Z8 PLZ/ASM

Zilog

Assembly Language Programming Manual

December 1980

Zilog

Z8 PLZ/ASM

Assembly Language Programming Manual

December 1980

Preface

This reference manual describes assembly language programming for Zilog's Z8 single-chip microcomputer. The first three sections of the manual focus on Z8 design features and the assembly-language instruction set. Sections 4 and 5 provide additional information needed to build a source program, including the use of high-level PLZ statements.

This manual is one in a series describing the Z8. You will need several other manuals to develop, debug, and run Z8 assembly-language programs. Programs are developed on either Zilog's microcomputer system (MCZ) or the Zilog development system (ZDS) using the software capabilities of the RIO operating system. The manuals needed to use the operating system are:

Z80 RIO Operating System User's Manual, 03-0072-01

Z80 RIO Text Editor User's Manual, 03-0074-00

The Z8 assembler produces relocatable object modules. Operation of the assembler and object module linkage and relocation are described in the:

Z8 PLZ/ASM Assembler User Guide, 03-3048-02

PLZ Linker User Guide, 03-3098-02

Finally, while this programming manual includes an overview of the Z8 architecture, you will need the following manual for detailed hardware and configuration information:

Z8 Microcomputer Technical Manual, 03-3047-02

Contents

CONTENTS (cont.)

LIST OF ILLUSTRATIONS

Section 1
Architectural Overview

1.1 Introduction

Zilog's Z8 microcomputer introduces a new generation of single-chip architecture. Compared to earlier single-chip microcomputers, the Z8 offers faster execution, more efficient use of memory, more sophisticated interrupt, input/output (I/O), and bit-manipulation capabilities, and easier system expansion. Under program control, the Z8 configuration can be tailored to the needs of its user. It can serve as an I/O-intensive microcomputer, as an intelligent peripheral controller within a larger system, or as a memory-intensive microprocessor.

The Z8's features include a powerful repertoire of 43 instructions, similar in form to the instruction sets of the Z80 and Z8000 microprocessor families. The efficiency of these instructions and of the Z8's internal register-addressing scheme not only speeds program execution, but also packs more program into the Z8 chip than would be possible with comparable microcomputers. This is, of course, extremely important for single-chip devices where on-chip memory space is limited.

Real-time control applications, for which the Z8 is particularly suited, require fast instruction execution and fast interrupt response. Operating from an 8 MHz clock source (internal 4 MHz clock rate), the Z8 executes most instructions in 1.5 to 2.5 microseconds (6 to 10 machine cycles). The longest instruction takes 5 microseconds (20 cycles).

The following summarizes the main features of the Z8:

- 40-pin package, offering more I/O program control than previously available in single-chip microcomputers;
- On-chip, 2K-byte, read-only (ROM) program memory with possible expansion by 62K of external program memory;
- On-chip, 144-byte, random-access (RAM) register memory, including 4 I/O ports and 16 control registers;
- Possible 62K bytes external data memory;
- Six maskable and prioritized interrupts;
- Two on-chip interval timers, also programmable as event counters;

- Independent on-chip UART with hardware parity generator and checker;

- On-chip clock for internal timing.

The remainder of this section describes in more detail those Z8 features of primary interest to assembly-language programmers. See the Z8 Technical Manual for detailed architectural and configuration information.

1.2 Memory Segments

As shown in Figure 1-1, the Z8 has three separate memory segments for storing program instructions and data.

- Program memory (chip resident or external)
- Data memory (external)
- Register memory (chip resident)

The latter includes I/O registers, control and status registers, and general purpose data registers.

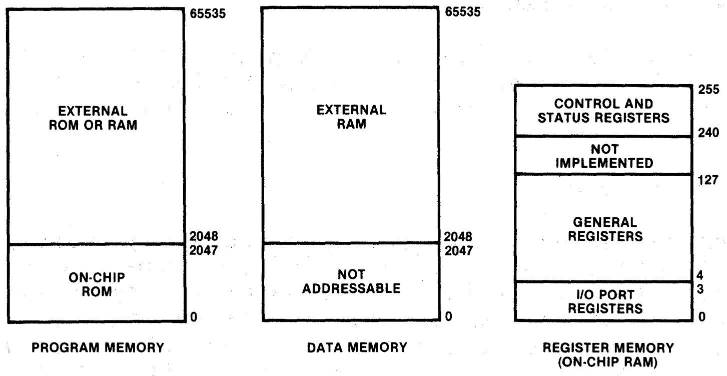


Figure 1-1. Z8 Memory Segments

The Z8 hardware environment must be specifically configured to access external program or data memory. Both segments can be accessed by 16-bit addresses.

1.2.1 Program Memory

The first 2048 bytes of program memory consist of on-chip programmable storage addressed by the program counter. For addresses 2048 or greater, the Z8 automatically executes external program memory fetches (provided the Z8 is configured accordingly). The first example below jumps to address 1500 if the Zero flag (Z) is set. The second calls a procedure whose starting address is location 20000 in external program memory.

```
JP Z,1500
CALL 20000
```

The first 12 bytes of program memory are reserved for the Z8's interrupt mechanism. Addresses 0-11 contain six 16-bit addresses corresponding to the six possible interrupts available on the Z8, IRQ0 through IRQ5, respectively. When an interrupt occurs, control passes to the address corresponding to that particular interrupt. A system reset forces the program counter to 12, the first address available for the user program. See the discussion of interrupts in Section 1.4.

1.2.2 External Data Memory

A Z8 system can directly access as much as 62K bytes of external data memory. This segment is addressed beginning with data address 2048. External I/O is also mapped into this segment.

1.2.3 Register Memory

Register memory includes 124 general-purpose registers, 4 I/O ports, and 16 status and control registers. The I/O port and control registers are included in register memory to allow any Z8 instruction to process I/O or control information directly, thus eliminating the need for special I/O or control instructions. The Z8 instruction set permits direct access to any of these 144 registers. Each of the 124 general-purpose registers can function as an accumulator, an address pointer, or an index register.

Z8 instructions can access registers directly or indirectly using an 8-bit address field. The Z8 also allows 4-bit addressing of registers, which generally saves bytes, and speeds program execution and task switching. In this 4-bit addressing mode, the register file is divided into 9 working-register groups, each occupying 16 contiguous register locations (Figure 1-2). A register pointer (one of the control registers) addresses the starting location of the currently active working-register group.

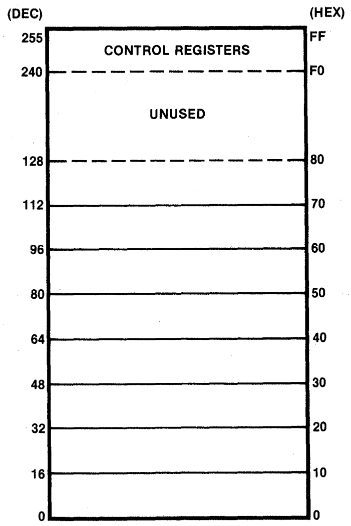


Figure 1-2. Working-Register Groups

NOTE: Changing the value of the register pointer is an easy way to save the 16 currently-active working registers (as during interrupt processing). Reserving one or more working-register groups for the use of interrupt-handling routines is a recommended programming practice.

In the following example, the Set Register Pointer (SRP) instruction sets the register pointer to 240, the starting address of the control register group. The following Load (LD) instruction initializes register 252 to ten.

```
SRP #240            !Register Pointer contains F0 (hex)!
LD  R12, #10        !Working register 12 occupies register
                     location 252!
```

Because of their special significance, the I/O port registers (0-3) and control registers (240-255) are referenced later in this section. The control registers are particularly pervasive in all Z8 operations, since they are used in the handling of I/O, interrupts, the timer/counter, program control flags, and the program stack, as well as to point to the current working-register group. To provide a quick reference in the following sections, the control registers are listed in Figure 1-3. They are described in bit-level detail in the Z8 Technical Manual. Z8 instructions can reference control registers by number or by their predefined symbolic identifiers shown in the following figure.

NOTE

Register memory addresses 128-239 do not exist and should not be specified in Z8 instructions. The result of accessing these locations is undefined.

LOCATION		IDENTIFIERS
255	STACK POINTER (BITS 7-0)	SPL
254	STACK POINTER (BITS 15-8)	SPH
253	REGISTER POINTER	RP
252	PROGRAM CONTROL FLAGS	FLAGS
251	INTERRUPT MASK REGISTER	IMR
250	INTERRUPT REQUEST REGISTER	IRQ
249	INTERRUPT PRIORITY REGISTER	IPR
248	PORTS 0-1 MODE	P01M
247	PORT 3 MODE	P3M
246	PORT 2 MODE	P2M
245	T0 PRESCALER LOAD	PRE0
244	TIMER/COUNTER 0 LOAD	T0
243	T1 PRESCALER LOAD	PRE1
242	TIMER/COUNTER 1 LOAD	T1
241	TIMER MODE	TMR
240	SERIAL I/O	SIO
	NOT IMPLEMENTED	
127 4	GENERAL-PURPOSE REGISTERS	
3	PORT 3	P3
2	PORT 2	P2
1	PORT 1	P1
0	PORT 0	P0

Figure 1-3. Control Registers

1.2.4 Data Lengths

Z8 instructions can operate on individual bits, 4-bit Binary Coded Decimal (BCD) digits or nibbles, 8-bit bytes, or 16-bit words (Figure 1-4). Bits can be set, reset, or tested. Nibbles are used in BCD arithmetic operations. Bytes are used for character or small integer values (in the range 0 to 255 if unsigned, or in the range -128 to 127 if signed). Words are used for larger integer values (in the range 0 to 65535 if unsigned, or in the range -32768 to 32767 if signed).

The basic data element of the Z8 is the byte. Memory locations (whether they reside in program, data, or register memory segments) are ordinarily accessed eight bits at a time. Increment Word (INCW) and Decrement Word (DECW) are the only instructions that operate on 16-bit words.

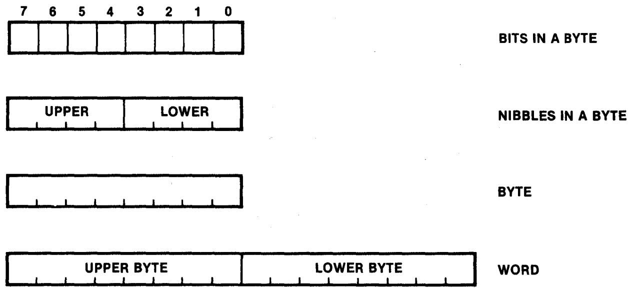


Figure 1-4. Data Lengths

1.3 Input/Output

Thirty-two of the Z8's 40 lines are dedicated to input and output. These 32 lines are grouped into 4 ports of 8 lines each and can be configured as input, output, or address/data. Under software control, the ports can be programmed to provide timing, interrupt requests, status signals, and serial or parallel I/O features with or without handshake.

In the following explanation of the various port functions, Port 1 is described before Port 0 for convenience. The Z8 architecture diagrams (Figures 1-5 and 1-6) show the I/O lines and signals referenced. Although they are pictured separately in Figure 1-5, remember that from a programming standpoint the I/O ports, timers, interrupt controls, flags, and register pointer are all manipulated through register memory.

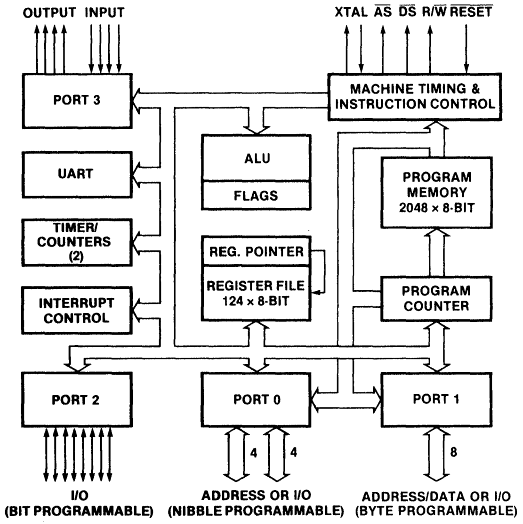
1.3.1 Port 1

Port 1 can be programmed as a byte I/O port or as an address/data port for interfacing to external memory. Associated with Port 1 are the Address Strobe (AS), Data Strobe (DS) and Read/Write (R/W) timing signals. Under program control, two lines from Port 3 (lines P33 and P34) can be used with Port 1 as the handshake control lines (DAV1 and RDY1) or as a Port 1 interrupt request input (IRQ1) and an external data memory access (DM) status output.

If external data memory is to be accessed, Port 1 is programmed as an address/data port through which the external address and data are passed. In this case, the lower eight bits of the address (A0-A7) are multiplexed with data bits (D0-D7). If an address longer than eight bits is required, the additional address bits (A8-A15) originate from Port 0.

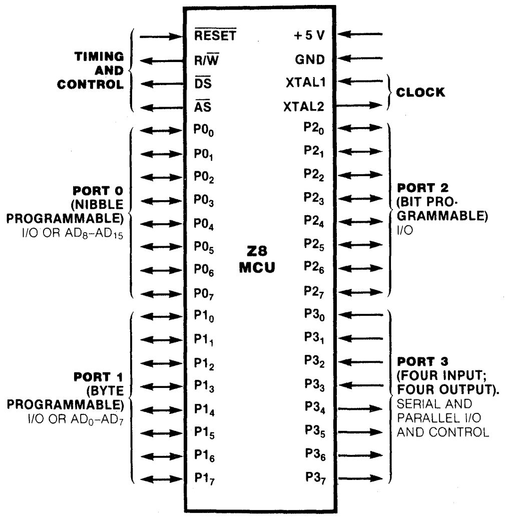
1.3.2 Port 0

Port 0 can be programmed to be either an I/O port or an address output for external memory. Depending on the size of the address, Port 0 provides either bits 8-11 or bits 8-15 of the address. (Port 1 provides bits 0-7). If the address is 12 bits or less, the upper four bits (nibble) of Port 0 can be programmed independently as I/O while the lower nibble is used for addressing. When Port 0 is used in the I/O mode, two lines from Port 3 (lines P32 and P35) can be used for the handshake controls DAV0 and RDY0.



