BIT 7 = 0
        LD      E,A             ;USE ASCII AS INDEX INTO EBCDIC TABLE
        LD      D,0
        ADD     HL,DE
        LD      A,(HL)          ;GET EBCDIC
        RET

;ASCII TO EBCDIC TABLE
;  A PRINTABLE ASCII CHARACTER WITH NO EBCDIC EQUIVALENT IS
;  TRANSLATED TO AN EBCDIC SPACE (040H), A NONPRINTABLE ASCII CHARACTER
;  WITH NO EQUIVALENT IS TRANSLATED TO A EBCDIC NUL (000H)
EBCDIC:
        ;       NUL  SOH  STX  ETX  EOT  ENQ  ACK  BEL          ;ASCII
        DB      000H,001H,002H,003H,037H,02DH,02EH,02FH         ;EBCDIC
        ;       BS   HT   LF   VT   FF   CR   SO   SI           ;ASCII
        DB      016H,005H,025H,00BH,00CH,00DH,00EH,00FH         ;EBCDIC
        ;       DLE  DC1  DC2  DC3  DC4  NAK  SYN  ETB          ;ASCII
        DB      010H,011H,012H,013H,03CH,03DH,032H,026H         ;EBCDIC
        ;       CAN  EM   SUB  ESC  IFS  IGS  IRS  IUS          ;ASCII
        DB      018H,019H,03FH,027H,01CH,01DH,01EH,01FH         ;EBCDIC
        ;       SPACE !   "    #    $    %    &    '            ;ASCII
        DB      040H,05AH,07FH,07BH,05BH,06CH,050H,00DH         ;EBCDIC
        ;        (    )   *    +    ,    _    .    /            ;ASCII
        DB      04DH,05DH,05CH,04EH,06BH,060H,04BH,061H         ;EBCDIC
        ;        0    1    2    3    4    5    6    7           ;ASCII
        DB      0F0H,0F1H,0F2H,0F3H,0F4H,0F5H,0F6H,0F7H         ;EBCDIC
        ;        8    9    :    ;    <    =    >    ?           ;ASCII
        DB      0F8H,0F9H,07AH,05EH,04CH,07EH,06EH,06FH         ;EBCDIC
        ;        @    A    B    C    D    E    F    G           ;ASCII
        DB      07CH,0C1H,0C2H,0C3H,0C4H,0C5H,0C6H,0C7H         ;EBCDIC
        ;        H    I    J    K    L    M    N    O           ;ASCII
        DB      0C8H,0C9H,0D1H,0D2H,0D3H,0D4H,0D5H,0D6H         ;EBCDIC
        ;        P    Q    R    S    T    U    V    W           ;ASCII
        DB      0D7H,0D8H,0D9H,0E2H,0E3H,0E4H,0E5H,0E6H         ;EBCDIC
        ;        X    Y    Z    [    \    ]    ^    -           ;ASCII
        DB      0E7H,0E8H,0E9H,040H,0E0H,040H,040H,06DH         ;EBCDIC
        ;        `    a    b    c    d    e    f    g           ;ASCII
        DB      009H,081H,082H,083H,084H,085H,086H,087H         ;EBCDIC
        ;        h    i    j    k    l    m    n    o           ;ASCII
        DB      088H,089H,091H,092H,093H,094H,095H,096H         ;EBCDIC
        ;        p    q    r    s    t    u    v    w           ;ASCII
        DB      097H,098H,099H,0A2H,0A3H,0A4H,0A5H,0A6H         ;EBCDIC
        ;        x    y    z    {    |    }    ~    DEL         ;ASCII
        DB      0A7H,0A8H,0A9H,0C0H,06AH,0D0H,0A1H,007H         ;EBCDIC
```

```
;                                                                        ;
;                                                                        ;
;       SAMPLE EXECUTION:                                                ;
;                                                                        ;
;                                                                        ;

SC4H:
        ;CONVERT ASCII 'A' TO EBCDIC
        LD      A,'A'           ;ASCII 'A'
        CALL    ASC2EB          ;EBCDIC 'A' = 0C1H

        ;CONVERT ASCII '1' TO EBCDIC
        LD      A,'1'           ;ASCII '1'
        CALL    ASC2EB          ;EBCDIC '1' = 0F1H

        ;CONVERT ASCII 'a' TO EBCDIC
        LD      A,'a'           ;ASCII 'a'
        CALL    ASC2EB          ;EBCDIC 'a' = 081H

        JR      SC4H

        END
```

EBCDIC to ASCII Conversion (EB2ASC) 41

Converts an EBCDIC character to its ASCII equivalent.

Procedure: The program uses a simple table lookup with the data as the index and address ASCII as the base. A printable EBCDIC character with no ASCII equivalent is translated to an ASCII space (20_{16}); a non-printable EBCDIC character with no ASCII equivalent is translated to an ASCII NUL (00_{16}).

Registers Used: AF, DE, HL
Execution Time: 48 cycles
Program Size: 9 bytes, plus 256 bytes for the conversion table
Data Memory Required: None

Entry Conditions

EBCDIC character in A

Exit Conditions

ASCII equivalent in A

Examples

1.	Data:	(A) = 85_{16} (EBCDIC e)	2.	Data:	(A) = $4E_{16}$ (EBCDIC +)
	Result:	(A) = 65_{16} (ASCII e)		Result:	(A) = $2B_{16}$ (ASCII +)

```
;                                                                  ;
;                                                                  ;
;                                                                  ;
;                                                                  ;
;       Title           EBCDIC to ASCII conversion                 ;
;       Name:           EB2ASC                                     ;
;                                                                  ;
;                                                                  ;


;                                                                  ;
;       Purpose:        Convert an EBCDIC character to its         ;
;                       corresponding ASCII character              ;
;                                                                  ;
;       Entry:          Register A = EBCDIC character              ;
;                                                                  ;
;       Exit:           Register A = ASCII character               ;
;                                                                  ;
;       Registers used: AF,DE,HL                                   ;
;                                                                  ;
;       Time:           48 cycles                                       ;
;                                                                       ;
;       Size:           Program 9 bytes                                 ;
;                       Data    256 bytes for the table                 ;
;                                                                       ;
;                                                                       ;

EB2ASC:
        LD      HL,ASCII        ;GET BASE ADDRESS OF ASCII TABLE
        LD      E,A             ;USE EBCDIC AS INDEX
        LD      D,0
        ADD     HL,DE
        LD      A,(HL)          ;GET ASCII CHARACTER
        RET

;EBCDIC TO ASCII TABLE
; A PRINTABLE EBCDIC CHARACTER WITH NO ASCII EQUIVALENT IS
; TRANSLATED TO AN ASCII SPACE (020H), A NONPRINTABLE EBCDIC CHARACTER
; WITH NO EQUIVALENT IS TRANSLATED TO AN ASCII NUL (000H)
ASCII:
        ;       NUL  SOH  STX  ETX        HT        DEL          ;EBCDIC
        DB      000H,001H,002H,003H,000H,009H,000H,07FH         ;ASCII
        ;                       VT   FF   CR   SO   SI           ;EBCDIC
        DB      000H,000H,000H,00BH,00CH,00DH,00EH,00FH         ;ASCII
        ;       DLE  DC1  DC2  DC3             BS                ;EBCDIC
        DB      010H,011H,012H,013H,000H,000H,008H,000H         ;ASCII
        ;       CAN  EM             IFS  IGS  IRS  IUS           ;EBCDIC
        DB      018H,019H,000H,000H,01CH,01DH,01EH,01FH         ;ASCII
        ;                                 LF  ETB  ESC           ;EBCDIC
        DB      000H,000H,000H,000H,000H,00AH,017H,01BH         ;ASCII
        ;                                ENQ  ACK  BEL           ;EBCDIC
        DB      000H,000H,000H,000H,000H,005H,006H,007H         ;ASCII
        ;                 SYN                      EOT           ;EBCDIC
        DB      000H,000H,016H,000H,000H,000H,000H,004H         ;ASCII
        ;                           DC4  NAK       SUB           ;EBCDIC
        DB      000H,000H,000H,000H,014H,015H,000H,01AH         ;ASCII
        ;       SPACE                                           ;EBCDIC
        DB      ' ' ,000H,000H,000H,000H,000H,000H,000H         ;ASCII
        ;                       .    <    (    +                ;EBCDIC
        DB      000H,000H,' ' ,'.' ,'<' ,'(' ,'+' ,' '          ;ASCII
        ;        &                                              ;EBCDIC
        DB      '&' ,000H,000H,000H,000H,000H,000H,000H         ;ASCII
        ;                  !    $    *    )    ;                ;EBCDIC
        DB      000H,000H,'!' ,'$' ,'*' ,')' ,';' ,' '          ;ASCII
        ;        _    /                                         ;EBCDIC
        DB      '_' ,'/' ,000H,000H,000H,000H,000H,000H         ;ASCII
        ;                  ¦    ,    %    -    >    ?           ;EBCDIC
        DB      000H,000H,'¦' ,',' ,'%' ,'-' ,'>' ,'?'          ;ASCII
        ;                                                       ;EBCDIC
        DB      000H,000H,000H,000H,000H,000H,000H,000H         ;ASCII
        ;             `    :    #    @    '    =    "           ;EBCDIC
        DB      000H,'`' ,':' ,'#' ,'@' ,'''','=' ,'"'          ;ASCII
        ;             a    b    c    d    e    f    g           ;EBCDIC
        DB      000H,'a' ,'b' ,'c' ,'d' ,'e' ,'f' ,'g'          ;ASCII
        ;        h    i                                         ;EBCDIC
        DB      'h' ,'i' ,000H,000H,000H,000H,000H,000H            ;ASCII
        ;            j    k    l    m    n    o    p               ;EBCDIC
        DB      000H,'j' ,'k' ,'l' ,'m' ,'n' ,'o' ,'p'             ;ASCII
        ;        q    r                                            ;EBCDIC
        DB      'q' ,'r' ,000H,000H,000H,000H,000H,000H            ;ASCII
        ;             ~    s    t    u    v    w    x              ;EBCDIC
        DB      000H,'~' ,'s' ,'t' ,'u' ,'v' ,'w', 'x'             ;ASCII
        ;        y    z                                            ;EBCDIC
        DB      'y' ,'z' ,000H,000H,000H,000H,000H,000H            ;ASCII
        ;                                                          ;EBCDIC
        DB      000H,000H,000H,000H,000H,000H,000H,000H            ;ASCII
        ;                                                          ;EBCDIC
        DB      000H,000H,000H,000H,000H,000H,000H,000H            ;ASCII
        ;        {    A    B    C    D    E    F    G              ;EBCDIC
        DB      '{' ,'A' ,'B' ,'C' ,'D' ,'E' ,'F' ,'G'             ;ASCII
        ;        H    I                                            ;EBCDIC
        DB      'H' ,'I' ,000H,000H,000H,000H,000H,000H            ;ASCII
        ;        }    J    K    L    M    N    O    P              ;EBCDIC
        DB      '}' ,'J' ,'K' ,'L' ,'M' ,'N' ,'O' ,'P'             ;ASCII
        ;        Q    R                                            ;EBCDIC
        DB      'Q' ,'R' ,000H,000H,000H,000H,000H,000H            ;ASCII
        ;        \         S    T    U    V    W    X              ;EBCDIC
        DB      '\' ,000H,'S' ,'T' ,'U' ,'V' ,'W', 'X'             ;ASCII
        ;        Y    Z                                            ;EBCDIC
        DB      'Y' ,'Z' ,000H,000H,000H,000H,000H,000H            ;ASCII
        ;        0    1    2    3    4    5    6    7              ;EBCDIC
        DB      '0' ,'1' ,'2' ,'3' ,'4' ,'5' ,'6' ,'7'             ;ASCII
        ;        9                                                 ;EBCDIC
        DB      '9' ,000H,000H,000H,000H,000H,000H,000H            ;ASCII


;                                                                          ;
;                                                                          ;
;       SAMPLE EXECUTION:                                                  ;
;                                                                          ;
;                                                                          ;

SC4I:
        ;CONVERT EBCDIC 'A' TO ASCII
        LD      A,0C1H          ;EBCDIC 'A'
        CALL    EB2ASC          ;ASCII 'A' = 041H

        ;CONVERT EBCDIC '1' TO ASCII
        LD      A,0F1H          ;EBCDIC '1'
        CALL    EB2ASC          ;ASCII '1' = 031H

        ;CONVERT EBCDIC 'a' TO ASCII
        LD      A,081H          ;EBCDIC 'a'
        CALL    EB2ASC          ;ASCII 'a' = 061H

        JR      SC4I

        END
```

Memory Fill (MFILL)

Places a specified value in each byte of a memory area of known size, starting at a given address.

Procedure: The program stores the specified value in the first byte and then uses a block move to fill the remaining bytes. The block move simply transfers the value a byte ahead during each iteration.

Registers Used: AF, BC, DE, HL

Execution Time: Approximately 21 cycles per byte plus 50 cycles overhead

Program Size: 11 bytes

Data Memory Required: None

Special Cases:

1. A size of 0000_{16} is interpreted as 10000_{16}. It therefore causes the program to fill 65,536 bytes with the specified value.
2. Filling areas occupied or used by the program itself will cause unpredictable results. Obviously, filling the stack area requires special caution, since the return address is saved there.

Entry Conditions

Starting address of memory area in HL

Area size (number of bytes) in BC

Value to be placed in memory in A

Exit Conditions

The area from the base address through the number of bytes given by the area size is filled with the specified value. The area thus filled starts at BASE and continues through BASE + SIZE − 1 (BASE is the base address and SIZE is the area size).

Examples

1. Data: Value = FF_{16}
   Area size (in bytes) = 0380_{16}
   Base address = $1AE0_{16}$

   Result: FF_{16} placed in addresses $1AE0_{16}$ through $1E5F_{16}$

2. Data: Value = 00_{16} (Z80 operation code for NOP)
   Area size (in bytes) = $1C65_{16}$
   Base address = $E34C_{16}$

   Result: 00_{16} placed in addresses $E34C_{16}$ through $FFB0_{16}$

```
;                                                                       ;
;                                                                       ;
;                                                                       ;
;                                                                       ;
;       Title           Memory fill                                     ;
;       Name:           MFILL                                           ;
;                                                                       ;
;                                                                       ;

;                                                                       ;
;       Purpose:        Fill an area of memory with a value            ;
;                                                                       ;
;       Entry:          Register H = High byte of base address          ;
;                       Register L = Low byte of base address           ;
;                       Register B = High byte of area size             ;
;                       Register C = Low byte of area size              ;
;                       Register A = Value to be placed in memory       ;
;                                                                       ;
;                       Note: A size of 0 is interpreted as 65536       ;
;                                                                       ;
;       Exit:           Area filled with value                          ;
;                                                                       ;
;       Registers used: AF,BC,DE,HL                                     ;
;                                                                       ;
;       Time:           Approximately 21 cycles per byte plus           ;
;                       50 cycles overhead                              ;
;                                                                       ;
;       Size:           Program 11 bytes                                ;
;                       Data    None                                    ;
;                                                                       ;
;                                                                       ;

MFILL:
        LD      (HL),A          ;FILL FIRST BYTE WITH VALUE
        LD      D,H             ;DESTINATION PTR = SOURCE PTR + 1
        LD      E,L
        INC     DE
        DEC     BC              ;ELIMINATE FIRST BYTE FROM COUNT
        LD      A,B             ;ARE THERE MORE BYTES TO FILL?
        OR      C
        RET     Z               ;NO, RETURN - SIZE WAS 1
        LDIR                    ;YES, USE BLOCK MOVE TO FILL REST
                                ; BY MOVING VALUE AHEAD 1 BYTE
        RET


;                                                                       ;
;                                                                       ;
;       SAMPLE EXECUTION:                                               ;
;                                                                       ;
;                                                                       ;

SC5A:
        ;FILL BF1 THROUGH BF1+15 WITH 00
        LD      HL,BF1          ;STARTING ADDRESS
        LD      BC,SIZE1        ;NUMBER OF BYTES
        LD      A,0             ;VALUE TO FILL
        CALL    MFILL           ;FILL MEMORY

        ;FILL BF2 THROUGH BF2+1999 WITH FF
        LD      HL,BF2          ;STARTING ADDRESS
        LD      BC,SIZE2        ;NUMBER OF BYTES
        LD      A,0FFH          ;VALUE TO FILL
        CALL    MFILL           ;FILL MEMORY

        JR      SC5A

SIZE1   EQU     16              ;SIZE OF BUFFER 1 (10 HEX)
SIZE2   EQU     2000            ;SIZE OF BUFFER 2 (07D0 HEX)
BF1:    DS      SIZE1
BF2:    DS      SIZE2

        END
```

Block Move (BLKMOV) 5B

Moves a block of data from a source area to a destination area.

Procedure: The program determines if the base address of the destination area is within the source area. If it is, then working up from the base address would overwrite some source data. To avoid overwriting, the program works down from the highest address (this is sometimes called a *move right*). If the base address of the destination area is not within the source area, the program simply moves the data starting from the lowest address (this is sometimes called a *move left*). An area size (number of bytes to move) of 0000_{16} causes an exit with no memory changed. The program provides automatic address wraparound mod 64K.

Registers Used: AF, BC, DE, HL

Execution Time: 21 cycles per byte plus 97 cycles overhead if data can be moved starting from the lowest address (i.e., left) or 134 cycles overhead if data must be moved starting from the highest address (i.e., right) because of overlap.

Program Size: 27 bytes

Data Memory Required: None

Special Cases:

1. A size (number of bytes to move) of 0 causes an immediate exit with no memory changed.
2. Moving data to or from areas occupied or used by the program itself or by the stack will have unpredictable results.

Entry Conditions

Base address of source area in HL

Base address of destination area in DE

Number of bytes to move in register BC

Exit Conditions

The block of memory is moved from the source area to the destination area. If the number of bytes to be moved is NBYTES, the base address of the destination area is DEST, and the base address of the source area is SOURCE, then the data in addresses SOURCE through SOURCE + NBYTES − 1 is moved to addresses DEST through DEST + NBYTES − 1.

Examples

1. Data: Number of bytes to move = 0200_{16}
   Base address of destination area = $05D1_{16}$
   Base address of source area = $035E_{16}$

   Result: The contents of locations $035E_{16}$ through $055D_{16}$ are moved to $05D1_{16}$ through $07D0_{16}$

2. Data: Number of bytes to move = $1B7A_{16}$
   Base address of destination area = $C946_{16}$
   Base address of source area = $C300_{16}$

   Result: The contents of locations $C300_{16}$ through $DE79_{16}$ are moved to $C946_{16}$ through $E4BF_{16}$

Note that Example 2 is a more difficult problem than Example 1 because the source and destination areas overlap. If, for instance, the program were simply to move data to the destination area starting from the lowest address, it would initially move the contents of $C300_{16}$ to $C946_{16}$. This would destroy the old contents of $C946_{16}$, which are needed later in the move. The solution to this problem is to move the data starting from the highest address if the destination area is above the source area but overlaps it.

```
;                                                                    ;
;                                                                    ;
;                                                                    ;
;                                                                    ;
;       Title           Block Move                                   ;
;       Name:           BLKMOV                                       ;
;                                                                    ;
;                                                                    ;

;                                                                    ;
;       Purpose:        Move data from source to destination         ;
;                                                                    ;
;       Entry:          Register H = High byte of source address     ;
;                       Register L = Low byte of source address      ;
;                       Register D = High byte of destination address    ;
;                       Register E = Low byte of destination address     ;
;                       Register B = High byte of number of bytes to move;
;                       Register C = Low byte of number of bytes to move ;
;                                                                    ;
;       Exit:           Data moved from source to destination        ;
;                                                                    ;
;       Registers used:AF,BC,DE,HL                                   ;
;                                                                    ;
;       Time:           21 cycles per byte plus 97 cycles overhead   ;
;                       if no overlap exists, 134 cycles overhead    ;
;                       if overlap occurs                            ;
;                                                                    ;
;       Size:           Program 27 bytes                             ;
;                                                                    ;
;                                                                    ;

BLKMOV:
        LD      A,B             ;IS SIZE OF AREA 0?
        OR      C
        RET     Z               ;YES, RETURN WITH NOTHING MOVED

        ;DETERMINE IF DESTINATION AREA IS ABOVE SOURCE AREA AND OVERLAPS
        ; IT (OVERLAP CAN BE MOD 64K).  OVERLAP OCCURS IF
        ; STARTING DESTINATION ADDRESS MINUS STARTING SOURCE ADDRESS
        ; (MOD 64K) IS LESS THAN NUMBER OF BYTES TO MOVE
        EX      DE,HL           ;CALCULATE DESTINATION - SOURCE
        PUSH    HL              ;SAVE DESTINATION
        AND     A               ;CLEAR CARRY
        SBC     HL,DE
        AND     A               ;THEN SUBTRACT AREA SIZE
        SBC     HL,BC
        POP     HL              ;RESTORE DESTINATION
        EX      DE,HL
        JR      NC,DOLEFT       ;JUMP IF NO PROBLEM WITH OVERLAP

        ;DESTINATION AREA IS ABOVE SOURCE AREA AND OVERLAPS IT
        ;MOVE FROM HIGHEST ADDRESS TO AVOID DESTROYING DATA
        ADD     HL,BC           ;SOURCE = SOURCE + LENGTH - 1
        DEC     HL
        EX      DE,HL           ;DEST = DEST + LENGTH - 1
        ADD     HL,BC
        DEC     HL
        EX      DE,HL
        LDDR                    ;BLOCK MOVE HIGH TO LOW
        RET

        ;ORDINARY MOVE STARTING AT LOWEST ADDRESS
DOLEFT:
        LDIR                    ;BLOCK MOVE LOW TO HIGH
        RET


;                                                                       ;
;                                                                       ;
;       SAMPLE EXECUTION:                                               ;
;                                                                       ;
;                                                                       ;

SOURCE  EQU     2000H           ;BASE ADDRESS OF SOURCE AREA
DEST    EQU     2010H           ;BASE ADDRESS OF DESTINATION AREA
LEN     EQU     11H             ;NUMBER OF BYTES TO MOVE

        ;MOVE 11 HEX BYTES FROM 2000-2010 HEX TO 2010-2020 HEX
SC5B:
        LD      HL,SOURCE
        LD      DE,DEST
        LD      BC,LEN
        CALL    BLKMOV          ;MOVE DATA FROM SOURCE TO DESTINATION

        JR      SC5B

        END
```

Two-Dimensional Byte Array Indexing (D2BYTE) 5C

Calculates the address of an element of a two-dimensional byte-length array, given the base address of the array, the two subscripts of the element, and the size of a row (that is, the number of columns). The array is assumed to be stored in row major order (that is, by rows) and both subscripts are assumed to begin at 0.

Procedure: The program multiplies the row size (number of columns in a row) times the row subscript (since the elements are stored by rows) and adds the product to the column subscript. It then adds the sum to the base address. The program performs the multiplication using a standard shift-and-add algorithm (see Subroutine 6A).

Registers Used: AF, BC, DE, HL

Execution Time: Approximately 1100 cycles, depending mainly on the amount of time required to perform the multiplication.

Program Size: 44 bytes

Data Memory Required: Four bytes anywhere in memory to hold the return address (two bytes starting at address RETADR) and the column subscript (two bytes starting at address SS2).

Entry Conditions

Order in stack (starting from the top)

Less significant byte of return address
More significant byte of return address

Less significant byte of column subscript
More significant byte of column subscript

Less significant byte of the size of a row (in bytes)
More significant byte of the size of a row (in bytes)

Less significant byte of row subscript
More significant byte of row subscript

Less significant byte of base address of array
More significant byte of base address of array

Exit Conditions

Address of element in HL

Examples

1. Data: Base address = $3C00_{16}$
Column subscript = 0004_{16}
Size of row (number of columns) = 0018_{16}
Row subscript = 0003_{16}

Result: Element address = $3C00_{16} + 0003_{16} * 0018_{16} + 0004_{16} = 3C00_{16} + 0048_{16} + 0004_{16} = 3C4C_{16}$

That is, the address of ARRAY(3,4) is $3C4C_{16}$

Note that all subscripts are hexadecimal ($35_{16} = 53_{10}$).

The general formula is

ELEMENT ADDRESS = ARRAY BASE ADDRESS + ROW SUBSCRIPT * ROW SIZE + COLUMN SUBSCRIPT

2. Data: Base address = $6A4A_{16}$
Column subscript = 0035_{16}
Size of row (number of columns) = 0050_{16}
Row subscript = 0002_{16}

Result: Element address = $6A4A_{16} + 0002_{16} * 0050_{16} + 0035_{16} = 6A4A_{16} + 00A0_{16} + 0035_{16} = 6B1F_{16}$

That is, the address of ARRAY(2,35) is $6B1F_{16}$

Note that we refer to the *size* of the row subscript; the size is the number of consecutive memory addresses for which the subscript has the same value. This is also the number of bytes from the starting address of an element to the starting address of the element with the same column subscript but a row subscript one larger.

```
;                                                                      ;
;                                                                      ;
;                                                                      ;
;                                                                      ;
;       Title           Two-dimensional byte array indexing            ;
;       Name:           D2BYTE                                         ;
;                                                                      ;
;                                                                      ;

;                                                                      ;
;       Purpose:        Given the base address of a byte array, two    ;
;                       subscripts 'I','J', and the size of the first  ;
;                       subscript in bytes, calculate the address of   ;
;                       A[I,J]. The array is assumed to be stored in   ;
;                       row major order (A[0,0], A[0,1],..., A[K,L]),  ;
;                       and both dimensions are assumed to begin at    ;
;                       zero as in the following Pascal declaration:   ;
;                         A:ARRAY[0..2,0..7] OF BYTE;                  ;
;                                                                      ;
;       Entry:          TOP OF STACK                                   ;
;                        Low byte of return address,                   ;
;                        High byte of return address,                  ;
;                        Low byte of second subscript (column element), ;
;                        High byte of second subscript (column element),;
;                        Low byte of first subscript size, in bytes,   ;
;                        High byte of first subscript size, in bytes,  ;
;                        Low byte of first subscript (row element),    ;
;                        High byte of first subscript (row element),   ;
;                        Low byte of array base address,               ;
;                        High byte of array base address,              ;
;                       NOTE:                                          ;
;                         The first subscript size is length of a row  ;
;                         in bytes                                     ;
;                                                                      ;
;       Exit:           Register H = High byte of element address       ;
;                       Register L = Low byte of element address        ;
;                                                                       ;
;       Registers used: AF,BC,DE,HL                                     ;
;                                                                       ;
;       Time:           Approximately 1100 cycles                       ;
;                                                                       ;
;       Size:           Program 44 bytes                                ;
;                       Data     4 bytes                                ;
;                                                                       ;
;                                                                       ;

D2BYTE:
        ;SAVE RETURN ADDRESS
        POP     HL
        LD      (RETADR),HL

        ;GET SECOND SUBSCRIPT
        POP     HL
        LD      (SS2),HL
        ;GET SIZE OF FIRST SUBSCRIPT (ROW LENGTH), FIRST SUBSCRIPT
        POP     DE              ;GET LENGTH OF ROW
        POP     BC              ;GET FIRST SUBSCRIPT

        ;MULTIPLY FIRST SUBSCRIPT * ROW LENGTH USING SHIFT AND ADD
        ; ALGORITHM. PRODUCT IS IN HL
        LD      HL,0            ;PRODUCT = 0
        LD      A,15            ;COUNT = BIT LENGTH - 1
MLP:
        SLA     E               ;SHIFT LOW BYTE OF MULTIPLIER
        RL      D               ;ROTATE HIGH BYTE OF MULTIPLIER
        JR      NC,MLP1         ;JUMP IF MSB OF MULTIPLIER = 0
        ADD     HL,BC           ;ADD MULTIPLICAND TO PARTIAL PRODUCT
MLP1:   ADD     HL,HL           ;SHIFT PARTIAL PRODUCT
        DEC     A
        JR      NZ,MLP          ;CONTINUE THROUGH 15 BITS

        ;DO LAST ADD IF MSB OF MULTIPLIER IS 1
        OR      D               ;SIGN FLAG = MSB OF MULTIPLIER
        JP      P,MLP2
        ADD     HL,BC           ;ADD IN MULTIPLICAND IF SIGN = 1
        ;ADD IN SECOND SUBSCRIPT
MLP2:   LD      DE,(SS2)
        ADD     HL,DE

        ;ADD BASE ADDRESS TO FORM FINAL ADDRESS
        POP     DE              ;GET BASE ADDRESS OF ARRAY
        ADD     HL,DE           ;ADD BASE TO INDEX

        ;RETURN TO CALLER
        LD      DE,(RETADR)     ;RESTORE RETURN ADDRESS TO STACK
        PUSH    DE
        RET

        ;DATA
```

```
RETADR: DS      2                   ;TEMPORARY FOR RETURN ADDRESS
SS2:    DS      2                   ;TEMPORARY FOR SECOND SUBSCRIPT


;                                                                      ;
;                                                                      ;
;       SAMPLE EXECUTION:                                              ;
;                                                                      ;
;                                                                      ;

SC5C:
        LD      HL,ARY              ;PUSH BASE ADDRESS OF ARRAY
        PUSH    HL
        LD      HL,(SUBS1)          ;PUSH FIRST SUBSCRIPT
        PUSH    HL
        LD      HL,(SSUBS1)         ;PUSH SIZE OF FIRST SUBSCRIPT
        PUSH    HL
        LD      HL,(SUBS2)          ;PUSH SECOND SUBSCRIPT
        PUSH    HL
        CALL    D2BYTE              ;CALCULATE ADDRESS
                                    ;FOR THE INITIAL TEST DATA
                                    ;HL = ADDRESS OF ARY(2,4)
                                    ;   = ARY + (2*8) + 4
                                    ;   = ARY + 20 (CONTENTS ARE 21)
                                    ;NOTE BOTH SUBSCRIPTS START AT 0

        JR      SC5C

        ;DATA
SUBS1:  DW      2                   ;SUBSCRIPT 1
SSUBS1: DW      8                   ;SIZE OF SUBSCRIPT 1
SUBS2:  DW      4                   ;SUBSCRIPT 2

;THE ARRAY (3 ROWS OF 8 COLUMNS)
ARY:    DB      1 ,2 ,3 ,4 ,5 ,6 ,7 ,8
        DB      9 ,10,11,12,13,14,15,16
        DB      17,18,19,20,21,22,23,24

        END
```

Two-Dimensional Word Array Indexing (D2WORD) 5D

Calculates the starting address of an element of a two-dimensional word-length (16-bit) array, given the base address of the array, the two subscripts of the element, and the size of a row in bytes. The array is assumed to be stored in row major order (that is, by rows) and both subscripts are assumed to begin at 0.

Procedure: The program multiplies the row size (in bytes) times the row subscript (since the elements are stored by row), adds the product to the doubled column subscript (doubled because each element occupies two bytes), and adds the sum to the base address. The program uses a standard shift-and-add algorithm (see Subroutine 6A) to multiply.

Registers Used: AF, BC, DE, HL

Execution Time: Approximately 1100 cycles, depending mainly on how long it takes to multiply row size times row subscript

Program Size: 45 bytes

Data Memory Required: Four bytes anywhere in memory to hold the return address (two bytes starting at address RETADR) and the column subscript (two bytes starting at address SS2)

Entry Conditions

Order in stack (starting at the top)

Less significant byte of return address
More significant byte of return address

Less significant byte of column subscript
More significant byte of column subscript

Less significant byte of size of rows (in bytes)
More significant byte of size of rows (in bytes)

Less significant byte of row subscript
More significant byte of row subscript

Less significant byte of base address of array
More significant byte of base address of array

Exit Conditions

Starting address of element in HL
The element occupies the address in HL and the next higher address

Examples

1. Data: Base address = $5E14_{16}$
Column subscript = 0008_{16}
Size of a row (in bytes) = $001C_{16}$ (i.e., each row has 0014_{10} or $000E_{16}$ word-length elements)
Row subscript = 0005_{16}

Result: Element starting address = $5E14_{16} + 0005_{16} * 001C_{16} + 0008_{16} * 2 = 5E14_{16} + 008C_{16} + 0010_{16} = 5EB0_{16}$
That is, the starting address of ARRAY(5,8) is $5EB0_{16}$ and the element occupies $5EB0_{16}$ and $5EB1_{16}$

2. Data: Base address = $B100_{16}$
   Column subscript = 0002_{16}
   Size of a row (in bytes) = 0008_{16} (i.e., each row has four word-length elements)
   Row subscript = 0006_{16}

   Result: Element starting address = $B100_{16} + 0006_{16} * 0008_{16} + 0002_{16} * 2 = B100_{16} + 0030_{16} + 0004_{16} = B134_{16}$

   That is, the starting address of ARRAY(6,2) is $B134_{16}$ and the element occupies $B134_{16}$ and $B135_{16}$

The general formula is

ELEMENT STARTING ADDRESS = ARRAY BASE ADDRESS + ROW SUBSCRIPT * SIZE OF ROW + COLUMN SUBSCRIPT * 2

Note that one parameter of this routine is the size of a row in bytes. The size for word-length elements is the number of columns per row times 2 (the size of an element in bytes). The reason we chose this parameter rather than the number of columns or the maximum column index is that this parameter can be calculated once (when the array bounds are determined) and used whenever the array is accessed. The alternative parameters (number of columns or maximum column index) would require extra calculations during each indexing operation.

```
;                                                                    ;
;                                                                    ;
;                                                                    ;
;                                                                    ;
;       Title           Two-dimensional word array indexing          ;
;       Name:           D2WORD                                       ;
;                                                                    ;
;                                                                    ;

;                                                                    ;
;       Purpose:        Given the base address of a word array, two  ;
;                       subscripts 'I','J', and the size of the first ;
;                       subscript in bytes, calculate the address of ;
;                       A[I,J]. The array is assumed to be stored in ;
;                       row major order (A[0,0], A[0,1],..., A[K,L]), ;
;                       and both dimensions are assumed to begin at  ;
;                       zero as in the following Pascal declaration: ;
;                         A:ARRAY[0..2,0..7] OF WORD;                ;
;                                                                    ;
;       Entry:          TOP OF STACK                                 ;
;                         Low byte of return address,                ;
;                         High byte of return address,               ;
;                         Low byte of second subscript (column element) ;
;                         High byte of second subscript (column element);
;                         Low byte of first subscript size, in bytes, ;
;                         High byte of first subscript size, in bytes, ;
;                         Low byte of first subscript (row element), ;
;                         High byte of first subscript (row element), ;
;                         Low byte of array base address,            ;
;                         High byte of array base address,           ;
;                       NOTE:                                        ;
;                         The first subscript size is length of a row ;
;                         in words * 2                               ;
;                                                                       ;
;       Exit:           Register H = High byte of element address      ;
;                       Register L = High byte of element address      ;
;                                                                       ;
;       Registers used: AF,BC,DE,HL                                     ;
;                                                                       ;
;       Time:           Approximately 1100 cycles                       ;
;                                                                       ;
;       Size:           Program 45 bytes                                ;
;                       Data     4 bytes                                ;
;                                                                       ;
;                                                                       ;

D2WORD:
        ;SAVE RETURN ADDRESS
        POP     HL
        LD      (RETADR),HL

        ;GET SECOND SUBSCRIPT, MULTIPLY BY 2 FOR WORD-LENGTH ELEMENTS
        POP     HL
        ADD     HL,HL           ;* 2
        LD      (SS2),HL
        ;GET SIZE OF FIRST SUBSCRIPT (ROW LENGTH), FIRST SUBSCRIPT
        POP     DE              ;GET LENGTH OF ROW
        POP     BC              ;GET FIRST SUBSCRIPT

        ;MULTIPLY FIRST SUBSCRIPT * ROW LENGTH USING SHIFT AND ADD
        ; ALGORITHM. PRODUCT IS IN HL
        LD      HL,0            ;PRODUCT = 0
        LD      A,15            ;COUNT = BIT LENGTH - 1
MLP:
        SLA     E               ;SHIFT LOW BYTE OF MULTIPLIER
        RL      D               ;ROTATE HIGH BYTE OF MULTIPLIER
        JR      NC,MLP1         ;JUMP IF MSB OF MULTIPLIER = 0
        ADD     HL,BC           ;ADD MULTIPLICAND TO PARTIAL PRODUCT
MLP1:   ADD     HL,HL           ;SHIFT PARTIAL PRODUCT
        DEC     A
        JR      NZ,MLP          ;CONTINUE THROUGH 15 BITS

        ;ADD MULTIPLICAND IN LAST TIME IF MSB OF MULTIPLIER IS 1
        OR      D               ;SIGN FLAG = MSB OF MULTIPLIER
        JP      P,MLP2
        ADD     HL,BC           ;ADD IN MULTIPLICAND IF SIGN = 1
        ;ADD IN SECOND SUBSCRIPT
MLP2:   LD      DE,(SS2)
        ADD     HL,DE

        ;ADD BASE ADDRESS TO FORM FINAL ADDRESS
        POP     DE              ;GET BASE ADDRESS OF ARRAY
        ADD     HL,DE           ;ADD BASE TO INDEX

        ;RETURN TO CALLER
        LD      DE,(RETADR)     ;RESTORE RETURN ADDRESS TO STACK
        PUSH    DE
        RET
        ;DATA
RETADR: DS      2                   ;TEMPORARY FOR RETURN ADDRESS
SS2:    DS      2                   ;TEMPORARY FOR SECOND SUBSCRIPT


;                                                                         ;
;                                                                         ;
;       SAMPLE EXECUTION:                                                 ;
;                                                                         ;
;                                                                         ;

SC5D:
        LD      HL,ARY              ;PUSH BASE ADDRESS OF ARRAY
        PUSH    HL
        LD      HL,(SUBS1)          ;PUSH FIRST SUBSCRIPT
        PUSH    HL
        LD      HL,(SSUBS1)         ;PUSH SIZE OF FIRST SUBSCRIPT
        PUSH    HL
        LD      HL,(SUBS2)          ;PUSH SECOND SUBSRIPT
        PUSH    HL
        CALL    D2WORD              ;CALCULATE ADDRESS
                                    ;FOR THE INITIAL TEST DATA
                                    ;HL = ADDRESS OF ARY(2,4)
                                    ;   = ARY + (2*16) + 4 * 2
                                    ;   = ARY + 40 (CONTENTS ARE 2100H)
                                    ;NOTE BOTH SUBSCRIPTS START AT 0

        JR      SC5D

        ;DATA
SUBS1:  DW      2                   ;SUBSCRIPT 1
SSUBS1: DW      16                  ;SIZE OF SUBSCRIPT 1
SUBS2:  DW      4                   ;SUBSCRIPT 2

;THE ARRAY (3 ROWS OF 8 COLUMNS)
ARY:    DW      0100H,0200H,0300H,0400H,0500H,0600H,0700H,0800H
        DW      0900H,1000H,1100H,1200H,1300H,1400H,1500H,1600H
        DW      1700H,1800H,1900H,2000H,2100H,2200H,2300H,2400H

        END
```

N-Dimensional Array Indexing (NDIM)

Calculates the starting address of an element of an N-dimensional array given the base address and N pairs of sizes and subscripts. The size of a dimension is the number of bytes from the starting address of an element to the starting address of the element with an index one larger in the dimension but the same in all other dimensions. The array is assumed to be stored in row major order (that is, organized so that subscripts to the right change before subscripts to the left).

Note that the size of the rightmost subscript is simply the size of the elements (in bytes); the size of the next subscript is the size of the elements times the maximum value of the rightmost subscript plus 1, and so forth. All subscripts are assumed to begin at 0. Otherwise, the user must normalize the subscripts. (See the second example at the end of the listing.)

Procedure: The program loops on each dimension, calculating the offset in that dimension as the subscript times the size. If the size is an easy case (an integral power of 2), the program reduces the multiplication to left shifts. Otherwise, it performs each multiplication using the shift-and-add algorithm of Subroutine 6A. Once the program has calculated the overall offset, it adds that offset to the base address to obtain the starting address of the element.

Registers Used: AF, BC, DE, HL

Execution Time: Approximately 1300 cycles per dimension plus 165 cycles overhead (depending mainly on how much time is required to perform the multiplications)

Program Size: 120 bytes

Data Memory Required: Five bytes anywhere in memory to hold the return address (two bytes starting at address RETADR), the accumulated offset (two bytes starting at address OFFSET), and the number of dimensions (one byte at address NUMDIM)

Special Case: If the number of dimensions is 0, the program returns with the base address in HL.

Entry Conditions

Order in stack (starting from the top)

Less significant byte of return address
More significant byte of return address

Less significant byte of number of dimensions
More significant byte of number of dimensions (not used)

Less significant byte of size of rightmost dimension
More significant byte of size of rightmost dimension

Less significant byte of rightmost subscript
More significant byte of rightmost subscript

Exit Conditions

Starting address of element in HL

The element occupies memory addresses START through START + SIZE − 1, where START is the calculated address and SIZE is the size of an element in bytes.

.
.
.
Less significant byte of size of leftmost dimension
More significant byte of size of leftmost dimension
Less significant byte of leftmost subscript
More significant byte of leftmost subscript

Less significant byte of base address of array
More significant byte of base address of array

Example

1. Data: Base address = $3C00_{16}$
Number of dimensions = 0003_{16}
Rightmost subscript = 0005_{16}
Rightmost size = 0003_{16} (3-byte entries)
Middle subscript = 0003_{16}
Middle size = 0012_{16} (six 3-byte entries)
Leftmost subscript = 0004_{16}
Leftmost size = $007E_{16}$ (seven sets of six 3-byte entries)

Result: Element starting address = $3C00_{16} + 0005_{16} * 0003_{16} + 0003_{16} * 0012_{16} + 0004_{16} * 007E_{16} = 3C00_{16} + 000F_{16} + 0036_{16} + 01F8_{16} = 3E3D_{16}$

That is, the element is ARRAY(4,3,5); it occupies addresses $3E3D_{16}$ through $3E3F_{16}$ (the maximum values of the various subscripts are 6 (leftmost) and 5 (middle) with each element occupying three bytes)

The general formula is

$$\text{STARTING ADDRESS} = \text{BASE ADDRESS} + \sum_{i=0}^{N-1} \text{SUBSCRIPT}_i * \text{SIZE}_i$$

where

N is the number of dimensions
SUBSCRIPT_i is the *i*th subscript
SIZE_i is the size of the *i*th dimension

Note that we use the size of each dimension as a parameter to reduce the number of repetitive multiplications and to generalize the procedure. The sizes can be calculated and saved as soon as the bounds of the array are known. Those sizes can then be used whenever indexing is performed on that array. Obviously, the sizes do not change if the bounds are fixed, and they should not be recalculated as part of each indexing operation. The sizes are also general, since the elements can themselves consist of any number of bytes.

```
;                                                                       ;
;                                                                       ;
;                                                                       ;
;                                                                       ;
;       Title           N-dimensional array indexing                    ;
;       Name:           NDIM                                            ;
;                                                                       ;
;                                                                       ;

;                                                                       ;
;       Purpose:        Calculate the address of an element in an       ;
;                       N-dimensional array given the base address,     ;
;                       N pairs of size in bytes and subscript, and the ;
;                       number of dimensions of the array. The array is ;
;                       assumed to be stored in row major order         ;
;                       (A[0,0,0],A[0,0,1],...,A[0,1,0],A[0,1,1],...).  ;
;                       Also, it is assumed that all dimensions begin   ;
;                       at 0 as in the following Pascal declaration:    ;
;                         A:ARRAY[0..10,0..3,0..5] OF SOMETHING         ;
;                       For arrays that do not begin at 0 boundaries,   ;
;                       normalization must be performed before calling  ;
;                       this routine. An example is given at the end.   ;
;                                                                       ;
;       Entry:          TOP OF STACK                                    ;
;                         Low byte of return address,                   ;
;                         High byte of return address,                  ;
;                         Low byte of number dimensions,                ;
;                         High byte of number dimensions (not used),    ;
;                         Low byte of dim N-1 size                      ;
;                         High byte of dim N-1 size                     ;
;                         Low byte of dim N-1 subscript                 ;
;                         High byte of dim N-1 subscript                ;
;                         Low byte of dim N-2 size                      ;
;                         High byte of dim N-2 size                     ;
;                         Low byte of dim N-2 subscript                 ;
;                         High byte of dim N-2 subscript                ;
;                              .                                        ;
;                              .                                        ;
;                              .                                        ;
;                         Low byte of dim 0 size                        ;
;                         High byte of dim 0 size                       ;
;                         Low byte of dim 0 subscript                   ;
;                         High byte of dim 0 subscript                  ;
;                         Low byte of array base address                ;
;                         High byte of array base address               ;
;                       NOTE:                                           ;
;                         All sizes are in bytes                        ;
;                                                                       ;
;       Exit:           Register H = High byte of address               ;
;                       Register L = Low byte of address                ;
;                                                                       ;
;       Registers used: AF,BC,DE,HL                                     ;
;                                                                       ;
;       Time:           Approximately 1300 cycles per dimension         ;
;                       plus 165 cycles overhead                        ;
;                                                                       ;
;       Size:           Program 120 bytes                               ;
;                       Data      5 bytes                               ;
;                                                                       ;
;                                                                       ;

NDIM:
        ;POP PARAMETERS
        POP     HL
        LD      (RETADR),HL

        ;OFFSET := 0
        LD      HL,0
        LD      (OFFSET),HL

        ;GET NUMBER OF DIMENSIONS AND TEST FOR 0
        POP     HL
        LD      A,L
        LD      (NUMDIM),A              ;GET NUMBER OF DIMENSIONS
        OR      A                       ;TEST FOR 0
        JR      Z,ADBASE                ;RETURN WITH BASE ADDRESS IN HL
                                        ; IF THERE ARE NO DIMENSIONS

        ;LOOP ON EACH DIMENSION
        ; DOING OFFSET := OFFSET + (SUBSCRIPT * SIZE)
LOOP:
        POP     DE                      ;GET SIZE
        POP     HL                      ;GET SUBSCRIPT
        CALL    NXTOFF                  ;OFFSET := OFFSET + (SUBSCRIPT * SIZE)
        LD      HL,NUMDIM
        DEC     (HL)                    ;DECREMENT NUMBER OF DIMENSIONS
        JR      NZ,LOOP                 ;CONTINUE THROUGH ALL DIMENSIONS

ADBASE:
        ;CALCULATE STARTING ADDRESS OF ELEMENT
        ;OFFSET = BASE + OFFSET
        LD      HL,(OFFSET)
        POP     DE                      ;GET BASE ADDRESS
        ADD     HL,DE                   ;SUM WITH OFFSET

        ;RESTORE RETURN ADDRESS AND EXIT
        LD      DE,(RETADR)
        PUSH    DE
        RET

        ;------------------------------
        ;SUBROUTINE NXTOFF
        ;PURPOSE: OFFSET := OFFSET + (SUBSCRIPT * SIZE);
        ;ENTRY: OFFSET = CURRENT OFFSET
        ;       DE = CURRENT SIZE OF THIS DIMEMSION
        ;       HL = CURRENT SUBSCRIPT
        ;EXIT:  OFFSET = OFFSET + (SUBSCRIPT + SIZE);
        ;REGISTERS USED: AF, BC, DE, HL
        ;------------------------------
```

```
NXTOFF:
        PUSH    HL              ;SAVE CURRENT SUBSCRIPT IN STACK

        ;CHECK IF SIZE IS POWER OF 2 LESS THAN 256
        LD      A,D
        OR      A               ;HIGH BYTE = 0 ?
        JR      NZ,BIGSZ        ;JUMP IF SIZE IS LARGE

        LD      A,E             ;A = LOW BYTE OF SIZE
        LD      HL,EASYAY       ;HL = BASE ADDRESS OF EASYAY
        LD      B,SZEASY        ;B = SIZE OF EASY ARRAY
        LD      C,0             ;C = SHIFT COUNTER
EASYLP:
        CP      (HL)
        JR      Z,ISEASY        ;JUMP IF SIZE IS A POWER OF 2
        INC     HL              ;INCREMENT TO NEXT BYTE OF EASYAY
        INC     C               ;INCREMENT SHIFT COUNTER
        DJNZ    EASYLP          ;DECREMENT COUNT
        JR      BIGSZ           ;JUMP IF SIZE IS NOT EASY

ISEASY:
        POP     HL              ;GET SUBSCRIPT
        LD      A,C             ;GET NUMBER OF SHIFTS
        OR      A               ;TEST FOR 0
        JR      Z,ADDOFF        ;JUMP IF SHIFT FACTOR = 0

        ;ELEMENT SIZE * SUBSCRIPT REDUCES TO LEFT SHIFTS
        LD      B,A             ;B = SHIFT COUNT
SHIFT:
        ADD     HL,HL           ;MULTIPLY SUBSCRIPT BY 2
        DJNZ    SHIFT           ;CONTINUE UNTIL DONE
        JR      ADDOFF          ;DONE SO ADD OFFSET + SUBSCRIPT

BIGSZ:
        ;SIZE IS NOT POWER OF 2, MULTIPLY
        ; ELEMENT SIZE TIMES SUBSCRIPT THE HARD WAY
        POP     BC              ;GET SUBSCRIPT

        ;MULTIPLY FIRST SUBSCRIPT * ROW LENGTH USING SHIFT AND ADD
        ; ALGORITHM. RESULT IS IN HL
        ; BC = SUBSCRIPT (MULTIPLICAND)
        ; DE = SIZE (MULTIPLIER)
        LD      HL,0            ;PRODUCT = 0
        LD      A,15            ;COUNT = BIT LENGTH - 1
MLP:
        SLA     E               ;SHIFT LOW BYTE OF MULTIPLIER
        RL      D               ;ROTATE HIGH BYTE OF MULTIPLIER
        JR      NC,MLP1         ;JUMP IF MSB OF MULTIPLIER = 0
        ADD     HL,BC           ;ADD MULTIPLICAND TO PARTIAL PRODUCT
MLP1:   ADD     HL,HL           ;SHIFT PARTIAL PRODUCT
        DEC     A
        JR      NZ,MLP          ;CONTINUE THROUGH 15 BITS
        ;ADD IN MULTIPLICAND LAST TIME IF MSB OF MULTIPLIER IS 1
        OR      D               ;SIGN FLAG = MSB OF MULTIPLIER
        JP      P,ADDOFF
        ADD     HL,BC           ;ADD IN MULTIPLICAND IF SIGN = 1

        ;ADD SUBSCRIPT * SIZE TO OFFSET
ADDOFF:
        EX      DE,HL
        LD      HL,(OFFSET)     ;GET OFFSET
        ADD     HL,DE           ;ADD PRODUCT OF SUBSCRIPT * SIZE
        LD      (OFFSET),HL     ;SAVE OFFSET
        RET

EASYAY:                         ;SHIFT FACTOR
        DB      1               ;0
        DB      2               ;1
        DB      4               ;2
        DB      8               ;3
        DB      16              ;4
        DB      32              ;5
        DB      64              ;6
        DB      128             ;7
SZEASY  EQU     $-EASYAY

        ;DATA
RETADR: DS      2               ;TEMPORARY FOR RETURN ADDRESS
OFFSET: DS      2               ;TEMPORARY FOR PARTIAL OFFSET
NUMDIM: DS      1               ;NUMBER OF DIMENSIONS


;                                                                       ;
;                                                                       ;
;       SAMPLE EXECUTION:                                               ;
;                                                                       ;
;                                                                       ;

SC5E:
        ;FIND ADDRESS OF AY1[1,3,0]
        ;  SINCE LOWER BOUNDS OF ARRAY 1 ARE ALL ZERO IT IS NOT
        ;  NECESSARY TO NORMALIZE THEM

        ;PUSH BASE ADDRESS OF ARRAY 1
        LD      HL,AY1
        PUSH    HL

        ;PUSH SUBSCRIPT/SIZE FOR DIMENSION 1
        LD      HL,1
        PUSH    HL              ;SUBSCRIPT
        LD      HL,A1SZ1
        PUSH    HL              ;SIZE

        ;PUSH SUBSCRIPT/SIZE FOR DIMENSION 2
        LD      HL,3
        PUSH    HL              ;SUBSCRIPT
        LD      HL,A1SZ2
        PUSH    HL              ;SIZE

        ;PUSH SUBSCRIPT/SIZE FOR DIMENSION 3
        LD      HL,0
        PUSH    HL              ;SUBSCRIPT
        LD      HL,A1SZ3
        PUSH    HL              ;SIZE

        ;PUSH NUMBER OF DIMENSIONS
        LD      HL,A1DIM
        PUSH    HL
        CALL    NDIM            ;CALCULATE ADDRESS
                                ;AY = STARTING ADDRESS OF ARY1(1,3,0)
                                ;    = ARY + (1*126) + (3*21) + (0*3)
                                ;    = ARY + 189

        ;CALCULATE ADDRESS OF AY2[-1,6]
        ;  SINCE LOWER BOUNDS OF AY2 DO NOT START AT 0, SUBSCRIPTS
        ;  MUST BE NORMALIZED

        ;PUSH BASE ADDRESS OF ARRAY 2
        LD      HL,AY2
        PUSH    HL

        ;PUSH (SUBSCRIPT - LOWER BOUND)/SIZE FOR DIMENSION 1
        LD      HL,-1
        LD      DE,-A2D1L       ;NEGATIVE OF LOWER BOUND
        ADD     HL,DE           ;ADD NEGATIVE TO NORMALIZE TO 0
        PUSH    HL              ;SUBSCRIPT
        LD      HL,A2SZ1
        PUSH    HL              ;SIZE

        ;PUSH (SUBSCRIPT - LOWER BOUND)/SIZE FOR DIMENSION 2
        LD      HL,6
        LD      DE,-A2D2L       ;NEGATIVE OF LOWER BOUND
        ADD     HL,DE           ;ADD NEGATIVE TO NORMALIZE TO 0
        PUSH    HL              ;SUBSCRIPT
        LD      HL,A2SZ2
        PUSH    HL              ;SIZE

        ;PUSH NUMBER OF DIMENSIONS
        LD      HL,A2DIM
        PUSH    HL

        CALL    NDIM            ;CALCULATE ADDRESS
                                ;AY=STARTING ADDRESS OF ARY1(-1,6)
                                ;  =ARY+(((-1)-(-5))*18)+((6-2)*2)
                                ;  =ARY + 80

        JR      SC5E

;DATA
;AY1 : ARRAY[A1D1L..A1D1H,A1D2L..A1D2H,A1D3L..A1D3H] 3-BYTE ELEMENTS
;           [ 0  ..  3  ,  0  ..  5  ,  0  ..  6  ]
A1DIM   EQU     3                       ;NUMBER OF DIMENSIONS
A1D1L   EQU     0                       ;LOW BOUND OF DIMENSION 1
A1D1H   EQU     3                       ;HIGH BOUND OF DIMENSION 1
A1D2L   EQU     0                       ;LOW BOUND OF DIMENSION 2
A1D2H   EQU     5                       ;HIGH BOUND OF DIMENSION 2
```

```
A1D3L    EQU     0                        ;LOW BOUND OF DIMENSION 3
A1D3H    EQU     6                        ;HIGH BOUND OF DIMENSION 3
A1SZ3    EQU     3                        ;SIZE OF ELEMENT IN DIMENSION 3
A1SZ2    EQU     ((A1D3H-A1D3L)+1)*A1SZ3 ;SIZE OF ELEMENT IN DIMENSION 2
A1SZ1    EQU     ((A1D2H-A1D2L)+1)*A1SZ2 ;SIZE OF ELEMENT IN DIMENSION 1
AY1:     DS      ((A1D1H-A1D1L)+1)*A1SZ1 ;ARRAY

;AY2 : ARRAY[A1D1L..A1D1H,A1D2L..A1D2H] OF WORD
;            [  -5 ..  -1  ,  2  .. 10 ]
A2DIM    EQU     2                        ;NUMBER OF DIMENSIONS
A2D1L    EQU     -5                       ;LOW BOUND OF DIMENSION 1
A2D1H    EQU     -1                       ;HIGH BOUND OF DIMENSION 1
A2D2L    EQU     2                        ;LOW BOUND OF DIMENSION 2
A2D2H    EQU     10                       ;HIGH BOUND OF DIMENSION 2
A2SZ2    EQU     2                        ;SIZE OF ELEMENT IN DIMENSION 2
A2SZ1    EQU     ((A2D2H-A2D2L)+1)*A2SZ2 ;SIZE OF ELEMENT IN DIMENSION 1
AY2:     DS      ((A2D1H-A2D1L)+1)*A2SZ1 ;ARRAY

         END
```

16-Bit Multiplication (MUL16) 6A

Multiplies two 16-bit operands and returns the less significant (16-bit) word of the product.

Procedure: The program uses an ordinary shift-and-add algorithm, adding the multiplicand to the partial product each time it finds a 1 bit in the multiplier. The partial product and the multiplier are shifted left 15 times (the number of bits in the multiplier minus 1) to produce proper alignment. The more significant 16 bits of the product are lost.

Registers Used: AF, BC, DE, HL

Execution Time: Approximately 865 to 965 cycles, depending largely on the number of 1 bits in the multiplier

Program Size: 22 bytes

Data Memory Required: None

Entry Conditions

Multiplicand in HL
Multiplier in DE

Exit Conditions

Less significant word of product in HL

Examples

1. Data: Multiplier = 0012_{16}
   Multiplicand = $03D1_{16}$

   Result: Product = $44B2_{16}$
   The more significant word is 0.

2. Data: Multiplier = $37D1_{16}$
   Multiplicand = $A045_{16}$

   Result: Product = $AB55_{16}$
   This is actually the less significant 16-bit word of the 32-bit product $22F1AB55_{16}$.

Note that MUL16 returns only the less significant word of the product to maintain compatibility with other 16-bit arithmetic operations. The more significant word of the product is lost.

```
;                                                                    ;
;                                                                    ;
;                                                                    ;
;                                                                    ;
;       Title           16-bit Multiplication                        ;
;       Name:           MUL16                                        ;
;                                                                    ;
;                                                                    ;

;                                                                    ;
;       Purpose:        Multiply 2 signed or unsigned 16-bit words and ;
;                       return a 16-bit signed or unsigned product   ;
;                                                                        ;
;                       Answers needing more than 16 bits: bits higher  ;
;                       than bit 15 are lost                            ;
;                                                                        ;
;       Entry:          Register L = Low byte of multiplicand            ;
;                       Register H = High byte of multiplicand           ;
;                       Register E = Low byte of multiplier              ;
;                       Register D = High byte of multiplier             ;
;                                                                        ;
;       Exit:           Product = multiplicand * multiplier              ;
;                       Register L = Low byte of product                 ;
;                       Register H = High byte of product                ;
;                                                                        ;
;       Registers used: AF,BC,DE,HL                                      ;
;                                                                        ;
;       Time:           Approximately 865 to 965 cycles                  ;
;                                                                        ;
;       Size:           Program  22 bytes                                ;
;                                                                        ;
;                                                                        ;

        ;INITIALIZE PARTIAL PRODUCT, BIT COUNT
MUL16:
        LD      C,L             ;BC = MULTIPLIER
        LD      B,H
        LD      HL,0            ;PRODUCT = 0
        LD      A,15            ;COUNT = BIT LENGTH - 1

        ;SHIFT-AND-ADD ALGORITHM
        ; IF MSB OF MULTIPLIER IS 1, ADD MULTIPLICAND TO PARTIAL
        ;   PRODUCT
        ; SHIFT PARTIAL PRODUCT, MULTIPLIER LEFT 1 BIT
MLP:
        SLA     E               ;SHIFT MULTIPLIER LEFT 1 BIT
        RL      D
        JR      NC,MLP1         ;JUMP IF MSB OF MULTIPLIER = 0
        ADD     HL,BC           ;ADD MULTIPLICAND TO PARTIAL PRODUCT
MLP1:
        ADD     HL,HL           ;SHIFT PARTIAL PRODUCT LEFT
        DEC     A
        JR      NZ,MLP          ;CONTINUE UNTIL COUNT = 0

        ;ADD MULTIPLICAND ONE LAST TIME IF MSB OF MULTIPLIER IS 1

        OR      D               ;SIGN FLAG = MSB OF MULTIPLIER
        RET     P               ;EXIT IF MSB OF MULTIPLIER IS 0
        ADD     HL,BC           ;ADD MULTIPLICAND TO PRODUCT
        RET


;                                                                        ;
;                                                                        ;
;       SAMPLE EXECUTION:                                                ;
;                                                                        ;
;                                                                        ;
```

```
SC6A:
        LD      HL,-2           ;HL = MULTIPLICAND
        LD      DE,1023         ;DE = MULTIPLIER
        CALL    MUL16           ;16-BIT MULTIPLY
                                ;RESULT OF 1023 * -2 = -2046 = 0F802H
                                ; REGISTER L = 02H
                                ;          H = F8H

        JR      SC6A

        END
```

16-Bit Division (SDIV16, UDIV16) 6B

Divides two 16-bit operands and returns the quotient and the remainder. There are two entry points: SDIV16 divides two 16-bit signed operands, whereas UDIV16 divides two 16-bit unsigned operands. If the divisor is 0, the Carry flag is set to 1 and both quotient and remainder are set to 0; otherwise, the Carry flag is cleared.

Procedure: If the operands are signed, the program determines the sign of the quotient and takes the absolute values of any negative operands. It must also retain the sign of the dividend, since that determines the sign of the remainder. The program then performs an unsigned division using a shift-and-subtract algorithm. It shifts the quotient and dividend left, placing a 1 bit in the quotient each time a trial subtraction is successful. If the operands are signed, the program must negate (that is, subtract from 0) the quotient or remainder if either is negative. The Carry flag is cleared if the division is proper and set if the divisor is 0. A 0 divisor also causes a return with the quotient and remainder both set to 0.

Registers Used: AF, BC, DE, HL

Execution Time: Approximately 1770 to 2340 cycles, depending largely on how many trial subtractions are successful and thus require the replacement of the previous dividend by the remainder

Program Size: 104 bytes

Data Memory Required: 3 bytes anywhere in RAM for the sign of the quotient (address SQUOT), the sign of the remainder (address SREM), and a divide loop counter (address COUNT)

Special Case: If the divisor is 0, the program returns with the Carry set to 1, and both the quotient and the remainder set to 0.

Entry Conditions

Dividend in HL
Divisor in DE

Exit Conditions

Quotient in HL
Remainder in DE
If the divisor is non-zero, Carry = 0 and the result is normal.
If the divisor is 0, Carry = 1 and both quotient and remainder are 0000.

Examples

1. Data: Dividend = $03E0_{16}$
   Divisor = $00B6_{16}$

   Result: Quotient (from UDIV16) = 0005_{16}
   Remainder (from UDIV16) = 0052_{16}
   Carry = 0 (no divide-by-0 error)

2. Data: Dividend = $D73A_{16}$
   Divisor = $02F1_{16}$

   Result: Quotient (from SDIV16) = $FFF3_{16}$
   Remainder (from SDIV16) = $FD77_{16}$
   Carry = 0 (no divide-by-0 error)

The remainder of a signed division may be either positive or negative. In this procedure, the remainder always takes the sign of the dividend. A negative remainder can easily be converted into one that is always positive. Simply subtract 1 from the quotient and add the divisor to the remainder. The result of Example 2 is then

Quotient = $FFF2_{16} = -14_{10}$
Remainder (always positive) = 0068_{16}

```
;                                                                     ;
;                                                                     ;
;                                                                     ;
;                                                                     ;
;       Title           16-bit Division                               ;
;       Name:           SDIV16, UDIV16                                ;
;                                                                     ;
;                                                                     ;


;                                                                     ;
;       Purpose:        SDIV16                                        ;
;                         Divide 2 signed 16-bit words and return a   ;
;                         16-bit signed quotient and remainder        ;
;                                                                     ;
;                       UDIV16                                        ;
;                         Divide 2 unsigned 16-bit words and return a ;
;                         16-bit unsigned quotient and remainder      ;
;                                                                     ;
;       Entry:          Register L = Low byte of dividend             ;
;                       Register H = High byte of dividend            ;
;                       Register E = Low byte of divisor              ;
;                       Register D = High byte of divisor             ;
;                                                                     ;
;       Exit:           Register L = Low byte of quotient             ;
;                       Register H = High byte of quotient            ;
;                       Register E = Low byte of remainder            ;
;                       Register D = High byte of remainder           ;
;                                                                     ;
;                       If no errors then                             ;
;                         carry := 0                                  ;
;                       else                                          ;
;                         divide-by-zero error                        ;
;                         carry := 1                                  ;
;                         quotient := 0                               ;
;                         remainder := 0                              ;
;                                                                     ;
;       Registers used: AF,BC,DE,HL                                   ;
;                                                                     ;
;       Time:           Approximately 1770 to 2340 cycles             ;
;                                                                     ;
;       Size:           Program 108 bytes                             ;
;                       Data      3 bytes                             ;
;                                                                     ;
;                                                                     ;
```

```
        ;SIGNED DIVISION
SDIV16:
        ;DETERMINE SIGN OF QUOTIENT BY EXCLUSIVE ORING HIGH BYTES
        ; OF DIVIDEND AND DIVISOR.  QUOTIENT IS POSITIVE IF SIGNS
        ; ARE THE SAME, NEGATIVE IF SIGNS ARE DIFFERENT
        ;
        ;REMAINDER HAS SAME SIGN AS DIVIDEND
        LD      A,H             ;GET HIGH BYTE OF DIVIDEND
        LD      (SREM),A        ;SAVE AS SIGN OF REMAINDER
        XOR     D               ;EXCLUSIVE OR WITH HIGH BYTE OF DIVISOR
        LD      (SQUOT),A       ;SAVE SIGN OF QUOTIENT

        ;TAKE ABSOLUTE VALUE OF DIVISOR

        LD      A,D
        OR      A
        JP      P,CHKDE         ;JUMP IF DIVISOR IS POSITIVE
        SUB     A               ;SUBTRACT DIVISOR FROM ZERO
        SUB     E
        LD      E,A
        SBC     A,A             ;PROPAGATE BORROW (A=FF IF BORROW)
        SUB     D
        LD      D,A

        ;TAKE ABSOLUTE VALUE OF DIVIDEND
CHKDE:
        LD      A,H
        OR      A
        JP      P,DODIV         ;JUMP IF DIVIDEND IS POSITIVE
        SUB     A               ;SUBTRACT DIVIDEND FROM ZERO
        SUB     L
        LD      L,A
        SBC     A,A             ;PROPAGATE BORROW (A=FF IF BORROW)
        SUB     H
        LD      H,A

        ;DIVIDE ABSOLUTE VALUES
DODIV:
        CALL    UDIV16
        RET     C               ;EXIT IF DIVIDE BY ZERO

        ;NEGATE QUOTIENT IF IT IS NEGATIVE
        LD      A,(SQUOT)
        OR      A
        JP      P,DOREM         ;JUMP IF QUOTIENT IS POSITIVE
        SUB     A               ;SUBTRACT QUOTIENT FROM ZERO
        SUB     L
        LD      L,A
        SBC     A,A             ;PROPAGATE BORROW (A=FF IF BORROW)
        SUB     H
        LD      H,A

DOREM:
        ;NEGATE REMAINDER IF IT IS NEGATIVE
        LD      A,(SREM)
        OR      A
        RET     P                   ;RETURN IF REMAINDER IS POSITIVE
        SUB     A                   ;SUBTRACT REMAINDER FROM ZERO
        SUB     E
        LD      E,A
        SBC     A,A                 ;PROPAGATE BORROW (A=FF IF BORROW)
        SUB     D
        LD      D,A
        RET


        ;UNSIGNED DIVISION
UDIV16:
        ;CHECK FOR DIVISION BY ZERO
        LD      A,E
        OR      D
        JR      NZ,DIVIDE           ;BRANCH IF DIVISOR IS NON-ZERO
        LD      HL,0                ;DIVIDE BY 0 ERROR
        LD      D,H
        LD      E,L
        SCF                         ;SET CARRY, INVALID RESULT
        RET


DIVIDE:
        LD      C,L                 ;C = LOW BYTE OF DIVIDEND/QUOTIENT
        LD      A,H                 ;A = HIGH BYTE OF DIVIDEND/QUOTIENT
        LD      HL,0                ;HL = REMAINDER
        LD      B,16                ;16 BITS IN DIVIDEND
        OR      A                   ;CLEAR CARRY TO START


DVLOOP:
        ;SHIFT NEXT BIT OF QUOTIENT INTO BIT 0 OF DIVIDEND
        ;SHIFT NEXT MOST SIGNIFICANT BIT OF DIVIDEND INTO
        ; LEAST SIGNIFICANT BIT OF REMAINDER
        ;BC HOLDS BOTH DIVIDEND AND QUOTIENT. WHILE WE SHIFT A
        ; BIT FROM MSB OF DIVIDEND, WE SHIFT NEXT BIT OF QUOTIENT
        ; IN FROM CARRY
        ;HL HOLDS REMAINDER
        ;
        ;DO A 32-BIT LEFT SHIFT, SHIFTING
        ; CARRY TO C, C TO A, A TO L, L TO H
        RL      C                   ;CARRY (NEXT BIT OF QUOTIENT) TO BIT 0,
        RLA                         ; SHIFT REMAINING BYTES
        RL      L
        RL      H                   ;CLEARS CARRY SINCE HL WAS 0

        ;IF REMAINDER IS GREATER THAN OR EQUAL TO DIVISOR, NEXT
        ; BIT OF QUOTIENT IS 1. THIS BIT GOES TO CARRY
        PUSH    HL                  ;SAVE CURRENT REMAINDER
        SBC     HL,DE               ;SUBTRACT DIVISOR FROM REMAINDER
        CCF                         ;COMPLEMENT BORROW SO 1 INDICATES
                                    ; A SUCCESSFUL SUBTRACTION
                                    ; (THIS IS NEXT BIT OF QUOTIENT)
        JR      C,DROP              ;JUMP IF REMAINDER IS >= DIVIDEND
        EX      (SP),HL             ;OTHERWISE RESTORE REMAINDER
```

```
DROP:
        INC     SP              ;DROP REMAINDER FROM TOP OF STACK
        INC     SP
        DJNZ    DVLOOP          ;CONTINUE UNTIL ALL BITS DONE

        ;SHIFT LAST CARRY BIT INTO QUOTIENT
        EX      DE,HL           ;DE = REMAINDER
        RL      C               ;CARRY TO C
        LD      L,C             ;L = LOW BYTE OF QUOTIENT
        RLA
        LD      H,A             ;H = HIGH BYTE OF QUOTIENT
        OR      A               ;CLEAR CARRY, VALID RESULT
        RET

        ;DATA
SQUOT:  DS      1               ;SIGN OF QUOTIENT
SREM:   DS      1               ;SIGN OF REMAINDER
COUNT:  DS      1               ;DIVIDE LOOP COUNTER

;                                                               ;
;                                                               ;
;       SAMPLE EXECUTION:                                       ;
;                                                               ;
;                                                               ;

SC6B:
                                ;SIGNED DIVISION
        LD      HL,-1023        ; HL = DIVIDEND
        LD      DE,123          ; DE = DIVISOR
        CALL    SDIV16          ;QUOTIENT OF -1023 / 123 = -8
                                ; L = F8H
                                ; H = FFH
                                ;REMAINDER OF -1023 / 123 = -39
                                ; E = D9H
                                ; D = FFH

                                ;UNSIGNED DIVISION
        LD      HL,64513        ; HL = DIVIDEND
        LD      DE,123          ; DE = DIVISOR
        CALL    UDIV16          ;QUOTIENT OF 64513 / 123 = 524
                                ; L = 0CH
                                ; H = 02H
                                ;REMAINDER OF 64513 / 123 = 61
                                ; E = 3DH
                                ; D = 00H

        JR      SC6B

        END
```

16-Bit Comparison (CMP16) 6C

Compares two 16-bit operands and sets the flags accordingly. The Zero flag always indicates whether the numbers are equal. If the operands are unsigned, the Carry flag indicates which is larger (Carry = 1 if subtrahend is larger and 0 otherwise). If the operands are signed, the Sign flag indicates which is larger (Sign= 1 if subtrahend is larger and 0 otherwise); two's complement overflow is considered and the Sign flag is inverted if it occurs.

Procedure: The program subtracts the subtrahend from the minuend. If two's complement overflow occurs (Parity/Overflow flag= 1), the program inverts the Sign flag by EXCLUSIVE ORing the sign bit with 1. This requires an extra right shift to retain the Carry in bit 7 initially, since XOR always clears Carry. The program then sets Carry to ensure a non-zero result and shifts the data back to the left. The extra left shift uses ADC A,A rather than RLA to set the Sign and Zero flags (RLA would affect only Carry). Bit 0 of the accumulator must be 1 after the shift (because the Carry was set), thus ensuring that the Zero flag is cleared. Obviously, the result cannot be 0 if the subtraction causes two's complement overflow. Note that after an addition or subtraction, PE (Parity/Overflow flag = 1) means "overflow set" while PO (Parity/Overflow flag = 0) means "overflow clear."

Registers Used: AF, HL
Execution Time: 30 cycles if no overflow, 57 cycles if overflow
Program Size: 11 bytes
Data Memory Required: None

Entry Conditions

Minuend in HL
Subtrahend in DE

Exit Conditions

Flags set as if subtrahend had been subtracted from minuend, with a correction if two's complement overflow occurred.

Zero flag= 1 if the subtrahend and minuend are equal; 0 if they are not equal.

Carry flag = 1 if subtrahend is larger than minuend in the unsigned sense; 0 if it is less than or equal to the minuend.

Sign flag = 1 if subtrahend is larger than minuend in the signed sense; 0 if it is less than or equal to the minuend. This flag is corrected (inverted) if two's complement overflow occurs.

Examples

1. Data: Minuend (HL) = $03E1_{16}$
   Subtrahend (DE) = $07E4_{16}$

   Result: Carry = 1, indicating subtrahend is larger in unsigned sense.
   Zero = 0, indicating operands are not equal.
   Sign = 1, indicating subtrahend is larger in signed sense.

2. Data: Minuend (HL) = $C51A_{16}$
   Subtrahend (DE) = $C51A_{16}$

   Result: Carry = 0, indicating subtrahend is not larger in unsigned sense.
   Zero = 1, indicating operands are equal.
   Sign = 0, indicating subtrahend is not larger in signed sense.

3. Data: Minuend (HL) = $A45D_{16}$
   Subtrahend (DE) = $77E1_{16}$

   Result: Carry = 0, indicating subtrahend is not larger in unsigned sense.
   Zero = 0, indicating operands are not equal.
   Sign = 1, indicating subtrahend is larger in signed sense.

In Example 3, the minuend is a negative two's complement number, whereas the subtrahend is a positive two's complement number. Subtracting produces a positive result ($3C7C_{16}$) with two's complement overflow.