OUTPUT INPUT
XTAL $\overline{AS}$ $\overline{DS}$ R/$\overline{W}$ $\overline{RESET}$
PORT 3
MACHINE TIMING & INSTRUCTION CONTROL
UART
ALU
FLAGS
PROGRAM MEMORY 2048 × 8-BIT
TIMER/ COUNTERS (2)
REG. POINTER
REGISTER FILE 124 × 8-BIT
INTERRUPT CONTROL
PROGRAM COUNTER
PORT 2
PORT 0
PORT 1
4
4
8
I/O (BIT PROGRAMMABLE)
ADDRESS OR I/O (NIBBLE PROGRAMMABLE)
ADDRESS/DATA OR I/O (BYTE PROGRAMMABLE)

Figure 1-5. Z8 Architecture Diagram



TIMING AND CONTROL
$\overline{RESET}$
R/$\overline{W}$
$\overline{DS}$
$\overline{AS}$
+5 V
GND
XTAL1
XTAL2
CLOCK
PORT 0 (NIBBLE PROGRAMMABLE) I/O OR AD_8-AD_{15}
$P0_0$ $P0_1$ $P0_2$ $P0_3$ $P0_4$ $P0_5$ $P0_6$ $P0_7$
Z8 MCU
$P2_0$ $P2_1$ $P2_2$ $P2_3$ $P2_4$ $P2_5$ $P2_6$ $P2_7$
PORT 2 (BIT PROGRAMMABLE) I/O
PORT 1 (BYTE PROGRAMMABLE) I/O OR AD_0-AD_7
$P1_0$ $P1_1$ $P1_2$ $P1_3$ $P1_4$ $P1_5$ $P1_6$ $P1_7$
$P3_0$ $P3_1$ $P3_2$ $P3_3$ $P3_4$ $P3_5$ $P3_6$ $P3_7$
PORT 3 (FOUR INPUT; FOUR OUTPUT). SERIAL AND PARALLEL I/O AND CONTROL

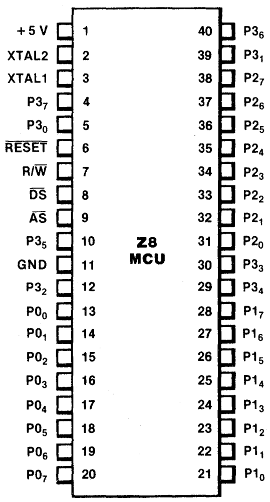


Figure 1-6. Z8 Pin Functions and Assignments

1.3.3 Port 2

Port 2 can be programmed for input or output on a line-by-line (bitwise) basis. As in the case of Ports 0 and 1, two lines from Port 3 (lines P31 and P36) can be programmed as the handshake control lines DAV2 and RDY2. The output buffers of Port 2 have a programmable option for inhibiting the active pull-ups to provide open-drain type outputs.

1.3.4 Port 3

Port 3 can be programmed for I/O and/or as a control port. In I/O mode, the direction of the eight lines is fixed as four in and four out. The control functions of Port 3 are handshake, interrupt request, timer in and out, and status out.

Two lines of Port 3 can be programmed as a serial input and a serial output interface. Each line has an 8-bit serial/ parallel register associated with it. Serial I/O uses an asynchronous format with the bit rate controlled by the internal timer. Interrupts are generated when a character is received or transmitted.

1.4 Interrupts

The Z8 allows six different interrupts from eight possible sources -- the four input lines of Port 3, serial in and out, and the two timers (T0 and T1, discussed in Section 1.5).

Six bits in the Interrupt-Mask control register can enable/ disable the six interrupts IRQ0-IRQ5 individually. When more than one interrupt is pending, priorities are resolved by a priority encoder, controlled by the Interrupt-Priority control register.

Interrupt requests are stored in an Interrupt-Request control register, which can also be used for polling. When an interrupt request is granted, the Z8 enters an "interrupt machine cycle" that globally disables all other interrupts, saves the program counter (address of the next program instruction to be executed) and status flags, and finally branches to the vector location for the interrupt. It is only at this point that control passes to the interrupt-handling procedure for the interrupt.

Before the Z8 can recognize interrupts following RESET, some nitialization tasks must be performed. RESET causes the nterrupt Request Register (IRQ0 - IRQ5) to be cleared and held o zero, and interrupts to be globally disabled (bit 7 of the nterrupt Mask Register = 0). The initialization routine should onfigure the Z8 interrupt requests to be enabled/disabled (via the IMR) as required by the target application, and prioritized for vectored interrupts (via the Interrupt Priority Register). Because RESET holds the IRQ register to zero, one final step is required before interrupts can function, even in polled mode. Specifically, interrupts must be globally enabled via the EI instruction; simply setting bit 7 of IMR is not sufficient. Subsequent to this EI, interrupts can be enabled either by IMR register manipulation or by use of the EI instruction, with equivalent effects.

Additionally interrupts must be disabled by executing a DI instruction before the IPR or IMR control registers can be modified. Interrupts can then be enabled by executing an EI instruction.

1.5 Timers/Counters

The Z8 has two 8-bit counters (T0 and T1), each driven by its own 6-bit prescaler. The prescalers can be driven, in turn, by either an internal (T0 and T1) or external (T1 only) clock source. T0 and T1 can operate independently of the processor instruction sequence and, consequently, can unburden the program from time-critical operations like event counting or elapsed-time calculation.

Each prescaler can be programmed to divide the input frequency of its clock source by any number from 1 to 64. The prescaler drives its counter, which decrements a value (0 through 255) stored in the timer register. When the timer register reaches end-of-count, a timer interrupt request--IRQ4 (T0) or IRQ5 (T1)--is generated.

Under program control, counters/prescalers can be started, stopped, restarted to continue counting, or restarted from the initial value of the counters. The counters can also be programmed to stop on reaching end-of-count, or to automatically reload the initial counter value and continue counting. Either counter can be read at any time without disturbing its value or count mode.

The clock source for the T1 counter/prescaler can be either the microprocessor clock or an external timer input. Under program control, the external timer input can function as an external clock (maximum frequency 1 MHz), a trigger input that can be retriggerable or not, or as a gate input for the internal clock.

)ne line of Port 3 also serves as a timer output through which ŗ0, T1, or the internal clock can be output. The timer output :oggles whenever an end-of-count occurs. If the timer is)rogrammed to reload the count value and continue at ɜnd-of-count, this line produces a 50% duty cycle. The counters :an be cascaded by feeding the timer input line with the timer)utput.

l.6 Status Flags and Program Controls

ľhe ability to test data and make decisions based on the result .s especially important in single-chip microcomputers. Programs ⁄ritten for these computers tend to be dominated by control .nstructions (conditional and unconditional jumps, calls, etc.) ınd by test and mask instructions.

:ontrol register 252 contains six flags for the use of the Z8 ›rocessor and programmer.

Carry	(C)
Zero	(Z)
Sign	(S)
Overflow	(V)
Decimal Adjust	(D)
Half Carry	(H)

he half carry and decimal adjust flags are used only by the Z8. he other flags can be used by the programmer with the Jump (JP) nd Jump Relative (JR) instructions to provide a repertoire of 19 onditional tests.

xamples:

```
JP  NC, SUBTOT    !Jump to routine named
                   SUBTOT if carry is not set!

JR  OV, $+50      !Jump 50 bytes ahead in program
                   if overflow has occurred!
```

.6.1 Carry Flag

he carry flag (C) is affected by Addition (ADD, ADC), ubtraction (SUB, SBC), Compare (CP), Decimal Adjust (DA), Rotate RL, RLC, RR, RRC), Swap (SWAP), and the Interrupt Return (IRET) nstructions. When set, it generally indicates a carry out of ıe bit 7 position of a register being used as an accumulator. ɔr example, adding two 8-bit numbers as in the following nstructions would cause a carry out of bit 7 and set the carry lag.

```
        LD     20, #225    !Load value 225 into location 20!
        ADD    20, #64     !Add 64 to contents of location 20!

        Bit         7 6 5 4 3 2 1 0
        ---         - - - - - - - -
        225         1 1 1 0 0 0 0 1
    +    64         0 1 0 0 0 0 0 0
        ---         - - - - - - - -
        289       ┌─0 0 1 0 0 0 0 1
                  │
                  │
                  └►1 = carry flag
```

The carry flag can be set to one by the Set Carry Flag (SCF) instruction, cleared to zero by the Reset Carry Flag (RCF) instruction, and complemented (changed to 0 if 1, and vice-versa) by the Complement Carry Flag (CCF) instruction.

1.6.2 Zero Flag

The zero flag is affected by the same instructions as the carry flag plus the Logical (AND, OR, XOR, COM), Increment and Decrement (INC, INCW, DEC, DECW), and Test (TCM, TM) instructions. In general, the zero flag is set when the "accumulator" register's contents following one of the above operations is zero.

```
        DEC    20            !Decrement location 20 contents!
        JP     Z, 1500       !Jump to program location 1500
                              if location 20 is zero after
                             decrementing!
```

1.6.3 Sign Flag

The sign flag is affected by the same instructions as the zero flag. The sign flag is set to one when bit 7 of the register used as an accumulator in these operations contains a one (a negative number in twos complement representation) following the operation.

```
        LD   20, SUB1            !Load value of variable named
                                  SUB1 into location 20!
        SUB  20, SUB2            !Subtract value of SUB2!
        JP   MI, NEG             !If S=1 (result is "minus"),
                                  jump to location labeled NEG!
```

1.6.4 Overflow Flag

The overflow flag is affected by the same instructions as the zero and sign flags. When set, the overflow flag indicates that a twos-complement number in a result register is in error since it has exceeded the largest (+127) or is less than the smallest (-128) number than can be represented in twos-complement notation. Consider the following, as an example:

```
      Bit      7 6 5 4 3 2 1 0

      120      0 1 1 1 1 0 0 0
    + 105      0 1 1 0 1 0 0 1
      225    ┌─1 1 1 0 0 0 0 1
             │
             │
             └►0 = carry flag
```

The result in this case (-95) is incorrect. In this case, the overflow flag would be set.

```
      SUB  20, 21       !Subtract location 21's contents from
                         location 20's!
      JR   OV, $-50     !Jump back 50 bytes in program
                         if overflow has occurred!
```

1.6.5 Decimal Adjust Flag

The decimal adjust flag is used for BCD arithmetic. Since the algorithm for correcting BCD operations is different for addition and subtraction, this flag is used to specify what type of instruction was executed last so that the subsequent Decimal Adjust (DA) operation can do its function correctly. The decimal adjust flag cannot normally be used as a test condition by the programmer.

1.6.6 Half Carry Flag

The half carry flag indicates a carry out of, or a borrow into bit 3 as the result of adding or subtracting two 8-bit bytes, each representing two BCD digits. The half carry flag is used by the Decimal Adjust (DA) instruction to convert the binary result of a previous addition or subtraction into the correct decimal (BCD) result. As in the case of the decimal-adjust flag, this flag is not normally accessed by the user.

1.7 Stack Memory

To support the power of its interrupt capability, the Z8 has a flexible stack scheme and a fast "context-switching" mechanism. Context switching refers to the saving and restoration of working registers, the program counter, flags, and other pertinent information when an interrupt occurs.

Under program control, the Z8 can use an internal (register memory) or external (data memory) stack, limited in size only by the available memory space. An 8-bit or 16-bit stack pointer occupies control register 255 or the control register pair 254-255.

CALL instructions use the stack to store the program counter before branching to a procedure. Interrupts automatically save the program counter and flag register. The RET instruction restores the program counter from the stack upon return from a procedure, while IRET also restores the flag register upon return from an interrupt procedure. In addition, the Z8 has PUSH and POP instructions that can save and restore any register of the register file.

Section 2
Z8 Assembler Conventions

2.1 Assembler Overview

The Z8 microcomputer is programmed in a symbolic assembly language (PLZ/ASM). This marks a significant improvement over coding in binary notation. The operation codes for assembly-language statements are easily memorized (DEC for decrement and DECW for decrement word). In addition, meaningful symbolic names can be assigned to program addresses and data (MULTIPLY_ROUTINE as the label of the first statement in a multiply procedure).

A Z8 source module consists of PLZ/ASM assembly language statements. These statements are then translated by the Z8 assembler into an object module that can either be separately executed by the Z8 microcomputer, or can be linked with other object modules to form a complete program. Because the assembler has some high-level features, a source module can also include PLZ constructs such as DO and IF statements. The user can also embed assembler directives, which control the operation of the assembler, in the source module. High-level statements and assembler directives are discussed in Sections 4 and 5.

Depending on the assembler directives used, addresses within an object module or program can be absolute (meaning addresses in the source program correspond exactly to Z8 program memory addresses) or relocatable (meaning addresses can be assigned relative to some base address at a later time). Object modules should be made relocatable wherever possible so they can be linked with other object modules, and so that object programs can be loaded anywhere in memory. It also allows the creation of libraries of commonly used procedures (including math or input/output routines) that can be linked selectively into several programs as desired.

Operation of the assembler, module linkage, address relocation, and program execution are discussed in the Z8 PLZ/ASM Assembler User Guide.

2.2 Assembly Language Statement Format

The most fundamental component of a PLZ/ASM program is the assembly-language statement consisting of an instruction and its operands. The instruction describes an action to be taken; the operands supply the data to be acted upon.

An assembly language statement can actually include four fields:

- Statement labels
- An instruction
- Operands
- Comments

The statement label and comment fields are always optional. The statement has zero, one, or two operands, depending on the instruction selected. The following statements have the same effect in a Z8 program, but the second is much more descriptive (and consequently more helpful in program debugging).

```
Label        Instruction       Operand(s)       Comment

             LD                T1, #255
INITT1:      LD                T1, #255         !Load Timer 1
                                                 initial value!
```

Each of the elements of a PLZ/ASM program must be separated from other elements by one or more delimiters. A delimiter is one of the characters: space (blank), comma, semicolon, tab, carriage return, line feed, or form feed. Note that carriage return is treated the same as any other delimiter, so that a single statement may span several lines, or several statements may appear on a single line. The delimiter used in a specific situation is up to the programmer. For the sake of illustration, this manual uses blanks to separate statement fields and commas to separate operands.

2.2.1 Program Labels and Identifiers

Any assembly-language (or high-level) statement in a Z8 program can be preceded by any number of labels. Any statement referenced by another statement must be labeled. A label consists of an identifier followed by a colon (:) in the form:

```
label1:  label2: ... labeln:  statement
```

A PLZ/ASM identifier can contain up to 127 characters, of which the first must be a letter. The remaining characters can be letters, digits, or the special character underscore (_). Letters can be capitalized or lower-cased, but each time an identifier is used, it must be written in exactly the same way. The following are valid identifiers:

```
START_UP_ROUTINE
Program_Initialization
A
Loop_12
N1
sort
```

In addition to their statement-labeling function, identifiers serve as symbolic names for constants (Section 2.3.2), data variables (Section 2.3.3), and procedures (Section 5.2.2). Certain identifiers serve as PLZ/ASM keywords and should not be used as programmer-defined identifiers (see Appendix D).

An identifier can be associated with only one item within the scope of its definition. Section 4.2.5 explains the scope of identifiers, including the scope of labels. For the moment, we can say that labels are accessible within the module in which they are defined, and are not accessible outside that module unless specifically declared to be GLOBAL or EXTERNAL.

2.2.2 Instruction

The instruction is the assembly-language mnemonic describing a specific action to be taken.

```
LD R5, R10      !Load register 5 from register 10!

SRP #%10        !Set Register Pointer to 10 (hex)!
```

The instruction must be separated from its operands by a delimiter.

2.2.3 Operand Field

Depending on the instruction specified, this field can have zero, one, or two operands. If two operands are needed, they must be separated by a delimiter.

```
CCF             !No operand!

SRP #%10        !One operand!

ADD R6, #210    !Two operands!
```

Operands supply the information the instruction needs to carry out its action. An operand can be:

- Data to be processed (immediate data);
- The address of a location from which data is to be taken (source address);
- The address of a location where data is to be put (destination address);
- The address of a program location to which program control is to be passed;
- A condition code, used to direct the flow of program control.

When these operand types are combined, the possible orderings are as follows:

Instruction Destination, Immediate

Instruction Destination, Source

Instruction Condition Code, Program Location

Immediate data can be in the form of a constant, an address, or an expression (constants and/or addresses combined by operators). Each of these forms is described in Section 2.3.

```
LD  R0, #K               !Load constant K into reg 0!

LD  R0, #COUNTER         !Load address of COUNTER into
                          reg 0!

ADD R0, #(CON/3 + 5)     !Add value of expression
                          (CON/3 + 5) to contents of
                          reg 0!
```

Source, destination, and program addresses can also take several forms. PLZ/ASM addressing modes are described in Section 2.4. Some examples are:

```
        LD   R0, @R5              !Load value whose address is in
                                   register 5 into register 0!

        LD   55, VAR1             !Load value located at address
                                   labeled VAR1 into register
                                   file location 55!

        JP   Z, LOOP1             !Jump to program address
                                   labeled LOOP1 if zero flag (Z)
                                   is set!

        JP  NZ, LOOP1 + 5         !Otherwise, jump to location
                                   five bytes after LOOP1!
```

Condition codes are listed in Section 3.2. They are used only by the Jump (JP) and Jump Relative (JR) instructions.

2.2.4 Comments

Comments are used to document program code as a guide to program logic and to simplify present or future program debugging. Comments can be inserted anywhere a program delimiter can appear. Comments are bounded by exclamation points (!) and can contain any characters except the exclamation point itself.

```
        !Module 3, Changed 7-25-78!

        RES R13, #3  !Turn off "convert" flag!
```

A single comment can cross line boundaries; that is, carriage returns can occur within a comment.

Comments can also start with //. This type of comment does not need a closing symbol; the carriage return, <CR>, at the end of the line on which this comment appears terminates the comment.

```
        //Module 3, Changed 7-25-78 <CR>

        RES    R13, #3  //Turn off "convert" flag <CR>
```

2.3 Arithmetic Operands

Constants and data variables are types of operands that can be used in assembly-time arithmetic expressions, or singly as operands for a Z8 instruction. This section describes the differences between assembly-time and run-time arithmetic, defines constants and data variables, and explains how expressions are formed using operators.

2.3.1 Run-Time Versus Assembly-Time Arithmetic

Arithmetic is performed in two ways in an assembly-language program. Run-time arithmetic is done while the program is actually executing.

```
        SUB  R10, R12    !Subtract the contents of reg 12
                          from the contents of reg 10!
```

Assembly-time arithmetic is done by the assembler when the program is assembled and involves the evaluation of arithmetic expressions in operands, such as the following:

```
        LD  R0, #(22/7 + X)

        JP  Z, LOOP1 + 12

        ADD R2, @HOLDREG-1
```

Assembly-time arithmetic is more limited than run-time arithmetic in such areas as signed versus unsigned arithmetic and the range of values permitted.

Only unsigned arithmetic is allowed in assembly-time expression evaluation. Run-time arithmetic uses both signed and unsigned modes, as determined from the assembly-language instruction specified and the meaning attached to operands by the programmer.

All assembly-time arithmetic is computed using 16-bit arithmetic, "modulo 65536". Values greater than or equal to 65536 are divided by 65536 and the remainder of the division is used as the result. If the result of assembly-time arithmetic is to be stored in a single byte location, the resulting value must be representable in 8 bits; that is, the range 0 to 255 (or -128 to 127 if signed representation is intended).

```
        LD  R10, #X+22          !Result of (X+22) must be in
                                 range -128 to 255!

        JP  X+22                !Modulo 65536.  Result is the
                                 address 22 bytes beyond X!
```

2.3.2 Constants

A constant value is one that does not change throughout the program. Constants can be expressed as numbers, as character sequences, or as a symbolic name representing a constant value.

Numbers can be written in decimal, hexadecimal, binary, or octal notation. The latter three are preceded by a percent sign (%) and, in the case of binary and octal, by a base specifier enclosed in parentheses. If a number has no prefix, decimal is assumed.

10	decimal
%10	hexadecimal
%AF0F	hexadecimal
%(2)10110010	binary
%(8)70	octal

A character sequence is a sequence of one or more characters enclosed in single quote marks. Any ASCII character (except a percent sign or single quote) can be included in the character sequence. Since constants are represented as 32-bit values, only the first four characters in a string literal, used as a constant, are meaningful, for example, 'ABCD = ABCDE'.

```
'A'
'This is a character sequence'
```

A character can also be represented in a character sequence in the form "%hh," where "hh" is the hexadecimal equivalent of the ASCII code for the character. (See Appendix E for the ASCII character set and its hexadecimal equivalents.)

```
'Here is an ESC character:  %1B'
```

For convenience, certain ASCII characters have been assigned shorter, more mnemonic codes as follows:

%L or %l	Linefeed
%T or %t	Tab
%R or %r	Carriage Return
%P or %p	Page (Form Feed)
%%	Percent Sign
%Q or %q	Single Quote

Example:

```
'First line%rSecond line%r'
'Quote%Qinside a quote%Q'
```

A constant can be assigned a symbolic name by a constant definition (CONSTANT) statement. A symbolic identifier, once associated with a constant value, retains that value through the entire program module.

Constant symbols are defined by the CONSTANT statement in the form shown below. Identifiers follow the rules outlined in Section 2.2.1. The special character pair ":=" can be read "is defined as".

```
        CONSTANT
           REC_LENGTH            := 64
           BUFFER_LENGTH         := 4*REC_LENGTH
           SEMICOLON             := ';'
           BIGNUMBER             := 65000
           smallnumber           := -1
```

Certain constants are predefined by the assembler and should not be used for programmer-defined constant symbols (see Appendix D).

2.3.3 Data Variables

A data variable can be thought of as a container that can hold different values from time to time. An 8-bit (BYTE or SHORT_INTEGER) variable can hold values in the range 0 to 255 if unsigned, or -128 to 127 if signed twos-complement representation is intended. Similarly, a 16-bit (WORD or INTEGER) variable can hold values in the range 0 to 65535 if unsigned, or -32768 to 32767 if signed.

NOTE

> BYTE and WORD variables should be used for unsigned values, and SHORT_INTEGER and INTEGER variables should be used for signed values; there are no restrictions on whether a particular variable is signed or unsigned. Therefore, BYTE and SHORT_INTEGER are treated as equivalent, as are WORD and INTEGER; the appropriate interpretation is made by the programmer.

A data variable name can be associated with either a register or data memory location; the value of the variable is the contents of that location at the time the variable is referenced. A data variable name is a symbolic identifier and follows the rules for identifiers in Section 2.2.1.

```
        LD    R5, MPLIER          !Load the value contained in the
                                   location symbolized by MPLIER!

        ADD   R5, 3 + SUBTOTAL    !Add the value contained in the
                                   location 3 bytes after the
                                   location named SUBTOTAL to the
                                   contents of reg 5!
```

If a data variable operand is preceded by #, it is treated as immediate data and the value used is the data address associated with the variable, not the contents of the location. For example, suppose register 50 has the symbolic name COUNTER and contains the bit pattern 11111111 (decimal 255).

```
        LD   T1, COUNTER         !255 is loaded into Timer 1!

        LD   T1, #COUNTER        !50 is loaded into Timer 1!

        LD   T1, COUNTER - 5     !Contents of reg 45 are loaded
                                  into Timer 1!

        LD   T1, #COUNTER - 5    !45 is loaded into Timer 1!
```

Every data variable name has a type and scope associated with it, as well as a value. The type and scope (and, optionally, the initial value) are defined in a variable declaration statement like the following:

```
        INTERNAL
            SWITCH1  BYTE
```

In this example, INTERNAL is the scope of the variable SWITCH1, and BYTE is its type.

Variable declaration is the subject of Section 5.3.4. For the purposes of this section, variables can have GLOBAL, EXTERNAL, INTERNAL, or LOCAL scope. They can be one of the simple types, BYTE or SHORT_INTEGER (for 8-bit values) or WORD or INTEGER (for 16-bit values). They can also be one of the structured types, ARRAY or RECORD.

2.3.4 Expressions and Operators

Expressions are formed using arithmetic, logical, shift, and relational operators in combination with constants and variables. These operators allow both unary (one-operand) and binary (two-operand) expressions, as shown below.

Arithmetic Operators

The arithmetic operators are as follows:

Operator	Operation
+	Unary plus, binary addition
-	Unary minus, binary subtraction
*	Unsigned multiplication

Operator	Operation
/	Unsigned division
MOD	Unsigned modulus

The division operator (/) truncates any remainder. The MOD operator returns the remainder from dividing its operands.

```
17/4 = 4

17 MOD 4 = 1
```

If zero is specified as the right operand for either of these division operators, the result is undefined.

Examples:

```
ADD  R5, #-3              !A minus 3 is added to
                           reg 5!

ADD  R5, #K + (5*3)       !Value of constant K + 15 is
                           added to reg 5!
```

Once again, expressions containing these operators are evaluated at assembly time and, consequently, the arithmetic performed is unsigned. Signed arithmetic can still be done at run time, however. Signed multiplication, for example, can be done by looping through a series of Shift and Add instructions.

Logical Operators

The logical operators are as follows:

Operator	Operation
LNOT	(Unary) Logical complement
LAND	Logical AND
LOR	Logical OR
LXOR	Logical EXCLUSIVE OR

LNOT simply complements the bit pattern of its (single) operand. All one bits are changed to zero and vice-versa.

```
LD  P2M, LNOT MASK        !Reverse the bits in the
                           mask used to program Port 2!
```

The effect of LAND, LOR, and LXOR can be seen from the following examples. Assume two constants A and B have the bit patterns 11110000 and 01010101, respectively. The expressions:

```
        A LAND B
        A LOR B
        A LXOR B
```

will result in the following evaluations of the operands:

```
LAND  11110000  LOR  11110000  LXOR  11110000
      01010101       01010101        01010101
      --------       --------        --------
      01010000       11110101        10100101
```

LAND sets a one bit whenever both ANDed bits are one; LOR sets a one bit whenever either ORed bit is one; LXOR sets a one bit when the two EXCLUSIVE-ORed bits are different.

The assembly-time logical operations performed by LNOT, LAND, LOR, and LXOR can also be done at run time by the Z8 instructions COM, AND, OR, and XOR. The assembly-time operations require less code and register manipulation. The run-time operations allow greater flexibility, however. For example, they can operate on registers (variables) whose contents are not known at assembly time, as well as on known constant values.

Shift Operators

The shift operators are as follows:

SHR	Logical shift right
SHL	Logical shift left

When used in expressions, the shift operators have the form

```
        d operator n
```

where "d" is the data to be shifted and "n" specifies the number of bits to be shifted. Vacated bits are replaced with zeros. For example, if the constant PRODUCT is equal to 10110011, the statement

```
        LD  R0, #(PRODUCT SHL 2)
```

would load the value 11001100 into working register 0.

If the second operand supplied is negative (that is, if the sign bit is set), it has the effect of reversing the direction of the shift.

```
        ADD  PRODUCT, #(MPLIER SHR -1)  !MPLIER is shifted left
                                         one bit position!
```

Relational Operators

The relational operators are as follows:

<	Less than
<=	Less than or equal
=	Equal
<>	Not equal
>=	Greater than or equal
>	Greater than

These six relational operators return a logical TRUE value (all ones) if the comparison of the two operands is true, and return a logical FALSE value (all zeros) otherwise. The operators assume both operands are unsigned.

```
LD  R0,#(1=2)              !Reg 0 is loaded with zeros!
LD  R0,#(2+2) < 5          !Reg 0 is loaded with ones!
```

Precedence of Operators. Expressions are generally evaluated left to right with operators having the highest precedence evaluated first. If two operators have equal precedence, the leftmost is evaluated first.

The following lists the PLZ/ASM operators in order of precedence:

1. Unary operators: +; -, LNOT

2. Multiply/Divide/Shift/AND: *, / , MOD, SHR, SHL, LAND

3. Add/Subtract/OR/XOR: +, -, LOR, LXOR

4. Relational operators: <, <=, =, <>, >=, >

Parentheses can be used to change the normal order of precedence. Items enclosed in parentheses are evaluated first. If parentheses are nested, the innermost are evaluated first.

```
20/5 - 12/3  = 0

20/(5 - 12/3) = 20
```

Modes of Arithmetic Expressions. All arithmetic expressions have a mode associated with them: absolute, relocatable, or external. These modes are defined in detail in Section 4.3.3.