```
;                                                                        ;
;                                                                        ;
;                                                                        ;
;                                                                        ;
;       Title           16-bit Compare                                   ;
;       Name:           CMP16                                            ;
;                                                                        ;
;                                                                        ;


;                                                                        ;
;       Purpose:        Compare 2 16-bit signed or unsigned words and    ;
;                       return the C,Z,S flags set or cleared            ;
;                                                                        ;
;       Entry:          Register L = Low byte of minuend                 ;
;                       Register H = High byte of minuend                ;
;                       Register E = Low byte of subtrahend              ;
;                       Register D = High byte of subtrahend             ;
;                                                                        ;
;       Exit:           Flags returned based on minuend - subtrahend     ;
;                       If both the minuend and subtrahend are 2's       ;
;                         complement numbers, then use the Z and S       ;
;                         flags;                                         ;
;                       Else use the Z and C flags                       ;
;                       IF minuend = subtrahend THEN                     ;
;                          Z=1,S=0,C=0                                   ;
;                        IF minuend > subtrahend THEN                    ;
;                          Z=0,S=0,C=0                                   ;
;                        IF minuend < subtrahend THEN                    ;
;                          Z=0,S=1,C=1                                   ;
;                                                                        ;
;       Registers used: AF,HL                                            ;
;                                                                        ;
;       Time:           30 cycles if no overflow, else 57 cycles         ;
;                                                                          ;
;       Size:           Program 11 bytes                                   ;
;                                                                          ;
;                                                                          ;

CMP16:
        OR      A               ;CLEAR CARRY
        SBC     HL,DE           ;SUBTRACT SUBTRAHEND FROM MINUEND
        RET     PO              ;RETURN IF NO OVERFLOW
        LD      A,H             ;OVERFLOW - INVERT SIGN FLAG
        RRA                     ;SAVE CARRY IN BIT 7
        XOR     01000000B       ;COMPLEMENT BIT 6 (SIGN BIT)
        SCF                     ;ENSURE A NON-ZERO RESULT
        ADC     A,A             ;RESTORE CARRY, COMPLEMENTED SIGN
                                ; ZERO FLAG = 0 FOR SURE
        RET


;                                                                          ;
;                                                                          ;
;       SAMPLE EXECUTION:                                                  ;
;                                                                          ;
;                                                                          ;

SC6C:
        ;COMPARE -32768 (8000 HEX) AND 1
        ;SINCE -32768 IS THE MOST NEGATIVE 16-BIT NUMBER,
        ;  THIS COMPARISON WILL SURELY CAUSE OVERFLOW
        LD      HL,-32768
        LD      DE,1
        CALL    CMP16           ;CY = 0, Z = 0, S = 1

        ;COMPARE -4 (FFFC HEX) AND -1 (FFFF HEX)
        LD      HL,-4
        LD      DE,-1
        CALL    CMP16           ;CY = 1, Z = 0, S = 1

        ;COMPARE -1234 AND -1234
        LD      HL,-1234
        LD      DE,-1234
        CALL    CMP16           ;CY = 0, Z = 1, S = 0

        JR      SC6C

        END
```

Multiple-Precision Binary Addition

(MPBADD)

Adds two multi-byte unsigned binary numbers. Both numbers are stored with their least significant bytes at the lowest address. The sum replaces the addend. The length of the numbers (in bytes) is 255 or less.

Procedure: The program clears the Carry flag initially and adds the operands one byte at a time, starting with the least significant bytes. The final Carry flag reflects the addition of the most significant bytes. A length of 00 causes an immediate exit with no addition.

Registers Used: AF, B, DE, HL

Execution Time: 46 cycles per byte plus 18 cycles overhead

Program Size: 11 bytes

Data Memory Required: None

Special Case: A length of 0 causes an immediate exit with the addend unchanged. The Carry flag is cleared.

Entry Conditions

Base address of addend in HL
Base address of adder in DE
Length of the operands in bytes in B

Exit Conditions

Addend replaced by addend plus adder

Example

1. Data: Length of operands (in bytes) = 6
   Addend = $19D028A193EA_{16}$
   Adder = $293EABF059C7_{16}$

   Result: Addend = $430ED491EDB1_{16}$
   Carry = 0

```
;                                                                ;
;                                                                ;
;                                                                ;
;                                                                ;
;       Title           Multiple-Precision Binary Addition       ;
;       Name:           MPBADD                                   ;
;                                                                ;
;                                                                ;


;                                                                ;
;       Purpose:        Add 2 arrays of binary bytes             ;
;                       Array1 = Array1 + Array2                 ;
;                                                                   ;
;       Entry:          Register pair HL = Base address of array 1  ;
;                       Register pair DE = Base address of array 2  ;
;                       Register B = Length of the arrays           ;
;                                                                   ;
;                         The arrays are unsigned binary numbers with a ;
;                         maximum length of 255 bytes, ARRAY[0] is the  ;
;                         least significant byte, and ARRAY[LENGTH-1]   ;
;                         the most significant byte.                    ;
;                                                                   ;
;       Exit:           Array1 := Array1 + Array2                   ;
;                                                                   ;
;       Registers used: AF,B,DE,HL                                  ;
;                                                                   ;
;       Time:           46 cycles per byte plus 18 cycles overhead  ;
;                                                                   ;
;       Size:           Program 11 bytes                            ;
;                                                                   ;
;                                                                   ;

MPBADD:
        ;CLEAR CARRY, EXIT IF ARRAY LENGTH IS 0
        LD      A,B
        AND     A               ;CLEAR CARRY, TEST ACCUMULATOR
        RET     Z               ;RETURN IF LENGTH = ZERO
LOOP:
        LD      A,(DE)          ;GET NEXT BYTE
        ADC     A,(HL)          ;ADD BYTES
        LD      (HL),A          ;STORE SUM
        INC     HL              ;INCREMENT ARRAY1 POINTER
        INC     DE              ;INCREMENT ARRAY2 POINTER
        DJNZ    LOOP            ;CONTINUE UNTIL COUNTER = 0
        RET


;                                                                   ;
;                                                                   ;
;       SAMPLE EXECUTION:                                           ;
;                                                                   ;
;                                                                   ;


SC6D:
        LD      HL,AY1          ;HL = BASE ADDRESS OF ARRAY 1
        LD      DE,AY2          ;DE = BASE ADDRESS OF ARRAY 2
        LD      B,SZAYS         ;B = LENGTH OF ARRAYS IN BYTES
        CALL    MPBADD          ;ADD THE ARRAYS
                                ;           AY1+0 = 56H
                                ;           AY1+1 = 13H
                                ;           AY1+2 = CFH
                                ;           AY1+3 = 8AH
                                ;           AY1+4 = 67H
                                ;           AY1+5 = 45H
                                ;           AY1+6 = 23H
                                ;           AY1+7 = 01H
        JR      SC6D

SZAYS   EQU     8           ;LENGTH OF ARRAYS IN BYTES
AY1:
        DB      0EFH
        DB      0CDH
        DB      0ABH
        DB      089H
        DB      067H
        DB      045H
        DB      023H
        DB      001H

AY2:
        DB      067H
        DB      045H
        DB      023H
        DB      001H
        DB      0
        DB      0
        DB      0
        DB      0

        END
```

Multiple-Precision Binary Subtraction (MPBSUB)

Subtracts two multi-byte unsigned binary numbers. Both numbers are stored with their least significant bytes at the lowest address. The difference replaces the minuend. The length of the numbers (in bytes) is 255 or less.

Procedure: The program clears the Carry flag initially and subtracts the operands one byte at a time, starting with the least significant bytes. The final Carry flag reflects the subtraction of the most significant bytes. A length of 0 causes an immediate exit with no subtraction.

Registers Used: AF, B, DE, HL

Execution Time: 46 cycles per byte plus 22 cycles overhead

Program Size: 12 bytes

Data Memory Required: None

Special Case: A length of 0 causes an immediate exit with the minuend unchanged (that is, the difference is equal to the minuend). The Carry flag is cleared.

Entry Conditions

Base address of minuend in HL
Base address of subtrahend in DE
Length of the operands in bytes in B

Exit Conditions

Minuend replaced by minuend minus subtrahend

Example

1. Data: Length of operands (in bytes) = 4
   Minuend = $2F5BA7C3_{16}$
   Subtrahend = $14DF35B8_{16}$

   Result: Minuend = $1A7C720B_{16}$
   The Carry flag is set to 0 since no borrow is necessary.

```
;                                                                   ;
;                                                                   ;
;                                                                   ;
;                                                                   ;
;       Title           Multiple-Precision Binary Subtraction       ;
;       Name:           MPBSUB                                      ;
;                                                                   ;
;                                                                   ;
;                                                                      ;
;       Purpose:        Subtract 2 arrays of binary bytes              ;
;                       Minuend = minuend - subtrahend                 ;
;                                                                      ;
;       Entry:          Register pair HL = Base address of minuend     ;
;                       Register pair DE = Base address of subtrahend  ;
;                       Register B = Length of the arrays              ;
;                                                                      ;
;                         The arrays are unsigned binary numbers with a ;
;                         maximum length of 255 bytes, ARRAY[0] is the ;
;                         least significant byte, and ARRAY[LENGTH-1]  ;
;                         the most significant byte.                   ;
;                                                                      ;
;       Exit:           Minuend := minuend - subtrahend                ;
;                                                                      ;
;       Registers used: AF,B,DE,HL                                     ;
;                                                                      ;
;       Time:           46 cycles per byte plus 22 cycles overhead     ;
;                                                                      ;
;       Size:           Program 12 bytes                               ;
;                                                                      ;
;                                                                      ;

MPBSUB:
        ;CLEAR CARRY, EXIT IF ARRAY LENGTH IS 0
        LD      A,B
        AND     A               ;CLEAR CARRY, TEST ACCUMULATOR
        RET     Z               ;RETURN IF LENGTH = ZERO
        EX      DE,HL           ;SWITCH ARRAY POINTERS
                                ; SO HL POINTS TO SUBTRAHEND
LOOP:
        LD      A,(DE)          ;GET NEXT BYTE OF MINUEND
        SBC     A,(HL)          ;SUBTRACT BYTES
        LD      (DE),A          ;STORE DIFFERENCE
        INC     DE              ;INCREMENT MINUEND POINTER
        INC     HL              ;INCREMENT SUBTRAHEND POINTER
        DJNZ    LOOP            ;CONTINUE UNTIL COUNTER = 0
        RET


;                                                                      ;
;                                                                      ;
;       SAMPLE EXECUTION:                                              ;
;                                                                      ;
;                                                                      ;


SC6E:
        LD      HL,AY1          ;HL = BASE ADDRESS OF MINUEND
        LD      DE,AY2          ;DE = BASE ADDRESS OF SUBTRAHEND
        LD      B,SZAYS         ;B = LENGTH OF ARRAYS IN BYTES
        CALL    MPBSUB          ;SUBTRACT THE ARRAYS
                                ;               AY1+0 = 88H
                                ;               AY1+1 = 88H
                                ;               AY1+2 = 88H
                                ;            AY1+3 = 88H
                                ;            AY1+4 = 67H
                                ;            AY1+5 = 45H
                                ;            AY1+6 = 23H
                                ;            AY1+7 = 01H

        JR      SC6E

SZAYS   EQU     8       ;LENGTH OF ARRAYS IN BYTES
AY1:
        DB      0EFH
        DB      0CDH
        DB      0ABH
        DB      089H
        DB      067H
        DB      045H
        DB      023H
        DB      001H

AY2:
        DB      067H
        DB      045H
        DB      023H
        DB      001H
        DB      0
        DB      0
        DB      0
        DB      0

        END
```

Multiple-Precision Binary Multiplication (MPBMUL) 6F

Multiplies two multi-byte unsigned binary numbers. Both numbers are stored with their least significant byte at the lowest address. The product replaces the multiplicand. The length of the numbers (in bytes) is 255 or less. Only the less significant bytes of the product are returned to retain compatibility with other multiple-precision binary operations.

Procedure: The program uses an ordinary shift-and-add algorithm, adding the multiplier to the partial product each time it finds a 1 bit in the multiplicand. The partial product and the multiplicand are shifted through the bit length plus 1; the extra loop moves the final Carry into the product. The program maintains a full double-length unsigned partial product in memory locations starting at HIPROD (more significant bytes) and in the multiplicand (less significant bytes). The less significant bytes of the product replace the multiplicand as it is shifted and examined for 1 bits. A 0 length causes an exit with no multiplication.

Registers Used: AF, BC, DE, HL

Execution Time: Depends on the length of the operands and on the number of 1 bits in the multiplicand (requiring actual additions). If the average number of 1 bits in the multiplicand is four per byte, the execution time is approximately 728 * LENGTH^2 + 883 * LENGTH + 300 cycles where LENGTH is the number of bytes in the operands.

Program Size: 104 bytes

Data Memory Required: 261 bytes anywhere in RAM. This is temporary storage for the more significant bytes of the product (255 bytes starting at address HIPROD), the loop counter (2 bytes starting at address COUNT), the address immediately following the most significant byte of the high product (2 bytes starting at address ENDHP), and the base address of the multiplier (2 bytes starting at address MLIER).

Special Case: A length of 0 causes an immediate exit with the product equal to the multiplicand. The Carry flag is cleared.

Entry Conditions

Base address of multiplicand in HL
Base address of multiplier in DE
Length of the operands in bytes in B

Exit Conditions

Multiplicand replaced by multiplicand times multiplier

Example

1. Data: Length of operands (in bytes) = 04
   Multiplicand = $0005D1F7_{16}$
   Multiplier = $00000AB1_{16}$

   Result: Multiplicand = $3E39D1C7_{16}$

Note that MPBMUL returns only the less significant bytes (that is, the number of bytes in the multiplicand and multiplier) of the product to maintain compatibility with other multiple-precision arithmetic operations. The more significant bits of the product are available starting with their least significant byte at address HIPROD. The user may need to check those bytes for a possible overflow or extend the operands with additional zeros.

```
;                                                                      ;
;                                                                      ;
;                                                                      ;
;                                                                      ;
;       Title           Multiple-Precision Binary Multiplication       ;
;       Name:           MPBMUL                                         ;
;                                                                      ;
;                                                                      ;


;                                                                      ;
;       Purpose:        Multiply 2 arrays of binary bytes              ;
;                       Multiplicand = multiplicand * multiplier       ;
;                                                                      ;
;       Entry:          Register pair HL = Base address of multiplicand ;
;                       Register pair DE = Base address of multiplier  ;
;                       Register B = Length of the arrays              ;
;                                                                      ;
;                         The arrays are unsigned binary numbers with a ;
;                         maximum length of 255 bytes, ARRAY[0] is the ;
;                         least significant byte, and ARRAY[LENGTH-1]  ;
;                         the most significant byte.                   ;
;                                                                      ;
;       Exit:           Multiplicand := multiplicand * multiplier      ;
;                                                                      ;
;       Registers used: AF,BC,DE,HL                                    ;
;                                                                      ;
;       Time:           Assuming the average number of 1 bits in multi- ;
;                       plicand is 4 * length, then the time is approxi-;
;                       mately                                         ;
;                        (728 * length^2) + (883 * length) + 300 cycles ;
;                                                                      ;
;       Size:           Program 104 bytes                              ;
;                       Data    261 bytes                              ;
;                                                                      ;
;                                                                      ;


MPBMUL:
        ;EXIT IF LENGTH IS ZERO
        LD      A,B
        AND     A               ;IS LENGTH OF ARRAYS = 0 ?
        RET     Z               ;YES, EXIT

        ;MAKE POINTERS POINT TO END OF OPERANDS
        LD      C,B             ;BC = LENGTH
        LD      B,0
        ADD     HL,BC           ;END = BASE + LENGTH
        EX      DE,HL           ;DE POINTS TO END OF MULTIPLICAND
        LD      (MLIER),HL      ;SAVE ADDRESS OF MULTIPLIER
        LD      HL,HIPROD
        ADD     HL,BC
        LD      (ENDHP),HL      ;SAVE ADDRESS AT END OF HIPROD

        ;SET COUNT TO NUMBER OF BITS IN ARRAY PLUS 1
        ; COUNT := (LENGTH * 8) + 1
        LD      L,C             ;MOVE LENGTH TO HL
        LD      H,B
        ADD     HL,HL           ;LENGTH * 8, SHIFT LEFT 3 TIMES
        ADD     HL,HL
        ADD     HL,HL
        INC     HL              ;ADD 1
        LD      (COUNT),HL      ;SAVE NUMBER OF BITS TO DO

        ;ZERO HIGH PRODUCT ARRAY
ZEROPD:
        LD      B,C             ;B = LENGTH IN BYTES
        LD      HL,HIPROD       ;GET ADDRESS OF HIPROD
ZEROLP:
        LD      (HL),0          ;STORE 0
        INC     HL
        DJNZ    ZEROLP          ;CONTINUE UNTIL HIPROD ARRAY IS ZERO

        ;MULTIPLY USING SHIFT AND ADD ALGORITHM
        AND     A               ;CLEAR CARRY FIRST TIME THROUGH
LOOP:
        ;SHIFT CARRY INTO HIPROD ARRAY AND LEAST SIGNIFICANT
        ; BIT OF HIPROD ARRAY TO CARRY
        LD      B,C             ;GET LENGTH IN BYTES
        LD      HL,(ENDHP)      ;GET LAST BYTE OF HIPROD + 1
SRPLP:
        DEC     HL              ;BACK UP TO NEXT BYTE
        RR      (HL)
        DJNZ    SRPLP           ;CONTINUE UNTIL INDEX = 0

        ;SHIFT CARRY (NEXT BIT OF LOWER PRODUCT) INTO MOST
        ; SIGNIFICANT BIT OF MULTIPLICAND.
        ; THIS ALSO SHIFTS NEXT BIT OF MULTIPLICAND TO CARRY
        LD      L,E             ;HL = ADDRESS OF END OF MULTIPLICAND
        LD      H,D
        LD      B,C             ;B = LENGTH IN BYTES
SRA1LP:
        DEC     HL              ;BACK UP TO NEXT BYTE
        RR      (HL)
        DJNZ    SRA1LP          ;CONTINUE UNTIL DONE

        ;IF NEXT BIT OF MULTIPLICAND IS 1 THEN
        ;  ADD MULTIPLIER TO HIPROD ARRAY
        JP      NC,DECCNT       ;JUMP IF NEXT BIT IS ZERO

        ;ADD MULTIPLIER TO HIPROD
        PUSH    DE              ;SAVE ADDRESS OF MULTIPLICAND
        LD      DE,(MLIER)      ;DE = ADDRESS OF MULTIPLIER
        LD      HL,HIPROD       ;HL = ADDRESS OF HIPROD
        LD      B,C             ;B = LENGTH IN BYTES
        AND     A               ;CLEAR CARRY
ADDLP:
        LD      A,(DE)          ;GET NEXT MULTIPLIER BYTE
        ADC     A,(HL)          ;ADD TO HIPROD
        LD      (HL),A          ;STORE NEW HIPROD
        INC     DE
        INC     HL
        DJNZ    ADDLP               ;CONTINUE UNTIL DONE
        POP     DE                  ;RESTORE ADDRESS OF MULTIPLICAND

        ;DECREMENT BIT COUNTER, EXIT IF DONE
        ; DOES NOT CHANGE CARRY!
DECCNT:
        LD      A,(COUNT)
        DEC     A
        LD      (COUNT),A
        JP      NZ,LOOP             ;BRANCH IF LSB OF COUNT NOT ZERO
        PUSH    AF                  ;SAVE CARRY
        LD      A,(COUNT+1)         ;GET HIGH BYTE OF COUNT
        AND     A                   ; IS IT ZERO?
        JP      Z,EXIT              ; EXIT IF SO
        DEC     A                   ;DECREMENT HIGH BYTE OF COUNT
        LD      (COUNT+1),A
        POP     AF                  ;RESTORE CARRY
        JP      LOOP                ;CONTINUE

EXIT:
        POP     AF                  ;DROP PSW FROM STACK
        RET                         ;RETURN

        ;DATA
COUNT:  DS      2                   ;TEMPORARY FOR LOOP COUNTER
ENDHP:  DS      2                   ;ADDRESS OF LAST BYTE OF HIPROD + 1
MLIER:  DS      2                   ;ADDRESS OF MULTIPLIER
HIPROD: DS      255                 ;HIGH PRODUCT BUFFER


;                                                                       ;
;                                                                       ;
;       SAMPLE EXECUTION:                                               ;
;                                                                       ;
;                                                                       ;

SC6F:
        LD      HL,AY1              ;HL = ADDRESS OF MULTIPLICAND
        LD      DE,AY2              ;DE = ADDRESS OF MULTIPLIER
        LD      B,SZAYS             ;B = LENGTH OF OPERANDS IN BYTES
        CALL    MPBMUL              ;MULTIPLE-PRECISION BINARY MULTIPLY
                                    ;RESULT OF 12345H * 1234H = 14B60404H
                                    ; IN MEMORY AY1      = 04H
                                    ;           AY1+1    = 04H
                                    ;           AY1+2    = B6H
                                    ;           AY1+3    = 14H
                                    ;           AY1+4    = 00H
                                    ;           AY1+5    = 00H
                                    ;           AY1+6    = 00H

        JR      SC6F

SZAYS   EQU     7       ;LENGTH OF OPERANDS IN BYTES
```

```
AY1:
            DB          045H
            DB          023H
            DB          001H
            DB          0
            DB          0
            DB          0
            DB          0

AY2:
            DB          034H
            DB          012H
            DB          0
            DB          0
            DB          0
            DB          0
            DB          0

            END
```

Multiple-Precision Binary Division

(MPBDIV) 6G

Divides two multi-byte unsigned binary numbers. Both numbers are stored with their least significant byte at the lowest address. The quotient replaces the dividend; the address of the least significant byte of the remainder is in HL. The length of the numbers (in bytes) is 255 or less. The Carry flag is cleared if no errors occur; if a divide by 0 is attempted, the Carry flag is set to 1, the dividend is left unchanged, and the remainder is set to 0.

Procedure: The program divides with the usual shift-and-subtract algorithm, shifting quotient and dividend and placing a 1 bit in the quotient each time a trial subtraction is successful. An extra buffer holds the result of the trial subtraction; that buffer is simply switched with the buffer holding the dividend if the trial subtraction is successful. The program exits immediately, setting the Carry flag, if it finds the divisor to be 0. The Carry flag is cleared otherwise.

Registers Used: AF, BC, DE, HL

Execution Time: Depends on the length of the operands and on the number of 1 bits in the quotient (requiring a buffer switch). If the average number of 1 bits in the quotient is 4 per byte, the execution time is approximately 1176 * LENGTH^2 + 2038 * LENGTH + 515 cycles where LENGTH is the number of bytes in the operands.

Program Size: 161 bytes

Data Memory Required: 522 bytes anywhere in RAM. This is temporary storage for the high dividend (255 bytes starting at address HIDE1), the result of the trial subtraction (255 bytes starting at address HIDE2), the base address of the dividend (2 bytes starting at address DVEND), the base address of the divisor (2 bytes starting at address DVSOR), pointers to the two temporary buffers for the high dividend (2 bytes starting at addresses HDEPTR and ODEPTR, respectively), a loop counter (2 bytes starting at address COUNT), and a subtraction loop counter (1 byte at address SUBCNT).

Special Cases:

1. A length of 0 causes an immediate exit with the Carry flag cleared, the quotient equal to the original dividend, and the remainder undefined.

2. A divisor of 0 causes an exit with the Carry flag set to 1, the quotient equal to the original dividend, and the remainder equal to 0.

Entry Conditions

Base address of dividend in HL
Base address of divisor in DE
Length of the operands in bytes in B

Exit Conditions

Dividend replaced by dividend divided by divisor
If the divisor is non-zero, Carry = 0 and the result is normal.
If the divisor is 0, Carry = 1, the dividend is unchanged, and the remainder is 0.
The remainder is stored starting with its least significant byte at the address in HL.

Example

1. Data: Length of operands (in bytes) = 03
   Divisor = $000F45_{16}$
   Dividend = $35A2F7_{16}$

   Result: Dividend = 000383_{16}
   Remainder (starting at address in HL) = $0003A8_{16}$
   Carry flag is 0 to indicate no divide-by-0 error.

```
;                                                                   ;
;                                                                   ;
;                                                                   ;
;                                                                   ;
;       Title               Multiple-Precision Binary Division      ;
;       Name:               MPBDIV                                  ;
;                                                                   ;
;                                                                   ;


;                                                                   ;
;       Purpose:            Divide 2 arrays of binary bytes         ;
;                           Dividend = dividend / divisor           ;
;                                                                   ;
;       Entry:              Register pair HL = Base address of dividend   ;
;                           Register pair DE = Base address of divisor    ;
;                           Register B = Length of operands in bytes      ;
;                                                                   ;
;                             The arrays are unsigned binary numbers with a ;
;                             maximum length of 255 bytes, ARRAY[0] is the  ;
;                             least significant byte, and ARRAY[LENGTH-1]   ;
;                             the most significant byte.            ;
;                                                                   ;
;       Exit:               Dividend := dividend / divisor          ;
;                           Register pair HL = Base address of remainder  ;
;                           If no errors then                       ;
;                             carry := 0                            ;
;                           ELSE                                    ;
;                             divide-by-0 error                     ;
;                             carry := 1                            ;
;                             dividend unchanged                    ;
;                             remainder := 0                        ;
;                                                                   ;
;       Registers used: AF,BC,DE,HL                                 ;
;                                                                   ;
;       Time:               Assuming there are length/2 1 bits in the     ;
;                           quotient then the time is approximately       ;
;                           (1176 * length^2) + (2038 * length) + 515 cycles;
;                                                                   ;
;       Size:               Program 161 bytes                       ;
;                           Data    522 bytes                       ;
;                                                                             ;
;                                                                             ;

        ;TEST LENGTH OF OPERANDS, INITIALIZE POINTERS
MPBDIV:
        LD      A,B
        OR      A               ;IS LENGTH OF ARRAYS = 0?
        JP      Z,OKEXIT        ;EXIT IF SO
        LD      (DVEND),HL      ;SAVE BASE ADDRESS OF DIVIDEND
        LD      (DVSOR),DE      ;SAVE BASE ADDRESS OF DIVISOR
        LD      C,B             ;C = LENGTH OF OPERANDS

        ;SET COUNT TO NUMBER OF BITS IN THE ARRAYS
        ; COUNT := (LENGTH * 8) + 1
        LD      L,C             ;HL = LENGTH IN BYTES
        LD      H,0
        ADD     HL,HL           ;LENGTH * 2
        ADD     HL,HL           ;LENGTH * 4
        ADD     HL,HL           ;LENGTH * 8
        INC     HL              ;LENGTH * 8 + 1
        LD      (COUNT),HL      ;SAVE BIT COUNT

        ;ZERO BOTH HIGH DIVIDEND ARRAYS
        LD      HL,HIDE1        ;HL = ADDRESS OF HIDE1
        LD      DE,HIDE2        ;DE = ADDRESS OF HIDE2
        LD      B,C             ;B = LENGTH IN BYTES
        SUB     A               ;GET 0 FOR FILL
ZEROLP:
        LD      (HL),A          ;ZERO HIDE1
        LD      (DE),A          ; AND HIDE2
        INC     HL
        INC     DE
        DJNZ    ZEROLP

        ;SET HIGH DIVIDEND POINTER TO HIDE1
        LD      HL,HIDE1
        LD      (HDEPTR),HL

        ;SET OTHER HIGH DIVIDEND POINTER TO HIDE2
        LD      HL,HIDE2
        LD      (ODEPTR),HL

        ;CHECK IF DIVISOR IS ZERO BY LOGICALLY ORING ALL BYTES
        LD      HL,(DVSOR)      ;HL = ADDRESS OF DIVISOR
        LD      B,C             ;B = LENGTH IN BYTES
        SUB     A               ;START LOGICAL OR AT 0
CHKOLP:
        OR      (HL)            ;OR NEXT BYTE
        INC     HL              ;INCREMENT TO NEXT BYTE
        DJNZ    CHKOLP          ;CONTINUE UNTIL ALL BYTES ORED
        OR      A               ;SET FLAGS FROM LOGICAL OR
        JR      Z,EREXIT        ;ERROR EXIT IF DIVISOR IS 0

        ;DIVIDE USING TRIAL SUBTRACTION ALGORITHM
        OR      A               ;CLEAR CARRY FIRST TIME THROUGH
```

```
LOOP:
        ;C = LENGTH
        ;DE = ADDRESS OF DIVISOR
        ;CARRY = NEXT BIT OF QUOTIENT
        ;SHIFT CARRY INTO LOWER DIVIDEND ARRAY AS NEXT BIT OF QUOTIENT
        ; AND MOST SIGNIFICANT BIT OF LOWER DIVIDEND TO CARRY
        LD      B,C             ;B = NUMBER OF BYTES TO ROTATE
        LD      HL,(DVEND)      ;HL = ADDRESS OF DIVIDEND
SLLP1:
        RL      (HL)            ;ROTATE BYTE OF DIVIDEND LEFT
        INC     HL              ;NEXT BYTE
        DJNZ    SLLP1           ;CONTINUE UNTIL ALL BYTES SHIFTED

        ;DECREMENT BIT COUNTER AND EXIT IF DONE
        ;CARRY IS NOT CHANGED!
DECCNT:
        LD      A,(COUNT)
        DEC     A
        LD      (COUNT),A
        JR      NZ,CONT         ;CONTINUE IF LOWER BYTE NOT ZERO
        LD      A,(COUNT+1)
        DEC     A
        LD      (COUNT+1),A
        JP      M,OKEXIT        ;EXIT WHEN COUNT BECOMES NEGATIVE

        ;SHIFT CARRY INTO LSB OF UPPER DIVIDEND
CONT:
        LD      HL,(HDEPTR)     ;HL = CURRENT HIGH DIVIDEND POINTER
        LD      B,C             ;B = LENGTH IN BYTES
SLLP2:
        RL      (HL)            ;ROTATE BYTE OF UPPER DIVIDEND
        INC     HL              ;INCREMENT TO NEXT BYTE
        DJNZ    SLLP2           ;CONTINUE UNTIL ALL BYTES SHIFTED

        ;SUBTRACT DIVISOR FROM HIGH DIVIDEND, PLACE DIFFERENCE IN
        ; OTHER HIGH DIVIDEND ARRAY
        PUSH    BC              ;SAVE LENGTH
        LD      A,C
        LD      (SUBCNT),A      ;SUBCNT = LENGTH IN BYTES
        LD      BC,(ODEPTR)     ;BC = OTHER DIVIDEND
        LD      DE,(HDEPTR)     ;DE = HIGH DIVIDEND
        LD      HL,(DVSOR)      ;HL = DIVISOR
        OR      A               ;CLEAR CARRY
SUBLP:
        LD      A,(DE)          ;NEXT BYTE OF HIGH DIVIDEND
        SBC     A,(HL)          ;SUBTRACT DIVISOR
        LD      (BC),A          ;SAVE IN OTHER HIGH DIVIDEND
        INC     HL              ;INCREMENT POINTERS
        INC     DE
        INC     BC
        LD      A,(SUBCNT)      ;DECREMENT COUNT
        DEC     A
        LD      (SUBCNT),A
        JR      NZ,SUBLP        ;CONTINUE UNTIL DIFFERENCE COMPLETE
        POP     BC              ;RESTORE LENGTH
        ;IF CARRY IS 1, HIGH DIVIDEND IS LESS THAN DIVISOR
        ; SO NEXT BIT OF QUOTIENT IS 0. IF CARRY IS 0
        ; NEXT BIT OF QUOTIENT IS 1 AND WE REPLACE DIVIDEND
        ; WITH REMAINDER BY SWITCHING POINTERS.
        CCF                     ;COMPLEMENT BORROW SO IT EQUALS
                                ; NEXT BIT OF QUOTIENT
        JR      NC,LOOP         ;JUMP IF NEXT BIT OF QUOTIENT 0
        LD      HL,(HDEPTR)     ;OTHERWISE EXCHANGE HDEPTR AND ODEPTR
        LD      DE,(ODEPTR)
        LD      (ODEPTR),HL
        LD      (HDEPTR),DE

        ;CONTINUE WITH NEXT BIT OF QUOTIENT 1 (CARRY = 1)
        JP      LOOP

        ;SET CARRY TO INDICATE DIVIDE-BY-ZERO ERROR
EREXIT:
        SCF                     ;SET CARRY, INVALID RESULT
        JP      EXIT

        ;CLEAR CARRY TO INDICATE NO ERRORS
OKEXIT:
        OR      A               ;CLEAR CARRY, VALID RESULT

        ;ARRAY 1 IS QUOTIENT
        ;HDEPTR CONTAINS ADDRESS OF REMAINDER
EXIT:   LD      HL,(HDEPTR)     ;HL = BASE ADDRESS OF REMAINDER
        RET

        ;DATA
DVEND:  DS      2               ;ADDRESS OF DIVIDEND
DVSOR:  DS      2               ;ADDRESS OF DIVISOR
HDEPTR: DS      2               ;ADDRESS OF CURRENT HIGH DIVIDEND ARRAY
ODEPTR: DS      2               ;ADDRESS OF OTHER HIGH DIVIDEND ARRAY
COUNT:  DS      2               ;TEMPORARY FOR LOOP COUNTER
SUBCNT: DS      1               ;SUBTRACT LOOP COUNT
HIDE1:  DS      255             ;HIGH DIVIDEND BUFFER 1
HIDE2:  DS      255             ;HIGH DIVIDEND BUFFER 2


;                                                                       ;
;                                                                       ;
;       SAMPLE EXECUTION:                                               ;
;                                                                       ;
;                                                                       ;

SC6G:
        LD      HL,AY1          ;HL = BASE ADDRESS OF DIVIDEND
        LD      DE,AY2          ;DE = BASE ADDRESS OF DIVISOR
        LD      B,SZAYS         ;B = LENGTH OF ARRAYS IN BYTES
        CALL    MPBDIV          ;MULTIPLE-PRECISION BINARY DIVIDE
                                ;RESULT OF 14B60404H / 1234H = 12345H
                                ; IN MEMORY AY1     = 45H
                                ;           AY1+1   = 23H
                                ;           AY1+2   = 01H
```

```
                                        ;          AY1+3    = 00H
                                        ;          AY1+4    = 00H
                                        ;          AY1+5    = 00H
                                        ;          AY1+6    = 00H

        JR      SC6G

SZAYS   EQU     7           ;LENGTH OF ARRAYS IN BYTES
AY1:
        DB      004H
        DB      004H
        DB      0B6H
        DB      014H
        DB      0
        DB      0
        DB      0

AY2:
        DB      034H
        DB      012H
        DB      0
        DB      0
        DB      0
        DB      0
        DB      0

        END
```

Multiple-Precision Binary Comparison (MPBCMP) 6H

Compares two multi-byte unsigned binary numbers and sets the Carry and Zero flags appropriately. The Zero flag is set to 1 if the operands are equal and to 0 if they are not equal. The Carry flag is set to 1 if the subtrahend is larger than the minuend; the Carry flag is cleared otherwise. Thus, the flags are set as if the subtrahend had been subtracted from the minuend.

Procedure: The program compares the operands one byte at a time, starting with the most significant bytes and continuing until it finds corresponding bytes that are not equal. If all the bytes are equal, it exits with the Zero flag set to 1. Note that the comparison works through the operands starting with the most significant bytes, whereas the subtraction (Subroutine 6E) starts with the least significant bytes.

Registers Used: AF, BC, DE, HL

Execution Time: 44 cycles per byte that must be examined plus approximately 60 cycles overhead. That is, the program continues until it finds corresponding bytes that are not the same; each pair of bytes it must examine requires 44 cycles.

Examples:

1. Comparing two 6-byte numbers that are equal:
   $44 * 6 + 60 = 324$ cycles
2. Comparing two 8-byte numbers that differ in the next to most significant bytes:
   $44 * 2 + 60 = 148$ cycles

Program Size: 19 bytes

Data Memory Required: None

Special Case: A length of 0 causes an immediate exit with the Carry flag cleared and the Zero flag set to 1.

Entry Conditions

Base address of minuend in HL
Base address of subtrahend in DE
Length of the operands in bytes in B

Exit Conditions

Flags set as if subtrahend had been subtracted from minuend.

Zero flag = 1 if subtrahend and minuend are equal, 0 if they are not equal.

Carry flag = 1 if subtrahend is larger than minuend in the unsigned sense, 0 if it is less than or equal to the minuend.

Examples

1. Data: Length of operands (in bytes) = 6
   Subtrahend = $19D028A193EA_{16}$
   Minuend = $4E67BC15A266_{16}$

   Result: Zero flag = 0 (operands are not equal)
   Carry flag = 0 (subtrahend is not larger than minuend)

3. Data: Length of operands (in bytes) = 6
   Subtrahend = $19D028A193EA_{16}$
   Minuend = $0F37E5991D7C_{16}$

   Result: Zero flag = 0 (operands are not equal)
   Carry flag = 1 (subtrahend is larger than minuend)

2. Data: Length of operands (in bytes) = 6
   Subtrahend = $19D028A193EA_{16}$
   Minuend = $19D028A193EA_{16}$

   Result: Zero flag = 1 (operands are equal)
   Carry flag = 0 (subtrahend is not larger than minuend)

```
;                                                                     ;
;                                                                     ;
;                                                                     ;
;                                                                     ;
;       Title           Multiple-Precision Binary Comparison          ;
;       Name:           MPBCMP                                        ;
;                                                                     ;
;                                                                     ;
;                                                                     ;

;                                                                     ;
;       Purpose:        Compare 2 arrays of binary bytes and return   ;
;                       the Carry and Zero flags set or cleared       ;
;                                                                     ;
;       Entry:          Register pair HL = Base address of minuend    ;
;                       Register pair DE = Base address of subtrahend ;
;                       Register B = Length of operands in bytes      ;
;                                                                     ;
;                         The arrays are unsigned binary numbers with a ;
;                         maximum length of 255 bytes, ARRAY[0] is the ;
;                         least significant byte, and ARRAY[LENGTH-1] ;
;                         the most significant byte.                  ;
;                                                                     ;
;       Exit:           IF minuend = subtrahend THEN                  ;
;                         C=0,Z=1                                     ;
;                       IF minuend > subtrahend THEN                  ;
;                         C=0,Z=0                                     ;
;                       IF minuend < subtrahend THEN                  ;
;                         C=1,Z=0                                     ;
;                                                                     ;
;       Registers used: AF,BC,DE,HL                                   ;
;                                                                     ;
;       Time:           44 cycles per byte that must be examined plus ;
;                       60 cycles overhead                            ;
;                                                                     ;
;       Size:           Program 19 bytes                              ;
;                                                                     ;
;                                                                     ;

MPBCMP:
        ;TEST LENGTH OF OPERANDS, SET POINTERS TO MSB'S
        LD      A,B
        OR      A               ;IS LENGTH OF ARRAYS = 0?
        RET     Z               ;YES, EXIT WITH C=0, Z=1
        LD      C,B             ;BC = LENGTH
        LD      B,0
        ADD     HL,BC
        EX      DE,HL           ;DE POINTS TO END OF MINUEND
        ADD     HL,BC           ;HL POINTS TO END OF SUBTRAHEND
        LD      B,C             ;B = LENGTH
        OR      A               ;CLEAR CARRY INITIALLY
        ;SUBTRACT BYTES, STARTING WITH MOST SIGNIFICANT
        ;EXIT WITH FLAGS SET IF CORRESPONDING BYTES NOT EQUAL
LOOP:
        DEC     HL              ;BACK UP TO LESS SIGNIFICANT BYTE
        DEC     DE
        LD      A,(DE)          ;GET NEXT BYTE OF MINUEND
        SBC     A,(HL)          ;SUBTRACT BYTE OF SUBTRAHEND
        RET     NZ              ;RETURN IF NOT EQUAL WITH FLAGS
                                ; SET
        DJNZ    LOOP            ;CONTINUE UNTIL ALL BYTES COMPARED
        RET                     ;EQUAL, RETURN WITH C=0, Z=1


;                                                                       ;
;                                                                       ;
;       SAMPLE EXECUTION:                                               ;
;                                                                       ;
;                                                                       ;


SC6H:
        LD      HL,AY1          ;HL = BASE ADDRESS OF MINUEND
        LD      DE,AY2          ;DE = BASE ADDRESS OF SUBTRAHEND
        LD      B,SZAYS         ;B = LENGTH OF OPERANDS IN BYTES
        CALL    MPBCMP          ;MULTIPLE-PRECISION BINARY COMPARISON
                                ;RESULT OF COMPARE(7654321H,1234567H) IS
                                ; C=0,Z=0

        JR      SC6H

SZAYS   EQU     7       ;LENGTH OF OPERANDS IN BYTES
AY1:
        DB      021H
        DB      043H
        DB      065H
        DB      007H
        DB      0
        DB      0
        DB      0

AY2:
        DB      067H
        DB      045H
        DB      023H
        DB      001H
        DB      0
        DB      0
        DB      0

        END
```

Multiple-Precision Decimal Addition (MPDADD)

Adds two multi-byte unsigned decimal numbers. Both numbers are stored with their least significant digits at the lowest address. The sum replaces the addend. The length of the numbers (in bytes) is 255 or less.

Procedure: The program first clears the Carry flag and then adds the operands one byte (two digits) at a time, starting with the least significant digits. The sum replaces the addend. A length of 00 causes an immediate exit with no addition. The final Carry flag reflects the addition of the most significant digits.

Registers Used: AF, B, DE, HL

Execution Time: 50 cycles per byte plus 18 cycles overhead

Program Size: 12 bytes

Data Memory Required: None

Special Case: A length of 0 causes an immediate exit with the addend unchanged and the Carry flag cleared.

Entry Conditions

Base address of addend in HL
Base address of adder in DE
Length of the operands in bytes in register B

Exit Conditions

Addend replaced by addend plus adder

Example

1. Data: Length of operands (in bytes) = 6
   Addend = 196028819315_{16}
   Adder = 293471605987_{16}

   Result: Addend = 489500425302_{16}
   Carry = 0

```
;                                                               ;
;                                                               ;
;                                                               ;
;                                                               ;
;       Title           Multiple-Precision Decimal Addition     ;
;       Name:           MPDADD                                  ;
;                                                               ;
;                                                               ;
;       Purpose:        Add 2 arrays of BCD bytes                     ;
;                       Array1 = Array1 + Array2                      ;
;                                                                     ;
;       Entry:          Register pair HL = Base address of array 1    ;
;                       Register pair DE = Base address of array 2    ;
;                       Register B = Length of arrays in bytes        ;
;                                                                     ;
;                         The arrays are unsigned BCD numbers with a  ;
;                         maximum length of 255 bytes, ARRAY[0] is the ;
;                         least significant byte, and ARRAY[LENGTH-1] ;
;                         the most significant byte.                  ;
;                                                                     ;
;       Exit:           Array1 := Array1 + Array2                     ;
;                                                                     ;
;       Registers used: A,B,DE,F,HL                                   ;
;                                                                     ;
;       Time:           50 cycles per byte plus 18 cycles overhead    ;
;                                                                     ;
;       Size:           Program 12 bytes                              ;
;                                                                     ;
;                                                                     ;

MPDADD:
        ;TEST ARRAY LENGTH FOR ZERO, CLEAR CARRY
        LD      A,B
        OR      A               ;TEST LENGTH AND CLEAR CARRY
        RET     Z               ;EXIT IF LENGTH IS 0
        ;ADD OPERANDS 2 DIGITS AT A TIME
        ;  NOTE CARRY IS 0 INITIALLY
LOOP:
        LD      A,(DE)
        ADC     A,(HL)          ;ADD NEXT BYTES
        DAA                     ;CHANGE TO DECIMAL
        LD      (HL),A          ;STORE SUM
        INC     HL              ;INCREMENT TO NEXT BYTE
        INC     DE
        DJNZ    LOOP            ;CONTINUE UNTIL ALL BYTES SUMMED
        RET


;                                                                     ;
;                                                                     ;
;       SAMPLE EXECUTION:                                             ;
;                                                                     ;
;                                                                     ;


SC6I:
        LD      HL,AY1          ;HL = BASE ADDRESS OF ARRAY 1
        LD      DE,AY2          ;DE = BASE ADDRESS OF ARRAY 2
        LD      B,SZAYS         ;B = LENGTH OF ARRAYS IN BYTES
        CALL    MPDADD          ;MULTIPLE-PRECISION BCD ADDITION
                                ;RESULT OF 1234567 + 1234567 = 2469134
                                ; IN MEMORY AY1     = 34H
                                ;           AY1+1   = 91H
                                ;           AY1+2   = 46H
```

```
                                        ;          AY1+3   = 02H
                                        ;          AY1+4   = 00H
                                        ;          AY1+5   = 00H
                                        ;          AY1+6   = 00H

        JR      SC6I

SZAYS   EQU     7       ;LENGTH OF ARRAYS IN BYTES
AY1:
        DB      067H
        DB      045H
        DB      023H
        DB      001H
        DB      0
        DB      0
        DB      0

AY2:
        DB      067H
        DB      045H
        DB      023H
        DB      001H
        DB      0
        DB      0
        DB      0

        END
```

Multiple-Precision Decimal Subtraction (MPDSUB) 6J

Subtracts two multi-byte unsigned decimal numbers. Both numbers are stored with their least significant digits at the lowest address. The difference replaces the minuend. The length of the numbers (in bytes) is 255 or less.

Procedure: The program first clears the Carry flag and then subtracts the subtrahend from the minuend one byte (two digits) at a time, starting with the least significant digits. A length of 0 causes an immediate exit with no subtraction. The final Carry flag reflects the subtraction of the most significant digits.

Registers Used: A, B, DE, F, HL

Execution Time: 50 cycles per byte plus 22 cycles overhead

Program Size: 13 bytes

Data Memory Required: None

Special Case: A length of 0 causes an immediate exit with the minuend unchanged (that is, the difference is equal to the minuend). The Carry flag is cleared.

Entry Conditions

Base address of minuend in HL
Base address of subtrahend in DE
Length of the operands in bytes in B

Exit Conditions

Minuend replaced by minuend minus subtrahend

Example

1 Data: Length of operands (in bytes) = 6
Minuend = 293471605987_{16}
Subtrahend = 196028819315_{16}

Result: Minuend = 097442786672_{16}
Carry = 0, since no borrow is necessary

```
;                                                                        ;
;                                                                        ;
;                                                                        ;
;                                                                        ;
;       Title           Multiple-Precision Decimal Subtraction          ;
;       Name:           MPDSUB                                           ;
;                                                                        ;
;                                                                        ;

;                                                                        ;
;       Purpose:        Subtract 2 arrays of BCD bytes                 ;
;                       Minuend = minuend - subtrahend                 ;
;                                                                      ;
;       Entry:          Register pair HL = Base address of minuend     ;
;                       Register pair DE = Base address of subtrahend  ;
;                       Register B = Length of arrays in bytes         ;
;                                                                      ;
;                         The arrays are unsigned BCD numbers with a   ;
;                         maximum length of 255 bytes, ARRAY[0] is the ;
;                         least significant byte, and ARRAY[LENGTH-1]  ;
;                         the most significant byte.                   ;
;                                                                      ;
;       Exit:           Minuend := minuend - subtrahend                ;
;                                                                      ;
;       Registers used: A,B,DE,F,HL                                    ;
;                                                                      ;
;       Time:           50 cycles per byte plus 22 cycles overhead     ;
;                                                                      ;
;       Size:           Program 13 bytes                               ;
;                                                                      ;
;                                                                      ;

MPDSUB:
        ;TEST ARRAY LENGTH FOR ZERO, CLEAR CARRY
        LD      A,B
        OR      A               ;TEST ARRAY LENGTH, CLEAR CARRY
        RET     Z               ;EXIT IF LENGTH IS 0
        EX      DE,HL           ;HL = SUBTRAHEND
                                ;DE = MINUEND
        ;SUBTRACT OPERANDS 2 DIGITS AT A TIME
        ; NOTE CARRY IS INITIALLY 0
LOOP:
        LD      A,(DE)          ;GET BYTE OF MINUEND
        SBC     A,(HL)          ;SUBTRACT BYTE OF SUBTRAHEND
        DAA                     ;CHANGE TO DECIMAL
        LD      (DE),A          ;STORE BYTE OF DIFFERENCE
        INC     HL              ;INCREMENT TO NEXT BYTE
        INC     DE
        DJNZ    LOOP            ;CONTINUE UNTIL ALL BYTES SUBTRACTED
        RET


;                                                                      ;
;                                                                      ;
;       SAMPLE EXECUTION:                                              ;
;                                                                      ;
;                                                                      ;

SC6J:
        LD      HL,AY1          ;HL = BASE ADDRESS OF MINUEND
        LD      DE,AY2          ;DE = BASE ADDRESS OF SUBTRAHEND
        LD      B,SZAYS         ;B = LENGTH OF ARRAYS IN BYTES
        CALL    MPDSUB          ;MULTIPLE-PRECISION BCD SUBTRACTION
                                ;RESULT OF 2469134 - 1234567 = 1234567
                                ; IN MEMORY AY1     = 67H
                                ;           AY1+1   = 45H
                                ;              AY1+2    = 23H
                                ;              AY1+3    = 01H
                                ;              AY1+4    = 00H
                                ;              AY1+5    = 00H
                                ;              AY1+6    = 00H

        JR      SC6J

SZAYS   EQU     7       ;LENGTH OF ARRAYS IN BYTES
AY1:
        DB      034H
        DB      091H
        DB      046H
        DB      002H
        DB      0
        DB      0
        DB      0

AY2:
        DB      067H
        DB      045H
        DB      023H
        DB      001H
        DB      0
        DB      0
        DB      0

        END
```

Multiple-Precision Decimal Multiplication (MPDMUL) 6K

Multiplies two multi-byte unsigned decimal numbers. Both numbers are stored with their least significant digits at the lowest address. The product replaces the multiplicand. The length of the numbers (in bytes) is 255 or less. Only the least significant bytes of the product are returned to retain compatibility with other multiple-precision decimal operations.

Procedure: The program handles each digit of the multiplicand separately. It masks the digit off, shifts it (if it is the upper nibble of a byte), and then uses it as a counter to determine how many times to add the multiplier to the partial product. The least significant digit of the partial product is saved as the next digit of the full product and the partial product is shifted right four bits. The program uses a flag to determine whether it is currently working with the upper or lower digit of a byte. A length of 00 causes an exit with no multiplication.

Registers Used: AF, BC, DE, HL

Execution Time: Depends on the length of the operands and on the size of the digits in the multiplicand (since those digits determine how many times the multiplier must be added to the partial product). If the average digit in the multiplicand has a value of 5, then the execution time is approximately 694 * LENGTH^2 + 1555 * LENGTH + 272 cycles where LENGTH is the number of bytes in the operands.

Program Size: 167 bytes

Data Memory Required: 520 bytes anywhere in RAM. This is temporary storage for the high bytes of the partial product (255 bytes starting at address PROD), the multiplicand (255 bytes starting at address MCAND), the length of the arrays (1 byte at address LEN), a digit counter indicating upper or lower digit (1 byte at address DCNT), a loop counter (1 byte at address LPCNT), an overflow byte (1 byte at address OVRFLW), pointers to the multiplicand and multiplier (2 bytes each starting at addresses MCADR and MPADR, respectively), and the next byte of the multiplicand (1 byte at address NBYTE).

Special Case: A length of 0 causes an immediate exit with the multiplicand unchanged. The more significant bytes of the product (starting at address PROD) are undefined.

Entry Conditions

Base address of multiplicand in HL
Base address of multiplier in DE
Length of the operands in bytes in B

Exit Conditions

Multiplicand replaced by multiplicand times multiplier

Example

1. Data: Length of operands (in bytes) = 04
   Multiplier = 00003518_{16}
   Multiplicand = 00006294_{16}

   Result: Multiplicand = 22142292_{16}

Note that MPDMUL returns only the less significant bytes of the product (that is, the number of bytes in the multiplicand and multiplier) to maintain compatibility with other multiple-precision decimal arithmetic operations. The more significant bytes of the product are available starting with their least significant digits at address PROD. The user may need to check those bytes for a possible overflow or extend the operands with zeros.

```
;                                                                        ;
;                                                                        ;
;                                                                        ;
;                                                                        ;
;       Title           Multiple-Precision Decimal Multiplication        ;
;       Name:           MPDMUL                                           ;
;                                                                        ;
;                                                                        ;

;                                                                        ;
;       Purpose:        Multiply 2 arrays of BCD bytes                   ;
;                       Multiplicand = multiplicand * multiplier         ;
;                                                                        ;
;       Entry:          Register pair HL = Multiplicand base address     ;
;                       Register pair DE = Multiplier base address       ;
;                       Register B = Length of arrays in bytes           ;
;                                                                        ;
;                         The arrays are unsigned BCD numbers with a     ;
;                         maximum length of 255 bytes, ARRAY[0] is the   ;
;                         least significant byte, and ARRAY[LENGTH-1]    ;
;                         the most significant byte.                     ;
;                                                                        ;
;       Exit:           Multiplicand := multiplicand * multiplier        ;
;                                                                        ;
;       Registers used: AF,BC,DE,HL                                      ;
;                                                                        ;
;       Time:           Assuming the average digit value of multiplicand ;
;                       is 5, the time is approximately                  ;
;                        (694 * length^2) + (1555 * length) + 272 cycles;
;                                                                        ;
;       Size:           Program 167 bytes                                ;
;                       Data    520 bytes                                ;
;                                                                        ;
;                                                                        ;


MPDMUL:
        ;INITIALIZE COUNTERS AND POINTERS
        LD      A,B             ;TEST LENGTH OF OPERANDS
        OR      A
        RET     Z               ;EXIT IF LENGTH IS 0
        LD      (LEN),A         ;SAVE LENGTH
        LD      (LPCNT),A       ;LOOP COUNTER = LENGTH IN BYTES
        LD      (MCADR),HL      ;SAVE MULTIPLICAND ADDRESS
        LD      (MPADR),DE      ;SAVE MULTIPLIER ADDRESS
        ;SAVE MULTIPLICAND IN TEMPORARY BUFFER (MCAND)
        LD      DE,MCAND          ;DE POINTS TO TEMPORARY MULTIPLICAND
        LD      (NBYTE),DE
                                  ;HL POINTS TO MULTIPLICAND
        LD      C,B               ;BC = LENGTH
        LD      B,0
        LDIR                      ;MOVE MULTIPLICAND TO BUFFER
        ;CLEAR PARTIAL PRODUCT, CONSISTING OF UPPER BYTES
        ;  STARTING AT PROD AND LOWER BYTES REPLACING
        ;  MULTIPLICAND
        LD      HL,(MCADR)
        LD      A,(LEN)
        CALL    ZEROBUF           ;ZERO MULTIPLICAND

        ;ZERO PRODUCT
        LD      HL,PROD
        CALL    ZEROBUF           ;ZERO PRODUCT ARRAY

        ;
        ;LOOP THROUGH ALL BYTES OF MULTIPLICAND
LOOP:
        LD      A,1
        LD      (DCNT),A          ;START WITH LOWER DIGIT

        ;LOOP THROUGH 2 DIGITS PER BYTE
        ; DURING LOWER DIGIT DCNT = 1
        ; DURING UPPER DIGIT DCNT = 0
DLOOP:
        SUB     A                 ;A = 0
        LD      (OVRFLW),A        ;CLEAR OVERFLOW BYTE
        LD      A,(DCNT)
        OR      A                 ;TEST FOR LOWER DIGIT (Z=0)
        LD      HL,(NBYTE)        ;GET NEXT BYTE
        LD      A,(HL)
        JR      NZ,DLOOP1         ;JUMP IF LOWER DIGIT
        RRCA                      ;SHIFT UPPER DIGIT RIGHT 4 BITS
        RRCA
        RRCA
        RRCA
DLOOP1:
        AND     0FH               ;KEEP ONLY CURRENT DIGIT
        JR      Z,SDIGIT          ;BRANCH IF DIGIT IS ZERO
        LD      C,A               ;C = DIGIT

        ;ADD MULTIPLIER TO PRODUCT NDIGIT TIMES
ADDLP:
        LD      HL,(MPADR)        ;HL = MULTIPLIER ADDRESS
        LD      DE,PROD           ;DE = PRODUCT ADDRESS
        LD      A,(LEN)
        LD      B,A               ;B = LENGTH
        OR      A                 ;CLEAR CARRY INITIALLY

INNER:
        LD      A,(DE)            ;GET NEXT BYTE OF PRODUCT
        ADC     A,(HL)            ;ADD NEXT BYTE OF MULTIPLIER
        DAA                         ;DECIMAL ADJUST
        LD      (DE),A              ;STORE SUM IN PRODUCT
        INC     HL
        INC     DE
        DJNZ    INNER               ;CONTINUE UNTIL ALL BYTES ADDED
        JR      NC,DECND            ;JUMP IF NO OVERFLOW FROM ADDITION
        LD      HL,OVRFLW           ;ELSE INCREMENT OVERFLOW BYTE
        INC     (HL)

DECND:
        DEC     C
        JR      NZ,ADDLP            ;CONTINUE UNTIL DIGIT = 0

        ;STORE LEAST SIGNIFICANT DIGIT OF PRODUCT
        ; AS NEXT DIGIT OF MULTIPLICAND
SDIGIT:
        LD      A,(PROD)            ;GET LOW BYTE OF PRODUCT
        AND     0FH
        LD      B,A                 ;SAVE IN B
        LD      A,(DCNT)
        OR      A                   ;TEST FOR LOWER DIGIT (Z=0)
        LD      A,B                 ;A = NEXT DIGIT
        JR      NZ,SD1              ;JUMP IF WORKING ON LOWER DIGIT
        RRCA                        ;ELSE MOVE DIGIT TO HIGH BITS
        RRCA
        RRCA
        RRCA
SD1:
        LD      HL,(MCADR)          ;PLACE NEXT DIGIT IN MULTIPLICAND
        OR      (HL)
        LD      (HL),A

        ;SHIFT PRODUCT RIGHT 1 DIGIT (4 BITS)
        LD      A,(LEN)
        LD      B,A                 ;B = LENGTH
        LD      E,A
        LD      D,0
        LD      HL,PROD
        ADD     HL,DE               ;HL POINTS BEYOND END OF PROD
        LD      A,(OVRFLW)          ;A = OVERFLOW BYTE

SHFTLP:
        DEC     HL                  ;DECREMENT, POINT TO NEXT BYTE
        RRD                         ;ROTATE BYTE OF PRODUCT RIGHT 1 DIGIT
        DJNZ    SHFTLP              ;CONTINUE UNTIL DONE

        ;CHECK IF DONE WITH BOTH DIGITS OF THIS BYTE
        LD      HL,DCNT             ;ARE WE ON LOWER DIGIT?
        DEC     (HL)
        JR      Z,DLOOP             ;YES, DO UPPER DIGIT OF SAME BYTE

        ;INCREMENT TO NEXT BYTE AND SEE IF DONE
        LD      HL,(NBYTE)          ;INCREMENT TO NEXT MULTIPLICAND BYTE
        INC     HL
        LD      (NBYTE),HL
        LD      HL,(MCADR)          ;INCREMENT TO NEXT RESULT BYTE
        INC     HL
        LD      (MCADR),HL
        LD      HL,LPCNT            ;DECREMENT LOOP COUNTER
        DEC     (HL)
        JR      NZ,LOOP

EXIT:
        RET

        ;----------------------------------------
        ;ROUTINE: ZEROBUF
        ;PURPOSE: ZERO A BUFFER
        ;ENTRY: HL POINTS TO FIRST BYTE OF BUFFER
        ;       LEN = LENGTH OF BUFFER
        ;EXIT:  BUFFER ZEROED
        ;REGISTERS USED: AF,BC,DE,HL
        ;----------------------------------------
ZEROBUF:
        LD      (HL),0              ;ZERO FIRST BYTE
        LD      A,(LEN)
        DEC     A
        RET     Z                   ;RETURN IF ONLY ONE BYTE
        LD      D,H
        LD      E,L
        INC     DE                  ;DE = SECOND BYTE
        LD      C,A                 ;BC = LENGTH OF ARRAY
        LD      B,0
        LDIR                        ;CLEAR REST OF BUFFER BY
        RET                         ; PROPAGATING ZEROS FROM ONE
                                    ; BYTE TO THE NEXT
        ;DATA
LEN:    DS      1                   ;LENGTH OF ARRAYS
DCNT:   DS      1                   ;DIGIT COUNTER FOR BYTES
LPCNT:  DS      1                   ;LOOP COUNTER
OVRFLW: DS      1                   ;OVERFLOW BYTE
MCADR:  DS      2                   ;NEXT BYTE TO STORE INTO
MPADR:  DS      2                   ;ADDRESS OF MULTIPLIER
NBYTE:  DS      2                   ;NEXT DIGIT OF MULTIPLICAND
PROD:   DS      255                 ;PRODUCT BUFFER
MCAND:  DS      255                 ;MULTIPLICAND BUFFER


;                                                                  ;
;                                                                  ;
;       SAMPLE EXECUTION:                                          ;
;                                                                  ;
;                                                                  ;


SC6K:
        LD      HL,AY1              ;BASE ADDRESS OF MULTIPLICAND
        LD      DE,AY2              ;BASE ADDRESS OF MULTIPLIER
        LD      B,SZAYS             ;LENGTH OF ARRAYS IN BYTES
        CALL    MPDMUL              ;MULTIPLE-PRECISION BCD MULTIPLICATION
                                    ;RESULT OF 1234 * 1234 = 1522756
```

```
                                ; IN MEMORY AY1      = 56H
                                ;           AY1+1    = 27H
                                ;           AY1+2    = 52H
                                ;           AY1+3    = 01H
                                ;           AY1+4    = 00H
                                ;           AY1+5    = 00H
                                ;           AY1+6    = 00H

        JR      SC6K

SZAYS   EQU     7               ;LENGTH OF ARRAYS IN BYTES
AY1:
        DB      034H
        DB      012H
        DB      0
        DB      0
        DB      0
        DB      0
        DB      0

AY2:
        DB      034H
        DB      012H
        DB      0
        DB      0
        DB      0
        DB      0
        DB      0

        END
```

Multiple-Precision Decimal Division (MPDDIV)

6L

Divides two multi-byte unsigned decimal numbers. Both numbers are stored with their least significant digits at the lowest address. The quotient replaces the dividend; the remainder is not returned, but its base address is in memory locations HDEPTR and HDEPTR+1. The length of the numbers (in bytes) is 255 or less. The Carry flag is cleared if no errors occur; if a divide by 0 is attempted, the Carry flag is set to 1, the dividend is unchanged, and the remainder is set to 0.

Procedure: The program divides by determining how many times the divisor can be subtracted from the dividend. It saves that number in the quotient, makes the remainder into the new dividend, and rotates the dividend and the quotient left one digit. The program exits immediately, setting the Carry flag, if it finds the divisor to be 0. The Carry flag is cleared otherwise.

Registers Used: AF, BC, DE, HL

Execution Time: Depends on the length of the operands and on the size of the digits in the quotient (determining how many times the divisor must be subtracted from the dividend). If the average digit in the quotient has a value of 5, the execution time is approximately 1054 * LENGTH^2 + 2297 * LENGTH + 390 cycles where LENGTH is the number of bytes in the operands.

Program Size: 168 bytes

Data Memory Required: 523 bytes anywhere in RAM. This is storage for the high dividend (255 bytes starting at address HIDE1), the result of the subtraction (255 bytes starting at address HIDE2), the length of the operands (1 byte at address LENGTH), the next digit in the array (1 byte at address NDIGIT), the counter for the subtraction loop (1 byte at address CNT), pointers to the dividend, divisor, current high dividend and remainder, and other high dividend (2 bytes each starting at addresses DVADR, DSADR, HDEPTR, and ODEPTR, respectively), and the divide loop counter (2 bytes starting at address COUNT).

Special Cases:

1. A length of 0 causes an immediate exit with the Carry flag cleared, the quotient equal to the original dividend, and the remainder undefined.

2. A divisor of 0 causes an exit with the Carry flag set to 1, the quotient equal to the original dividend, and the remainder equal to 0.

Entry Conditions

Base address of dividend in HL
Base address of divisor in DE
Length of the operands in bytes in B

Exit Conditions

Dividend replaced by dividend divided by divisor
If the divisor is non-zero, Carry = 0 and the result is normal.
If the divisor is 0, Carry = 1, the dividend is unchanged, and the remainder is 0.
The base address of the remainder (i.e., the address of its least significant digits) is in HDEPTR and HDEPTR+1.

Example

1. Data: Length of operands (in bytes) = 04
   Dividend = 22142298_{16}
   Divisor = 00006294_{16}

   Result: Dividend = 00003518_{16}
   Remainder (base address in HDEPTR and HDEPTR + 1) = 00000006_{16}
   Carry flag is 0 to indicate no divide-by-0 error.

```
;                                                                      ;
;                                                                      ;
;                                                                      ;
;                                                                      ;
;       Title              Multiple-Precision Decimal Division        ;
;       Name:              MPDDIV                                      ;
;                                                                      ;
;                                                                      ;

;                                                                      ;
;       Purpose:           Divide 2 arrays of BCD bytes                ;
;                          Quotient := dividend / divisor              ;
;                                                                      ;
;       Entry:             Register pair HL = Base address of dividend ;
;                          Register pair DE = Base address of divisor  ;
;                          Register B = Length of operands in bytes    ;
;                                                                      ;
;                            The arrays are unsigned BCD numbers with a ;
;                            maximum length of 255 bytes, ARRAY[0] is the ;
;                            least significant byte, and ARRAY[LENGTH-1] ;
;                            the most significant byte.                ;
;                                                                      ;
;       Exit:              Dividend := dividend / divisor              ;
;                          Remainder := base address in HDEPTR         ;
;                          If no errors then                           ;
;                            carry := 0                                ;
;                          ELSE                                        ;
;                            divide-by-0 error                         ;
;                            carry := 1                                ;
;                            dividend unchanged                        ;
;                            remainder := 0                            ;
;                                                                      ;
;       Registers used: AF,BC,DE,HL                                    ;
;                                                                      ;
;       Time:              Assuming the average digit value in the     ;
;                          quotient is 5 then the time is approximately ;
;                          (1054 * length^2) + (2297 * length) + 390 cycles;
;                                                                      ;
;       Size:              Program 168 bytes                           ;
;                          Data    523 bytes                           ;
;                                                                      ;
```

```
MPDDIV:
        ;SAVE PARAMETERS AND CHECK FOR ZERO LENGTH
        LD      (DVADR),HL      ;SAVE DIVIDEND ADDRESS
        LD      (DSADR),DE      ;SAVE DIVISOR ADDRESS
        LD      A,B
        LD      (LENGTH),A      ;SAVE LENGTH
        OR      A               ;TEST LENGTH
        JP      Z,OKEXIT        ;EXIT IF LENGTH = 0

        ;ZERO BOTH DIVIDEND BUFFERS
        ; AND SET UP THE DIVIDEND POINTERS
        LD      HL,HIDE1        ;HL = ADDRESS OF HIGH DIVIDEND 1
        LD      (HDEPTR),HL     ;HIGH DIVIDEND PTR = HIDE1
        LD      DE,HIDE2        ;DE = ADDRESS OF HIGH DIVIDEND 2
        LD      (ODEPTR),DE     ;OTHER DIVIDEND PTR = HIDE2
        SUB     A               ;GET 0 TO USE IN FILLING BUFFERS

                                ;B = LENGTH IN BYTES
        ;FILL BOTH DIVIDEND BUFFERS WITH ZEROS
INITLP:
        LD      (HL),A          ;ZERO BYTE OF HIDE1
        LD      (DE),A          ;ZERO BYTE OF HIDE2
        INC     HL
        INC     DE
        DJNZ    INITLP

        ;SET COUNT TO NUMBER OF DIGITS PLUS 1
        ; COUNT := (LENGTH * 2) + 1;
        LD      A,(LENGTH)      ;EXTEND LENGTH TO 16 BITS
        LD      L,A
        LD      H,0
        ADD     HL,HL           ;LENGTH * 2
        INC     HL              ;LENGTH * 2 + 1
        LD      (COUNT),HL      ;COUNT = LENGTH * 2 + 1

        ;CHECK FOR DIVIDE BY ZERO
        ; LOGICALLY OR ENTIRE DIVISOR TO SEE IF ALL BYTES ARE 0
        LD      HL,(DSADR)      ;HL = ADDRESS OF DIVISOR
        LD      A,(LENGTH)
        LD      B,A             ;B = LENGTH IN BYTES
        SUB     A               ;START LOGICAL OR WITH 0
DVO1:
        OR      (HL)            ;OR NEXT BYTE OF DIVISOR
        INC     HL
        DJNZ    DVO1
        OR      A               ;TEST FOR ZERO DIVISOR
        JR      Z,EREXIT        ;ERROR EXIT IF DIVISOR IS 0

        SUB     A
        LD      (NDIGIT),A      ;START NEXT DIGIT AT 0

        ;DIVIDE BY DETERMINING HOW MANY TIMES DIVISOR CAN
        ;  BE SUBTRACTED FROM DIVIDEND FOR EACH DIGIT
        ;  POSITION
```

```
DVLOOP:
        ;ROTATE LEFT LOWER DIVIDEND AND QUOTIENT:
        ; HIGH DIGIT OF NDIGIT BECOMES LEAST SIGNIFICANT DIGIT
        ; OF QUOTIENT (DIVIDEND ARRAY) AND MOST SIGNIFICANT DIGIT
        ; OF DIVIDEND ARRAY GOES TO HIGH DIGIT OF NDIGIT
        LD      HL,(DVADR)
        CALL    RLARY           ;ROTATE LOW DIVIDEND

        ;IF DIGIT COUNT = 0 THEN WE ARE DONE
        LD      HL,(COUNT)      ;DECREMENT COUNT BY 1
        DEC     HL
        LD      (COUNT),HL
        LD      A,H             ;TEST 16-BIT COUNT FOR 0
        OR      L
        JR      Z,OKEXIT        ;EXIT WHEN COUNT = 0

        ;
        ;ROTATE LEFT HIGH DIVIDEND, LEAST SIGNIFICANT DIGIT
        ; OF HIGH DIVIDEND BECOMES HIGH DIGIT OF NDIGIT
        LD      HL,(HDEPTR)

        CALL    RLARY           ;ROTATE HIGH DIVIDEND

        ;
        ;SEE HOW MANY TIMES DIVISOR GOES INTO HIGH DIVIDEND
        ; ON EXIT FROM THIS LOOP, HIGH DIGIT OF NDIGIT IS NEXT
        ; QUOTIENT DIGIT AND HIGH DIVIDEND IS REMAINDER
        SUB     A               ;CLEAR NUMBER OF TIMES INITIALLY
        LD      (NDIGIT),A

SUBLP:
        LD      HL,(DSADR)      ;HL POINTS TO DIVISOR
        LD      DE,(HDEPTR)     ;DE POINTS TO CURRENT HIGH DIVIDEND
        LD      BC,(ODEPTR)     ;BC POINTS TO OTHER HIGH DIVIDEND
        LD      A,(LENGTH)
        LD      (CNT),A         ;LOOP COUNTER = LENGTH
        OR      A               ;CLEAR CARRY INITIALLY

INNER:
        LD      A,(DE)          ;GET NEXT BYTE OF DIVIDEND
        SBC     A,(HL)          ;SUBTRACT DIVISOR
        DAA                     ;CHANGE TO DECIMAL
        LD      (BC),A          ;STORE DIFFERENCE IN OTHER DIVIDEND
        INC     HL              ;INCREMENT TO NEXT BYTE
        INC     DE
        INC     BC
        LD      A,(CNT)         ;DECREMENT COUNTER
        DEC     A
        LD      (CNT),A
        JR      NZ,INNER        ;CONTINUE THROUGH ALL BYTES
        JR      C,DVLOOP        ;JUMP WHEN BORROW OCCURS
                                ;NDIGIT IS NUMBER OF TIMES DIVISOR
                                ; GOES INTO ORIGINAL HIGH DIVIDEND
                                ;HIGH DIVIDEND CONTAINS REMAINDER
        ;DIFFERENCE IS NOT NEGATIVE, SO ADD 1 TO
        ;  NUMBER OF SUCCESSFUL SUBTRACTIONS
        ;  (LOW DIGIT OF NDIGIT)
        LD      HL,NDIGIT       ;NDIGIT = NDIGIT + 1
        INC     (HL)

        ;EXCHANGE POINTERS, THUS MAKING DIFFERENCE NEW DIVIDEND
        LD      HL,(HDEPTR)
        LD      DE,(ODEPTR)
        LD      (HDEPTR),DE
        LD      (ODEPTR),HL
        JR      SUBLP           ;CONTINUE UNTIL DIFFERENCE NEGATIVE

        ;NO ERRORS, CLEAR CARRY
OKEXIT:
        OR      A               ;CLEAR CARRY, VALID RESULT
        RET

        ;DIVIDE-BY-ZERO ERROR, SET CARRY
EREXIT:
        SCF                     ;SET CARRY, INVALID RESULT
        RET


        ;**********************************
        ;SUBROUTINE: RLARY
        ;PURPOSE:    ROTATE LEFT AN ARRAY ONE DIGIT (4 BITS)
        ;ENTRY: HL = BASE ADDRESS OF ARRAY
        ;       LOW DIGIT OF NDIGIT IS DIGIT TO ROTATE THROUGH
        ;EXIT:  ARRAY ROTATED LEFT THROUGH LOW DIGIT OF NDIGIT
        ;REGISTERS USED: AF, BC, DE, HL
        ;**********************************

RLARY:
        ;SHIFT NDIGIT INTO LOW DIGIT OF ARRAY AND
        ; SHIFT ARRAY LEFT
        LD      A,(LENGTH)
        LD      B,A             ;B = LENGTH OF ARRAY IN BYTES
        LD      A,(NDIGIT)      ;A = NDIGIT

SHIFT:
        RLD                     ;SHIFT BYTE LEFT 1 DIGIT (4 BITS)
        INC     HL
        DJNZ    SHIFT           ;CONTINUE UNTIL ALL BYTES SHIFTED
        LD      (NDIGIT),A      ;SAVE NEW NEXT DIGIT
        RET


        ;DATA
LENGTH: DS      1               ;LENGTH OF ARRAYS IN BYTES
NDIGIT: DS      1               ;NEXT DIGIT IN ARRAY
CNT:    DS      1               ;COUNTER FOR SUBTRACT LOOP
DVADR:  DS      2               ;DIVIDEND ADDRESS
DSADR:  DS      2               ;DIVISOR ADDRESS
HDEPTR: DS      2               ;HIGH DIVIDEND POINTER
ODEPTR: DS      2               ;OTHER DIVIDEND POINTER
```

```
COUNT:  DS      2                 ;DIVIDE LOOP COUNTER
HIDE1:  DS      255               ;HIGH DIVIDEND BUFFER 1
HIDE2:  DS      255               ;HIGH DIVIDEND BUFFER 2


;                                                                       ;
;                                                                       ;
;       SAMPLE EXECUTION:                                               ;
;                                                                       ;
;                                                                       ;

SC6L:
        LD      HL,AY1            ;BASE ADDRESS OF DIVIDEND
        LD      DE,AY2            ;BASE ADDRESS OF DIVISOR
        LD      B,SZAYS           ;LENGTH OF ARRAYS IN BYTES
        CALL    MPDDIV            ;MULTIPLE-PRECISION BCD DIVISION
                                  ;RESULT OF 1522756 / 1234 = 1234
                                  ; IN MEMORY AY1      = 34H
                                  ;           AY1+1    = 12H
                                  ;           AY1+2    = 00H
                                  ;           AY1+3    = 00H
                                  ;           AY1+4    = 00H
                                  ;           AY1+5    = 00H
                                  ;           AY1+6    = 00H

        JR      SC6L

SZAYS   EQU     7       ;LENGTH OF ARRAYS IN BYTES

        AY1:
                DB      056H
                DB      027H
                DB      052H
                DB      01H
                DB      0
                DB      0
                DB      0

        AY2:
                DB      034H
                DB      012H
                DB      0
                DB      0
                DB      0
                DB      0
                DB      0

                END
```

Multiple-Precision Decimal Comparison 6M

Compares two multi-byte unsigned decimal (BCD) numbers and sets the Carry and Zero flags appropriately. The Zero flag is set to 1 if the operands are equal and to 0 if they are not equal. The Carry flag is set to 1 if the subtrahend is larger than the minuend; the Carry flag is cleared otherwise. Thus the flags are set as if the subtrahend had been subtracted from the minuend.

Note: This program is exactly the same as Subroutine 6H, the multiple-precision binary comparison, since the form of the operands does not matter if they are only being compared. See Subroutine 6H for a listing and other details.

Examples

1. Data: Length of operands (in bytes) = 6
   Subtrahend = 196528719340_{16}
   Minuend = 456780153266_{16}

   Result: Zero flag = 0 (operands are not equal)
   Carry flag = 0 (subtrahend is not larger than minuend)

2. Data: Length of operands (in bytes) = 6
   Subtrahend = 196528719340_{16}
   Minuend = 196528719340_{16}

   Result: Zero flag = 1 (operands are equal)
   Carry flag = 0 (subtrahend is not larger than minuend)

3. Data: Length of operands (in bytes) = 6
   Subtrahend = 196528719340_{16}
   Minuend = 073785991074_{16}

   Result: Zero flag = 0 (operands are not equal)
   Carry flag = 1 (subtrahend is larger than minuend)

Bit Field Extraction (BFE) 7A

Extracts a field of bits from a byte and returns the field in the least significant bit positions. The width of the field and its lowest bit position are parameters.

Procedure: The program obtains a mask with the specified number of 1 bits from a table, shifts the mask left to align it with the specified lowest bit position, and obtains the field by logically ANDing the mask with the data. It then normalizes the bit field by shifting it right so that it starts in bit 0.

Registers Used: AF, BC, DE, HL

Execution Time: 21 * LOWEST BIT POSITION plus 86 cycles overhead. (The lowest bit position determines the number of times the mask must be shifted left and the bit field right.)

Program Size: 32 bytes

Data Memory Required: None

Special Cases:

1. Requesting a field that would extend beyond the end of the byte causes the program to return with only the bits through bit 7. That is, no wraparound is provided. If, for example, the user asks for a 6-bit field starting at bit 5, the program will return only 3 bits (bits 5 through 7).

2. Both the lowest bit position and the number of bits in the field are interpreted mod 8. That is, for example, bit position 11 is equivalent to bit position 3 and a field of 10 bits is equivalent to a field of 2 bits. Note, however, that the number of bits in the field is interpreted in the range 1 to 8. That is, a field of 16 bits is equivalent to a field of 8 bits, not to a field of 0 bits.

3. Requesting a field of width 0 causes a return with a result of 0.

Entry Conditions

Starting (lowest) bit position in the field (0 to 7) in A
Number of bits in the field (1 to 8) in D
Data byte in E

Exit Conditions

Bit field in A (normalized to bit 0)

Examples

1. Data: Data value = $F6_{16} = 11110110_2$
   Lowest bit position = 4
   Number of bits in the field = 3

   Result: Bit field = $07_{16} = 00000111_2$
   Three bits, starting at bit 4, have been extracted (that is, bits 4 through 6).

2. Data: Data value = $A2_{16} = 10100010_2$
   Lowest bit position = 6
   Number of bits in the field = 5

   Result: Bit field = $02_{16} = 00000010_2$
   Two bits, starting at bit 6, have been extracted (that is, bits 6 and 7); that was all that was available, although five bits were requested.