2.4 Z8 Addressing Modes

With the exception of immediate data and condition codes, all assembly-language operands are expressed as addresses: register addresses, program memory addresses, and external data memory addresses. The various address modes recognized by the Z8 assembler are as follows:

- Register
- Indirect Register
- Indexed Address
- Direct Address
- Relative Address
- Immediate

Special characters are used in operands to identify certain of these address modes. The characters are:

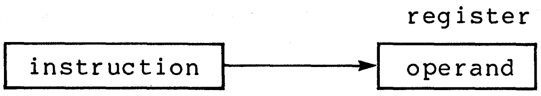
- "R" preceding a working-register number;
- "RR" preceding a working-register pair;
- "@" preceding an indirect-register reference;
- "#" preceding immediate data;
- "()" used to enclose the index register part of an indexed address;
- "$" signifying the current program counter location, usually used in relative addressing.

The use of these characters is shown in the following sections.

Not every address mode can be used by every instruction. The individual instruction descriptions in Section 3 describe which address modes can be used for each instruction.

2.4.1 Register Address

In register addressing mode, the operand value is the contents of the specified register.



The register can be addressed in either of two ways. The address can be:

- an 8-bit address in the range 0-127, 240-255, or

- a 4-bit working-register address.

The full 8-bit address is indicated in the operand field by supplying an expression (see Section 2.3.4) which evaluates to either the number of the register or a variable name associated with a register. Neither of these has a special-character prefix.

```
LD  55, #%F3           !Load register 55 with the
                        hexadecimal value F3!

LD  FLAGS, #ZEROS      !Load register named FLAGS
                        (i.e., reg 252) with the value
                        of the constant named ZEROS!

ADD  120, SUBTOTAL     !Add contents of register named
                        SUBTOTAL to register 120!
```

Working-Register Address

Designating a register by a 4-bit working-register address, rather than an 8-bit register address, often reduces the length of an instruction and results in a shorter execution time. In this case, the full address is formed by concatenating the 4-bit field (address range 0-15) with the upper 4 bits of the Register Pointer; thus the working-register set can be varied dynamically simply by changing the value of the Register Pointer (control register 253).

A working-register operand is indicated by a number in the range 0-15 preceded by the letter "R".

```
LD  R0, R15        !Load the contents of working
                    register 15 into working register 0!

LD  53, R15        !Load the contents of working register
                    15 into register 53!

ADD R6, AUGEND     !Add contents of register named
                    AUGEND to working register 6!
```

Register Pair Address

Registers can be used in pairs to designate 16-bit values or memory addresses. A register pair can be specified as an expression which evaluates to an <u>even number</u> in the range 0,2,4...126 or 240,242...254. It can also be designated as the variable name of an even-numbered register.

```
        DECW   20          !Decrement contents of registers 20
                            and 21!
```

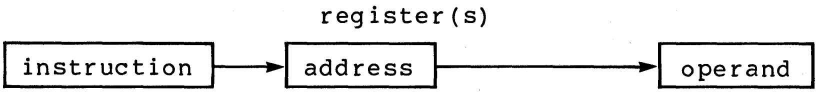
Working-Register Pair Address

Working-register pairs are indicated by an even number in the range 0, 2, 4, 6, ... 14. If the low-order (odd) byte of a pair is specified, an error will result. In the case of a working register pair, the register number is preceded by the letters "RR." For example, RR0 refers to R0 and R1.

```
        INCW   RR10        !Increment contents of working
                            registers 10 and 11!
```

2.4.2 Indirect-Register Address

In indirect addressing, the value of the operand is not the contents of a register. Instead, the register (register pair, working register or working-register pair) contains the address of the location whose contents are to be used as the operand value.



Depending on the instruction selected, the address may point to a register, program memory, or external data memory location. Register pairs or working-register pairs are used to hold the 16-bit addresses when accessing program or external data memory. Pairs are indicated by an even number (see Section 2.4.1).

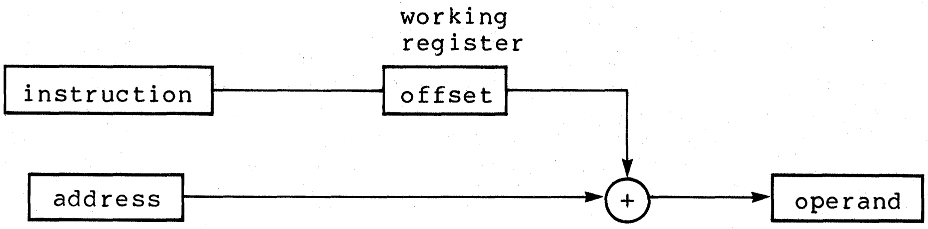
The indirect-register address mode may save space and improve execution speed when data is accessed from consecutive locations. This mode can also be used to simulate more complex addressing modes, since addresses can be computed before the associated data is accessed.

An indirect address can be an expression which evaluates to a register number, the variable name of a register file location, or a register-pair designator. It can also be a working-register designator, or a working-register-pair designator. In all cases, the register specification must be preceded by a commercial at symbol (@).

```
JP      @RR0                  !Pass control (jump) to the
                               program memory location
                               addressed by working
                               register pair 0-1!

JP      @20                   !Jump to program location
                               addressed by register pair
                               20-21!

LD      @TOTALS, 30           !Load contents of register
                               30 into location addressed by
                               register named TOTALS!

LD      @TOTALS, #30          !Load immediate value 30 into
                               location addressed by TOTALS!
```

2.4.3 Indexed Address

An indexed address consists of a register address offset by the contents of a designated working register (the index). This offset is added to the register address and the resulting address points to the location whose value is used by the instruction. This address mode is used only by the Load (LD) instruction.



The register address is specified as an expression which evaluates to a register-file number or to the variable name of a register file location. This address is followed by the index, a working-register designator enclosed in parentheses.

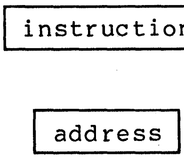
```
LD  R10, TABLE(R0)       !Load the contents of the
                          location addressed by TABLE
                          plus the contents of reg 0
                          into reg 10!
```

```
        LD   240(R0), R10          !Load the contents of reg 10 into
                                    the control register whose
                                    address is 240 plus the contents
                                    of reg 0!
```

Since the specified register address and working register contents are both 8-bit values, the indexed-address mode can be used for base addressing. The base address is loaded into the working register and the offset (register address) then completes the address field.

2.4.4 Direct Address

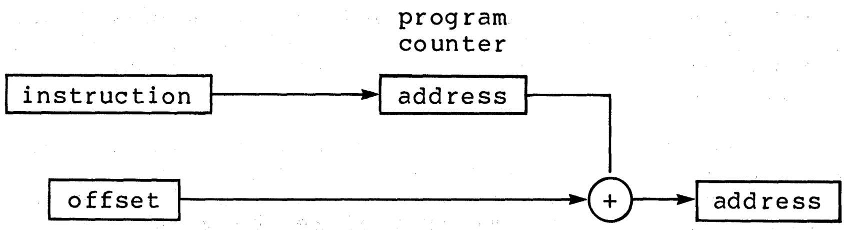
Direct-address mode is used only by the Jump (JP) and Call (CALL) instructions to specify the destination where program control is to be transferred. The address may be specified as an expression which evaluates to a number or a program label.



```
        CALL  MATH_ROUTINE         !Transfer control to the
                                    procedure labeled MATH_ROUTINE!

        JP    C,%2000              !Jump to location 2000(hex) if
                                    the carry flag is set!
```

2.4.5 Relative Address

Relative-address mode is implied by its instruction. It is used only by the Jump Relative (JR) and the Decrement and Jump If Nonzero (DJNZ) instructions and is the only mode available to these instructions. The operand, in this case, represents an offset that is added to the contents of the program counter to form the destination address (the program address where control is to be transferred). The original contents of the program counter is taken to be the address of the instruction byte following the JR or DJNZ instruction, while the offset value is an 8-bit signed value in the range -128 to 127.



The offset value can be expressed in two ways. In the first case, the programmer provides a specific offset in the form "$+n" where n is a constant expression in the range +129, -126 and $ represents the contents of the program counter at the start of the JR or DJNZ instruction (each instruction is two bytes in length).

```
        JR  OV, $+K      !Add value of constant K to program
                          counter and jump to new location if
                          overflow has occurred!
```

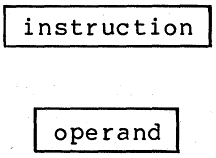
In the second method, the assembler calculates the offset. The programmer simply specifies an expression which evaluates to a number or a program label as in direct addressing. The address specified by the operand must be in the same module and section as the JR or DJNZ instruction and must be within the offset range (+127, -128).

```
        JR  OV, MATH_ERROR

        DJNZ R5, LOOP    !Decrement reg 5 and jump to LOOP if
                          the result is not zero!
```

The Jump (JPR) control instruction causes the assembler to produce a JR whenever possible. Refer to Section 5.2.6 for detailed information on the Jump optimization.

2.4.6 Immediate Data

Immediate data is an address mode for the purposes of this discussion. The operand value used by the instruction in this case is the value supplied in the operand field itself.



Immediate address mode is often used to load registers with their initial values. The Z8 is optimized for this function, providing several short immediate data instructions to reduce the byte count of programs.

Immediate data is preceded by the special character # and may be a constant (including character constants and symbols representing constants) or an expression as described in section 2.3.4. Remember, if a variable name is prefixed by #, the value used is the address represented by the variable, not the contents of the address. Immediate data values must be in the range -128 to 255 (that is, expressible in 8 bits), or an error message will be generated.

```
     LD  100, #%12                 !Load 12(hex) into reg 100!

     LD  COUNTER, #COUNT-50        !Load register named
                                    COUNTER with value of
                                    constant COUNT-50!

     LD POINTER, #ITEM             !Load register named
                                    POINTER with the value of the
                                    address of variable ITEM!
```

Two special operators are provided to ease the manipulation of 16-bit addresses: the HI operator gives the high-order byte, and LO gives the low-order byte of a 16-bit expression. Since HI and LO can only be used where the immediate addressing mode is applicable, the # character must precede them.

```
     LD  R6,#HI DLABEL             !Load bits 8-15 of 16-bit
                                    address of DLABEL into reg 6!

     LD  R7,#LO DLABEL             !Load bits 0-7 of 16-bit
                                    address of DLABEL into reg 7!
```

2.4.7 A Note on the Register Pointer

For the assembly-language programmer, dealing with working registers is an automatic procedure. The assembler determines the appropriate binary instruction code so that the specified working-register address can be formed at run time using the current value of the register pointer. The upper four bits of the memory address are taken from the register pointer; the lower four bits are taken from the instruction code, and together they specify the designated working register.

When programming in binary code, an additional possibility exists. When a full 8-bit register designator is required by the instruction format (register or register-pair address modes), a working register or working-register pair can be specified without knowing the current value of the register pointer. If the upper nibble of the register field is coded as a hexadecimal E, the lower nibble is interpreted as a working-register (E0-EF hex are not implemented register addresses). The final register address is formed at run time by replacing E with the register pointer value. This mechanism is handled automatically by the assembler whenever a working register designator is used.

The working-register mechanism allows sections of code using different register-pointer values to share a common procedure and pass parameters to it via the working registers.

Section 3
Assembly Language Instruction Set

3.1 Functional Summary

PLZ/ASM instructions can be divided functionally into the following eight groups:

- Load
- Arithmetic
- Logical
- Program Control (Branch)
- Bit Manipulation (Test)
- Block Transfer
- Rotate and Shift
- CPU Control

The following summary shows the instructions belonging to each group and the number of operands required for each, where "src" is the source operand, "dst" is the destination operand, and "cc" is a condition code.

Load Instructions

Instruction	Operands	Name of Instruction
CLR	dst	Clear
LD	dst,src	Load
LDC	dst,src	Load Constant
LDE	dst,src	Load External Data
POP	dst	Pop
PUSH	src	Push

Arithmetic Instructions

Instruction	Operands	Name of Instruction
ADC	dst,src	Add With Carry
ADD	dst,src	Add
CP	dst,src	Compare
DA	dst	Decimal Adjust
DEC	dst	Decrement
DECW	dst	Decrement Word
INC	dst	Increment
INCW	dst	Increment Word
SBC	dst,src	Subtract With Carry
SUB	dst,src	Subtract

Logical Instructions

Instruction	Operands	Name of Instruction
AND	dst,src	Logical And
COM	dst	Complement
OR	dst,src	Logical Or
XOR	dst,src	Logical Exclusive Or

Program-Control Instructions

Instruction	Operands	Name of Instruction
CALL	dst	Call Procedure
DJNZ	r,dst	Decrement and Jump If Nonzero
IRET		Interrupt Return
JP	cc,dst	Jump
JR	cc,dst	Jump Relative
RET		Return

Bit-Manipulation Instructions

Instruction	Operands	Name of Instruction
TCM	dst,src	Test Complement Under Mask
TM	dst,src	Test Under Mask

Block-Transfer Instruction

Instruction	Operands	Name of Instruction
LDCI	dst,src	Load Constant Autoincrement
LDEI	dst,src	Load External Data Autoincrement

Rotate and Shift Instructions

Instruction	Operand	Name of Instruction
RL	dst	Rotate Left
RLC	dst	Rotate Left Through Carry
RR	dst	Rotate Right
RRC	dst	Rotate Right Through Carry
SRA	dst	Shift Right Arithmetic
SWAP	dst	Swap Nibbles

CPU Control Instructions

Instruction	Operand	Name of Instruction
CCF		Complement Carry Flag
DI		Disable Interrupts
EI		Enable Interrupts
NOP		No Operation
RCF		Reset Carry Flag
SCF		Set Carry Flag
SRP	src	Set Register Pointer

3.2 Notation

Operands and status flags are represented by a notational shorthand in the detailed instruction descriptions that make up the rest of this chapter. The notation for operands (condition codes and address modes) and the actual operands they represent are as follows:

Notation	Address Mode	Actual Operand/Range
cc	Condition Code	See condition code list below
r	Working register only	Rn, where n = 0-15
R	Register or working register	reg=0-127, 240-255 or Rn as defined above
RR	Register pair or working register pair	reg, where reg is an even number in the range above or a variable whose address is even or RRp where p = 0,2,4,6...14
Ir	Indirect working register only	@Rn, where n = 0-15
IR	Indirect register or working register	@reg, where reg is as defined above or @Rn, as defined above
Irr	Indirect working register pair only	@RRp, where p = 0,2,4, 6...14
IRR	Indirect register pair or working register pair	@reg, where reg is an even number in the range defined above, or a variable whose address is even or @RRp as defined above

Notation	Address Mode	Actual Operand/Range
X	Indexed	reg(Rn), where reg and Rn are as defined abo e
DA	Direct Address	Program label or expression
RA	Relative Address	Program label or $ + or - offset, where the location addressed must be in the range +127, -128 bytes from the start of the next instruction
IM	Immediate	#data, where data is an expression

Status Flags are represented as follows:

C	Carry flag
Z	Zero flag
S	Sign flag
V	Overflow flag
D	Decimal adjust flag
H	Half carry flag

The condition codes and the flag settings they represent are:

Code	Meaning	Flag Settings	Binary
F	Always false	-	0000
(blank)	Always true	-	1000
Z	Zero	Z = 1	0110
NZ	Not zero	Z = 0	1110
C	Carry	C = 1	0111
NC	No carry	C = 0	1111
PL	Plus	S = 0	1101
MI	Minus	S = 1	0101
NE	Not equal	Z = 0	1110
EQ	Equal	Z = 1	0110
OV	Overflow	V = 1	0100
NOV	No overflow	V = 0	1100
GE	Greater than or equal	(S XOR V) = 0	1001
LT	Less than	(S XOR V) = 1	0001
GT	Greater than	(Z OR (S XOR V)) = 0	1010
LE	Less than or equal	(Z OR (S XOR V)) = 1	0010

Code	Meaning	Flag Settings	Binary
UGE	Unsigned greater than or equal	C=0	1111
ULT	Unsigned less than	C=1	0111
UGT	Unsigned greater than	((C=0) & (Z=0)) = 1	1011
ULE	Unsigned less than or equal	(C OR Z) = 1	0011

Note that some of the condition codes correspond to identical flag settings: Z-EQ, NZ-NE, C-ULT, NC-UGE.

3.3 Assembly-Language Instructions

In the remainder of this section, Z8 assembly-language instructions are described in detail in alphabetical order. Each description includes:

- The name of the instruction
- The binary instruction formats
- The operation performed by the instruction
- The status flags affected by the instruction
- The number of machine cycles used to execute the instruction
- The number of bytes used by the instruction
- A short example showing the use of the instruction

The description of each instruction's operation includes a shorthand summary. In addition to symbols already listed above, the following are also used in these summaries.

Symbol	Meaning
SP	Stack pointer (control registers 254-255)
PC	Program counter
FLAGS	Flag register (control register 252)
RP	Register pointer (control register 253)
IMR	Interrupt mask register (control register 251)

Assignment of a value is indicated by the symbol "<-". For example,

```
dst <- dst + src
```

indicates that the source data is added to the destination data and the result is stored in the destination location. The form addr(n) is used to refer to bit n of a given location. For example, dst(7) refers to bit 7 of the destination byte.

ADC dst,src

INSTRUCTION FORMATS:

	mode	dst	src
$\boxed{0001}\boxed{\text{mode}}$ $\boxed{\text{dst}}\boxed{\text{src}}$	0010	r	r
	0011	r	Ir
$\boxed{0001}\boxed{\text{mode}}$ $\boxed{\text{src}}$ $\boxed{\text{dst}}$	0100	R	R
	0101	R	IR
$\boxed{0001}\boxed{\text{mode}}$ $\boxed{\text{dst}}$ $\boxed{\text{src}}$	0110	R	IM
	0111	IR	IM

OPERATION: dst <- dst+src+C

The source byte, along with the setting of the carry flag, is added to the destination byte. The result is stored in the destination location.

FLAGS:

C:	Set if there was a carry from the most significant bit of the result; cleared otherwise
Z:	Set if the result is zero; cleared otherwise
V:	Set if arithmetic overflow occurred; cleared otherwis
S:	Set if the result is less than zero; cleared otherwis
H:	Set if a carry from the low-order nibble occurred
D:	Always reset to zero

BYTES AND CYCLES:

dst	src	bytes	cycles
r	r,Ir	2	6
other combinations		3	10

EXAMPLE: If the register named SUM contains %16, the carry flag is set to one, working register 10 contains %20 (32 decimal), and register 32 contains %10, the statement

```
ADC  SUM,@R10
```

will leave the value %27 in register SUM.

DD

dd

ADD dst,src

INSTRUCTION FORMATS:

			mode	dst	src
0000 | mode	dst | src		0010	r	r
			0011	r	Ir
0000 | mode	src	dst	0100	R	R
			0101	R	IR
0000 | mode	dst	src	0110	R	IM
			0111	IR	IM

OPERATION: dst <- dst+src

The source byte is added to the destination byte and the sum stored in the destination location.

FLAGS:

C:	Set if there was a carry from the most significant bit of the result; cleared otherwise
Z:	Set if the result is zero; cleared otherwise
V:	Set if arithmetic overflow occurred; cleared otherwise
S:	Set if the result is less than zero; cleared otherwise
H:	Set if a carry from the low-order nibble occurred
D:	Always reset to zero

BYTES AND CYCLES:

dst	src	bytes	cycles
r	r,Ir	2	6
other combinations		3	10

EXAMPLE: If the register named SUM contains %44 and the register named AUGEND contains %11, the statement

```
ADD  SUM, AUGEND
```

will leave the value %55 in register SUM.

AND dst,src

INSTRUCTION FORMATS:

```
                                                     mode   dst   src
[0101|mode]  [dst | src]                             0010   r     r
                                                     0011   r     Ir

[0101|mode]  [   src   ]  [   dst   ]                0100   R     R
                                                     0101   R     IR

[0101|mode]  [   dst   ]  [   src   ]                0110   R     IM
                                                     0111   IR    IM
```

OPERATION: dst <- dst AND src

The source byte is logically ANDed with the destination byte. The result is stored in the destination location. The AND operation results in a one bit being stored whenever the bits matched in the two operands are both ones.

FLAGS:		
	C:	Unaffected
	Z:	Set if result is zero; cleared otherwise
	V:	Always reset to zero
	S:	Set if the result bit 7 is set; cleared otherwise
	H:	Unaffected
	D:	Unaffected

BYTES AND CYCLES:

dst	src	bytes	cycles
r	r,Ir	2	6
other combinations		3	10

EXAMPLE: If the source operand is the immediate value %7B (01111011) and the register named TARGET contains %C3 (11000011), the statement

```
AND  TARGET, #%7B
```

will leave the value %43 (01000011) in register TARGET.

CALL

Call Procedure

CALL dst

INSTRUCTION FORMATS:

			dst
1101	0110	dst	DA
1101	0100	dst	IRR

OPERATION:

```
SP <- SP-2
@SP <- PC
PC <- dst
```

The current contents of the program counter (PC) are pushed onto the top of stack. (The program counter value used is the address of the first instruction byte following the CALL instruction.) The specified destination address is then loaded into the PC and points to the first instruction of a procedure.

At the end of the procedure a RETurn instruction can be used to return to the original program flow. RET pops the top of the stack back into the PC.

FLAGS: No flags affected.

BYTES AND CYCLES:

dst	bytes	cycles
DA	3	20
IRR	2	20

EXAMPLE: If the contents of the program counter are %1A47 and the contents of the stack pointer (control registers 254-5) are %3002, the statement

```
CALL %3521
```

causes the stack pointer to be decremented to %3000, %1A4A (the address following the instruction) is stored in external data memory %3000-%3001, and the program counter is loaded with %3521. The program counter now points to the address of the first statement in the procedure to be executed.

CCF

INSTRUCTION FORMAT:

1110	1111

OPERATION: C <- NOT C

The carry flag is complemented; if C=1, it is changed to C=0, and vice-versa.

FLAGS: C: Complemented
No other flags affected

BYTES AND CYCLES:

<u>bytes</u>	<u>cycles</u>
1	6

EXAMPLE: If the carry flag contains a zero, the statement

```
CCF
```

will change the zero to one.

CLR

Clear

CLR dst

INSTRUCTION FORMAT:

1011	mode		dst

mode	dst
0000	R
0001	IR

OPERATION: dst <- 0

The destination location is cleared to zero.

FLAGS: No flags affected.

BYTES AND CYCLES:

dst	bytes	cycles
R,IR	2	6

EXAMPLE: If working register 6 contains %AF, the statement

```
    CLR    R6
```

will leave the value 0 in that register.

COM dst

INSTRUCTION FORMAT:

0110	mode	dst

mode	dst
0000	R
0001	IR

OPERATION: dst <- NOT dst

The contents of the destination location are complemented (ones complement); all one bits are changed to zero, and vice-versa.

FLAGS:

C:	Unaffected
Z:	Set if result is zero; cleared otherwise
V:	Always reset to zero
S:	Set if result bit 7 is set; cleared otherwise
H:	Unaffected
D:	Unaffected

BYTES AND CYCLES:

dst	bytes	cycles
R,IR	2	6

EXAMPLE: If working register 8 contains %24 (00100100), the statement

```
COM  R8
```

will leave the value %DB (11011011) in that register.

CP

Compare

CP dst,src

INSTRUCTION FORMATS:

			mode	dst	src
1010 | mode	dst | src		0010	r	r
			0011	r	Ir
1010 | mode	src	dst	0100	R	R
			0101	R	IR
1010 | mode	dst	src	0110	R	IM
			0111	IR	IM

OPERATION: dst - src

The source byte is compared to (subtracted from) the destination byte, and the appropriate flags set accordingly. The contents of the destination byte are unaffected by the comparison.

FLAGS:	
C:	Cleared if there is a carry from the most significant bit of the result; set otherwise, indicating a borrow
Z:	Set if the result is zero; cleared otherwise
V:	Set if arithmetic overflow occurred; cleared otherwise
S:	Set if the result is negative; cleared otherwise
H:	Unaffected
D:	Unaffected

BYTES AND CYCLES:

dst	src	bytes	cycles
r	r,Ir	2	6
other combinations		3	10

EXAMPLE: If the register named TEST contains %63, working register 0 contains %30 (48 decimal), and register 48 contains %63, the statement

```
CP  TEST, @RO
```

sets (only) the Z flag. If this statement is followed by "JP EQ, true_routine", the jump will be taken.

DA dst

INSTRUCTION FORMAT:

0100	mode		dst

mode	dst
0000	R
0001	IR

OPERATION: dst <- DA dst

The destination byte is adjusted to form two 4-bit BCD digits following an addition or subtraction operation. For addition (ADD, ADC), or subtraction (SUB, SBC), the following table indicates the operation performed:

Instruction	Carry Before DA	Bits 4-7 Value (Hex)	H Flag Before DA	Bits 0-3 Value (Hex)	Number Added To Byte	Carry After DA
ADD ADC	0	0-9	0	0-9	00	0
	0	0-8	0	A-F	06	0
	0	0-9	1	0-3	06	0
	0	A-F	0	0-9	60	1
	0	9-F	0	A-F	66	1
	0	A-F	1	0-3	66	1
	1	0-2	0	0-9	60	1
	1	0-2	0	A-F	66	1
SUB SBC	0	0-9	0	0-9	00	0
	0	0-8	1	6-F	FA	0
	1	7-F	0	0-9	A0	1
	1	6-F	1	6-F	9A	1

The operation is undefined if the destination byte was not the result of a valid addition or subtraction of BCD digits.

FLAGS:

C:	Set if there was a carry from the most significant bit; cleared otherwise (see table above)
Z:	Set if the result is zero; cleared otherwise
V:	Undefined
S:	Set if the result bit 7 is set; cleared otherwise
H:	Unaffected
D:	Unaffected

BYTES AND CYCLES:

dst	bytes	cycles
R,IR	2	8

EXAMPLE: If addition is performed using the BCD values 15 and 27, the result should be 42. The sum is incorrect, however, when the binary representations are added in the destination location using standard binary arithmetic.

```
  0001  0101
+ 0010  0111
  0011  1100 = %3C
```

The DA statement adjusts this result so that the correct BCD representation is obtained.

```
  0011  1100
+ 0000  0110
  0100  0010 = 42
```

DEC dst

INSTRUCTION FORMAT:

0000	mode	dst

mode	dst
0000	R
0001	IR

OPERATION: dst <- dst-1

The destination byte's contents are decremented by one

FLAGS:

C:	Unaffected
Z:	Set if the result is zero; cleared otherwise
V:	Set if arithmetic overflow occurred; cleared otherwise
S:	Set if the result is negative; cleared otherwise
H:	Unaffected
D:	Unaffected

BYTES AND CYCLES:

dst	bytes	cycles
R,IR	2	6

EXAMPLE: If working register 10 contains %2A, the statement

```
DEC R10
```

will leave the value %29 in that register.

DECW

Decrement Word

DECW dst

INSTRUCTION FORMAT:

1000	mode	dst

mode	dst
0000	RR
0001	IR

OPERATION: dst <- dst-1

The contents of the destination location (which must be an even address) and the byte following that location are treated as a single 16-bit value which is decremented by one.

FLAGS:

C:	Unaffected
Z:	Set if the result is zero; cleared, otherwise
V:	Set if arithmetic overflow occurred; cleared otherwise
S:	Set if the result is negative; cleared otherwise
H:	Unaffected
D:	Unaffected

BYTES AND CYCLES:

dst	bytes	cycles
RR,IR	2	10

EXAMPLE: If working register 0 contains %30 (48 decimal) and registers 48-49 contain the value %FAF3, the statement

```
    DECW  @R0
```

will leave the value %FAF2 in registers 48 and 49.

DI

INSTRUCTION FORMAT:

$\boxed{1000}\boxed{1111}$

OPERATION: IMR(7) <- 0

Bit 7 of control register 251 (the Interrupt Mask Register) is reset to 0. All interrupts are disabled, although they remain <u>potentially</u> enabled.