```
;                                                                  ;
;                                                                  ;
;                                                                  ;
;                                                                  ;
;       Title           Bit Field Extraction                       ;
;       Name:           BFE                                        ;
;                                                                  ;
;                                                                  ;

;                                                                  ;
;       Purpose:        Extract a field of bits from a byte and    ;
;                       return the field normalized to bit 0       ;
;                       NOTE: IF THE REQUESTED FIELD IS TOO LONG, THEN ;
;                             ONLY THE BITS THROUGH BIT 7 WILL BE  ;
;                             RETURNED. FOR EXAMPLE, IF A 4-BIT FIELD IS;
;                             REQUESTED STARTING AT BIT 7, ONLY 1  ;
;                             BIT (BIT 7) WILL BE RETURNED.        ;
;                                                                  ;
;       Entry:          Register D = Number of bits in field (1 to 8) ;
;                       Register E = Data byte                     ;
;                       Register A = Starting (lowest) bit position in ;
;                                    the field (0 to 7)            ;
;                                                                  ;
;       Exit:           Register A = Field                         ;
;                                                                  ;
;       Registers used: AF,BC,DE,HL                                ;
;                                                                  ;
;       Time:           86 cycles overhead plus                    ;
;                         (21 * lowest bit position) cycles        ;
;                                                                  ;
;       Size:           Program 32 bytes                           ;
;                                                                  ;
;                                                                  ;

BFE:
        ;SHIFT DATA TO NORMALIZE TO BIT 0
        ;  NO SHIFTING NEEDED IF LOWEST POSITION IS 0
        AND     00000111B       ;ONLY ALLOW POSITIONS 0 TO 7
        JR      Z,EXTR          ;JUMP IF NO SHIFTING NEEDED
        LD      B,A             ;MOVE SHIFT COUNT TO B
SHFT:
        SRL     E               ;SHIFT DATA RIGHT
        DJNZ    SHFT            ;CONTINUE UNTIL NORMALIZED

        ;EXTRACT FIELD BY MASKING WITH 1'S
EXTR:
        LD      A,D             ;TEST NUMBER OF BITS FOR ZERO
        OR      A
        RET     Z               ;EXIT IF NUMBER OF BITS = 0
                                ;  FIELD IS 0 ON EXIT
        DEC     A               ;DECREMENT A TO NORMALIZE TO 0
        AND     00000111B       ;ONLY ALLOW 0 THROUGH 7
        LD      C,A             ;BC = INDEX INTO MASK ARRAY
        LD      B,0
        LD      HL,MSKARY       ;HL = BASE OF MASK ARRAY
        ADD     HL,BC
        LD      A,E             ;GET DATA
        AND     (HL)            ;MASK OFF UNWANTED BITS
        RET

        ;MASK ARRAY WITH 1 TO 8 ONE BITS
MSKARY:
        DB      00000001B
        DB      00000011B
        DB      00000111B
        DB      00001111B
        DB      00011111B
        DB      00111111B
        DB      01111111B
        DB      11111111B


;                                                                       ;
;                                                                       ;
;       SAMPLE EXECUTION:                                               ;
;                                                                       ;
;                                                                       ;

SC7A:
        LD      E,00011000B     ;REGISTER E = DATA
        LD      D,3             ;REGISTER D = NUMBER OF BITS
        LD      A,2             ;ACCUMULATOR = LOWEST BIT POSITION
        CALL    BFE             ;EXTRACT 3 BITS STARTING WITH #2

        JR      SC7A            ;  RESULT = 00000110B

        END
```

Bit Field Insertion (BFI) 7B

Inserts a field of bits into a byte. The width of the field and its starting (lowest) bit position are parameters.

Procedure: The program obtains a mask with the specified number of 0 bits from a table. It then shifts the mask and the bit field left to align them with the specified lowest bit position. It logically ANDs the mask with the original data byte, thus clearing the required bit positions, and then logically ORs the result with the shifted bit field.

Registers Used: AF, BC, DE, HL

Execution Time: 25 * LOWEST BIT POSITION plus 133 cycles overhead. (The lowest bit position of the field determines how many times the mask and the field must be shifted left.)

Program Size: 40 bytes

Data Memory Required: None

Special Cases:

1. Attempting to insert a field that would extend beyond the end of the byte causes the program to insert only the bits through bit 7. That is, no wraparound is provided. If, for example, the user attempts to insert a 6-bit field starting at bit 4, only 4 bits (bits 4 through 7) are actually replaced.

2. Both the starting bit position and the width of the bit field (number of bits) are interpreted mod 8. That is, for example, bit position 11 is the same as bit position 3 and a 12-bit field is the same as a 4-bit field. Note, however, that the width of the field is mapped into the range 1 to 8. That is, for example, a 16-bit field is the same as an 8-bit field.

3. Attempting to insert a field of width 0 causes a return with a result of 0.

Entry Conditions

Data in A
Number of bits in the field (1 to 8) in B
Starting (lowest) bit position of field in C
Field to insert in E

Exit Conditions

Result in A
The result is the original data with the bit field inserted, starting at the specified bit position.

Examples

1. Data: Value = $F6_{16}$ = 11110110_2
Lowest bit position = 4
Number of bits in the field = 2
Bit field = 01_{16} = 00000001_2

Result: Value with bit field inserted = $D6_{16}$ = 11010110_2
The 2-bit field has been inserted into the original value starting at bit 4 (into bits 4 and 5).

2. Data: Value = $B8_{16}$ = 10111000_2
Lowest bit position = 1
Number of bits in the field = 5
Bit field = 15_{16} = 00010101_2

Result: Value with bit field inserted = AA_{16} = 10101010_2
The 5-bit field has been inserted into the original value starting at bit 1 (into bits 1 through 5), changing 11100_2 ($1C_{16}$) to 10101_2 (15_{16}).

```
;                                                                  ;
;                                                                  ;
;                                                                  ;
;                                                                  ;
;       Title           Bit Field Insertion                        ;
;       Name:           BFI                                        ;
;                                                                  ;
;                                                                  ;


;                                                                  ;
;       Purpose:        Insert a field of bits into a byte and return  ;
;                       the byte                                   ;
;                       NOTE: IF THE REQUESTED FIELD IS TOO LONG,  ;
;                             ONLY THE BITS THROUGH BIT 7 WILL BE  ;
;                             INSERTED. FOR EXAMPLE, IF A 4-BIT FIELD IS;
;                             TO BE INSERTED STARTING AT BIT 7 THEN  ;
;                             ONLY 1 BIT (BIT 7) WILL BE INSERTED.  ;
;                                                                  ;
;       Entry:          Register A = Byte of data                  ;
;                       Register B = Number of bits in the field (1  ;
;                                    to 8)                         ;
;                       Register C = Starting (lowest) bit position in  ;
;                                    which the data will be inserted  ;
;                                    (0 to 7)                      ;
;                       Register E = Field to insert               ;
;                                                                  ;
;       Exit:           Register A = Data                          ;
;                                                                  ;
;       Registers used: AF,BC,DE,HL                                ;
;                                                                  ;
;       Time:           133 cycles overhead plus                   ;
;                         (25 * starting bit position) cycles      ;
;                                                                  ;
;       Size:           Program 40 bytes                           ;
;                                                                  ;
;                                                                  ;

BFI:
        PUSH    AF                  ;SAVE DATA BYTE

        ;GET MASK WITH REQUIRED NUMBER OF 0 BITS
        PUSH    BC                  ;SAVE STARTING BIT POSITION
        LD      HL,MSKARY
        LD      A,B                 ;GET NUMBER OF BITS
        AND     A                   ;TEST NUMBER OF BITS FOR 0
        RET     Z                   ;RETURN WITH 0 RESULT IF NUMBER
                                    ; OF BITS IS 0
        DEC     A                   ;NORMALIZE TO 0...7
        AND     00000111B           ;ONLY ALLOW 0...7
        LD      C,A
        LD      B,0
        ADD     HL,BC               ;INDEX INTO MASK ARRAY
        LD      D,(HL)              ;D = MASK WITH ZEROS FOR CLEARING
        POP     BC                  ;RESTORE STARTING BIT

        ;TEST IF STARTING BIT IS 0
        LD      A,C
        AND     00000111B       ;RESTRICT STARTING BIT TO 0...7
        JR      Z,INSRT         ;JUMP IF STARTING BIT IS 0
                                ; NO ALIGNMENT IS NECESSARY

        ;ALIGN FIELD TO INSERT AND MASK IF STARTING BIT NON-ZERO
        LD      B,C             ;B = STARTING BIT NUMBER
        LD      A,D             ;A = MASK
SFIELD:
        SLA     E               ;SHIFT FIELD LEFT TO INSERT
        RLCA                    ;ROTATE MASK
        DJNZ    SFIELD          ;CONTINUE UNTIL ALIGNED
        LD      D,A

        ;INSERT FIELD
INSRT:
        POP     AF              ;GET DATA BACK
        AND     D               ;AND OFF MASK AREA
        OR      E               ;OR IN FIELD
        RET

        ;MASK ARRAY - 1 TO 8 ZERO BITS
MSKARY:
        DB      11111110B
        DB      11111100B
        DB      11111000B
        DB      11110000B
        DB      11100000B
        DB      11000000B
        DB      10000000B
        DB      00000000B


;                                                                       ;
;                                                                       ;
;       SAMPLE EXECUTION:                                               ;
;                                                                       ;
;                                                                       ;

SC7B:
        LD      A,11111111B     ;REGISTER A = DATA
        LD      B,3             ;REGISTER B = NUMBER OF BITS
        LD      C,2             ;REGISTER C = LOWEST BIT POSITION
        LD      E,00000101B     ;REGISTER E = FIELD TO INSERT
        CALL    BFI             ;INSERT 3-BIT FIELD STARTING AT

        JR      SC7B            ; BIT 2, RESULT = 11110111B

        END
```

Multiple-Precision Arithmetic Shift Right (MPASR) 7C

Shifts a multi-byte operand right arithmetically by a specified number of bit positions. The length of the operand (in bytes) is 255 or less. The Carry flag is set from the last bit shifted out of the rightmost bit position. The operand is stored with its least significant byte at the lowest address.

Procedure: The program obtains the sign bit from the most significant byte, saves that bit in the Carry, and then rotates the entire operand right one bit, starting with the most significant byte. It repeats the operation for the specified number of shifts.

Registers Used: AF, BC, DE, HL

Execution Time: NUMBER OF SHIFTS * (46+ 34 * LENGTH OF OPERANDS IN BYTES) + 59 cycles

Program Size: 28 bytes

Data Memory Required: None

Special Cases:

1. If the length of the operand is 0, the program exits immediately with the operand unchanged and the Carry flag cleared.

2. If the number of shifts is 0, the program exits immediately with the operand unchanged and the Carry flag cleared.

Entry Conditions

Base address of operand in HL
Length of the operand in bytes in B
Number of shifts (bit positions) in C

Exit Conditions

Operand shifted right arithmetically by the specified number of bit positions. The original sign bit is extended to the right. The Carry flag is set from the last bit shifted out of the rightmost bit position. Carry is cleared if either the number of shifts or the length of the operand is 0.

Examples

1. Data: Length of operand (in bytes) = 08
Operand = $85A4C719FE06741E_{16}$
Number of shifts = 04

Result: Shifted operand = $F85A4C719FE06741_{16}$
This is the original operand shifted right four bits arithmetically; the four most significant bits all take the value of the original sign bit (1).
Carry = 1, since the last bit shifted from the rightmost bit position was 1.

2. Data: Length of operand (in bytes) = 04
Operand = $3F6A42D3_{16}$
Number of shifts = 03

Result: Shifted operand = $07ED485A_{16}$
This is the original operand shifted right three bits arithmetically; the three most significant bits all take the value of the original sign bit (0).
Carry = 0, since the last bit shifted from the rightmost bit position was 0.

```
;                                                                          ;
;                                                                          ;
;                                                                          ;
;                                                                          ;
;       Title           Multiple-Precision Arithmetic Shift Right          ;
;       Name:           MPASR                                              ;
;                                                                          ;
;                                                                          ;


;                                                                          ;
;       Purpose:        Arithmetic shift right a multi-byte operand        ;
;                       N bits                                             ;
;                                                                          ;
;       Entry:          Register pair HL = Base address of operand         ;
;                       Register B = Length of operand in bytes            ;
;                       Register C = Number of bits to shift               ;
;                                                                          ;
;                         The operand is stored with ARRAY[0] as its       ;
;                         least significant byte and ARRAY[LENGTH-1]       ;
;                         its most significant byte, where ARRAY           ;
;                         is its base address.                             ;
;                                                                          ;
;       Exit:           Operand shifted right with the most significant    ;
;                       bit propagated.                                    ;
;                       CARRY := Last bit shifted from least               ;
;                                significant position.                     ;
;                                                                          ;
;       Registers used: AF,BC,DE,HL                                        ;
;                                                                          ;
;       Time:            59 cycles overhead plus                           ;
;                        ((34 * length) + 46) cycles per shift             ;
;                                                                          ;
;       Size:           Program 28 bytes                                   ;
;                                                                          ;
;                                                                          ;

MPASR:
        ;EXIT IF NUMBER OF SHIFTS OR LENGTH OF OPERAND IS 0
        ;OR CLEARS CARRY IN EITHER CASE
        LD      A,C
        OR      A
        RET     Z               ;RETURN IF NUMBER OF SHIFTS IS 0
        LD      A,B
        OR      A
        RET     Z               ;RETURN IF LENGTH OF OPERAND IS 0

        ;CALCULATE ADDRESS OF MOST SIGNIFICANT (LAST) BYTE
        LD      E,B             ;ADDRESS OF MSB = BASE+LENGTH-1
        LD      D,0
        ADD     HL,DE
        DEC     HL              ;HL = ADDRESS OF MSB
                                ;C = NUMBER OF SHIFTS
                                ;A = LENGTH OF OPERAND
        ;LOOP ON NUMBER OF SHIFTS TO PERFORM
        ;INITIAL CARRY = MOST SIGNIFICANT BIT OF ENTIRE OPERAND
```

```
LOOP:
        LD      B,(HL)          ;GET MOST SIGNIFICANT BYTE
        RL      B               ;CARRY = MOST SIGNIFICANT BIT
        LD      B,A
        LD      E,L             ;SAVE ADDRESS OF MSB
        LD      D,H
        ;ROTATE BYTES RIGHT STARTING WITH MOST SIGNIFICANT
ASRLP:
        RR      (HL)            ;ROTATE A BYTE RIGHT
        DEC     HL              ;DECREMENT TO LESS SIGNIFICANT BYTE
        DJNZ    ASRLP
CONT:
        LD      L,E             ;RESTORE ADDRESS OF MSB
        LD      H,D
        DEC     C               ;DECREMENT NUMBER OF SHIFTS
        JR      NZ,LOOP
        RET


;                                                                       ;
;                                                                       ;
;       SAMPLE EXECUTION:                                               ;
;                                                                       ;
;                                                                       ;

SC7C:
        LD      HL,AY           ;BASE ADDRESS OF OPERAND
        LD      B,SZAY          ;LENGTH OF OPERAND IN BYTES
        LD      C,SHIFTS        ;NUMBER OF SHIFTS
        CALL    MPASR           ;SHIFT
                        ;RESULT OF SHIFTING  EDCBA987654321H, 4 BITS IS
                        ;                    FEDCBA98765432H, C=0
                        ;  IN MEMORY AY   = 032H
                        ;            AY+1 = 054H
                        ;            AY+2 = 076H
                        ;            AY+3 = 098H
                        ;            AY+4 = 0BAH
                        ;            AY+5 = 0DCH
                        ;            AY+6 = 0FEH

        JR      SC7C

        ;DATA SECTION
SZAY    EQU     7       ;LENGTH OF OPERAND IN BYTES
SHIFTS  EQU     4       ;NUMBER OF SHIFTS
AY:     DB      21H,43H,65H,87H,0A9H,0CBH,0EDH

        END
```

Multiple-Precision Logical Shift Left (MPLSL) 7D

Shifts a multi-byte operand left logically by a specified number of bit positions. The length of the operand (in bytes) is 255 or less. The Carry flag is set from the last bit shifted out of the leftmost bit position. The operand is stored with its least significant byte at the lowest address.

Procedure: The program clears the Carry initially (to fill with a 0 bit) and then shifts the entire operand left one bit, starting with the least significant byte. It repeats the operation for the specified number of shifts.

Registers Used: AF, BC, DE

Execution Time: NUMBER OF SHIFTS * (27+34* LENGTH OF OPERAND IN BYTES) + 31 cycles

Program Size: 21 bytes

Data Memory Required: None

Special Cases:

1. If the length of the operand is 0, the program exits immediately with the operand unchanged and the Carry flag cleared.
2. If the number of shifts is 0, the program exits immediately with the operand unchanged and the Carry flag cleared.

Entry Conditions

Base address of operand in HL
Length of operand in bytes in B
Number of shifts (bit positions) in C

Exit Conditions

Operand shifted left logically by the specified number of bit positions (the least significant bit positions are filled with 0's). The Carry flag is set from the last bit shifted out of the leftmost bit position. Carry is cleared if either the number of shifts or the length of the operand is 0.

Examples

1. Data: Length of operand (in bytes) = 08
   Operand = $85A4C719FE06741E_{16}$
   Number of shifts = 04

   Result: Shifted operand = $5A4C719FE06741E0_{16}$
   This is the original operand shifted left four bits logically; the four least significant bits are all cleared.
   Carry = 0, since the last bit shifted from the leftmost bit position was 0.

2. Data: Length of operand (in bytes) = 04
   Operand = $3F6A42D3_{16}$
   Number of shifts = 03

   Result: Shifted operand = $FB521698_{16}$
   This is the original operand shifted left three bits logically; the three least significant bits are all cleared.
   Carry = 1, since the last bit shifted from the leftmost bit position was 1.

```
;                                                                    ;
;                                                                    ;
;                                                                    ;
;                                                                    ;
;       Title           Multiple-Precision Logical Shift Left        ;
;       Name:           MPLSL                                        ;
;                                                                    ;
;                                                                    ;

;                                                                    ;
;       Purpose:        Logical shift left a multi-byte operand      ;
;                       N bits                                       ;
;                                                                    ;
;       Entry:          Register pair HL = Base address of operand   ;
;                       Register B = Length of operand in bytes      ;
;                       Register C = Number of bits to shift         ;
;                                                                    ;
;                         The operand is stored with ARRAY[0] as its ;
;                         least significant byte and ARRAY[LENGTH-1] ;
;                         its most significant byte, where ARRAY     ;
;                         is its base address.                       ;
;                                                                    ;
;       Exit:           Operand shifted left filling the least       ;
;                       significant bits with zeros                  ;
;                       CARRY := Last bit shifted from               ;
;                         most significant position                  ;
;                                                                    ;
;       Registers used: AF,BC,DE                                     ;
;                                                                    ;
;       Time:            31 cycles overhead plus                     ;
;                        ((34 * length) + 27) cycles per shift       ;
;                                                                    ;
;       Size:           Program 21 bytes                             ;
;                                                                    ;
;                                                                    ;

MPLSL:
        ;EXIT IF NUMBER OF SHIFTS OR LENGTH OF OPERAND IS 0
        ;OR CLEARS CARRY IN EITHER CASE
        LD      A,C
        OR      A
        RET     Z               ;RETURN IF NUMBER OF SHIFTS IS 0
        LD      A,B
        OR      A
        RET     Z               ;RETURN IF LENGTH OF OPERAND IS 0

        ;LOOP ON NUMBER OF SHIFTS TO PERFORM
        ;A = LENGTH OF OPERAND
        ;C = NUMBER OF SHIFTS
        ;HL = ADDRESS OF LEAST SIGNIFICANT (FIRST) BYTE OF OPERAND
        ;CARRY = 0 INITIALLY FOR LOGICAL SHIFT
LOOP:
        LD      E,L             ;SAVE ADDRESS OF LSB
        LD      D,H
        LD      B,A             ;B = LENGTH OF OPERAND
        OR      A               ;CLEAR CARRY FOR LOGICAL SHIFT
        ;ROTATE BYTES STARTING WITH LEAST SIGNIFICANT
LSLLP:
        RL      (HL)            ;ROTATE NEXT BYTE LEFT
        INC     HL              ;INCREMENT TO MORE SIGNIFICANT BYTE
        DJNZ    LSLLP
        LD      L,E             ;RESTORE ADDRESS OF LSB
        LD      H,D
        DEC     C               ;DECREMENT NUMBER OF SHIFTS
        JR      NZ,LOOP
        RET


;                                                                       ;
;                                                                       ;
;       SAMPLE EXECUTION:                                               ;
;                                                                       ;
;                                                                       ;

SC7D:
        LD      HL,AY           ;HL = BASE ADDRESS OF OPERAND
        LD      B,SZAY          ;B = LENGTH OF OPERAND IN BYTES
        LD      C,SHIFTS        ;C = NUMBER OF SHIFTS
        CALL    MPLSL           ;SHIFT
                        ;RESULT OF SHIFTING  EDCBA987654321H, 4 BITS IS
                        ;                    DCBA9876543210H, C=0
                        ;  IN MEMORY AY   = 010H
                        ;            AY+1 = 032H
                        ;            AY+2 = 054H
                        ;            AY+3 = 076H
                        ;            AY+4 = 098H
                        ;            AY+5 = 0BAH
                        ;            AY+6 = 0DCH

        JR      SC7D

        ;DATA SECTION
SZAY    EQU     7       ;LENGTH OF OPERAND IN BYTES
SHIFTS  EQU     4       ;NUMBER OF SHIFTS
AY:     DB      21H,43H,65H,87H,0A9H,0CBH,0EDH

        END
```

Multiple-Precision Logical Shift Right (MPLSR)

Shifts a multi-byte operand right logically by a specified number of bit positions. The length of the operand (in bytes) is 255 or less. The Carry flag is set from the last bit shifted out of the rightmost bit position. The operand is stored with its least significant byte at the lowest address.

Procedure: The program clears the Carry initially (to fill with a 0 bit) and then shifts the entire operand right one bit, starting with the most significant byte. It repeats the operation for the specified number of shifts.

Registers Used: AF, BC, DE, HL

Execution Time: NUMBER OF SHIFTS * (35 + 34 * LENGTH OF OPERAND IN BYTES) + 59 cycles

Program Size: 26 bytes

Data Memory Required: None

Special Cases:

1. If the length of the operand is 0, the program exits immediately with the operand unchanged and the Carry flag cleared.
2. If the number of shifts is 0, the program exits immediately with the operand unchanged and the Carry flag cleared.

Entry Conditions

Base address of operand in HL
Length of operand in bytes in B
Number of shifts (bit positions) in C

Exit Conditions

Operand shifted right logically by the specified number of bit positions. (The most significant bit positions are filled with 0's.)
The Carry flag is set from the last bit shifted out of the rightmost bit position. Carry is cleared if either the number of shifts or the length of the operand is 0.

Examples

1. Data: Length of the operand (in bytes) = 08
   Operand = $85A4C719FE06741E_{16}$
   Number of shifts = 04

   Result: Shifted operand = $085A4C719FE06741_{16}$
   This is the original operand shifted right four bits logically; the four most significant bits are all cleared.
   Carry = 1, since the last bit shifted from the rightmost bit position was 1.

2. Data: Length of operand (in bytes) = 04
   Operand = $3F6A42D3_{16}$
   Number of shifts = 03

   Result: Shifted operand = $07ED485A_{16}$
   This is the original operand shifted right three bits logically; the three most significant bits are all cleared.
   Carry = 0, since the last bit shifted from the rightmost bit position was 0.

```
;                                                                       ;
;                                                                       ;
;                                                                       ;
;                                                                       ;
;       Title               Multiple-Precision Logical Shift Right      ;
;       Name:               MPLSR                                       ;
;                                                                       ;
;                                                                       ;

;                                                                       ;
;       Purpose:            Logical shift right a multi-byte operand N bits ;
;                                                                       ;
;       Entry:              Register pair HL = Base address of operand  ;
;                           Register B = Length of operand in bytes     ;
;                           Register C = Number of bits to shift        ;
;                                                                       ;
;                             The operand is stored with ARRAY[0] as its ;
;                             least significant byte and ARRAY[LENGTH-1] ;
;                             its most significant byte, where ARRAY    ;
;                             is its base address.                      ;
;                                                                       ;
;       Exit:               Operand shifted right filling the most      ;
;                           significant bits with zeros                 ;
;                                                                       ;
;                           CARRY := Last bit shifted from least        ;
;                                    significant position               ;
;                                                                       ;
;       Registers used: AF,BC,DE,HL                                     ;
;                                                                       ;
;       Time:                59 cycles overhead plus                    ;
;                            ((34 * length) + 35) cycles per shift      ;
;                                                                       ;
;       Size:               Program 26 bytes                            ;
;                                                                       ;
;                                                                       ;

MPLSR:
        ;EXIT IF NUMBER OF SHIFTS OR LENGTH OF OPERAND IS 0
        ;OR CLEARS CARRY IN EITHER CASE
        LD      A,C
        OR      A
        RET     Z               ;RETURN IF NUMBER OF SHIFTS IS 0
        LD      A,B
        OR      A
        RET     Z               ;RETURN IF LENGTH OF OPERAND IS 0

        ;CALCULATE ADDRESS OF MOST SIGNIFICANT (LAST) BYTE
        LD      E,B             ;ADDRESS OF MSB = BASE+LENGTH-1
        LD      D,0
        ADD     HL,DE
        DEC     HL              ;HL = ADDRESS OF MSB
                                ;C = NUMBER OF SHIFTS
                                ;A = LENGTH OF OPERAND
        ;LOOP ON NUMBER OF SHIFTS TO PERFORM
        ;START WITH CARRY = 0 FOR LOGICAL SHIFT
```

```
LOOP:
        OR      A               ;CLEAR CARRY FOR LOGICAL SHIFT
        LD      B,A             ;B = LENGTH OF OPERAND
        LD      E,L             ;SAVE ADDRESS OF MSB
        LD      D,H
        ;ROTATE BYTES STARTING WITH MOST SIGNIFICANT
LSRLP:
        RR      (HL)            ;ROTATE A BYTE RIGHT
        DEC     HL              ;DECREMENT TO LESS SIGNIFICANT BYTE
        DJNZ    LSRLP
        LD      L,E             ;RESTORE ADDRESS OF MSB
        LD      H,D
        DEC     C               ;DECREMENT NUMBER OF SHIFTS
        JR      NZ,LOOP
        RET


;                                                                       ;
;                                                                       ;
;       SAMPLE EXECUTION:                                               ;
;                                                                       ;
;                                                                       ;

SC7E:
        LD      HL,AY     ;HL = BASE ADDRESS OF OPERAND
        LD      B,SZAY    ;B = LENGTH OF OPERAND IN BYTES
        LD      C,SHIFTS  ;C = NUMBER OF SHIFTS
        CALL    MPLSR     ;SHIFT
                         ;RESULT OF SHIFTING  EDCBA987654321H, 4 BITS IS
                         ;                    0EDCBA98765432H, C=0
                         ;  IN MEMORY AY   = 032H
                         ;           AY+1  = 054H
                         ;           AY+2  = 076H
                         ;           AY+3  = 098H
                         ;           AY+4  = 0BAH
                         ;           AY+5  = 0DCH
                         ;           AY+6  = 00EH

        JR      SC7E

        ;DATA SECTION
SZAY    EQU     7        ;LENGTH OF OPERAND IN BYTES
SHIFTS  EQU     4        ;NUMBER OF SHIFTS
AY:     DB      21H,43H,65H,87H,0A9H,0CBH,0EDH

        END
```

Multiple-Precision Rotate Right (MPRR) 7F

Rotates a multi-byte operand right by a specified number of bit positions as if the most significant bit and least significant bit were connected. The length of the operand (in bytes) is 255 or less. The Carry flag is set from the last bit shifted out of the rightmost bit position. The operand is stored with its least significant byte at the lowest address.

Procedure: The program shifts bit 0 of the least significant byte of the operand to the Carry flag and then rotates the entire operand right one bit, starting with the most significant byte. It repeats the operation for the specified number of rotates.

Registers Used: AF, BC, DE, HL, IX

Execution Time: NUMBER OF ROTATES * (58 + 34 * LENGTH OF OPERAND IN BYTES) + 83 cycles

Program Size: 33 bytes

Data Memory Required: None

Special Cases:

1. If the length of the operand is 0, the program exits immediately with the operand unchanged and the Carry flag cleared.
2. If the number of rotates is 0, the program exits immediately with the operand unchanged and the Carry flag cleared.

Entry Conditions

Base address of operand in HL
Length of operand in bytes in B
Number of rotates (bit positions) in C

Exit Conditions

Operand rotated right logically by the specified number of bit positions (the most significant bit positions are filled from the least significant bit positions). The Carry flag is set from the last bit shifted out of the rightmost bit position. Carry is cleared if either the number of rotates or the length of the operand is 0.

Examples

1. Data: Length of operand (in bytes) = 08
   Operand = $85A4C719FE06741E_{16}$
   Number of rotates = 04

   Result: Rotated operand = $E85A4C719FE06741_{16}$
   This is the original operand rotated right four bits; the four most significant bits are equivalent to the original four least significant bits.
   Carry = 1, since the last bit shifted from the rightmost bit position was 1.

2. Data: Length of operand (in bytes) = 04
   Operand = $3F6A42D3_{16}$
   Number of rotates = 03

   Result: Rotated operand = $67ED485A_{16}$
   This is the original operand rotated right three bits; the three most significant bits are equivalent to the original three least significant bits.
   Carry = 0, since the last bit shifted from the rightmost bit position was 0.

```
;                                                                    ;
;                                                                    ;
;                                                                    ;
;                                                                    ;
;       Title               Multiple-Precision Rotate Right          ;
;       Name:               MPRR                                     ;
;                                                                    ;
;                                                                    ;


;                                                                    ;
;       Purpose:            Rotate right a multi-byte operand N bits ;
;                                                                    ;
;       Entry:              Register pair HL = Base address of operand ;
;                           Register B = Length of operand in bytes  ;
;                           Register C = Number of bits to rotate    ;
;                                                                    ;
;                             The operand is stored with ARRAY[0] as its ;
;                             least significant byte and ARRAY[LENGTH-1] ;
;                             its most significant byte, where ARRAY ;
;                             is its base address.                   ;
;                                                                    ;
;       Exit:               Operand rotated right                    ;
;                           CARRY := Last bit shifted from least     ;
;                                    significant position            ;
;                                                                    ;
;       Registers used: AF,BC,DE,HL,IX                               ;
;                                                                    ;
;       Time:                83 cycles overhead plus                 ;
;                            ((34 * length) + 58) cycles per rotate  ;
;                                                                    ;
;       Size:               Program 33 bytes                         ;
;                                                                    ;
;                                                                    ;

MPRR:
        ;EXIT IF NUMBER OF ROTATES OR LENGTH OF OPERAND IS 0
        ;OR CLEARS CARRY IN EITHER CASE
        LD      A,C
        OR      A
        RET     Z               ;RETURN IF NUMBER OF ROTATES IS 0
        LD      A,B
        OR      A
        RET     Z               ;RETURN IF LENGTH OF OPERAND IS 0

        ;CALCULATE ADDRESS OF MOST SIGNIFICANT (LAST) BYTE
        PUSH    HL
        POP     IX              ;IX POINTS TO LSB (FIRST BYTE)
        LD      E,B             ;ADDRESS OF MSB = BASE+LENGTH-1
        LD      D,0
        ADD     HL,DE
        DEC     HL              ;HL POINTS TO MSB (LAST BYTE)
                                ;C = NUMBER OF ROTATES
                                ;A = LENGTH OF OPERAND
        ;LOOP ON NUMBER OF ROTATES TO PERFORM
        ;CARRY = LEAST SIGNIFICANT BIT OF ENTIRE OPERAND
```

```
LOOP:
        LD      B,(IX+0)        ;GET LSB
        RR      B               ;CARRY = BIT 0 OF LSB
        LD      B,A             ;B = LENGTH OF OPERAND IN BYTES
        LD      E,L             ;SAVE ADDRESS OF MSB
        LD      D,H
        ;ROTATE BYTES RIGHT STARTING WITH MOST SIGNIFICANT
RRLP:
        RR      (HL)            ;ROTATE A BYTE RIGHT
        DEC     HL              ;DECREMENT TO LESS SIGNIFICANT BYTE
        DJNZ    RRLP
        LD      L,E             ;RESTORE ADDRESS OF MSB
        LD      H,D
        DEC     C               ;DECREMENT NUMBER OF ROTATES
        JR      NZ,LOOP
        RET


;                                                                       ;
;                                                                       ;
;       SAMPLE EXECUTION:                                               ;
;                                                                       ;
;                                                                       ;

SC7F:
        LD      HL,AY           ;BASE ADDRESS OF OPERAND
        LD      B,SZAY          ;LENGTH OF OPERAND IN BYTES
        LD      C,ROTATS        ;NUMBER OF ROTATES
        CALL    MPRR    ;ROTATE
                        ;RESULT OF ROTATING EDCBA987654321H, 4 BITS IS
                        ;                  1EDCBA98765432H, C=0
                        ;  IN MEMORY AY   = 032H
                        ;            AY+1 = 054H
                        ;            AY+2 = 076H
                        ;            AY+3 = 098H
                        ;            AY+4 = 0BAH
                        ;            AY+5 = 0DCH
                        ;            AY+6 = 01EH

        JR      SC7F

        ;DATA SECTION
SZAY    EQU     7       ;LENGTH OF OPERAND IN BYTES
ROTATS  EQU     4       ;NUMBER OF ROTATES
AY:     DB      21H,43H,65H,87H,0A9H,0CBH,0EDH

        END
```

Multiple-Precision Rotate Left (MPRL) 7G

Rotates a multi-byte operand left by a specified number of bit positions as if the most significant bit and least significant bit were connected. The length of the operand (in bytes) is 255 or less. The Carry flag is set from the last bit shifted out of the leftmost bit position. The operand is stored with its least significant byte at the lowest address.

Procedure: The program shifts bit 7 of the most significant byte of the operand to the Carry flag. It then rotates the entire operand left one bit, starting with the least significant byte. It repeats the operation for the specified number of rotates.

Registers Used: AF, BC, DE, HL, IX

Execution Time: NUMBER OF ROTATES * (58 + 34 * LENGTH OF OPERAND IN BYTES) + 104 cycles

Program Size: 35 bytes

Data Memory Required: None

Special Cases:

1. If the length of the operand is 0, the program exits immediately with the operand unchanged and the Carry flag cleared.

2. If the number of rotates is 0, the program exits immediately with the operand unchanged and the Carry flag cleared.

Entry Conditions

Base address of operand in HL
Length of operand in bytes in B
Number of rotates (bit positions) in C

Exit Conditions

Operand rotated left the specified number of bit positions (the least significant bit positions are filled from the most significant bit positions). The Carry flag is set from the last bit shifted out of the leftmost bit position. Carry is cleared if either the number of rotates or the length of the operand is 0.

Examples

1. Data: Length of operand (in bytes) = 08
Operand = $85A4C719FE06741E_{16}$
Number of rotates = 04

Result: Rotated operand = $5A4C719FE06741E8_{16}$
This is the original operand rotated left four bits; the four least significant bits are equivalent to the original four most significant bits.
Carry = 0, since the last bit shifted from the leftmost bit position was 0.

2. Data: Length of operand (in bytes) = 04
Operand = $3F6A42D3_{16}$
Number of rotates = 03

Result: Rotated operand = $FB521699_{16}$
This is the original operand rotated left three bits; the three least significant bits are equivalent to the original three most significant bits.
Carry = 1, since the last bit shifted from the leftmost bit position was 1.

```
;                                                                    ;
;                                                                    ;
;                                                                    ;
;                                                                    ;
;       Title           Multiple-Precision Rotate Left               ;
;       Name:           MPRL                                         ;
;                                                                    ;
;                                                                    ;

;                                                                    ;
;       Purpose:        Rotate left a multi-byte operand N bits      ;
;                                                                    ;
;       Entry:          Register pair HL = Base address of operand   ;
;                       Register B = Length of operand in bytes      ;
;                       Register C = Number of bits to rotate        ;
;                                                                    ;
;                         The operand is stored with ARRAY[0] as its ;
;                         least significant byte and ARRAY[LENGTH-1] ;
;                         its most significant byte, where ARRAY     ;
;                         is its base address.                       ;
;                                                                    ;
;       Exit:           Operand rotated left                         ;
;                       CARRY := Last bit shifted from most          ;
;                                significant position                ;
;                                                                    ;
;       Registers used: AF,BC,DE,HL,IX                               ;
;                                                                    ;
;       Time:            104 cycles overhead plus                    ;
;                        ((34 * length) + 58) cycles per rotate      ;
;                                                                    ;
;       Size:           Program 35 bytes                             ;
;                                                                    ;
;                                                                    ;

MPRL:
        ;EXIT IF NUMBER OF ROTATES OR LENGTH OF OPERAND IS 0
        ;OR CLEARS CARRY IN EITHER CASE
        LD      A,C
        OR      A
        RET     Z               ;RETURN IF NUMBER OF ROTATES IS 0
        LD      A,B
        OR      A
        RET     Z               ;RETURN IF LENGTH OF OPERAND IS 0

        ;CALCULATE ADDRESS OF MOST SIGNIFICANT (LAST) BYTE
        PUSH    HL              ;SAVE ADDRESS OF FIRST BYTE
        LD      E,B             ;ADDRESS OF MSB = BASE+LENGTH-1
        LD      D,0
        ADD     HL,DE
        DEC     HL
        PUSH    HL
        POP     IX              ;IX POINTS TO MOST SIGNIFICANT BYTE
        POP     HL              ;HL POINTS TO LEAST SIGNIFICANT BYTE
                                ;C = NUMBER OF ROTATES
                                ;A = LENGTH OF OPERAND
        ;LOOP ON NUMBER OF ROTATES TO PERFORM
        ;CARRY = MOST SIGNIFICANT BIT OF ENTIRE OPERAND
LOOP:
        LD      B,(IX+0)        ;GET MOST SIGNIFICANT BYTE
        RL      B               ;CARRY = BIT 7 OF MSB
        LD      B,A             ;B = LENGTH OF OPERAND IN BYTES
        LD      E,L             ;SAVE ADDRESS OF LSB
        LD      D,H
        ;ROTATE BYTES LEFT STARTING WITH LEAST SIGNIFICANT
RLLP:
        RL      (HL)            ;ROTATE A BYTE LEFT
        INC     HL              ;INCREMENT TO MORE SIGNIFICANT BYTE
        DJNZ    RLLP
        LD      L,E             ;RESTORE ADDRESS OF LSB
        LD      H,D
        DEC     C               ;DECREMENT NUMBER OF ROTATES
        JR      NZ,LOOP
        RET


;                                                                       ;
;                                                                       ;
;       SAMPLE EXECUTION:                                               ;
;                                                                       ;
;                                                                       ;

SC7G:
        LD      HL,AY           ;HL = BASE ADDRESS OF OPERAND
        LD      B,SZAY          ;B = LENGTH OF OPERAND IN BYTES
        LD      C,ROTATS        ;C = NUMBER OF ROTATES
        CALL    MPRL    ;ROTATE
                        ;RESULT OF ROTATING  EDCBA987654321H, 4 BITS IS
                        ;                    DCBA987654321EH, C=0
                        ;  IN MEMORY AY    = 01EH
                        ;            AY+1  = 032H
                        ;            AY+2  = 054H
                        ;            AY+3  = 076H
                        ;            AY+4  = 098H
                        ;            AY+5  = 0BAH
                        ;            AY+6  = 0DCH

        JR      SC7G

        ;DATA SECTION
SZAY    EQU     7       ;LENGTH OF OPERAND IN BYTES
ROTATS  EQU     4       ;NUMBER OF ROTATES
AY:     DB      21H,43H,65H,87H,0A9H,0CBH,0EDH

        END
```

String Compare (STRCMP) 8A

Compares two strings and sets the Carry and Zero flags appropriately. The Zero flag is set to 1 if the strings are identical and to 0 otherwise. The Carry flag is set to 1 if the string with the base address in DE (string 2) is larger than the string with the base address in HL (string 1); the Carry flag is set to 0 otherwise. The strings are a maximum of 255 bytes long and the actual characters are preceded by a byte containing the length. If the two strings are identical through the length of the shorter, the longer string is considered to be larger.

Procedure: The program first determines which string is shorter from the lengths that precede the actual characters. It then compares the strings one byte at a time through the length of the shorter. The program exits with the flags set if it finds corresponding bytes that differ. If the strings are the same through the length of the shorter, the program sets the flags by comparing the lengths.

Registers Used: AF, BC, DE, HL

Execution Time:

1. If the strings are not identical through the length of the shorter, the time is 91 + 60 * NUMBER OF CHARACTERS COMPARED. If, for example, the routine compares five characters before finding a disparity, the execution time is
   91 + 60 * 5 = 91 + 300 = 391 cycles
2. If the strings are identical through the length of the shorter, the time is 131 + 60 * LENGTH OF SHORTER STRING. If, for example, the shorter string is eight bytes long, the execution time is
   131 + 60 * 8 = 131 + 480 = 611 cycles

Program Size: 32 bytes

Data Memory Required: Two bytes anywhere in RAM for the lengths of the strings (addresses LENS1 and LENS2).

Entry Conditions

Base address of string 2 in DE
Base address of string 1 in HL

Exit Conditions

Flags set as if string 2 had been subtracted from string 1. If the strings are the same through the length of the shorter, the flags are set as if the length of string 2 had been subtracted from the length of string 1.

Zero flag = 1 if strings are identical, 0 if they are not.

Carry flag = 1 if string 2 is larger than string 1, 0 if they are identical or string 1 is larger. If the strings are the same through the length of the shorter, the longer one is considered to be larger.

Examples

1. Data: String 1 = 05'PRINT' (05 is the length of the string)
   String 2 = 03'END' (03 is the length of the string)

   Result: Zero flag = 0 (strings are not identical)
   Carry flag = 0 (string 2 is not larger than string 1)

2. Data: String 1 = 05'PRINT' (05 is the length of the string)
   String 2 = 02'PR' (02 is the length of the string)

   Result: Zero flag = 0 (strings are not identical)
   Carry flag = 0 (string 2 is not larger than string 1)

The longer string (string 1) is considered to be larger. If you want to determine whether string 2 is an abbreviation of string 1, you could use Subroutine 8C (Find the Position of a Substring) and determine whether string 2 was part of string 1 and started at the first character.

3. Data: String 1 = 05'PRINT' (05 is the length of the string)
   String 2 = 06'SYSTEM' (06 is the length of the string)

   Result: Zero flag = 0 (strings are not identical)
   Carry flag = 1 (string 2 is larger than string 1)

We are assuming here that the strings consist of ASCII characters. Note that the byte preceding the actual characters contains a hexadecimal number (the length of the string), not a character. We have represented this byte as two hexadecimal digits in front of the string. The string itself is shown surrounded by single quotation marks. These serve only to delimit strings in the examples; they are not actually part of the data. This format is used to display string data in the examples throughout this chapter.

This routine treats spaces like other characters. If, for example, the strings are ASCII, the routine will find that SPRINGMAID is larger than SPRING MAID, since an ASCII M ($4D_{16}$) is larger than an ASCII space (20_{16}).

```
;                                                                      ;
;                                                                      ;
;                                                                      ;
;                                                                      ;
;       Title           String compare                                 ;
;       Name:           STRCMP                                         ;
;                                                                      ;
;                                                                      ;


;                                                                      ;
;       Purpose:        Compare 2 strings and return C and Z flags set ;
;                       or cleared                                     ;
;                                                                      ;
;       Entry:          Register pair HL = Base address of string 1    ;
;                       Register pair DE = Base address of string 2    ;
;                                                                      ;
;                         A string is a maximum of 255 bytes long plus ;
;                         a length byte which precedes it.             ;
;                                                                      ;
;       Exit:           IF string 1 = string 2 THEN               ;
;                         Z=1,C=0                                  ;
;                       IF string 1 > string 2 THEN               ;
;                         Z=0,C=0                                  ;
;                       IF string 1 < string 2 THEN               ;
;                         Z=0,C=1                                  ;
;                                                                  ;
;       Registers used: AF,BC,DE,HL                                ;
;                                                                  ;
;       Time:           91 cycles overhead plus 60 cycles per byte plus ;
;                       40 cycles if strings are identical         ;
;                       through length of shorter                  ;
;                                                                  ;
;       Size:           Program 32 bytes                           ;
;                       Data     2 bytes                           ;
;                                                                  ;
;                                                                  ;

STRCMP:
        ;DETERMINE WHICH STRING IS SHORTER
        ;LENGTH OF SHORTER = NUMBER OF BYTES TO COMPARE
        LD      A,(HL)          ;SAVE LENGTH OF STRING 1
        LD      (LENS1),A
        LD      A,(DE)          ;SAVE LENGTH OF STRING 2
        LD      (LENS2),A
        CP      (HL)            ;COMPARE TO LENGTH OF STRING 1
        JR      C,BEGCMP        ;JUMP IF STRING 2 IS SHORTER
        LD      A,(HL)          ;ELSE STRING 1 IS SHORTER

        ;COMPARE STRINGS THROUGH LENGTH OF SHORTER
BEGCMP:
        OR      A               ;TEST LENGTH OF SHORTER STRING
        JR      Z,CMPLEN        ;COMPARE LENGTHS
                                ; IF LENGTH IS ZERO
        LD      B,A             ;B = NUMBER OF BYTES TO COMPARE
        EX      DE,HL           ;DE = STRING 1
                                ;HL = STRING 2
CMPLP:
        INC     HL              ;INCREMENT TO NEXT BYTES
        INC     DE
        LD      A,(DE)          ;GET A BYTE OF STRING 1
        CP      (HL)            ;COMPARE TO BYTE OF STRING 2
        RET     NZ              ;RETURN WITH FLAGS SET IF BYTES
                                ; NOT EQUAL
        DJNZ    CMPLP           ;CONTINUE THROUGH ALL BYTES

        ;STRINGS SAME THROUGH LENGTH OF SHORTER
        ;SO USE LENGTHS TO SET FLAGS
CMPLEN: LD      A,(LENS1)       ;COMPARE LENGTHS
        LD      HL,LENS2
        CP      (HL)
        RET                     ;RETURN WITH FLAGS SET OR CLEARED

        ;DATA
LENS1:  DS      1               ;LENGTH OF STRING 1
LENS2:  DS      1               ;LENGTH OF STRING 2
```

```
;                                                                       ;
;                                                                       ;
;       SAMPLE EXECUTION:                                               ;
;                                                                       ;
;                                                                       ;

SC8A:
        LD      HL,S1           ;BASE ADDRESS OF STRING 1
        LD      DE,S2           ;BASE ADDRESS OF STRING 2
        CALL    STRCMP          ;COMPARE STRINGS
                                ;COMPARING "STRING 1" AND "STRING 2"
                                ; RESULTS IN STRING 1 LESS THAN
                                ; STRING 2, SO Z=0,C=1

        JR      SC8A            ;LOOP FOR ANOTHER TEST

S1:     DB      20H,'STRING 1                    '
S2:     DB      20H,'STRING 2                    '

        END
```

String Concatenation (CONCAT) 8B

Combines (concatenates) two strings, placing the second immediately after the first in memory. If the concatenation produces a string longer than a specified maximum, the program concatenates only enough of string 2 to give the combined string its maximum length. The Carry flag is cleared if all of string 2 can be concatenated or set to 1 if part of string 2 must be dropped. Both strings are a maximum of 255 bytes long and the actual characters are preceded by a byte containing the length.

Procedure: The program uses the length of string 1 to determine where to start adding characters and the length of string 2 to determine how many characters to add. If the sum of the lengths exceeds the maximum, the program indicates an overflow and reduces the number of characters it must add (the number is the maximum length minus the length of string 1). It then moves the appropriate number of characters from string 2 to the end of string 1, updates the length of string 1, and sets the Carry flag to indicate whether any characters were discarded.

Registers Used: AF, BC, DE, HL

Execution Time: Approximately 21 * NUMBER OF CHARACTERS CONCATENATED plus 288 cycles overhead. NUMBER OF CHARACTERS CONCATENATED is normally the length of string 2, but it will be the maximum length of string 1 minus its current length if the combined string would be too long. If, for example, NUMBER OF CHARACTERS CONCATENATED is 14_{16} (20_{10}), the execution time is

$$21 * 20 + 288 = 420 + 288 = 708 \text{ cycles}$$

Program Size: 83 bytes

Data Memory Required: Five bytes anywhere in RAM for the base address of string 1 (2 bytes starting at address S1ADR), the lengths of the strings (addresses S1LEN and S2LEN), and a flag that indicates whether the combined strings overflowed (address STRGOV).

Special Cases:

1. If concatenating would make the string longer than its specified maximum length, the program concatenates only enough of string 2 to reach the maximum. If any of string 2 must be truncated, the Carry flag is set to 1.
2. If string 2 has a length of 0, the program exits with the Carry flag cleared (no errors) and string 1 unchanged. That is, a length of 0 for either string is interpreted as 0, not as 256.
3. If the original length of string 1 exceeds the specified maximum, the program exits with the Carry flag set to 1 (indicating an error) and string 1 unchanged.

Entry Conditions

Base address of string 2 in DE
Base address of string 1 in HL
Maximum length of string 1 in B

Exit Conditions

String 2 concatenated at the end of string 1 and the length of string 1 increased appropriately. If the resulting string would exceed the maximum length, only the part of string 2 that would give string 1 its maximum length is concatenated. If any part of string 2 must be dropped, the Carry flag is set to 1. Otherwise, the Carry flag is cleared.

Examples

1. Data: Maximum length of string 1 = $0E_{16} = 14_{10}$
   String 1 = 07'JOHNSON' (07 is the length of the string)
   String 2 = 05',DON' (05 is the length of the string)

   Result: String 1 = 0C'JOHNSON, DON' ($0C_{16} = 12_{10}$ is the length of the combined string with string 2 placed after string 1)
   Carry = 0, since the concatenation did not produce a string exceeding the maximum length.

Note that we are representing the initial byte (containing the length of the string) as two hexadecimal digits in both examples.

2. Data: String 1 = 07'JOHNSON' (07 is the length of the string)
   String 2 = 09',RICHARD' (09 is the length of the string)

   Result: String 1 = 0E'JOHNSON, RICHA' ($0E_{16} = 14_{10}$ is the maximum length allowed, so the last two characters of string 2 have been dropped)
   Carry = 1, since the concatenation produced a string longer than the maximum length.

```
;                                                             ;
;                                                             ;
;                                                             ;
;                                                             ;
;       Title           String Concatenation                  ;
;       Name:           CONCAT                                ;
;                                                             ;
;                                                             ;

;                                                             ;
;       Purpose:        Concatenate 2 strings into one string ;
;                                                             ;
;       Entry:          Register pair HL = Base address of string 1 ;
;                       Register pair DE = Base address of string 2 ;
;                       Register B = Maximum length of string 1     ;
;                                                             ;
;                         A string is a maximum of 255 bytes long plus ;
;                         a length byte which precedes it.    ;
;                                                             ;
;       Exit:           String 1 := string 1 concatenated with string 2 ;
;                       If no errors then                     ;
;                         CARRY := 0                          ;
;                       else                                  ;
;                         begin                               ;
;                           CARRY := 1                        ;
;                           if the concatenation makes string 1 too ;
;                           long, concatenate only enough of string 2 ;
;                           to give string 1 its maximum length. ;
;                           if length(string1) > maximum length then ;
;                             no concatenation is done        ;
;                         end;                                ;
;                                                             ;
;       Registers used: AF,BC,DE,HL                                        ;
;                                                                          ;
;       Time:           Approximately 21 * (length of string 2) cycles    ;
;                       plus 288 cycles overhead                           ;
;                                                                          ;
;       Size:           Program 83 bytes                                   ;
;                       Data     5 bytes                                   ;
;                                                                          ;
;                                                                          ;

CONCAT:
        ;DETERMINE WHERE TO START CONCATENATING
        ;CONCATENATION STARTS AT THE END OF STRING 1
        ;END OF STRING 1 = BASE1 + LENGTH1 + 1, WHERE
        ;  THE EXTRA 1 MAKES UP FOR THE LENGTH BYTE
        ;NEW CHARACTERS COME FROM STRING 2, STARTING AT
        ;  BASE2 + 1 (SKIPPING OVER LENGTH BYTE)
        LD      (S1ADR),HL      ;SAVE ADDRESS OF STRING 1
        PUSH    BC              ;SAVE MAXIMUM LENGTH OF STRING 1
        LD      A,(HL)          ;SAVE LENGTH OF STRING 1
        LD      (S1LEN),A
        LD      C,A             ;END1 = BASE1 + LENGTH1 + 1
        LD      B,0
        ADD     HL,BC
        INC     HL              ;HL = START OF CONCATENATION
        LD      A,(DE)          ;SAVE LENGTH OF STRING 2
        LD      (S2LEN),A
        INC     DE              ;DE = FIRST CHARACTER OF STRING 2
        POP     BC              ;RESTORE MAXIMUM LENGTH

        ;DETERMINE HOW MANY CHARACTERS TO CONCATENATE
        LD      C,A             ;ADD LENGTHS OF STRINGS
        LD      A,(S1LEN)
        ADD     A,C
        JR      C,TOOLNG        ;JUMP IF SUM EXCEEDS 255
        CP      B               ;COMPARE TO MAXIMUM LENGTH
        JR      Z,LENOK         ;JUMP IF NEW STRING IS MAX LENGTH
        JR      C,LENOK         ;  OR LESS

        ;COMBINED STRING IS TOO LONG
        ; INDICATE A STRING OVERFLOW, STRGOV := 0FFH
        ; NUMBER OF CHARACTERS TO CONCATENATE =  MAXLEN - S1LEN
        ; LENGTH OF STRING 1 = MAXIMUM LENGTH

TOOLNG:
        LD      A,0FFH          ;INDICATE STRING OVERFLOW
        LD      (STRGOV),A
        LD      A,(S1LEN)       ;CALCULATE MAXLEN - S1LEN
        LD      C,A
        LD      A,B
        SUB     C
        RET     C               ;EXIT IF ORIGINAL STRING TOO LONG
        LD      (S2LEN),A       ;CHANGE S2LEN TO MAXLEN - S1LEN
        LD      A,B             ;LENGTH OF STRING 1 = MAXIMUM
        LD      (S1LEN),A
        JR      DOCAT           ;PERFORM CONCATENATION
        ;RESULTING LENGTH DOES NOT EXCEED MAXIMUM
        ; LENGTH OF STRING 1 = S1LEN + S2LEN
        ; INDICATE NO OVERFLOW, STRGOV := 0
        ; NUMBER OF CHARACTERS TO CONCATENATE = LENGTH OF STRING 2
LENOK:
        LD      (S1LEN),A       ;SAVE SUM OF LENGTHS
        SUB     A               ;INDICATE NO OVERFLOW
        LD      (STRGOV),A

        ;CONCATENATE STRINGS BY MOVING CHARACTERS FROM STRING 2
        ; TO END OF STRING 1
DOCAT:
        LD      A,(S2LEN)       ;GET NUMBER OF CHARACTERS
        OR      A
        JR      Z,EXIT          ;EXIT IF NOTHING TO CONCATENATE
        LD      C,A             ;BC = NUMBER OF CHARACTERS
        LD      B,0
        EX      DE,HL           ;DE = DESTINATION
                                ;HL = SOURCE
        LDIR                    ;MOVE CHARACTERS

EXIT:
        LD      A,(S1LEN)       ;ESTABLISH NEW LENGTH OF STRING 1
        LD      HL,(S1ADR)
        LD      (HL),A
        LD      A,(STRGOV)      ;CARRY = 1 IF OVERFLOW, 0 IF NOT
        RRA
        RET

        ;DATA
S1ADR:  DS      2               ;BASE ADDRESS OF STRING 1
S1LEN:  DS      1               ;LENGTH OF STRING 1
S2LEN:  DS      1               ;LENGTH OF STRING 2
STRGOV: DS      1               ;STRING OVERFLOW FLAG



;                                                                       ;
;                                                                       ;
;       SAMPLE EXECUTION:                                               ;
;                                                                       ;
;                                                                       ;


SC8B:
        LD      HL,S1           ;HL = BASE ADDRESS OF S1
        LD      DE,S2           ;DE = BASE ADDRESS OF S2
        LD      B,20H           ;B = MAXIMUM LENGTH OF STRING 1
        CALL    CONCAT          ;CONCATENATE STRINGS

        JR      SC8B            ;RESULT OF CONCATENATING
                                ; "LASTNAME" AND ", FIRSTNAME"
                                ; IS S1 = 13H,"LASTNAME, FIRSTNAME"

        ;TEST DATA, CHANGE FOR OTHER VALUES
S1:     DB      8H              ;LENGTH OF S1
        DB      'LASTNAME                        ' ;32 BYTE MAX LENGTH
```

```
S2:     DB      0BH                 ;LENGTH OF S2
        DB      ', FIRSTNAME                          ' ;32 BYTE MAX LENGTH

        END
```

Find the Position of a Substring (POS)

Searches for the first occurrence of a substring within a string. Returns the index at which the substring starts if it is found and 0 if it is not found. The string and the substring are both a maximum of 255 bytes long, and the actual characters are preceded by a byte containing the length. Thus, if the substring is found, its starting index cannot be less than 1 or more than 255.

Procedure: The program searches the string for the substring until either it finds the substring or the remaining part of the string is shorter than the substring and hence cannot possibly contain it. If the substring is not in the string, the program clears the accumulator; otherwise, the program places the starting index of the substring in the accumulator.

Registers Used: AF, BC, DE, HL

Execution Time: Data-dependent, but the overhead is 157 cycles, each successful match of 1 character takes 56 cycles, and each unsuccessful match of 1 character takes 148 cycles. The worst case is when the string and substring always match except for the last character in the substring, such as

String = 'AAAAAAAAB'
Substring = 'AAB'

The execution time in that case is

(STRING LENGTH − SUBSTRING LENGTH + 1) * (56 * (SUBSTRING LENGTH − 1) + 148) + 154

If, for example, STRING LENGTH = 9 and SUBSTRING LENGTH = 3 (as in the case shown), the execution time is

(9 − 3 + 1) * (56 * (3 − 1) + 148) + 154 = 7 * 260 + 154 = 1820 + 154 = 1974 cycles

Program Size: 69 bytes

Data Memory Required: Seven bytes anywhere in RAM for the base address of the string (2 bytes starting at address STRING), the base address of the substring (2 bytes starting at address SUBSTG), the length of the string (address SLEN), the length of the substring (address SUBLEN), and the current starting index in the string (address INDEX).

Special Cases:

1. If either the string or the substring has a length of 0, the program exits with 0 in the accumulator, indicating that it did not find the substring.

2. If the substring is longer than the string, the program exits with 0 in the accumulator, indicating that it did not find the substring.

3. If the program returns an index of 1, the substring may be regarded as an abbreviation of the string. That is, the substring occurs in the string, starting at the first character. A typical example would be a string PRINT and a substring PR.

4. If the substring occurs more than once in the string, the program will return only the index to the first occurrence (the occurrence with the lowest starting index).

Entry Conditions

Base address of substring in DE
Base address of string in HL

Exit Conditions

A contains index at which first occurrence of substring starts if it is found and contains 0 if substring is not found.

Examples

1. Data: String = 1D'ENTER SPEED IN MILES PER HOUR' ($1D_{16} = 29_{10}$ is the length of the string)
   Substring = 05'MILES' (05 is the length of the substring)
   Result: A contains 10_{16} (16_{10}), the index at which the substring 'MILES' starts.

2. Data: String = 1B'SALES FIGURES FOR JUNE 1981' ($1B_{16} = 27_{10}$ is the length of the string)
   Substring = 04'JUNE' (04 is the length of the substring)
   Result: A contains 13_{16} (19_{10}), the index at which the substring 'JUNE' starts.

3. Data: String = 10'LET Y1 = X1 + R7' ($10_{16} = 16_{10}$ is the length of the string)
   Substring = 02'R4' (02 is the length of the substring)
   Result: A contains 0, since the substring 'R4' does not appear in the string LET Y1 = X1 + R7.

4. Data: String = 07'RESTORE' (07 is the length of the string)
   Substring = 03'RES' (03 is the length of the substring)
   Result: A contains 1, the index at which the substring 'RES' starts. An index of 1 indicates that the substring could be an abbreviation of the string. Interactive programs, such as BASIC intepreters and word processors, often use such abbreviations to save on typing and storage.

```
;                                                                       ;
;                                                                       ;
;                                                                       ;
;                                                                       ;
;       Title           Find the position of a substring in a string    ;
;       Name:           POS                                             ;
;                                                                       ;
;                                                                       ;

;                                                                       ;
;       Purpose:        Search for the first occurrence of a substring  ;
;                       within a string and return its starting index.  ;
;                       If the substring is not found a 0 is returned.  ;
;                                                                       ;
;       Entry:          Register pair HL = Base address of string       ;
;                       Register pair DE = Base address of substring    ;
;                                                                       ;
;                         A string is a maximum of 255 bytes long plus  ;
;                         a length byte which precedes it.              ;
;                                                                       ;
;       Exit:           If the substring is found then                  ;
;                         Register A = its starting index               ;
;                       else                                            ;
;                         Register A = 0                                ;
;                                                                       ;
;       Registers used: AF,BC,DE,HL                                     ;
;                                                                       ;
;                                                                       ;
;       Time:           Since the algorithm is so data-dependent,       ;
;                       a simple formula is impossible; but the          ;
;                       following statements are true, and a             ;
;                       worst case is given.                             ;
;                                                                        ;
;                       154 cycles overhead                              ;
;                       Each match of 1 character takes 56 cycles        ;
;                       A mismatch takes 148 cycles                      ;
;                                                                        ;
;                       Worst case timing will be when the               ;
;                       string and substring always match                ;
;                       except for the last character of the             ;
;                       substring, such as                               ;
;                           string = 'AAAAAAAAAB'                        ;
;                           substring = 'AAB'                            ;
;                                                                        ;
;       Size:           Program 69 bytes                                 ;
;                       Data     7 bytes                                 ;
;                                                                        ;
;                                                                        ;

POS:
        ;SET UP TEMPORARIES
        ;EXIT IF STRING OR SUBSTRING HAS ZERO LENGTH
        LD      (STRING),HL     ;SAVE STRING ADDRESS
        EX      DE,HL
        LD      A,(HL)          ;TEST LENGTH OF SUBSTRING
        OR      A
        JR      Z,NOTFND        ;EXIT IF LENGTH OF SUBSTRING = 0
        INC     HL              ;MOVE PAST LENGTH BYTE OF SUBSTRING
        LD      (SUBSTG),HL     ;SAVE SUBSTRING ADDRESS
        LD      (SUBLEN),A
        LD      C,A             ;C = SUBSTRING LENGTH
        LD      A,(DE)          ;TEST LENGTH OF STRING
        OR      A
        JR      Z,NOTFND        ;EXIT IF LENGTH OF STRING = 0

        ;NUMBER OF SEARCHES = STRING LENGTH - SUBSTRING LENGTH
        ; + 1. AFTER THAT, NO USE SEARCHING SINCE THERE AREN'T
        ; ENOUGH CHARACTERS LEFT TO HOLD SUBSTRING
        ;
        ;IF SUBSTRING IS LONGER THAN STRING, EXIT IMMEDIATELY AND
        ; INDICATE SUBSTRING NOT FOUND
        SUB     C               ;A = STRING LENGTH - SUBSTRING LENGTH
        JR      C,NOTFND        ;EXIT IF STRING SHORTER THAN SUBSTRING
        INC     A               ;COUNT = DIFFERENCE IN LENGTHS + 1
        LD      B,A
        SUB     A               ;INITIAL STARTING INDEX = 0
        LD      (INDEX),A

        ;SEARCH UNTIL REMAINING STRING SHORTER THAN SUBSTRING
SLP1:
        LD      HL,INDEX        ;INCREMENT STARTING INDEX
        INC     (HL)
        LD      HL,SUBLEN       ;C = LENGTH OF SUBSTRING
        LD      C,(HL)
        LD      HL,(STRING)     ;INCREMENT TO NEXT BYTE OF STRING
        INC     HL
        LD      (STRING),HL     ;HL = NEXT ADDRESS IN STRING
        LD      DE,(SUBSTG)     ;DE = STARTING ADDRESS OF SUBSTRING
                                ;C = CURRENT VALUE OF COUNT

        ;TRY TO MATCH SUBSTRING STARTING AT INDEX
        ;MATCH INVOLVES COMPARING CORRESPONDING CHARACTERS
        ; ONE AT A TIME
CMPLP:
        LD      A,(DE)          ;GET A CHARACTER OF SUBSTRING
        CP      (HL)            ;COMPARE TO CHARACTER OF STRING
        JR      NZ,SLP2         ;JUMP IF NOT SAME
        DEC     C
        JR      Z,FOUND         ;JUMP IF SUBSTRING FOUND
        INC     HL              ;PROCEED TO NEXT CHARACTERS
        INC     DE
        JR      CMPLP

        ;ARRIVE HERE IF MATCH FAILS, SUBSTRING NOT YET FOUND
SLP2:
        DJNZ    SLP1            ;TRY NEXT HIGHER INDEX IF
                                ; ENOUGH STRING LEFT
        JR      NOTFND          ;ELSE EXIT NOT FOUND

        ;FOUND SUBSTRING, RETURN ITS STARTING INDEX
FOUND:
        LD      A,(INDEX)       ;SUBSTRING FOUND, A = STARTING INDEX
        RET

        ;COULD NOT FIND SUBSTRING, RETURN 0 AS INDEX
NOTFND:
        SUB     A               ;SUBSTRING NOT FOUND, A = 0
        RET

        ;DATA
STRING: DS      2               ;BASE ADDRESS OF STRING
SUBSTG: DS      2               ;BASE ADDRESS OF SUBSTRING
SLEN:   DS      1               ;LENGTH OF STRING
SUBLEN: DS      1               ;LENGTH OF SUBSTRING
INDEX:  DS      1               ;CURRENT INDEX INTO STRING


;                                                                       ;
;                                                                       ;
;       SAMPLE EXECUTION:                                               ;
;                                                                       ;
;                                                                       ;


SC8C:
        LD      HL,STG          ;HL = BASE ADDRESS OF STRING
        LD      DE,SSTG         ;DE = BASE ADDRESS OF SUBSTRING
        CALL    POS             ;FIND POSITION OF SUBSTRING
                                ; SEARCHING "AAAAAAAAAB" FOR "AAB"
                                ; RESULTS IN REGISTER A = 8
        JR      SC8C            ;LOOP FOR ANOTHER TEST

        ;TEST DATA, CHANGE FOR OTHER VALUES
STG:    DB      0AH             ;LENGTH OF STRING
        DB      'AAAAAAAAAB                     ' ;32 BYTE MAX LENGTH
SSTG:   DB      3H              ;LENGTH OF SUBSTRING
        DB      'AAB                            ' ;32 BYTE MAX LENGTH

        END
```

Copy a Substring from a String (COPY) 8D

Copies a substring from a string, given a starting index and the number of bytes to copy. The strings are a maximum of 255 bytes long, and the actual characters are preceded by a byte containing the length. If the starting index of the substring is 0 (that is, the substring would start in the length byte) or is beyond the end of the string, the substring is given a length of 0 and the Carry flag is set to 1. If the substring would exceed its maximum length or would extend beyond the end of the string, then only the maximum number or the available number of characters (up to the end of the string) is placed in the substring, and the Carry flag is set to 1. If the substring can be formed as specified, the Carry flag is cleared.

Procedure: The program exits immediately if the number of bytes to copy, the maximum length of the substring, or the starting index is 0. It also exits immediately if the starting index exceeds the length of the string. If none of these conditions holds, the program checks if the number of bytes to copy exceeds either the maximum length of the substring or the number of characters available in the string. If either is exceeded, the program reduces the number of bytes to copy appropriately. It then copies the proper number of bytes from the string to the substring. The program clears the Carry flag if the substring can be formed as specified and sets the Carry flag if it cannot.

Registers Used: AF, BC, DE, HL

Execution Time: Approximately 21 * NUMBER OF BYTES COPIED plus 237 cycles overhead. NUMBER OF BYTES COPIED is the number specified if no problems occur, or the number available, or the maximum length of the substring if copying would extend beyond either the string or the substring. If, for example, NUMBER OF BYTES COPIED = 12_{10} ($0C_{16}$), the execution time is

$21 * 12 + 237 = 252 + 237 = 489$ cycles

Program Size: 73 bytes

Data Memory Required: Two bytes anywhere in RAM for the maximum length of the substring (address MAXLEN) and an error flag (address CPYERR)

Special Cases:

1. If the number of bytes to copy is 0, the program assigns the substring a length of 0 and clears the Carry flag, indicating no errors.
2. If the maximum length of the substring is 0, the program assigns the substring a length of 0 and sets the Carry flag to 1, indicating an error.
3. If the starting index of the substring is 0, the program assigns the substring a length of 0 and sets the Carry flag to 1, indicating an error.
4. If the source string does not even reach the specified starting index, the program assigns the substring a length of 0 and sets the Carry flag to 1, indicating an error.
5. If the substring would extend beyond the end of the source string, the program places all the available characters in the substring and sets the Carry flag to 1, indicating an error. The available characters are the ones from the starting index to the end of the string.
6. If the substring would exceed its specified maximum length, the program places only the specified maximum number of characters in the substring. It sets the Carry flag to 1, indicating an error.

Entry Conditions

Base address of substring in DE
Base address of string in HL
Number of bytes to copy in B
Starting index to copy from in C
Maximum length of substring in A

Exit Conditions

Substring contains characters copied from string. If the starting index is 0, the maximum length of the substring is 0, or the starting index is beyond the length of the string, the substring will have a length of 0 and the Carry flag will be set to 1. If the substring would extend beyond the end of the string or would exceed its specified maximum length, only the available characters from the string (up to the maximum length of the substring) are copied into the substring; the Carry flag is set in this case also. If no problems occur in forming the substring, the Carry flag is cleared.

Examples

1. Data: String = 10'LET Y1 = R7 + X4'
   ($10_{16} = 16_{10}$ is the length of the string)
   Maximum length of substring = 2
   Number of bytes to copy = 2
   Starting index = 5

   Result: Substring = 02'Y1' (2 is the length of the substring)
   Two bytes from the string were copied, starting at character #5 (that is, characters 5 and 6)
   Carry = 0, since no problems occurred in forming the substring.

2. Data: String = 0E'8657 POWELL ST'
   ($0E_{16} = 14_{10}$ is the length of the string)
   Maximum length of substring = $10_{16} = 16_{10}$
   Number of bytes to copy = $0D_{16} = 13_{10}$
   Starting index = 6

   Result: Substring = 09'POWELL ST' (09 is the length of the substring)
   Carry = 1, since there were not enough characters available in the string to provide the specified number of bytes to copy.

3. Data: String = 16'9414 HEGENBERGER DRIVE'
   ($16_{16} = 22_{10}$ is the length of the string)
   Maximum length of substring = $10_{16} = 16_{10}$
   Number of bytes to copy = $11_{16} = 17_{10}$
   Starting index = 6

   Result: Substring = 10'HEGENBERGER DRIV'
   ($10_{16} = 16_{10}$ is the length of the substring)
   Carry = 1, since the number of bytes to copy exceeded the maximum length of the substring.