FLAGS: No flags affected

BYTES AND CYCLES:

<u>bytes</u>	<u>cycles</u>
1	6

EXAMPLE: If control register 251 contains %8A (10001010, that is, interrupts IRQ1 and IRQ3 are enabled), the statement

```
DI
```

sets control register 251 to %0A and disables these interrupts.

DJNZ

Decrement and Jump if Nonzero

DJNZ r, dst

INSTRUCTION FORMAT:

r	1010	dst

reg	dst
0000 to 1111	RA

OPERATION: r <- r-1
If r <> 0, PC <- PC+dst

The working register being used as a counter is decremented. If the contents of the register are not zero after decrementing, the relative address is added to the program counter (PC) and control passes to the statement whose address is now in the PC. The range of the relative address is +127, -128, and the original value of the program counter is taken to be the address of the instruction byte following the DJNZ statement. When the working-register counter reaches zero, control falls through to the statement following DJNZ.

NOTE: The DJNZ instruction cannot be used on either I/O ports or control registers. The result of such an operation is undefined.

FLAGS: No flags affected

BYTES AND CYCLES:

bytes	cycles
2	12 if jump taken 10 if jump not taken

EXAMPLE: DJNZ is typically used to control a "loop" of instructions. In this example, 12 bytes are moved from one buffer area in the register file to another. The steps involved are:

1. Load 12 into the counter (working register 6)
2. Set up the loop to perform the moves
3. End the loop with DJNZ

```
       LD R6, #12              !Load Counter!

LOOP:  LD R9,OLDBUF (R6)       !Move one byte to!

       LD NEWBUF (R6), R9      !New location!

       DJNZ R6, LOOP           !Decrement and !
                               !Loop until counter = 0!
```

EI

Enable Interrupts

EI

INSTRUCTION FORMAT:

1001	1111

OPERATION: IMR(7) <- 1

Bit 7 of control register 251 (the Interrupt Mask Register) is set to 1. This allows any potentially enabled interrupts to become enabled.

FLAGS: No flags affected

BYTES AND CYCLES:

bytes	cycles
1	6

EXAMPLE: If control register 251 contains %0A (00001010, that is, interrupts IRQ1 and IRQ3 potentially enabled), the statement

```
EI
```

sets control register 251 to %8A (10001010) and enables these interrupts.

INC dst

INSTRUCTION FORMAT:

Format	mode	dst
| dst | 1110 |		r
| 0010 | mode | | dst |	0000	R
	0001	IR

OPERATION: dst <- dst+1

The destination byte's contents are incremented by one.

FLAGS:

C:	Unaffected
Z:	Set if result is zero; cleared otherwise
V:	Set if arithmetic overflow occurred; cleared otherwise
S:	Set if result is negative; cleared otherwise
H:	Unaffected
D:	Unaffected

BYTES AND CYCLES:

dst	bytes	cycles
r	1	6
R,IR	2	6

EXAMPLE: If working register 10 contains %2A, the statement

```
INC  R10
```

will leave the value %2B in that register.

INCW

Increment Word

INCW dst

INSTRUCTION FORMAT:

1010	mode	dst

mode	dst
0000	RR
0001	IR

OPERATION: dst <- dst+1

The contents of the destination location (which must be an even address) and the byte following that location are treated as a single 16-bit value which is incremented by one.

FLAGS:

C:	Unaffected
Z:	Set if result is zero; cleared otherwise
V:	Set if result is arithmetic overflow; cleared otherwise
S:	Set if result is negative; cleared otherwise
H:	Unaffected
D:	Unaffected

BYTES AND CYCLES:

dst	bytes	cycles
RR,IR	2	10

EXAMPLE: If working-register pair 0-1 contains the value %FAF3, the statement

```
INCW RR0
```

will leave the value %FAF4 in working-register pair 0-1.

IRET

INSTRUCTION FORMAT:

1011	1111

OPERATION:

```
FLAGS <- @SP
SP <- SP+1
PC <- @SP
SP <- SP+2
IMR(7) <- 1
```

This instruction is issued at the end of an interrupt service routine. It restores the flag register (control register 252) and the program counter. It also reenables any interrupts that are potentially enabled.

FLAGS: All flags are restored to original settings (before interrupt occurred).

BYTES AND CYCLES:

bytes	cycles
1	16

JP

Jump

JP cc,dst

INSTRUCTION FORMATS:

				cc	dst
Conditional	cc	1101	dst	0000 to 1111	DA
Unconditional	0011	0000	dst		IRR

OPERATION: If cc is true, PC <- dst

A conditional jump transfers program control to the designated location if the condition specified by "cc" is true. See section 3.1 for a list of condition codes.

The unconditional jump simply replaces the contents of the program counter with the contents of the specified register pair. Control then passes to the statement addressed by the program counter.

FLAGS: No flags affected

BYTES AND CYCLES:

dst	bytes	cycles
DA	3	12 if jump taken 10 if jump not taken
IRR	2	8

EXAMPLE: If the carry flag is set, the statement

```
JP  C, %1520
```

replaces the contents of the program counter with %1520 and transfers control to that location. Had the carry flag not been set, control would have fallen through to the statement following the JP.

JR

Jump Relative

JR cc,dst

INSTRUCTION FORMAT:

cc	1011	dst

cc	dst
0000 to 1111	RA

OPERATION: If cc is true, PC <- PC+dst

If the condition specified by "cc" is true, the relative address is added to the program counter and control passes to the statement whose address is now in the PC. (See Section 3.1 for a list of condition codes). The range of the relative address is +127,-128, and the original value of the program counter is taken to be the address of the first instruction byte following the JR statement.

FLAGS: No flags affected

BYTES AND CYCLES:

bytes	cycles
2	12 if jump taken 10 if jump not taken

EXAMPLES: If the result of the last arithmetic operation executed is negative, the following four statements (which occupy a total of seven bytes) are skipped with the statement

```
    JR  MI, $+9
```

If the result is not negative, execution continues with the statement following the JR. A short form of a jump to label L0 is

```
    JR  L0
```

where L0 must be within the allowed range. The condition code is "blank" in this case, and JR has the effect of an unconditional JP instruction.

LD

Load

LD dst,src

INSTRUCTION FORMATS:

					mode	dst	src
dst	mode	src			1100	r	IM
					1000	r	R
src	mode	dst			1001	R*	r
mode	0011	dst	src		1110	r	Ir
					1111	Ir	r
1110	mode	src		dst	0100	R	R
					0101	R	IR
1110	mode	dst		src	0110	R	IM
					0111	IR	IM
1111	0101	src		dst		IR	R
1100	0111	dst	x	src		r	X
1101	0111	src	x	dst		X	r

* In this instance only a full register address can be used.

OPERATION: dst <- src

The data specified or pointed to by the source operand is loaded into the destination location.

FLAGS: No flags affected

BYTES AND CYCLES:

dst	src	bytes	cycles
r	Ir,IM,R	2	6
Ir,R	r	2	6
r	X	3	10
X	r	3	10
other combinations		3	10

EXAMPLE: If working register 0 contains %0B (11 decimal) and working register 10 contains %83, the statement

```
LD  240(R0), R10
```

will load the value %83 into register 251 (240 + 11). Since this is the interrupt mask register, the Load statement has the effect of enabling IRQ0 and IRQ1. The contents of working register 10 are unaffected by the load.

LDC

Load Constant

LDC dst,src

INSTRUCTION FORMAT:

					dst	src
1100	0010	dst	src		r	Irr
1101	0010	src	dst		Irr	r

OPERATION: dst <- src

This instruction is used to load a byte constant from program memory into a working register, or vice-versa. The address of the program-memory location is specified by a working-register pair.

FLAGS: No flags affected

BYTES AND CYCLES:

bytes	cycles
2	12

EXAMPLE: If the working-register pair 6-7 contains %30A2 and program-memory location %30A2 contains the value %22, the statement

```
LDC  R2, @RR6
```

will load the value %22 into working register 2. The value of location %30A2 is unchanged by the load.

LDCI

Load Constant Autoincrement

LDCI dst,src

INSTRUCTION FORMAT:

				dst	src
1100	0011	dst	src	Ir	Irr
1101	0011	src	dst	Irr	Ir

OPERATION:

```
dst <- src
r <- r+1
rr <- rr+1
```

This instruction is used for block transfers of data between program memory and register memory. The address of the program-memory location is specified by a working-register pair, and the address of the register-memory location is specified by a working-register. The contents of the source location are loaded into the destination location. Both addresses are then incremented automatically.

FLAGS: No flags affected

BYTES AND CYCLES:

bytes	cycles
2	18

EXAMPLE: If the working-register pair 6-7 contains %30A2 and program-memory locations %30A2 and %30A3 contain %22BC, and if working register 2 contains %20 (32 decimal), the statement

```
LDCI  @R2,  @RR6
```

will load the value %22 into register 32. A second

```
LDCI  @R2,  @RR6
```

will load the value %BC into register 33.

LDE

Load External Data

LDE dst,src

INSTRUCTION FORMAT:

				dst	src
1000	0010	dst	src	r	Irr
1001	0010	src	dst	Irr	r

OPERATION: dst <- src

This instruction is used to load one byte from external data memory into a working register or vice-versa. The address of the external data-memory location is specified by a working-register pair.

FLAGS: No flags affected

BYTES AND CYCLES:

bytes	cycles
2	12

EXAMPLE: If the working-register pair 6-7 contains %404A and working register 2 contains %22, the statement

```
LDE   @RR6, R2
```

will load the value %22 into external data-memory location %404A.

LDEI

Load External Data Autoincrement

LDEI dst,src

INSTRUCTION FORMAT:

				dst	src
1000	0011	dst	src	Ir	Irr
1001	0011	src	dst	Irr	Ir

```
OPERATION:   dst <- src
             r <- r+1
             rr <- rr+1
```

This instruction is used for block transfers of data between external data memory and register memory. The address of the external data-memory location is specified by a working-register pair, and the address of the register-memory location is specified by a working register. The contents of the source location are loaded into the destination location. Both addresses are then incremented automatically.

FLAGS: No flags affected

BYTES AND CYCLES:

bytes	cycles
2	18

EXAMPLE: If the working register pair 6-7 contains %404A, working register 2 contains %22 (34 decimal), and registers 34-35 contain %ABC3, the statement

```
LDEI  @RR6, @R2
```

will load the value %AB into external data-memory location %404A. A second

```
LDEI  @RR6,@R2
```

will load the value %C3 into external location %404B.

NOP

No Operation

NOP

INSTRUCTION FORMAT:

1111	1111

OPERATION: No action is performed by this instruction. It is typically used for timing delays.

FLAGS: No flags affected

BYTES AND CYCLES:

bytes	cycles
1	6

OR dst,src

INSTRUCTION FORMATS:

			mode	dst	src
| 0100 | mode |	| dst | src |		0010	r	r
			0011	r	Ir
| 0100 | mode |	| src |	| dst |	0100	R	R
			0101	R	IR
| 0100 | mode |	| dst |	| src |	0110	R	IM
			0111I	R	IM

OPERATION: dst <- dst OR src

The source byte is logically ORed with the destination location. The OR operation results in a one bit being stored whenever either of the bits matched in the two operands is one.

FLAGS:		
	C:	Unaffected
	Z:	Set if result is zero; cleared otherwise
	V:	Always reset to zero
	S:	Set if the result bit 7 is set; cleared otherwise
	H:	Unaffected
	D:	Unaffected

BYTES AND CYCLES:

dst	src	bytes	cycles
r	r,Ir	2	6
other combinations		3	10

EXAMPLE: If the source operand is the immediate value %7B (01111011) and the register named TARGET contains %C3 (11000011), the statement

```
OR TARGET,#%7B
```

will leave the value %FB (11111011) in register TARGET.

POP

Pop

POP dst

INSTRUCTION FORMAT:

0101	mode	dst

mode	dst
0000	R
0001	IR

OPERATION:

```
dst <- @SP
SP <- SP+1
```

The contents of the location addressed by the stack pointer are loaded into the destination location. The stack pointer is then incremented automatically.

FLAGS: No flags affected

BYTES AND CYCLES:

bytes	cycles
2	10

EXAMPLE: If the stack pointer (control registers 254-255) contains %1000, external data-memory location %1000 contains %55, and working register 6 contains %22 (34 decimal), the statement

```
POP @R6
```

will load the value %55 into register 34. After the POP operation, the stack pointer will contain %1001.

PUSH

Push

PUSH src

INSTRUCTION FORMAT:

0111	mode		src

mode	src
0000	R
0001	IR

OPERATION:
```
SP <- SP-1
@SP <- src
```

When the PUSH instruction is encountered, the stack pointer is decremented immediately. The contents of the source location are then loaded into the location addressed by the decremented stack pointer.

FLAGS: No flags affected

BYTES AND CYCLES:

src	bytes	cycles	
R	2	10	(internal stack)
IR	2	12	(internal stack)
R	2	12	(external stack)
IR	2	14	(external stack)

EXAMPLE: If the stack pointer contains %1001, the statement

```
PUSH FLAGS
```

will store the contents of the register named FLAGS in location %1000. After the PUSH operation, the stack pointer will contain %1000.

RCF

Reset Carry Flag

RCF

INSTRUCTION FORMAT:

1100	1111

OPERATION: C <- 0

The carry flag is reset to zero, regardless of its previous content.

FLAGS: C: Reset to zero
No other flags affected

BYTES AND CYCLES:

bytes	cycles
1	6

RET

INSTRUCTION FORMAT:

1010	1111

OPERATION: PC <- @SP
SP <- SP+2

This instruction is normally used to return to the previously executing procedure at the end of a procedure entered by a CALL statement. The contents of the location addressed by the stack pointer are popped into the program counter. The next statement executed is that addressed by the new contents of the PC.

FLAGS: No flags affected

BYTES AND CYCLES:

bytes	cycles
1	14

EXAMPLE: If the program counter contains %35B4, the stack pointer contains %2000, external data-memory location %2000 contains %18, and location %2001 contains %B5, then the statement

```
RET
```

will leave the value %2002 in the stack pointer and the program counter will contain %18B5, the address of the next instruction.