```
;                                                                  ;
;                                                                  ;
;                                                                  ;
;                                                                  ;
;       Title           Copy a substring from a string             ;
;       Name:           Copy                                       ;
;                                                                  ;
;                                                                  ;
;                                                                       ;
;       Purpose:        Copy a substring from a string given a starting ;
;                       index and the number of bytes                   ;
;                                                                       ;
;       Entry:          Register pair HL = Address of source string     ;
;                       Register pair DE = Address of destination string;
;                       Register A = Maximum length of destination      ;
;                                    string                             ;
;                       Register B = Number of bytes to copy            ;
;                       Register C = Starting index into source string  ;
;                                    Index of 1 is first character of   ;
;                                    string                             ;
;                                                                       ;
;                         A string is a maximum of 255 bytes long plus  ;
;                         a length byte which precedes it.              ;
;                                                                       ;
;       Exit:           Destination string := The substring from the    ;
;                       string.                                         ;
;                       if no errors then                               ;
;                         CARRY := 0                                    ;
;                       else                                            ;
;                         begin                                         ;
;                           the following conditions cause an           ;
;                           error and the CARRY flag = 1.               ;
;                           if (index = 0) or (maxlen = 0) or           ;
;                               (index > length(source)) then           ;
;                             the destination string will have a zero   ;
;                             length.                                   ;
;                           if (index + count - 1) > length(source)     ;
;                           then                                        ;
;                             the destination string becomes everything ;
;                             from index to the end of source string.   ;
;                         END;                                          ;
;                                                                       ;
;       Registers used: AF,BC,DE,HL                                     ;
;                                                                       ;
;                                                                       ;
;       Time:           Approximately (21 * count) cycles plus 237      ;
;                       cycles overhead.                                ;
;                                                                       ;
;       Size:           Program 73 bytes                                ;
;                       Data     2 bytes                                ;
;                                                                       ;
;                                                                       ;

COPY:
        ;SAVE MAXIMUM LENGTH OF DESTINATION STRING
        LD      (MAXLEN),A      ;SAVE MAXIMUM LENGTH

        ;INITIALIZE LENGTH OF DESTINATION STRING AND ERROR FLAG
        SUB     A
        LD      (DE),A          ;LENGTH OF DESTINATION STRING = ZERO
        LD      (CPYERR),A      ;ASSUME NO ERRORS

        ;IF NUMBER OF BYTES TO COPY IS 0, EXIT WITH NO ERRORS
        OR      B               ;TEST NUMBER OF BYTES TO COPY
        RET     Z                   ;EXIT WITH NO ERRORS
                                    ; CARRY = 0

        ;IF MAXIMUM LENGTH IS 0, TAKE ERROR EXIT
        LD      A,(MAXLEN)          ;TEST MAXIMUM LENGTH
        OR      A
        JR      Z,EREXIT            ;ERROR EXIT IF MAX LENGTH IS 0

        ;IF STARTING INDEX IS ZERO, TAKE ERROR EXIT
        LD      A,C                 ;TEST STARTING INDEX
        OR      A
        JR      Z,EREXIT            ;ERROR EXIT IF INDEX IS 0

        ;IF STARTING INDEX IS GREATER THAN LENGTH OF SOURCE
        ; STRING, TAKE ERROR EXIT
        LD      A,(HL)              ;GET LENGTH OF SOURCE STRING
        CP      C                   ;COMPARE TO STARTING INDEX
        RET     C                   ;ERROR EXIT IF LENGTH LESS THAN INDEX
                                    ; CARRY = 1

        ;CHECK IF COPY AREA FITS IN SOURCE STRING
        ;OTHERWISE, COPY ONLY TO END OF STRING
        ;COPY AREA FITS IF STARTING INDEX + NUMBER OF
        ; CHARACTERS TO COPY - 1 IS LESS THAN OR EQUAL TO
        ; LENGTH OF SOURCE STRING
        ;NOTE THAT STRINGS ARE NEVER MORE THAN 255 BYTES LONG
        LD      A,C                 ;FORM STARTING INDEX + COPY LENGTH
        ADD     A,B
        JR      C,RECALC            ;JUMP IF SUM > 255
        DEC     A
        CP      (HL)
        JR      C,CNT1OK            ;JUMP IF MORE THAN ENOUGH TO COPY
        JR      Z,CNT1OK            ;JUMP IF EXACTLY ENOUGH

        ;CALLER ASKED FOR TOO MANY CHARACTERS. RETURN EVERYTHING
        ; BETWEEN INDEX AND END OF SOURCE STRING.
        ; SET COUNT := LENGTH(SOURCE) - INDEX + 1;
RECALC:
        LD      A,0FFH              ;INDICATE TRUNCATION OF COUNT
        LD      (CPYERR),A
        LD      A,(HL)              ;COUNT = LENGTH - INDEX + 1
        SUB     C
        INC     A
        LD      B,A                 ;CHANGE NUMBER OF BYTES

        ;CHECK IF COUNT LESS THAN OR EQUAL TO MAXIMUM LENGTH OF
        ; DESTINATION STRING. IF NOT, SET COUNT TO MAXIMUM LENGTH
        ;  IF COUNT > MAXLEN THEN
        ;    COUNT := MAXLEN
CNT1OK:
        LD      A,(MAXLEN)          ;IS MAX LENGTH LARGE ENOUGH?
        CP      B
        JR      NC,CNT2OK           ;JUMP IF IT IS
        LD      B,A                 ;ELSE LIMIT COPY TO MAXLEN
        LD      A,0FFH              ;INDICATE STRING OVERFLOW
        LD      (CPYERR),A

        ;MOVE SUBSTRING TO DESTINATION STRING
CNT2OK:
        LD      A,B                 ;TEST NUMBER OF BYTES TO COPY
        OR      A
        JR      Z,EREXIT            ;ERROR EXIT IF NO BYTES TO COPY
        LD      B,0                 ;START COPYING AT STARTING INDEX
        ADD     HL,BC
        LD      (DE),A              ;SET LENGTH OF DESTINATION STRING
        LD      C,A                 ;RESTORE NUMBER OF BYTES
        INC     DE                  ;MOVE DESTINATION ADDRESS PAST
                                    ; LENGTH BYTE
        LDIR                        ;COPY SUBSTRING

        ;CHECK FOR COPY ERROR
        LD      A,(CPYERR)          ;TEST FOR ERRORS
OKEXIT:
        OR      A
        RET     Z                   ;RETURN WITH C = 0 IF NO ERRORS

        ;ERROR EXIT
EREXIT:
        SCF                         ;SET CARRY TO INDICATE AN ERROR
        RET

        ;DATA SECTION
MAXLEN: DS      1                   ;MAXIMUM LENGTH OF DESTINATION STRING
CPYERR: DS      1                   ;COPY ERROR FLAG



;                                                                       ;
;                                                                       ;
;       SAMPLE EXECUTION:                                               ;
;                                                                       ;
;                                                                       ;


SC8D:
        LD      HL,SSTG             ;SOURCE STRING
        LD      DE,DSTG             ;DESTINATION STRING
        LD      A,(IDX)
        LD      C,A                 ;STARTING INDEX FOR COPYING
        LD      A,(CNT)
        LD      B,A                 ;NUMBER OF BYTES TO COPY
        LD      A,(MXLEN)           ;MAXIMUM LENGTH OF SUBSTRING
        CALL    COPY                ;COPY SUBSTRING
                                    ;COPYING 3 CHARACTERS STARTING AT
                                    ; INDEX 4 FROM '12.345E+10' GIVES '345'

        JR      SC8D                ;LOOP FOR MORE TESTING

        ;DATA SECTION
IDX:    DB      4                   ;STARTING INDEX FOR COPYING
CNT:    DB      3                   ;NUMBER OF CHARACTERS TO COPY
```

```
MXLEN:  DB      20H                 ;MAXIMUM LENGTH OF DESTINATION STRING
SSTG:   DB      0AH                 ;LENGTH OF STRING
        DB      '12.345E+10                         ' ;32 BYTE MAX LENGTH
DSTG:   DB      0                   ;LENGTH OF SUBSTRING
        DB      '                                   ' ;32 BYTE MAX LENGTH

        END
```

Delete a Substring from a String (DELETE) 8E

Deletes a substring from a string, given a starting index and a length. The string is a maximum of 255 bytes long, and the actual characters are preceded by a byte containing the length. The Carry flag is cleared if the deletion can be performed as specified. The Carry flag is set if the starting index is 0 or beyond the length of the string; the string is left unchanged in either case. If the deletion extends beyond the end of the string, the Carry flag is set to 1 and only the characters from the starting index to the end of the string are deleted.

Procedure: The program exits immediately if either the starting index or the number of bytes to delete is 0. It also exits if the starting index is beyond the length of the string. If none of these conditions holds, the program checks to see if the string extends beyond the area to be deleted. If it does not, the program simply truncates the string by setting the new length to the starting index minus 1. If it does, the program compacts the resulting string by moving the bytes above the deleted area down. The program then determines the new string's length and exits with the Carry cleared if the specified number of bytes were deleted or with the Carry set to 1 if any errors occurred.

Registers Used: AF, BC, DE, HL

Execution Time: Approximately 21 * NUMBER OF BYTES MOVED DOWN + 224 cycles, where NUMBER OF BYTES MOVED DOWN is zero if the string can be truncated and is STRING LENGTH − STARTING INDEX − NUMBER OF BYTES TO DELETE + 1 if the string must be compacted. That is, it takes extra time when the deletion creates a "hole" in the string that must be filled by compaction.

Examples

1. STRING LENGTH = 20_{16} (32_{10})
   STARTING INDEX = 19_{16} (25_{10})
   NUMBER OF BYTES TO DELETE = 08

Since there are exactly eight bytes left in the string starting at index 19_{16}, all the routine must do is truncate (that is, cut off the end of the string). This takes

$21 * 0 + 224 = 224$ cycles

2. STRING LENGTH = 40_{16} (64_{10})
   STARTING INDEX = 19_{16} (25_{10})
   NUMBER OF BYTES TO DELETE = 08

Since there are 20_{16} (32_{10}) bytes above the truncated area, the routine must move them down eight positions to fill the "hole." Thus NUMBER OF BYTES MOVED DOWN = 32_{10} and the execution time is

$21 * 32 + 224 = 672 + 224 = 896$ cycles

Program Size: 58 bytes

Data Memory Required: One byte anywhere in RAM for an error flag (address DELERR)

Special Cases:

1. If the number of bytes to delete is 0, the program exits with the Carry flag cleared (no errors) and the string unchanged.

2. If the string does not even extend to the specified starting index, the program exits with the Carry flag set to 1 (indicating an error) and the string unchanged.

3. If the number of bytes to delete exceeds the number available, the program deletes all bytes from the starting index to the end of the string and exits with the Carry flag set to 1 (indicating an error).

Entry Conditions

Base address of string in HL
Number of bytes to delete in B
Starting index to delete from in C

Exit Conditions

Substring deleted from string. If no errors occur, the Carry flag is cleared. If the starting index is 0 or beyond the length of the string, the Carry flag is set and the string is unchanged. If the number of bytes to delete would go beyond the end of the string, the Carry flag is set and the characters from the starting index to the end of the string are deleted.

Examples

1. Data: String = 26'SALES FOR MARCH AND APRIL OF THIS YEAR' ($26_{16} = 38_{10}$ is the length of the string)
   Number of bytes to delete = $0A_{16} = 10_{10}$
   Starting index to delete from = $10_{16} = 16_{10}$

   Result: String = 1C 'SALES FOR MARCH OF THIS YEAR' ($1C_{16} = 28_{10}$ is the length of the string with ten bytes deleted starting with the 16th character—the deleted material is 'AND APRIL ')
   Carry = 0, since no problems occurred in the deletion.

2. Data: String = 28'THE PRICE IS $3.00 ($2.00 BEFORE JUNE 1)' ($28_{16} = 40_{10}$ is the length of the string)
   Number of bytes to delete = $30_{16} = 48_{10}$
   Starting index to delete from = $13_{16} = 19_{10}$

   Result: String = 12'THE PRICE IS $3.00' ($12_{16} = 18_{10}$ is the length of the string with all remaining bytes deleted)
   Carry = 1, since there were not as many bytes left in the string as were supposed to be deleted.

```
;                                                                       ;
;                                                                       ;
;                                                                       ;
;                                                                       ;
;       Title:          Delete a substring from a string                ;
;       Name:           Delete                                          ;
;                                                                       ;
;                                                                       ;

;                                                                       ;
;       Purpose:        Delete a substring from a string given a        ;
;                       starting index and a length                     ;
;                                                                       ;
;       Entry:          Register pair HL = Base address of string       ;
;                       Register B = Number of bytes to delete          ;
;                       Register C = Starting index into the string.    ;
;                                 An index of 1 is the first character  ;
;                                                                       ;
;                         A string is a maximum of 255 bytes long plus  ;
;                         a length byte which precedes it.              ;
;                                                                       ;
;       Exit:           Substring deleted.                              ;
;                       if no errors then                               ;
;                         CARRY := 0                                    ;
;                       else                                            ;
;                       begin                                          ;
;                         the following conditions cause an            ;
;                         error with CARRY = 1.                        ;
;                         if (index = 0) or (index > length(string))   ;
;                           then do not change string                  ;
;                         if count is too large then                   ;
;                           delete only the characters from            ;
;                           index to end of string                     ;
;                       end;                                           ;
;                                                                      ;
;       Registers used: AF,BC,DE,HL                                    ;
;                                                                      ;
;       Time:           Approximately 21 * (LENGTH(STRG)-INDEX-COUNT+1) ;
;                       plus 224 cycles overhead                       ;
;                                                                      ;
;       Size:           Program 58 bytes                               ;
;                       Data     1 bytes                               ;
;                                                                      ;
;                                                                      ;

DELETE:
        ;INITIALIZE ERROR INDICATOR (DELERR) TO 0
        SUB     A
        LD      (DELERR),A      ;ASSUME NO ERRORS

        ;CHECK IF COUNT AND INDEX ARE BOTH NON-ZERO
        OR      B               ;TEST NUMBER OF BYTES TO DELETE
        RET     Z               ;RETURN WITH CARRY = 0 (NO ERRORS) IF
                                ; 0 BYTES TO DELETE
        LD      A,C             ;TEST STARTING INDEX
        OR      A
        SCF                     ;CARRY = 1
        RET     Z               ;ERROR EXIT (CARRY = 1) IF
                                ; STARTING INDEX = 0

        ;CHECK IF STARTING INDEX WITHIN STRING
        ; ERROR EXIT IF NOT
        LD      A,(HL)          ;GET LENGTH
        CP      C               ;IS INDEX WITHIN STRING?
        RET     C               ;NO, TAKE ERROR EXIT

        ;BE SURE ENOUGH CHARACTERS ARE AVAILABLE
        ; IF NOT, DELETE ONLY TO END OF STRING
        ; IF INDEX + NUMBER OF CHARACTERS - 1 > LENGTH(STRING) THEN
        ;   NUMBER OF CHARACTERS := LENGTH(STRING) - INDEX + 1
        LD      A,C             ;GET INDEX
        ADD     A,B             ;ADD NUMBER OF CHARACTERS TO DELETE
        JR      C,TRUNC         ;TRUNCATE IF SUM > 255
        LD      E,A             ;SAVE SUM AS STARTING INDEX FOR MOVE
        DEC     A
        CP      (HL)            ;COMPARE TO LENGTH
        JR      C,CNTOK         ;JUMP IF ENOUGH CHARACTERS AVAILABLE
        JR      Z,TRUNC         ;TRUNCATE BUT NO ERRORS (EXACTLY ENOUGH
                                ; CHARACTERS)
        LD      A,0FFH          ;INDICATE ERROR - NOT ENOUGH CHARACTERS
        LD      (DELERR),A      ; AVAILABLE FOR DELETION
        ;TRUNCATE STRING - NO COMPACTING NECESSARY
        ; STRING LENGTH = INDEX - 1
TRUNC:
        LD      A,C             ;STRING LENGTH = INDEX - 1
        DEC     A
        LD      (HL),A
        LD      A,(DELERR)
        RRA                     ;CARRY = 0 IF NO ERRORS
        RET                     ;EXIT

        ;DELETE SUBSTRING BY COMPACTING
        ; MOVE ALL CHARACTERS ABOVE DELETED AREA DOWN
        ;NEW LENGTH = OLD LENGTH - NUMBER OF BYTES TO DELETE
CNTOK:
        LD      A,(HL)
        LD      D,A             ;SAVE OLD LENGTH
        SUB     B               ;SET NEW LENGTH
        LD      (HL),A

        ;CALCULATE NUMBER OF CHARACTERS TO MOVE
        ; NUMBER = STRING LENGTH - (INDEX + NUMBER OF BYTES) + 1
        LD      A,D             ;GET OLD LENGTH
        SUB     E               ;SUBTRACT INDEX + NUMBER OF BYTES
        INC     A               ;A = NUMBER OF CHARACTERS TO MOVE

        ;CALCULATE SOURCE AND DESTINATION ADDRESSES FOR MOVE
        ;SOURCE = BASE + INDEX + NUMBER OF BYTES TO DELETE
        ;DESTINATION = BASE + INDEX
        PUSH    HL              ;SAVE STRING ADDRESS
        LD      B,0             ;DESTINATION = BASE + INDEX
        ADD     HL,BC
        EX      (SP),HL         ;SOURCE = BASE + INDEX + NUMBER
        LD      D,0             ;  OF BYTES TO DELETE
        ADD     HL,DE           ;HL = SOURCE (ABOVE DELETED AREA)
        POP     DE              ;DE = DESTINATION
        LD      C,A             ;BC = NUMBER OF CHARACTERS TO MOVE

        LDIR                    ;COMPACT STRING BY MOVING DOWN

        ;GOOD EXIT
OKEXIT:
        OR      A               ;CLEAR CARRY, NO ERRORS
        RET

        ;DATA
DELERR: DS      1               ;DELETE ERROR FLAG


;                                                                       ;
;                                                                       ;
;       SAMPLE EXECUTION:                                               ;
;                                                                       ;
;                                                                       ;


SC8E:
        LD      HL,SSTG ;HL = BASE ADDRESS OF STRING
        LD      A,(IDX)
        LD      C,A     ;C = STARTING INDEX FOR DELETION
        LD      A,(CNT)
        LD      B,A     ;B = NUMBER OF CHARACTERS TO DELETE
        CALL    DELETE  ;DELETE CHARACTERS
                        ;DELETING 4 CHARACTERS STARTING AT INDEX 1
                        ; FROM "JOE HANDOVER" LEAVES "HANDOVER"

        JR      SC8E    ;LOOP FOR ANOTHER TEST

        ;DATA SECTION
IDX:    DB      1               ;STARTING INDEX FOR DELETION
CNT:    DB      4               ;NUMBER OF CHARACTERS TO DELETE
SSTG:   DB      12              ;LENGTH OF STRING
        DB      'JOE HANDOVER'

        END
```

Insert a Substring into a String (INSERT) 8F

Inserts a substring into a string, given a starting index. The string and substring are both a maximum of 255 bytes long, and the actual characters are preceded by a byte containing the length. The Carry flag is cleared if the insertion can be accomplished with no problems. The Carry flag is set if the starting index is 0 or beyond the length of the string. In the second case, the substring is concatenated to the end of the string. The Carry flag is also set if the string with the insertion will exceed a specified maximum length. In that case, the program inserts only enough of the substring to give the string its maximum length.

Procedure: The program exits immediately if the starting index or the length of the substring is 0. If neither is 0, the program checks to see if the insertion will produce a string longer than the specified maximum length. If this is the case, the program truncates the substring. The program then checks to see if the starting index is within the string. If it is not, the program simply concatenates the substring by moving it to the memory locations immediately after the end of the string. If the starting index is within the string, the program must first make room for the insertion by moving the remaining characters up in memory. This move must start at the highest address to avoid writing over any data. Finally, the program can move the substring into the open area. The program then determines the new string length and exits with the Carry flag set appropriately (to 0 if no problems occurred and to 1 if the starting index was 0, if the substring had to be truncated, or if the starting index was beyond the length of the string).

Registers Used: AF, BC, DE, HL

Execution Time: Approximately 21 * NUMBER OF BYTES MOVED + 21 * NUMBER OF BYTES INSERTED + 290. NUMBER OF BYTES MOVED is the number of bytes that must be moved to create space for the insertion. If the starting index is beyond the end of the string, NUMBER OF BYTES MOVED is 0 since the substring is simply concatenated to the string. Otherwise, it is STRING LENGTH − STARTING INDEX + 1, since the bytes at or above the starting index must be moved. NUMBER OF BYTES INSERTED is the length of the substring if no truncation occurs. It is the maximum length of the string minus its current length if inserting the substring produces a string longer than the maximum.

Examples

1. STRING LENGTH = 20_{16} (32_{10})
   STARTING INDEX = 19_{16} (25_{10})
   MAXIMUM LENGTH = 30_{16} (48_{10})
   SUBSTRING LENGTH = 06

We want to insert a substring six bytes long, starting at the 25th character. Since eight bytes must be moved up (NUMBER OF BYTES MOVED = 32 − 25 + 1) and six bytes must be inserted, the execution time is approximately

$21 * 8 + 21 * 6 + 290 = 168 + 126 + 290 = 584$ cycles

2. STRING LENGTH = 20_{16} (32_{10})
   STARTING INDEX = 19_{16} (25_{10})
   MAXIMUM LENGTH = 24_{16} (36_{10})
   SUBSTRING LENGTH = 06

Unlike Example 1, here we can insert only four bytes of the substring without exceeding the maximum length of the string. Thus, NUMBER OF BYTES MOVED = 8 and NUMBER OF BYTES INSERTED = 4. The execution time is approximately

$21 * 8 + 21 * 4 + 290 = 168 + 84 + 290 = 542$ cycles

Program Size: 90 bytes

Data Memory Required: One byte anywhere in RAM for an error flag (address INSERR).

Special Cases:

1. If the length of the substring (the insertion) is 0, the program exits with the Carry flag cleared (no errors) and the string unchanged.

2. If the starting index for the insertion is 0 (that is, the insertion would start in the length byte), the program exits with the Carry flag set to 1 (indicating an error) and the string unchanged.

3. If the string with the substring inserted exceeds the specified maximum length, the program inserts only enough characters to reach the maximum length. The Carry flag is set to 1 to indicate that the insertion has been truncated.

4. If the starting index of the insertion is beyond the end of the string, the program concatenates the insertion at the end of the string and indicates an error by setting the Carry flag to 1.

5. If the original length of the string exceeds its specified maximum length, the program exits with the Carry flag set to 1 (indicating an error) and the string unchanged.

Entry Conditions

Base address of substring in DE
Base address of string in HL
Maximum length of string in B
Starting index at which to insert the substring in C

Exit Conditions

Substring inserted into string. If no errors occur, the Carry flag is cleared. If the starting index or the length of the substring is 0, the Carry flag is set and the string is not changed. If the starting index is beyond the length of the string, the Carry flag is set and the substring is concatenated to the end of the string. If the string with the substring inserted would exceed the specified maximum length, the Carry flag is set and only those characters from the substring which bring the string to maximum length are inserted.

Examples

1. Data: String = 0A'JOHN SMITH' ($0A_{16} = 10_{10}$ is the length of the string)
Substring = 08'WILLIAM' (08 is the length of the substring)
Maximum length of string = $14_{16} = 20_{10}$
Starting index = 06

Result: String = 12'JOHN WILLIAM SMITH' ($12_{16} = 18_{10}$ is the length of the string with the substring inserted)
Carry = 0, since no problems occurred in the insertion.

2. Data: String = 0A'JOHN SMITH' ($0A_{16} = 10_{10}$ is the length of the string)
Substring = 0C'ROCKEFELLER' ($0C_{16} = 12_{10}$ is the length of the substring)
Maximum length of string = $14_{16} = 20_{10}$
Starting index = 06

Result: String = 14'JOHN ROCKEFELLESMITH' ($14_{16} = 20_{10}$ is the length of the string with as much of the substring inserted as the maximum length would allow)
Carry = 1, since some of the substring could not be inserted without exceeding the maximum length of the string.

```
;                                                                 ;
;                                                                 ;
;                                                                 ;
;                                                                 ;
;       Title:          Insert a substring into a string          ;
;       Name:           Insert                                    ;
;                                                                 ;
;                                                                 ;


;                                                                 ;
;       Purpose:        Insert a substring into a string given a  ;
;                       starting index                            ;
;                                                                 ;
;       Entry:          Register pair HL = Address of string      ;
;                       Register pair DE = Address of substring to;
;                                            insert               ;
;                       Register B = Maximum length of string     ;
;                       Register C = Starting index to insert the ;
;                                    substring                    ;
;                                                                 ;
;                         A string is a maximum of 255 bytes long plus ;
;                         a length byte which precedes it.        ;
;                                                                 ;
;       Exit:           Substring inserted into string.           ;
;                       if no errors then                         ;
;                         CARRY = 0                               ;
;                       else                                      ;
;                         begin                                   ;
;                           the following conditions cause the    ;
;                           CARRY flag to be set.                 ;
;                           if index = 0 then                     ;
;                             do not insert the substring         ;
;                           if length(strg) > maximum length then ;
;                             do not insert the substring         ;
;                           if index > length(strg) then          ;
;                             concatenate substg onto the end of the ;
;                             source string                       ;
;                           if length(strg)+length(substring) > maxlen ;
;                             then insert only enough of the substring ;
;                             to reach maximum length             ;
;                         end;                                    ;
;                                                                 ;
;       Registers used: AF,BC,DE,HL                               ;
;                                                                 ;
;       Time:           Approximately                             ;
;                        21 * (LENGTH(STRG) - INDEX + 1) +        ;
;                        21 * (LENGTH(SUBSTG)) +                  ;
;                        290 cycles overhead                      ;
;                                                                 ;
;       Size:           Program 90 bytes                          ;
;                       Data     1 byte                           ;
;                                                                 ;


INSERT:
        ;INITIALIZE ERROR FLAG
        SUB     A                   ;ERROR FLAG = 0 (NO ERRORS)
        LD      (INSERR),A

        ;GET SUBSTRING AND STRING LENGTHS
        ; IF LENGTH(SUBSTG) = 0 THEN EXIT BUT NO ERROR
        LD      A,(DE)              ;TEST LENGTH OF SUBSTRING
        OR      A
        RET     Z                   ;EXIT IF SUBSTRING EMPTY
                                    ; CARRY = 0 (NO ERRORS)

        ;IF STARTING INDEX IS ZERO, TAKE ERROR EXIT
IDX0:
        LD      A,C                 ;TEST STARTING INDEX
        OR      A
        SCF                         ;ASSUME AN ERROR
        RET     Z                   ;RETURN WITH ERROR IF INDEX = 0

        ;CHECK WHETHER INSERTION WILL MAKE STRING TOO LONG
        ; IF IT WILL, TRUNCATE SUBSTRING AND SET
        ; TRUNCATION FLAG.
        ;INSERTION TOO LONG IF STRING LENGTH + SUBSTRING LENGTH
        ; EXCEEDS MAXIMUM LENGTH.  REMEMBER, STRINGS CANNOT BE
        ; MORE THAN 255 BYTES LONG
CHKLEN:
        LD      A,(DE)              ;TOTAL = STRING + SUBSTRING
        ADD     A,(HL)
        JR      C,TRUNC             ;TRUNCATE SUBSTRING IF NEW LENGTH > 255
        CP      B                   ;COMPARE TO MAXIMUM LENGTH OF STRING
        LD      A,(DE)              ;A = LENGTH OF SUBSTRING
        JR      C,IDXLEN            ;JUMP IF TOTAL < MAX LENGTH
        JR      Z,IDXLEN            ; OR EQUAL

        ;SUBSTRING DOES NOT FIT, SO TRUNCATE IT
        ; SET ERROR FLAG TO INDICATE TRUNCATION
        ; LENGTH THAT FITS = MAXIMUM LENGTH - STRING LENGTH
TRUNC:
        LD      A,0FFH              ;INDICATE SUBSTRING TRUNCATED
        LD      (INSERR),A
        LD      A,B                 ;LENGTH = MAX - STRING LENGTH
        SUB     (HL)
        RET     C                   ;RETURN WITH ERROR IF STRING TOO
        SCF                         ; LONG INITIALLY OR ALREADY MAX
        RET     Z                   ; LENGTH SO NO ROOM FOR SUBSTRING

        ;CHECK IF INDEX WITHIN STRING. IF NOT, CONCATENATE
        ; SUBSTRING ONTO END OF STRING
IDXLEN:
        LD      B,A                 ;B = LENGTH OF SUBSTRING
        LD      A,(HL)              ;GET STRING LENGTH
        CP      C                   ;COMPARE TO INDEX
        JR      NC,LENOK            ;JUMP IF STARTING INDEX WITHIN STRING

        ;INDEX NOT WITHIN STRING, SO CONCATENATE
        ; NEW LENGTH OF STRING = OLD LENGTH + SUBSTRING LENGTH
        LD      C,A                 ;SAVE CURRENT STRING LENGTH
        ADD     A,B                 ;ADD LENGTH OF SUBSTRING
        LD      (HL),A          ;SET NEW LENGTH OF STRING

        ;SET ADDRESSES FOR CONCATENATION
        ; DE = STRING ADDRESS + LENGTH(STRING) + 1

        ; HL = SUBSTRING ADDRESS
        EX      DE,HL           ;HL = SUBSTRING ADDRESS
        LD      A,C             ;DE = END OF STRING
        INC     A
        ADD     A,E
        LD      E,A
        JR      NC,IDXL1
        INC     D
IDXL1:
        LD      A,0FFH          ;INDICATE INSERTION ERROR
        LD      (INSERR),A
        JR      MVESUB          ;JUST MOVE, NOTHING TO OPEN UP

        ;OPEN UP SPACE IN SOURCE STRING FOR SUBSTRING BY MOVING
        ; CHARACTERS FROM END OF SOURCE STRING DOWN TO INDEX, UP BY
        ; SIZE OF SUBSTRING.
        ; A = LENGTH(STRING)
LENOK:
        PUSH    BC              ;SAVE LENGTH OF SUBSTRING
        PUSH    DE              ;SAVE ADDRESS OF SUBSTRING

        ;NEW LENGTH OF STRING = OLD LENGTH + SUBSTRING LENGTH
        LD      E,A             ;DE = STRING LENGTH
        LD      D,0
        ADD     A,B
        LD      (HL),A          ;STORE NEW LENGTH OF STRING

        ;CALCULATE NUMBER OF CHARACTERS TO MOVE
        ; = STRING LENGTH - STARTING INDEX + 1
        LD      A,E             ;GET ORIGINAL LENGTH OF STRING
        SUB     C
        INC     A               ;A = NUMBER OF CHARACTERS TO MOVE

        ;CALCULATE ADDRESS OF LAST CHARACTER IN STRING. THIS IS
        ; SOURCE ADDRESS = STRING ADDRESS + LENGTH(STRING)
        ADD     HL,DE           ;HL POINTS TO LAST CHARACTER IN STRING
        LD      E,L             ;DE ALSO
        LD      D,H

        ;CALCULATE DESTINATION ADDRESS
        ; = STRING ADDRESS + LENGTH(STRING) + LENGTH OF SUBSTRING
        ;THIS MOVE MUST START AT HIGHEST ADDRESS AND WORK DOWN
        ; TO AVOID OVERWRITING PART OF THE STRING
        LD      C,B             ;BC = LENGTH OF SUBSTRING
        LD      B,0
        ADD     HL,BC
        EX      DE,HL           ;HL = SOURCE ADDRESS
                                ;DE = DESTINATION ADDRESS
        LD      C,A             ;BC = NUMBER OF CHARACTERS TO MOVE
        LDDR                        ;OPEN UP FOR SUBSTRING

        ;RESTORE REGISTERS
        EX      DE,HL
        INC     DE                  ;DE = ADDRESS TO MOVE STRING TO
        POP     HL                  ;HL = ADDRESS OF SUBSTRING
        POP     BC                  ;B = LENGTH OF SUBSTRING

        ;MOVE SUBSTRING INTO OPEN AREA
        ; HL = ADDRESS OF SUBSTRING
        ; DE = ADDRESS TO MOVE SUBSTRING TO
        ; C = LENGTH OF SUBSTRING
MVESUB:
        INC     HL                  ;INCREMENT PAST LENGTH BYTE OF SUBSTRING
        LD      C,B                 ;BC = LENGTH OF SUBSTRING TO MOVE
        LD      B,0
        LDIR                        ;MOVE SUBSTRING INTO OPEN AREA

        LD      A,(INSERR)          ;GET ERROR FLAG
        RRA                         ;IF INSERR <> 0 THEN CARRY = 1
                                    ; TO INDICATE AN ERROR
        RET

        ;DATA SECTION
INSERR: DS      1                   ;FLAG USED TO INDICATE ERROR



;                                                                       ;
;                                                                       ;
;       SAMPLE EXECUTION:                                               ;
;                                                                       ;
;                                                                       ;


SC8F:
        LD      HL,STG   ;HL = BASE ADDRESS OF STRING
        LD      DE,SSTG  ;DE = BASE ADDRESS OF SUBSTRING
        LD      A,(IDX)
        LD      C,A      ;C = STARTING INDEX FOR INSERTION
        LD      A,(MXLEN)
        LD      B,A      ;B = MAXIMUM LENGTH OF STRING
        CALL    INSERT   ;INSERT SUBSTRING
                         ;RESULT OF INSERTING '-' INTO '123456' AT
                         ; INDEX 1 IS '-123456'

        JR      SC8F     ;LOOP FOR ANOTHER TEST

        ;DATA SECTION
IDX:    DB      1                   ;STARTING INDEX FOR INSERTION
MXLEN:  DB      20H                 ;MAXIMUM LENGTH OF DESTINATION
STG:    DB      06H                 ;LENGTH OF STRING
        DB      '123456                          ' ;32 BYTE MAX LENGTH
SSTG:   DB      1                   ;LENGTH OF SUBSTRING
        DB      '-                               ' ;32 BYTE MAX LENGTH

        END
```

8-Bit Array Summation (ASUM8) 9A

Adds the elements of an array, producing a 16-bit sum. The array consists of up to 255 byte-length elements.

Procedure: The program clears the sum initially. It then adds elements one at a time to the less significant byte of the sum, starting at the base address. Whenever an addition produces a carry, the program increments the more significant byte of the sum.

Registers Used: AF, B, DE, HL

Execution Time: Approximately 38 cycles per byte-length element plus 49 cycles overhead

Program Size: 19 bytes

Data Memory Required: None

Special Case: An array size of 0 causes an immediate exit with the sum equal to 0.

Entry Conditions

Base address of array in HL
Size of array in bytes in B

Exit Conditions

Sum in HL

Example

1. Data: Array consists of

$F7_{16}$	$5A_{16}$
23_{16}	16_{16}
31_{16}	CB_{16}
70_{16}	$E1_{16}$

Result: Sum = (HL) = $03D7_{16}$

```
;                                                                       ;
;                                                                       ;
;                                                                       ;
;                                                                       ;
;       Title           8-bit array summation                           ;
;       Name:           ASUM8                                           ;
;                                                                       ;
;                                                                       ;

;                                                                       ;
;       Purpose:        Sum the elements of an array, yielding a 16-bit ;
;                       result. Maximum size is 255                     ;
;                                                                       ;
;       Entry:          Register pair HL = Base address of array        ;
;                       Register B = Size of array in bytes             ;
;                                                                       ;
;       Exit:           Register pair HL = Sum                          ;
;                                                                       ;
;       Registers used: AF,B,DE,HL                                      ;
;                                                                       ;
;       Time:           Approximately 38 cycles per element plus        ;
;                       49 cycles overhead                              ;
;                                                                       ;
;       Size:           Program 19 bytes                                ;
;                                                                       ;
;                                                                       ;

ASUM8:
        ;TEST ARRAY LENGTH
        ;EXIT WITH SUM = 0 IF NOTHING IN ARRAY
        EX      DE,HL           ;SAVE BASE ADDRESS OF ARRAY
        LD      HL,0            ;INITIALIZE SUM TO 0

        ;CHECK FOR LENGTH OF ZERO
        LD      A,B             ;TEST ARRAY LENGTH
        OR      A
        RET     Z               ;EXIT WITH SUM = 0 IF LENGTH = 0

        ;INITIALIZE ARRAY POINTER, SUM
        EX      DE,HL           ;RESTORE BASE ADDRESS OF ARRAY
                                ; HIGH BYTE OF SUM = 0
        SUB     A               ;A = LOW BYTE OF SUM = 0
                                ;D = HIGH BYTE OF SUM

        ;ADD BYTE-LENGTH ELEMENTS TO SUM ONE AT A TIME
        ;INCREMENT HIGH BYTE OF SUM WHENEVER A CARRY OCCURS
SUMLP:
        ADD     A,(HL)          ;ADD NEXT BYTE
        JR      NC,DECCNT       ;JUMP IF NO CARRY
        INC     D               ; ELSE INCREMENT HIGH BYTE OF SUM
DECCNT:
        INC     HL
        DJNZ    SUMLP
EXIT:
        LD      L,A             ;HL = SUM
        LD      H,D
        RET


;                                                                       ;
;                                                                       ;
;       SAMPLE EXECUTION                                                ;
;                                                                       ;
;                                                                       ;

SC9A:
        LD      HL,BUF          ;HL = BASE ADDRESS OF BUFFER
        LD      A,(BUFSZ)
        LD      B,A             ;B = SIZE OF BUFFER IN BYTES
        CALL    ASUM8
                                ;SUM OF TEST DATA IS 07F8 HEX,
                                ; HL = 07F8H

        JR      SC9A

;TEST DATA, CHANGE FOR OTHER VALUES
SIZE    EQU     010H            ;SIZE OF BUFFER IN BYTES
BUFSZ:  DB      SIZE            ;SIZE OF BUFFER IN BYTES

BUF:    DB      00H             ;BUFFER
        DB      11H             ;DECIMAL ELEMENTS ARE 0,17,34,51,68
        DB      22H             ; 85,102,119,135,153,170,187,204
        DB      33H             ; 221,238,255
        DB      44H
        DB      55H
        DB      66H
        DB      77H
        DB      88H
        DB      99H
        DB      0AAH
        DB      0BBH
        DB      0CCH
        DB      0DDH
        DB      0EEH
        DB      0FFH            ;SUM = 07F8 (2040 DECIMAL)

        END
```

16-Bit Array Summation (ASUM16) 9B

Adds the elements of an array, producing a 24-bit sum. The array consists of up to 255 word-length (16-bit) elements. The elements are arranged in the usual Z80 format with the less significant bytes first.

Procedure: The program clears the sum initially. It then adds elements to the less significant bytes of the sum one at a time, starting at the base address. Whenever an addition produces a carry, the program increments the most significant byte of the sum.

Registers Used: AF, BC, DE, HL

Execution Time: Approximately 68 cycles per 16-bit element plus 49 cycles overhead

Program Size: 25 bytes

Data Memory Required: None

Special Case: An array size of 0 causes an immediate exit with the sum equal to 0.

Entry Conditions

Base address of array in HL
Size of array in 16-bit words in B

Exit Conditions

Most significant byte of sum in E
Middle and least significant bytes of sum in HL

Example

1. Data: Array (in 16-bit words) consists of

$F7A1_{16}$	$5A36_{16}$
$239B_{16}$	$166C_{16}$
$31D5_{16}$	$CBF5_{16}$
$70F2_{16}$	$E107_{16}$

Result: Sum = $03DBA1_{16}$
(E) = 03_{16}
(HL) = $DBA1_{16}$

```
;                                                                   ;
;                                                                   ;
;                                                                   ;
;                                                                   ;
;       Title           16-bit array summation                      ;
;       Name:           ASUM16                                      ;
;                                                                   ;
;                                                                   ;
;                                                                      ;
;       Purpose:         Sum the elements of an array, yielding a 24-bit ;
;                        result. Maximum size is 255 16-bit elements   ;
;                                                                      ;
;       Entry:           Register pair HL = Base address of array      ;
;                        Register B = Size of array in words           ;
;                                                                      ;
;       Exit:            Register A = High byte of sum                 ;
;                        Register H = Middle byte of sum               ;
;                        Register L = Low byte of sum                  ;
;                                                                      ;
;       Registers used: AF,BC,DE,HL                                    ;
;                                                                      ;
;       Time:            Approximately 68 cycles per element plus      ;
;                        49 cycles overhead                            ;
;                                                                      ;
;       Size:            Program 25 bytes                              ;
;                                                                      ;
;                                                                      ;

ASUM16:
        ;TEST ARRAY LENGTH
        ;EXIT WITH SUM = 0 IF NOTHING IN ARRAY
        EX      DE,HL           ;SAVE BASE ADDRESS OF ARRAY
        LD      HL,0            ;INITIALIZE SUM TO 0

        ;CHECK FOR ARRAY LENGTH OF ZERO
        LD      A,B             ;TEST ARRAY LENGTH
        OR      A
        RET     Z               ;EXIT WITH SUM = 0 IF LENGTH = 0

        ;INITIALIZE ARRAY POINTER, SUM
        EX      DE,HL           ;BASE ADDRESS BACK TO HL
                                ; LOW, MIDDLE BYTES OF SUM = 0
        LD      C,E             ;C = HIGH BYTE OF SUM = 0
                                ;D = MIDDLE BYTE OF SUM
                                ;E = LOW BYTE OF SUM

        ;ADD WORD-LENGTH ELEMENTS TO SUM ONE AT A TIME
        ;INCREMENT HIGH BYTE OF SUM WHENEVER A CARRY OCCURS
SUMLP:
        LD      A,E             ;ADD LOW BYTES OF ELEMENT AND SUM
        ADD     A,(HL)
        LD      E,A
        INC     HL              ;ADD HIGH BYTE OF ELEMENT TO
        LD      A,D             ; MIDDLE BYTE OF SUM
        ADC     A,(HL)
        LD      D,A
        JR      NC,DECCNT       ;JUMP IF NO CARRY
        INC     C               ; ELSE INCREMENT HIGH BYTE OF SUM

DECCNT:
        INC     HL
        DJNZ    SUMLP
EXIT:
        EX      DE,HL           ;HL = MIDDLE AND LOW BYTES OF SUM
        LD      A,C                 ;A  = HIGH BYTE OF SUM
        RET


;                                                                  ;
;                                                                  ;
;       SAMPLE EXECUTION                                           ;
;                                                                  ;
;                                                                  ;

SC9B:
        LD      HL,BUF              ;HL = BASE ADDRESS OF BUFFER
        LD      A,(BUFSZ)
        LD      B,A                 ;B = SIZE OF BUFFER IN WORDS
        CALL    ASUM16
                                    ;SUM OF TEST DATA IS 31FF8 HEX,
                                    ; REGISTER PAIR HL = 1FF8H
                                    ; REGISTER A = 3

        JR      SC9B

;TEST DATA, CHANGE FOR OTHER VALUES
SIZE    EQU     010H                ;SIZE OF BUFFER IN WORDS
BUFSZ:  DB      SIZE                ;SIZE OF BUFFER IN WORDS

BUF:    DW      000H                ;BUFFER
        DW      111H                ;DECIMAL ELEMENTS ARE 0,273,546,819,1092
        DW      222H                ; 1365,1638,1911,2184,2457,2730,3003,3276
        DW      333H                ; 56797,61166,65535
        DW      444H
        DW      555H
        DW      666H
        DW      777H
        DW      888H
        DW      999H
        DW      0AAAH
        DW      0BBBH
        DW      0CCCH
        DW      0DDDDH
        DW      0EEEEH
        DW      0FFFFH              ;SUM = 31FF8 (204792 DECIMAL)

        END
```

Find Maximum Byte-Length Element (MAXELM) 9C

Finds the maximum element in an array. The array consists of up to 255 unsigned byte-length elements.

Procedure: The program exits immediately (setting Carry to 1) if the array has no elements. Otherwise, the program assumes that the element at the base address is the maximum. It then proceeds through the array, comparing the supposed maximum with each element and retaining the larger value and its address. Finally, the program clears Carry to indicate a valid result.

Registers Used: AF, B, DE, HL

Execution Time: Approximately 36 to 58 cycles per element plus 35 cycles overhead. If, on the average, the program must replace the maximum in half of the iterations, the execution time is approximately 94 * ARRAY SIZE/2 + 35 cycles.

Program Size: 19 bytes

Data Memory Required: None

Special Cases:

1. An array size of 0 causes an immediate exit with the Carry flag set to 1 to indicate an invalid result.

2. If the largest unsigned value occurs more than once, the program returns with the lowest possible address. That is, it returns with the address closest to the base address that contains the maximum value.

Entry Conditions

Base address of array in HL
Size of array in bytes in B

Exit Conditions

Largest unsigned element in A
Address of largest unsigned element in HL
Carry = 0 if result is valid; 1 if size of array is 0 and result is meaningless.

Example

1. Data: Array (in bytes) consists of

35_{16}	44_{16}
$A6_{16}$	59_{16}
$D2_{16}$	$7A_{16}$
$1B_{16}$	CF_{16}

Result: The largest unsigned element is element #2 ($D2_{16}$)
(A) = largest element ($D2_{16}$)
(HL) = BASE + 2 (lowest address containing $D2_{16}$)
Carry flag = 0, indicating that array size is non-zero and the result is valid.

```
;                                                                          ;
;                                                                          ;
;                                                                          ;
;                                                                          ;
;       Title               Find maximum byte-length element               ;
;                                                                          ;
;       Name:               MAXELM                                         ;
;                                                                          ;
;                                                                          ;

;                                                                          ;
;       Purpose:            Given the base address and size of an array,   ;
;                           find the largest element                       ;
;                                                                          ;
;       Entry:              Register pair HL = Base address of array       ;
;                           Register B = Size of array in bytes            ;
;                                                                          ;
;       Exit:               If size of array not zero then                 ;
;                             Carry flag = 0                               ;
;                             Register A = Largest element                 ;
;                             Register pair HL = Address of that element   ;
;                              if there are duplicate values of the largest ;
;                              element, register pair HL has the address   ;
;                              nearest to the base address                 ;
;                           else                                           ;
;                             Carry flag = 1                               ;
;                                                                          ;
;       Registers used: AF,B,DE,HL                                         ;
;                                                                          ;
;       Time:               Approximately 36 to 58 cycles per element      ;
;                           plus 35 cycles overhead                        ;
;                                                                          ;
;       Size:               Program 19 bytes                               ;
;                                                                          ;
;                                                                          ;

MAXELM:
        ;EXIT WITH CARRY SET IF NO ELEMENTS IN ARRAY
        LD      A,B             ;TEST ARRAY SIZE
        OR      A
        SCF                     ;SET CARRY TO INDICATE ERROR EXIT
        RET     Z               ;RETURN IF NO ELEMENTS

        ;REPLACE PREVIOUS GUESS AT LARGEST ELEMENT WITH
        ; CURRENT ELEMENT.  FIRST TIME THROUGH, TAKE FIRST
        ; ELEMENT AS GUESS AT LARGEST
MAXLP:  LD      A,(HL)          ;LARGEST = CURRENT ELEMENT
        LD      E,L             ;SAVE ADDRESS OF LARGEST
        LD      D,H

        ;COMPARE CURRENT ELEMENT TO LARGEST
        ;KEEP LOOKING UNLESS CURRENT ELEMENT IS LARGER
MAXLP1:
        DEC     B
        JR      Z,EXIT
        INC     HL
        CP      (HL)            ;COMPARE CURRENT ELEMENT, LARGEST
        JR      NC,MAXLP1       ;CONTINUE UNLESS CURRENT ELEMENT LARGER
        JR      MAXLP           ;ELSE CHANGE LARGEST
EXIT:
        OR      A               ;CLEAR CARRY TO INDICATE NO ERRORS
        EX      DE,HL           ;HL = ADDRESS OF LARGEST ELEMENT
        RET


;                                                                       ;
;                                                                       ;
;       SAMPLE EXECUTION:                                               ;
;                                                                       ;
;                                                                       ;

SC9C:
        LD      HL,ARY          ;HL = BASE ADDRESS OF ARRAY
        LD      B,SZARY         ;B = SIZE OF ARRAY IN BYTES
        CALL    MAXELM
                                ;RESULT FOR TEST DATA IS
                                ; A = FF HEX (MAXIMUM), HL = ADDRESS OF
                                ; FF IN ARY

        JR      SC9C            ;LOOP FOR MORE TESTING

SZARY   EQU     10H             ;SIZE OF ARRAY IN BYTES
ARY:    DB      8
        DB      7
        DB      6
        DB      5
        DB      4
        DB      3
        DB      2
        DB      1
        DB      0FFH
        DB      0FEH
        DB      0FDH
        DB      0FCH
        DB      0FBH
        DB      0FAH
        DB      0F9H
        DB      0F8H

        END
```

Find Minimum Byte-Length Element (MINELM) 9D

Finds the minimum element in an array. The array consists of up to 255 unsigned byte-length elements.

Procedure: The program exits immediately (setting Carry to 1) if the array has no elements. Otherwise, the program assumes that the element at the base address is the minimum. It then proceeds through the array, comparing the supposed minimum to each element and retaining the smaller value and its address. Finally, the program clears Carry to indicate a valid result.

Registers Used: AF, B, DE, HL

Execution Time: Approximately 36 to 65 cycles per element plus 35 cycles overhead. If, on the average, the program must replace the minimum in half of the iterations, the execution time is approximately 101 * ARRAY SIZE/2 +35 cycles.

Program Size: 21 bytes

Data Memory Required: None

Special Cases:

1. An array size of 0 causes an immediate exit with the Carry flag set to 1 to indicate an invalid result.

2. If the smallest unsigned value occurs more than once, the program returns with the lowest possible address. That is, it returns with the address closest to the base address that contains the minimum value.

Entry Conditions

Base address of array in HL
Size of array in bytes in B

Exit Conditions

Smallest unsigned element in A
Address of smallest unsigned element in HL
Carry = 0 if result is valid; 1 if size of array is 0 and result is meaningless.

Example

1. Data: Array (in bytes) consists of

35_{16}	44_{16}
$A6_{16}$	59_{16}
$D2_{16}$	$7A_{16}$
$1B_{16}$	CF_{16}

Result: The smallest unsigned element is element #3 ($1B_{16}$)
(A) = smallest element ($1B_{16}$)
(HL) = BASE + 3 (lowest address containing $1B_{16}$)
Carry flag = 0, indicating that array size is non-zero and the result is valid.

```
;                                                                     ;
;                                                                     ;
;                                                                     ;
;                                                                     ;
;       Title           Find minimum byte-length element             ;
;                                                                     ;
;       Name:           MINELM                                        ;
;                                                                     ;
;                                                                     ;


;                                                                     ;
;       Purpose:        Given the base address and size of an array,  ;
;                       find the smallest element                     ;
;                                                                     ;
;       Entry:          Register pair HL = Base address of array      ;
;                       Register B = Size of array in bytes           ;
;                                                                     ;
;       Exit:           If size of array not zero then                ;
;                         Carry flag = 0                              ;
;                         Register A = Smallest element               ;
;                         Register pair HL = Address of that element  ;
;                          if there are duplicate values of the smallest;
;                          element, HL will have the address          ;
;                          nearest to the base address                ;
;                       else                                          ;
;                          Carry flag = 1                             ;
;                                                                     ;
;       Registers used: AF,B,DE,HL                                    ;
;                                                                     ;
;       Time:           Approximately 36 to 65 cycles per element     ;
;                       plus 35 cycles overhead                       ;
;                                                                     ;
;       Size:           Program 21 bytes                              ;
;                                                                     ;
;                                                                     ;

MINELM:
        ;EXIT WITH CARRY SET IF NO ELEMENTS IN ARRAY
        LD      A,B                 ;TEST ARRAY SIZE
        OR      A
        SCF                         ;SET CARRY TO INDICATE AN ERROR EXIT
        RET     Z                   ;RETURN IF NO ELEMENTS

        ;REPLACE PREVIOUS GUESS AT SMALLEST ELEMENT WITH
        ; CURRENT ELEMENT.  FIRST TIME THROUGH, TAKE FIRST
        ; ELEMENT AS GUESS AT SMALLEST
MINLP:  LD      A,(HL)              ;SMALLEST = CURRENT ELEMENT
        LD      E,L                 ;SAVE ADDRESS OF SMALLEST
        LD      D,H

        ;COMPARE CURRENT ELEMENT TO SMALLEST
        ;KEEP LOOKING UNLESS CURRENT ELEMENT IS SMALLER
MINLP1:
        DEC     B
        JR      Z,EXIT
        INC     HL
        CP      (HL)            ;COMPARE CURRENT ELEMENT, SMALLEST
        JR      C,MINLP1        ;CONTINUE IF CURRENT ELEMENT LARGER
        JR      Z,MINLP1        ; OR SAME
        JR      MINLP           ;ELSE CHANGE SMALLEST
EXIT:
        OR      A               ;CLEAR CARRY TO INDICATE NO ERRORS
        EX      DE,HL           ;HL = ADDRESS OF SMALLEST ELEMENT
        RET


;                                                                  ;
;                                                                  ;
;       SAMPLE EXECUTION:                                          ;
;                                                                  ;
;                                                                  ;

SC9D:
        LD      HL,ARY          ;HL = BASE ADDRESS OF ARRAY
        LD      B,SZARY         ;B = SIZE OF ARRAY IN BYTES
        CALL    MINELM
                                ;RESULT FOR TEST DATA IS
                                ; A = 1 HEX (MINIMUM), HL = ADDRESS OF
                                ; 1 IN ARY

        JR      SC9D            ;LOOP FOR MORE TESTING

SZARY   EQU     10H             ;SIZE OF ARRAY IN BYTES
ARY:    DB      8
        DB      7
        DB      6
        DB      5
        DB      4
        DB      3
        DB      2
        DB      1
        DB      0FFH
        DB      0FEH
        DB      0FDH
        DB      0FCH
        DB      0FBH
        DB      0FAH
        DB      0F9H
        DB      0F8H

        END
```

Searches an array of unsigned byte-length elements for a particular value. The elements are assumed to be arranged in increasing order. Clears Carry if it finds the value and sets Carry to 1 if it does not. Returns the address of the value if found. The size of the array is specified and is a maximum of 255 bytes.

Procedure: The program performs a binary search, repeatedly comparing the value with the middle remaining element. After each comparison, the program discards the part of the array that cannot contain the value (because of the ordering). The program retains upper and lower bounds for the remaining part. If the value is larger than the middle element, the program discards the middle and everything below it. The new lower bound is the address of the middle element plus 1. If the value is smaller than the middle element, the program discards the middle and everything above it. The new upper bound is the address of the middle element minus 1. The program exits if it finds a match or if there is nothing left to search.

For example, assume that the array is

01_{16}, 02_{16}, 05_{16}, 07_{16}, 09_{16}, 09_{16}, $0D_{16}$, 10_{16},
$2E_{16}$, 37_{16}, $5D_{16}$, $7E_{16}$, $A1_{16}$, $B4_{16}$, $D7_{16}$, $E0_{16}$

and the value to be found is $0D_{16}$. The procedure works as follows.

In the first iteration, the lower bound is the base address and the upper bound is the address of the last element. So the result is

LOWER BOUND = BASE
UPPER BOUND = BASE + SIZE − 1 = BASE + $0F_{16}$
GUESS = (UPPER BOUND + LOWER BOUND)/2
(the result is truncated) = BASE + 7
(GUESS) = ARRAY(7) = 10_{16} = 16_{10}

Since the value ($0D_{16}$) is less than ARRAY(7), the elements beyond #6 can be discarded. So the result is

Registers Used: AF, BC, DE, HL

Execution Time: Approximately 114 cycles per iteration plus 53 cycles overhead. A binary search requires on the order of $\log_2 N$ iterations, where N is the number of elements in the array.

Program Size: 37 bytes

Data Memory Required: None

Special Case: A size of 0 causes an immediate exit with the Carry flag set to 1. That is, the array contains no elements and the value surely cannot be found.

LOWER BOUND = BASE
UPPER BOUND = GUESS − 1 = BASE + 6
GUESS = (UPPER BOUND + LOWER BOUND)/2
= BASE + 3
(GUESS) = ARRAY(3) = 07

Since the value ($0D_{16}$) is greater than ARRAY(3), the elements below #4 can be discarded. So the result is

LOWER BOUND = GUESS + 1 = BASE + 4
UPPER BOUND = BASE + 6
GUESS = (UPPER BOUND + LOWER BOUND)/2
= BASE + 5
(GUESS) = ARRAY(5) = 09

Since the value ($0D_{16}$) is greater than ARRAY(5), the elements below #6 can be discarded. So the result is

LOWER BOUND = GUESS + 1 = BASE + 6
UPPER BOUND = BASE + 6
GUESS = (UPPER BOUND + LOWER BOUND)/2
= 06
(GUESS) = ARRAY(6) = $0D_{16}$

Since the value ($0D_{16}$) is equal to ARRAY(6), the element has been found. If, on the other hand, the value were $0E_{16}$, the new lower bound would be BASE + 7 and there would be nothing left to search.

Entry Conditions

Value to find in A
Size of the array in bytes in C
Base address of array (address of smallest unsigned element) in HL

Exit Conditions

Carry = 0 if the value is found; 1 if it is not found.
If the value is found, (HL) = its address.

Examples

Length of array = 10_{16}
Elements of array are 01_{16}, 02_{16}, 05_{16}, 07_{16}, 09_{16}, 09_{16}, $0D_{16}$, 10_{16}, $2E_{16}$, 37_{16}, $5D_{16}$, $7E_{16}$, $A1_{16}$, $B4_{16}$, $D7_{16}$, $E0_{16}$

1. Data: Value to find = $0D_{16}$

   Result: Carry = 0, indicating value found
   (HL) = BASE + 6 (address containing $0D_{16}$)

2. Data: Value to find = $9B_{16}$

   Result: Carry = 1, indicating value not found

```
;                                                                         ;
;                                                                         ;
;                                                                         ;
;                                                                         ;
;       Title           Binary search                                     ;
;       Name:           BINSCH                                            ;
;                                                                         ;
;                                                                         ;

;                                                                         ;
;       Purpose:        Search an ordered array of unsigned bytes         ;
;                       with a maximum size of 255 elements               ;
;                                                                         ;
;       Entry:          Register pair HL = Base address of array          ;
;                       Register C = Size of array                        ;
;                       Register A = Byte to find                         ;
;                                                                         ;
;       Exit:           If the value is found then                        ;
;                         Carry flag = 0                                  ;
;                         Register pair HL = Address of value             ;
;                       ELSE                                              ;
;                         Carry flag = 1                                  ;
;                                                                         ;
;       Registers used: AF,BC,DE,HL                                       ;
;                                                                       ;
;       Time:           Approximately 114 cycles for each iteration of  ;
;                       the search loop plus 53 cycles overhead         ;
;                                                                       ;
;                       A binary search takes on the order of log       ;
;                       base 2 of N searches, where N is the number of  ;
;                       elements in the array.                          ;
;                                                                       ;
;       Size:           Program 37 bytes                                ;
;                                                                       ;
;                                                                       ;

BINSCH:
        ;EXIT WITH CARRY SET IF NO ELEMENTS IN ARRAY
        INC     C               ;TEST ARRAY SIZE
        DEC     C
        SCF                     ;SET CARRY IN CASE SIZE IS 0
        RET     Z               ;RETURN INDICATING VALUE NOT FOUND
                                ; IF SIZE IS 0

        ;INITIALIZE LOWER BOUND, UPPER BOUND OF SEARCH AREA
        ;LOWER BOUND (DE) = BASE ADDRESS
        ;UPPER BOUND (HL) = ADDRESS OF LAST ELEMENT
        ; = BASE ADDRESS + SIZE - 1
        LD      E,L             ;LOWER BOUND = BASE ADDRESS
        LD      D,H
        LD      B,0             ;EXTEND SIZE TO 16 BITS
        ADD     HL,BC           ;UPPER BOUND = BASE + SIZE - 1
        DEC     HL

        ;SAVE VALUE BEING SOUGHT
        LD      C,A             ;SAVE VALUE

        ;ITERATION OF BINARY SEARCH
        ;1) COMPARE VALUE TO MIDDLE ELEMENT
        ;2) IF THEY ARE NOT EQUAL, DISCARD HALF THAT
        ;   CANNOT POSSIBLY CONTAIN VALUE (BECAUSE OF ORDERING)
        ;3) CONTINUE IF THERE IS ANYTHING LEFT TO SEARCH
LOOP:
        ;HL = UPPER BOUND
        ;DE = LOWER BOUND
        ;C = VALUE TO FIND
        ;FIND MIDDLE ELEMENT
        ;MIDDLE = (UPPER BOUND + LOWER BOUND) / 2
        PUSH    HL              ;SAVE UPPER BOUND ON STACK
        ADD     HL,DE           ;ADD UPPER BOUND AND LOWER BOUND
        RR      H               ;DIVIDE 17-BIT SUM BY 2
        RR      L
        LD      A,(HL)          ;GET MIDDLE ELEMENT

        ;COMPARE MIDDLE ELEMENT AND VALUE
        CP      C               ;COMPARE MIDDLE ELEMENT AND VALUE
        JR      NC,TOOLRG       ;JUMP IF VALUE SAME OR LARGER

        ;MIDDLE ELEMENT LESS THAN VALUE
        ; SO CHANGE LOWER BOUND TO MIDDLE + 1
        ; SINCE EVERYTHING BELOW MIDDLE IS EVEN SMALLER
        EX      DE,HL           ;LOWER BOUND = MIDDLE + 1
        INC     DE
        POP     HL              ;RESTORE UPPER BOUND
        JR      CONT

        ;MIDDLE ELEMENT GREATER THAN OR EQUAL TO VALUE
        ; SO CHANGE UPPER BOUND TO MIDDLE - 1
        ; SINCE EVERYTHING ABOVE MIDDLE IS EVEN LARGER
        ;EXIT WITH CARRY CLEAR IF VALUE FOUND
TOOLRG:
        INC     SP              ;DISCARD OLD UPPER BOUND FROM STACK
        INC     SP
        RET     Z               ;IF MIDDLE ELEMENT SAME AS VALUE
                                ; RETURN WITH CARRY CLEAR
                                ; AND HL = ADDRESS CONTAINING VALUE
        DEC     HL              ;UPPER BOUND = MIDDLE - 1

        ;CONTINUE IF THERE IS ANYTHING LEFT TO BE SEARCHED
        ;NOTHING LEFT WHEN LOWER BOUND ABOVE UPPER BOUND
CONT:
        LD      A,L             ;FORM UPPER BOUND - LOWER BOUND
        CP      E               ; MUST SAVE BOTH, SO USE 8-BIT SUBTRACT
        LD      A,H
        SBC     A,D
        JR      NC,LOOP         ;CONTINUE IF ANYTHING LEFT TO SEARCH

        ;NOTHING LEFT TO SEARCH SO COULD NOT FIND VALUE
        ;RETURN WITH CARRY SET (MUST BE OR JR NC WOULD HAVE BRANCHED)
        RET


;                                                                       ;
;                                                                       ;
;       SAMPLE EXECUTION                                                ;
;                                                                       ;
;                                                                       ;

SC9E:
        ;SEARCH FOR A VALUE THAT IS IN THE ARRAY
        LD      HL,BF           ;HL = BASE ADDRESS OF ARRAY
        LD      A,(BFSZ)
        LD      C,A             ;C = ARRAY SIZE IN BYTES
        LD      A,7             ;A = VALUE TO FIND
        CALL    BINSCH          ;SEARCH
                                ;CARRY FLAG = 0 (VALUE FOUND)
                                ;HL = BF + 4 (ADDRESS OF 7 IN ARRAY)

        ;SEARCH FOR A VALUE THAT IS NOT IN THE ARRAY
        LD      HL,BF           ;HL = BASE ADDRESS OF ARRAY
        LD      A,(BFSZ)
        LD      C,A             ;C = ARRAY SIZE IN BYTES
        LD      A,0             ;A = VALUE TO FIND
        CALL    BINSCH          ;SEARCH
                                ;CARRY FLAG = 1 (VALUE NOT FOUND)
        JR      SC9E            ;LOOP FOR MORE TESTS

        ;DATA
SIZE    EQU     010H            ;SIZE OF ARRAY IN BYTES
BFSZ:   DB      SIZE            ;SIZE OF ARRAY IN BYTES
BF:     DB      1               ;BUFFER
        DB      2
        DB      4
        DB      5
        DB      7
        DB      9
        DB      10
        DB      11
        DB      23
        DB      50
        DB      81
        DB      123
        DB      191
        DB      199
        DB      250
        DB      255

        END
```

Quicksort (QSORT) 9F

Arranges an array of unsigned word-length elements into ascending order using a quicksort algorithm. Each iteration selects an element and divides the array into two parts, one consisting of elements larger than the selected element and the other consisting of elements smaller than the selected element. Elements equal to the selected element may end up in either part. The parts are then sorted recursively in the same way. The algorithm continues until all parts contain either no elements or only one element. An alternative is to stop recursion when a part contains few enough elements (say, less than 20) to make a bubble sort practical.

The parameters are the array's base address, the address of its last element, and the lowest available stack address. The array can thus occupy all available memory, as long as there is room for the stack. Since the procedures that obtain the selected element, compare elements, move forward and backward in the array, and swap elements are all subroutines, they could be changed readily to handle other types of elements.

Ideally, quicksort should divide the array in half during each iteration. How closely the procedure approaches this ideal depends on how well the selected element is chosen. Since this element serves as a midpoint or pivot, the best choice would be the central value (or median). Of course, the true median is unknown. A simple but reasonable approximation is to select the median of the first, middle, and last elements.

Procedure: The program first deals with the entire array. It selects the median of the current first, last, and middle elements to use in dividing the array. It moves that element to the first position and divides the array into two parts or partitions. It then operates recursively on the parts, dividing them further into parts and stopping when a part contains no elements or only one element. Since each recursion places six bytes on the stack, the program must guard against stack overflow by checking whether the stack has grown to within a small buffer of its lowest available address.

Registers Used: AF, BC, DE, HL

Execution Time: Approximately N * $\log_2$N loops through PARTLP plus 2 * N + 1 overhead calls to SORT. Each iteration of PARTLP takes approximately 200 cycles and each overhead call to SORT takes approximately 300 cycles. Thus, the total execution time is on the order of 200 * N * $\log_2$N + 300 * (2 * N + 1).

Program Size: 206 bytes

Data Memory Required: 8 bytes anywhere in RAM for pointers to the first and last elements of a partition (2 bytes starting at addresses FIRST and LAST, respectively), a pointer to the bottom of the stack (2 bytes starting at address STKBTM), and the original value of the stack pointer (2 bytes starting at address OLDSP).

Special Case: If the stack overflows (i.e., comes too close to its boundary), the program exits with the Carry flag set to 1.

Note that the selected element always ends up in the correct position after an iteration. Therefore, it need not be included in either partition.

The rules for choosing the middle element are as follows, assuming that the first element is #1:

1. If the array has an odd number of elements, take the one in the center. For example, if the array has 11 elements, take #6.

2. If the array has an even number of elements and its base address is even, take the element on the lower (base address) side of the center. For example, if the array starts in 0300_{16} and has 12 elements, take #6.

3. If the array has an even number of elements and its base address is odd, take the element on the upper side of the center. For example, if the array starts in 0301_{16} and has 12 elements, take #7.

Entry Conditions

Base address of array in HL
Address of last word of array in DE
Lowest available stack address in BC

Exit Conditions

Array sorted into ascending order, considering the elements as unsigned words. Thus, the smallest unsigned word ends up stored starting at the base address. Carry = 0 if the stack did not overflow and the result is proper. Carry = 1 if the stack overflowed and the final array is not sorted.

Example

1. Data: Length (size) of array = $0C_{16}$
Elements = $2B_{16}$, 57_{16}, $1D_{16}$, 26_{16}, 22_{16}, $2E_{16}$, $0C_{16}$, 44_{16}, 17_{16}, $4B_{16}$, 37_{16}, 27_{16}

Result: The result of the first iteration is:

Selected element = median of the first (#1 = $2B_{16}$), middle (#6 = $2E_{16}$), and last (#12 = 27_{16}) elements. The selected element is therefore #1 ($2B_{16}$), and no swapping is necessary since it is already in the first position.

At the end of the iteration, the array is

27_{16}, 17_{16}, $1D_{16}$, 26_{16},
22_{16}, $0C_{16}$, $2B_{16}$, 44_{16},
$2E_{16}$, $4B_{16}$, 37_{16}, 57_{16}.

The first partition, consisting of elements less than $2B_{16}$, is 27_{16}, 17_{16}, $1D_{16}$, 26_{16}, 22_{16}, and $0C_{16}$.

The second partition, consisting of elements greater than $2B_{16}$, is 44_{16}, $2E_{16}$, $4B_{16}$, 37_{16}, and 57_{16}.

Note that the selected element ($2B_{16}$) is now in the correct position and need not be included in either partition.

The first partition may now be sorted recursively in the same way:

Selected element = median of the first (#1 = 27_{16}), middle (#3 = $1D_{16}$), and last (#7 = $0C_{16}$) elements. Here, #4 is the median and must be exchanged initially with #1.

The final order of the elements in the first partition is

$0C_{16}$, 17_{16}, $1D_{16}$, 26_{16}, 22_{16}, 27_{16}.

The first partition of the first partition (consisting of elements less than $1D_{16}$) is $0C_{16}$, 17_{16}. This will be referred to as the (1,1) partition.

The second partition of the first partition (consisting of elements greater than $1D_{16}$) is 26_{16}, 22_{16}, and 27_{16}.

As in the first iteration, the selected element ($1D_{16}$) is in the correct position and need not be considered further.

The (1,1) partition may now be sorted recursively as follows:

Selected element = median of the first (#1 = $0C_{16}$), middle (#1 = $0C_{16}$), and last (#2 = 17_{16}) elements. Thus the selected element is the first element (#1 = $0C_{16}$) and no initial swap is necessary.

The final order is obviously the same as the initial order, and the two resulting partitions contain 0 and 1 elements, respectively. Thus the next iteration concludes the recursion, and the other partitions are sorted by the same method. Obviously, quicksort's overhead is large when the number of elements is small. This is why one might use a bubble sort once quicksort has created small enough partitions.

Note that the example array does not contain any identical elements. During an iteration, elements that are the same as the selected element are never moved. Thus they may end up in either partition. Strictly speaking, then, the two partitions consist of elements "less than or possibly equal to the selected element" and elements "greater than or possibly equal to the selected element."

REFERENCES

Augenstein, M.J., and Tenenbaum, A.M. *Data Structures and PL/I Programming.* Englewood Cliffs, N.J.: Prentice-Hall, 1979, pp. 460–71. There is also a Pascal version of this book entitled *Data Structures Using Pascal* (Englewood Cliffs, N.J.: Prentice-Hall, 1982).

Bowles, K.L. *Microcomputer Problem Solving Using Pascal.* New York: Springer-Verlag, 1977, Chapter 15.

Knuth, D.E. *The Art of Computer Programming, Volume 3: Searching and Sorting.* Reading, Mass.: Addison-Wesley, 1973, pp. 114–23.

```
;                                                             ;
;                                                             ;
;                                                             ;
;                                                             ;
;       Title          Quicksort                              ;
;       Name:          QSORT                                  ;
;                                                             ;
;                                                             ;

;                                                             ;
;       Purpose:       Arrange an array of unsigned words into ;
;                      ascending order using quicksort, with a ;
;                      maximum size of 32,767 words            ;
;                                                             ;
;       Entry:         Register pair HL = Address of first word in the ;
;                                         array                ;
;                   Register pair DE = Address of last word in the ;
;                                      array                       ;
;                   Register pair BC = Lowest available stack      ;
;                                      address                     ;
;                                                                  ;
;       Exit:       If the stack did not overflow then             ;
;                     array is sorted into ascending order.        ;
;                     Carry flag = 0                               ;
;                   Else                                           ;
;                     Carry flag = 1                               ;
;                                                                  ;
;       Registers used: AF,BC,DE,HL                                ;
;                                                                  ;
;       Time:       The timing is highly data-dependent but the    ;
;                   quicksort algorithm takes approximately        ;
;                   N * log (N) loops through PARTLP. There will be ;
;                   2 * N+1 calls to Sort. The number of recursions ;
;                   will probably be a fraction of N but if all    ;
;                   data is the same, the recursion could be up to ;
;                   N. Therefore the amount of stack space should  ;
;                   be maximized. NOTE: Each recursion level takes ;
;                   6 bytes of stack space.                        ;
;                                                                  ;
;                   In the above discussion N is the number of     ;
;                   array elements.                                ;
;                                                                  ;
;                   For example, sorting a 16,384-word array took  ;
;                   about 27 seconds and 1200 bytes of stack space ;
;                   on a 6 MHz Z80.                                ;
;                                                                  ;
;       Size:       Program 206 bytes                              ;
;                   Data      8 bytes                              ;
;                                                                  ;
;                                                                  ;

QSORT:
        ;WATCH FOR STACK OVERFLOW
        ;CALCULATE A THRESHOLD TO WARN OF OVERFLOW
        ; (10 BYTES FROM THE END OF THE STACK)
        ;SAVE THIS THRESHOLD FOR LATER COMPARISONS
        ;ALSO SAVE THE POSITION OF THIS ROUTINE'S RETURN ADDRESS
        ; IN THE EVENT WE MUST ABORT BECAUSE OF STACK OVERFLOW
        PUSH    HL              ;SAVE BASE ADDRESS OF ARRAY
        LD      HL,10           ;ADD SMALL BUFFER (10 BYTES) TO
        ADD     HL,BC           ; LOWEST STACK ADDRESS
        LD      (STKBTM),HL     ;SAVE SUM AS BOTTOM OF STACK
                                ; FOR FIGURING WHEN TO ABORT
        LD      HL,2            ;SAVE POINTER TO RETURN ADDRESS
        ADD     HL,SP           ; IN CASE OF ABORT
        LD      (OLDSP),HL
        POP     HL              ;RESTORE BASE ADDRESS

        ;WORK RECURSIVELY THROUGH THE QUICKSORT ALGORITHM AS
        ; FOLLOWS:
```

```
        ;  1. CHECK IF THE PARTITION CONTAINS 0 OR 1 ELEMENT.
        ;     MOVE UP A RECURSION LEVEL IF IT DOES.
        ;  2. USE MEDIAN TO OBTAIN A REASONABLE CENTRAL VALUE
        ;     FOR DIVIDING THE CURRENT PARTITION INTO TWO
        ;     PARTS.
        ;  3. MOVE THROUGH ARRAY SWAPPING ELEMENTS THAT
        ;     ARE OUT OF ORDER UNTIL ALL ELEMENTS BELOW THE
        ;     CENTRAL VALUE ARE AHEAD OF ALL ELEMENTS ABOVE
        ;     THE CENTRAL VALUE.  SUBROUTINE COMPARE
        ;     COMPARES ELEMENTS, SWAP EXCHANGES ELEMENTS,
        ;     PREV MOVES UPPER BOUNDARY DOWN ONE ELEMENT,
        ;     AND NEXT MOVES LOWER BOUNDARY UP ONE ELEMENT.
        ;  4. CHECK IF THE STACK IS ABOUT TO OVERFLOW.  IF IT
        ;     IS, ABORT AND EXIT.
        ;  5. ESTABLISH THE BOUNDARIES FOR THE FIRST PARTITION
        ;     (CONSISTING OF ELEMENTS LESS THAN THE CENTRAL VALUE)
        ;     AND SORT IT RECURSIVELY.
        ;  6. ESTABLISH THE BOUNDARIES FOR THE SECOND PARTITION
        ;     (CONSISTING OF ELEMENTS GREATER THAN THE CENTRAL
        ;     VALUE) AND SORT IT RECURSIVELY.
SORT:
        ;SAVE BASE ADDRESS AND FINAL ADDRESS IN LOCAL STORAGE
        LD      (FIRST),HL      ;SAVE FIRST IN LOCAL AREA
        EX      DE,HL
        LD      (LAST),HL       ;SAVE LAST IN LOCAL AREA

        ;CHECK IF PARTITION CONTAINS 0 OR 1 ELEMENTS
        ; IT DOES IF FIRST IS EITHER LARGER THAN (0)
        ; OR EQUAL TO (1) LAST.
PARTION:
        ;STOP WHEN FIRST >= LAST
        ;DE = ADDRESS OF FIRST
        ;HL = ADDRESS OF LAST
        LD      A,E             ;CALCULATE FIRST - LAST
        SUB     L               ; MUST KEEP BOTH, SO USE 8-BIT SUBTRACT
        LD      A,D
        SBC     A,H
        RET     NC              ;IF DIFFERENCE POSITIVE, RETURN
                                ; THIS PART IS SORTED

        ;USE MEDIAN TO FIND A REASONABLE CENTRAL (PIVOT) ELEMENT
        ;MOVE CENTRAL ELEMENT TO FIRST POSITION
        CALL    MEDIAN          ;SELECT CENTRAL ELEMENT, MOVE IT
                                ; TO FIRST POSITION
        LD      C,0             ;BIT 0 OF REGISTER C = DIRECTION
                                ; IF IT'S 0 THEN DIRECTION IS UP
                                ; ELSE DIRECTION IS DOWN

        ;REORDER ARRAY BY COMPARING OTHER ELEMENTS WITH
        ; CENTRAL ELEMENT.  START BY COMPARING THAT ELEMENT WITH
        ; LAST ELEMENT.  EACH TIME WE FIND AN ELEMENT THAT
        ; BELONGS IN THE FIRST PART (THAT IS, IT IS LESS THAN
        ; THE CENTRAL ELEMENT), SWAP IT INTO THE FIRST PART IF IT
        ; IS NOT ALREADY THERE AND MOVE THE BOUNDARY OF THE
        ; FIRST PART DOWN ONE ELEMENT.  SIMILARLY, EACH TIME WE
        ; FIND  AN ELEMENT THAT BELONGS IN THE SECOND PART (THAT
        ; IS, IT IS GREATER THAN THE CENTRAL ELEMENT), SWAP IT INTO
        ; THE SECOND PART IF IT IS NOT ALREADY THERE AND MOVE
        ; THE BOUNDARY OF THE SECOND PART UP ONE ELEMENT.
        ;ULTIMATELY, THE BOUNDARIES COME TOGETHER
        ; AND THE DIVISION OF THE ARRAY IS THEN COMPLETE
        ;NOTE THAT ELEMENTS EQUAL TO THE CENTRAL ELEMENT ARE NEVER
        ; SWAPPED AND SO MAY END UP IN EITHER PART
PARTLP:
        ;LOOP SORTING UNEXAMINED PART OF THE PARTITION
        ; UNTIL THERE IS NOTHING LEFT IN IT
        LD      A,E             ;LOWER BOUNDARY - UPPER BOUNDARY
        SUB     L               ; MUST KEEP BOTH, SO USE 8-BIT SUBTRACT
        LD      A,D
        SBC     A,H
        JR      NC,DONE         ;EXIT WHEN EVERYTHING EXAMINED

        ;COMPARE NEXT 2 ELEMENTS.  IF OUT OF ORDER, SWAP THEM
        ;AND CHANGE DIRECTION OF SEARCH
        ; IF FIRST > LAST THEN SWAP
        CALL    COMPARE         ;COMPARE ELEMENTS
        JR      C,OK            ;JUMP IF ALREADY IN ORDER
        JR      Z,OK            ; OF IF ELEMENTS EQUAL

        ;ELEMENTS OUT OF ORDER.  SWAP THEM
        CALL    SWAP            ;SWAP ELEMENTS
        INC     C               ;CHANGE DIRECTION

        ;REDUCE SIZE OF UNEXAMINED AREA
        ;IF NEW ELEMENT LESS THAN CENTRAL ELEMENT, MOVE
        ; TOP BOUNDARY DOWN
        ;IF NEW ELEMENT GREATER THAN CENTRAL ELEMENT, MOVE
        ; BOTTOM BOUNDARY UP
        ;IF ELEMENTS EQUAL, CONTINUE IN LATEST DIRECTION
OK:
        BIT     0,C             ;BIT 0 OF C TELLS WHICH WAY TO GO
        JR      Z,UP            ;JUMP IF MOVING UP
        EX      DE,HL
        CALL    NEXT            ;ELSE MOVE TOP BOUNDARY DOWN BY
        EX      DE,HL           ; ONE ELEMENT
        JR      PARTLP
UP:
        CALL    PREV            ;MOVE BOTTOM BOUNDARY UP BY
                                ; ONE ELEMENT
        JR      PARTLP

        ;THIS PARTITION HAS NOW BEEN SUBDIVIDED INTO TWO
        ; PARTITIONS. ONE STARTS AT THE TOP AND ENDS JUST
        ; ABOVE THE CENTRAL ELEMENT.  THE OTHER STARTS
        ; JUST BELOW THE CENTRAL ELEMENT AND CONTINUES
        ; TO THE BOTTOM.  THE CENTRAL ELEMENT IS NOW IN
        ; ITS PROPER SORTED POSITION AND NEED NOT BE
        ; INCLUDED IN EITHER PARTITION
```

```
DONE:
        ;FIRST CHECK WHETHER STACK MIGHT OVERFLOW
        ;IF IT IS GETTING TOO CLOSE TO THE BOTTOM, ABORT
        ; THE PROGRAM AND EXIT
        LD      HL,(STKBTM)     ;CALCULATE STKBTM - SP
        OR      A               ;CLEAR CARRY
        SBC     HL,SP
        JR      NC,ABORT        ;EXIT IF STACK TOO LARGE

        ;ESTABLISH BOUNDARIES FOR FIRST (LOWER) PARTITION
        ;LOWER BOUNDARY IS SAME AS BEFORE
        ;UPPER BOUNDARY IS ELEMENT JUST BELOW CENTRAL ELEMENT
        ;THEN RECURSIVELY QUICKSORT FIRST PARTITION
        PUSH    DE              ;SAVE ADDRESS OF CENTRAL ELEMENT
        LD      HL,(LAST)
        PUSH    HL              ;SAVE ADDRESS OF LAST
        EX      DE,HL
        CALL    PREV            ;CALCULATE LAST FOR FIRST PART
        EX      DE,HL
        LD      HL,(FIRST)      ;FIRST IS SAME AS BEFORE
        CALL    SORT            ;QUICKSORT FIRST PART

        ;ESTABLISH BOUNDARIES FOR SECOND (UPPER) PARTITION
        ;UPPER BOUNDARY IS SAME AS BEFORE
        ;LOWER BOUNDARY IS ELEMENT JUST ABOVE CENTRAL ELEMENT
        ;THEN RECURSIVELY QUICKSORT SECOND PARTITION
        POP     DE              ;LAST IS SAME AS BEFORE
        POP     HL              ;CALCULATE FIRST FOR SECOND PART
        CALL    NEXT
        CALL    SORT            ;QUICKSORT SECOND PART
        OR      A               ;CARRY = 0 FOR NO ERRORS
        RET

        ;ERROR EXIT - SET CARRY
ABORT:  LD      SP,(OLDSP)      ;TOP OF STACK IS ORIGINAL
                                ; RETURN ADDRESS
        SCF                     ;INDICATE ERROR IN SORT
        RET                     ;RETURN TO ORIGINAL CALLER

        ;******************************
        ;ROUTINE: MEDIAN
        ;PURPOSE: DETERMINE WHICH VALUE IN A PARTITION
        ;         SHOULD BE USED AS THE CENTRAL ELEMENT OR PIVOT
        ;ENTRY: DE = ADDRESS OF FIRST VALUE
        ;       HL = ADDRESS OF LAST VALUE
        ;EXIT:  DE IS ADDRESS OF CENTRAL ELEMENT
        ;REGISTERS USED: AF,BC,DE
        ;******************************
MEDIAN:
        ;DETERMINE ADDRESS OF MIDDLE ELEMENT
        ; MIDDLE := ALIGNED (FIRST + LAST) DIV 2
        LD      A,L             ;ADD ADDRESSES OF FIRST, LAST
        ADD     A,E             ;MUST KEEP BOTH, SO USE 8-BIT
        LD      C,A             ; ADD INSTEAD OF 16-BIT
        LD      A,H
        ADC     A,D
        LD      B,A
        RR      B                ;DIVIDE SUM BY 2, BYTE AT A TIME
        RR      C
        RES     0,C              ;CLEAR BIT 0 FOR ALIGNMENT
        BIT     0,E              ;ALIGN MIDDLE TO BOUNDARY OF FIRST
        JR      Z,MED1           ; JUMP IF BIT 0 OF FIRST IS 0
        INC     C                ; ELSE MAKE BIT 0 OF MIDDLE 1

        ;DETERMINE WHICH OF FIRST, MIDDLE, LAST IS
        ; MEDIAN (CENTRAL VALUE)
        ;COMPARE FIRST AND MIDDLE
MED1:
        PUSH    HL               ;SAVE LAST
        LD      L,C
        LD      H,B
        CALL    COMPARE          ;COMPARE FIRST AND MIDDLE
        POP     HL               ;RESTORE LAST
        JR      NC,MIDD1         ;JUMP IF FIRST >= MIDDLE

        ;WE KNOW (MIDDLE > FIRST)
        ; SO COMPARE MIDDLE AND LAST
        PUSH    DE               ;SAVE FIRST
        LD      E,C
        LD      D,B
        CALL    COMPARE          ;COMPARE MIDDLE AND LAST
        POP     DE               ;RESTORE LAST
        JR      C,SWAPMF         ;JUMP IF LAST >= MIDDLE
        JR      Z,SWAPMF         ; MIDDLE IS MEDIAN

        ;WE KNOW (MIDDLE > FIRST) AND (MIDDLE > LAST)
        ; SO COMPARE FIRST AND LAST
        CALL    COMPARE          ;COMPARE FIRST AND LAST
        RET     NC               ;RETURN IF LAST >= FIRST
                                 ; FIRST IS MEDIAN
        JR      SWAPLF           ;ELSE LAST IS MEDIAN

        ;WE KNOW (FIRST >= MIDDLE)
        ; SO COMPARE FIRST AND LAST
MIDD1:
        CALL    COMPARE          ;COMPARE LAST AND FIRST
        RET     C                ;RETURN IF LAST >= FIRST
        RET     Z                ; FIRST IS MEDIAN

        ;WE KNOW (FIRST >= MIDDLE) AND (FIRST > LAST)
        ; SO COMPARE MIDDLE AND LAST
        PUSH    DE               ;SAVE FIRST
        LD      E,C              ;DE = MIDDLE
        LD      D,B
        CALL    COMPARE          ;COMPARE MIDDLE AND LAST
        POP     DE               ;RESTORE FIRST
        JR      C,SWAPLF         ;JUMP IF LAST > MIDDLE
                                 ; LAST IS MEDIAN
        ;MIDDLE IS MEDIAN, SWAP IT WITH FIRST
SWAPMF:
        PUSH    HL                      ;SAVE LAST
        LD      L,C                     ;HL = ADDRESS OF MIDDLE
        LD      H,B
        CALL    SWAP                    ;SWAP MIDDLE, FIRST
        POP     HL                      ;RESTORE LAST
        RET

        ;LAST IS MEDIAN, SWAP IT WITH FIRST
SWAPLF:
        CALL    SWAP                    ;SWAP FIRST AND LAST
        RET

        ;*******************************
        ;ROUTINE: NEXT
        ;PURPOSE: MAKE HL POINT TO NEXT ELEMENT
        ;ENTRY: HL = ADDRESS OF CURRENT ELEMENT
        ;EXIT:  HL = ADDRESS OF NEXT ELEMENT
        ;REGISTERS USED: HL
        ;*******************************

NEXT:
        INC     HL                      ;INCREMENT TO NEXT ELEMENT
        INC     HL
        RET

        ;*******************************
        ;ROUTINE: PREV
        ;PURPOSE: MAKE HL POINT TO PREVIOUS ELEMENT
        ;ENTRY: HL = ADDRESS OF CURRENT ELEMENT
        ;EXIT:  HL = ADDRESS OF PREVIOUS ELEMENT
        ;REGISTERS USED: HL
        ;*******************************
PREV:
        DEC     HL                      ;DECREMENT TO PREVIOUS ELEMENT
        DEC     HL
        RET

        ;*******************************
        ;ROUTINE: COMPARE
        ;PURPOSE: COMPARE DATA ITEMS POINTED TO BY DE AND HL
        ;ENTRY: DE = ADDRESS OF DATA ELEMENT 1
        ;       HL = ADDRESS OF DATA ELEMENT 2
        ;EXIT:  IF ELEMENT 1 > ELEMENT 2 THEN
        ;         C = 0
        ;         Z = 0
        ;       IF ELEMENT 1 < ELEMENT 2 THEN
        ;         C = 1
        ;         Z = 0
        ;       IF ELEMENT 1 = ELEMENT 2 THEN
        ;         C = 0
        ;         Z = 1
        ;REGISTERS USED: AF
        ;*******************************
```

```
COMPARE:
        INC     HL              ;POINT TO HIGH BYTES
        INC     DE
        LD      A,(DE)
        CP      (HL)            ;COMPARE HIGH BYTES
        DEC     DE              ;POINT TO LOW BYTES
        DEC     HL
        RET     NZ              ;RETURN IF HIGH BYTES NOT EQUAL
        LD      A,(DE)          ;OTHERWISE, COMPARE LOW BYTES
        CP      (HL)
        RET

        ;********************************
        ;ROUTINE: SWAP
        ;PURPOSE: SWAP ELEMENTS POINTED TO BY DE,HL
        ;ENTRY: DE = ADDRESS OF ELEMENT 1
        ;       HL = ADDRESS OF ELEMENT 2
        ;EXIT:  ELEMENTS SWAPPED
        ;REGISTERS USED: AF,B
        ;*********************************
SWAP:
        ;SWAP LOW BYTES
        LD      B,(HL)          ;GET ELEMENT 2
        LD      A,(DE)          ;GET ELEMENT 1
        LD      (HL),A          ;STORE NEW ELEMENT 2
        LD      A,B
        LD      (DE),A          ;STORE NEW ELEMENT 1
        INC     HL
        INC     DE

        ;SWAP HIGH BYTES
        LD      B,(HL)          ;GET ELEMENT 2
        LD      A,(DE)          ;GET ELEMENT 1
        LD      (HL),A          ;STORE NEW ELEMENT 2
        LD      A,B
        LD      (DE),A          ;STORE NEW ELEMENT 1
        DEC     HL
        DEC     DE
        RET

        ;DATA SECTION
FIRST:  DS      2               ;POINTER TO FIRST ELEMENT OF PART
LAST:   DS      2               ;POINTER TO LAST ELEMENT OF PART
STKBTM: DS      2               ;THRESHOLD FOR STACK OVERFLOW
OLDSP:  DS      2               ;POINTER TO ORIGINAL RETURN ADDRESS


;                                                                       ;
;                                                                       ;
;       SAMPLE EXECUTION:                                               ;
;                                                                       ;
;                                                                       ;
```

```
SC9F:
        ;SORT AN ARRAY BETWEEN BEGBUF (FIRST ELEMENT)
        ; AND ENDBUF (LAST ELEMENT)
        ;START STACK AT 5000 HEX AND ALLOW IT TO EXPAND
        ; AS FAR AS 4F00 HEX
        LD      SP,5000H        ;SET UP A STACK AREA
        LD      BC,4F00H        ;BC = LOWEST AVAILABLE STACK ADDRESS
        LD      HL,BEGBUF       ;HL = ADDRESS OF FIRST ELEMENT OF ARRAY
        LD      DE,ENDBUF       ;DE = ADDRESS OF LAST ELEMENT OF ARRAY

        CALL    QSORT           ;SORT
                                ;RESULT FOR TEST DATA IS
                                ; 0,1,2,3, ... ,14,15

        JR      SC9F            ;LOOP FOR MORE TESTS

        ;DATA SECTION
BEGBUF: DW      15
        DW      14
        DW      13
        DW      12
        DW      11
        DW      10
        DW      9
        DW      8
        DW      7
        DW      6
        DW      5
        DW      4
        DW      3
        DW      2
        DW      1
ENDBUF: DW      0

        END
```

RAM Test (RAMTST) 9G

Tests a RAM area specified by a base address and a length in bytes. Writes the values 0, FF_{16}, AA_{16} (10101010_2), and 55_{16} (01010101_2) into each byte and checks whether they can be read back correctly. Places 1 in each bit position of each byte and checks whether it can be read back correctly with all other bits cleared. Clears the Carry flag if all tests run properly. If it finds an error, it exits immediately, setting the Carry flag and returning the test value and the address at which the error occurred.

Procedure: The program performs the single value checks (with 0, FF_{16}, AA_{16}, and 55_{16}) by first filling the memory area and then comparing each byte with the specified value. Filling the entire area first should provide enough delay between writing and reading to detect a failure to retain data (perhaps caused by improperly designed refresh circuitry). The program then performs the walking bit test, starting with bit 7; here it writes the data into memory and attempts to read it back immediately for a comparison.

Registers Used: AF, BC, DE, HL

Execution Time: Approximately 633 cycles per byte tested plus 663 cycles overhead

Program Size: 82 bytes

Data Memory Required: None

Special Cases:

1. An area size of 0000_{16} causes an immediate exit with no memory tested. The Carry flag is cleared to indicate no errors.
2. Since the routine changes all bytes in the tested area, using it to test an area that includes itself will have unpredictable results.

Note that Case 1 means this routine cannot be asked to test the entire memory. Such a request would be meaningless anyway since it would require the routine to test itself.

3. Testing a ROM causes a return with an error indication after the first occasion on which the test value differs from the memory's contents.

Entry Conditions

Base address of test area in HL
Size of test area in bytes in DE

Exit Conditions

If an error is found:
  Carry = 1
  Address containing error in HL
  Test value in A
If no error is found:
  Carry = 0
  All bytes in test area contain 0

Example

1. Data: Base address = 0380_{16}
   Length (size) of area = 0200_{16}

   Result: Area tested is the 0200_{16} bytes starting at address 0380_{16}, that is, addresses 0380_{16} through $057F_{16}$. The order of the tests is

1. Write and read 0
2. Write and read FF_{16}
3. Write and read AA_{16} (10101010_2)
4. Write and read 55_{16} (01010101_2)
5. Walking bit test, starting with 1 in bit 7. That is, start with 10000000_2 (80_{16}) and move the 1 one position right for each subsequent test of a byte.