RL dst

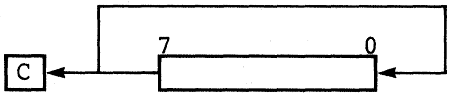
INSTRUCTION FORMAT:

1001	mode	dst

mode	dst
0000	R
0001	IR

OPERATION:
```
C <- dst(7)
dst(0) <- dst(7)
dst(n+1) <- dst(n) n=0-6
```

The contents of the destination byte are rotated left one bit position. The initial value of bit 7 is moved to the bit 0 position and also replaces the carry flag.



FLAGS:

C:	Set if there is a carry from the most significant bit (i.e., bit 7 was 1);
Z:	Set if the result is zero; cleared otherwise
V:	Set if arithmetic overflow occurred; cleared otherwise
S:	Set if the result bit 7 is set; cleared otherwise
H:	Unaffected
D:	Unaffected

BYTES AND CYCLES:

bytes	cycles
2	6

EXAMPLE: If the contents of the register named SHIFTER are %88 (10001000), the statement

```
RL   SHIFTER
```

will leave the value %11 (00010001) in that register. The carry flag will be set to one.

RLC dst

INSTRUCTION FORMAT:

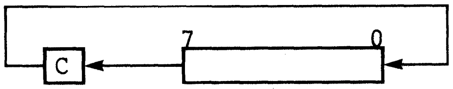
0001	mode	dst

mode	dst
0000	R
0001	IR

OPERATION:

```
dst(0) <- C
C <- dst(7)
dst(n+1) <- dst(n) n=0-6
```

The contents of the destination byte with the carry flag are rotated left one bit position. The initial value of bit 7 replaces the carry flag; the initial value of the carry flag replaces bit 0.



FLAGS:

C:	Set if there is a carry from the most significant bit (i.e., bit 7 was 1)
Z:	Set if the result is zero; cleared otherwise
V:	Set if arithmetic overflow occurred; cleared otherwis
S:	Set if the result bit 7 is set; cleared otherwise
H:	Unaffected
D:	Unaffected

BYTES AND CYCLES:

bytes	cycles
2	6

EXAMPLE: If the carry flag is reset (=0) and the register named SHIFTER contains %8F (10001111), the statement

```
RLC  SHIFTER
```

will set the carry flag (=1) and SHIFTER will contain %1E (00011110).

RR

Rotate Right

RR dst

INSTRUCTION FORMAT:

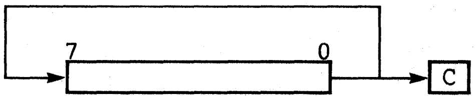
1110	mode	dst

mode	dst
0000	R
0001	IR

OPERATION:

```
C <- dst(0)
dst(7) <- dst(0)
dst(n) <- dst(n+1) n=0-6
```

The contents of the destination byte are rotated right one bit position. The initial value of bit 0 is moved to bit 7 and also replaces the carry flag.



FLAGS:

C:	Set if there is a carry from the least significant bit (i.e., bit 0 was 1)
Z:	Set if the result is zero; cleared otherwise
V:	Set if arithmetic overflow occurred; cleared otherwise
S:	Set if the result bit 7; cleared otherwise
H:	Unaffected
D:	Unaffected

BYTES AND CYCLES:

bytes	cycles
2	6

EXAMPLE: If the contents of working register 6 are %31 (00110001), the statement

```
RR   R6
```

will set the carry flag (=1) and leave the value %98 (10011000) in working register 6. Since bit 7 now equals 1, the sign flag will be set also.

RRC dst

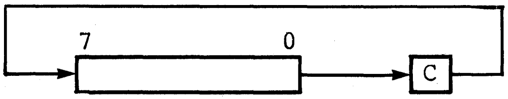
INSTRUCTION FORMAT:

1100	mode	dst

mode	dst
0000	R
0001	IR

OPERATION:
```
dst(7) <- C
C <- dst(0)
dst(n) <- dst(n+1) n=0-6
```

The contents of the destination byte with the carry flag are rotated right one bit position. The initial value of bit 0 replaces the carry flag; the initial value of the carry flag replaces bit 7.



FLAGS:

C:	Set if there is a carry from the least significant bit (i.e., bit 0 was 1)
Z:	Set if the result is zero; cleared otherwise
V:	Set if arithmetic overflow occurred; cleared otherwis
S:	Set if the result bit 7 is set; cleared otherwise
H:	Unaffected
D:	Unaffected

BYTES AND CYCLES:

bytes	cycles
2	6

EXAMPLE: If the contents of the register named SHIFTER are %DD (11011101) and the carry flag is reset (=0), the statement

```
RRC   SHIFTER
```

will set the carry flag (=1) and leave the value %6E (01101110) in the register.

SBC

Subtract With Carry

SBC dst,src

INSTRUCTION FORMATS:

					mode	dst	src
0011	mode	dst	src		0010	r	r
					0011	r	Ir
0011	mode	src		dst	0100	R	R
					0101	R	IR
0011	mode	dst		src	0110	R	IM
					0111	IR	IM

OPERATION: dst <- dst-src-C

The source byte, along with the setting of the carry flag, is subtracted from the destination byte. The result is stored in the destination location.

FLAGS:

C:	Cleared if there is a carry from the most significant bit of the result; set otherwise, indicating a "borrow"
Z:	Set if the result is zero; cleared otherwise
V:	Set if arithmetic overflow occurred; reset otherwise
S:	Set if the result is negative; cleared otherwise
H:	Set if a carry from the low-order nibble occurred
D:	Always set to one

BYTES AND CYCLES:

dst	src	bytes	cycles
r	r,Ir	2	6
other combinations		3	10

EXAMPLE: If the register named MINUEND contains %16, the carry flag is set to one, working register 10 contains %20 (32 decimal), and register 32 contains %05, the statement

```
    SBC  MINUEND, @R10
```

will leave the value %10 in register MINUEND.

SCF

INSTRUCTION FORMAT:

1101	1111

OPERATION: C <- 1

The carry flag is set to one, regardless of its previous contents.

FLAGS: C: Set to one
No other flags affected

BYTES AND CYCLES:

bytes	cycles
1	6

SRA

Shift Right Arithmetic

SRA dst

INSTRUCTION FORMAT:

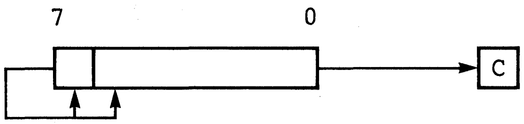
1101	mode	dst

mode	dst
0000	R
0001	IR

OPERATION:

```
dst(7) <- dst(7)
C <- dst(0)
dst(n) <- dst(n+1) n=0-6
```

An arithmetic shift right one bit position is performed on the destination byte. Bit 0 replaces the carry flag. Bit 7 (the sign bit) is unchanged, but its value is also carried into bit position 6.



FLAGS:

C:	Set if there is a carry from the least significant bit (i.e., bit 0 was 1)
Z:	Set if the result is zero; cleared otherwise
V:	Always reset to zero
S:	Set if the result is negative; cleared otherwise
H:	Unaffected
D:	Unaffected

BYTES AND CYCLES:

bytes	cycles
2	6

EXAMPLE: If the register named SHIFTER contains %B8 (10111000), the statement

```
SRA  SHIFTER
```

will reset the carry flag (=0) and leave the value %DC (11011100) in register SHIFTER. The sign flag will be set also (bit 7=1).

SRP src

INSTRUCTION FORMAT:

0011	0001	src

src
IM

OPERATION: RP <- src

The specified value is loaded into bits 4-7 of the register pointer (control register 253). Bits 0-3 of the register pointer are always set to zero. The source data (with bits 0-3 forced to zero) is the starting address of a working-register. The working-register group starting addresses are:

Hex	Decimal
%00	0
%10	16
%20	32
%30	48
%40	64
%50	80
%60	96
%70	112
%F0	240 (control registers)

Values in the range %80-E0 are invalid. See Figure 1-2.

FLAGS: No flags affected

BYTES AND CYCLES:

bytes	cycles
2	6

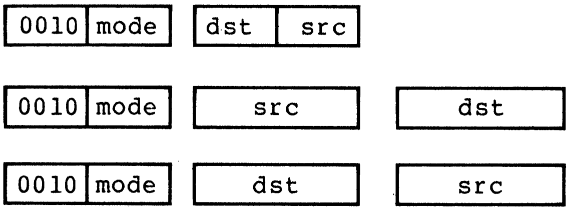
EXAMPLE: Assume the register pointer currently addresses the control register group and the program has just entered an interrupt service routine. The statement

```
    SRP  #%70
```

saves the contents of the control registers by setting the register pointer to %70 (01110000), or 112 decimal. Any reference to working registers in the interrupt routine will point to registers 112-127.

SUB dst,src

INSTRUCTION FORMATS:



mode	dst	src
0010	r	r
0011	r	Ir
0100	R	R
0101	R	IR
0110	R	IM
0111	IR	IM

OPERATION: dst <- dst - src

The source byte is subtracted from the destination byte and the result is held in the destination location.

FLAGS:

C:	Cleared if there is a carry from the most significant bit of the result; set otherwise, indicating a "borrow
Z:	Set if the result is zero; cleared otherwise
V:	Set if arithmetic overflow occurred; cleared otherwise
S:	Set if the result is less than zero; cleared otherwise
H:	Set if a carry from the low-order nibble occurred
D:	Always set to one

BYTES AND CYCLES:

dst	src	bytes	cycles
r	r,Ir	2	6
other combinations		3	10

EXAMPLE: If the register named MINUEND contains %29, the statement

```
SUB MINUEND, #%11
```

will leave the value %18 in the register.

SWAP

Swap Nibbles

SWAP dst

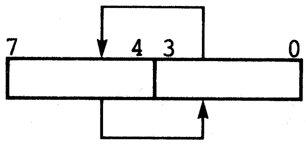
INSTRUCTION FORMAT:

1111	mode	dst

mode	dst
0000	R
0001	IR

OPERATION: dst(0-3) <-> dst(4-7)

The contents of the lower four bits and upper four bits of the destination byte are swapped.



C:	Undefined
Z:	Set if the result is zero; cleared otherwise
V:	Undefined
S:	Set if the result bit 7 is set; cleared otherwise
H:	Unaffected
D:	Unaffected

BYTES AND CYCLES:

bytes	cycles
2	8

EXAMPLE: Suppose the register named BCD_Operands contains %B3 (10110011). The statement

```
SWAP  BCD_Operands
```

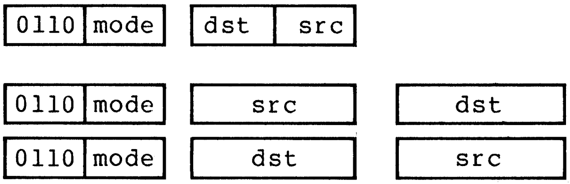
will leave the value %3B (00111011) in the register.

TCM

Test Complement Under Mask

TCM dst,src

INSTRUCTION FORMATS:



mode	dst	src
0010	r	r
0011	r	Ir
0100	R	R
0101	R	IR
0110	R	IM
0111	IR	IM

OPERATION: NOT dst AND src

This instruction tests selected bits in the destination byte for a logical "1" value. The bits to be tested are specified by setting a one bit in the corresponding position of the source byte (mask). The TCM statement complements the destination byte, which is then ANDed with the source mask. The Zero (Z) flag can then be checked to determine the result. When the TCM operation is complete, the destination location still contains its original value.

FLAGS:

C:	Unaffected
Z:	Set if the result is zero; cleared otherwise
V:	Always reset to zero
S:	Set if the result bit 7 is set; cleared otherwise
H:	Unaffected
D:	Unaffected

BYTES AND CYCLES:

dst	src	bytes	cycles
r	r,Ir	2	6
other combinations		3	10

EXAMPLE: If the register named TESTER contains %F6 (11110110) and the register named MASK contains %06 (00000110), that is, bits 1 and 2 are being tested for a one value, the statement

```
    TCM  TESTER, MASK
```

will complement TESTER (to 00001001) and then do a logical AND with register MASK, resulting in %00. A subsequent test of the Z flag,

```
    JP  Z,plabel
```

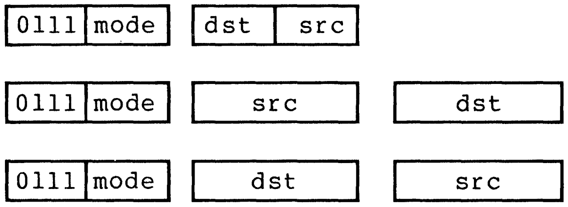
will cause a transfer of program control. At the end of this sequence, TESTER still contains %F6.

TM

Test Under Mask

TM dst,src

INSTRUCTION FORMATS:



mode	dst	src
0010	r	r
0011	r	Ir
0100	R	R
0101	R	IR
0110	R	IM
0111	IR	IM

OPERATION: dst AND src

This instruction tests selected bits in the destination byte for a logical "0" value. The bits to be tested are specified by setting a one bit in the corresponding position of the source byte (mask), which is ANDed with the destination byte. The Z (zero) flag can be checked to determine the result. When the TM operation is complete, the destination location still contains its original value.

FLAGS:

C:	Unaffected
Z:	Set if the result is zero; cleared otherwise
V:	Always reset to zero
S:	Set if the result bit 7 is set; cleared otherwise
H:	Unaffected
D:	Unaffected

BYTES AND CYCLES:

dst	src	bytes	cycles
r	r,Ir	2	6
other combinations		3	10

EXAMPLE: If the register named TESTER contains %F6 (11110110) and the register named MASK contains %06 (00000110), that is, bits 1 and 2 are being tested for a zero value, the statement

```
TM  TESTER, MASK
```

will result in the value %06 (00000110). A subsequent test for nonzero

```
JP  NZ, plabel
```

will cause a transfer of program control. At the end of this sequence, TESTER still contains %F6.

XOR dst,src

INSTRUCTION FORMATS:

			mode	dst	src
1011 | mode	dst | src		0010	r	r
			0011	r	Ir
1011 | mode	src	dst	0100	R	R
			0101	R	IR
1011 | mode	dst	src	0110	R	IM
			0111	IR	IM

OPERATION: dst <- dst XOR src

The source byte is logically EXCLUSIVE ORed with the destination byte and the result stored in the destination location. The EXCLUSIVE OR operation results in a one bit being stored whenever the two bits matched in the operands are different.