```
;                                                                ;
;                                                                ;
;                                                                ;
;                                                                ;
;       Title             RAM test                               ;
;       Name:             RAMTST                                 ;
;                                                                ;
;                                                                ;

;                                                                ;
;       Purpose:          Test a RAM (read/write memory) area    ;
;                           1) Write all 0 and test              ;
;                           2) Write all FF hex and test         ;
;                           3) Write all AA hex and test         ;
;                           4) Write all 55 hex and test         ;
;                           5) Shift a single 1 through each bit,;
;                              while clearing all other bits     ;
;                                                                ;
;                           If the program finds an error, it exits ;
;                           immediately with the Carry flag set and ;
;                           indicates where the error occurred and ;
;                           what value it used in the test.      ;
;                                                                ;
;       Entry:            Register pair HL = Base address of test area ;
;                         Register pair DE = Size of area in bytes ;
;                                                                ;
;       Exit:             If there are no errors then            ;
;                           Carry flag = 0                       ;
;                           test area contains 0 in all bytes    ;
;                         else                                   ;
;                           Carry flag = 1                       ;
;                           Register pair HL = Address of error  ;
;                           Register A = Expected value          ;
;                                                                ;
;       Registers used: AF,BC,DE.HL                              ;
;                                                                ;
;       Time:             Approximately 633 cycles per byte plus ;
;                         663 cycles overhead                    ;
;                                                                ;
;       Size:             Program 82 bytes                       ;
;                                                                ;
;                                                                ;
```

```
RAMTST:
        ;EXIT WITH NO ERRORS IF AREA SIZE IS 0
        LD      A,D                 ;TEST AREA SIZE
        OR      E
        RET     Z                   ;EXIT WITH NO ERRORS IF SIZE IS ZERO
        LD      B,D                 ;BC = AREA SIZE
        LD      C,E

        ;FILL MEMORY WITH 0 AND TEST
        SUB     A
        CALL    FILCMP
        RET     C                   ;EXIT IF ERROR FOUND

        ;FILL MEMORY WITH FF HEX (ALL 1'S) AND TEST
        LD      A,0FFH
        CALL    FILCMP
        RET     C                   ;EXIT IF ERROR FOUND

        ;FILL MEMORY WITH AA HEX (ALTERNATING 1'S AND 0'S) AND TEST
        LD      A,0AAH
        CALL    FILCMP
        RET     C                   ;EXIT IF ERROR FOUND

        ;FILL MEMORY WITH 55 HEX (ALTERNATING 0'S AND 1'S) AND TEST
        LD      A,55H
        CALL    FILCMP
        RET     C                   ;EXIT IF ERROR FOUND

        ;PERFORM WALKING BIT TEST.  PLACE A 1 IN BIT 7 AND
        ; SEE IF IT CAN BE READ BACK.  THEN MOVE THE 1 TO
        ; BITS 6, 5, 4, 3, 2, 1, AND 0 AND SEE IF IT CAN
        ; BE READ BACK
WLKLP:
        LD      A,10000000B         ;MAKE BIT 7 1, ALL OTHER BITS 0
WLKLP1:
        LD      (HL),A              ;STORE TEST PATTERN IN MEMORY
        CP      (HL)                ;TRY TO READ IT BACK
        SCF                         ;SET CARRY IN CASE OF ERROR
        RET     NZ                  ;RETURN IF ERROR
        RRCA                        ;ROTATE PATTERN TO MOVE 1 RIGHT
        CP      10000000B
        JR      NZ,WLKLP1           ;CONTINUE UNTIL 1 IS BACK IN BIT 7
        LD      (HL),0              ;CLEAR BYTE JUST CHECKED
        INC     HL
        DEC     BC                  ;DECREMENT AND TEST 16-BIT COUNTER
        LD      A,B
        OR      C
        JR      NZ,WLKLP            ;CONTINUE UNTIL MEMORY TESTED
        RET                         ;NO ERRORS (NOTE OR C CLEARS CARRY)


        ;**********************************
        ;ROUTINE: FILCMP
        ;PURPOSE: FILL MEMORY WITH A VALUE AND TEST
        ;         THAT IT CAN BE READ BACK
        ;ENTRY:  A = TEST VALUE
        ;        HL = BASE ADDRESS
        ;        BC = SIZE OF AREA IN BYTES
        ;EXIT:   IF NO ERRORS THEN
        ;          CARRY FLAG IS 0
        ;        ELSE
        ;          CARRY FLAG IS 1
        ;          HL = ADDRESS OF ERROR
        ;          DE = BASE ADDRESS
        ;          BC = SIZE OF AREA IN BYTES
        ;          A = TEST VALUE
        ;REGISTERS USED: AF,BC,DE,HL
        ;************************************
FILCMP:
        PUSH    HL              ;SAVE BASE ADDRESS
        PUSH    BC              ;SAVE SIZE OF AREA
        LD      E,A             ;SAVE TEST VALUE
        LD      (HL),A          ;STORE TEST VALUE IN FIRST BYTE
        DEC     BC              ;REMAINING AREA = SIZE - 1
        LD      A,B             ;CHECK IF ANYTHING IN REMAINING AREA
        OR      C
        LD      A,E             ;RESTORE TEST VALUE
        JR      Z,COMPARE       ;BRANCH IF AREA WAS ONLY 1 BYTE

        ;FILL REST OF AREA USING BLOCK MOVE
        ; EACH ITERATION MOVES TEST VALUE TO NEXT HIGHER ADDRESS
        LD      D,H             ;DESTINATION IS ALWAYS SOURCE + 1
        LD      E,L
        INC     DE
        LDIR                    ;FILL MEMORY

        ;NOW THAT MEMORY HAS BEEN FILLED, TEST TO SEE IF
        ; EACH BYTE CAN BE READ BACK CORRECTLY
COMPARE:
        POP     BC              ;RESTORE SIZE OF AREA
        POP     HL              ;RESTORE BASE ADDRESS
        PUSH    HL              ;SAVE BASE ADDRESS
        PUSH    BC              ;SAVE SIZE OF VALUE

        ;COMPARE MEMORY AND TEST VALUE
CMPLP:
        CPI
        JR      NZ,CMPER        ;JUMP IF NOT EQUAL
        JP      PE,CMPLP        ;CONTINUE THROUGH ENTIRE AREA
                                ; NOTE CPI CLEARS P/V FLAG IF IT
                                ; DECREMENTS BC TO 0

        ;NO ERRORS FOUND, SO CLEAR CARRY
        POP     BC              ;BC = SIZE OF AREA
        POP     HL              ;HL = BASE ADDRESS
        OR      A               ;CLEAR CARRY, INDICATING NO ERRORS
        RET

        ;ERROR EXIT, SET CARRY
        ;HL = ADDRESS OF ERROR
        ;A = TEST VALUE
```

```
CMPER:
        POP     BC                  ;DE = SIZE OF AREA
        POP     DE                  ;BC = BASE ADDRESS
        SCF                         ;SET CARRY, INDICATING AN ERROR
        RET


;                                                                       ;
;                                                                       ;
;       SAMPLE EXECUTION                                                ;
;                                                                       ;
;                                                                       ;

SC9G:
        ;TEST RAM FROM 2000 HEX THROUGH 300F HEX
        ;  SIZE OF AREA = 1010 HEX BYTES
        LD      HL,2000H            ;HL = BASE ADDRESS
        LD      DE,1010H            ;DE = NUMBER OF BYTES
        CALL    RAMTST              ;TEST MEMORY
                                    ;CARRY FLAG SHOULD BE 0

        JR      SC9G                ;LOOP FOR MORE TESTING

        END
```

Jump Table (JTAB) 9H

Transferring control to an address selected from a table according to an index. The addresses are stored in the usual Z80 format (less significant byte first), starting at address JMPTAB. The size of the table (number of addresses) is a constant, LENSUB, which must be less than or equal to 128. If the index is greater than or equal to LENSUB, the program returns control immediately with the Carry flag set to 1.

Procedure: The program first checks if the index is greater than or equal to the size of the table (LENSUB). If it is, the program returns control with the Carry flag set. If it is not, the program obtains the starting address of the appropriate subroutine from the table and jumps to it.

Registers Used: AF

Execution Time: 117 cycles overhead, besides the time required to execute the actual subroutine

Program Size: 21 bytes plus 2 * LENSUB bytes for the table of starting addresses, where LENSUB is the number of subroutines

Data Memory Required: None

Special Case: Entry with an index greater than or equal to LENSUB causes an immediate exit with the Carry flag set to 1.

Entry Conditions

Index in A

Exit Conditions

If (A) is greater than LENSUB, an immediate return with Carry = 1. Otherwise, control is transferred to the appropriate subroutine as if an indexed call had been performed. The return address remains at the top of the stack.

Example

1.	Data:	LENSUB (size of subroutine table) = 03 Table consists of addresses SUB0, SUB1, and SUB2. Index = (A) = 02
	Result:	Control transferred to address SUB2 (PC = SUB2)

```
;                                                                  ;
;                                                                  ;
;                                                                  ;
;                                                                  ;
;       Title           Jump table                                 ;
;       Name:           JTAB                                       ;
;                                                                  ;
;                                                                  ;


;                                                                  ;
;       Purpose:        Given an index, jump to the subroutine with ;
;                       that index in a table.                     ;
;                                                                  ;
;       Entry:          Register A is the subroutine number (0 to  ;
;                                  LENSUB-1, the number of subroutines) ;
;                                  LENSUB must be less than or equal to ;
;                                  128.                            ;
;                                                                  ;
;       Exit:           If the routine number is valid then        ;
;                         execute the routine                      ;
;                       else                                       ;
;                         Carry flag = 1                           ;
;                                                                  ;
;       Registers used: AF                                         ;
;                                                                  ;
;       Time:           117 cycles plus execution time of subroutine ;
;                                                                  ;
;       Size:           Program 21 bytes plus size of table (2*LENSUB) ;
;                                                                  ;
;                                                                  ;

        ;EXIT WITH CARRY SET IF ROUTINE NUMBER IS INVALID
        ;THAT IS, IF IT IS TOO LARGE FOR TABLE (>LENSUB - 1)
JTAB:
        CP      LENSUB          ;COMPARE ROUTINE NUMBER, TABLE SIZE
        CCF                     ;COMPLEMENT CARRY FOR ERROR INDICATOR
        RET     C               ;RETURN IF ROUTINE NUMBER TOO LARGE
                                ; WITH CARRY SET

        ;INDEX INTO TABLE OF WORD-LENGTH ADDRESSES
        ;LEAVE REGISTER PAIRS UNCHANGED SO THEY CAN BE USED
        ;  FOR PASSING PARAMETERS
        PUSH    HL              ;SAVE HL
        ADD     A,A             ;DOUBLE INDEX FOR WORD-LENGTH ENTRIES
        LD      HL,JMPTAB       ;INDEX INTO TABLE USING 8-BIT
        ADD     A,L             ; ADDITION TO AVOID DISTURBING
        LD      L,A             ; ANOTHER REGISTER PAIR
        LD      A,0
        ADC     A,H
        LD      H,A             ;ACCESS ROUTINE ADDRESS

        ;OBTAIN ROUTINE ADDRESS FROM TABLE AND TRANSFER
        ;  CONTROL TO IT, LEAVING ALL REGISTER PAIRS UNCHANGED
        LD      A,(HL)          ;MOVE ROUTINE ADDRESS TO HL
        INC     HL
        LD      H,(HL)
        LD      L,A
        EX      (SP),HL         ;RESTORE OLD HL, PUSH ROUTINE ADDRESS
        RET                     ;JUMP TO ROUTINE

LENSUB  EQU     3               ;NUMBER OF SUBROUTINES IN TABLE

JMPTAB:                         ;JUMP TABLE
        DW      SUB0            ;ROUTINE 0
        DW      SUB1            ;ROUTINE 1
        DW      SUB2            ;ROUTINE 2

        ;THREE TEST SUBROUTINES FOR JUMP TABLE
SUB0:
        LD      A,1             ;TEST ROUTINE 0 SETS (A) = 1
        RET

SUB1:
        LD      A,2             ;TEST ROUTINE 1 SETS (A) = 2
        RET

SUB2:
        LD      A,3             ;TEST ROUTINE 2 SETS (A) = 3
        RET


;                                                                       ;
;                                                                       ;
;       SAMPLE EXECUTION:                                               ;
;                                                                       ;
;                                                                       ;

SC9H:
        SUB     A               ;EXECUTE ROUTINE 0
        CALL    JTAB            ; AFTER EXECUTION, (A) = 1
        LD      A,1             ;EXECUTE ROUTINE 1
        CALL    JTAB            ; AFTER EXECUTION, (A) = 2
        LD      A,2             ;EXECUTE ROUTINE 2
        CALL    JTAB            ; AFTER EXECUTION, (A) = 3
        LD      A,3             ;EXECUTE ROUTINE 3
        CALL    JTAB            ; AFTER EXECUTION, CARRY = 1

        JR      SC9H            ;LOOP FOR MORE TESTS

        END
```

Read a Line from a Terminal (RDLINE) 10A

Reads a line of ASCII characters ending with a carriage return and saves them in a buffer. Defines the control characters Control H (08 hex), which deletes the latest character, and Control X (18 hex), which deletes the entire line. Sends a bell character (07 hex) to the terminal if the buffer overflows. Echoes each character placed in the buffer. Echoes non-printable characters as an up arrow or caret (^) followed by the printable equivalent (see Table 10-1). Sends a new line sequence (typically carriage return, line feed) to the terminal before exiting.

RDLINE assumes the following system-dependent subroutines:

1. RDCHAR reads a character from the terminal and puts it in the accumulator.

2. WRCHAR sends the character in the accumulator to the terminal.

3. WRNEWL sends a new line sequence to the terminal.

These subroutines are assumed to change all user registers.

RDLINE is an example of a terminal input handler. The control characters and I/O subroutines in a real system will, of course, be computer-dependent. A specific example in the listing is for a computer running the CP/M operating system with a standard Basic Disk Operating System (BDOS) accessed by calling memory address 0005_{16}. Table 10-2 lists commonly used CP/M BDOS functions. For more information on CP/M, see *Osborne CP/M User Guide*, Second Edition by Thom Hogan (Berkeley: Osborne/McGraw-Hill, 1982).

Procedure: The program starts the loop by reading a character. If the character is a carriage return, the program sends a new line sequence to the terminal and exits. Otherwise, it checks for the special characters Control H and Control X. If the buffer is not empty, Control H makes the program decrement the buffer pointer and character count by 1 and send a backspace string (cursor left, space, cursor left) to the terminal. Control X makes the program delete characters until it empties the buffer.

If the character is not special, the program determines whether the buffer is full. If it is, the program sends a bell character to the terminal. If not, the program stores the character in the buffer, echoes it to the terminal, and increments the character count and buffer pointer.

Before echoing a character or deleting one from the display, the program must determine whether the character is printable. If it is not (that is, it is a non-printable ASCII control character), the program must display or delete two characters, the control indicator (up arrow or caret) and the printable equivalent (see Table 10-1). Note, however, that the character is stored in its non-printable form.

Registers Used: AF, BC, DE, HL

Execution Time: Approximately 162 cycles to place an ordinary character in the buffer, not including the execution time of RDCHAR or WRCHAR

Program Size: 148 bytes

Data Memory Required: None

Special Cases:

1. Typing Control H (delete one character) or Control X (delete the entire line) when the buffer is empty has no effect.

2. The program discards an ordinary character received when the buffer is full, and sends a bell character to the terminal (ringing the bell).

Table 10-1: ASCII Control Characters and Printable Equivalents

Name	Hex Value	Printable Equivalent	Name	Hex Value	Printable Equivalent
NUL	00	Control @	DLE	10	Control P
SOH	01	Control A	DC1	11	Control Q
STX	02	Control B	DC2	12	Control R
ETX	03	Control C	DC3	13	Control S
EOT	04	Control D	DC4	14	Control T
ENQ	05	Control E	NAK	15	Control U
ACK	06	Control F	SYN	16	Control V
BEL	07	Control G	ETB	17	Control W
BS	08	Control H	CAN	18	Control X
HT	09	Control I	EM	19	Control Y
LF	0A	Control J	SUB	1A	Control Z
VT	0B	Control K	ESC	1B	Control [
FF	0C	Control L	FS	1C	Control ◄
CR	0D	Control M	GS	1D	Control]
SO	0E	Control N	RS	1E	Control ^
SI	0F	Control O	VS	1F	Control _

Table 10-2: BDOS Functions for CP/M 2.0

Function Number (Decimal in Register C)	Function Name	Input Parameters	Output Parameters
0	System Reset	None	None
1	Console Input	None	A = ASCII character
2	Console Output	E = ASCII character	None
3	Reader Input	None	A = ASCII character
4	Punch Output	E = ASCII character	None
5	List Output	E = ASCII character	None
6	Direct Console Input	E = FF_{16}	A = ASCII character or 00 if no character is available
6	Direct Console Output	E = ASCII character	None
7	Get I/O Byte	None	A = IOBYTE
8	Set I/O Byte	E = IOBYTE	None
9	Print String	DE = String Address	None
10	Read Console Buffer	DE = Buffer Address	(Data in buffer)
11	Get Console Status	None	A = 00 (no character) or A = FF_{16} (character ready)

Entry Conditions

Base address of buffer in HL
Length (size) of buffer in bytes in A

Exit Conditions

Number of characters in the buffer in A

Examples

1. Data: Line from keyboard is 'ENTERcr'
   Result: Character count = 5 (line length)
   Buffer contains 'ENTER'
   'ENTER' is sent to terminal, followed by a new line sequence (typically either carriage return, line feed or just carriage return).
   Note that the 'cr' (carriage return) character does not appear in the buffer.

2. Data: Line from keyboard is 'DMcontrolHNcontrol XENTETcontrolHRcr'
   Result: Character count = 5 (length of final line)
   Buffer contains 'ENTER'
   'DMBackspaceStringNBackspaceStringBackspaceStringENTETBackspaceStringR' is sent to terminal, followed by a new line sequence. The Backspace String deletes a character from the screen and moves the cursor left one space.
   The sequence of operations is as follows:

Character Typed	Initial Buffer	Final Buffer	Sent to Terminal
D	Empty	'D'	D
M	'D'	'DM'	M
Control H	'DM'	'D'	Backspace string
N	'D'	'DN'	N
Control X	'DN'	Empty	2 Backspace strings
E	Empty	'E'	E
N	'E'	'EN'	N
T	'EN'	'ENT'	T
E	'ENT'	'ENTE'	E
T	'ENTE'	'ENTET'	T
Control H	'ENTET'	'ENTE'	Backspace string
R	'ENTE'	'ENTER'	R
cr	'ENTER'	'ENTER'	New line string

What has happened is

a. The operator types 'D', 'M'.

b. The operator sees that 'M' is wrong (it should be 'N'), types Control H to delete it, and types 'N'.

c. The operator then sees that the initial 'D' is also wrong (it should be 'E'). Since the error is not in the latest character, the operator types Control X to delete the entire line, and then types 'ENTET'.

d. The operator sees that the second 'T' is wrong (it should be 'R'), types Control H to delete it, and types 'R'.

e. The operator types a carriage return to end the line.

```
;                                                                   ;
;                                                                   ;
;                                                                   ;
;                                                                   ;
;       Title           Read line                                   ;
;       Name:           RDLINE                                      ;
;                                                                   ;
;                                                                   ;


;                                                                   ;
;       Purpose:        Read characters from CP/M BDOS CON: device  ;
;                       until carriage return encountered. All control ;
;                       characters but the following are placed in the ;
;                       buffer and displayed as the equivalent printable;
;                       ASCII character preceded by a caret.        ;
;                         Control H: delete last character          ;
;                         Control X: delete entire line             ;
;                                                                   ;
;       Entry:          Register pair HL = Base address of buffer   ;
;                       Register A = Length of buffer in bytes      ;
;                                                                   ;
;       Exit:           Register A = Number of characters in buffer ;
;                                                                   ;
;       Registers used: AF,BC,DE,HL                                 ;
;                                                                   ;
;       Time:           Not applicable                              ;
;                                                                   ;
;       Size:           Program 148 bytes                           ;
;                                                                   ;
;                                                                   ;

        ;EQUATES
BELL    EQU     07H     ;BELL CHARACTER (RINGS BELL ON TERMINAL)
BSKEY   EQU     08H     ;BACKSPACE KEYBOARD CHARACTER
CR      EQU     0DH     ;CARRIAGE RETURN FOR CONSOLE
CRKEY   EQU     0DH     ;CARRIAGE RETURN KEYBOARD CHARACTER
CSRLFT  EQU     08H     ;MOVE CURSOR LEFT FOR CONSOLE
DELKEY  EQU     18H     ;DELETE LINE KEYBOARD CHARACTER
LF      EQU     0AH     ;LINE FEED FOR CONSOLE
SPACE   EQU     20H     ;SPACE CHARACTER
UPARRW  EQU     5EH     ;UP ARROW OR CARET USED AS CONTROL INDICATOR

BDOS    EQU     0005H   ;BDOS ENTRY POINT
DIRIO   EQU     6       ;BDOS DIRECT I/O FUNCTION
PSTRG   EQU     9       ;BDOS PRINT STRING FUNCTION
STERM   EQU     '$'     ;CP/M STRING TERMINATOR

RDLINE:
        LD      C,A             ;C = BUFFER LENGTH
                                ;HL = BUFFER POINTER

        ;INITIALIZE CHARACTER COUNT TO ZERO
INIT:
        LD      B,0             ;CHARACTER COUNT = 0

        ;READ CHARACTERS UNTIL A CARRIAGE RETURN IS TYPED
```

```
RDLOOP:
        CALL    RDCHAR          ;READ CHARACTER FROM KEYBOARD - NO ECHO

        ;CHECK FOR CARRIAGE RETURN, EXIT IF FOUND
        CP      CRKEY
        JR      Z,EXITRD        ;END OF LINE IF CARRIAGE RETURN

        ;CHECK FOR BACKSPACE AND DELETE CHARACTER IF FOUND
        CP      BSKEY
        JR      NZ,RDLP1        ;BRANCH IF NOT BACKSPACE
        CALL    BACKSP          ;IF BACKSPACE, DELETE ONE CHARACTER
        JR      RDLOOP          ; THEN START READ LOOP AGAIN

        ;CHECK FOR DELETE LINE CHARACTER AND EMPTY BUFFER IF FOUND
RDLP1:
        CP      DELKEY
        JR      NZ,RDLP2        ;BRANCH IF NOT DELETE LINE
DEL1:
        CALL    BACKSP          ;DELETE A CHARACTER
        JR      NZ,DEL1         ;CONTINUE UNTIL BUFFER EMPTY
        JR      RDLOOP          ;THIS ACTUALLY BACKS UP OVER EACH
                                ; CHARACTER RATHER THAN JUST MOVING
                                ; UP A LINE

        ;NOT A SPECIAL CHARACTER
        ; CHECK IF BUFFER IS FULL
        ; IF FULL, RING BELL AND CONTINUE
        ; IF NOT FULL, STORE CHARACTER AND ECHO
RDLP2:
        LD      E,A             ;SAVE CHARACTER
        LD      A,B             ;IS BUFFER FULL?
        CP      C               ; COMPARE COUNT AND BUFFER LENGTH
        JR      C,STRCH         ;JUMP IF BUFFER NOT FULL
        LD      A,BELL          ;FULL, RING THE TERMINAL'S BELL
        CALL    WRCHAR
        JR      RDLOOP          ;THEN CONTINUE THE READ LOOP

        ;BUFFER NOT FULL, STORE CHARACTER
STRCH:
        LD      A,E             ;GET CHARACTER BACK
        LD      (HL),A          ;STORE CHARACTER IN BUFFER
        INC     HL              ;INCREMENT BUFFER POINTER
        INC     B               ;INCREMENT CHARACTER COUNT

        ;IF CHARACTER IS CONTROL, THEN OUTPUT
        ; UP ARROW FOLLOWED BY PRINTABLE EQUIVALENT
        CP      SPACE           ;CONTROL IF LESS THAN SPACE (20 HEX)
        JR      NC,PRCH         ;JUMP IF A PRINTABLE CHARACTER
        PUSH    AF              ;SAVE CHARACTER
        LD      A,UPARRW        ;WRITE UP ARROW OR CARET
        CALL    WRCHAR
        POP     AF              ;RECOVER CHARACTER
        ADD     A,40H           ;CHANGE TO PRINTABLE FORM
PRCH:   CALL    WRCHAR          ;ECHO CHARACTER TO TERMINAL
        JR      RDLOOP          ;THEN CONTINUE READ LOOP
        ;EXIT
        ;SEND NEW LINE SEQUENCE (USUALLY CR,LF) TO TERMINAL
        ;GET LENGTH OF LINE
EXITRD:
        CALL    WRNEWL          ;SEND NEW LINE SEQUENCE
        LD      A,B             ;LINE LENGTH = CHARACTER COUNT
        RET


        ;*********************************************
        ;ROUTINE: RDCHAR
        ;PURPOSE: READ CHARACTER BUT DO NOT ECHO TO CONSOLE
        ;ENTRY: NONE
        ;EXIT:  REGISTER A = CHARACTER
        ;REGISTERS USED: ALL EXCEPT BC, HL
        ;****************************************
RDCHAR:
        PUSH    HL              ;SAVE BC,HL
        PUSH    BC

        ;WAIT FOR CHARACTER FROM CONSOLE
RDWAIT:
        LD      C,DIRIO         ;DIRECT CONSOLE I/O
        LD      E,0FFH          ;INDICATE INPUT
        CALL    BDOS            ;READ CHARACTER FROM CONSOLE
        OR      A               ;LOOP IF NO CHARACTER (A = 0)
        JR      Z,RDWAIT

        POP     DE              ;RESTORE BC,HL
        POP     HL
        RET                     ;RETURN WITH CHARACTER IN REGISTER A

        ;****************************************
        ;ROUTINE: WRCHAR
        ;PURPOSE: WRITE CHARACTER TO CONSOLE
        ;ENTRY: REGISTER A = CHARACTER
        ;EXIT:  NONE
        ;REGISTERS USED: ALL EXCEPT BC, HL
        ;****************************************
WRCHAR:
        PUSH    HL              ;SAVE BC, HL
        PUSH    BC

        ;WRITE A CHARACTER
        LD      C,DIRIO         ;DIRECT CONSOLE I/O
        LD      E,A             ;INDICATE OUTPUT - CHARACTER IN E
        CALL    BDOS            ;WRITE CHARACTER ON CONSOLE

        POP     BC              ;RESTORE BC,HL
        POP     HL
        RET

        ;****************************************
        ;ROUTINE: WRNEWL
        ;PURPOSE: ISSUE NEW LINE SEQUENCE TO CONSOLE
        ;          NORMALLY, THIS IS A CARRIAGE RETURN AND
        ;          LINE FEED, BUT SOME COMPUTERS REQUIRE ONLY
        ;          A CARRIAGE RETURN.
        ;ENTRY: NONE
        ;EXIT:  NONE
        ;REGISTERS USED: ALL EXCEPT BC,HL
        ;*****************************************
WRNEWL:
        PUSH    HL              ;SAVE BC,HL
        PUSH    BC

        ;SEND NEW LINE STRING TO CONSOLE
        LD      DE,NLSTRG       ;POINT TO NEW LINE STRING
        CALL    WRSTRG          ;SEND STRING TO CONSOLE

        POP     BC              ;RESTORE BC, HL
        POP     HL
        RET

NLSTRG: DB      CR,LF,STERM     ;NEW LINE STRING
                                ; NOTE: STERM ($) IS CP/M TERMINATOR

        ;*****************************************
        ;ROUTINE: BACKSP
        ;PURPOSE: PERFORM A DESTRUCTIVE BACKSPACE
        ;ENTRY: B = NUMBER OF CHARACTERS IN BUFFER
        ;       HL = NEXT AVAILABLE BUFFER ADDRESS
        ;EXIT:  IF NO CHARACTERS IN BUFFER
        ;         Z = 1
        ;       ELSE
        ;         Z = 0
        ;         CHARACTER REMOVED FROM BUFFER
        ;REGISTERS USED: ALL EXCEPT C, HL
        ;*****************************************
BACKSP:
        ;CHECK FOR EMPTY BUFFER
        LD      A,B             ;TEST NUMBER OF CHARACTERS
        OR      A
        RET     Z               ;EXIT IF BUFFER EMPTY

        ;OUTPUT BACKSPACE STRING
        ; TO REMOVE CHARACTER FROM DISPLAY
        DEC     HL              ;DECREMENT BUFFER POINTER
        PUSH    HL              ;SAVE BC, HL
        PUSH    BC
        LD      A,(HL)          ;GET CHARACTER
        CP      20H             ;IS IT A CONTROL?
        JR      NC,BS1          ; NO, BRANCH, DELETE ONLY ONE CHARACTER
        LD      DE,BSSTRG       ; YES, DELETE 2 CHARACTERS
                                ;   (UP ARROW AND PRINTABLE EQUIVALENT)
        CALL    WRSTRG          ;WRITE BACKSPACE STRING
BS1:    LD      DE,BSSTRG
        CALL    WRSTRG          ;WRITE BACKSPACE STRING
        POP     BC              ;RESTORE BC, HL
        POP     HL
        ;DECREMENT CHARACTER COUNT BY 1
        DEC     B                   ;ONE LESS CHARACTER IN BUFFER
        RET

        ;DESTRUCTIVE BACKSPACE STRING FOR CONSOLE
        ;MOVES CURSOR LEFT, PRINTS SPACE OVER CHARACTER, MOVES
        ; CURSOR LEFT
        ;NOTE: STERM ($) IS CP/M STRING TERMINATOR
BSSTRG: DB      CSRLFT,SPACE,CSRLFT,STERM

        ;*******************************
        ;ROUTINE: WRSTRG
        ;PURPOSE: OUTPUT STRING TO CONSOLE
        ;ENTRY: HL = BASE ADDRESS OF STRING
        ;EXIT:  NONE
        ;REGISTERS USED: ALL EXCEPT BC
        ;*******************************
WRSTRG:
        PUSH    BC                  ;SAVE BC
        LD      C,PSTRG             ;FUNCTION IS PRINT STRING
        CALL    BDOS                ;OUTPUT STRING TERMINATED WITH $
        POP     BC                  ;RESTORE BC
        RET


;                                                                       ;
;                                                                       ;
;       SAMPLE EXECUTION:                                               ;
;                                                                       ;
;                                                                       ;

        ;EQUATES
PROMPT  EQU     '?'                 ;OPERATOR PROMPT = QUESTION MARK

SC10A:
        ;READ LINE FROM CONSOLE
        LD      A,PROMPT            ;OUTPUT PROMPT (?)
        CALL    WRCHAR
        LD      HL,INBUFF           ;HL = INPUT BUFFER ADDRESS
        LD      A,LINBUF            ;A = BUFFER LENGTH
        CALL    RDLINE              ;READ A LINE
        OR      A                   ;TEST LINE LENGTH
        JR      Z,SC10A             ;NEXT LINE IF LENGTH IS 0

        ;ECHO LINE TO CONSOLE
        LD      B,A                 ;SAVE NUMBER OF CHARACTERS IN BUFFER
        LD      HL,INBUFF           ;POINT TO START OF BUFFER
TLOOP:
        LD      A,(HL)              ;OUTPUT NEXT CHARACTER
        CALL    WRCHAR
        INC     HL                  ;INCREMENT BUFFER POINTER
        DJNZ    TLOOP               ;DECREMENT CHARACTER COUNT
                                    ; CONTINUE UNTIL ALL CHARACTERS SENT
        CALL    WRNEWL              ;THEN END WITH CR,LF

        JR      SC10A
        ;DATA SECTION
LINBUF  EQU     16              ;LENGTH OF INPUT BUFFER
INBUFF: DS      LINBUF          ;INPUT BUFFER

        END
```

Write a Line to an Output Device (WRLINE) 10B

Writes characters until it empties a buffer with given length and base address. Assumes the system-dependent subroutine WRCHAR, which sends the character in the accumulator to the output device.

WRLINE is an example of an output driver. The actual I/O subroutines will, of course, be computer-dependent. A specific example in the listing is for a CP/M-based computer with a standard Basic Disk Operating System (BDOS) accessed by calling address 0005_{16}.

Procedure: The program exits immediately if the buffer is empty. Otherwise, it sends characters to the output device one at a time until it empties the buffer. The program saves all temporary data in memory rather than in registers to avoid dependence on WRCHAR.

Registers Used: AF, BC, DE, HL

Execution Time: 18 cycles overhead plus 43 cycles per byte besides the execution time of subroutine WRCHAR

Program Size: 22 bytes

Data Memory Required: None

Special Case: An empty buffer causes an immediate exit with nothing sent to the output device.

Entry Conditions

Base address of buffer in HL

Number of characters in the buffer in A

Exit Conditions

None

Example

1.	Data:	Number of characters = 5
		Buffer contains 'ENTER'
	Result:	'ENTER' sent to the output device

```
;                                                               ;
;                                                               ;
;                                                               ;
;                                                               ;
;       Title           Write line                              ;
;       Name:           WRLINE                                  ;
;                                                               ;
;                                                               ;
;       Purpose:        Write characters to CP/M BDOS CON: device       ;
;                                                                       ;
;       Entry:          Register pair HL = Base address of buffer       ;
;                       Register A = Number of characters in buffer     ;
;                                                                       ;
;       Exit:           None                                            ;
;                                                                       ;
;       Registers used: All                                             ;
;                                                                       ;
;       Time:           Not applicable                                  ;
;                                                                       ;
;       Size:           Program 22 bytes                                ;
;                                                                       ;
;                                                                       ;

        ;EQUATES
BDOS    EQU     0005H           ;BDOS ENTRY POINT
DIRIO   EQU     6               ;BDOS DIRECT I/O FUNCTION

WRLINE:
        ;EXIT IMMEDIATELY IF BUFFER IS EMPTY
        OR      A               ;TEST NUMBER OF CHARACTERS
        RET     Z               ; RETURN IF BUFFER EMPTY
        LD      B,A             ;B = COUNTER
                                ;HL = BASE ADDRESS OF BUFFER

        ;LOOP SENDING CHARACTERS TO OUTPUT DEVICE
WRLLP:
        LD      A,(HL)          ;GET NEXT CHARACTER
        CALL    WRCHAR          ;SEND CHARACTER TO OUTPUT DEVICE
        INC     HL              ;INCREMENT BUFFER POINTER
        DJNZ    WRLLP           ;DECREMENT COUNTER
                                ; CONTINUE UNTIL ALL CHARACTERS SENT
        RET


        ;****************************************
        ;ROUTINE: WRCHAR
        ;PURPOSE: WRITE CHARACTER TO OUTPUT DEVICE
        ;ENTRY: REGISTER A = CHARACTER
        ;EXIT:  NONE
        ;REGISTERS USED: AF,DE
        ;****************************************

WRCHAR:
        PUSH    HL              ;SAVE BC, HL
        PUSH    BC

        LD      C,DIRIO         ;DIRECT I/O FUNCTION

        LD      E,A             ;CHARACTER IN REGISTER E
        CALL    BDOS            ;OUTPUT CHARACTER

        POP     BC              ;RESTORE BC, HL
        POP     HL
        RET
```

```
;                                                                        ;
;                                                                        ;
;       SAMPLE EXECUTION:                                                ;
;                                                                        ;
;                                                                        ;

RCBUF   EQU     10                  ;BDOS READ CONSOLE BUFFER FUNCTION

        ;BDOS READ CONSOLE BUFFER FUNCTION USES
        ; THE FOLLOWING BUFFER FORMAT:
        ;    BYTE 0   : BUFFER LENGTH (MAXIMUM NUMBER OF CHARACTERS)
        ;    BYTE 1   : NUMBER OF CHARACTERS READ (LINE LENGTH)
        ;    BYTE 2 ON: ACTUAL CHARACTERS

        ;CHARACTER EQUATES
CR      EQU     0DH                 ;CARRIAGE RETURN FOR CONSOLE
LF      EQU     0AH                 ;LINE FEED FOR CONSOLE
PROMPT  EQU     '?'                 ;OPERATOR PROMPT = QUESTION MARK

SC10B:
        ;READ LINE FROM CONSOLE
        LD      A,PROMPT            ;OUTPUT PROMPT (?)
        CALL    WRCHAR
        LD      DE,INBUFF           ;POINT TO INPUT BUFFER
        LD      C,RCBUF             ;BDOS READ LINE FUNCTION
        CALL    BDOS                ;READ LINE FROM CONSOLE
        LD      A,LF                ;OUTPUT LINE FEED
        CALL    WRCHAR

        ;WRITE LINE TO CONSOLE
        LD      HL,INBUFF+1         ;POINT AT LENGTH BYTE RETURNED BY CP/M
        LD      A,M                 ;GET LENGTH OF LINE
        INC     HL                  ;POINT TO FIRST DATA BYTE OF INBUFF
        CALL    WRLINE              ;WRITE LINE
        LD      HL,CRLF             ;OUTPUT CARRIAGE RETURN, LINE FEED
        LD      A,2                 ;LENGTH OF CRLF STRING
        CALL    WRLINE              ;WRITE CRLF STRING

        JR      SC10B               ;CONTINUE

        ;DATA SECTION
CRLF:   DB      CR,LF               ;CARRIAGE RETURN, LINE FEED
LINBUF  EQU     10H                 ;LENGTH OF INPUT BUFFER
INBUFF: DB      LINBUF              ;LENGTH OF INPUT BUFFER
        DS      LINBUF              ;DATA BUFFER

        END
```

CRC-16 Checking and Generation (ICRC16,CRC16,GCRC16) 10C

Generates a 16-bit cyclic redundancy check (CRC) based on the IBM Binary Synchronous Communications protocol (BSC or Bisync). Uses the polynomial $X^{16} + X^{15} + X^2 + 1$. Entry point ICRC16 initializes the CRC to 0 and the polynomial to the appropriate bit pattern. Entry point CRC16 combines the previous CRC with the CRC generated from the current data byte. Entry point GCRC16 returns the CRC.

Procedure: Subroutine ICRC16 initializes the CRC to 0 and the polynomial to a 1 in each bit position corresponding to a power of X present in the formula. Subroutine CRC16 updates the CRC for a data byte. It shifts both the data and the CRC left eight times; after each shift, it EXCLUSIVE-ORs the CRC with the polynomial if the EXCLUSIVE-OR of the data bit and the CRC's most significant bit is 1. Subroutine CRC16 leaves the CRC in memory locations CRC (less significant byte) and CRC+1 (more significant byte). Subroutine GCRC16 loads the CRC into HL.

Registers Used:
1. ICRC16: HL
2. CRC16: None
3. GCRC16: HL

Execution Time:
1. ICRC16: 62 cycles
2. CRC16: 148 cycles overhead plus an average of 584 cycles per data byte, assuming that the previous CRC and the polynomial must be EXCLUSIVE-ORed in half of the iterations
3. GCRC16: 26 cycles

Program Size:
1. ICRC16: 13 bytes
2. CRC16: 39 bytes
3. GCRC16: 4 bytes

Data Memory Required: 4 bytes anywhere in RAM for the CRC (2 bytes starting at address CRC) and the polynomial (2 bytes starting at address PLY)

Entry Conditions

1. ICRC16: none

2. CRC16: data byte in A, previous CRC in memory locations CRC (less significant byte) and CRC+1 (more significant byte), CRC polynomial in memory locations PLY (less significant byte) and PLY+1 (more significant byte)

3. GCRC16: CRC in memory locations CRC (less significant byte) and CRC+1 (more significant byte)

Exit Conditions

1. ICRC16: 0 (initial CRC value) in memory locations CRC (less significant byte) and CRC+1 (more significant byte), CRC polynomial in memory locations PLY (less significant byte) and PLY+1 (more significant byte)

2. CRC16: CRC with current data byte included in memory locations CRC (less significant byte) and CRC+1 (more significant byte)

3. GCRC16: CRC in HL

Examples

1. Generating a CRC
   a. Call ICRC16 for initialization and to start the CRC at 0.
   b. Call CRC16 repeatedly to update the CRC for each data byte.
   c. Call GCRC16 to obtain the final CRC.
2. Checking a CRC
   a. Call ICRC16 for initialization and to start the CRC at 0.
   b. Call CRC16 repeatedly to update the CRC for each data byte (including the stored CRC) for checking.
   c. Call GCRC16 to obtain the final CRC; it will be 0 if there were no errors.

Note that only ICRC16 depends on the particular CRC polynomial used. To change the polynomial, simply change the data ICRC16 loads into memory locations PLY (less significant byte) and PLY+1 (more significant byte).

REFERENCE

J.E.McNamara. *Technical Aspects of Data Communications*, 2nd ed. Billerica, Mass.: Digital Press, 1982. This book contains explanations of CRC and communications protocols.

```
;                                                                  ;
;                                                                  ;
;                                                                  ;
;                                                                  ;
;       Title           Generate CRC-16                            ;
;       Name:           CRC16                                      ;
;                                                                  ;
;                                                                  ;

;                                                                  ;
;       Purpose:        Generate a 16-bit CRC based on IBM's Binary ;
;                       Synchronous Communications protocol. The CRC is ;
;                       based on the polynomial:                   ;
;                         (^ indicates "to the power")             ;
;                              X^16 + X^15 + X^2 + 1               ;
;                                                                  ;
;                       To generate a CRC:                         ;
;                           1)  Call ICRC16 to initialize the CRC  ;
;                               polynomial and clear the CRC.      ;
;                           2)  Call CRC16 for each data byte.     ;
;                           3)  Call GCRC16 to obtain the CRC.     ;
;                               It should then be appended to the data, ;
;                               high byte first.                   ;
;                                                                       ;
;                       To check a CRC:                                 ;
;                           1)  Call ICRC16 to initialize the CRC.     ;
;                           2)  Call CRC16 for each data byte and      ;
;                               the 2 bytes of CRC previously generated.;
;                           3)  Call GCRC16 to obtain the CRC. It will ;
;                               be zero if no errors occurred.          ;
;                                                                       ;
;       Entry:          ICRC16 - None                                   ;
;                       CRC16  - Register A = Data byte                 ;
;                       GCRC16 - None                                   ;
;                                                                       ;
;                                                                       ;
;       Exit:           ICRC16 - CRC, PLY initialized                   ;
;                       CRC16  - CRC updated                            ;
;                       GCRC16 - HL = CRC                               ;
;                                                                       ;
;       Registers used: None                                            ;
;                                                                       ;
;       Time:           148 cycles overhead plus an average of 584      ;
;                       cycles per byte, assuming that half the         ;
;                       iterations require EXCLUSIVE-ORing the CRC      ;
;                       and the polynomial.                             ;
;                                                                       ;
;       Size:           Program 56 bytes                                ;
;                       Data     4 bytes                                ;
;                                                                       ;
;                                                                       ;

CRC16:
        ;SAVE REGISTERS
        PUSH    AF
        PUSH    BC
        PUSH    DE
        PUSH    HL

        ;LOOP THROUGH EACH BIT GENERATING THE CRC
        LD      B,8             ;8 BITS PER BYTE
        LD      DE,(PLY)        ;GET POLYNOMIAL
        LD      HL,(CRC)        ;GET CURRENT CRC VALUE
CRCLP:
        LD      C,A             ;SAVE DATA C
        AND     10000000B       ;GET BIT 7 OF DATA
        XOR     H               ;EXCLUSIVE-OR BIT 7 WITH BIT 15 OF CRC
        LD      H,A
        ADD     HL,HL           ;SHIFT CRC LEFT
        JR      NC,CRCLP1       ;JUMP IF BIT 7 OF EXCLUSIVE-OR WAS 0

        ;BIT 7 WAS 1, SO EXCLUSIVE-OR CRC WITH POLYNOMIAL
        LD      A,E             ;GET LOW BYTE OF POLYNOMIAL
        XOR     L               ;EXCLUSIVE-OR WITH LOW BYTE OF CRC
        LD      L,A
        LD      A,D             ;GET HIGH BYTE OF POLYNOMIAL
        XOR     H               ;EXCLUSIVE-OR WITH HIGH BYTE OF CRC
        LD      H,A
```

```
CRCLP1:
        LD      A,C             ;RESTORE DATA
        RLA                     ;SHIFT NEXT DATA BIT TO BIT 7
        DJNZ    CRCLP           ;DECREMENT BIT COUNTER
                                ; JUMP IF NOT THROUGH 8 BITS
        LD      (CRC),HL        ;SAVE UPDATED CRC

        ;RESTORE REGISTERS AND EXIT
        POP     HL
        POP     DE
        POP     BC
        POP     AF
        RET

        ;****************************************
        ;ROUTINE: ICRC16
        ;PURPOSE: INITIALIZE CRC AND PLY
        ;ENTRY: NONE
        ;EXIT:  CRC AND POLYNOMIAL INITIALIZED
        ;REGISTERS USED: HL
        ;****************************************
ICRC16:
        LD      HL,0            ;CRC = 0
        LD      (CRC),HL
        LD      HL,08005H       ;PLY = 8005H
        LD      (PLY),HL        ;8005H IS FOR X^16+X^15+X^2+1
                                ;A 1 IS IN EACH BIT POSITION
                                ; FOR WHICH A POWER APPEARS IN
                                ; THE FORMULA (BITS 0, 2, AND 15)
        RET

        ;****************************************
        ;ROUTINE: GCRC16
        ;PURPOSE: GET CRC VALUE
        ;ENTRY: NONE
        ;EXIT:  REGISTER PAIR HL = CRC VALUE
        ;REGISTERS USED: HL
        ;****************************************
GCRC16:
        LD      HL,(CRC)        ;HL = CRC
        RET

        ;DATA
CRC:    DS      2               ;CRC VALUE
PLY:    DS      2               ;POLYNOMIAL VALUE


;                                                                       ;
;                                                                       ;
;       SAMPLE EXECUTION:                                               ;
;                                                                       ;
;                                                                       ;

        ;GENERATE A CRC FOR THE NUMBER 1 AND CHECK IT
SC10C:
        CALL    ICRC16          ;INITIALIZE CRC, POLYNOMIAL
        LD      A,1             ;GENERATE CRC FOR 1
        CALL    CRC16
        CALL    GCRC16
        EX      DE,HL           ;SAVE CRC IN DE
        CALL    ICRC16          ;INITIALIZE AGAIN
        LD      A,1
        CALL    CRC16           ;CHECK CRC BY GENERATING IT FOR DATA
        LD      A,D
        CALL    CRC16           ; AND STORED CRC ALSO
        LD      A,E
        CALL    CRC16
        CALL    GCRC16          ;CRC SHOULD BE ZERO IN HL

        ;GENERATE CRC FOR THE SEQUENCE 0,1,2,...,255 AND CHECK IT
        CALL    ICRC16          ;INITIALIZE CRC, POLYNOMIAL
        LD      B,0             ;START DATA BYTES AT 0
GENLP:
        LD      A,B             ;GET DATA BYTE
        CALL    CRC16           ;UPDATE CRC
        INC     B               ;ADD 1 TO PRODUCE NEXT DATA BYTE
        JR      NZ,GENLP        ;BRANCH IF NOT DONE

        CALL    GCRC16          ;GET RESULTING CRC
        EX      DE,HL           ;AND SAVE IT IN DE

        ;CHECK CRC BY GENERATING IT AGAIN
        CALL    ICRC16          ;INITIALIZE CRC, POLYNOMIAL
        LD      B,0             ;START DATA BYTES AT 0
CHKLP:
        LD      A,B             ;GET DATA BYTE
        CALL    CRC16           ;UPDATE CRC
        INC     B               ;ADD 1 TO PRODUCE NEXT DATA BYTE
        JR      NZ,CHKLP

        ;ALSO INCLUDE STORED CRC IN CHECK
        LD      A,D             ;UPDATE FOR HIGH BYTE OF STORED CRC
        CALL    CRC16
        LD      A,E             ;UPDATE FOR LOW BYTE OF STORED CRC
        CALL    CRC16

        CALL    GCRC16          ;GET RESULTING CRC
                                ;IT SHOULD BE 0

        JR      SC10C

        END
```

I/O Device Table Handler (IOHDLR) 10D

Performs input and output in a device-independent manner using I/O control blocks and an I/O device table. The I/O device table is a linked list; each entry contains a link to the next entry, the device number, and starting addresses for routines that initialize the device, determine its input status, read data from it, determine its output status, and write data to it. An I/O control block is an array containing the device number, operation number, device status, and the base address and length of the device's buffer. The user must provide IOHDLR with the base address of an I/O control block and the data if only one byte is to be written. IOHDLR returns the status byte and the data (if only one byte is read).

This subroutine is an example of handling input and output in a device-independent manner. The I/O device table must be constructed using subroutines INITDL, which initializes the device list to empty, and ADDDL, which adds a device to the list.

An applications program will perform input or output by obtaining or constructing an I/O control block and then calling IOHDLR. IOHDLR uses the I/O device table to determine how to transfer control to the I/O driver.

Procedure: The program first initializes the status byte to 0, indicating no errors. It then searches the device table, trying to match the device number in the I/O control block. If it does not find a match, it exits with an error number in the status byte. If it finds a match, it checks for a valid operation and transfers control to the appropriate routine from the device table entry. That routine must end by transferring control back to the original caller. If the operation is invalid (the operation number is too large or the starting address for the routine is 0), the program returns with an error number in the status byte.

Subroutine INITDL initializes the device list, setting the initial link to 0.

Subroutine ADDDL adds an entry to the device list, making its base address the head of the list and setting its link field to the old head of the list.

Registers Used:

1. IOHDLR: AF,BC,DE,HL,IX
2. INITDL: HL
3. ADDDL: DE

Execution Time:

1. IOHDLR: 270 cycles overhead plus 90 cycles for each unsuccessful match of a device number
2. INITDL: 36 cycles
3. ADDDL: 72 cycles

Program Size:

1. IOHDLR: 70 bytes
2. INITDL: 7 bytes
3. ADDDL: 12 bytes

Data Memory Required: 3 bytes anywhere in RAM for the device list header (2 bytes starting at address DVLST) and temporary storage for data to be written without a buffer (1 byte at address BDATA)

Entry Conditions

1. IOHDLR: Base address of input/output control block in IX

Exit Conditions

1. IOHDLR: I/O control block status byte in A if an error is found;

Data byte (if the operation is to write one byte) in A

2. INITDL: None
3. ADDDL: Base address of a device table entry in HL

otherwise, the routine exits to the appropriate I/O driver. Data byte in A if the operation is to read one byte

2. INITDL: Device list header (addresses DVLST and DVLST+1) cleared to indicate empty list
3. ADDDL: Device table entry added to list

Example

1. The example in the listing uses the following structure:

Input/Output Operations

Operation Number	**Operation**
0	Initialize device
1	Determine input status
2	Read 1 byte from input device
3	Read N bytes (normally 1 line) from input device
4	Determine output status
5	Write 1 byte to output device
6	Write N bytes (normally 1 line) to output device

Input/Output Control Block

Index	**Contents**
0	Device number
1	Operation number
2	Status
3	Less significant byte of base address of buffer
4	More significant byte of base address of buffer
5	Less significant byte of buffer length
6	More significant byte of buffer length

Device Table Entry

Index	**Contents**
0	Less significant byte of link field (base address of next entry)
1	More significant byte of link field (base address of next entry)
2	Device number
3	Less significant byte of starting address of device initialization routine
4	More significant byte of starting address of device initialization routine
5	Less significant byte of starting address of input status determination routine
6	More significant byte of starting address of input status determination routine
7	Less significant byte of starting address of input driver (read 1 byte only)
8	More significant byte of starting address of input driver (read 1 byte only)
9	Less significant byte of starting address of input driver (N bytes or 1 line)
10	More significant byte of starting address of input driver (N bytes or 1 line)
11	Less significant byte of starting address of output status determination routine
12	More significant byte of starting address of output status determination routine
13	Less significant byte of starting address of output driver (write 1 byte only)
14	More significant byte of starting address of output driver (write 1 byte only)
15	Less significant byte of starting address of output driver (N bytes or 1 line)
16	More significant byte of starting address of output driver (N bytes or 1 line)

If an operation is irrelevant or undefined (such as output status determination for a keyboard or input driver for a printer), the corresponding starting address in the device table is 0.

Status Values

Value	Description
0	No errors
1	Bad device number (no such device)
2	Bad operation number (no such operation or invalid operation)
3	Input data available or output device ready
254	Buffer too small for use by CP/M BDOS function 10 (Read Console Buffer). This function requires 2 bytes for the buffer length and character count.

```
;                                                                     ;
;                                                                     ;
;                                                                     ;
;                                                                     ;
;       Title           I/O Device Table Handler                      ;
;       Name:           IOHDLR                                        ;
;                                                                     ;
;                                                                     ;

;                                                                     ;
;       Purpose:        Perform I/O in a device-independent manner.   ;
;                       This can be done only by accessing all        ;
;                       devices in the same way using an I/O Control  ;
;                       Block (IOCB) and a device table. The routines ;
;                       here allow the following operations:          ;
;                                                                     ;
;                       Operation number  Description                 ;
;                           0            Initialize device            ;
;                           1            Determine input status       ;
;                           2            Read 1 byte                  ;
;                           3            Read N bytes                 ;
;                           4            Determine output status      ;
;                           5            Write 1 byte                 ;
;                           6            Write N bytes                ;
;                                                                     ;
;                       Other operations that could be included are   ;
;                       Open, Close, Delete, Rename, and Append, which ;
;                       would support devices such as floppy disks.   ;
;                                                                     ;
;                       A IOCB is an array of the following form:     ;
;                                                                     ;
;                       IOCB + 0 = Device number                      ;
;                       IOCB + 1 = Operation number                   ;
;                       IOCB + 2 = Status                             ;
;                       IOCB + 3 = Low byte of buffer address         ;
;                       IOCB + 4 = High byte of buffer address        ;
;                       IOCB + 5 = Low byte of buffer length          ;
;                       IOCB + 6 = High byte of buffer length         ;
;                                                                        ;
;                       The device table is implemented as a linked      ;
;                       list. Two routines maintain the list: INITDL,    ;
;                       which initializes the device list to empty, and  ;
;                       ADDDL, which adds a device to the list.          ;
;                       A device table entry has the following form:     ;
;                                                                        ;
;                       DVTBL + 0 = Low byte of link field               ;
;                       DVTBL + 1 = High byte of link field              ;
;                       DVTBL + 2 = Device number                        ;
;                       DVTBL + 3 = Low byte of device initialization    ;
;                       DVTBL + 4 = High byte of device initialization   ;
;                       DVTBL + 5 = Low byte of input status routine     ;
;                       DVTBL + 6 = High byte of input status routine    ;
;                       DVTBL + 7 = Low byte of input 1 byte routine     ;
;                       DVTBL + 8 = High byte of input 1 byte routine    ;
;                       DVTBL + 9 = Low byte of input N bytes routine    ;
;                       DVTBL + 10= High byte of input N bytes routine   ;
;                       DVTBL + 11= Low byte of output status routine    ;
;                       DVTBL + 12= High byte of output status routine   ;
;                       DVTBL + 13= Low byte of output 1 byte routine    ;
;                       DVTBL + 14= High byte of output 1 byte routine   ;
;                       DVTBL + 15= Low byte of output N bytes routine   ;
;                       DVTBL + 16= High byte of output N bytes routine  ;
;                                                                        ;
;       Entry:          Register IX = Base address of IOCB               ;
;                       Register A = For write 1 byte, contains the      ;
;                                    data (no buffer is used)            ;
;                                                                        ;
;       Exit:           Register A = Copy of the IOCB status byte        ;
;                                    Except contains the data for        ;
;                                    read 1 byte (no buffer is used)     ;
;                       Status byte of IOCB is 0 if the operation was    ;
;                       completed successfully; otherwise, it contains   ;
;                       the error number.                                ;
;                                                                        ;
;                       Status value    Description                      ;
;                           0           No errors                        ;
;                           1           Bad device number                ;
;                           2           Bad operation number             ;
;                           3           Input data available or output   ;
;                                        device ready                    ;
;                           254         Buffer too small for CP/M BDOS   ;
;                                        function 10 (Read Console       ;
;                                        Buffer)                         ;
;                                                                        ;
;       Registers used: AF,BC,DE,HL,IX                                   ;
;                                                                        ;
;       Time:           270 cycles minimum plus 90 cycles for each       ;
;                       device in the list which is not the requested    ;
;                       device                                           ;
;                                                                        ;
;       Size:           Program 89 bytes                                 ;
;                       Data     3 bytes                                 ;
;                                                                        ;
;                                                                        ;
```

```
        ;IOCB AND DEVICE TABLE EQUATES
IOCBDN  EQU     0       ;IOCB DEVICE NUMBER
IOCBOP  EQU     1       ;IOCB OPERATION NUMBER
IOCBST  EQU     2       ;IOCB STATUS
IOCBBA  EQU     3       ;IOCB BUFFER ADDRESS
IOCBBL  EQU     5       ;IOCB BUFFER LENGTH
DTLNK   EQU     0       ;DEVICE TABLE LINK FIELD
DTDN    EQU     2       ;DEVICE TABLE DEVICE NUMBER
DTSR    EQU     3       ;BEGINNING OF DEVICE TABLE SUBROUTINES

        ;OPERATION NUMBERS
NUMOP   EQU     7       ;NUMBER OF OPERATIONS
INIT    EQU     0       ;INITIALIZATION
ISTAT   EQU     1       ;INPUT STATUS
R1BYTE  EQU     2       ;READ 1 BYTE
RNBYTE  EQU     3       ;READ N BYTES
OSTAT   EQU     4       ;OUTPUT STATUS
W1BYTE  EQU     5       ;WRITE 1 BYTE
WNBYTE  EQU     6       ;WRITE N BYTES

        ;STATUS VALUES
NOERR   EQU     0       ;NO ERRORS
DEVERR  EQU     1       ;BAD DEVICE NUMBER
OPERR   EQU     2       ;BAD OPERATION NUMBER
DEVRDY  EQU     3       ;INPUT DATA AVAILABLE OR OUTPUT DEVICE READY
BUFERR  EQU     254     ;BUFFER TOO SMALL FOR BDOS READ CONSOLE BUFFER


IOHDLR:
        LD      (BDATA),A       ;SAVE DATA BYTE FOR WRITE 1 BYTE

        ;INITIALIZE STATUS BYTE TO ZERO (NO ERRORS)
        LD      (IX+IOCBST),NOERR       ;STATUS = NO ERRORS

        ;CHECK THAT OPERATION IS VALID
        LD      A,(IX+IOCBOP)   ;GET OPERATION NUMBER FROM IOCB
        LD      B,A             ;SAVE OPERATION NUMBER
        CP      NUMOP           ;IS OPERATION NUMBER WITHIN LIMIT?
        JR      NC,BADOP        ;JUMP IF OPERATION NUMBER TOO LARGE

        ;SEARCH DEVICE LIST FOR THIS DEVICE
        ;C = IOCB DEVICE NUMBER
        ;DE = POINTER TO DEVICE LIST
        LD      C,(IX+IOCBDN)   ;C = IOCB DEVICE NUMBER
        LD      DE,(DVLST)      ;DE = FIRST ENTRY IN DEVICE LIST

        ;DE = POINTER TO DEVICE LIST
        ;B = OPERATION NUMBER
        ;C = REQUESTED DEVICE NUMBER

SRCHLP:
        ;CHECK IF AT END OF DEVICE LIST (LINK FIELD = 0000)
        LD      A,D             ;TEST LINK FIELD
        OR      E
        JR      Z,BADDN         ;BRANCH IF NO MORE DEVICE ENTRIES
        ;CHECK IF CURRENT ENTRY IS DEVICE IN IOCB
        LD      HL,DTDN         ;POINT TO DEVICE NUMBER IN ENTRY
        ADD     HL,DE
        LD      A,(HL)
        CP      C               ;COMPARE TO REQUESTED DEVICE
        JR      Z,FOUND         ;BRANCH IF DEVICE FOUND

        ;DEVICE NOT FOUND, SO ADVANCE TO NEXT DEVICE
        ; TABLE ENTRY THROUGH LINK FIELD
        ; MAKE CURRENT DEVICE = LINK
        EX      DE,HL           ;POINT TO LINK FIELD (FIRST WORD)
        LD      E,(HL)          ;GET LOW BYTE OF LINK
        INC     HL
        LD      D,(HL)          ;GET HIGH BYTE OF LINK
        JR      SRCHLP          ;CHECK NEXT ENTRY IN DEVICE TABLE

        ;FOUND DEVICE, SO VECTOR TO APPROPRIATE ROUTINE IF ANY
        ;DE = ADDRESS OF DEVICE TABLE ENTRY
        ;B = OPERATION NUMBER
FOUND:
        ;GET ROUTINE ADDRESS (ZERO INDICATES INVALID OPERATION)
        LD      L,B             ;HL = 16-BIT OPERATION NUMBER
        LD      H,0
        ADD     HL,HL           ;MULTIPLY BY 2 FOR ADDRESS ENTRIES
        LD      BC,DTSR
        ADD     HL,BC           ;HL = OFFSET TO SUBROUTINE IN
                                ;      DEVICE TABLE ENTRY
        ADD     HL,DE           ;HL = ADDRESS OF SUBROUTINE
        LD      A,(HL)          ;GET SUBROUTINE'S STARTING ADDRESS
        INC     HL
        LD      H,(HL)
        LD      L,A             ;IS STARTING ADDRESS ZERO?
        OR      H
        JR      Z,BADOP         ;YES, JUMP (OPERATION INVALID)
        LD      A,(BDATA)       ;GET DATA BYTE FOR WRITE 1 BYTE
        JP      (HL)            ;GOTO SUBROUTINE

BADDN:
        LD      A,DEVERR        ;ERROR CODE -- NO SUCH DEVICE
        JR      EREXIT

BADOP:
        LD      A,OPERR         ;ERROR CODE -- NO SUCH OPERATION

EREXIT:
        LD      (IX+IOCBST),A   ;SET STATUS BYTE IN IOCB
        RET


        ;****************************************
        ;ROUTINE: INITDL
        ;PURPOSE: INITIALIZE DEVICE LIST TO EMPTY
        ;ENTRY: NONE
        ;EXIT:  DEVICE LIST SET TO NO ITEMS
        ;REGISTERS USED: HL
        ;****************************************
```

```
INITDL:
        ;INITIALIZE DEVICE LIST HEADER TO 0 TO INDICATE NO DEVICES
        LD      HL,0            ;HEADER = 0 (EMPTY LIST)
        LD      (DVLST),HL
        RET

        ;****************************************
        ;ROUTINE: ADDDL
        ;PURPOSE: ADD DEVICE TO DEVICE LIST
        ;ENTRY: REGISTER HL = ADDRESS OF DEVICE TABLE ENTRY
        ;EXIT:  DEVICE ADDED TO DEVICE LIST
        ;REGISTERS USED: DE
        ;****************************************
ADDDL:
        LD      DE,(DVLST)      ;GET CURRENT HEAD OF DEVICE LIST
        LD      (HL),E          ;STORE CURRENT HEAD OF DEVICE LIST
        INC     HL              ; INTO LINK FIELD OF NEW DEVICE
        LD      (HL),D
        DEC     HL
        LD      (DVLST),HL      ;MAKE DVLST POINT AT NEW DEVICE
        RET

        ;DATA SECTION
DVLST:  DS      2               ;DEVICE LIST HEADER
BDATA:  DS      1               ;DATA BYTE FOR WRITE 1 BYTE


;                                                                       ;
;                                                                       ;
;       SAMPLE EXECUTION:                                               ;
;                                                                       ;
;         This test routine sets up the CP/M console as                 ;
;         device 1 and the CP/M printer as device 2.                    ;
;         The routine then reads a line from the console and            ;
;         echoes it to the console and the printer.                     ;
;                                                                       ;
;                                                                       ;

        ;CHARACTER EQUATES
CR      EQU     0DH             ;CARRIAGE RETURN CHARACTER
LF      EQU     0AH             ;LINE FEED CHARACTER

        ;CP/M EQUATES
BDOS    EQU     0005H           ;ADDRESS OF CP/M BDOS ENTRY POINT
CINP    EQU     1               ;BDOS CONSOLE INPUT FUNCTION
COUTP   EQU     2               ;BDOS CONSOLE OUTPUT FUNCTION
LOUTP   EQU     5               ;BDOS LIST OUTPUT FUNCTION
RCBUF   EQU     10              ;BDOS READ CONSOLE BUFFER FUNCTION
CSTAT   EQU     11              ;BDOS CONSOLE STATUS FUNCTION

SC10D:
        ;INITIALIZE DEVICE LIST, POINT TO IOCB
        CALL    INITDL          ;INITIALIZE DEVICE LIST
        LD      IX,IOCB         ;POINT TO IOCB

        ;SET UP CONSOLE AS DEVICE 1 AND INITIALIZE IT
        LD      HL,CONDV          ;POINT TO CONSOLE DEVICE ENTRY
        CALL    ADDDL             ;ADD CONSOLE TO DEVICE LIST
        LD      (IX+IOCBOP),INIT ;INITIALIZE OPERATION
        LD      (IX+IOCBDN),1    ;DEVICE NUMBER = 1
        CALL    IOHDLR            ;INITIALIZE CONSOLE

        ;SET UP PRINTER AS DEVICE 2 AND INITIALIZE IT
        LD      HL,PRTDV          ;POINT TO PRINTER DEVICE ENTRY
        CALL    ADDDL             ;ADD PRINTER TO DEVICE LIST
        LD      (IX+IOCBOP),INIT ;INITIALIZE OPERATION
        LD      (IX+IOCBDN),2    ;DEVICE NUMBER = 2
        CALL    IOHDLR            ;INITIALIZE PRINTER

        ;LOOP READING LINES FROM CONSOLE, AND ECHOING THEM TO
        ; CONSOLE AND PRINTER UNTIL A BLANK LINE IS ENTERED
TSTLP:
        LD      (IX+IOCBDN),1    ;DEVICE NUMBER = 1 (CONSOLE)
        LD      (IX+IOCBOP),RNBYTE ;OPERATION IS READ N BYTES
        LD      HL,LENBUF
        LD      (IOCB+IOCBBL),HL ;SET BUFFER LENGTH TO LENBUF
        CALL    IOHDLR            ;READ A LINE

        ;OUTPUT LINE FEED TO CONSOLE
        LD      (IX+IOCBOP),W1BYTE ;OPERATION IS WRITE 1 BYTE
        LD      A,LF              ;CHARACTER IS LINE FEED
        CALL    IOHDLR            ;WRITE 1 BYTE (LINE FEED)

        ;ECHO LINE TO DEVICE 1 AND 2
        LD      A,1
        CALL    ECHO              ;ECHO LINE TO DEVICE 1
        LD      A,2
        CALL    ECHO              ;ECHO LINE TO DEVICE 2

        ;STOP IF LINE LENGTH IS 0
        LD      HL,(IOCB+IOCBBL) ;GET LINE LENGTH
        LD      A,H               ;TEST LINE LENGTH
        OR      L
        JR      NZ,TSTLP          ;CONTINUE IF LENGTH NOT ZERO

        JR      SC10D             ;AGAIN

ECHO:
        ;OUTPUT LINE
        LD      (IX+IOCBDN),A    ;SET DEVICE NUMBER IN IOCB
                                  ;NOTE THAT ECHO WILL SEND A LINE
                                  ; TO ANY DEVICE.  THE DEVICE NUMBER
                                  ; IS IN THE ACCUMULATOR
        LD      (IX+IOCBOP),WNBYTE ;SET OPERATION TO WRITE N BYTES
        CALL    IOHDLR            ;WRITE N BYTES

        ;OUTPUT CARRIAGE RETURN/LINE FEED
        LD      (IX+IOCBOP),W1BYTE ;SET OPERATION TO WRITE 1 BYTE
        LD      A,CR              ;CHARACTER IS CARRIAGE RETURN
        CALL    IOHDLR            ;WRITE 1 BYTE
        LD      A,LF              ;CHARACTER IS LINE FEED
        CALL    IOHDLR            ;WRITE 1 BYTE
        RET

        ;IOCB FOR PERFORMING I/O
IOCB:   DS      1                   ;DEVICE NUMBER
        DS      1                   ;OPERATION NUMBER
        DS      1                   ;STATUS
        DW      BUFFER              ;BUFFER ADDRESS
        DS      2                   ;BUFFER LENGTH

        ;BUFFER
LENBUF  EQU     127
BUFFER: DS      LENBUF

        ;DEVICE TABLE ENTRIES
CONDV:  DW      0                   ;LINK FIELD
        DB      1                   ;DEVICE 1
        DW      CINIT               ;CONSOLE INITIALIZE
        DW      CISTAT              ;CONSOLE INPUT STATUS
        DW      CIN                 ;CONSOLE INPUT 1 BYTE
        DW      CINN                ;CONSOLE INPUT N BYTES
        DW      COSTAT              ;CONSOLE OUTPUT STATUS
        DW      COUT                ;CONSOLE OUTPUT 1 BYTE
        DW      COUTN               ;CONSOLE OUTPUT N BYTES

PRTDV:  DW      0                   ;LINK FIELD
        DB      2                   ;DEVICE 2
        DW      PINIT               ;PRINTER INITIALIZE
        DW      0                   ;NO PRINTER INPUT STATUS
        DW      0                   ;NO PRINTER INPUT 1 BYTE
        DW      0                   ;NO PRINTER INPUT N BYTES
        DW      POSTAT              ;PRINTER OUTPUT STATUS
        DW      POUT                ;PRINTER OUTPUT 1 BYTE
        DW      POUTN               ;PRINTER OUTPUT N BYTES


        ;***********************************
        ;CONSOLE I/O ROUTINES
        ;***********************************


        ;CONSOLE INITIALIZE
CINIT:
        SUB     A                   ;STATUS = NO ERRORS
        RET                         ;NO INITIALIZATION NECESSARY

        ;CONSOLE INPUT STATUS
CISTAT:
        PUSH    IX                  ;SAVE IOCB ADDRESS
        LD      C,CSTAT             ;BDOS CONSOLE STATUS FUNCTION
        CALL    BDOS                ;GET CONSOLE STATUS
        POP     IX                  ;RESTORE IOCB ADDRESS
        OR      A
        JR      Z,CIS1              ;JUMP IF NOT READY
        LD      A,DEVRDY            ;INDICATE CHARACTER READY
CIS1:   LD      (IX+IOCBST),A       ;STORE STATUS AND LEAVE IT IN REGISTER A
        RET
        ;CONSOLE READ 1 BYTE
CIN:
        PUSH    IX                  ;SAVE IX
        LD      C,CINP              ;BDOS CONSOLE INPUT FUNCTION
        CALL    BDOS                ;READ 1 BYTE FROM CONSOLE
        POP     IX                  ;RESTORE IX
        RET

        ;CONSOLE READ N BYTES
CINN:
        ;READ LINE USING BDOS READ CONSOLE BUFFER FUNCTION
        ;BDOS READ CONSOLE BUFFER FUNCTION USES THE FOLLOWING BUFFER FORMAT:
        ;    BYTE 0: BUFFER LENGTH (MAXIMUM NUMBER OF CHARACTERS)
        ;    BYTE 1: NUMBER OF CHARACTERS READ (LINE LENGTH)
        ;    BYTES 2 ON: ACTUAL CHARACTERS
        PUSH    IX                  ;SAVE BASE ADDRESS OF IOCB
        LD      A,(IX+IOCBBL)       ;GET BUFFER LENGTH
        SUB     3                   ;BUFFER MUST BE AT LEAST 3 CHARACTERS
                                    ; TO ALLOW FOR MAXIMUM LENGTH AND COUNT
                                    ; USED BY BDOS READ CONSOLE BUFFER
        JR      NC,CINN1            ;JUMP IF BUFFER LONG ENOUGH
        LD      (IX+IOCBST),BUFERR ;SET ERROR STATUS - BUFFER TOO SMALL
        RET

CINN1:  INC     A                   ;ADD ONE BACK TO DETERMINE HOW MUCH
                                    ; SPACE IS AVAILABLE IN BUFFER FOR DATA
        LD      E,(IX+IOCBBA)       ;GET BUFFER ADDRESS FROM IOCB
        LD      D,(IX+IOCBBA+1)
        PUSH    DE                  ;SAVE BUFFER ADDRESS
        LD      (DE),A              ;SET MAXIMUM LENGTH IN BUFFER
        LD      C,RCBUF             ;BDOS READ CONSOLE BUFFER FUNCTION
        CALL    BDOS                ;READ BUFFER

        ;RETURN NUMBER OF CHARACTERS READ IN THE IOCB
        POP     HL                  ;RESTORE BUFFER ADDRESS
        POP     IX                  ;RESTORE BASE ADDRESS OF IOCB
        INC     HL                  ;POINT TO NUMBER OF CHARACTERS READ
        LD      A,(HL)              ;GET NUMBER OF CHARACTERS READ
        LD      (IX+IOCBBL),A       ;SET BUFFER LENGTH IN IOCB
        LD      (IX+IOCBBL+1),0     ; WITH UPPER BYTE = 0

        ;MOVE DATA TO FIRST BYTE OF BUFFER
        ;DROPPING OVERHEAD (BUFFER LENGTH, LINE LENGTH)
        ; RETURNED BY CP/M. LINE LENGTH IS NOW IN THE IOCB
        OR      A                   ;TEST LINE LENGTH
        RET     Z                   ;RETURN IF LENGTH WAS 0

        LD      C,A                 ;BC = NUMBER OF BYTES
        LD      B,0
        LD      D,H                 ;POINT TO START OF BUFFER + 1
        LD      E,L
        INC     HL                  ;HL = SOURCE = FIRST BYTE OF DATA
                                    ; 2 BYTES BEYOND START
        DEC     DE                  ;DE = DESTINATION (FIRST BYTE OF BUFFER)
        LDIR                        ;MOVE DATA DOWN 2 BYTES IN BUFFER
        SUB     A                   ;STATUS = NO ERRORS
        RET

        ;CONSOLE OUTPUT STATUS
COSTAT:
        LD      A,DEVRDY        ;STATUS = ALWAYS READY TO OUTPUT
        RET

        ;CONSOLE OUTPUT 1 BYTE
COUT:
        PUSH    IX              ;SAVE IX
        LD      C,COUTP         ;BDOS CONSOLE OUTPUT OPERATION
        LD      E,A             ;E = CHARACTER
        CALL    BDOS            ;OUTPUT 1 BYTE
        POP     IX              ;RESTORE IX
        SUB     A               ;RETURN, NO ERRORS
        RET

        ;CONSOLE OUTPUT N BYTES
COUTN:
        LD      HL,COUT         ;HL POINTS TO OUTPUT CHARACTER ROUTINE
        CALL    OUTN            ;CALL OUTPUT N CHARACTERS
        SUB     A               ;STATUS = NO ERRORS
        RET


        ;************************************
        ;PRINTER ROUTINES
        ;************************************


        ;PRINTER INITIALIZE
PINIT:
        SUB     A               ;NOTHING TO DO, RETURN NO ERRORS
        RET

        ;PRINTER OUTPUT STATUS
POSTAT:
        LD      A,DEVRDY        ;STATUS = ALWAYS READY TO OUTPUT
        RET

        ;PRINTER OUTPUT 1 BYTE
POUT:
        PUSH    IX              ;SAVE IX
        LD      C,LOUTP         ;BDOS LIST OUTPUT FUNCTION
        LD      E,A             ;E = CHARACTER
        CALL    BDOS            ;OUTPUT TO PRINTER
        POP     IX              ;RESTORE IX
        SUB     A               ;STATUS = NO ERRORS
        RET

        ;PRINTER OUTPUT N BYTES
POUTN:
        LD      HL,POUT         ;HL = ADDRESS OF OUTPUT ROUTINE
        CALL    OUTN            ;OUTPUT N CHARACTERS
        SUB     A               ;NO ERRORS
        RET
        ;****************************************
        ;ROUTINE: OUTN
        ;PURPOSE: OUTPUT N CHARACTERS
        ;ENTRY: REGISTER HL = CHARACTER OUTPUT SUBROUTINE ADDRESS
        ;       REGISTER IX = BASE ADDRESS OF AN IOCB
        ;EXIT:  DATA OUTPUT
        ;REGISTERS USED: AF,BC,HL
        ;****************************************
OUTN:
        ;STORE ADDRESS OF CHARACTER OUTPUT SUBROUTINE
        LD      (COSR),HL       ;SAVE ADDRESS

        ;GET NUMBER OF BYTES, EXIT IF LENGTH IS 0
        ; BC = NUMBER OF BYTES
        LD      C,(IX+IOCBBL)   ;BC = BUFFER LENGTH
        LD      B,(IX+IOCBBL+1)
        LD      A,B             ;TEST BUFFER LENGTH
        OR      C
        RET     Z               ;EXIT IF BUFFER EMPTY

        ;GET OUTPUT BUFFER ADDRESS FROM IOCB
        ; HL = BUFFER ADDRESS
        LD      L,(IX+IOCBBA)   ;HL = BUFFER ADDRESS
        LD      H,(IX+IOCBBA+1)

OUTLP:
        LD      A,(HL)
        PUSH    HL              ;SAVE BUFFER POINTER, COUNT
        PUSH    BC
        CALL    DOSUB           ;OUTPUT CHARACTER
        POP     BC              ;RESTORE COUNT, BUFFER POINTER
        POP     HL
        INC     HL              ;POINT TO NEXT CHARACTER
        DEC     BC              ;DECREMENT AND TEST COUNT
        LD      A,B
        OR      C
        JR      NZ,OUTLP        ;CONTINUE UNTIL COUNT = 0
        RET

DOSUB:  LD      HL,(COSR)
        JP      (HL)            ;GOTO ROUTINE

COSR:   DW      0               ;ADDRESS OF CHARACTER OUTPUT SUBROUTINE

        END
```

Initialize I/O Ports (IPORTS)

Registers Used: AF, BC, DE, HL

Execution Time: 22 cycles overhead plus 46 + 21 * N cycles for each port, where N is the number of bytes sent.

Program Size: 11 bytes plus the size of the table (at least 3 bytes per port plus 1 byte for a terminator)

Data Memory Required: None

Initializes a set of I/O ports from an array of port device addresses and data values. Examples are given of initializing the common Z80 programmable I/O devices: CTC, PIO, and SIO.

This subroutine is a generalized method for initializing I/O sections. The initialization may involve data ports, data direction registers that determine whether bits are inputs or outputs, control or command registers that determine the operating modes of programmable devices, counters (in timers), priority registers, and other external registers or storage locations.

Tasks the user may perform with this routine include:

1. Assigning bidirectional I/O lines as inputs or outputs
2. Initializing output ports
3. Enabling or disabling interrupts from peripheral chips
4. Determining operating modes, such as whether inputs are latched, whether strobes are produced, how priorities are assigned, whether timers operate continuously or only on demand, etc.
5. Loading starting values into timers and counters
6. Selecting bit rates for communications
7. Clearing or resetting devices that are not tied to the overall system reset line
8. Initializing priority registers or assigning initial priorities to interrupts or other operations
9. Initializing vectors used in servicing interrupts, DMA requests, and other inputs.

Procedure: For each port, the program obtains the number of bytes to be sent and the device address. It then sends the data values to the port using a repeated block output instruction. This approach does not depend on the number or type of devices in the I/O section. The user may add or delete devices or change the initialization by changing the array rather than the program. Each entry in the array consists of a series of byte-length elements in the following order:

1. Number of bytes to be sent to the port
2. 8-bit device address for the port
3. Data bytes in sequential order.

The array ends with a terminator that has 0 in its first byte.

Note that an entry may consist of an arbitrary number of bytes. The first element determines how many bytes are sent to the device address in the second element. The subsequent elements contain the data values. The terminator need consist only of a single 0 byte.

Entry Conditions

Base address of initialization array in HL

Exit Conditions

All data values sent to appropriate ports

Example

1. Data: Array elements are
   3 (number of bytes for port 1)
      Port 1 device address, first value, second value, third value
   2 (number of bytes for port 2)
      Port 2 device address, first value, second value
   4 (number of bytes for port 3)
      Port 3 device address, first value, second value, third value, fourth value
   0 (terminator)

   Result: Three values sent to port 1's device address
   Two values sent to port 2's device address
   Four values sent to port 3's device address

```
;                                                                        ;
;                                                                        ;
;                                                                        ;
;                                                                        ;
;       Title           Initialize I/O Ports                             ;
;       Name:           IPORTS                                           ;
;                                                                        ;
;                                                                        ;


;                                                                        ;
;       Purpose:        Initialize I/O ports from an array of port       ;
;                       addresses and values                             ;
;                                                                        ;
;       Entry:          Register pair HL = Base address of array         ;
;                                                                        ;
;                       The array consists of byte-length elements       ;
;                       in the following order: number of bytes to       ;
;                       be sent to the port, port device address, data   ;
;                       values for the port. This sequence is repeated   ;
;                       for any number of ports. The array is terminated;
;                       by an entry with 0 in the  number of bytes.      ;
;                          array+0 = Number of bytes for this port       ;
;                          array+1 = Port device address                 ;
;                          array+2 = First value for this port           ;
;                                   .                                    ;
;                                   .                                    ;
;                          array+2+(N-1) = Last value for this port      ;
;                                   .                                    ;
;                                   .                                    ;
;                                   .                                    ;
;                                                                        ;
;       Exit:           None                                             ;
;                                                                        ;
;       Registers used: AF,BC,DE,HL                                      ;
;                                                                        ;
;       Time:           22 cycles overhead plus 46 + (N * 21) cycles for;
;                       each port, where N is the number of bytes sent   ;
;                                                                    ;
;       Size:           Program 11 bytes                             ;
;                                                                    ;
;                                                                    ;

IPORTS:
        ;GET NUMBER OF DATA BYTES TO SEND TO CURRENT PORT
        ;EXIT IF NUMBER OF BYTES IS 0, INDICATING TERMINATOR
        LD      A,(HL)          ;GET NUMBER OF BYTES
        OR      A               ;TEST FOR ZERO (TERMINATOR)
        RET     Z               ;RETURN IF NUMBER OF BYTES = 0
        LD      B,A
        INC     HL              ;POINT TO PORT ADDRESS (NEXT BYTE)

        ;C = PORT ADDRESS
        ;HL = ADDRESS OF DATA TO OUTPUT
        LD      C,(HL)          ;GET PORT ADDRESS
        INC     HL              ;POINT TO FIRST DATA VALUE (NEXT BYTE)

        ;OUTPUT DATA AND CONTINUE TO NEXT PORT
        OTIR                    ;SEND DATA VALUES TO PORT
        JR      IPORTS          ;CONTINUE TO NEXT PORT ENTRY


;                                                                    ;
;                                                                    ;
;       SAMPLE EXECUTION:                                            ;
;                                                                    ;
;                                                                    ;


        ;INITIALIZE
        ;   Z80 CTC (PROGRAMMABLE TIMER/COUNTER)
        ;   Z80 SIO (PROGRAMMABLE SERIAL INTERFACE)
        ;   Z80 PIO (PROGRAMMABLE PARALLEL INTERFACE)

        ;ARBITRARY PORT ADDRESSES
        ;   CTC PORT ASSIGNMENTS
CTC0    EQU     70H             ;CTC CHANNEL 0
CTC1    EQU     71H             ;CTC CHANNEL 1
CTC2    EQU     72H             ;CTC CHANNEL 2
CTC3    EQU     73H             ;CTC CHANNEL 3

        ;   SIO PORT ASSIGNMENTS
SIOCAD  EQU     80H             ;SIO CHANNEL A DATA
SIOCBD  EQU     81H             ;SIO CHANNEL B DATA
SIOCAS  EQU     82H             ;SIO CHANNEL A COMMANDS/STATUS
SIOCBS  EQU     83H             ;SIO CHANNEL B COMMANDS/STATUS

        ;   PIO PORT ASSIGNMENTS
PIOAD   EQU     0F0H            ;PIO PORT A DATA
PIOBD   EQU     0F1H            ;PIO PORT B DATA
PIOAC   EQU     0F2H            ;PIO PORT A CONTROL
PIOBC   EQU     0F3H            ;PIO PORT B CONTROL

        ;   INTERRUPT VECTORS
SIOIV   EQU     0C0H            ;SIO INTERRUPT VECTOR
```

```
PIOIVA  EQU     ODOH            ;PIO PORT A INTERRUPT VECTOR
PIOIVB  EQU     OD2H            ;PIO PORT B INTERRUPT VECTOR

SC10E:
        LD      HL,PINIT        ;POINT TO INITIALIZATION ARRAY
        CALL    IPORTS          ;INITIALIZE PORTS

        JR      SC10E

PINIT:
        ;INITIALIZE Z80 CTC CHANNEL 0
        ;    RESET CHANNEL
        ;    OPERATE CHANNEL IN COUNTER MODE, DECREMENTING DOWN COUNTER
        ;     AFTER EACH POSITIVE (RISING) EDGE ON CLOCK INPUT.
        ;    SET INITIAL TIME CONSTANT TO 26 CLOCK CYCLES.
        ;    NOTE: CTC RELOADS TIME CONSTANT REGISTER INTO DOWN COUNTER
        ;     AUTOMATICALLY AFTER EACH COUNTDOWN TO 0.

        ;    THIS INITIALIZATION PRODUCES AN SIO CLOCK FOR 9600 BAUD
        ;     TRANSMISSION.
        ;    IT ASSUMES 4 MHZ CLOCK INPUT TO PIN 23, SO A COUNT OF
        ;      4,000,000/(16*9600)  = 26 WILLL GENERATE A 153,600
        ;      (16*9600) HZ SQUARE WAVE ON PIN 7 FOR SIO PINS 13 AND 14.

;       SIO IS OPERATING IN DIVIDE BY 16 MODE.
DB       2                  ;OUTPUT TWO BYTES
DB       CTC0               ;DESTINATION IS CHANNEL CONTROL REGISTER
DB       01010111B          ;BIT 0 = 1 (WRITE CHANNEL CONTROL WORD)
                            ;BIT 1 = 1 (RESET CHANNEL)
                            ;BIT 2 = 1 (TIME CONSTANT FOLLOWS)
                            ;BIT 3 = 0 (NOT USED IN COUNTER MODE)
                            ;BIT 4 = 1 (DECREMENT COUNTER ON
                            ;           POSITIVE CLOCK EDGE)
                            ;BIT 5 = 0 (NOT USED IN COUNTER MODE)
                            ;BIT 6 = 1 (COUNTER MODE)
                            ;BIT 7 = 0 (NO INTERRUPT)
DB       26                 ;TIMER COUNTDOWN VALUE FOR 9600 BAUD


;INITIALIZE Z80 SIO CHANNEL A FOR ASYNCHRONOUS SERIAL I/O.
;    SET INTERRUPT VECTOR (ALWAYS IN CHANNEL B) TO SIOIV
;    NO PARITY, 2 STOP BITS, 16 TIMES CLOCK.
;    RECEIVE AND TRANSMIT 8 BITS/CHAR, NO SPECIAL CONTROLS.
;    ENABLE TRANSMIT INTERRUPT, RECEIVE INTERRUPTS ON ALL CHARS;
;    PARITY OR STATUS DOES NOT AFFECT INTERRUPT VECTORS.


;        SET INTERRUPT VECTOR
DB       2                  ;OUTPUT TWO BYTES
DB       SIOCBS             ;DESTINATION IS COMMAND REGISTER B
DB       00000010B          ;SELECT WRITE REGISTER 2
DB       SIOIV              ;SET INTERRUPT VECTOR FOR SIO

;        INITIALIZE CHANNEL A
DB       9                  ;OUTPUT NINE BYTES
DB       SIOCAS             ;DESTINATION IS COMMAND REGISTER A
```

```
;       RESET THE CHANNEL
DB      00011000B       ;SELECT WRITE REGISTER 0
                        ;BITS 2,1,0 = 0 (WRITE REGISTER 0)
                        ;BITS 5,4,3 = 011 (CHANNEL RESET)
                        ;BITS 7,6 = 0 (DO NOT CARE)

;       INITIALIZE BAUD RATE CONTROL
;       NO PARITY, 2 STOP BITS, 16 TIMES CLOCK
DB      00000100B       ;SELECT WRITE REGISTER 4
DB      01001100B       ;BIT 0 = 0 (NO PARITY)
                        ;BIT 1 = 0 (DON'T CARE)
                        ;BITS 3,2 = 11 (2 STOP BITS)
                        ;BITS 5,4 = 00 (DON'T CARE)
                        ;BITS 7,6 = 01 (16 TIMES CLOCK)

;       INITIALIZE RECEIVE CONTROL
;       8 BITS PER CHARACTER, ENABLE RECEIVER, NO AUTO ENABLE
DB      00000011B       ;SELECT WRITE REGISTER 3
DB      11100001B       ;BIT 0 = 1 (RECEIVE ENABLE)
                        ;BITS 4,3,2,1 = 0 (DON'T CARE)
                        ;BIT 5 = 0 (NO AUTO ENABLE)
                        ;BIT 7,6 = 11 (RECEIVE 8 BITS/CHAR)

;       INITIALIZE TRANSMIT CONTROL
;       8 BITS PER CHARACTER, ENABLE TRANSMIT, NO BREAK OR CRC
DB      00000101B       ;SELECT WRITE REGISTER 5
DB      11101010B       ;BIT 0 = 0 (NO CRC ON TRANSMIT)
                        ;BIT 1 = 1 (REQUEST TO SEND)
                        ;BIT 2 = 0 (DON'T CARE)
                        ;BIT 3 = 1 (TRANSMIT ENABLE)
                        ;BIT 4 = 0 (DO NOT SEND BREAK)
                        ;BITS 6,5 = 11 (TRANSMIT 8 BITS/CHAR)
                        ;BIT 7 = 1 (DATA TERMINAL READY)


;INITIALIZE INTERRUPT CONTROL
;       RESET INTERRUPTS FIRST
;       ENABLE TRANSMIT INTERRUPT, RECEIVE INTERRUPTS ON ALL CHARS
;       NEITHER STATUS NOR PARITY ERRORS AFFECT INTERRUPT VECTOR
;       DO NOT CONTROL THE WAIT/READY OUTPUT LINE
DB      00010001B       ;SELECT WRITE REGISTER 1 AND
                        ; RESET EXTERNAL/STATUS INTERRUPTS
DB      00011010B       ;BIT 0 = 0 (NO EXTERNAL INTERRUPTS)
                        ;BIT 1 = 1 (ENABLE TRANSMIT INTERRUPT)
                        ;BIT 2 = 0 (STATUS DOES NOT AFFECT
                        ;           VECTOR)
                        ;BITS 4,3 = 11 (RECEIVE INTERRUPTS ON
                        ;               ALL CHARS, PARITY DOES NOT
                        ;               AFFECT VECTOR)
                        ;BITS 7,6,5 = 000 (NO WAIT/READY
                        ;                  FUNCTION)

;       TRANSMIT A NULL BYTE TO START INTERRUPT PROCESSING
DB      1               ;OUTPUT 1 BYTE
DB      SIOCAD          ;DESTINATION IS CHANNEL A DATA
DB      0               ;NULL CHARACTER (00 HEX)
```

```
;INITIALIZE Z80 PIO
;       PORT A - INPUT PORT WITH INTERRUPT ENABLED
;       PORT B - CONTROL PORT WITH INTERRUPT ENABLED.  AN INTERRUPT IS
;         GENERATED IF ANY OF BITS 0, 4, OR 7 BECOME 1
;

;       INITIALIZE PIO PORT A
DB      3                  ;OUTPUT 3 BYTES
DB      PIOAC              ;DESTINATION IS PORT A CONTROL
DB      PIOIVA             ;SET INTERRUPT VECTOR FOR PORT A
DB      10001111B          ;BITS 3,2,1,0 = 1111 (MODE SELECT)
                           ;BITS 5,4 = 00 (DON'T CARE)
                           ;BITS 7,6 = 01 (INPUT MODE)
DB      10000111B          ;BITS 3,2,1,0 = 0111 (INTERRUPT CONTROL)
                           ;BITS 6,5,4 = 000 (DON'T CARE)
                           ;BITS 7 = 1 (ENABLE INTERRUPTS)

;       INITIALIZE PIO PORT B
DB      4                  ;OUTPUT 4 BYTES
DB      PIOBC              ;DESTINATION IS PORT B CONTROL
DB      PIOIVB             ;SET INTERRUPT VECTOR FOR PORT B
DB      11001111B          ;BITS 3,2,1,0 = 1111 (MODE SELECT)
                           ;BITS 5,4 = 00 (DON'T CARE)
                           ;BITS 7,6 = 11 (CONTROL MODE)
DB      10110111B          ;BITS 3,2,1,0 = 0111 (INTERRUPT CONTROL)
                           ;BIT 4 = 1 (MASK FOLLOWS)
                           ;BIT 5 = 1 (ACTIVE STATE ON MONITORED
                           ; INPUT LINES IS 1 FOR AN INTERRUPT)
                           ;BIT 6 = 0 (INTERRUPT IF ANY OF THE
                           ; MONITORED INPUT LINES IS ACTIVE)
                           ;BIT 7 = 1 (ENABLE INTERRUPTS)
DB      10010001B          ;MONITOR INPUT BITS 0, 4, AND 7
                           ; FOR INTERRUPTS

;       END OF PORT INITIALIZATION DATA
DB      0                  ;TERMINATOR

END
```

Delay Milliseconds (DELAY)

Provides a delay of between 1 and 256 milliseconds, depending on the parameter supplied. A parameter value of 0 is interpreted as 256. The user must calculate the value CPMS (cycles per millisecond) to fit a particular computer. Typical values are 2000 for a 2 MHz clock, 4000 for a 4 MHz clock, and 6000 for a 6 MHz clock.

Procedure: The program simply counts down register B for the appropriate amount of time as determined by the user-supplied constant. Extra instructions account for the CALL instruction, RET instruction, and routine overhead without changing anything.

Registers Used: AF
Execution Time: 1 ms * (A)
Program Size: 51 bytes
Data Memory Required: None
Special Case: (A) = 0 causes a delay of 256 ms

Entry Conditions

Number of milliseconds to delay (1 to 256) in A

Exit Conditions

Returns after the specified delay with (A) = 0

Example

1. Data: (A) = number of milliseconds = $2A_{16}$ (42_{10})
   Result: Software delay of $2A_{16}$ milliseconds, with proper CPMS supplied by user

```
;                                                                       ;
;                                                                       ;
;                                                                       ;
;                                                                       ;
;       Title           Delay milliseconds                              ;
;       Name:           Delay                                           ;
;                                                                       ;
;                                                                       ;
;                                                                       ;
;       Purpose:        Delay from 1 to 256 milliseconds                ;
;                                                                       ;
;       Entry:          Register A = Number of milliseconds to delay    ;
;                                A 0 equals 256 milliseconds            ;
;                                                                       ;
;       Exit:           Returns to calling routine after the            ;
;                       specified delay                                 ;
;                                                                    ;
;        Registers used: AF                                          ;
;                                                                    ;
;        Time:           1 millisecond * Register A                  ;
;                                                                    ;
;        Size:           Program 51 bytes                            ;
;                                                                    ;

        ;EQUATES
        ;CYCLES PER MILLISECOND - USER-SUPPLIED
CPMS    EQU     4000    ;2000 = 2 MHZ CLOCK
                        ;4000 = 4 MHZ CLOCK
                        ;6000 = 6 MHZ CLOCK
                        ;      .
                        ;      .

        ;METHOD:
        ; THE ROUTINE IS DIVIDED INTO 2 PARTS. THE CALL TO
        ; THE "DLY" ROUTINE DELAYS EXACTLY 1 LESS THAN THE
        ; REQUIRED NUMBER OF MILLISECONDS. THE LAST ITERATION
        ; TAKES INTO ACCOUNT THE OVERHEAD TO CALL "DELAY" AND
        ; "DLY". THIS OVERHEAD IS:
        ;               17 CYCLES ==> CALL DELAY
        ;               11 CYCLES ==> PUSH BC
        ;               17 CYCLES ==> CALL DLY
        ;                4 CYCLES ==> DEC  A
        ;               11 CYCLES ==> RET  Z
        ;                7 CYCLES ==> LD   B,(CPMS/100)-1
        ;               10 CYCLES ==> POP  BC
        ;               13 CYCLES ==> LD   A,(DELAY)
        ;               10 CYCLES ==> RET
        ;               ------
        ;              100 CYCLES OVERHEAD
DELAY:
        ;DO ALL BUT THE LAST MILLISECOND
                                ;17 CYCLES FOR THE USER'S CALL
        PUSH    BC              ;11 CYCLES
        CALL    DLY             ;32 CYCLES TO RETURN FROM DLY

        ;DO 2 LESS THAN 1 MILLISECOND FOR OVERHEAD
        LD      B,+(CPMS/50)-2  ;7 CYCLES
                                ;-----
                                ;67 CYCLES

LDLP:
        JP      LDLY1           ;10 CYCLES
LDLY1:  JP      LDLY2           ;10 CYCLES
LDLY2:  JP      LDLY3           ;10 CYCLES
LDLY3:  ADD     A,0             ;7 CYCLES
        DJNZ    LDLP            ;13 CYCLES
                                ;---
                                ;50 CYCLES

        ;EXIT IN 33 CYCLES
        POP     BC              ;10 CYCLES
        LD      A,(DELAY)       ;13 CYCLES
        RET                             ;10 CYCLES
                                        ;---
                                        ;33 CYCLES


        ;****************************
        ;ROUTINE: DLY
        ;PURPOSE: DELAY ALL BUT LAST MILLISECOND
        ;ENTRY: REGISTER A = TOTAL NUMBER OF MILLISECONDS
        ;EXIT:  DELAY ALL BUT LAST MILLISECOND
        ;REGISTERS USED: AF,BC,HL
        ;****************************
DLY:
        DEC     A                       ;4 CYCLES
        RET     Z                       ;5 CYCLES (RETURN WHEN DONE 11 CYCLES)
        LD      B,+(CPMS/50)-1          ;7 CYCLES
                                        ;---
                                        ;16 CYCLES

DLP:
        JP      DLY1                    ;10 CYCLES
DLY1:   JP      DLY2                    ;10 CYCLES
DLY2:   JP      DLY3                    ;10 CYCLES
DLY3:   ADD     A,0                     ;7 CYCLES
        DJNZ    DLP                     ;13 CYCLES
                                        ;---
                                        ;50 CYCLES

        ;EXIT IN 34 CYCLES
        JP      DLY4                    ;10 CYCLES
DLY4:   JP      DLY5                    ;10 CYCLES
DLY5:   NOP                             ;4 CYCLES
        JP      DLY                     ;10 CYCLES
                                        ;---
                                        ;34 CYCLES


;                                                                       ;
;                                                                       ;
;       SAMPLE EXECUTION:                                               ;
;                                                                       ;
;                                                                       ;

SC10F:

        ;DELAY 10 SECONDS
        ;  CALL DELAY 40 TIMES AT 250 MILLISECONDS EACH
        LD      B,40                    ;40 TIMES (28 HEX)
QTRSCD:
        LD      A,250                   ;250 MILLISECONDS (FA HEX)
        CALL    DELAY
        DJNZ    QTRSCD                  ;CONTINUE UNTIL DONE

        JR      SC10F

        END
```

Unbuffered Input/Output Using an SIO (SINTIO) 11A

Performs interrupt-driven input and output using an SIO and single-character input and output buffers. Consists of the following subroutines:

1. INCH reads a character from the input buffer.

2. INST determines whether the input buffer is empty.

3. OUTCH writes a character into the output buffer.

4. OUTST determines whether the output buffer is full.

5. INIT initializes the SIO, the interrupt vectors, and the software flags. The flags are used to manage data transfers between the main program and the interrupt service routines.

The actual service routines are

1. RDHDLR responds to the input interrupt by reading a character from the SIO into the input buffer.

2. WRHDLR responds to the output interrupt by writing a character from the output buffer into the SIO.

Registers Used:

1. INCH: AF
2. INST: AF
3. OUTCH: AF
4. OUTST: AF
5. INIT: AF, BC, HL, I

Execution Time:

1. INCH: 72 cycles if a character is available
2. INST: 27 cycles
3. OUTCH: 150 cycles if the output buffer is not full and an output interrupt is expected; 75 additional cycles to send the data to the SIO if no output interrupt is expected.
4. OUTST: 27 cycles
5. INIT: 618 cycles
6. RDHDLR: 82 cycles
7. WRHDLR: 160 cycles

Program Size: 202 bytes

Data Memory Required: 5 bytes anywhere in RAM for the received data (address RECDAT), Receive Data flag (address RECDF), transmit data (address TRNDAT), Transmit Data flag (address TRNDF), and Output Interrupt Expected flag address OIE)

Procedures

1. INCH waits for a character to become available, clears the Data Ready flag (RECDF), and loads the character into the accumulator.

2. INST sets Carry from the Data Ready flag (RECDF).

3. OUTCH waits for the output buffer to empty, stores the character in the buffer, and sets the Character Available flag (TRNDF). If no output interrupt is expected (i.e., the interrupt has been reset because it occurred when no data was available), OUTCH sends the data to the SIO immediately.

4. OUTST sets Carry from the Character Available flag (TRNDF).

5. INIT clears the software flags, sets up the interrupt vectors, and initializes the SIO by placing the appropriate values in its control registers. See Subroutine 10E for more details about initializing SIOs.

6. RDHDLR reads the data, saves it in the input buffer, and sets the Data Ready flag (RECDF).

7. WRHDLR determines whether data is available. If not, it simply resets the output interrupt. If data is available, the program sends it to the SIO and clears the Character Available flag (TRNDF).

The special problem here is that an output interrupt may occur when no data is available. It cannot be ignored or it will assert itself indefinitely, causing an endless loop. The solution is simply to reset the SIO's transmit interrupt without sending any data.

But now a new problem arises when output data becomes available. That is, since the interrupt has been reset, it obviously cannot inform the system that the SIO is ready to transmit. The solution is to have a flag that indicates (with a 0 value) that the output interrupt has occurred without being serviced. This flag is called OIE (Output Interrupt Expected).

The initialization routine clears OIE (since the SIO surely starts out ready to transmit). The output service routine clears it when an output interrupt occurs that cannot be serviced (no data is available) and sets it after sending data to the SIO (in case it might have been cleared). Now the output routine OUTCH can check OIE to determine whether an output interrupt is expected. If not, OUTCH simply sends the data immediately.

Note that an SIO interrupt can be reset without actually sending any data. This is not possible with a PIO (see Subroutine 11B), so the procedure there is slightly different.

Unserviceable interrupts occur only with output devices, since input devices always have data ready to transfer when they request service. Thus, output devices cause more initialization and sequencing problems in interrupt-driven systems than do input devices.

Entry Conditions

1. INCH: none
2. INST: none
3. OUTCH: character to transmit in A
4. OUTST: none
5. INIT: none

Exit Conditions

1. INCH: character in A
2. INST: Carry = 0 if input buffer empty, 1 if full
3. OUTCH: none
4. OUTST: Carry = 0 if output buffer empty, 1 if full
5. INIT: none

```
;                                                                ;
;                                                                ;
;                                                                ;
;                                                                ;
;       Title           Simple interrupt input and output using an SIO ;
;                       and single character buffers             ;
;       Name:           SINTIO                                   ;
;                                                                ;
;                                                                       ;
;                                                                       ;
;       Purpose:        This program consists of 5 subroutines which    ;
;                       perform-interrupt driven input and output using ;
;                       an SIO.                                         ;
;                                                                       ;
;                       INCH                                            ;
;                         Read a character                              ;
;                       INST                                            ;
;                         Determine input status (whether input         ;
;                         buffer is empty)                              ;
;                       OUTCH                                           ;
;                         Write a character                             ;
;                       OUTST                                           ;
;                         Determine output status (whether output       ;
;                         buffer is full)                               ;
;                       INIT                                            ;
;                         Initialize SIO and interrupt system           ;
;                                                                       ;
;       Entry:          INCH                                            ;
;                         No parameters                                 ;
;                       INST                                            ;
;                         No parameters                                 ;
;                       OUTCH                                           ;
;                         Register A = character to transmit            ;
;                       OUTST                                           ;
;                         No parameters                                 ;
;                       INIT                                            ;
;                         No parameters                                 ;
;                                                                       ;
;       Exit:           INCH                                            ;
;                         Register A = character                        ;
;                       INST                                            ;
;                         Carry = 0 if input buffer is empty,           ;
;                         1 if character is available                   ;
;                       OUTCH                                           ;
;                         No parameters                                 ;
;                       OUTST                                           ;
;                         Carry = 0 if output buffer is not             ;
;                         full, 1 if it is full                         ;
;                       INIT                                            ;
;                         No parameters                                 ;
;                                                                       ;
;       Registers used: INCH - AF                                       ;
;                       INST - AF                                       ;
;                       OUTCH - AF                                      ;
;                       OUTST - AF                                      ;
;                       INIT - AF,BC,HL,I                               ;
;                                                                       ;
;       Time:           INCH                                            ;
;                         72 cycles if a character is available         ;
;                       INST                                            ;
;                         27 cycles                                     ;
;                       OUTCH                                       ;
;                         150 cycles if output buffer is not full   ;
;                           and output interrupt is expected        ;
;                       OUTST                                       ;
;                         27 cycles                                 ;
;                       INIT                                        ;
;                         618 cycles                                ;
;                       RDHDLR                                      ;
;                         82 cycles                                 ;
;                       WRHDLR                                      ;
;                         160 cycles                                ;
;                                                                   ;
;       Size:           Program 202 bytes                           ;
;                       Data      5 bytes                           ;
;                                                                   ;
;                                                                   ;


        ;SIO EQUATES
        ; SIO IS PROGRAMMED FOR:
        ;  ASYNCHRONOUS OPERATION
        ;  16 X BAUD RATE
        ;  8-BIT CHARACTERS
        ;  1 1/2 STOP BITS
        ;ARBITRARY SIO PORT ADDRESSES
SIOCAD  EQU     1CH             ;SIO CHANNEL A DATA
SIOCBD  EQU     1EH             ;SIO CHANNEL B DATA
SIOCAS  EQU     1DH             ;SIO CHANNEL A COMMANDS/STATUS
SIOCBS  EQU     1FH             ;SIO CHANNEL B COMMANDS/STATUS
SIOIV   EQU     8000H           ;INTERRUPT VECTOR
SIOWV   EQU     SIOIV+8         ;SIO CHANNEL A WRITE INTERRUPT VECTOR
SIOEV   EQU     SIOIV+10        ;SIO CHANNEL A EXTERNAL/STATUS INTERRUPT
                                ; VECTOR
SIORV   EQU     SIOIV+12        ;SIO CHANNEL A READ INTERRUPT VECTOR
SIOSV   EQU     SIOIV+14        ;SIO CHANNEL A SPECIAL RECEIVE INTERRUPT
                                ; VECTOR

        ;READ CHARACTER
INCH:
        CALL    INST            ;GET INPUT STATUS
        JR      NC,INCH         ;WAIT IF NO CHARACTER AVAILABLE
        DI                      ;DISABLE INTERRUPTS
        SUB     A
        LD      (RECDF),A       ;INDICATE INPUT BUFFER EMPTY
        LD      A,(RECDAT)      ;GET CHARACTER FROM INPUT BUFFER
        EI                      ;ENABLE INTERRUPTS
        RET

        ;RETURN INPUT STATUS (CARRY = 1 IF INPUT DATA IS AVAILABLE)
INST:
        LD      A,(RECDF)       ;GET DATA READY FLAG
        RRA                     ;SET CARRY FROM DATA READY FLAG
                                ; IF CARRY = 1, CHARACTER IS AVAILABLE
        RET
        ;WRITE CHARACTER
```

```
OUTCH:
        PUSH    AF              ;SAVE CHARACTER TO WRITE

        ;WAIT FOR CHARACTER BUFFER TO EMPTY, THEN STORE NEXT CHARACTER
WAITOC:
        CALL    OUTST           ;GET OUTPUT STATUS
        JR      C,WAITOC        ;WAIT IF OUTPUT BUFFER IS FULL
        DI                      ;DISABLE INTERRUPTS WHILE LOOKING AT
                                ; SOFTWARE FLAGS
        POP     AF              ;GET CHARACTER
        LD      (TRNDAT),A      ;STORE CHARACTER IN OUTPUT BUFFER
        LD      A,0FFH          ;INDICATE OUTPUT BUFFER FULL
        LD      (TRNDF),A
        LD      A,(OIE)         ;TEST OUTPUT INTERRUPT EXPECTED FLAG
        OR      A
        CALL    Z,OUTDAT        ;OUTPUT CHARACTER IMMEDIATELY IF
                                ; NO OUTPUT INTERRUPT EXPECTED
        EI                      ;ENABLE INTERRUPTS
        RET

        ;OUTPUT STATUS (CARRY = 1 IF OUTPUT BUFFER IS FULL)
OUTST:
        LD      A,(TRNDF)       ;GET TRANSMIT FLAG
        RRA                     ;SET CARRY FROM TRANSMIT FLAG
        RET                     ; CARRY = 1 IF BUFFER FULL

        ;INITIALIZE INTERRUPT SYSTEM AND SIO
INIT:
        DI                      ;DISABLE INTERRUPTS FOR INITIALIZATION

        ;INITIALIZE SOFTWARE FLAGS
        SUB     A
        LD      (RECDF),A       ;NO INPUT DATA AVAILABLE
        LD      (TRNDF),A       ;OUTPUT BUFFER EMPTY
        LD      (OIE),A         ;NO OUTPUT INTERRUPT EXPECTED
                                ; SIO IS READY TO TRANSMIT INITIALLY

        ;INITIALIZE INTERRUPT VECTORS
        LD      A,SIOIV SHR 8   ;GET INTERRUPT PAGE NUMBER
        LD      I,A             ;SET INTERRUPT VECTOR IN Z80
        IM      2               ;INTERRUPT MODE 2 - VECTORS IN TABLE
                                ; ON INTERRUPT PAGE
        LD      HL,RDHDLR       ;STORE READ VECTOR (INPUT INTERRUPT)
        LD      (SIORV),HL
        LD      HL,WRHDLR       ;STORE WRITE VECTOR (OUTPUT INTERRUPT)
        LD      (SIOWV),HL
        LD      HL,EXHDLR       ;STORE EXTERNAL/STATUS VECTOR
        LD      (SIOEV),HL
        LD      HL,REHDLR       ;STORE RECEIVE ERROR VECTOR
        LD      (SIOSV),HL

        ;INITIALIZE SIO
        LD      HL,SIOINT       ;GET BASE OF INITIALIZATION ARRAY
        CALL    IPORTS          ;INITIALIZE SIO
        EI                      ;ENABLE INTERRUPTS
        RET
        ;INPUT (READ) INTERRUPT HANDLER
RDHDLR:
        PUSH    AF                  ;SAVE AF
RD1:    IN      A,(SIOCAD)          ;READ DATA FROM SIO
        LD      (RECDAT),A          ;SAVE DATA IN INPUT BUFFER
        LD      A,0FFH
        LD      (RECDF),A           ;INDICATE INPUT DATA AVAILABLE
        POP     AF                  ;RESTORE AF
        EI                          ;REENABLE INTERRUPTS
        RETI

        ;OUTPUT (WRITE) INTERRUPT HANDLER
WRHDLR:
        PUSH    AF
        LD      A.(TRNDF)           ;GET DATA AVAILABLE FLAG
        RRA
        JR      NC,NODATA           ;JUMP IF NO DATA TO TRANSMIT
        CALL    OUTDAT              ;OUTPUT DATA TO SIO
        JR      WRDONE

        ;IF AN OUTPUT INTERRUPT OCCURS WHEN NO DATA IS AVAILABLE.
        ; WE MUST RESET IT TO AVOID AN ENDLESS LOOP. LATER. WHEN A
        ; CHARACTER BECOMES AVAILABLE, WE NEED TO KNOW THAT AN OUTPUT
        ; INTERRUPT HAS OCCURRED WITHOUT BEING SERVICED.  THE KEY HERE
        ; IS THE OUTPUT INTERRUPT EXPECTED FLAG OIE.  THIS FLAG IS
        ; CLEARED WHEN AN OUTPUT INTERRUPT HAS OCCURRED BUT HAS NOT
        ; BEEN SERVICED.  IT IS ALSO CLEARED INITIALLY SINCE THE
        ; SIO STARTS OUT READY.  OIE IS SET WHENEVER DATA IS ACTUALLY
        ; SENT TO THE SIO. THUS THE OUTPUT ROUTINE OUTCH CAN CHECK
        ; OIE TO DETERMINE WHETHER TO SEND THE DATA IMMEDIATELY
        ; OR WAIT FOR AN OUTPUT INTERRUPT.
        ;THE PROBLEM IS THAT AN OUTPUT DEVICE MAY REQUEST SERVICE BEFORE
        ; THE COMPUTER HAS ANYTHING TO SEND (UNLIKE AN INPUT DEVICE
        ; THAT HAS DATA WHEN IT REQUESTS SERVICE).  THE OIE FLAG
        ; SOLVES THE PROBLEM OF AN UNSERVICED OUTPUT INTERRUPT ASSERTING
        ; ITSELF REPEATEDLY, WHILE STILL ENSURING THE RECOGNITION OF
        ; OUTPUT INTERRUPTS.
NODATA:
        SUB     A
        LD      (OIE),A             ;DO NOT EXPECT AN INTERRUPT
        OUT     (SIOCAS),A          ;SELECT REGISTER 0
        LD      A,00101000B         ;RESET SIO TRANSMITTER INTERRUPT
        OUT     (SIOCAS),A

WRDONE:
        POP     AF                  ;RESTORE AF
        EI
        RETI

        ;EXTERNAL/STATUS CHANGED INTERRUPT HANDLER
EXHDLR:
        PUSH    AF
        LD      A,00010000B         ;RESET STATUS INTERRUPT
        OUT     (SIOCAS).A
        EI                          ;DCD OR CTS CHANGED STATE, OR A BREAK
        POP     AF                  ; WAS DETECTED
        RETI                        ;SERVICE HERE IF NECESSARY
        ;SPECIAL RECEIVE ERROR INTERRUPT
REHDLR:
        PUSH    AF
        LD      A,00110000B         ;RESET RECEIVE ERROR INTERRUPT
        OUT     (SIOCAS),A
        EI                          ;FRAMING ERROR OR OVERRUN ERROR
        POP     AF                  ; OCCURRED
        RETI                        ;SERVICE HERE IF NECESSARY


        ;************************************
        ;ROUTINE: OUTDAT
        ;PURPOSE: SEND CHARACTER TO SIO
        ;ENTRY: TRNDAT = CHARACTER
        ;EXIT:  NONE
        ;REGISTERS USED: AF
        ;*************************************
OUTDAT:
        LD      A,(TRNDAT)          ;GET DATA FROM OUTPUT BUFFER
        OUT     (SIOCAD).A          ;SEND DATA TO SIO
        SUB     A                   ;INDICATE OUTPUT BUFFER EMPTY
        LD      (TRNDF),A
        DEC     A                   ;INDICATE OUTPUT INTERRUPT EXPECTED
        LD      (OIE),A             ; OIE = FF HEX
        RET

        ;************************************
        ;ROUTINE: IPORTS
        ;PURPOSE: INITIALIZE I/O PORTS
        ;ENTRY: HL = BASE ADDRESS OF INITIALIZATION ARRAY
        ;EXIT:  DATA OUTPUT TO PORTS
        ;REGISTERS USED: AF,BC,HL
        ;************************************
IPORTS:
        ;GET NUMBER OF DATA BYTES TO SEND TO CURRENT PORT
        ;EXIT IF NUMBER OF BYTES IS 0, INDICATING TERMINATOR
        LD      A,(HL)              ;GET NUMBER OF BYTES
        OR      A                   ;TEST FOR ZERO (TERMINATOR)
        RET     Z                   ;RETURN IF NUMBER OF BYTES = 0
        LD      B,A
        INC     HL                  ;POINT TO PORT ADDRESS (NEXT BYTE)

        ;C = PORT ADDRESS
        ;HL = BASE ADDRESS OF OUTPUT DATA
        LD      C,(HL)              ;GET PORT ADDRESS
        INC     HL                  ;POINT TO FIRST DATA VALUE (NEXT BYTE)

        ;OUTPUT DATA AND CONTINUE TO NEXT PORT
        OTIR                        ;SEND DATA VALUES TO PORT
        JR      IPORTS              ;CONTINUE TO NEXT PORT ENTRY

        ;SIO INITIALIZATION DATA
```

```
SIOINT:
        ;RESET CHANNEL A
        DB      1                   ;OUTPUT 1 BYTE
        DB      SIOCAS              ;DESTINATION IS CHANNEL A COMMAND/STATUS
        DB      00011000B           ;SELECT WRITE REGISTER 0
                                    ;BITS 2,1,0 = 0 (WRITE REGISTER 0)
                                    ;BITS 5.4.3 = 011 (CHANNEL RESET)
                                    ;BITS 7,6 = 0 (DO NOT CARE)

        ;SET INTERRUPT VECTOR AND ALLOW STATUS TO AFFECT IT
        DB      4                   ;OUTPUT 2 BYTES
        DB      SIOCBS              ;DESTINATION IS COMMAND REGISTER B
        DB      00000010B           ;SELECT WRITE REGISTER 2
        DB      SIOIV AND OFFH      ;SET INTERRUPT VECTOR FOR SIO
        DB      00000001B           ;SELECT WRITE REGISTER 1
        DB      00000100B           ;ALLOW STATUS TO AFFECT VECTOR

        ;INITIALIZE CHANNEL A
        DB      8                   ;OUTPUT 8 BYTES
        DB      SIOCAS              ;DESTINATION IS COMMAND REGISTER A

        ;INITIALIZE BAUD RATE CONTROL
        DB      00010100B           ;SELECT WRITE REGISTER 4
                                    ; RESET EXTERNAL/STATUS INTERRUPT
        DB      01001000B           ;BIT 0 = 0 (NO PARITY)
                                    ;BIT 1 = 0 (DON'T CARE)
                                    ;BITS 3.2 = 10 (1 1/2 STOP BITS)
                                    ;BITS 5,4 = 00 (DON'T CARE)
                                    ;BITS 7,6 = 01 (16 TIMES CLOCK)

        ;INITIALIZE RECEIVE CONTROL
        DB      00000011B           ;SELECT WRITE REGISTER 3
        DB      11000001B           ;BIT 0 = 1 (RECEIVE ENABLE)
                                    ;BITS 4,3,2,1 = 0 (DON'T CARE)
                                    ;BIT 5 = 0 (NO AUTO ENABLE)
                                    ;BIT 7,6 = 11 (RECEIVE 8 BITS/CHAR)

        ;INITIALIZE TRANSMIT CONTROL
        DB      00000101B           ;SELECT WRITE REGISTER 5
        DB      11101010B           ;BIT 0 = 0 (NO CRC ON TRANSMIT)
                                    ;BIT 1 = 1 (REQUEST TO SEND)
                                    ;BIT 2 = 0 (DON'T CARE)
                                    ;BIT 3 = 1 (TRANSMIT ENABLE)
                                    ;BIT 4 = 0 (DO NOT SEND BREAK)
                                    ;BITS 6,5 = 11 (TRANSMIT 8 BITS/CHAR)
                                    ;BIT 7 = 1 (DATA TERMINAL READY)
        DB      00000001B           ;SELECT WRITE REGISTER 1
        DB      00011011B           ;BIT 0 = 1 (EXTERNAL INTERRUPTS)
                                    ;BIT 1 = 1 (ENABLE TRANSMIT INTERRUPT)
                                    ;BIT 2 = 0 (DO NOT CARE)
                                    ;BITS 4,3 = 11 (RECEIVE INTERRUPTS ON
                                    ; ALL CHARS, PARITY DOES NOT AFFECT
                                    ; VECTOR)
                                    ;BITS 7,6,5 = 000 (NO WAIT/READY
                                    ; FUNCTION)
        DB      0                   ;TERMINATOR FOR INITIALIZATION ARRAY

        ;DATA SECTION
RECDAT: DS      1                   ;RECEIVE DATA
RECDF:  DS      1                   ;RECEIVE DATA FLAG
                                    ; (0 = NO DATA, FF = DATA AVAILABLE)
TRNDAT: DS      1                   ;TRANSMIT DATA
TRNDF:  DS      1                   ;TRANSMIT DATA FLAG
                                    ; (0 = BUFFER EMPTY, FF = BUFFER FULL)
OIE:    DS      1                   ;OUTPUT INTERRUPT EXPECTED
                                    ; (0 = NO INTERRUPT EXPECTED,
                                    ; FF = INTERRUPT EXPECTED)


;                                                                       ;
;                                                                       ;
;       SAMPLE EXECUTION:                                               ;
;                                                                       ;
;                                                                       ;


        ;CHARACTER EQUATES
ESCAPE  EQU     1BH                 ;ASCII ESCAPE CHARACTER
TESTCH  EQU     'A'                 ;TEST CHARACTER = A

SC11A:
        CALL    INIT                ;INITIALIZE SIO, INTERRUPT SYSTEM

        ;SIMPLE EXAMPLE - READ AND ECHO CHARACTERS
        ; UNTIL AN ESC IS RECEIVED
LOOP:
        CALL    INCH                ;READ CHARACTER
        PUSH    AF
        CALL    OUTCH               ;ECHO CHARACTER
        POP     AF
        CP      ESCAPE              ;IS CHARACTER AN ESCAPE?
        JR      NZ,LOOP             ;STAY IN LOOP IF NOT

        ;AN ASYNCHRONOUS EXAMPLE
        ; OUTPUT "A" TO CONSOLE CONTINUOUSLY, BUT ALSO LOOK AT
        ; INPUT SIDE, READING AND ECHOING ANY INPUT CHARACTERS
ASYNLP:
        ;OUTPUT AN "A" IF OUTPUT IS NOT BUSY
        CALL    OUTST               ;IS OUTPUT BUSY?
        JR      C,ASYNLP            ;JUMP IF IT IS
        LD      A,TESTCH
        CALL    OUTCH               ;OUTPUT TEST CHARACTER

        ;CHECK INPUT PORT
        ;ECHO CHARACTER IF ONE IS AVAILABLE
        ;EXIT ON ESCAPE CHARACTER
        CALL    INST                ;IS INPUT DATA AVAILABLE?
        JR      NC,ASYNLP           ;JUMP IF NOT (SEND ANOTHER "A")
        CALL    INCH                ;GET CHARACTER
        CP      ESCAPE              ;IS IT AN ESCAPE?
        JR      Z,DONE              ;BRANCH IF IT IS
        CALL    OUTCH               ;ELSE ECHO CHARACTER
        JP      ASYNLP              ;AND CONTINUE

DONE:
        JP      LOOP

        END
```

Unbuffered Input/Output Using a PIO (PINTIO) 11B

Performs interrupt-driven input and output using a PIO and single-character input and output buffers. It consists of the following subroutines:

1. INCH reads a character from the input buffer.

2. INST determines whether the input buffer is empty.

3. OUTCH writes a character into the output buffer.

4. OUTST determines whether the output buffer is full.

5. INIT initializes the PIO, the interrupt vectors, and the software flags. The flags are used to manage data transfers between the main program and the interrupt service routines.

The actual service routines are

1. RDHDLR responds to the input interrupt by reading a character from the PIO into the input buffer.

2. WRHDLR responds to the output interrupt by writing a character from the output buffer into the PIO.

Registers Used:

1. INCH: AF
2. INST: AF
3. OUTCH: AF
4. OUTST: AF
5. INIT: AF, BC, HL, I

Execution Time:

1. INCH: 72 cycles if a character is available
2. INST: 27 cycles
3. OUTCH: 150 cycles if the output buffer is not full and an output interrupt is expected; 93 additional cycles to send the data to the PIO if no output interrupt is expected.
4. OUTST: 27 cycles
5. INIT: 377 cycles
6. RDHDLR: 82 cycles
7. WRHDLR: 178 cycles

Program Size: 166 bytes

Data Memory Required: 5 bytes anywhere in RAM for the received data (address RECDAT), Receive Data flag (address RECDF), transmit data (address TRNDAT), Transmit Data flag (address TRNDF), and Output Interrupt Expected flag (address OIE)

Procedures

1. INCH waits for a character to become available, clears the Data Ready flag (RECDF), and loads the character into the accumulator.

2. INST sets Carry from the Data Ready flag (RECDF).

3. OUTCH waits for the output buffer to empty, stores the character in the buffer, and sets the Character Available flag (TRNDF). If no output interrupt is expected (i.e., the interrupt has been disabled because it occurred when no data was available), OUTCH sends the data to the PIO immediately.

4. OUTST sets Carry from the Character Available flag (TRNDF).

5. INIT clears the software flags, sets up the interrupt vectors, and initializes the PIO by loading its control registers and interrupt vector. See Chapter 1 and Subroutine 10E for more details about initializing PIOs.

6. RDHDLR reads the data, saves it in the input buffer, and sets the Data Ready flag (RECDF).

7. WRHDLR determines whether data is available. If not, it simply disables the output (PIO port B) interrupt. If data is available, WRHDLR sends it to the PIO and clears the Character Available flag (TRNDF).

The special problem here is that an output interrupt may occur when no data is available. It cannot simply be ignored or it will assert itself indefinitely, causing an endless loop. The solution is simply to disable the output interrupt from PIO port B.

But now a new problem arises when output data becomes available. That is, since the interrupt has been disabled, it obviously cannot inform the system that the output device is ready for data. The solution is to have a flag that indicates (with a 0 value) that the output interrupt has occurred without being serviced. This flag is called OIE (Output Interrupt Expected).

The initialization routine clears OIE (since the output device surely starts out ready for data). The output service routine clears it when an output interrupt occurs that cannot be serviced (no data is available) and sets it after sending data to the PIO (in case it might have been cleared). Now the output routine OUTCH can check OIE to determine whether an output interrupt is expected. If not, OUTCH simply sends the data immediately.

Note that a PIO interrupt cannot be cleared without actually sending any data. This is possible with an SIO (see Subroutine 11A), so the procedure there is slightly different.

Unserviceable interrupts occur only with output devices, since input devices always have data ready to transfer when they request service. Thus, output devices cause more initialization and sequencing problems in interrupt-driven systems than do input devices.

Entry Conditions

1. INCH: none
2. INST: none
3. OUTCH: character to transmit in A
4. OUTST: none
5. INIT: none

Exit Conditions

1. INCH: character in A
2. INST: Carry = 0 if input buffer empty, 1 if full
3. OUTCH: none
4. OUTST: Carry = 0 if output buffer empty, 1 if full
5. INIT: none

```
;                                                                       ;
;                                                                       ;
;                                                                       ;
;                                                                       ;
;       Title           Simple interrupt input and output using a Z80   ;
;                       PIO and single character buffers                ;
;       Name:           PINTIO                                          ;
;                                                                        ;
;                                                                        ;

;                                                                        ;
;       Purpose:        This program consists of 5 subroutines which     ;
;                       perform interrupt driven input and output using  ;
;                       a Z80 PIO.                                        ;
;                                                                        ;
;                       INCH                                             ;
;                         Read a character                               ;
;                       INST                                             ;
;                         Determine input status (whether input          ;
;                         buffer is empty)                               ;
;                       OUTCH                                            ;
;                         Write a character                              ;
;                       OUTST                                            ;
;                         Determine output status (whether output        ;
;                         buffer is full)                                ;
;                       INIT                                             ;
;                         Initialize PIO and interrupt system            ;
;                                                                        ;
;       Entry:          INCH                                             ;
;                         No parameters                                  ;
;                       INST                                             ;
;                         No parameters                                  ;
;                       OUTCH                                            ;
;                         Register A = character to transmit             ;
;                       OUTST                                            ;
;                         No parameters                                  ;
;                       INIT                                             ;
;                         No parameters                                  ;
;                                                                        ;
;       Exit:           INCH                                             ;
;                         Register A = character                         ;
;                       INST                                             ;
;                         Carry = 0 if input buffer is empty,            ;
;                         1 if character is available                    ;
;                       OUTCH                                            ;
;                         No parameters                                  ;
;                       OUTST                                            ;
;                         Carry = 0 if output buffer is not              ;
;                         full, 1 if it is full                          ;
;                       INIT                                             ;
;                         No parameters                                  ;
;                                                                        ;
;       Registers used: INCH                                             ;
;                         A,F                                            ;
;                       INST                                             ;
;                         A,F                                            ;
;                       OUTCH                                            ;
;                         A,F                                            ;
;                       OUTST                                            ;
;                         A,F                                            ;
;                       INIT                                             ;
;                         A,F,BC,HL,I                                    ;
;                                                                        ;
;       Time:           INCH                                          ;
;                         72 cycles if a character is available       ;
;                       INST                                          ;
;                         27 cycles                                   ;
;                       OUTCH                                         ;
;                         150 cycles if output buffer is not full     ;
;                           and output interrupt is expected          ;
;                       OUTST                                         ;
;                         27 cycles                                   ;
;                       INIT                                          ;
;                         377 cycles                                  ;
;                       RDHDLR                                        ;
;                         82 cycles                                   ;
;                       WRHDLR                                        ;
;                         178 cycles                                  ;
;                                                                     ;
;       Size:           Program 166 bytes                             ;
;                       Data      5 bytes                             ;
;                                                                     ;
;                                                                     ;

        ;PIO EQUATES
        ; PIO IS PROGRAMMED FOR:
        ;  PORT A INPUT
        ;  PORT B OUTPUT
        ;ARBITRARY PIO PORT ADDRESSES
PIOAD   EQU     90H             ;PORT A DATA
PIOAC   EQU     91H             ;PORT A CONTROL
PIOBD   EQU     92H             ;PORT B DATA
PIOBC   EQU     93H             ;PORT B CONTROL
INTRPV  EQU     8000H           ;BASE OF INTERRUPT VECTORS
PIOIVA  EQU     INTRPV          ;INTERRUPT VECTOR FOR PORT A
PIOIVB  EQU     INTRPV+2        ;INTERRUPT VECTOR FOR PORT B

        ;READ CHARACTER
INCH:
        CALL    INST            ;GET INPUT STATUS
        JR      NC,INCH         ;WAIT IF NO CHARACTER AVAILABLE
        DI                      ;DISABLE INTERRUPTS
        SUB     A
        LD      (RECDF),A       ;INDICATE INPUT BUFFER EMPTY
        LD      A,(RECDAT)      ;GET CHARACTER FROM INPUT BUFFER
        EI                      ;REENABLE INTERRUPTS
        RET

        ;RETURN INPUT STATUS (CARRY = 1 IF INPUT DATA IS AVAILABLE)
INST:
        LD      A,(RECDF)       ;GET DATA READY FLAG
        RRA                     ;SET CARRY FROM DATA READY FLAG
                                ; IF CARRY = 1, CHARACTER IS AVAILABLE
        RET

        ;WRITE CHARACTER

OUTCH:
        PUSH    AF              ;SAVE CHARACTER TO WRITE
        ;WAIT FOR CHARACTER BUFFER TO EMPTY, THEN STORE NEXT CHARACTER
WAITOC:
        CALL    OUTST           ;GET OUTPUT STATUS
        JR      C,WAITOC        ;WAIT IF OUTPUT BUFFER IS FULL
        DI                      ;DISABLE INTERRUPTS WHILE LOOKING AT
                                ; SOFTWARE FLAGS
        POP     AF              ;GET CHARACTER
        LD      (TRNDAT),A      ;STORE CHARACTER IN OUTPUT BUFFER
        LD      A,0FFH          ;INDICATE OUTPUT BUFFER FULL
        LD      (TRNDF).A
        LD      A,(OIE)         ;TEST OUTPUT INTERRUPT EXPECTED FLAG
        OR      A
        CALL    Z,OUTDAT        ;OUTPUT CHARACTER IMMEDIATELY IF
                                ; NO OUTPUT INTERRUPT EXPECTED
        EI                      ;ENABLE INTERRUPTS
        RET

        ;OUTPUT STATUS (CARRY = 1 IF OUTPUT BUFFER IS FULL)
OUTST:
        LD      A,(TRNDF)       ;GET TRANSMIT FLAG
        RRA                     ;SET CARRY FROM TRANSMIT FLAG
        RET                     ; CARRY = 1 IF OUTPUT BUFFER FULL

        ;INITIALIZE PIO AND INTERRUPT SYSTEM
INIT:
        DI                      ;DISABLE INTERRUPTS

        ;INITIALIZE SOFTWARE FLAGS
        SUB     A
        LD      (RECDF),A       ;NO INPUT DATA AVAILABLE
        LD      (TRNDF),A       ;OUTPUT BUFFER EMPTY
        LD      (OIE),A         ;NO OUTPUT INTERRUPT EXPECTED
                                ; DEVICE IS READY INITIALLY

        ;INITIALIZE INTERRUPT VECTORS
        LD      A,INTRPV SHR 8  ;GET HIGH BYTE OF INTERRUPT PAGE
        LD      I,A             ;SET INTERRUPT VECTOR IN Z80
        IM      2               ;INTERRUPT MODE 2 - VECTORS IN TABLE
                                ; ON INTERRUPT PAGE
        LD      HL,RDHDLR       ;STORE READ VECTOR (INPUT INTERRUPT)
        LD      (PIOIVA),HL
        LD      HL,WRHDLR       ;STORE WRITE VECTOR (OUTPUT INTERRUPT)
        LD      (PIOIVB),HL

        ;INITIALIZE PIO
        LD      HL,PIOINT       ;BASE ADDRESS OF INITIALIZATION ARRAY
        CALL    IPORTS          ;INITIALIZE PIO
        EI                      ; ENABLE INTERRUPTS
        RET

        ;INPUT (READ) INTERRUPT HANDLER
RDHDLR:
        PUSH    AF
        IN      A,(PIOAD)       ;READ DATA FROM PIO
        LD      (RECDAT),A      ;SAVE DATA IN INPUT BUFFER
        LD      A,0FFH          ;INDICATE INPUT DATA AVAILABLE
        LD      (RECDF),A
        POP     AF
        EI                      ;REENABLE INTERRUPTS
        RETI

        ;OUTPUT (WRITE) INTERRUPT HANDLER
WRHDLR:
        PUSH    AF
        LD      A,(TRNDF)       ;GET DATA AVAILABLE FLAG
        RRA
        JR      NC,NODATA       ;JUMP IF NO DATA TO TRANSMIT
        CALL    OUTDAT          ;OUTPUT DATA TO PIO
        JR      WRDONE

        ;IF AN OUTPUT INTERRUPT OCCURS WHEN NO DATA IS AVAILABLE,
        ; WE MUST DISABLE IT TO AVOID AN ENDLESS LOOP.  LATER, WHEN A
        ; CHARACTER BECOMES AVAILABLE, WE NEED TO KNOW THAT AN OUTPUT
        ; INTERRUPT HAS OCCURRED WITHOUT BEING SERVICED.  THE KEY HERE
        ; IS THE OUTPUT INTERRUPT EXPECTED FLAG OIE.  THIS FLAG IS
        ; CLEARED WHEN AN OUTPUT INTERRUPT HAS OCCURRED BUT HAS NOT
        ; BEEN SERVICED. IT IS ALSO CLEARED INITIALLY SINCE THE
        ; OUTPUT DEVICE IS ASSUMED TO START OUT READY.  OIE IS SET
        ; WHENEVER DATA IS ACTUALLY SENT TO THE PIO. THUS THE OUTPUT ROUTINE
        ; OUTCH CAN CHECK OIE TO DETERMINE WHETHER TO SEND THE DATA
        ; IMMEDIATELY OR WAIT FOR AN OUTPUT INTERRUPT.
        ;THE PROBLEM IS THAT AN OUTPUT DEVICE MAY REQUEST SERVICE BEFORE
        ; THE COMPUTER HAS ANYTHING TO SEND (UNLIKE AN INPUT DEVICE
        ; THAT HAS DATA WHEN IT REQUESTS SERVICE).  THE OIE FLAG SOLVES
        ; THE PROBLEM OF AN UNSERVICED OUTPUT INTERRUPT ASSERTING ITSELF
        ; REPEATEDLY, WHILE STILL ENSURING THE RECOGNITION OF
        ; OUTPUT INTERRUPTS.
NODATA:
        SUB     A
        LD      (OIE),A         ;INDICATE NO OUTPUT INTERRUPT EXPECTED
        LD      A,00000011B     ;DISABLE OUTPUT INTERRUPTS
        OUT     (PIOBC),A

WRDONE:
        POP     AF              ;RESTORE REGISTERS
        EI
        RETI


        ;************************************
        ;ROUTINE: OUTDAT
        ;PURPOSE: SEND CHARACTER TO PIO PORT B
        ;ENTRY: TRNDAT = CHARACTER
        ;EXIT:  NONE
        ;REGISTERS USED: AF
        ;************************************
OUTDAT:
        LD      A,(TRNDAT)      ;GET DATA FROM OUTPUT BUFFER
        OUT     (PIOBD),A       ;SEND DATA TO PIO
        SUB     A               ;INDICATE OUTPUT BUFFER EMPTY
        LD      (TRNDF),A
        DEC     A                   ;INDICATE OUTPUT INTERRUPT EXPECTED
        LD      (OIE),A             ; OIE = FF HEX
        LD      A,10000011B         ;ENABLE OUTPUT INTERRUPTS
        OUT     (PIOBC).A
        RET

        ;************************************
        ;ROUTINE: IPORTS
        ;PURPOSE: INITIALIZE I/O PORTS
        ;ENTRY: HL = BASE ADDRESS OF INITIALIZATION ARRAY
        :EXIT:  DATA OUTPUT TO PORTS
        ;REGISTERS USED: AF,BC,HL
        ;************************************
IPORTS:
        ;GET NUMBER OF DATA BYTES TO SEND TO CURRENT PORT
        ;EXIT IF NUMBER OF BYTES IS 0, INDICATING TERMINATOR
        LD      A,(HL)              ;GET NUMBER OF BYTES
        OR      A                   ;TEST FOR ZERO (TERMINATOR)
        RET     Z                   :RETURN IF NUMBER OF BYTES = 0
        LD      B,A
        INC     HL                  ;POINT TO PORT ADDRESS (NEXT BYTE)

        :C = PORT ADDRESS
        ;HL = BASE ADDRESS OF OUTPUT DATA
        LD      C.(HL)              :GET PORT ADDRESS
        INC     HL                  ;POINT TO FIRST DATA VALUE (NEXT BYTE)

        ;OUTPUT DATA AND CONTINUE TO NEXT PORT
        OTIR                        ;SEND DATA VALUES TO PORT
        JR      IPORTS              :CONTINUE TO NEXT PORT ENTRY

        ;PIO INITIALIZATION DATA
        : PORT A = INPUT
        ; PORT B = OUTPUT
PIOINT:
        DB      3                   ;OUTPUT 3 BYTES
        DB      PIOAC               :DESTINATION IS PORT A CONTROL
        DB      PIOIVA AND OFFH     ;SET INTERRUPT VECTOR FOR PORT A
        DB      10001111B           ;BITS 3,2,1,0 = 1111 (MODE SELECT)
                                    ;BITS 5,4 = 00 (DON'T CARE)
                                    ;BITS 7,6 = 01 (INPUT MODE)
        DB      10000111B           :BITS 3.2.1.0 = 0111 (INTERRUPT CONTROL)
                                    ;BITS 6,5,4 = 000 (DON'T CARE)
                                    ;BITS 7 = 1 (ENABLE INTERRUPTS)

        DB      3                   ;OUTPUT 3 BYTES
        DB      PIOBC               :DESTINATION IS PORT B CONTROL
        DB      PIOIVB AND OFFH     ;SET INTERRUPT VECTOR FOR PORT B
        DB      11001111B           ;BITS 3,2,1,0 = 1111 (MODE SELECT)
                                    ;BITS 5,4 = 00 (DON'T CARE)
                                    ;BITS 7,6 = 00 (CONTROL MODE)
        DB      00000111B           ;BITS 3.2,1,0 = 0111 (INTERRUPT CONTROL)
                                    ;BIT 4,5,6 = 000 (DON'T CARE)
                                    ;BITS 7 = 0 (DISABLE INTERRUPTS)
        DB      0                  ;TERMINATOR FOR INITIALIZATION ARRAY

        ;DATA SECTION
RECDAT: DS      1                  ;RECEIVE DATA
RECDF:  DS      1                  ;RECEIVE DATA FLAG
                                   ; (0 = NO DATA, FF = DATA)

TRNDAT: DS      1                  ;TRANSMIT DATA
TRNDF:  DS      1                  ;TRANSMIT DATA FLAG
                                   ; (0 = BUFFER EMPTY, FF = BUFFER FULL)
OIE:    DS      1                  ;OUTPUT INTERRUPT EXPECTED
                                   ; (0 = NO INTERRUPT EXPECTED,
                                   ; FF = INTERRUPT EXPECTED)


;                                                                        ;
;                                                                        ;
;       SAMPLE EXECUTION:                                                ;
;                                                                        ;
;                                                                        ;

        ;CHARACTER EQUATES
ESCAPE  EQU     1BH                ;ASCII ESCAPE CHARACTER
TESTCH  EQU     'A'                ;TEST CHARACTER = A

SC11B:
        CALL    INIT               ;INITIALIZE PIO, INTERRUPT SYSTEM

        ;SIMPLE EXAMPLE - READ AND ECHO CHARACTERS
        ; UNTIL AN ESC IS RECEIVED
LOOP:
        CALL    INCH               ;READ CHARACTER
        PUSH    AF
        CALL    OUTCH              ;ECHO CHARACTER
        POP     AF
        CP      ESCAPE             ;IS CHARACTER AN ESCAPE?
        JR      NZ,LOOP            ;STAY IN LOOP IF NOT

        ;AN ASYNCHRONOUS EXAMPLE
        ; OUTPUT "A" TO CONSOLE CONTINUOUSLY, BUT ALSO LOOK AT
        ; INPUT SIDE, READING AND ECHOING ANY INPUT CHARACTERS
ASYNLP:
        ;OUTPUT AN "A" IF OUTPUT IS NOT BUSY
        CALL    OUTST              ;IS OUTPUT BUSY?
        JR      C,ASYNLP           ;JUMP IF IT IS
        LD      A,TESTCH
        CALL    OUTCH              ;OUTPUT TEST CHARACTER

        ;CHECK INPUT PORT
        ;ECHO CHARACTER IF ONE IS AVAILABLE
        ;EXIT ON ESCAPE CHARACTER
        CALL    INST               ;IS INPUT DATA AVAILABLE?
        JR      NC,ASYNLP          ;JUMP IF NOT (SEND ANOTHER "A")
        CALL    INCH               ;GET THE CHARACTER
        CP      ESCAPE             ;IS IT AN ESCAPE CHARACTER?
        JR      Z,ASDONE        ;JUMP IF IT IS
        CALL    OUTCH           ;ELSE ECHO CHARACTER
        JP      ASYNLP          ;AND CONTINUE

ASDONE:
        JP      LOOP

        END
```

Buffered Input/Output Using an SIO (SINTB) 11C

Performs interrupt-driven input and output using an SIO and multiple-character buffers. Consists of the following subroutines:

1. INCH reads a character from the input buffer.

2. INST determines whether the input buffer is empty.

3. OUTCH writes a character into the output buffer.

4. OUTST determines whether the output buffer is full.

5. INIT initializes the buffers, the interrupt system, and the SIO.

The actual service routines are

1. RDHDLR responds to the input interrupt by reading a character from the SIO into the input buffer.

2. WRHDLR responds to the output interrupt by writing a character from the output buffer into the SIO.

Registers Used:

1. INCH: AF, C, DE, HL
2. INST: AF
3. OUTCH: AF, DE, HL
4. OUTST: AF
5. INIT: AF, BC, HL, I

Execution Time:

1. INCH: 197 cycles if a character is available
2. INST: 39 cycles
3. OUTCH: 240 cycles if the output buffer is not full and an output interrupt is expected; 160 additional cycles to send the data to the SIO if no output interrupt is expected.
4. OUTST: 34 cycles
5. INIT: 732 cycles
6. RDHDLR: 249 cycles
7. WRHDLR: 308 cycles

Program Size: 299 bytes

Data Memory Required: 11 bytes anywhere in RAM for the heads and tails of the input and output buffers (2 bytes starting at addresses IHEAD, ITAIL, OHEAD, and OTAIL, respectively), the numbers of characters in the buffers (2 bytes at addresses ICNT and OCNT), and the Output Interrupt Expected flag (address OIE). This does not include the actual input and output buffers.

Procedures

1. INCH waits for a character to become available, gets the character from the head of the input buffer, moves the head up one position, and decreases the input buffer counter by 1.

2. INST clears Carry if the input buffer counter is 0 and sets it otherwise.

3. OUTCH waits until there is space in the output buffer (that is, until the output buffer is not full), stores the character at the tail of the buffer, moves the tail up one position, and increases the output buffer counter by 1.

4. OUTST sets Carry if the output buffer counter is equal to the buffer's length (i.e., if the output buffer is full) and clears Carry otherwise.

5. INIT clears the buffer counters, sets both the heads and the tails of the buffers to their base addresses, sets up the interrupt vectors, and initializes the SIO by storing the appropriate values in its control registers. See Subroutine 10E for more details about initializing SIOs. INIT also clears the Output Interrupt Expected flag, indicating that the SIO is initially ready to transmit data.

6. RDHDLR reads a character from the SIO. If there is room in the input buffer, it stores the character at the tail of the buffer, moves the tail up one position, and increases the input buffer counter by 1. If the buffer is full, RDHDLR simply discards the character.

7. WRHDLR determines whether output data is available. If not, it simply resets the output interrupt. If data is available, WRHDLR obtains a character from the head of the output buffer, moves the head up one position, and decreases the output buffer counter by 1.

The new problem with multiple-character buffers is the management of queues. The main program must read the data in the order in which the input interrupt service routine receives it. Similarly, the output interrupt service routine must send the data in the order in which the main program stores it. Thus, there are the following requirements for handling input:

1. The main program must know whether the input buffer is empty.

2. If the input buffer is not empty, the main program must know where the oldest character is (that is, the one that was received first).

3. The input interrupt service routine must know whether the input buffer is full.

4. If the input buffer is not full, the interrupt service routine must know where the next empty place is (that is, where it should store the new character).

The output interrupt service routine and the main program have similar requirements for the output buffer, although the roles of sender and receiver are reversed.

Requirements 1 and 3 are met by maintaining a counter ICNT. INIT initializes ICNT to 0, the interrupt service routine adds 1 to it whenever it receives a character (assuming the buffer is not full), and the main program subtracts 1 from it whenever it removes a character from the buffer. Thus, the main program can determine whether the input buffer is empty by checking if ICNT is 0. Similarly, the interrupt service routine can determine whether the input buffer is full by checking if ICNT is equal to the size of the buffer.

Requirements 2 and 4 are met by maintaining two pointers, IHEAD and ITAIL, defined as follows:

1. ITAIL contains the address of the next empty location in the input buffer.

2. IHEAD contains the address of the oldest character in the input buffer.

INIT initializes IHEAD and ITAIL to the base address of the input buffer. Whenever the interrupt service routine receives a character, it places it in the buffer at ITAIL and moves ITAIL up one position (assuming that the buffer is not full). Whenever the main program reads a character, it removes it from the buffer at IHEAD and moves IHEAD up one position. Thus, IHEAD "chases" ITAIL across the buffer with the service routine entering characters at one end (the tail) while the main program removes them from the other end (the head).

The occupied part of the buffer could thus start and end anywhere. If either IHEAD or ITAIL goes beyond the end of the buffer, the program simply sets it back to the buffer's base address, thus providing wraparound. That is, the occupied part of the buffer could start near the end (say, at byte #195 of a 200-byte buffer) and continue back past the beginning (say, to byte #10). Then IHEAD would be BASE+194, ITAIL would be BASE+9, and the buffer would contain 15 characters occupying addresses BASE+194 through BASE+199 and BASE through BASE+8.

Entry Conditions

1. INCH: none
2. INST: none
3. OUTCH: character to transmit in A
4. OUTST: none
5. INIT: none

Exit Conditions

1. INCH: character in A
2. INST: Carry = 0 if input buffer empty, 1 if otherwise
3. OUTCH: none
4. OUTST: Carry = 0 if output buffer not full, 1 if full
5. INIT: none

```
;                                                              ;
;                                                              ;
;                                                              ;
;                                                              ;
;       Title           Interrupt input and output using a Z80 SIO and ;
;                       multiple-character buffers             ;
;                                                              ;

;                                                              ;
;       Name:           SINTB                                  ;
;                                                              ;
;       Purpose:        This program consists of 5 subroutines which ;
;                       perform interrupt driven input and output using ;
;                       a Z80 SIO.                             ;
;                                                              ;
;                       INCH                                   ;
;                         Read a character                     ;
;                       INST                                   ;
;                         Determine input status (whether input ;
;                         buffer is empty)                     ;
;                       OUTCH                                  ;
;                         Write a character                    ;
;                       OUTST                                  ;
;                         Determine output status (whether output ;
;                         buffer is full)                      ;
;                       INIT                                   ;
;                         Initialize SIO and interrupt system  ;
;                                                              ;
;       Entry:          INCH                                   ;
;                         No parameters                        ;
;                       INST                                   ;
;                         No parameters                        ;
;                       OUTCH                                  ;
;                         Register A = character to transmit   ;
;                       OUTST                                  ;
;                         No parameters                        ;
;                       INIT                                   ;
;                         No parameters                        ;
;                                                                       ;
;       Exit:           INCH                                            ;
;                         Register A = character                        ;
;                       INST                                            ;
;                         Carry = 0 if input buffer is empty,           ;
;                         1 if character is available                   ;
;                       OUTCH                                           ;
;                         No parameters                                 ;
;                       OUTST                                           ;
;                         Carry = 0 if output buffer is not             ;
;                         full, 1 if it is full                         ;
;                       INIT                                            ;
;                         No parameters                                 ;
;                                                                       ;
;       Registers used: INCH                                            ;
;                         AF,C,DE,HL                                    ;
;                       INST                                            ;
;                         AF                                            ;
;                       OUTCH                                           ;
;                         AF,DE,HL                                      ;
;                       OUTST                                           ;
;                         AF                                            ;
;                       INIT                                            ;
;                         AF,BC,HL,I                                    ;
;                                                                       ;
;       Time:           INCH                                            ;
;                         Approximately 197 cycles if a character is    ;
;                         available                                     ;
;                       INST                                            ;
;                         39 cycles                                     ;
;                       OUTCH                                           ;
;                         Approximately 240 cycles if output buffer     ;
;                         is not full and output interrupt is expected  ;
;                       OUTST                                           ;
;                         34 cycles                                     ;
;                       INIT                                            ;
;                         732 cycles                                    ;
;                       RDHDLR                                          ;
;                         Approximately 249 cycles                      ;
;                       WRHDLR                                          ;
;                         Approximately 308 cycles                      ;
;                                                                       ;
;       Size:           Program 299 bytes                               ;
;                       Data     11 bytes plus size of buffers          ;
;                                                                       ;
;                                                                       ;

        ;SIO EQUATES
        ; SIO IS PROGRAMMED FOR:
        ;  ASYNCHRONOUS OPERATION
        ;  16 X BAUD RATE
        ;  8-BIT CHARACTERS
        ;  1 1/2 STOP BITS
        ;ARBITRARY SIO PORT ADDRESSES
SIOCAD  EQU     1CH             ;SIO CHANNEL A DATA
```

```
SIOCBD  EQU     1EH             ;SIO CHANNEL B DATA
SIOCAS  EQU     1DH             ;SIO CHANNEL A COMMANDS/STATUS
SIOCBS  EQU     1FH             ;SIO CHANNEL B COMMANDS/STATUS
SIOIV   EQU     8000H           ;INTERRUPT VECTOR
SIOWV   EQU     SIOIV+8         ;SIO CHANNEL A WRITE INTERRUPT VECTOR
SIOEV   EQU     SIOIV+10        ;SIO CHANNEL A EXTERNAL/STATUS
                                ; INTERRUPT VECTOR
SIORV   EQU     SIOIV+12        ;SIO CHANNEL A READ INTERRUPT VECTOR
SIOSV   EQU     SIOIV+14        ;SIO CHANNEL A SPECIAL RECEIVE
                                ; INTERRUPT VECTOR

        ;READ CHARACTER
INCH:
        CALL    INST            ;GET INPUT STATUS
        JR      NC,INCH         ;WAIT IF NO CHARACTER AVAILABLE
        DI                      ;DISABLE INTERRUPTS
        LD      HL,ICNT         ;REDUCE INPUT BUFFER COUNT BY 1
        DEC     (HL)
        LD      HL,(IHEAD)      ;GET CHARACTER FROM HEAD OF INPUT BUFFER
        LD      C,(HL)
        CALL    INCIPTR         ;MOVE HEAD POINTER UP 1
        LD      (IHEAD),HL
        LD      A,C
        EI                      ;REENABLE INTERRUPTS
        RET

        ;RETURN INPUT STATUS (CARRY = 1 IF INPUT DATA IS AVAILABLE)
INST:
        LD      A,(ICNT)        ;TEST INPUT BUFFER COUNT
        OR      A               ;CLEAR CARRY ALWAYS
        RET     Z               ;RETURN, CARRY = 0 IF NO DATA
        SCF                     ;SET CARRY
        RET                     ;RETURN, CARRY = 1 IF DATA AVAILABLE

        ;WRITE CHARACTER
OUTCH:
        PUSH    AF              ;SAVE CHARACTER TO OUTPUT

        ;WAIT FOR OUTPUT BUFFER NOT FULL,  THEN STORE NEXT CHARACTER
WAITOC:
        CALL    OUTST           ;GET OUTPUT STATUS
        JR      C,WAITOC        ;WAIT IF OUTPUT BUFFER IS FULL
        DI                      ;DISABLE INTERRUPTS WHILE LOOKING AT
                                ; BUFFER, INTERRUPT STATUS
        LD      HL,OCNT         ;INCREASE OUTPUT BUFFER COUNT BY 1
        INC     (HL)
        LD      HL,(OTAIL)      ;POINT TO NEXT EMPTY BYTE IN BUFFER
        POP     AF              ;GET CHARACTER
        LD      (HL),A          ;STORE CHARACTER AT TAIL OF BUFFER
        CALL    INCOPTR         ;MOVE TAIL POINTER UP 1
        LD      (OTAIL),HL
        LD      A,(OIE)         ;TEST OUTPUT INTERRUPT EXPECTED FLAG
        OR      A
        CALL    Z,OUTDAT        ;OUTPUT CHARACTER IMMEDIATELY IF
                                ; OUTPUT INTERRUPT NOT EXPECTED
        EI                              ;REENABLE INTERRUPTS
        RET

        ;OUTPUT STATUS (CARRY = 1 IF BUFFER IS FULL)
OUTST:
        LD      A,(OCNT)                ;GET CURRENT OUTPUT BUFFER COUNT
        CP      SZOBUF                  ;COMPARE TO MAXIMUM
        CCF                             ;COMPLEMENT CARRY
        RET                             ;CARRY = 1 IF BUFFER FULL, 0 IF NOT

        ;INITIALIZE SIO, INTERRUPT SYSTEM
INIT:
        DI                              ;DISABLE INTERRUPTS

        ;INITIALIZE BUFFER COUNTERS AND POINTERS, INTERRUPT FLAG
        SUB     A
        LD      (OIE),A                 ;INDICATE NO OUTPUT INTERRUPTS
        LD      (ICNT),A                ;BUFFER COUNTERS = 0
        LD      (OCNT),A
        LD      HL,IBUF                 ;ALL BUFFER POINTERS = BASE ADDRESS
        LD      (IHEAD),HL
        LD      (ITAIL),HL
        LD      HL,OBUF
        LD      (OHEAD),HL
        LD      (OTAIL),HL

        ;INITIALIZE INTERRUPT VECTORS
        LD      A,SIOIV SHR 8           ;GET HIGH BYTE OF INTERRUPT PAGE
        LD      I,A                     ;SET INTERRUPT VECTOR IN Z80
        IM      2                       ;INTERRUPT MODE 2 - VECTORS IN TABLE
        LD      HL,RDHDLR               ; ON INTERRUPT PAGE
        LD      (SIORV),HL              ;STORE READ VECTOR
        LD      HL,WRHDLR
        LD      (SIOWV),HL              ;STORE WRITE VECTOR
        LD      HL,EXHDLR
        LD      (SIOEV),HL              ;STORE EXTERNAL/STATUS VECTOR
        LD      HL,REHDLR
        LD      (SIOSV),HL              ;STORE SPECIAL RECEIVE VECTOR

        ;INITIALIZE I/O PORTS
        LD      HL,SIOINT               ;BASE ADDRESS OF INITIALIZATION ARRAY
        CALL    IPORTS                  ;INITIALIZE SIO
        EI                              ; ENABLE INTERRUPTS
        RET

        ;INPUT (READ) INTERRUPT HANDLER
RDHDLR:
        PUSH    AF                      ;SAVE REGISTERS
        PUSH    BC
        PUSH    DE
        PUSH    HL
RD1:
        IN      A,(SIOCAD)              ;READ DATA FROM SIO
        LD      C,A                     ;SAVE DATA IN REGISTER C
        LD      HL,ICNT                 ;ANY ROOM IN INPUT BUFFER?
        LD      A,(HL)
        CP      SZIBUF
        JR      NC,XITRH        ;JUMP IF NO ROOM
        INC     (HL)            ;INCREMENT INPUT BUFFER COUNTER
        LD      HL,(ITAIL)      ;STORE CHARACTER AT TAIL OF INPUT BUFFER
        LD      (HL),C
        CALL    INCIPTR         ;INCREMENT TAIL POINTER
        LD      (ITAIL),HL

XITRH:
        POP     HL              ;RESTORE REGISTERS
        POP     DE
        POP     BC
        POP     AF
        EI                      ;REENABLE INTERRUPTS
        RETI

        ;OUTPUT (WRITE) INTERRUPT HANDLER
WRHDLR:
        PUSH    AF              ;SAVE REGISTERS
        PUSH    BC
        PUSH    DE
        PUSH    HL

        LD      A,(OCNT)        ;GET OUTPUT BUFFER COUNTER
        OR      A               ;TEST FOR EMPTY BUFFER
        JR      Z,NODATA        ;JUMP IF NO DATA TO TRANSMIT
        CALL    OUTDAT          ;ELSE OUTPUT DATA
        JR      WRDONE

        ;IF AN OUTPUT INTERRUPT OCCURS WHEN NO DATA IS AVAILABLE,
        ; WE MUST DISABLE OUTPUT INTERRUPTS TO AVOID AN ENDLESS LOOP.
        ; WHEN THE NEXT CHARACTER IS READY, IT MUST BE SENT IMMEDIATELY
        ; SINCE NO INTERRUPT WILL OCCUR.  THIS STATE IN WHICH AN OUTPUT
        ; INTERRUPT HAS OCCURRED BUT HAS NOT BEEN SERVICED IS INDICATED
        ; BY CLEARING OIE (OUTPUT INTERRUPT EXPECTED FLAG).
NODATA:
        SUB     A
        LD      (OIE),A         ;DO NOT EXPECT AN INTERRUPT
        OUT     (SIOCAS),A      ;SELECT REGISTER 0
        LD      A,00101000B     ;RESET TRANSMITTER INTERRUPT
        OUT     (SIOCAS),A

WRDONE:
        POP     HL              ;RESTORE REGISTERS
        POP     DE
        POP     BC
        POP     AF
        EI
        RETI

        ;EXTERNAL/STATUS CHANGED INTERRUPT HANDLER
EXHDLR:
        PUSH    AF
        LD      A,00010000B     ;RESET STATUS INTERRUPT
        OUT     (SIOCAS),A
        POP     AF
        EI                      ;DCD OR CTS LINE CHANGED STATE, OR A
        RETI                    ; BREAK WAS DETECTED
                                ;SERVICE HERE IF NECESSARY

        ;SPECIAL RECEIVE ERROR INTERRUPT
REHDLR:
        PUSH    AF
        LD      A,00110000B     ;RESET RECEIVE ERROR INTERRUPT
        OUT     (SIOCAS),A
        POP     AF
        EI                      ;FRAMING ERROR OR OVERRUN ERROR OCCURRED
        RETI                    ; SERVICE HERE IF NECESSARY


        ;************************************
        ;ROUTINE: OUTDAT
        ;PURPOSE: SEND CHARACTER TO SIO
        ;ENTRY: NONE
        ;EXIT:  NONE
        ;REGISTERS USED: AF,DE,HL
        ;************************************
OUTDAT:
        LD      HL,(OHEAD)
        LD      A,(HL)          ;GET DATA FROM HEAD OF OUTPUT BUFFER
        OUT     (SIOCAD),A      ;OUTPUT DATA
        CALL    INCOPTR         ;INCREMENT HEAD POINTER
        LD      (OHEAD),HL
        LD      HL,OCNT         ;DECREMENT OUTPUT BUFFER COUNT
        DEC     (HL)
        LD      A,0FFH
        LD      (OIE),A         ;EXPECT AN OUTPUT INTERRUPT
        RET

        ;************************************
        ;ROUTINE: INCIPTR
        ;PURPOSE: INCREMENT POINTER INTO INPUT
        ;          BUFFER WITH WRAPAROUND
        ;ENTRY: HL = POINTER
        ;EXIT:  HL = POINTER INCREMENTED WITH WRAPAROUND
        ;REGISTERS USED: AF,DE,HL
        ;************************************
INCIPTR:
        INC     HL              ;INCREMENT POINTER
        LD      DE,EIBUF        ;COMPARE POINTER, END OF BUFFER
        LD      A,L
        CP      E
        RET     NZ
        LD      A,H
        CP      D
        RET     NZ              ;RETURN IF NOT EQUAL
        LD      HL,IBUF         ;IF POINTER AT END OF BUFFER,
        RET                     ; SET IT BACK TO BASE ADDRESS
        ;*************************************
        ;ROUTINE: INCOPTR
        ;PURPOSE: INCREMENT POINTER INTO OUTPUT
        ;         BUFFER WITH WRAPAROUND
        ;ENTRY: HL = POINTER
        ;EXIT:  HL = POINTER INCREMENTED WITH WRAPAROUND
        ;REGISTERS USED: AF,DE,HL
        ;*************************************
INCOPTR:
        INC     HL              ;INCREMENT POINTER
        LD      DE,EOBUF        ;COMPARE POINTER, END OF BUFFER
        LD      A.L
        CP      E
        RET     NZ
        LD      A.H
        CP      D
        RET     NZ
        LD      HL,OBUF         ;IF POINTER AT END OF BUFFER,
        RET                     ; SET IT BACK TO BASE ADDRESS

        ;*************************************
        ;ROUTINE: IPORTS
        ;PURPOSE: INITIALIZE I/O PORTS
        ;ENTRY: HL = BASE ADDRESS OF INITIALIZATION ARRAY
        ;EXIT:  DATA OUTPUT TO PORTS
        ;REGISTERS USED: AF.BC.HL
        ;*************************************
IPORTS:
        ;GET NUMBER OF DATA BYTES TO SEND TO CURRENT PORT
        ;EXIT IF NUMBER OF BYTES IS 0, INDICATING TERMINATOR
        LD      A,(HL)          ;GET NUMBER OF BYTES
        OR      A               ;TEST FOR ZERO (TERMINATOR)
        RET     Z               ;RETURN IF NUMBER OF BYTES = 0
        LD      B,A
        INC     HL              ;POINT TO PORT ADDRESS (NEXT BYTE)

        ;C = PORT ADDRESS
        ;HL = BASE ADDRESS OF OUTPUT DATA
        LD      C,(HL)          ;GET PORT ADDRESS
        INC     HL              ;POINT TO FIRST DATA VALUE (NEXT BYTE)

        ;OUTPUT DATA AND CONTINUE TO NEXT PORT
        OTIR                    ;SEND DATA VALUES TO PORT
        JR      IPORTS          ;CONTINUE TO NEXT PORT ENTRY

        ;SIO INITIALIZATION DATA
SIOINT:
        ;RESET CHANNEL A
        DB      1               ;OUTPUT 1 BYTE
        DB      SIOCAS          ;TO CHANNEL A COMMAND/STATUS
        DB      00011000B       ;SELECT WRITE REGISTER 0
                                ;BITS 2,1,0 = 0 (WRITE REGISTER 0)
                                ;BITS 5,4,3 = 011 (CHANNEL RESET)
                                ;BITS 7,6 = 0 (DO NOT CARE)
        ;SET INTERRUPT VECTOR AND ALLOW STATUS TO AFFECT IT
        DB      4                   ;OUTPUT 2 BYTES
        DB      SIOCBS              ;DESTINATION IS COMMAND REGISTER B
        DB      00000010B           ;SELECT WRITE REGISTER 2
        DB      SIOIV AND OFFH      ;SET INTERRUPT VECTOR FOR SIO
        DB      00000001B           ;SELECT WRITE REGISTER 1
        DB      00000100B           ;TURN ON STATUS AFFECTS VECTOR

        ;INITIALIZE CHANNEL A
        DB      8                   ;OUTPUT 8 BYTES
        DB      SIOCAS              ;DESTINATION IS COMMAND REGISTER A

        ;INITIALIZE BAUD RATE CONTROL
        DB      00010100B           ;SELECT WRITE REGISTER 4
                                    ; RESET EXTERNAL/STATUS INTERRUPT
        DB      01001000B           ;BIT 0 = 0 (NO PARITY)
                                    ;BIT 1 = 0 (DON'T CARE)
                                    ;BITS 3,2 = 10 (1 1/2 STOP BITS)
                                    ;BITS 5,4 = 00 (DON'T CARE)
                                    ;BITS 7,6 = 01 (16 TIMES CLOCK)

        ;INITIALIZE RECEIVE CONTROL
        DB      00000011B           ;SELECT WRITE REGISTER 3
        DB      11000001B           ;BIT 0 = 1 (RECEIVE ENABLE)
                                    ;BITS 4,3,2,1 = 0 (DON'T CARE)
                                    ;BIT 5 = 0 (NO AUTO ENABLE)
                                    ;BIT 7,6 = 11 (RECEIVE 8 BITS/CHAR)

        ;INITIALIZE TRANSMIT CONTROL
        DB      00000101B           ;SELECT WRITE REGISTER 5
        DB      11101010B           ;BIT 0 = 0 (NO CRC ON TRANSMIT)
                                    ;BIT 1 = 1 (REQUEST TO SEND)
                                    ;BIT 2 = 0 (DON'T CARE)
                                    ;BIT 3 = 1 (TRANSMIT ENABLE)
                                    ;BIT 4 = 0 (DO NOT SEND BREAK)
                                    ;BITS 6,5 = 11 (TRANSMIT 8 BITS/CHAR)
                                    ;BIT 7 = 1 (DATA TERMINAL READY)
        DB      00000001B           ;SELECT WRITE REGISTER 1
        DB      00011011B           ;BIT 0 = 1 (EXTERNAL INTERRUPTS)
                                    ;BIT 1 = 1 (ENABLE TRANSMIT INTERRUPT)
                                    ;BIT 2 = 0 (DO NOT CARE)
                                    ;BITS 4,3 = 11 (RECEIVE INTERRUPTS ON
                                    ; ALL CHARS, PARITY DOES NOT AFFECT
                                    ; VECTOR)
                                    ;BITS 7,6,5 = 000 (NO WAIT/READY
                                    ; FUNCTION)
        DB      0                   ;END OF TABLE

        ;DATA SECTION
IHEAD:  DS      2                   ;ADDRESS OF OLDEST CHARACTER IN INPUT
                                    ; BUFFER
ITAIL:  DS      2                   ;ADDRESS OF NEWEST CHARACTER IN INPUT
                                    ; BUFFER
ICNT:   DS      1                   ;NUMBER OF CHARACTERS IN INPUT BUFFER
OHEAD:  DS      2                   ;ADDRESS OF OLDEST CHARACTER IN OUTPUT
                                    ; BUFFER
```

```
OTAIL:  DS      2               ;ADDRESS OF NEWEST CHARACTER IN OUTPUT
                                ; BUFFER
OCNT:   DS      1               ;NUMBER OF CHARACTERS IN OUTPUT BUFFER
OIE:    DS      1               ;OUTPUT INTERRUPT EXPECTED
                                ; (0 = NO INTERRUPT EXPECTED,
                                ; FF = INTERRUPT EXPECTED)
SZIBUF  EQU     1               ;SIZE OF INPUT BUFFER
IBUF:   DS      SZIBUF          ;INPUT BUFFER
EIBUF   EQU     $               ;END OF INPUT BUFFER
SZOBUF  EQU     255             ;SIZE OF OUTPUT BUFFER
OBUF:   DS      SZOBUF          ;OUTPUT BUFFER
EOBUF   EQU     $               ;END OF OUTPUT BUFFER


;                                                                       ;
;                                                                       ;
;       SAMPLE EXECUTION:                                               ;
;                                                                       ;
;                                                                       ;


        ;CHARACTER EQUATES
ESCAPE  EQU     1BH             ;ASCII ESCAPE CHARACTER
TESTCH  EQU     'A'             ;TEST CHARACTER = A

SC11C:
        CALL    INIT            ;INITIALIZE SIO, INTERRUPT SYSTEM

        ;SIMPLE EXAMPLE - READ AND ECHO CHARACTER
        ; UNTIL AN ESC IS RECEIVED
LOOP:
        CALL    INCH            ;READ CHARACTER
        PUSH    AF
        CALL    OUTCH           ;ECHO CHARACTER
        POP     AF
        CP      ESCAPE          ;IS CHARACTER AN ESCAPE?
        JR      NZ,LOOP         ;STAY IN LOOP IF NOT

        ;AN ASYNCHRONOUS EXAMPLE
        ; OUTPUT "A" TO CONSOLE CONTINUOUSLY BUT ALSO LOOK AT
        ; INPUT SIDE, READING AND ECHOING ANY INPUT CHARACTERS
ASYNLP:
        ;OUTPUT AN "A" IF OUTPUT IS NOT BUSY
        CALL    OUTST           ;IS OUTPUT BUSY?
        JR      C,ASYNLP        ;JUMP IF IT IS
        LD      A.TESTCH
        CALL    OUTCH           ;OUTPUT CHARACTER

        ;CHECK INPUT PORT
        ;ECHO CHARACTER IF ONE IS AVAILABLE
        ;EXIT ON ESCAPE CHARACTER
        CALL    INST            ;IS INPUT DATA AVAILABLE?
        JR      NC,ASYNLP       ;JUMP IF NOT (SEND ANOTHER "A")
        CALL    INCH            ;GET CHARACTER
        CP      ESCAPE          ;IS IT AN ESCAPE CHARACTER?
        JR      Z,DONE          ;BRANCH IF IT IS
        CALL    OUTCH           ;ELSE ECHO CHARACTER
        JP      ASYNLP          ;AND CONTINUE

DONE:
        JP      LOOP

        END
```

Real-Time Clock and Calendar (CLOCK) 11D

Maintains a time-of-day 24-hour clock and a calendar based on a real-time clock interrupt generated from a Z80 CTC. Consists of the following subroutines:

1. CLOCK returns the base address of the clock variables.

2. ICLK initializes the clock interrupt and the clock variables.

3. CLKINT updates the clock after each interrupt (assumed to be spaced one tick apart).

Registers Used:

1. CLOCK: HL
2. ICLK: AF,HL,I
3. CLKINT: None

Execution Time:

1. CLOCK: 20 cycles
2. ICLK: 251 cycles
3. CLKINT: 93 cycles if only TICK must be decremented; 498 cycles maximum if changing to a new year.

Program Size: 171 bytes

Data Memory Required: 8 bytes for the clock variables starting at address CLKVAR

Procedures

1. CLOCK loads the base address of the clock variables into register pair HL. The clock variables are stored in the following order (lowest address first): ticks, seconds, minutes, hours, days, months, less significant byte of year, more significant byte of year.

2. ICLK initializes the CTC, the interrupt system, and the clock variables. The arbitrary starting time is 00:00:00, January 1, 1980. A real application would clearly require some kind of outside intervention to load or change the clock.

3. CLKINT decrements the remaining tick count by 1 and updates the rest of the clock if necessary. Of course, the number of seconds and minutes must be less than 60 and the number of hours must be less than 24. The day of the month must be less than or equal to the last day for the current month; an array of the last days of each month begins at address LASTDY. If the month is February (#2), the program checks if the current year is a leap year. This involves determining whether the two least significant bits of memory location YEAR are both 0s. If the current year is a leap year, the last day of February is the 29th, not the 28th. The month number may not exceed 12 (December) or a carry to the year number is necessary. The program must reinitialize the variables properly when carries occur; that is, TICK to DTICK; seconds, minutes, and hours to 0; day and month to 1 (meaning the first day and January, respectively).

Entry Conditions

1. CLOCK: none
2. ICLK: none
3. CLKINT: none

Exit Conditions

1. CLOCK: base address of clock variables in HL
2. ICLK: none
3. CLKINT: none

Examples

These examples assume that the tick rate is DTICK Hz (less than 256 Hz — typical values would be 60 Hz or 100 Hz) and that the clock and calendar are saved in memory locations:

TICK	ticks before a carry, counted down from DTICK
SEC	seconds (0 to 59)
MIN	minutes (0 to 59)
HOUR	hour of day (0 to 23)
DAY	day of month (1 to 28, 29, 30, or 31)
MONTH	month of year (1 through 12)
YEAR and YEAR+1	current year

1. Data:
   March 7, 1982, 11:59.59 P.M. and 1 tick left
   (TICK) = 1 (SEC) = 59 (DAY) = 07
   (MIN) = 59 (MONTH) = 03
   (HOUR) = 23 (YEAR) = 1982

   Result:
   March 8, 1982, 12:00.00 A.M. and DTICK ticks
   (TICK) = DTICK (SEC) = 0 (DAY) = 08
   (MIN) = 0 (MONTH) = 03
   (HOUR) = 0 (YEAR) = 1982

2. Data:
   Dec. 31, 1982, 11:59.59 P.M. and 1 tick left
   (TICK) = 1 (SEC) = 59 (DAY) = 31
   (MIN) = 59 (MONTH) = 12
   (HOUR) = 23 (YEAR) = 1982

   Result:
   Jan. 1, 1983, 12:00.00 A.M./and DTICK ticks
   (TICK) = DTICK (SEC) = 0 (DAY) = 1
   (MIN) = 0 (MONTH) = 1
   (HOUR) = 0 (YEAR) = 1

```
;                                                                   ;
;                                                                   ;
;                                                                   ;
;                                                                   ;
;       Title           Real-time clock and calendar                ;
;       Name:           CLOCK                                       ;
;                                                                   ;
;                                                                   ;


;                                                                   ;
;       Purpose:        This program maintains a time-of-day 24-hour ;
;                       clock and a calendar based on a real-time clock ;
;                       interrupt from a Z80 CTC.                   ;
;                                                                   ;
;                       CLOCK                                       ;
;                         Returns base address of clock variables   ;
;                       ICLK                                        ;
;                         Initializes CTC and clock interrupt       ;
;                       CLKINT                                      ;
;                         Updates clock variables for each tick     ;
;                                                                   ;
;       Entry:          CLOCK                                       ;
;                         None                                      ;
;                       ICLK                                        ;
;                         None                                      ;
;                                                                   ;
;       Exit:           CLOCK                                       ;
;                         Register HL = Base address of time variables  ;
;                       ICLK                                            ;
;                         None                                          ;
;                                                                       ;
;       Registers used: CLOCK                                           ;
;                         HL                                            ;
;                       ICLK                                            ;
;                         AF,HL,I                                       ;
;                       CLKINT                                          ;
;                         None                                          ;
;                                                                       ;
;       Time:           CLOCK                                           ;
;                         20 cycles                                     ;
;                       ICLK                                            ;
;                         251 cycles                                    ;
;                       CLKINT                                          ;
;                         93 cycles normally if decrementing tick       ;
;                        498 cycles maximum if changing to a new year   ;
;                                                                       ;
;       Size:           Program 171 bytes                               ;
;                       Data      8 bytes                               ;
;                                                                       ;
;                                                                       ;

        ;ARBITRARY PORT ADDRESSES FOR Z80 CTC
CTCCH0  EQU     80H             ;CTC CHANNEL 0 PORT
CTCITRP EQU     08000H          ;CTC INTERRUPT ADDRESS
CTCCMD  EQU     10100111B       ;BIT 7 = 1 INTERRUPTS ENABLED
                                ;BIT 6 = 0 TIMER MODE
                                ;BIT 5 = 1 256 COUNT PRESCALER
                                ;BIT 4 = 0 NEGATIVE EDGE TRIGGER
                                ;BIT 3 = 0 START TIMER AFTER TIME CONST
                                ;BIT 2 = 1 TIME CONSTANT FOLLOWS
                                ;BIT 1 = 1 RESET CHANNEL
                                ;BIT 0 = 1 CONTROL WORD
CTCTC   EQU     250             ;TIME CONSTANT

        ;CALCULATION FOR TICK
        ; ASSUME A 4 MHZ CLOCK FOR CTC WITH PRESCALER = 256
        ;   AND COUNT = 250 = (4 * 10^6) / (256 * 250)
        ;   IS ABOUT 62 TICKS PER SECOND
DFLTS:
DTICK   EQU     62              ;DEFAULT TICK

        ;RETURN BASE ADDRESS OF CLOCK VARIABLES
CLOCK:
        LD      HL,CLKVAR       ;GET BASE ADDRESS OF CLOCK VARIABLES
        RET

        ;INITIALIZE CTC CHANNEL 0 AS A REAL-TIME CLOCK INTERRUPT
ICLK:
        DI                      ;DISABLE INTERRUPTS
        LD      A,CTCITRP SHR 8
        LD      I,A             ;SET UP INTERRUPT VECTOR
        IM      2               ;SET INTERRUPT MODE 2 - VECTORS IN
                                ; TABLE ON INTERRUPT PAGE
        LD      HL,CLKINT
        LD      (CTCITRP),HL    ;SET INTERRUPT ADDRESS
        LD      A,1
        OUT     (CTCCH0),A      ;DISABLE CHANNEL 0
        LD      A,CTCITRP AND 0FFH      ;LOW BYTE OF CTC INTERRUPT
        OUT     (CTCCH0),A      ;VECTOR TO CTC
        LD      A,CTCCMD
        OUT     (CTCCH0),A      ;OUTPUT CTC COMMAND
        LD      A,CTCTC
        OUT     (CTCCH0),A      ;OUTPUT TIME CONSTANT

        ;INITIALIZE CLOCK VARIABLES TO ARBITRARY VALUE
        ;JANUARY 1, 1980 00:00.00
        ;A REAL CLOCK WOULD NEED OUTSIDE INTERVENTION
        ; TO SET OR CHANGE VALUES
        LD      HL,TICK
        LD      (HL),DTICK      ;INITIALIZE TICKS
        INC     HL
        SUB     A
        LD      (HL),A          ;SECOND = 0
        INC     HL
        LD      (HL),A          ;MINUTE = 0
        INC     HL
        LD      (HL),A          ;HOUR = 0
        INC     A               ;A = 1
        INC     HL
        LD      (HL),A          ;DAY = 1 (FIRST)
        INC     HL
        LD      (HL),A          ;MONTH = 1 (JANUARY)
        LD      HL,1980
        LD      (YEAR),HL       ;YEAR = 1980

        EI
        RET

        ;HANDLE CLOCK INTERRUPT
CLKINT:
        PUSH    AF              ;SAVE AF,HL
        PUSH    HL
        LD      HL,TICK
        DEC     (HL)            ;DECREMENT TICK COUNT
        JR      NZ,EXIT1        ;JUMP IF TICK NOT ZERO
        LD      (HL),DTICK      ;SET TICK COUNT BACK TO DEFAULT

        PUSH    BC              ;SAVE BC,DE
        PUSH    DE

        LD      B,0             ;0 = DEFAULT FOR SECONDS, MINUTES, HOURS

        ;INCREMENT SECONDS
        INC     HL              ;POINT AT SECONDS
        INC     (HL)            ;INCREMENT TO NEXT SECOND
        LD      A,(HL)
        CP      60              ;SECONDS = 60?
        JR      C,EXIT0         ;EXIT IF LESS THAN 60 SECONDS
        LD      (HL),B          ;ELSE SECONDS = 0

        ;INCREMENT MINUTES
        INC     HL              ;POINT AT MINUTES
        INC     (HL)            ;INCREMENT TO NEXT MINUTE
        LD      A,(HL)
        CP      60              ;MINUTES = 60?
        JR      C,EXITO         ;EXIT IF LESS THAN 60 MINUTES
        LD      (HL),B          ;ELSE MINUTES = 0

        ;INCREMENT HOUR
        INC     HL              ;POINT AT HOUR
        INC     (HL)            ;INCREMENT TO NEXT HOUR
        LD      A,(HL)
        CP      24              ;HOURS = 24?
        JR      C,EXITO         ;EXIT IF LESS THAN 24 HOURS
        LD      (HL),B          ;ELSE HOUR = 0

        ;INCREMENT DAY
        EX      DE,HL           ;SAVE ADDRESS OF HOUR
        LD      HL,LASTDY-1
        LD      A,(MONTH)       ;GET CURRENT MONTH
        LD      C,A             ;REGISTER C = MONTH
        LD      B,0
        ADD     HL,BC           ;POINT AT LAST DAY OF MONTH
        EX      DE,HL           ;RESTORE ADDRESS OF HOUR
        INC     HL              ;POINT AT DAY
        LD      A,(HL)          ;GET CURRENT DAY
        INC     (HL)            ;INCREMENT TO NEXT DAY
        EX      DE,HL           ;DE = ADDRESS OF DAY
        LD      B,A             ;REGISTER B = DAY
        CP      (HL)            ;IS CURRENT DAY END OF MONTH?
        EX      DE,HL           ;HL = ADDRESS OF DAY
        JR      C,EXITO         ;EXIT IF NOT AT END OF MONTH

        ;DETERMINE IF THIS IS END OF FEBRUARY IN A LEAP
        ; YEAR (YEAR DIVISIBLE BY 4)
        LD      A,C             ;GET MONTH
        CP      2               ;IS THIS FEBRUARY?
        JR      NZ,INCMTH       ;JUMP IF NOT, INCREMENT MONTH
        LD      A,(YEAR)        ;IS IT A LEAP YEAR?
        AND     00000011B
        JR      NZ,INCMTH       ;JUMP IF NOT

        ;FEBRUARY OF A LEAP YEAR HAS 29 DAYS, NOT 28 DAYS
        LD      A,B             ;GET DAY
        CP      29
        JR      C,EXITO         ;EXIT IF NOT 1ST OF MARCH

INCMTH:
        LD      B,1             ;DEFAULT IS 1 FOR DAY AND MONTH
        LD      (HL),B          ;DAY = 1

        INC     HL
        INC     (HL)            ;INCREMENT MONTH
        LD      A,C             ;GET OLD MONTH
        CP      12              ;WAS OLD MONTH DECEMBER?
        JR      NC,EXITO        ;EXIT IF NOT
        LD      (HL),B          ;ELSE
                                ; CHANGE MONTH TO 1 (JANUARY)

        ;INCREMENT YEAR
        LD      HL,(YEAR)
        INC     HL
        LD      (YEAR).HL

EXITO:
        ;RESTORE REGISTERS
        POP     DE              ;RESTORE BC.DE
        POP     BC
EXIT1:
        POP     HL              ;RESTORE HL.AF
        POP     AF
        EI                      ;REENABLE INTERRUPTS
        RETI                    ;RETURN

        ;ARRAY OF LAST DAYS OF EACH MONTH
LASTDY:
        DB      31              ;JANUARY
        DB      28              ;FEBRUARY (EXCEPT LEAP YEARS)
        DB      31              ;MARCH
        DB      30              ;APRIL
        DB      31              ;MAY
        DB      30              ;JUNE
        DB      31              ;JULY
        DB      31              ;AUGUST
        DB      30              ;SEPTEMBER
        DB      31              ;OCTOBER
        DB      30              ;NOVEMBER
        DB      31              ;DECEMBER

        ;CLOCK VARIABLES
CLKVAR:
TICK:   DS      1               ;TICKS LEFT IN CURRENT SECOND
SEC:    DS      1               ;SECONDS (0 TO 59)
MIN:    DS      1               ;MINUTES (0 TO 59)
HOUR:   DS      1               ;HOURS (0 TO 23)
DAY:    DS      1               ;DAY (1 TO NUMBER OF DAYS IN A MONTH)
MONTH:  DS      1               ;MONTH 1=JANUARY .. 12=DECEMBER
YEAR:   DS      2               ;YEAR


;                                                                       ;
;                                                                       ;
;       SAMPLE EXECUTION                                                ;
;                                                                       ;
;                                                                       ;

        ;CLOCK VARIABLE INDEXES
TCKIDX  EQU     0               ;INDEX TO TICK
```

```
SECIDX  EQU     1               ;INDEX TO SECOND
MINIDX  EQU     2               ;INDEX TO MINUTE
HRIDX   EQU     3               ;INDEX TO HOUR
DAYIDX  EQU     4               ;INDEX TO DAY
MTHIDX  EQU     5               ;INDEX TO MONTH
YRIDX   EQU     6               ;INDEX TO YEAR

SC11D:
        CALL    ICLK            ;INITIALIZE CLOCK

        ;INITIALIZE CLOCK TO 2/7/83 14:00:00 (2 PM. FEB. 7, 1983)
        CALL    CLOCK           ;HL = BASE ADDRESS OF CLOCK VARIABLES
        DI
        PUSH    HL
        POP     IX              ;IX = ADDRESS OF TICKS
        LD      (IX+SECIDX),0   ;SECONDS = 0
        LD      (IX+MINIDX).0   ;MINUTES = 0
        LD      (IX+HRIDX),14   ;HOUR = 14 (2 PM)
        LD      (IX+DAYIDX).7   ;DAY = 7
        LD      (IX+MTHIDX),2   ;MONTH = 2 (FEBRUARY)
        LD      HL.1983
        LD      (IX+YRIDX),L    ;YEAR = 1983
        LD      (IX+YRIDX+1).H
        EI

        ;WAIT FOR CLOCK TO BE 2/7/83 14:01:20 (2:01.20 PM, FEB.7, 1983)
        ;IX = BASE ADDRESS OF CLOCK VARIABLES

        ;NOTE:  MUST BE CAREFUL TO EXIT IF CLOCK IS ACCIDENTALLY
        ; SET AHEAD.  IF WE CHECK ONLY FOR EQUALITY, WE MIGHT NEVER
        ; FIND IT.  THUS WE HAVE >= IN TESTS BELOW, NOT JUST =.

        ;WAIT FOR YEAR >= 1983
        LD      DE,1983
WAITYR: DI                      ;DISABLE INTERRUPTS TO LOAD 2-BYTE YEAR
        LD      H.(IX+YRIDX+1)  ;GET YEAR
        LD      L,(IX+YRIDX)
        EI
        OR      A               ;CLEAR CARRY
        SBC     HL,DE           ;COMPARE YEAR, 1983
        JR      C.WAITYR        ;JUMP IF NOT 1983

        ;WAIT FOR MONTH >= 2
        PUSH    IX
        POP     HL              ;HL = BASE ADDRESS OF CLOCK VARIABLES
        LD      DE.MTHIDX
        ADD     HL,DE           ;POINT AT MONTH
        LD      B,2
        CALL    WAIT            ;WAIT FOR FEBRUARY OR LATER

        ;WAIT FOR DAY >= 7
        DEC     HL              ;POINT AT DAY
        LD      B,7
        CALL    WAIT            ;WAIT FOR 7TH OR LATER

        ;WAIT FOR HOUR >= 14
        DEC     HL              ;POINT AT HOUR
        LD      B,14
        CALL    WAIT            ;WAIT FOR 2 PM OR LATER

        ;WAIT FOR MINUTE >= 1
        DEC     HL              ;POINT AT MINUTE
        LD      B,1
        CALL    WAIT            ;WAIT FOR 2:01 OR LATER

        ;WAIT FOR SECOND >= 20
        DEC     HL              ;POINT AT SECOND
        LD      B,20
        CALL    WAIT            ;WAIT FOR 2:01.20 OR LATER

        ;DONE
HERE:
        JP      HERE            ;IT IS NOW TIME OR LATER


        ;********************************
        ;ROUTINE: WAIT
        ;PURPOSE: WAIT FOR VALUE POINTED TO BY HL
        ;         TO BECOME GREATER THAN OR EQUAL TO VALUE IN B
        ;ENTRY: HL = ADDRESS OF VARIABLE TO WATCH
        ;       B = VALUE TO WAIT FOR
        ;EXIT:  WHEN B >= (HL)
        ;USED:  AF
        ;********************************
WAIT:
        LD      A,(HL)          ;GET PART OF CLOCK TIME
        CP      B               ;COMPARE TO TARGET
        JR      C,WAIT          ;WAIT IF TARGET NOT REACHED
        RET

        END
```

Appendix A **Z80 Instruction Set Summary**

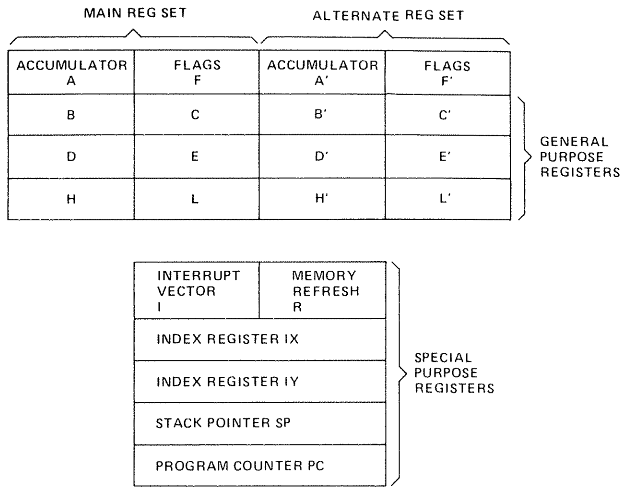


Figure A-1. Z80 internal register organization

7	6	5	4	3	2	1	0
S	Z	X	H	X	P/V	N	C

WHERE:

C	=	CARRY FLAG
N	=	ADD/SUBTRACT FLAG
P/V	=	PARITY/OVERFLOW FLAG
H	=	HALF-CARRY FLAG
Z	=	ZERO FLAG
S	=	SIGN FLAG
X	=	NOT USED

Figure A-2. Organization of the Z80 flag register

MASKABLE ($\overline{\text{INT}}$)
MODE 0

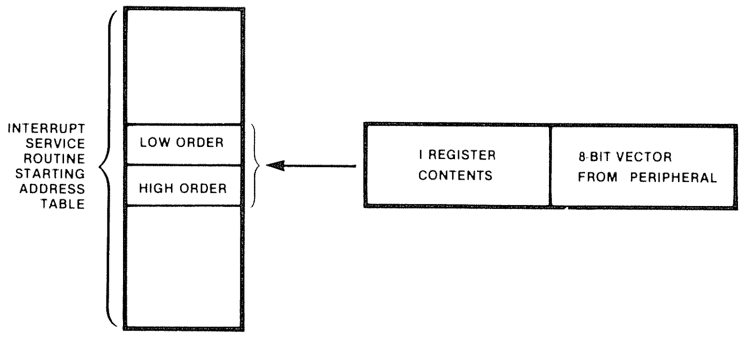
PLACE INSTRUCTION ONTO DATA BUS DURING $\overline{\text{INTACK}} = \overline{\text{MI}} \cdot \overline{\text{IORQ}}$ LIKE 8080A

MODE 1

RESTART TO 38_H OR 56_{10} ('RST 56')

MODE 2

USED BY Z80 PERIPHERALS


INTERRUPT SERVICE ROUTINE STARTING ADDRESS TABLE
LOW ORDER
HIGH ORDER
I REGISTER CONTENTS
8-BIT VECTOR FROM PERIPHERAL

NON MASKABLE ($\overline{\text{NMI}}$)

RESTART TO 66_H OR 102_{10}

INTERRUPT ENABLE / DISABLE FLIP-FLOPS

ACTION	IFF_1	IFF_2	
CPU RESET	0	0	
DI	0	0	
EI	1	1	
LD A, I	•	•	IFF_2 - PARITY FLAG
LD A, R	•	•	IFF_2 - PARITY FLAG
ACCEPT $\overline{\text{NMI}}$	0	•	
RETN	IFF_2	•	IFF_2 - IFF_1
ACCEPT $\overline{\text{INT}}$	0	0	
RETI	•	•	

"•" INDICATES NO CHANGE

Figure A-3. Z80 interrupt structure

Table A-1. Z80 Instructions in Alphabetical Order

ADC HL, ss	Add with Carry Reg. pair ss to HL
ADC A, s	Add with carry operand s to Acc.
ADD A, n	Add value n to Acc.
ADD A, r	Add Reg. r to Acc.
ADD A, (HL)	Add location (HL) to Acc.
ADD A, (IX+d)	Add location (IX+d) to Acc.
ADD A, (IY+d)	Add location (IY+d) to Acc.
ADD HL, ss	Add Reg. pair ss to HL
ADD IX, pp	Add Reg. pair pp to IX
ADD IY, rr	Add Reg. pair rr to IY
AND s	Logical 'AND' of operand s and Acc.
BIT b, (HL)	Test BIT b of location (HL)
BIT b, (IX+d)	Test BIT b of location (IX+d)
BIT b, (IY+d)	Test BIT b of location (IY+d)
BIT b, r	Test BIT b of Reg. r
CALL cc, nn	Call subroutine at location nn if condition cc if true
CALL nn	Unconditional call subroutine at location nn
CCF	Complement carry flag
CP s	Compare operand s with Acc.
CPD	Compare location (HL) and Acc. decrement HL and BC
CPDR	Compare location (HL) and Acc. decrement HL and BC, repeat until BC=0
CPI	Compare location (HL) and Acc. increment HL and decrement BC
CPIR	Compare location (HL) and Acc. increment HL, decrement BC repeat until BC=0
CPL	Complement Acc. (1's comp)
DAA	Decimal adjust Acc.
DEC m	Decrement operand m
DEC IX	Decrement IX
DEC IY	Decrement IY
DEC ss	Decrement Reg. pair ss
DI	Disable interrupts
DJNZ e	Decrement B and Jump relative if B≠0
EI	Enable interrupts
EX (SP), HL	Exchange the location (SP) and HL
EX (SP), IX	Exchange the location (SP) and IX
EX (SP), IY	Exchange the location (SP) and IY
EX AF, AF'	Exchange the contents of AF and AF'
EX DE, HL	Exchange the contents of DE and HL
EXX	Exchange the contents of BC, DE, HL with contents of BC', DE', HL' respectively
HALT	HALT (wait for interrupt or reset)
IM 0	Set interrupt mode 0
IM 1	Set interrupt mode 1
IM 2	Set interrupt mode 2
IN A, (n)	Load the Acc. with input from device n
IN r, (C)	Load the Reg. r with input from device (C)
INC (HL)	Increment location (HL)
INC IX	Increment IX
INC (IX+d)	Increment location (IX+d)

Table A-1. (Continued)

INC IY	Increment IY	LD A, (nn)	Load Acc. with location nn
INC (IY+d)	Increment location (IY+d)	LD A, R	Load Acc. with Reg. R
INC r	Increment Reg. r	LD (BC), A	Load location (BC) with Acc.
INC ss	Increment Reg. pair ss	LD (DE), A	Load location (DE) with Acc.
IND	Load location (HL) with input from port (C), decrement HL and B	LD (HL), n	Load location (HL) with value n
		LD dd, nn	Load Reg. pair dd with value nn
INDR	Load location (HL) with input from port (C), decrement HL and decrement B, repeat until B=0	LD HL, (nn)	Load HL with location (nn)
		LD (HL), r	Load location (HL) with Reg. r
INI	Load location (HL) with input from port (C); and increment HL and decrement B	LD I, A	Load I with Acc.
		LF IX, nn	Load IX with value nn
		LD IX, (nn)	Load IX with location (nn)
INIR	Load location (HL) with input from port (C), increment HL and decrement B, repeat until B=0	LD (IX+d), n	Load location (IX+d) with value n
		LD (IX+d), r	Load location (IX+d) with Reg. r
		LD IY, nn	Load IY with value nn
JP (HL)	Unconditional Jump to (HL)	LD IY, (nn)	Load IY with location (nn)
JP (IX)	Unconditional Jump to (IX)	LD (IY+d), n	Load location (IY+d) with value n
JP (IY)	Unconditonal Jump to (IY)	LD (IY+d), r	Load location (IY+d) with Reg. r
JP cc, nn	Jump to location nn if condition cc is true	LD (nn), A	Load location (nn) with Acc.
JP nn	Unconditional jump to location nn	LD (nn), dd	Load location (nn) with Reg. pair dd
		LD (nn), HL	Load location (nn) with HL
JP C, e	Jump relative to PC+e if carry=1	LD (nn), IX	Load location (nn) with IX
JR e	Unconditional Jump relative to PC+e	LD (nn), IY	Load location (nn) with IY
JP NC, e	Jump relative to PC+e if carry=0	LD R, A	Load R with Acc.
JR NZ, e	Jump relative to PC+e if non zero (Z=0)	LD r, (HL)	Load Reg. r with location (HL)
		LD r, (IX+d)	Load Reg. r with location (IX+d)
JR Z, e	Jump relative to PC+e if zero (Z=1)	LD r, (IY+d)	Load Reg. r with location (IY+d)
LD A, (BC)	Load Acc. with location (BC)	LD r, n	Load Reg. r with value n
LD A, (DE)	Load Acc. with location (DE)	LD r, r′	Load Reg. r with Reg. r′
LD A, I	Load Acc. with I	LD SP, HL	Load SP with HL

Table A-1. (Continued)

LD SP, IX	Load SP with IX	RES b, m	Reset Bit b of operand m
LD SP, IY	Load SP with IY	RET	Return from subroutine
LDD	Load location (DE) with location (HL), decrement DE, HL and BC	RET cc	Return from subroutine if condition cc is true
LDDR	Load location (DE) with location (HL), decrement DE, HL and BC; repeat until BC=0	RETI	Return from interrupt
		RETN	Return from non maskable interrupt
LDI	Load location (DE) with location (HL), increment DE, HL, decrement BC	RL m	Rotate left through carry operand m
		RLA	Rotate left Acc. through carry
LDIR	Load location (DE) with location (HL), increment DE, HL, decrement BC and repeat until BC=0	RLC (HL)	Rotate location (HL) left circular
		RLC (IX+d)	Rotate location (IX+d) left circular
		RLC (IY+d)	Rotate location (IY+d) left circular
NEG	Negate Acc. (2's complement)	RLC r	Rotate Reg. r left circular
NOP	No operation	RLCA	Rotate left circular Acc.
OR s	Logical 'OR' or operand s and Acc.	RLD	Rotate digit left and right between Acc. and location (HL)
OTDR	Load output port (C) with location (HL) decrement HL and B, repeat until B=0	RR m	Rotate right through carry operand m
		RRA	Rotate right Acc. through carry
OTIR	Load output port (C) with location (HL), increment HL, decrement B, repeat until B=0	RRC m	Rotate operand m right circular
		RRCA	Rotate right circular Acc.
OUT (C), r	Load output port (C) with Reg. r	RRD	Rotate digit right and left between Acc. and location (HL)
OUT (n), A	Load output port (n) with Acc.		
OUTD	Load output port (C) with location (HL), decrement HL and B	RST p	Restart to location p
OUTI	Load output port (C) with location (HL), increment HL and decrement B	SBC A, s	Subtract operand s from Acc. with carry
		SBC HL, ss	Subtract Reg. pair ss from HL with carry
POP IX	Load IX with top of stack		
POP IY	Load IY with top of stack	SCF	Set carry flag (C=1)
POP qq	Load Reg. pair qq with top of stack	SET b, (HL)	Set Bit b of location (HL)
PUSH IX	Load IX onto stack	SET b, (IX+d)	Set Bit b of location (IX+d)
PUSH IY	Load IY onto stack	SET b, (IY+d)	Set Bit b of location (IY+d)
PUSH qq	Load Reg. pair qq onto stack	SET b, r	Set Bit b of Reg. r

Table A-1. (Continued)

SLA m	Shift operand m left arithmetic	SUB s	Subtract operand s from Acc.
SRA m	Shift operand m right arithmetic	XOR s	Exclusive 'OR' operand s and Acc.
SRL m	Shift operand m right logical		

Table A-2. Z80 Operation Codes in Numerical Order

OBJECT CODE	INSTRUCTION	
00	NOP	
01 yyyy	LD	BC,data16
02	LD	(BC),A
03	INC	BC
04	INC	B
05	DEC	B
06 yy	LD	B,data
07	RLCA	
08	EX	AF,AF'
09	ADD	HL,BC
0A	LD	A,(BC)
0B	DEC	BC
0C	INC	C
0D	DEC	C
0E yy	LD	C,data
0F	RRCA	
10 disp-2	DJNZ	disp
11 yyyy	LD	DE,data16
12	LD	(DE),A
13	INC	DE
14	INC	D
15	DEC	D
16 yy	LD	D,data
17	RLA	
18 disp-2	JR	disp
19	ADD	HL,DE
1A	LD	A,(DE)
1B	DEC	DE
1C	INC	E
1D	DEC	E
1E yy	LD	E,data
1F	RRA	
20 disp-2	JR	NZ,disp
21 yyyy	LD	HL,data16
22 ppqq	LD	(addr),HL
23	INC	HL
24	INC	H
25	DEC	H
26 yy	LD	H,data
27	DAA	
28 disp-2	JR	Z,disp
29	ADD	HL,HL
2A ppqq	LD	HL,(addr)
2B	DEC	HL
2C	INC	L
2D	DEC	L
2E	LD	L,data
2F	CPL	
30 disp-2	JR	NC,disp
31 yyyy	LD	SP,data16

Table A-2. (Continued)

OBJECT CODE	INSTRUCTION	
32 ppqq	LD	(addr),A
33	INC	SP
34	INC	(HL)
35	DEC	(HL)
36 yy	LD	(HL),data
37	SCF	
38	JR	C,disp
39	ADD	HL,SP
3A ppqq	LD	A,(addr)
3B	DEC	SP
3C	INC	A
3D	DEC	A
3E yy	LD	A,data
3F	CCF	
4 0sss	LD	B,reg
46	LD	B,(HL)
4 1sss	LD	C,reg
4E	LD	C,(HL)
5 0sss	LD	D,reg
56	LD	D,(HL)
5 1sss	LD	E,reg
5E	LD	E,(HL)
6 0sss	LD	H,reg
66	LD	H,(HL)
6 1sss	LD	L,reg
6E	LD	L,(HL)
7 0sss	LD	(HL),reg
76	HALT	
7 1sss	LD	A,reg
7E	LD	A,(HL)
8 0rrr	ADD	A,reg
86	ADD	A,(HL)
8 1rrr	ADC	A,reg
8E	ADC	A,(HL)
9 0rrr	SUB	reg
96	SUB	(HL)
9 1rrr	SBC	A,reg
9E	SBC	A,(HL)
A 0rrr	AND	reg
A6	AND	(HL)
A 1rrr	XOR	reg
AE	XOR	(HL)
B 0rrr	OR	reg
B6	OR	(HL)
B 1rrr	CP	reg
BE	CP	(HL)
C0	RET	NZ
C1	POP	BC
C2 ppqq	JP	NZ,addr
C3 ppqq	JP	addr
C4 ppqq	CALL	NZ,addr
C5	PUSH	BC
C6 yy	ADD	A,data
C7	RST	00H
C8	RET	Z
C9	RET	
CA ppqq	JP	Z,addr

OBJECT CODE	INSTRUCTION	
CB 0 0rrr	RLC	reg
CB 06	RLC	(HL)
CB 0 1rrr	RRC	reg
CB 0E	RRC	(HL)
CB 1 0rrr	RL	reg
CB 16	RL	(HL)
CB 1 1rrr	RR	reg
CB 1E	RR	(HL)
CB 2 0rrr	SLA	reg
CB 26	SLA	(HL)
CB 2 1rrr	SRA	reg
CB 2E	SRA	(HL)
CB 3 1rrr	SRL	reg
CB 3E	SRL	(HL)
CB 01bbbrrr	BIT	b,reg
CB 01bbb110	BIT	b,(HL)
CB 10bbbrrr	RES	b,reg
CB 10bbb110	RES	b,(HL)
CB 11bbbrrr	SET	b,reg
CB 11bbb110	SET	b,(HL)
CC ppqq	CALL	Z,addr
CD ppqq	CALL	addr
CE yy	ADC	A,data
CF	RST	08H
D0	RET	NC
D1	POP	DE
D2 ppqq	JP	NC,addr
D3 yy	OUT	(port),A
D4 ppqq	CALL	NC,addr
D5	PUSH	DE
D6 yy	SUB	data
D7	RST	10H
D8	RET	C
D9	EXX	
DA ppqq	JP	C,addr
DB yy	IN	A,(port)
DC ppqq	CALL	C,addr
DD 00xx 9	ADD	IX,pp
DD 21 yyyy	LD	IX,data16
DD 22 ppqq	LD	(addr),IX
DD 23	INC	IX
DD 2A ppqq	LD	IX,(addr)
DD 2B	DEC	IX
DD 34 disp	INC	(IX + disp)
DD 35 disp	DEC	(IX + disp)
DD 36 disp yy	LD	(IX + disp),data
DD 01ddd110 disp	LD	reg,(IX + disp)
DD 7 0sss disp	LD	(IX + disp),reg
DD 86 disp	ADD	A,(IX + disp)
DD 8E disp	ADC	A,(IX + disp)
DD 96 disp	SUB	(IX + disp)
DD 9E disp	SBC	A,(IX + disp)
DD A6 disp	AND	(IX + disp)
DD AE disp	XOR	(IX + disp)
DD B6 disp	OR	(IX + disp)
DD BE disp	CP	(IX + disp)
DD CB disp 06	RLC	(IX + disp)

Table A-2. (Continued)

OBJECT CODE	INSTRUCTION	
DD CB disp 0E	RRC	(IX + disp)
DD CB disp 16	RL	(IX + disp)
DD CB disp 1E	RR	(IX + disp)
DD CB disp 26	SLA	(IX + disp)
DD CB disp 2E	SRA	(IX + disp)
DD CB disp 3E	SRL	(IX + disp)
DD CB disp 01bbb110	BIT	b,(IX + disp)
DD CB disp 10bbb110	RES	b,(IX + disp)
DD CB disp 11bbb110	SET	b,(IX + disp)
DD E1	POP	IX
DD E3	EX	(SP),IX
DD E5	PUSH	IX
DD E9	JP	(IX)
DD F9	LD	SP,IX
DE yy	SBC	A,data
DF	RST	18H
E0	RET	PO
E1	POP	HL
E2 ppqq	JP	PO,addr
E3	EX	(SP),HL
E4 ppqq	CALL	PO,addr
E5	PUSH	HL
E6 yy	AND	data
E7	RST	20H
E8	RET	PE
E9	JP	(HL)
EA ppqq	JP	PE,addr
EB	EX	DE,HL
EC ppqq	CALL	PE,addr
ED 01ddd000	IN	reg,(C)
ED 01sss001	OUT	(C),reg
ED 01xx 2	SBC	HL,rp
ED 01xx 3 ppqq	LD	(addr),rp
ED 44	NEG	
ED 45	RETN	
ED 010nn110	IM	m
ED 47	LD	I,A
ED 01xx≠A	ADC	HL,rp
ED 01xx B ppqq	LD	rp,(addr)
ED 4D	RETI	
ED 4F	LD	R,A
ED 57	LD	A,I
ED 5F	LD	A,R
ED 67	RRD	
ED 6F	RLD	
ED A0	LDI	
ED A1	CPI	
ED A2	INI	
ED A3	OUTI	
ED A8	LDD	
ED A9	CPD	
ED AA	IND	
ED AB	OUTD	
ED B0	LDIR	
ED B1	CPIR	
ED B2	INIR	
ED B3	OTIR	
ED B8	LDDR	
ED B9	CPDR	
ED BA	INDR	
ED BB	OTDR	
EE yy	XOR	data
EF	RST	28H
F0	RET	P
F1	POP	AF
F2 ppqq	JP	P,addr
F3	DI	
F4 ppqq	CALL	P,addr
F5	PUSH	AF
F6 yy	OR	data
F7	RST	30H
F8	RET	M
F9	LD	SP,HL
FA ppqq	JP	M,addr
FB	EI	
FC ppqq	CALL	M,addr
FD 00xx 9	ADD	IY,rr
FD 21 yyyy	LD	IY,data16
FD 22 ppqq	LD	(addr),IY
FD 23	INC	IY
FD 2A ppqq	LD	IY,(addr)
FD 2B	DEC	IY
FD 34 disp	INC	(IY + disp)
FD 35 disp	DEC	(IY + disp)
FD 36 disp yy	LD	(IY + disp),data
FD 01ddd110 disp	LD	reg,(IY + disp)
FD 7 0sss disp	LD	(IY + disp),reg
FD 86 disp	ADD	A,(IY + disp)
FD 8E disp	ADC	A,(IY + disp)
FD 96 disp	SUB	(IY + disp)
FD 9E disp	SBC	A,(IY + disp)
FD A6 disp	AND	(IY + disp)
FD AE disp	XOR	(IY + disp)
FD B6 disp	OR	(IY + disp)
FD BE disp	CP	(IY + disp)
FD CB disp 06	RLC	(IY + disp)
FD CB disp 0E	RRC	(IY + disp)
FD CB disp 16	RL	(IY + disp)
FD CB disp 1E	RR	(IY + disp)
FD CB disp 26	SLA	(IY + disp)
FD CB disp 2E	SRA	(IY + disp)
FD CB disp 3E	SRL	(IY + disp)
FD CB disp 01bbb110	BIT	b,(IY + disp)
FD CB disp 10bbb110	RES	b,(IY + disp)
FD CB disp 11bbb110	SET	b,(IY + disp)
FD E1	POP	IY
FD E3	EX	(SP),IY
FD E5	PUSH	IY
FD E9	JP	(IY)
FD F9	LD	SP,IY
FE yy	CP	data
FF	RST	38H

Table A-3. Z80 8-Bit Load Instructions

Mnemonic	Symbolic Operation	Flags						OP-Code	No. of Bytes	No. of M Cycles	No. of T Cycles	Comments	
		C	Z	P/V	S	N	H	76 543 210					
LD r, r'	r ← r'	•	•	•	•	•	•	01 r r'	1	1	4	r, r'	Reg.
LD r, n	r ← n	•	•	•	•	•	•	00 r 110	2	2	7	000	B
								← n →				001	C
LD r, (HL)	r ← (HL)	•	•	•	•	•	•	01 r 110	1	2	7	010	D
LD r, (IX+d)	r ← (IX+d)	•	•	•	•	•	•	11 011 101	3	5	19	011	E
								01 r 110				100	H
								← d →				101	L
LD r, (IY+d)	r ← (IY+d)	•	•	•	•	•	•	11 111 101	3	5	19	111	A
								01 r 110					
								← d →					
LD (HL), r	(HL) ← r	•	•	•	•	•	•	01 110 r	1	2	7		
LD (IX+d), r	(IX+d) ← r	•	•	•	•	•	•	11 011 101	3	5	19		
								01 110 r					
								← d →					
LD (IY+d), r	(IY+d) ← r	•	•	•	•	•	•	11 111 101	3	5	19		
								01 110 r					
								← d →					
LD (HL), n	(HL) ← n	•	•	•	•	•	•	00 110 110	2	3	10		
								← n →					
LD (IX+d), n	(IX+d) ← n	•	•	•	•	•	•	11 011 101	4	5	19		
								00 110 110					
								← d →					
								← n →					
LD (IY+d), n	(IY+d) ← n	•	•	•	•	•	•	11 111 101	4	5	19		
								00 110 110					
								← d →					
								← n →					
LD A, (BC)	A ← (BC)	•	•	•	•	•	•	00 001 010	1	2	7		
LD A, (DE)	A ← (DE)	•	•	•	•	•	•	00 011 010	1	2	7		
LD A, (nn)	A ← (nn)	•	•	•	•	•	•	00 111 010	3	4	13		
								← n →					
								← n →					
LD (BC), A	(BC) ← A	•	•	•	•	•	•	00 000 010	1	2	7		
LD (DE), A	(DE) ← A	•	•	•	•	•	•	00 010 010	1	2	7		
LD (nn), A	(nn) ← A	•	•	•	•	•	•	00 110 010	3	4	13		
								← n →					
								← n →					
LD A, I	A ← I	•	↕	IFF	↕	0	0	11 101 101	2	2	9		
								01 010 111					
LD A, R	A ← R	•	↕	IFF	↕	0	0	11 101 101	2	2	9		
								01 011 111					
LD I, A	I ← A	•	•	•	•	•	•	11 101 101	2	2	9		
								01 000 111					
LD R, A	R ← A	•	•	•	•	•	•	11 101 101	2	2	9		
								01 001 111					

Notes: r, r' means any of the registers A, B, C, D, E, H, L

IFF the content of the interrupt enable flip-flop (IFF) is copied into the P/V flag

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,

↕ = flag is affected according to the result of the operation.

Table A-4. Z80 16-Bit Load Instructions

Mnemonic	Symbolic Operation	Flags						Op-Code	No. of Bytes	No. of M Cycles	No. of T States	Comments	
		C	Z	P/V	S	N	H	76 543 210					
LD dd, nn	dd ← nn	•	•	•	•	•	•	00 dd0 001	3	3	10	dd	Pair
								← n →				00	BC
								← n →				01	DE
LD IX, nn	IX ← nn	•	•	•	•	•	•	11 011 101	4	4	14	10	HL
								00 100 001				11	SP
								← n →					
								← n →					
LD IY, nn	IY ← nn	•	•	•	•	•	•	11 111 101	4	4	14		
								00 100 001					
								← n →					
								← n →					
LD HL, (nn)	H ← (nn+1)	•	•	•	•	•	•	00 101 010	3	5	16		
	L ← (nn)							← n →					
								← n →					
LD dd, (nn)	dd_H ← (nn+1)	•	•	•	•	•	•	11 101 101	4	6	20		
	dd_L ← (nn)							01 dd1 011					
								← n →					
								← n →					
LD IX, (nn)	IX_H ← (nn+1)	•	•	•	•	•	•	11 011 101	4	6	20		
	IX_L ← (nn)							00 101 010					
								← n →					
								← n →					
LD IY, (nn)	IY_H ← (nn+1)	•	•	•	•	•	•	11 111 101	4	6	20		
	IY_L ← (nn)							00 101 010					
								← n →					
								← n →					
LD (nn), HL	(nn+1) ← H	•	•	•	•	•	•	00 100 010	3	5	16		
	(nn) ← L							← n →					
								← n →					
LD (nn), dd	(nn+1) ← dd_H	•	•	•	•	•	•	11 101 101	4	6	20		
	(nn) ← dd_L							01 dd0 011					
								← n →					
								← n →					
LD (nn), IX	(nn+1) ← IX_H	•	•	•	•	•	•	11 011 101	4	6	20		
	(nn) ← IX_L							00 100 010					
								← n →					
								← n →					
LD (nn), IY	(nn+1) ← IY_H	•	•	•	•	•	•	11 111 101	4	6	20		
	(nn) ← IY_L							00 100 010					
								← n →					
								← n →					
LD SP, HL	SP ← HL	•	•	•	•	•	•	11 111 001	1	1	6		
LD SP, IX	SP ← IX	•	•	•	•	•	•	11 011 101	2	2	10		
								11 111 001					
LD SP, IY	SP ← IY	•	•	•	•	•	•	11 111 101	2	2	10		
								11 111 001				qq	Pair
PUSH qq	(SP-2) ← qq_L	•	•	•	•	•	•	11 qq0 101	1	3	11	00	BC
	(SP-1) ← qq_H											01	DE
PUSH IX	(SP-2) ← IX_L	•	•	•	•	•	•	11 011 101	2	4	15	10	HL
	(SP-1) ← IX_H							11 100 101				11	AF
PUSH IY	(SP-2) ← IY_L	•	•	•	•	•	•	11 111 101	2	4	15		
	(SP-1) ← IY_H							11 100 101					
POP qq	qq_H ← (SP+1)	•	•	•	•	•	•	11 qq0 001	1	3	10		
	qq_L ← (SP)												
POP IX	IX_H ← (SP+1)	•	•	•	•	•	•	11 011 101	2	4	14		
	IX_L ← (SP)							11 100 001					
POP IY	IY_H ← (SP+1)	•	•	•	•	•	•	11 111 101	2	4	14		
	IY_L ← (SP)							11 100 001					

Notes: dd is any of the register pairs BC, DE, HL, SP
qq is any of the register pairs AF, BC, DE, HL
$(PAIR)_H$, $(PAIR)_L$ refer to high order and low order eight bits of the register pair respectively
E.g. BC_L = C, AF_H = A

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ flag is affected according to the result of the operation.

Table A-5. Z80 Exchange, Block Transfer, and Block Search Instructions

		Flags						Op-Code				
Mnemonic	Symbolic Operation	C	Z	P/V	S	N	H	76 543 210	No. of Bytes	No. of M Cycles	No. of T States	Comments
EX DE, HL	DE ↔ HL	•	•	•	•	•	•	11 101 011	1	1	4	
EX AF, AF'	AF ↔ AF'	•	•	•	•	•	•	00 001 000	1	1	4	
EXX	$\begin{pmatrix}BC\\DE\\HL\end{pmatrix} \leftrightarrow \begin{pmatrix}BC'\\DE'\\HL'\end{pmatrix}$	•	•	•	•	•	•	11 011 001	1	1	4	Register bank and auxiliary register bank exchange
EX (SP), HL	H ↔ (SP+1) L ↔ (SP)	•	•	•	•	•	•	11 100 011	1	5	19	
EX (SP), IX	IX_H ↔ (SP+1) IX_L ↔ (SP)	•	•	•	•	•	•	11 011 101 11 100 011	2	6	23	
EX (SP), IY	IY_H ↔ (SP+1) IY_L ↔ (SP)	•	•	•	•	•	•	11 111 101 11 100 011	2	6	23	
LDI	(DE) ← (HL) DE ← DE+1 HL ← HL+1 BC ← BC-1	•	•	① ↕	•	0	0	11 101 101 10 100 000	2	4	16	Load (HL) into (DE), increment the pointers and decrement the byte counter (BC)
LDIR	(DE) ← (HL) DE ← DE+1 HL ← HL+1 BC ← BC-1 Repeat until BC = 0	•	•	0	•	0	0	11 101 101 10 110 000	2 2	5 4	21 16	If BC ≠ 0 If BC = 0
LDD	(DE) ← (HL) DE ← DE-1 HL ← HL-1 BC ← BC-1	•	•	① ↕	•	0	0	11 101 101 10 101 000	2	4	16	
LDDR	(DE) ← (HL) DE ← DE-1 HL ← HL-1 BC ← BC-1 Repeat until BC = 0	•	•	0	•	0	0	11 101 101 10 111 000	2 2	5 4	21 16	If BC ≠ 0 If BC = 0
CPI	A - (HL) HL ← HL+1 BC ← BC - 1	•	② ↕	① ↕	↕	1	↕	11 101 101 10 100 001	2	4	16	
CPIR	A - (HL) HL ← HL+1 BC ← BC-1 Repeat until A = (HL) or BC = 0	•	② ↕	① ↕	↕	1	↕	11 101 101 10 110 001	2 2	5 4	21 16	If BC ≠ 0 and A ≠ (HL) If BC = 0 or A = (HL)
CPD	A - (HL) HL ← HL-1 BC ← BC-1	•	② ↕	① ↕	↕	1	↕	11 101 101 10 101 001	2	4	16	
CPDR	A - (HL) HL ← HL-1 BC ← BC-1 Repeat until A = (HL) or BC = 0	•	② ↕	① ↕	↕	1	↕	11 101 101 10 111 001	2 2	5 4	21 16	If BC ≠ 0 and A ≠ (HL) If BC = 0 or A = (HL)

Notes: ① P/V flag is 0 if the result of BC-1 = 0, otherwise P/V = 1
② Z flag is 1 if A = (HL), otherwise Z = 0.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ = flag is affected according to the result of the operation.

Table A-6. Z80 8-Bit Arithmetic and Logical Instructions

Mnemonic	Symbolic Operation	Flags C	Z	P/V	S	N	H	Op-Code 76 543 210	No. of Bytes	No. of M Cycles	No. of T States	Comments
ADD A, r	A ← A + r	↕	↕	V	↕	0	↕	10 $\boxed{000}$ r	1	1	4	r Reg.
ADD A, n	A ← A + n	↕	↕	V	↕	0	↕	11 $\boxed{000}$ 110	2	2	7	000 B
								← n →				001 C
ADD A, (HL)	A ← A + (HL)	↕	↕	V	↕	0	↕	10 $\boxed{000}$ 110	1	2	7	010 D
ADD A, (IX+d)	A←A + (IX+d)	↕	↕	V	↕	0	↕	11 011 101	3	5	19	011 E
								10 $\boxed{000}$ 110				100 H
								← d →				101 L 111 A
ADD A, (IY+d)	A←A+(IY+d)	↕	↕	V	↕	0	↕	11 111 101	3	5	19	
								10 $\boxed{000}$ 110				
								← d →				
ADC A, s	A ← A + s + CY	↕	↕	V	↕	0	↕	$\boxed{001}$				s is any of r, n, (HL), (IX+d), (IY+d) as shown for ADD instruction
SUB s	A ← A − s	↕	↕	V	↕	1	↕	$\boxed{010}$				
SBC A, s	A ← A − s − CY	↕	↕	V	↕	1	↕	$\boxed{011}$				
AND s	A ← A ∧ s	0	↕	P	↕	0	1	$\boxed{100}$				
OR s	A ← A ∨ s	0	↕	P	↕	0	0	$\boxed{110}$				The indicated bits replace the 000 in the ADD set above
XOR s	A ← A ⊕ s	0	↕	P	↕	0	0	$\boxed{101}$				
CP s	A − s	↕	↕	V	↕	1	↕	$\boxed{111}$				
INC r	r ← r + 1	•	↕	V	↕	0	↕	00 r $\boxed{100}$	1	1	4	
INC (HL)	(HL) ← (HL)+1	•	↕	V	↕	0	↕	00 110 $\boxed{100}$	1	3	11	
INC (IX+d)	(IX+d) ← (IX+d)+1	•	↕	V	↕	0	↕	11 011 101	3	6	23	
								00 110 $\boxed{100}$				
								← d →				
INC (IY+d)	(IY+d) ← (IY+d) + 1	•	↕	V	↕	0	↕	11 111 101	3	6	23	
								00 110 $\boxed{100}$				
								← d →				
DEC m	m←m−1	•	↕	V	↕	1	↕	$\boxed{101}$				m is any of r, (HL), (IX+d), (IY+d) as shown for INC Same format and states as INC. Replace 100 with 101 in OP code.

Notes: The V symbol in the P/V flag column indicates that the P/V flag contains the overflow of the result of the operation. Similarly the P symbol indicates parity. V = 1 means overflow, V = 0 means not overflow. P = 1 means parity of the result is even, P = 0 means parity of the result is odd.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown.
↕ = flag is affected according to the result of the operation.

Table A-7. Z80 General-Purpose Arithmetic and CPU Control

Mnemonic	Symbolic Operation	Flags C	Z	P/V	S	N	H	Op-Code 76 543 210	No. of Bytes	No. of M Cycles	No. of T States	Comments
DAA	Converts acc. content into packed BCD following add or subtract with packed BCD operands	‡	‡	P	‡	●	‡	00 100 111	1	1	4	Decimal adjust accumulator
CPL	A ← $\overline{A}$	●	●	●	●	1	1	00 101 111	1	1	4	Complement accumulator (one's complement)
NEG	A ← 0 − A	‡	‡	V	‡	1	‡	11 101 101 01 000 100	2	2	8	Negate acc. (two's complement)
CCF	CY ← $\overline{CY}$	‡	●	●	●	0	X	00 111 111	1	1	4	Complement carry flag
SCF	CY ← 1	1	●	●	●	0	0	00 110 111	1	1	4	Set carry flag
NOP	No operation	●	●	●	●	●	●	00 000 000	1	1	4	
HALT	CPU halted	●	●	●	●	●	●	01 110 110	1	1	4	
DI	IFF ← 0	●	●	●	●	●	●	11 110 011	1	1	4	
EI	IFF ← 1	●	●	●	●	●	●	11 111 011	1	1	4	
IM 0	Set interrupt mode 0	●	●	●	●	●	●	11 101 101 01 000 110	2	2	8	
IM 1	Set interrupt mode 1	●	●	●	●	●	●	11 101 101 01 010 110	2	2	8	
IM2	Set interrupt mode 2	●	●	●	●	●	●	11 101 101 01 011 110	2	2	8	

Notes: IFF indicates the interrupt enable flip-flop
CY indicates the carry flip-flop.

Flag Notation: ● = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
‡ = flag is affected according to the result of the operation.

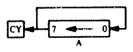
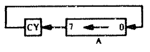
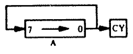
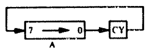
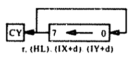
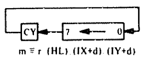
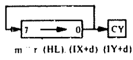
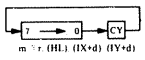
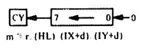
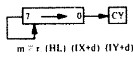
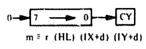
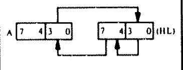
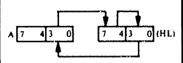
Table A-8. Z80 16-Bit Arithmetic Instructions

Mnemonic	Symbolic Operation	Flags						Op-Code	No. of Bytes	No. of M Cycles	No. of T States	Comments
		C	Z	P/V	S	N	H	76 543 210				
ADD HL, ss	HL ← HL+ss	↕	●	●	●	0	X	00 ss1 001	1	3	11	ss Reg. 00 BC 01 DE 10 HL 11 SP
ADC HL, ss	HL←HL+ss+CY	↕	↕	V	↕	0	X	11 101 101 01 ss1 010	2	4	15	
SBC HL, ss	HL←HL-ss-CY	↕	↕	V	↕	1	X	11 101 101 01 ss0 010	2	4	15	
ADD IX, pp	IX ← IX + pp	↕	●	●	●	0	X	11 011 101 00 pp1 001	2	4	15	pp Reg. 00 BC 01 DE 10 IX 11 SP
ADD IY, rr	IY←IY+rr	↕	●	●	●	0	X	11 111 101 00 rr1 001	2	4	15	rr Reg. 00 BC 01 DE 10 IY 11 SP
INC ss	ss ← ss + 1	●	●	●	●	●	●	00 ss0 011	1	1	6	
INC IX	IX ← IX + 1	●	●	●	●	●	●	11 011 101 00 100 011	2	2	10	
INC IY	IY ← IY + 1	●	●	●	●	●	●	11 111 101 00 100 011	2	2	10	
DEC ss	ss ← ss - 1	●	●	●	●	●	●	00 ss1 011	1	1	6	
DEC IX	IX ← IX - 1	●	●	●	●	●	●	11 011 101 00 101 011	2	2	10	
DEC IY	IY ← IY - 1	●	●	●	●	●	●	11 111 101 00 101 011	2	2	10	

Notes: ss is any of the register pairs BC, DE, HL, SP
pp is any of the register pairs BC, DE, IX, SP
rr is any of the register pairs BC, DE, IY, SP.

Flag Notation: ● = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ = flag is affected according to the result of the operation.

Table A-9. Z80 Rotate and Shift Instructions

Mnemonic	Symbolic Operation	Flags						Op-Code	No. of Bytes	No. of M Cycles	No. of T States	Comments
		C	Z	P/V	S	N	H	76 543 210				
RLCA		‡	•	•	•	0	0	00 000 111	1	1	4	Rotate left circular accumulator
RLA		‡	•	•	•	0	0	00 010 111	1	1	4	Rotate left accumulator
RRCA		‡	•	•	•	0	0	00 001 111	1	1	4	Rotate right circular accumulator
RRA		‡	•	•	•	0	0	00 011 111	1	1	4	Rotate right accumulator
RLC r		‡	‡	P	‡	0	0	11 001 011 00 $\boxed{000}$ r	2	2	8	Rotate left circular register r
RLC (HL)		‡	‡	P	‡	0	0	11 001 011 00 $\boxed{000}$ 110	2	4	15	r Reg.
RLC (IX+d)		‡	‡	P	‡	0	0	11 011 101 11 001 011 ← d → 00 $\boxed{000}$ 110	4	6	23	000 B 001 C 010 D 011 E 100 H 101 L 111 A
RLC (IY+d)		‡	‡	P	‡	0	0	11 111 101 11 001 011 ← d → 00 $\boxed{000}$ 110	4	6	23	
RL m		‡	‡	P	‡	0	0	$\boxed{010}$				Instruction format and states are as shown for RLC,m. To form new OP-code replace $\boxed{000}$ of RLC,m with shown code
RRC m		‡	‡	P	‡	0	0	$\boxed{001}$				
RR m		‡	‡	P	‡	0	0	$\boxed{011}$				
SLA m		‡	‡	P	‡	0	0	$\boxed{100}$				
SRA m		‡	‡	P	‡	0	0	$\boxed{101}$				
SRL m		‡	‡	P	‡	0	0	$\boxed{111}$				
RLD		•	‡	P	‡	0	0	11 101 101 01 101 111	2	5	18	Rotate digit left and right between the accumulator
RRD		•	‡	P	‡	0	0	11 101 101 01 100 111	2	5	18	and location (HL). The content of the upper half of the accumulator is unaffected

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown, ‡ = flag is affected according to the result of the operation.

Table A-10. Z80 Bit Manipulation Instructions

Mnemonic	Symbolic Operation	Flags C	Z	P/V	S	N	H	Op-Code 76 543 210	No. of Bytes	No. of M Cycles	No. of T States	Comments
BIT b, r	$Z \leftarrow \overline{r_b}$	•	‡	X	X	0	1	11 001 011	2	2	8	
								01 b r				
BIT b, (HL)	$Z \leftarrow \overline{(HL)}_b$	•	‡	X	X	0	1	11 001 011	2	3	12	
								01 b 110				
BIT b, (IX+d)	$Z \leftarrow \overline{(IX+d)}_b$	•	‡	X	X	0	1	11 011 101	4	5	20	
								11 001 011				
								← d →				
								01 b 110				
BIT b, (IY+d)	$Z \leftarrow \overline{(IY+d)}_b$	•	‡	X	X	0	1	11 111 101	4	5	20	
								11 001 011				
								← d →				
								01 b 110				
SET b, r	$r_b \leftarrow 1$	•	•	•	•	•	•	11 001 011	2	2	8	
								$\boxed{11}$ b r				
SET b, (HL)	$(HL)_b \leftarrow 1$	•	•	•	•	•	•	11 001 011	2	4	15	
								$\boxed{11}$ b 110				
SET b, (IX+d)	$(IX+d)_b \leftarrow 1$	•	•	•	•	•	•	11 011 101	4	6	23	
								11 001 011				
								← d →				
								$\boxed{11}$ b 110				
SET b, (IY+d)	$(IY+d)_b \leftarrow 1$	•	•	•	•	•	•	11 111 101	4	6	23	
								11 001 011				
								← d →				
								$\boxed{11}$ b 110				
RES b, m	$s_b \leftarrow 0$ m≡r, (HL), (IX+d), (IY+d)							$\boxed{10}$				To form new OP-code replace $\boxed{11}$ of SET b,m with $\boxed{10}$. Flags and time states for SET instruction

r	Reg.
000	B
001	C
010	D
011	E
100	H
101	L
111	A

b	Bit Tested
000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Notes: The notation s_b indicates bit b (0 to 7) or location s.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
‡ = flag is affected according to the result of the operation.

Table A-11. Z80 Jump Instructions

Mnemonic	Symbolic Operation	Flags						Op-Code	No. of Bytes	No. of M Cycles	No. of T States	Comments
		C	Z	P/V	S	N	H	76 543 210				
JP nn	PC ← nn	●	●	●	●	●	●	11 000 011 ← n → ← n →	3	3	10	
JP cc, nn	If condition cc is true PC ←nn, otherwise continue	●	●	●	●	●	●	11 cc 010 ← n → ← n →	3	3	10	
JR e	PC ← PC + e	●	●	●	●	●	●	00 011 000 ← e−2 →	2	3	12	
JR C, e	If C = 0, continue	●	●	●	●	●	●	00 111 000 ← e−2 →	2	2	7	If condition not met
	If C = 1, PC ← PC+e								2	3	12	If condition is met
JR NC, e	If C = 1, continue	●	●	●	●	●	●	00 110 000 ← e−2 →	2	2	7	If condition not met
	If C = 0, PC ← PC + e								2	3	12	If condition is met
JR Z, e	If Z = 0 continue	●	●	●	●	●	●	00 101 000 ← e−2 →	2	2	7	If condition not met
	If Z = 1, PC ← PC + e								2	3	12	If condition is met
JR NZ, e	If Z = 1, continue	●	●	●	●	●	●	00 100 000 ← e−2 →	2	2	7	If condition not me
	If Z = 0, PC ← PC + e								2	3	12	If condition met
JP (HL)	PC ← HL	●	●	●	●	●	●	11 101 001	1	1	4	
JP (IX)	PC ← IX	●	●	●	●	●	●	11 011 101 11 101 001	2	2	8	
JP (IY)	PC ← IY	●	●	●	●	●	●	11 111 101 11 101 001	2	2	8	
DJNZ,e	B ← B-1 If B = 0, continue	●	●	●	●	●	●	00 010 000 ← e−2 →	2	2	8	If B = 0
	If B ≠ 0, PC ← PC + e								2	3	13	If B ≠ 0

cc	Condition
000	NZ non zero
001	Z zero
010	NC non carry
011	C carry
100	PO parity odd
101	PE parity even
110	P sign positive
111	M sign negative

Notes: e represents the extension in the relative addressing mode.

e is a signed two's complement number in the range <−126, 129>

e−2 in the op-code provides an effective address of pc +e as PC is incremented by 2 prior to the addition of e.

Flag Notation: ● = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,

‡ = flag is affected according to the result of the operation.

Table A-12. Z80 Call and Return Instructions

Mnemonic	Symbolic Operation	Flags C	Flags Z	Flags P/V	Flags S	Flags N	Flags H	Op-Code 76 543 210	No. of Bytes	No. of M Cycles	No. of T States	Comments
CALL nn	$(SP-1) \leftarrow PC_H$	•	•	•	•	•	•	11 001 101	3	5	17	
	$(SP-2) \leftarrow PC_L$							← n →				
	$PC \leftarrow nn$							← n →				
CALL cc, nn	If condition cc is false	•	•	•	•	•	•	11 cc 100	3	3	10	If cc is false
	continue,							← n →				
	otherwise same as CALL nn							← n →	3	5	17	If cc is true
RET	$PC_L \leftarrow (SP)$	•	•	•	•	•	•	11 001 001	1	3	10	
	$PC_H \leftarrow (SP+1)$											
RET cc	If condition cc is false continue,	•	•	•	•	•	•	11 cc 000	1	1	5	If cc is false
	otherwise same as RET								1	3	11	If cc is true
RETI	Return from interrupt	•	•	•	•	•	•	11 101 101	2	4	14	
								01 001 101				
RETN	Return from non maskable interrupt	•	•	•	•	•	•	11 101 101	2	4	14	
								01 000 101				
RST p	$(SP-1) \leftarrow PC_H$	•	•	•	•	•	•	11 t 111	1	3	11	
	$(SP-2) \leftarrow PC_L$											
	$PC_H \leftarrow 0$											
	$PC_L \leftarrow P$											

cc	Condition	
000	NZ	non zero
001	Z	zero
010	NC	non carry
011	C	carry
100	PO	parity odd
101	PE	parity even
110	P	sign positive
111	M	sign negative

t	P
000	00H
001	08H
010	10H
011	18H
100	20H
101	28H
110	30H
111	38H

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown
‡ = flag is affected according to the result of the operation.

Table A-13. Z80 I/O Instructions

Mnemonic	Symbolic Operation	Flags						Op-Code	No. of Bytes	No. of M Cycles	No. of T States	Comments
		C	Z	P/V	S	N	H	76 543 210				
IN A, (n)	A ← (n)	•	•	•	•	•	•	11 011 011	2	3	11	n to $A_0 \sim A_7$
								← n →				Acc to $A_8 \sim A_{15}$
IN r, (C)	r ← (C)	•	↕	P	↕	0	↕	11 101 101	2	3	12	C to $A_0 \sim A_7$
	if r = 110 only the flags will be affected							01 r 000				B to $A_8 \sim A_{15}$
INI	(HL) ← (C)	X	① ↕	X	X	1	X	11 101 101	2	4	16	C to $A_0 \sim A_7$
	B ← B - 1							10 100 010				B to $A_8 \sim A_{15}$
	HL ← HL + 1											
INIR	(HL) ← (C)	X	1	X	X	1	X	11 101 101	2	5 (If B ≠ 0)	21	C to $A_0 \sim A_7$
	B ← B - 1							10 110 010				B to $A_8 \sim A_{15}$
	HL ← HL + 1								2	4 (If B = 0)	16	
	Repeat until B = 0											
IND	(HL) ← (C)	X	① ↕	X	X	1	X	11 101 101	2	4	16	C to $A_0 \sim A_7$
	B ← B - 1							10 101 010				B to $A_8 \sim A_{15}$
	HL ← HL - 1											
INDR	(HL) ← (C)	X	1	X	X	1	X	11 101 101	2	5 (If B ≠ 0)	21	C to $A_0 \sim A_7$
	B ← B - 1							10 111 010				B to $A_8 \sim A_{15}$
	HL ← HL -1								2	4 (If B = 0)	16	
	Repeat until B = 0											
OUT (n), A	(n) ← A	•	•	•	•	•	•	11 010 011	2	3	11	n to $A_0 \sim A_7$
								← n →				Acc to $A_8 \sim A_{15}$
OUT (C), r	(C) ← r	•	•	•	•	•	•	11 101 101	2	3	12	C to $A_0 \sim A_7$
								01 r 001				B to $A_8 \sim A_{15}$
OUTI	(C) ← (HL)	X	① ↕	X	X	1	X	11 101 101	2	4	16	C to $A_0 \sim A_7$
	B ← B - 1							10 100 011				B to $A_8 \sim A_{15}$
	HL ← HL + 1											
OTIR	(C) ← (HL)	X	1	X	X	1	X	11 101 101	2	5 (If B ≠ 0)	21	C to $A_0 \sim A_7$
	B ← B - 1							10 110 011				B to $A_8 \sim A_{15}$
	HL ← HL + 1								2	4 (If B = 0)	16	
	Repeat until B = 0											
OUTD	(C) ← (HL)	X	① ↕	X	X	1	X	11 101 101	2	4	16	C to $A_0 \sim A_7$
	B ← B - 1							10 101 011				B to $A_8 \sim A_{15}$
	HL ← HL - 1											
OTDR	(C) ← (HL)	X	1	X	X	1	X	11 101 101	2	5 (If B ≠ 0)	21	C to $A_0 \sim A_7$
	B ← B - 1							10 111 011				B to $A_8 \sim A_{15}$
	HL ← HL -1								2	4 (If B = 0)	16	
	Repeat until B = 0											

Notes: ① If the result of B - 1 is zero the Z flag is set, otherwise it is reset.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ = flag is affected according to the result of the operation.

Table A-14. Summary of Z80 Flag Operations

Instruction	C	Z	P/V	S	N	H	Comments
ADD A, s; ADC A,s	‡	‡	V	‡	0	‡	8-bit add or add with carry
SUB s; SBC A, s, CP s, NEG	‡	‡	V	‡	1	‡	8-bit subtract, subtract with carry, compare and negate accumulator
AND s	0	‡	P	‡	0	1	Logical operations
OR s; XOR s	0	‡	P	‡	0	0	And set's different flags
INC s	●	‡	V	‡	0	‡	8-bit increment
DEC m	●	‡	V	‡	1	‡	8-bit decrement
ADD DD, ss	‡	●	●	●	0	X	16-bit add
ADC HL, ss	‡	‡	V	‡	0	X	16-bit add with carry
SBC HL, ss	‡	‡	V	‡	1	X	16-bit subtract with carry
RLA; RLCA, RRA, RRCA	‡	●	●	●	0	0	Rotate accumulator
RL m; RLC m; RR m; RRC m SLA m; SRA m; SRL m	‡	‡	P	‡	0	0	Rotate and shift location m
RLD, RRD	●	‡	P	‡	0	0	Rotate digit left and right
DAA	‡	‡	P	‡	●	‡	Decimal adjust accumulator
CPL	●	●	●	●	1	1	Complement accumulator
SCF	1	●	●	●	0	0	Set carry
CCF	‡	●	●	●	0	X	Complement carry
IN r, (C)	●	‡	P	‡	0	0	Input register indirect
INI; IND; OUTI; OUTD	●	‡	X	X	1	X	Block input and output
INIR; INDR; OTIR; OTDR	●	1	X	X	1	X	Z = 0 if B ≠ 0 otherwise Z = 1
LDI, LDD	●	X	‡	X	0	0	Block transfer instructions
LDIR, LDDR	●	X	0	X	0	0	P/V = 1 if BC ≠ 0, otherwise P/V = 0
CPI, CPIR, CPD, CPDR	●	‡	‡	‡	1	X	Block search instructions Z = 1 if A = (HL), otherwise Z = 0 P/V = 1 if BC ≠ 0, otherwise P/V = 0
LD A, I; LD A, R	●	‡	IFF	‡	0	0	The content of the interrupt enable flip-flop (IFF) is copied into the P/V flag
BIT b, s	●	‡	X	X	0	1	The state of bit b of location s is copied into the Z flag
NEG	‡	‡	V	‡	1	‡	Negate accumulator

The following notation is used in this table:

Symbol	Operation
C	Carry/link flag. C=1 if the operation produced a carry from the MSB of the operand or result.
Z	Zero flag. Z=1 if the result of the operation is zero.
S	Sign flag. S=1 if the MSB of the result is one.
P/V	Parity or overflow flag. Parity (P) and overflow (V) share the same flag. Logical operations affect this flag with the parity of the result while arithmetic operations affect this flag with the overflow of the result. If P/V holds parity, P/V=1 if the result of the operation is even, P/V=0 if result is odd. If P/V holds overflow, P/V=1 if the result of the operation produced an overflow.
H	Half-carry flag. H=1 if the add or subtract operation produced a carry into or borrow from into bit 4 of the accumulator.
N	Add/Subtract flag. N=1 if the previous operation was a subtract.
	H and N flags are used in conjunction with the decimal adjust instruction (DAA) to properly correct the result into packed BCD format following addition or subtraction using operands with packed BCD format.
‡	The flag is affected according to the result of the operation.
●	The flag is unchanged by the operation.
0	The flag is reset by the operation.
1	The flag is set by the operation.
X	The flag is a "don't care."
V	P/V flag affected according to the overflow result of the operation.
P	P/V flag affected according to the parity result of the operation.
r	Any one of the CPU registers A, B, C, D, E, H, L.
s	Any 8-bit location for all the addressing modes allowed for the particular instruction. .
ss	Any 16-bit location for all the addressing modes allowed for that instruction.
ii	Any one of the two index registers IX or IY.
R	Refresh counter.
n	8-bit value in range <0, 255>
nn	16-bit value in range <0, 65535>
m	Any 8-bit location for all the addressing modes allowed for the particular instruction.

Table A-15. Summary of Z80 Restart Instructions

CALL ADDRESS	OP CODE	
0000_H	C7	'RST 0'
0008_H	CF	'RST 8'
0010_H	D7	'RST 16'
0018_H	DF	'RST 24'
0020_H	E7	'RST 32'
0028_H	EF	'RST 40'
0030_H	F7	'RST 48'
0038_H	FF	'RST 56'

Table A-16. Summary of the Z80 Assembler

ASSEMBLER FIELD STRUCTURE

The assembly language instructions have the standard field structure (see Table 2-1). The required delimiters are:

1) **A colon after a label, except for the pseudo-operations EQU, DEFL, and MACRO, which require a space.**
2) **A space after the operation code.**
3) **A comma between operands in the operand field. (Remember this one!)**
4) **A semicolon before a comment.**
5) **Parentheses around memory references.**

LABELS

The assembler allows six characters in labels; the first character must be a letter, while subsequent characters must be letters, numbers, ?, or the underbar character (_). We will use only capital letters or numbers, although some versions of the assembler allow lower-case letters and other symbols.

RESERVED NAMES

Some names are reserved as keywords and should not be used by the programmer. These are the register names (A, B, C, D, E, H, L, I, R), the double register names (IX, IY, SP), the register names (AF, BC, DE, HL, AF', BC', DE', HL'), and the states of the four testable flags (C, NC, Z, NZ, M, P, PE, PO).

PSEUDO-OPERATIONS

The assembler has the following basic pseudo-operations:

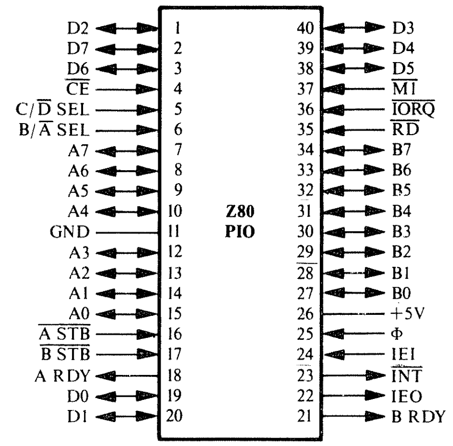
DEFB or DB	-	DEFINE BYTE
DEFL	-	DEFINE LABEL
DEFM	-	DEFINE STRING
DEFS or DS	-	DEFINE STORAGE
DEFW or DW	-	DEFINE WORD
END	-	END
EQU	-	EQUATE
ORG	-	ORIGIN

ADDRESSES

The Zilog Z80 assembler allows entries in the address field in any of the following forms

1) Decimal (the default case)
   Example: 1247
2) Hexadecimal (must start with a digit and end with an H)
   Examples: 142CH, 0E7H
3) Octal (must end with O or Q, but Q is far less confusing)
   Example: 1247Q or 1247O
4) Binary (must end with B)
   Example: 1001001000111B
5) ASCII (enclosed in single quotation marks)
   Example: 'HERE'
6) As an offset from the Program Counter ($)
   Example: $+237H

Appendix B Programming Reference for the Z80 PIO Device



Pin Name	Description	Type
D0-D7	Data Bus	Tristate, Bidirectional
$\overline{CE}$	Device Enable	Input
B/$\overline{A}$ SEL	Select Port A or Port B	Input
C/$\overline{D}$ SEL	Select Control or Data	Input
$\overline{M1}$	Instruction fetch machine cycle signal from CPU	Input
$\overline{IORQ}$	Input/Output request from CPU	Input
$\overline{RD}$	Read cycle status from CPU	Input
A0-A7	Port A Bus	Tristate, Bidirectional
A RDY	Register A Ready	Output
$\overline{A\ STB}$	Port A strobe pulse	Input
B0-B7	Port B Bus	Tristate, Bidirectional
B RDY	Register B Ready	Output
B STB	Port B strobe pulse	Input
IEI	Interrupt enable in	Input
IEO	Interrupt enable out	Output
$\overline{INT}$	Interrupt request	Output, Open-drain
Φ, +5V,GND	Clock, Power, and Ground	

Figure B-1. PIO pin assignments

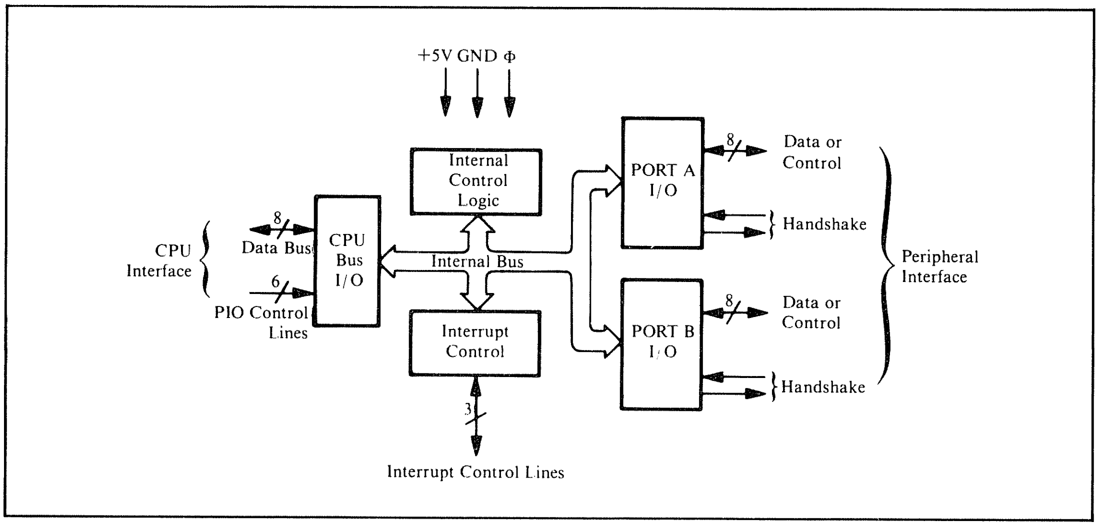


Figure B-2. Block diagram of the PIO

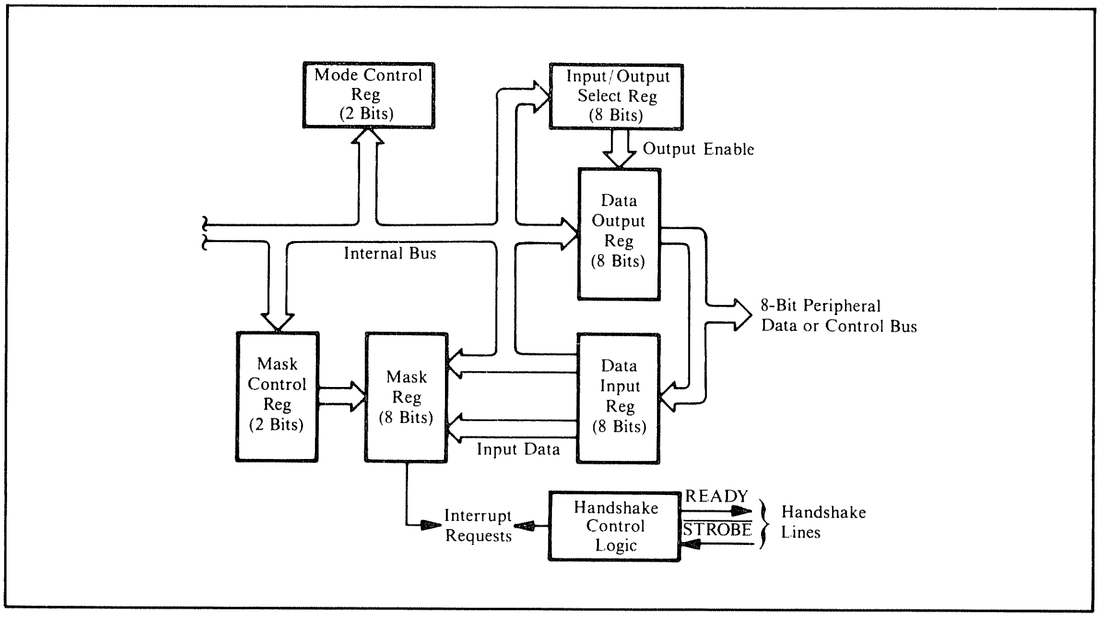


Figure B-3. Block diagram of PIO port I/O

Register Selection

Select Lines		Register Selected
C/D	B/A	
0	0	A Data
0	1	B Data
1	0	A Control
1	1	B Control

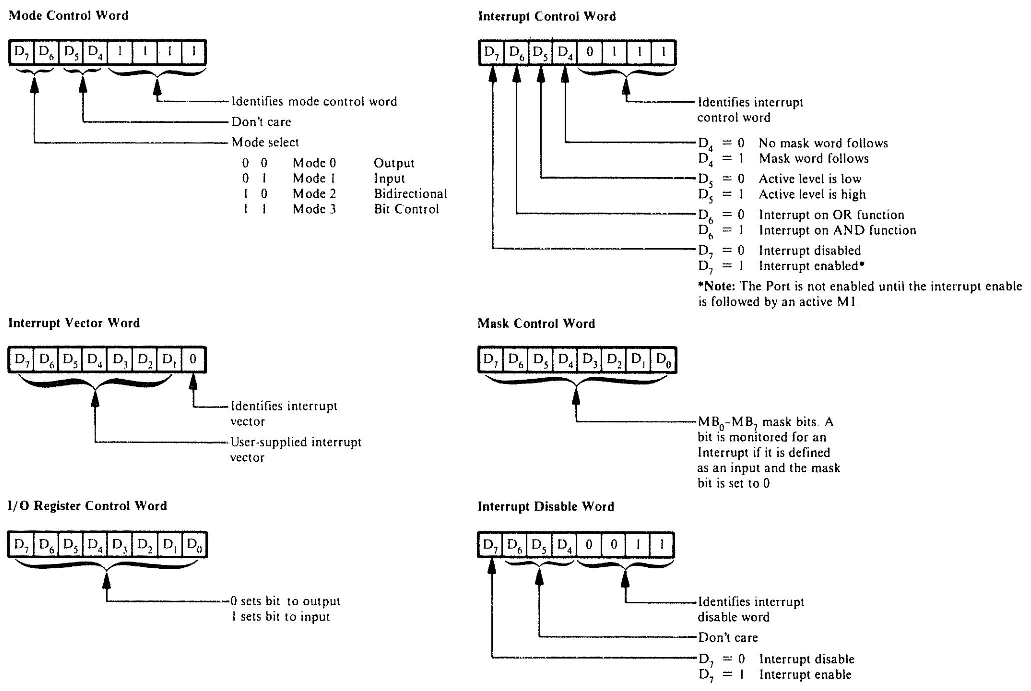

Mode Control Word
D7 D6 D5 D4 1 1 1 1
Identifies mode control word
Don't care
Mode select
0 0 Mode 0 Output
0 1 Mode 1 Input
1 0 Mode 2 Bidirectional
1 1 Mode 3 Bit Control
Interrupt Control Word
D7 D6 D5 D4 0 1 1 1
Identifies interrupt control word
D4 = 0 No mask word follows
D4 = 1 Mask word follows
D5 = 0 Active level is low
D5 = 1 Active level is high
D6 = 0 Interrupt on OR function
D6 = 1 Interrupt on AND function
D7 = 0 Interrupt disabled
D7 = 1 Interrupt enabled*
*Note: The Port is not enabled until the interrupt enable is followed by an active M1.
Interrupt Vector Word
D7 D6 D5 D4 D3 D2 D1 0
Identifies interrupt vector
User-supplied interrupt vector
Mask Control Word
D7 D6 D5 D4 D3 D2 D1 D0
MB0-MB7 mask bits. A bit is monitored for an Interrupt if it is defined as an input and the mask bit is set to 0
I/O Register Control Word
D7 D6 D5 D4 D3 D2 D1 D0
0 sets bit to output
1 sets bit to input
Interrupt Disable Word
D7 D6 D5 D4 0 0 1 1
Identifies interrupt disable word
Don't care
D7 = 0 Interrupt disable
D7 = 1 Interrupt enable

Figure B-4. Programming summary for the PIO

M1	M0	Mode
0	0	Output
0	1	Input
1	0	Bidirectional
1	1	Bit Control

M1	M0	X	X	1	1	1	1

When selecting Mode 3, the next byte must set the I/O Register:

I/O7	I/O6	I/O5	I/O4	I/O3	I/O2	I/O1	I/O0

I/O = 1 Sets bit to input
I/O = 0 Sets bit to output

PIO Mode	Meaning	Control Word	
		(Binary)	(Hex)
0	Output	00001111	0F
1	Input	01001111	4F
2	Bidirectional	10001111	8F
3	Control	11001111	CF

Note that bits 4 and 5 are not used and could have any values.

Figure B-5. Mode control for the PIO

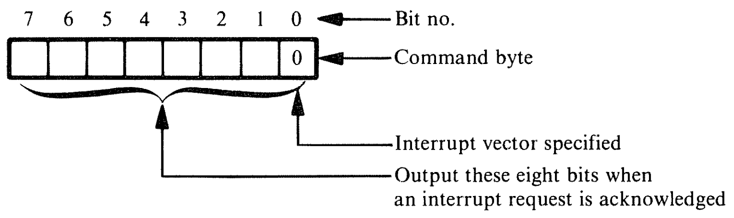

7 6 5 4 3 2 1 0
Bit no.
0
Command byte
Interrupt vector specified
Output these eight bits when an interrupt request is acknowledged

Figure B-6. Interrupt vector loading format for the PIO

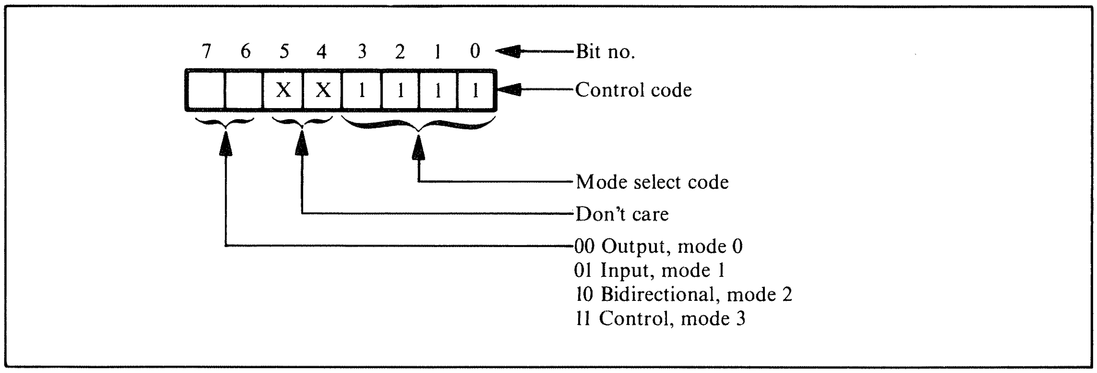


Figure B-7. Mode selection format for the PIO

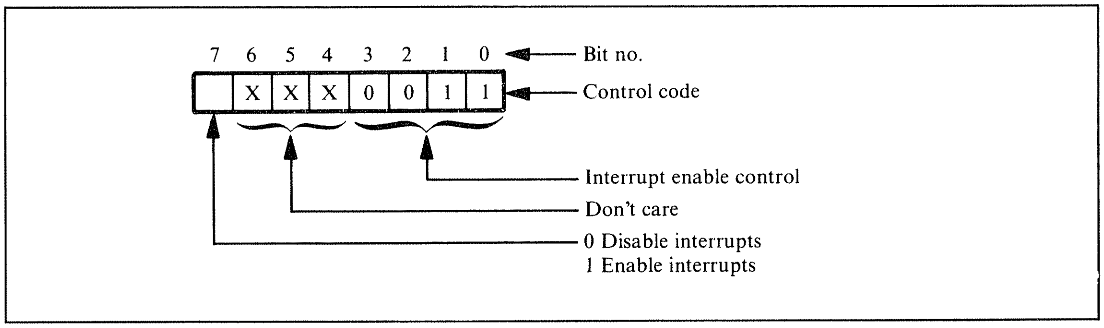


Figure B-8. Interrupt enable/disable format for the PIO

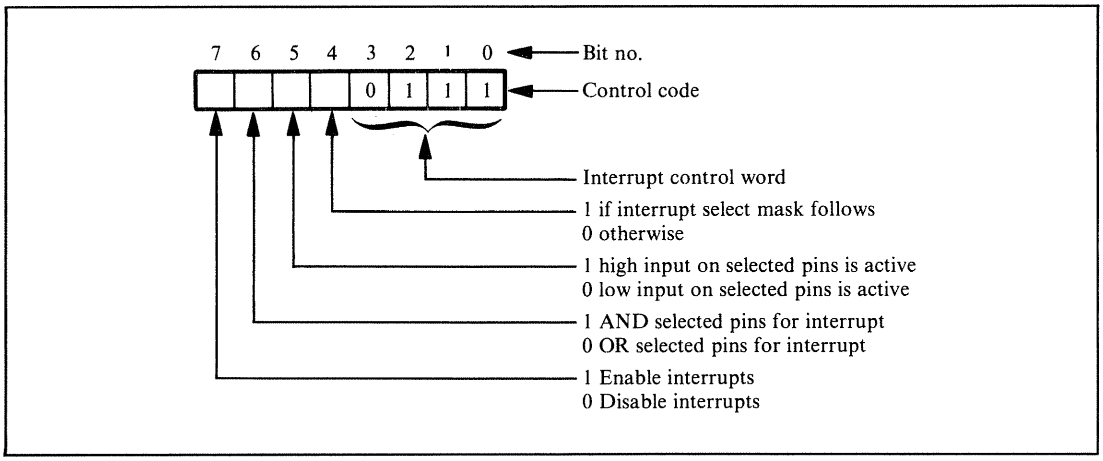

7 6 5 4 3 2 1 0
Bit no.
0 1 1 1
Control code
Interrupt control word
1 if interrupt select mask follows
0 otherwise
1 high input on selected pins is active
0 low input on selected pins is active
1 AND selected pins for interrupt
0 OR selected pins for interrupt
1 Enable interrupts
0 Disable interrupts

Figure B-9. Interrupt condition-setting format for the PIO (mode 3 only)

Table B-1. PIO Select Logic

Signal			Selected Location
$\overline{CE}$	$B/\overline{A}$ SEL	$C/\overline{D}$ SEL	
0	0	0	Port A data buffer
0	0	1	Port A control buffer
0	1	0	Port B data buffer
0	1	1	Port B control buffer
1	X	X	Device not selected

Table B-2. Addressing of PIO Control Registers

Register	Addressing
Mode control	$D_3 = D_2 = D_1 = D_0 = 1$
Input/Output control	Next byte after mode control sets mode 3
Mask control register	$D_3 = 0$, $D_2 = D_1 = D_0 = 1$
Interrupt mask register	Next byte after mask control register accessed with $D_4 = 1$
Interrupt enable	$D_3 = D_2 = 0$, $D_1 = D_0 = 1$
Interrupt vector	$D_0 = 1$

Appendix C ASCII Character Set

MSD / LSD		0 000	1 001	2 010	3 011	4 100	5 101	6 110	7 111
0	0000	NUL	DLE	SP	0	@	P	‘	p
1	0001	SOH	DC1	!	1	A	Q	a	q
2	0010	STX	DC2	″	2	B	R	b	r
3	0011	ETX	DC3	#	3	C	S	c	s
4	0100	EOT	DC4	$	4	D	T	d	t
5	0101	ENQ	NAK	%	5	E	U	e	u
6	0110	ACK	SYN	&	6	F	V	f	v
7	0111	BEL	ETB	′	7	G	W	g	w
8	1000	BS	CAN	(	8	H	X	h	x
9	1001	HT	EM	)	9	I	Y	i	y
A	1010	LF	SUB	*	:	J	Z	j	z
B	1011	VT	ESC	+	;	K	[	k	}
C	1100	FF	FS	,	<	L	\	l	|
D	1101	CR	GS	—	=	M	]	m	{
E	1110	SO	RS	•	>	N	^	n	~
F	1111	SI	US	/	?	O	_	o	DEL

REPRINTED BY PERMISSION SYNERTEK, INC.

Glossary

A

Absolute address. An address that identifies a storage location or an I/O device without the use of a base, offset, or other factor. *See also* Effective address, Relative offset.

Absolute addressing. An addressing mode in which the instruction contains the actual address required for its execution, as opposed to modes in which the instruction contains a relative offset or identifies a base register.

Accumulator. A register that is the implied source of one operand and the destination of the result in most arithmetic and logical operations.

Active transition. The edge on a strobe line that sets an indicator. The alternatives are a negative edge (1 to 0 transition) or a positive edge (0 to 1 transition).

Address. The identification code that distinguishes one memory location or I/O port from another and that can be used to select a specific one.

Addressing mode. The method for specifying the addresses to be used in executing an instruction. Common addressing modes are direct, immediate, indexed, indirect, and relative.

Address register. A register that contains a memory address.

Address space. The total range of addresses to which a particular computer may refer.

ALU. See Arithmetic-logic unit.

Arithmetic-logic unit (ALU). A device that can perform a variety of arithmetic and logical functions; function inputs select which one the device performs during a particular cycle.

Arithmetic shift. A shift operation that keeps the sign (most significant) bit the same. In a right shift, this results in copies of the sign bit moving right (called *sign extension*).

Arm. Usually refers specifically to interrupts. *See* Enable.

Array. A collection of related data items, usually stored in consecutive memory addresses.

ASCII (American Standard Code for Information Interchange). A 7-bit character code widely used in computers and communications.

Assembler. A computer program that converts assembly language programs into a form (machine language) that the computer can execute directly. The assembler translates mnemonic operation codes and names into their numerical equivalents and assigns locations in memory to data and instructions.

Assembly language. A computer language in which the programmer can use mnemonic operation codes, labels, and names to refer to their numerical equivalents.

Asynchronous. Operating without reference to an overall timing source, that is, at irregular intervals.

Autodecrementing. The automatic decrementing of an address register as part of the execution of an instruction that uses it.

Autoincrementing. The automatic incrementing of an address register as part of the execution of an instruction that uses it.

Automatic mode (of a peripheral chip). An operating mode in which the peripheral chip produces control signals automatically without specific program intervention.

B

Base address. The address in memory at which an array or table starts. Also called *starting address* or *base.*

Baud. A measure of the rate at which serial data is transmitted; bits per second, including both data bits and bits used for synchronization, error checking, and other purposes. Common baud rates are 110, 300, 1200, 2400, 4800, 9600, and 19,200.

Baud rate generator. A device that generates the proper time intervals between bits for serial data transmission.

BCD (Binary-Coded Decimal). A representation of decimal numbers in which each decimal digit is coded separately into a binary number.

Bidirectional. Capable of transporting signals in either direction.

Binary-coded decimal. See BCD.

Binary search. A search method that divides the set of items to be searched into two equal (or nearly equal) parts in each iteration. The part containing the item being sought is determined and then used as the set in the next iteration. Each iteration of a binary search thus halves the size of the set being searched. This method obviously assumes an ordered set of items.

BIOS (Basic Input/Output System). The part of CP/M that allows the operating system to use the I/O devices for a particular computer. The computer manufacturer or dealer typically supplies the BIOS; Digital Research, the originator of CP/M, provides only a sample BIOS with comments.

Bit test. An operation that determines whether a bit is 0 or 1. Usually refers to a logical AND operation with an appropriate mask.

Block. An entire group or section, such as a set of registers or a section of memory.

Block comparison (or *block compare*). A search that extends through a block of memory until either the item being sought is found or the entire block is examined.

Block move. Moving an entire set of data from one area of memory to another.

Block search. See Block comparison.

Boolean variable. A variable that has only two possible values, which may be represented as true and false or as 1 and 0. *See also* Flag.

Borrow. A bit that is set to 1 if a subtraction produces a negative result and to 0 if it produces a positive or zero result. The borrow is commonly used to subtract numbers that are too long to be handled in a single operation.

Bounce. Move back and forth between states before reaching a final state. Usually refers to mechanical switches that do not open or close cleanly, but rather move back and forth between positions for a while before settling down.

Branch instruction. See Jump instruction.

Breakpoint. A condition specified by the user under which program execution is to end temporarily, used as an aid in debugging programs. The specification of the conditions under which execution will end is referred to as *setting breakpoints,* and the deactivation of those conditions is referred to as *clearing breakpoints.*

BSC (Binary Synchronous Communications or Bisync). An older line protocol often used by IBM computers and terminals.

Bubble sort. A sorting technique that works through the elements of an array consecutively, exchanging an element with its successor if they are out of order.

Buffer. Temporary storage area generally used to hold data before they are transferred to their final destinations.

Buffer empty. A signal that is active when all data entered into a buffer or register have been transferred to their final destinations.

Buffer full. A signal that is active when a buffer or register is completely occupied with data that have not been transferred to their final destinations.

Buffer index. The index of the next available address in a buffer.

Buffer pointer. A storage location that contains the next available address in a buffer.

Bug. An error or flaw in a program.

Byte. A unit of eight bits. May be described as consisting of a high nibble or digit (the four most significant bits) and a low nibble or digit (the four least significant bits).

Byte-length. A length of eight bits per item.

C

Call (a subroutine). Transfer control to a subroutine while retaining the information required to resume the current program. A call differs from a jump or branch in that a call remembers the previous position in the program, whereas a jump or branch does not.

Carry. A bit that is 1 if an addition overflows into the succeeding digit position.

Carry flag. A flag that is 1 if the last operation generated a carry from the most significant bit and 0 if it did not.

CASE statement. A statement in a high-level computer language that directs the computer to perform one of several subprograms, depending on the value of a variable. That is, the computer performs the first subprogram if the variable has the first value specified, and so on. The computed GO TO statement serves a similar function in FORTRAN.

Central processing unit (CPU). The control section of the computer; the part that controls its operations, fetches and executes instructions, and performs arithmetic and logical functions.

Checksum. A logical sum that is included in a block of data to guard against recording or transmission errors. Also referred to as *longitudinal parity* or *longitudinal redundancy check* (LRC).

Circular shift. See Rotate.

Cleaning the stack. Removing unwanted items from the stack, usually by adjusting the stack pointer.

Clear. Set to zero.

Clock. A regular timing signal that governs transitions in a system.

Close (a file). To make a file inactive. The final contents of the file are the last information the user stored in it. The user must generally close a file after working with it.

Coding. Writing instructions in a computer language.

Combo chip. See Multifunction device.

Command register. See Control register.

Comment. A section of a program that has no function other than documentation. Comments are neither translated nor executed, but are simply copied into the program listing.

Complement. Invert; *see also* One's complement, Two's complement.

Concatenation. Linking together, chaining, or uniting in a series. In string operations, concatenation refers to the placing of one string after another.

Condition code. See Flag.

Control (or *command*) *register.* A register whose contents determine the state of a transfer or the operating mode of a device.

CP/M (Control Program/Microcomputer). A widely used disk operating system for Z80-based computers developed by Digital Research (Pacific Grove, CA).

CTC (Clock/Timer Circuit). A programmable timer chip in the Z80 family. A CTC contains four 8-bit timers, range controls (prescalers), and other circuits.

Cyclic redundancy check (CRC). An error-detecting code generated from a polynomial that can be added to a block of data or a storage area.

D

Data accepted. A signal that is active when the most recent data have been transferred successfully.

Data direction register. A register that determines whether bidirectional I/O lines are being used as inputs or outputs.

Data-link control. Conventions governing the format and timing of data exchange between communicating systems. Also called a *protocol.*

Data-link controller. A chip that performs all or most of the functions required by a data-link control. The SIO is a data-link controller in the Z80 family.

Data ready. A signal that is active when new data are available to the receiver. Same as *valid data.*

DDCMP (Digital Data Communications Message Protocol). A protocol that supports any method of physical data transfer (synchronous or asynchronous, serial or parallel).

Debounce. Convert the output from a contact with bounce into a single clean transition between states. Debouncing is most commonly applied to outputs from mechanical keys or switches that bounce back and forth before settling into their final positions.

Debounce time. The amount of time required to debounce a change of state.

Debugger. A systems program that helps users locate and correct errors in their programs. Some versions are referred to as dynamic debugging tools, or DDTs after the famous insecticide. Popular CP/M debuggers are SID (Symbolic Instruction Debugger) and ZSID (Z80 Symbolic Instruction Debugger) from Digital Research.

Debugging. Locating and correcting errors in a program.

Device address. The address of a port associated with an I/O device.

Diagnostic. A program that checks the operation of a device and reports its findings.

Digit shift. A shift of one BCD digit position or four bit positions.

Direct addressing. An addressing mode in which the instruction contains the address required for its execution. Note that the standard Z80 assembler requires parentheses around an address that is to be used directly.

Disable (or *disarm*). Prevent an activity from proceeding or a signal (such as an interrupt) from being recognized.

Disarm. Usually refers specifically to interrupts. *See* Disable.

Double word. When dealing with microprocessors, a unit of 32 bits.

Driver. See I/O driver.

Dump. A facility that displays the contents of an entire section of memory or group of registers on an output device.

Dynamic allocation (of memory). The allocation of memory for a subprogram from whatever is available when the subprogram is called. An alternative is *static allocation* of a fixed area of storage to each subprogram. Dynamic allocation often reduces overall memory usage because subprograms can share areas; it does, however, generally require additional execution time and overhead spent in memory management.

E

EBCDIC (Expanded Binary-Coded Decimal Interchange Code). An 8-bit character code often used in large computers.

Echo. Reflect transmitted information back to the transmitter; send back to a terminal the information received from it.

Editor. A program that manipulates text material and allows the user to make corrections, additions, deletions, and other changes. A popular CP/M editor is ED from Digital Research.

Effective address. The actual address used by an instruction to fetch or store data.

EIA RS-232. See RS-232.

Enable (or *arm*). Allow an activity to proceed or a signal (such as an interrupt) to be recognized.

Endless loop or *jump-to-self instruction.* An instruction that transfers control to itself, thus executing indefinitely (or until a hardware signal interrupts it).

Error-correcting code. A code that the receiver can use to correct errors in messages; the code itself does not contain any additional message.

Error-detecting code. A code that the receiver can use to detect errors in messages; the code itself does not contain any additional message.

Even parity. A 1-bit error-detecting code that makes the total number of 1 bits in a unit of data (including the parity bit) even.

EXCLUSIVE OR function. A logical function that is true if either, but not both, of its inputs is true. It is thus true if its inputs are not equal (that is, if one of them is a logic 1 and the other is a logic 0).

Extend (a number). Add digits to a number to conform to a format without changing its value. For example, one may extend an 8-bit unsigned result with zeros to fill a 16-bit word.

External reference. The use in a program of a name that is defined in another program.

F

F register. See Flag register.

Field. A set of one or more positions within a larger unit, such as a byte, word, or record.

File. A collection of related information that is treated as a unit for purposes of storage or retrieval.

Fill. Placing values in storage areas not previously in use, initializing memory or storage.

Flag (or *condition code* or *status bit*). A single bit that indicates a condition within the computer, often used to choose between alternative instruction sequences.

Flag (software). An indicator that is either on or off and can be used to select between two alternative courses of action. *Boolean variable* and *semaphore* are other terms with the same meaning.

Flag register. A Z80 register that holds all the flags. Also called the *(processor) status register.*

Free-running mode. An operating mode for a timer in which it indicates the end of a time interval and then starts another of the same length. Also called a *continuous mode.*

Function key. A key that causes a system to execute a procedure or perform a function (such as clearing the screen of a video terminal).

G

Global variable. A variable that is used in more than one section of a computer program rather than only locally.

H

Handshake. An asynchronous transfer in which sender and receiver exchange signals to establish synchronization and to indicate the status of the data transfer. Typically, the sender indicates that new data are available and the receiver reads the data and indicates that it is ready for more.

Hardware stack. A stack that the computer manages automatically when executing instructions that use it.

Head (of a queue). The location of the item most recently entered into a queue.

Header, queue. See Queue header.

Hexadecimal (or *hex*). Number system with base 16. The digits are the decimal numbers 0 through 9, followed by the letters A through F (representing 10 through 15 decimal).

Hex code. See Object code.

High-level language. A programming language that is aimed toward the solution of problems, rather than being designed for convenient conversion into computer instructions. A compiler or interpreter translates a program written in a high-level language into a form that the computer can execute. Common high-level languages include Ada, BASIC, C, COBOL, FORTRAN, and Pascal.

I

Immediate addressing. An addressing mode in which the data required by an instruction are part of the instruction. The data immediately follow the operation code in memory.

Index. A data item used to identify a particular element of an array or table.

Indexed addressing. An addressing mode in which the address is modified by the contents of an index register to determine the effective address (the actual address used).

Indexed indirect addressing. See Preindexing.

Index register. A register that can be used to modify memory addresses.

Indirect addressing. An addressing mode in which the effective address is the contents of the address included in the instruction, rather than the address itself.

Indirect indexed addressing. See Postindexing.

Indirect jump. A jump instruction that transfers control to the address stored in a register or memory location rather than to a fixed address.

Input/output control block (IOCB). A group of storage locations that contains the information required to control the operation of an I/O device. Typically included in the information are the addresses of routines that perform operations such as transferring a single unit of data or determining device status.

Input/output control system (IOCS). A set of computer routines that controls the performance of I/O operations.

Instruction. A group of bits that defines a computer operation and is part of the instruction set.

Instruction cycle. The process of fetching, decoding, and executing an instruction.

Instruction execution time. The time required to fetch, decode, and execute an instruction.

Instruction fetch. The process of addressing memory and reading an instruction into the CPU for decoding and execution.

Instruction length. The amount of memory needed to store a complete instruction.

Instruction set. The set of general-purpose instructions available on a given computer; the set of inputs to which the CPU will produce a known response when they are fetched, decoded, and executed.

Interpolation. Estimating values of a function at points between those at which the values are already known.

Interrupt. A signal that temporarily suspends the computer's normal sequence of operations and transfers control to a special routine.

Interrupt-driven. Dependent on interrupts for its operation; may idle until it receives an interrupt.

Interrupt flag. A bit in the input/output section that is set when an event occurs that requires servicing by the CPU. Typical events include an active transition on a control line and the exhaustion of a count by a timer.

Interrupt mask (or *interrupt enable*). A bit that determines whether interrupts will be recognized. A mask or disable bit must be cleared to allow interrupts, whereas an enable bit must be set.

Interrupt request. A signal that is active when a peripheral is requesting service, often used to cause a CPU interrupt. *See also* Interrupt flag.

Interrupt service routine. A program that performs the actions required to respond to an interrupt.

Interrupt vector. An address to which an interrupt directs the computer, usually the starting address of a service routine.

Inverted borrow. A bit that is set to 0 if a subtraction produces a negative result and to 1 if it produces a positive or 0 result. An inverted borrow can be used like a true borrow, except that the complement of its value (i.e., 1 minus its value) must be used in the extension to longer numbers.

IOCB. See Input/output control block.

IOCS. See Input/output control system.

I/O device table. A table that establishes the correspondence between the logical devices to which programs refer and the physical devices that are actually used in data transfers. An I/O device table must be placed in memory in order to run a program that refers to logical devices on a computer with a particular set of actual (physical) devices. The I/O device table may, for example, contain the starting addresses of the I/O drivers that handle the various devices.

I/O driver. A computer program that transfers data to or from an I/O device, also called a *driver* or *I/O utility.* The driver must perform initialization functions and handle status and control, as well as physically transfer the actual data.

Isolated input/output. An addressing method for I/O ports that uses a decoding system distinct from that used by the memory section. I/O ports thus do not occupy memory addresses.

J

Jump instruction (or *branch instruction*). An instruction that places a new value in the program counter, thus departing from the normal one-step incrementing. Jump instructions may be conditional; that is, the new value may be placed in the program counter only if a condition holds.

Jump table. A table consisting of the starting addresses of executable routines, used to transfer control to one of them.

L

Label. A name attached to an instruction or statement in a program that identifies the location in memory of the machine language code or assignment produced from that instruction or statement.

Latch. A device that retains its contents until new data are specifically entered into it.

Leading edge (of a binary pulse). The edge that marks the beginning of a pulse.

Least significant bit. The rightmost bit in a group of bits, that is, bit 0 of a byte or a 16-bit word.

Library program. A program that is part of a collection of programs and is written and documented according to a standard format.

LIFO (last-in, first-out) memory. A memory that is organized according to the order in which elements are entered and from which elements can be retrieved only in the order opposite of that in which they were entered. *See also* Stack.

Linearization. The mathematical approximation of a function by a straight line between two points at which its values are known.

Linked list. A list in which each item contains a pointer (or *link*) to the next item. Also called a *chain* or *chained list.*

List. An ordered set of items.

Logical device. The input or output device to which a program refers. The actual or physical device is determined by looking up the logical device in an I/O device table —a table containing actual I/O addresses (or starting addresses for I/O drivers) corresponding to the logical device numbers.

Logical shift. A shift operation that moves zeros in at either end as the original data are shifted.

Logical sum. A binary sum with no carries between bit positions. *See also* Checksum, EXCLUSIVE OR function.

Longitudinal parity. See Checksum.

Longitudinal redundancy check (LRC). *See* Checksum.

Lookup table. An array of data organized so that the answer to a problem may be determined merely by selecting the correct entry (without any calculations).

Low-level language. A computer language in which each statement is translated directly into a single machine language instruction.

M

Machine language. The programming language that the computer can execute directly with no translation other than numeric conversions.

Maintenance (of programs). Updating and correcting computer programs that are in use.

Majority logic. A combinational logic function that is true when more than half the inputs are true.

Mark. The 1 state on a serial data communications line.

Mask. A bit pattern that isolates one or more bits from a group of bits.

Maskable interrupt. An interrupt that the system can disable.

Memory capacity. The total number of different memory addresses (usually specified in bytes) that can be attached to a particular computer.

Memory-mapped I/O. An addressing method for I/O ports that uses the same decoding system used by the memory section. The I/O ports thus occupy memory addresses.

Microcomputer. A computer that has a microprocessor as its central processing unit.

Microprocessor. A complete central processing unit for a computer constructed from one or a few integrated circuits.

Mnemonic. A memory jogger, a name that suggests the actual meaning or purpose of the object to which it refers.

Modem (Modulator/demodulator). A device that adds or removes a carrier frequency, thereby allowing data to be transmitted on a high-frequency channel or received from such a channel.

Modular programming. A programming method whereby the overall program is divided into logically separate sections or *modules.*

Module. A part or section of a program.

Monitor. A program that allows the computer user to enter programs and data, run programs, examine the contents of the computer's memory and registers, and utilize the computer's peripherals. *See also* Operating system.

Most significant bit. The leftmost bit in a group of bits, that is, bit 7 of a byte or bit 15 of a 16-bit word.

Multifunction device. A device that performs more than one function in a computer system; the term commonly refers to devices containing memory, input/output ports, timers, and so forth.

Multitasking. Executing many tasks during a single period of time, usually by working on each one for a specified part of the period and suspending tasks that must wait for input, output, the completion of other tasks, or external events.

Murphy's Law. The famous maxim that "whatever can go wrong, will."

N

Negate. Find the two's complement (negative) of a number.

Negative edge (of a binary pulse). A 1-to-0 transition.

Negative flag. See Sign flag.

Negative logic. Circuitry in which a logic zero is the active or ON state.

Nesting. Constructing programs in a hierarchical manner with one level contained within another. The nesting level is the number of transfers of control required to reach a particular part of a program without ever returning to a higher level.

Nibble. A unit of four bits. A byte (eight bits) may be described as consisting of a high nibble (four most significant bits) and a low nibble (four least significant bits).

Nine's complement. The result of subtracting a decimal number from a number having nines in all digit positions.

Non-maskable interrupt. An interrupt that cannot be disabled within the CPU.

Non-volatile memory. A memory that retains its contents when power is removed.

Nop (or *no operation*). An instruction that does nothing except increment the program counter.

Normalization (of numbers). Adjusting a number into a regular or standard format. A typical example is the scaling of a binary fraction to make its most significant bit 1.

O

Object code (or *object program*). The program that is the output of a translator program, such as an assembler—usually a machine language program ready for execution.

Odd parity. A 1-bit error-detecting code that makes the total number of 1 bits in a unit of data (including the parity bit) odd.

One's complement. A bit-by-bit logical complement of a number, obtained by replacing each 0 bit with a 1 and each 1 bit with a 0.

One-shot. A device that produces a pulse output of known duration in response to a pulse input. A timer operates in a *one-shot mode* when it indicates the end of a single interval of known duration.

Open (a file). Make a file ready for use. The user generally must open a file before working with it.

Operating system (OS). A computer program that controls the overall operations of a computer and performs such functions as assigning places in memory to programs and data, scheduling the execution of programs, processing interrupts, and controlling the overall input/output system. Also known as a *monitor, executive,* or *master-control program,* although the term *monitor* is usually reserved for a simple operating system with limited functions.

Operation code (op code). The part of an instruction that specifies the operation to be performed.

OS. See Operating system.

Overflow (of a stack). Exceeding the amount of memory allocated to a stack.

Overflow, two's complement. See Two's complement overflow.

P

Packed decimal. A binary-coded decimal format in which each byte contains two decimal digits.

Page. A subdivision of the memory. In byte-oriented computers, a page is generally a 256-byte section of memory in which all addresses have the same eight most significant bits (or *page number*). For example, page C6 would consist of memory addresses C600 through C6FF.

Paged address. The identifier that characterizes a particular memory address on a known page. In byte-oriented computers, this is usually the eight least significant bits of a memory address.

Page number. The identifier that characterizes a particular page of memory. In byte-oriented computers, this is usually the eight most significant bits of a memory address.

Parallel interface. An interface between a CPU and input or output devices that handle data in parallel (more than one bit at a time). The PIO is a parallel interface in the Z80 family.

Parameter. An item that must be provided to a subroutine or program for it to be executed.

Parity. A 1-bit error-detecting code that makes the total number of 1 bits in a unit of data, including the parity bit, odd (odd parity) or even (even parity). Also called *vertical parity* or *vertical redundancy check* (VRC).

Passing parameters. Making the required parameters available to a subroutine.

Peripheral ready. A signal that is active when a peripheral can accept more data.

Physical device. An actual input or output device, as opposed to a logical device.

PIO (Parallel Input/Output Device). A parallel interface chip in the Z80 family. A PIO contains two 8-bit I/O ports, four control lines, and other circuitry.

Pointer. A storage place that contains the address of a data item rather than the item itself. A pointer tells where the item is located.

Polling. Determining which I/O devices are ready by examining the status of one device at a time.

Polling interrupt system. An interrupt system in which a program determines the source of a particular interrupt by examining the status of potential sources one at a time.

Pop. Remove an operand from a stack.

Port. The basic addressable unit of the computer's input/output section.

Positive edge (of a binary pulse). A 0-to-1 transition.

Postdecrementing. Decrementing an address register after using it.

Postincrementing. Incrementing an address register after using it.

Postindexing. An addressing mode in which the effective address is determined by first obtaining the base address indirectly and then indexing from that base address. The "post" refers to the fact that the indexing is performed after the indirection.

Power fail interrupt. An interrupt that informs the CPU of an impending loss of power.

Predecrementing. Decrementing an address register before using it.

Preincrementing. Incrementing an address register before using it.

Preindexing. An addressing mode in which the effective address is determined by indexing from the base address and then using the indexed address indirectly. The "pre" refers to the fact that the indexing is performed before the indirection. Of course, the array starting at the given base address must consist of addresses that can be used indirectly.

Priority interrupt system. An interrupt system in which some interrupts have precedence over others; that is, they will be serviced first or can interrupt the others' service routines.

Program counter (PC register). A register that contains the address of the next instruction to be fetched from memory.

Programmable I/O device. An I/O device that can have its mode of operation determined by loading registers under program control.

Programmable peripheral chip (or *programmable peripheral interface*). A chip that can operate in a variety of modes; its current operating mode is determined by loading control registers under program control.

Programmable timer. A device that can handle a variety of timing tasks, including the generation of delays, under program control. The CTC is a programmable timer in the Z80 family.

Programmed input/output. Input or output performed under program control without using interrupts or other special hardware techniques.

Program relative addressing. A form of relative addressing in which the base address is the program counter. Use of this form of addressing makes it easy to move programs from one place in memory to another.

Protocol. See Data-link control.

Pseudo-operation (or *pseudo-op* or *pseudo-instruction*). An assembly language operation code that directs the assembler to perform some action but does not result in the generation of a machine language instruction.

Pull. Remove an operand from a stack; same as *pop.*

Push. Store an operand in a stack.

Q

Queue. A set of tasks, storage addresses, or other items that are used in a first-in, first-out manner; that is, the first item entered into the queue is the first to be used or removed.

Queue header. A set of storage locations describing the current location and status of a queue.

R

RAM. See Random-access memory.

Random-access (read/write) memory (RAM). A memory that can be both read and altered (written) in normal operation.

Read-only memory (ROM). A memory that can be read but not altered in normal operation.

Ready for data. A signal that is active when the receiver can accept more data.

Real-time. In synchronization with the actual occurrence of events.

Real-time clock. A device that interrupts a CPU at regular time intervals.

Real-time operating system. An operating system that can act as a supervisor for programs that have real-time requirements. May also be referred to as a *real-time executive* or *real-time monitor.*

Reentrant. A program or routine that can be executed concurrently while the same routine is being interrupted or otherwise held in abeyance.

Refresh. Rewriting data into a memory before its contents are lost. Dynamic RAM must be refreshed periodically (typically every few milliseconds) or it will lose its contents spontaneously.

Register. A storage location inside the CPU.

Register pair. In Z80 terminology, two 8-bit registers that can be referenced as a 16-bit unit.

Relative addressing. An addressing mode in which the address specified in the instruction is the offset from a base address.

Relative offset. The difference between the actual address to be used in an instruction and the current value of the program counter.

Relocatable. Can be placed anywhere in memory without changes; that is, a program that can occupy any set of consecutive memory addresses.

Return (from a subroutine). Transfer control back to the program that originally called the subroutine and resume its execution.

ROM. See Read-only memory.

Rotate. A shift operation that treats the data as if they were arranged in a circle; that is, as if the most significant and least significant bits were connected either directly or through a Carry bit.

Row major order. Storing elements of a multidimensional array in memory by changing the indexes starting with the rightmost first. For example, if a typical element is A(I,J,K) and the elements begin with A(0,0,0), the order is A(0,0,0), A(0,0,1),...,A(0,1,0), A(0,1,1),.... The opposite technique (changing the leftmost index first) is called *column major order.*

RS-232 (or *EIA RS-232*). A standard interface for the transmission of digital data, sponsored by the Electronic Industries Association of Washington, D.C. It has been partially superseded by RS-449.

S

Scheduler. A program that determines when other programs should be started and terminated.

Scratchpad. An area of memory that is generally easy and quick to use for storing variable data or intermediate results.

SDLC. (Synchronous Data Link Control). The successor protocol to BSC for IBM computers and terminals.

Semaphore. See Flag.

Serial. One bit at a time.

Serial interface. An interface between a CPU and input or output devices that handle data serially. The SIO is a popular serial interface chip in the Z80 family. *See also* UART.

Setpoint. The value of a variable that a controller is expected to maintain.

Shift instruction. An instruction that moves all the bits of the data by a certain number of bit positions, just as in a shift register.

Signed number. A number in which one or more bits represent whether the number is positive or negative. A common format is for the most significant bit to represent the sign (0 = positive, 1 = negative).

Sign extension. The process of copying the sign (most significant) bit to the right as in an arithmetic shift. Sign extension preserves the sign when two's complement numbers are being divided or normalized.

Sign flag. A flag that contains the most significant bit of the result of the previous operation. It is sometimes called a *negative flag,* since a value of 1 indicates a negative signed number.

Sign function. A function that is 0 if its parameter is positive and 1 if its parameter is negative.

SIO (Serial Input/Output Device). A serial interface chip in the Z80 family. An SIO can be used as an asynchronous or synchronous serial interface (i.e., as a UART or USRT) or as a data-link controller.

Size (of an array dimension). The distance in memory between elements that are ordered consecutively in a particular dimension; the number of bytes between the starting address of an element and the starting address of the element with an index one larger in a particular dimension but the same in all other dimensions.

Software delay. A program that has no function other than to waste time.

Software interrupt. See Trap.

Software stack. A stack that is managed by means of specific instructions, as opposed to a hardware stack which the computer manages automatically.

Source code (or *source program*). A computer program written in assembly language or in a high-level language.

Space. The zero state on a serial data communications line.

Stack. A section of memory that can be accessed only in a last-in, first-out manner. That is, data can be added to or removed from the stack only through its top; new data are placed above the old data and the removal of a data item makes the item below it the new top.

Stack pointer. A register that contains the address of the top of a stack.

Standard (or *8,4,2,1*) *BCD.* A BCD representation in which the bit positions have the same weight as in ordinary binary numbers.

Standard teletypewriter. A teletypewriter that operates asynchronously at a rate of ten characters per second.

Start bit. A 1-bit signal that indicates the start of data transmission by an asynchronous device.

Static allocation (of memory). Assignment of fixed storage areas for data and programs; an alternative is *dynamic allocation,* in which storage areas are assigned when they are needed.

Status register. A register whose contents indicate the current state or operating mode of a device.

Status signal. A signal that describes the current state of a transfer or the operating mode of a device.

Stop bit. A 1-bit signal that indicates the end of data transmission by an asynchronous device.

String. An array (set of data) consisting of characters.

String functions. Procedures that allow the programmer to operate on data consisting of characters rather than numbers. Typical functions are insertion, deletion, concatenation, search, and replacement.

Strobe. A signal that identifies or describes another set of signals and can be used to control a buffer, latch, or register.

Subroutine. A subprogram that can be executed (called) from more than one place in a main program.

Subroutine call. The process whereby a computer transfers control from its current program to a subroutine while retaining the information required to resume the current program.

Subroutine linkage. The mechanism whereby a computer retains the information required to resume its current program after it completes the execution of a subroutine.

Suspend (a task). Halt execution and preserve the status of a task until some future time.

Synchronization (or *sync*) *character.* A character that is used only to synchronize the transmitter and the receiver.

Synchronous. Operating according to an overall timing source or clock, that is, at regular intervals.

Systems software. Programs that perform administrative functions or aid in the development of other programs but do not actually perform any of the computer's workload.

T

Tail (of a queue). The location of the oldest item in the queue, that is, the earliest entry.

Task. A self-contained program that can serve as part of an overall system under the control of a supervisor.

Task status. The set of parameters that specifies the current state of a task. A task can be suspended and resumed as long as its status is saved and restored.

Teletypewriter. A device containing a keyboard and a serial printer that is often used in communications and with computers. Also referred to as a Teletype (a registered trademark of Teletype Corporation of Skokie, Illinois) or TTY.

Ten's complement. The result of subtracting a decimal number from zero (ignoring the minus sign); the nine's complement plus one.

Terminator. A data item that has no function other than to signify the end of an array.

Threaded code. A program consisting of subroutines, each of which automatically transfers control to the next one upon its completion.

Timeout. A period during which no activity is allowed to proceed; an inactive period.

Top of the stack. The address containing the item most recently entered into the stack.

Trace. A debugging aid that provides information about a program while the program is being executed. The trace usually prints all or some of the intermediate results.

Trailing edge (of a binary pulse). The edge that marks the end of a pulse.

Translate instruction. An instruction that converts its operand into the corresponding entry in a table.

Transparent routine. A routine that operates without interfering with the operations of other routines.

Trap (or *software interrupt*). An instruction that forces a jump to a specific (CPU-dependent) address, often used to produce breakpoints or to indicate hardware or software errors.

True borrow. See Borrow.

True comparison. A comparison that finds the two operands to be equal.

Two's complement. A binary number that, when added to the original number in a binary adder, produces a zero result. The two's complement of a number may be obtained by subtracting the number from zero or by adding 1 to the one's complement.

Two's complement overflow. A situation in which a signed arithmetic operation produces a result that cannot be represented correctly; that is, the magnitude overflows into the sign bit.

U

UART (Universal Asynchronous Receiver/Transmitter). An LSI device that acts as an interface between systems that handle data in parallel and devices that handle data in asynchronous serial form.

Underflow (of a stack). Attempting to remove more data from a stack than has been entered into it.

Unsigned number. A number in which all the bits are used to represent magnitude.

USART (Universal Synchronous/Asynchronous Receiver/Transmitter). An LSI device (such as the SIO) that can serve as either a UART or a USRT.

USRT (Universal Synchronous Receiver/Transmitter). An LSI device that acts as an interface between systems that handle data in parallel and devices that handle data in synchronous serial form.

Utility. A general-purpose program, usually supplied by the computer manufacturer or part of an operating system, that executes a standard or common operation such as sorting, converting data from one format to another, or copying a file.

V

Valid data. A signal that is active when new data are available to the receiver.

Vectored interrupt. An interrupt that produces an identification code (or *vector*) that the CPU can use to transfer control to the appropriate service routine. The process whereby control is transferred to the service routine is called *vectoring.*

Volatile memory. A memory that loses its contents when power is removed.

W

Walking bit test. A procedure whereby a single 1 bit is moved through each bit position in an area of memory and a check is made as to whether it can be read back correctly.

Word. The basic grouping of bits that a computer can process at one time. When dealing with microprocessors, the term often refers to a 16-bit unit of data.

Word boundary. A boundary between 16-bit storage units containing two bytes of information. If information is being stored in word-length units, only pairs of bytes conforming to (aligned with) word boundaries contain valid information. Misaligned pairs of bytes contain one byte from one word and one byte from another.

Word-length. A length of 16 bits per item.

Wraparound. Organization in a circular manner as if the ends were connected. A storage area exhibits wraparound if operations on it act as if the boundary locations were contiguous.

Write-only register. A register that the CPU can change but cannot read. If a program must determine the contents of such a register, it must save a copy of the data placed there.

Z

Zero flag. A flag that is 1 if the last operation produced a result of zero and 0 if it did not.

Zoned decimal. A binary-coded decimal format in which each byte contains a single decimal digit.

Index

Z80®
Assembly Language Subroutines

by Lance A. Leventhal and Winthrop Saville

McGraw-Hill

ISBN 0-931988-91-8