FLAGS:

C:	Unaffected
Z:	Set if the result is zero; cleared otherwise
V:	Always reset to zero
S:	Set if the result bit 7 is set; cleared otherwise
H:	Unaffected
D:	Unaffected

BYTES AND CYCLES:

dst	src	bytes	cycles
r	r,Ir	2	6
other combinations		3	10

EXAMPLE: If the source operand is the immediate value %7B (01111011) and the register named TARGET contains %C3 (11000011), the statement

```
XOR  TARGET, #%7B
```

will leave the value %B8 (10111000) in the register.

Section 4
Structuring A Z8 Program

4.1 Introduction

This section introduces the high-level PLZ/ASM statements described in Section 5. The structuring of programs and the concepts of module linkage and relocation are discussed.

4.2 Program Structure

4.2.1 Modules

A Z8 PLZ/ASM program consists of one or more separately-coded and assembled modules. These modules are combined into an executable program using the module linkage and relocation facilities of an operating system such as Z80 RIO. One of the modules should include a main program; that is, a GLOBAL procedure whose name is supplied to the linking facility as the program's entry point.

PLZ/ASM modules are made up of high-level and assembly-language statements that either declare or define data or perform some action. The assembly-language statements described in Section 3 are action statements. In general, data definition equates an identifier with a fixed value or type, whereas data declaration equates an identifier with a variable and associates a type with it.

Data, labels, or procedures can be shared between modules by declaring them to be GLOBAL in one module and declaring them as EXTERNAL in other modules that reference them. Data, labels, or procedures declared as INTERNAL to a module may be referenced only within that module. See Section 4.2.5 for more explanation of the scope of these objects.

The following example is the skeleton of a module showing two procedures, the second of which is a main program.

```
bubble_sort MODULE

   CONSTANT
     false  := 0               !Constant definition!
     true   := 1

   EXTERNAL
     list ARRAY [10 WORD]      !External variables!
     limit BYTE                !Declared in another module!

   INTERNAL                    !First procedure!
     sort                      !Procedure name!
       PROCEDURE
           .
           .
           .
     END sort                  !End first procedure!

   GLOBAL                      !Second procedure!
     main
       PROCEDURE
       ENTRY
         LD limit, #9
         CALL sort
         RET
     END main                  !End second procedure!

END bubble_sort                !End of module!
```

4.2.2 Procedures

A procedure declaration defines an executable portion of a module (including both action and data statements). It also associates an identifier with that block of code so that it can be activated by the assembly-language CALL statement. (See the CALL sort statement in the preceding example).

Every procedure declaration also has a scope associated with it. It can be:

- GLOBAL (the procedure can be called from other modules);
- INTERNAL (it can be called only within the current module);
- EXTERNAL (it is called from the current module, but is declared as GLOBAL in another module).

A procedure declaration can include LOCAL variables or label declarations as shown in the following example. LOCAL variables and labels can be referenced only within the procedure in which they are declared.

```
INTERNAL                    !Procedure scope!
  sort                      !Procedure name!
    PROCEDURE
      LOCAL
        i, j BYTE           ! i, j, and switch are!
        switch BYTE         !local variables!
    ENTRY
    DO                      !DO loop introduced!
     .
     .                      !(action statements)!
     .
    OD                      !End DO loop!
  END sort                  !End of procedure!
```

Note in this example that the keyword ENTRY is used to separate the declarations of LOCAL variables from the executable action statements making up the body of the procedure. The example also carries the previous bubble-sort module example to the first inner level of detail--the procedure. At the same time it introduces another program structuring element--the DO loop.

4.2.3 DO Loops

The description of the DJNZ instruction in Section 3 pointed out that it is used primarily for loop control. DO loops perform a similar function, but in a more structured, high-level manner. The statements between the keywords DO and OD are executed repeatedly until control is diverted through a loop control statement (REPEAT, EXIT, or one of the assembly-language branch statements). The EXIT statement causes execution to continue at the first statement following the innermost DO loop containing the EXIT, while the REPEAT statement causes execution to continue at the first statement of the innermost DO loop containing the REPEAT.

A DO statement does not introduce a new scope. It contains only action statements.

The following example enlarges upon the original bubble_sort skeleton and also introduces the IF statement.

```
SRP #%10                     !Use reg. set %10 - %1F!
DO
  LD   switch, #false
  CLR  i
  DO                         !Note nested DO loop!
    CP  i, limit
    IF UGE THEN EXIT FI      !IF statement at inner level!
     .
     .
     .
    INC  i
  OD                         !End nested DO loop!
CPB  switch, #false
IF . . . FI                  !IF statement at outer level!
OD                           !End DO loop!
```

Note in this example that DO loops can contain IF statements or other DO loops. IF statements can also have DO loops or other IF statements nested within them.

4.2.4 IF Statements

The general form of the IF statement is:

```
IF condition-code  THEN  action1
        ELSE  action2
FI
```

The IF statement specifies that all statements between THEN and ELSE (action1) be executed if the condition specified is true. Otherwise, the statements between ELSE and FI (action2) are executed. The ELSE clause can be omitted, in which case no action is performed when the specified condition is false. Like the DO loop, the IF statement contains only executable statements.

The condition codes specified are the same as for the assembly-language JP and JR instructions (Section 3.2). As in the case of DO loops versus the DJNZ instruction, one can set up conditional statements using only assembly-language instructions, but the high-level IF statement may lead to a more structured program.

The following example shows a brief IF statement followed by a longer one.

```
        IF  UGE  THEN  EXIT  FI
               .
               .
               .
        IF UGT THEN
           LD    switch, #true
           LD    list(R2), R6
           LD    list(R2), R4
           LD    list+1(R2), R7
           LD    list(R3), R4
           LD    list+1(R3), R5
        FI
```

4.2.5 Scope

The scope of a variable, label, or procedure, refers to that portion of the program in which the object is recognized and handled in keeping with its declaration. When declaring an object for use solely within the current module or procedure, its use elsewhere is of no concern. Even though the same identifier is used in another scope, it refers to a different object.

In general, a scope can be an entire executable program, a module, or a procedure. An identifier is accessible in a scope S if it is either

- Declared in S or in the module containing S, or

- Declared EXTERNAL in the module containing S.

New identifiers are declared by their appearance in a variable, procedure, constant, or type declaration, or by their use as a label within the module. If a label identifier is not declared explicitly, it is assumed to have an INTERNAL scope and thus is accessible throughout the module in which it is defined. The scope of a label identifier can be explicitly declared to be either GLOBAL, INTERNAL, or LOCAL. In addition, a special form of statement labels is discussed in Section 5.3.5, which always have a LOCAL scope and therefore cannot be declared explicitly.

New identifiers introduced in a variable, procedure, or label declaration are accessible only within the newly-established scope as determined from the declaration class specified-GLOBAL, EXTERNAL, INTERNAL or LOCAL. A new identifier cannot be the same as an identifier already accessible in the current scope.

For variable and label declarations, GLOBAL specifies that the variable is declared in the current module but can be used in other modules. GLOBAL variables have a scope of the entire executable program. EXTERNAL specifies that the variable is used in the current module but declared as GLOBAL in another module. INTERNAL specifies that the variable is declared in the current module and is accessible only within the current module. LOCAL specifies that the variable can be accessed only inside the procedure in which it is declared. GLOBAL, EXTERNAL, and INTERNAL can only be specified at the module level; LOCAL can only be used at the procedure level.

For variables declared with type RECORD, the scope of a record field is the module or procedure in which it is defined. Note that this implies that record fields must have names that are unique within their scope.

Procedure declarations must be either GLOBAL, EXTERNAL, or INTERNAL, as defined for variables above. The name given to the procedure as part of the procedure declaration is recognized in the entire scope of the enclosing module.

Constant identifiers are defined using the CONSTANT class, and can be defined only at the module level. The scope of a constant identifier is the module scope, and it cannot be used outside the module unless it is redefined.

Type identifiers are defined using the TYPE class. They can be defined only at the module level and are recognized only within the scope of the module.

An identifier used in a scope and not declared in that scope is said to be free in that scope. Any identifier that is free in the scope of a procedure must be declared in the enclosing module. Procedures do not explicitly import variables into their scope as modules do (via the EXTERNAL declaration).

4.2.6 Summary

The preceding sections were intended only as an introduction to program structuring. The detailed formats of the declarations used to define the program structure are listed in Section 5. Even without those details, however, the following module should be comprehensible at this point. If any statement looks foreign, reread the part of Section 3 or 4 that explains its operation, with the exception of data types and their declarations which are explained in Section 5.3.

An example of a complete bubble_sort module for sorting a 10-word array is as follows:

```
bubble_sort MODULE                                  !Module declaration!

  CONSTANT                                          !Constant declarations!
    false  := 0
    true   := 1
  EXTERNAL
    list ARRAY [10 WORD]
    limit BYTE

INTERNAL                                            !Procedure declaration!
  sort
    PROCEDURE
      LOCAL                                         !Local variable declaration!
        i, j BYTE                                   !Pointers to array words!
        switch BYTE                                 !Loop control switch!
      ENTRY                                         !Begin executable part!
      SRP #%10                                      !Use reg. set %10 - %1F!
      DO
        LD   switch, #false                         !Initialize switch!
        CLR  i                                      !Clear array Pointer!
        DO
          CP  i, limit                              !Done?!
          IF UGE THEN EXIT FI
          LD j, i                                   !Initialize pointer j!
          INC j                                     !j = i+1!
          LD R2, i                                  !IF list[i] > list [j]!
          ADD R2, R2                                !Double for word index!
          LD  R4, list(R2)                          !Load array words!
          LD  R5, list+1(R2)
          LD  R3, j
          ADD R3, R3
          LD  R6, list(R3)
          LD  R7, list+1(R3)
          CP  R4, R6                                !Compare words!
          IF EQ THEN CP R5, R7 FI
          IF UGT THEN                               !Exchange to bubble largest
            LD switch, #true                        !number to top of array!
            LD list(R2), R6
            LD list+1(R2), R7
            LD list(R3), R4
            LD list+1(R3), R5
          FI
          INC i
        OD                                          !End nested DO loop!
        CP switch, #false                           !Test switch!
        IF EQ THEN RET FI
      OD                                            !End outer DO loop!
    END sort                                        !End of procedure!
  GLOBAL                                      !New procedure declaration!
    main                                      !Program entry procedure!
      PROCEDURE
      ENTRY                                   !No LOCALS!
        LD limit, #9                          !Initialize loop control!
        CALL sort                             !Call sort procedure!
        RET
    END main                                  !End of main procedure!
END bubble_sort                               !End of module!
```

4.3 Relocatability

The Z8 PLZ/ASM assembler produces relocatable object modules. Essentially, this frees the programmer from memory management concerns during program development (since object code can be relocated in memory) and also allows programs to be developed in modules whose addresses are resolved automatically when the modules are linked.

Modular program development offers numerous advantages to the developer. Complex programs can be divided into several smaller tasks and assigned to a development team, should schedule constraints require quick completion. An entire program need not be delayed while one module is awaiting development, since modules can be separately tested and assembled. A change affecting a single module will not have a ripple effect through the entire program and will require reassembly of the affected module only. In short, several simple programs are generally easier to write, test, and debug than one large, complex program.

The relocatability feature of the Z8 assembler is supported by the $ABS, $REL, and $SECTION assembler directives. These directives determine whether programs are assembled in relocatable mode or not, and where data and action statements are to be loaded into memory. An operating system program called a Linker relocates object modules and resolves intermodule references.

These assembler directives are summarized below. Other directives are listed in Appendix C.

4.3.1 Sections

In addition to the logical structuring provided by modules and procedures, it is possible to divide a program into sections which can be mapped into various areas of memory when the program is linked or loaded for execution. For example, the programmer can choose to group a set of data structures and the procedures which manipulate them together in the same module. But it can also be desirable to physically separate the object code for the procedures from the data in a system where read-only memory is used for the procedures and read/write memory is used for the data.

The assembler allows a program to be arbitrarily divided into named sections. Each section must be allocated to either the PROGRAM, DATA, or REGISTER address space. A single module can contain several sections, each of which is allocated to a different area in memory. Alternatively, the portions of a single section can be spread through several modules and the portions will be automatically combined into a single area by the linker. Sections provide the programmer with complete control over the mapping of a program into the address spaces of the Z8 processor.

Usually the full generality of arbitrary sections is not needed for a particular application. To simplify the programmer's job and to allow labels to be declared without conflicts, the assembly source code, by default, is sectioned into three areas:

- The PROGRAM section for program memory;
- The REGISTER section for register memory;
- The DATA section for external data memory.

The assembler maps all variable declaration into the register section and all procedure declarations into the program section. The $SECTION assembler directive can be used to override this default, however, and cause the code following it to be mapped into the memory area it designates. It has the format

```
        $SECTION  area
```

where area is either PROGRAM, REGISTER, or DATA.

The $SECTION assembler directive can also indicate user-defined sections. This directive causes the code following it to be associated with a symbolic identifier and to be mapped into one of the memory areas. It has the format

```
        $SECTION identifier  area
```

To return to the default memory mapping scheme, use the $SDEFAULT assembler directive.

NOTE: REGISTER memory cannot be initialized. This means that if default memory allocation is used, initial values cannot be provided as part of variable declarations (see Section

5.3.4). This also means that action statements cannot appear in the register SECTION.

Regardless of how many times the $SECTION directive is specified, the assembler produces one contiguous module consisting of the number of user defined sections, and possibly the three default sections (PROGRAM, REGISTER, DATA) as well. The linker program then combines similar sections from different modules and relocates them according to their respective sections. The linker will generate an error if two sections of the same name do not associate with the same memory space.

4.3.2 Location Counter Control

The assembler keeps track of the location of the current statement with a location counter, just as an executing program does with its program counter. There is a location counter associated with each section in a program. The counter value represents a 16-bit offset that the current section. The offset can be either an absolute value, or it can represent a relocatable value which can be adjusted depending on where the module's portion of the section is finally allocated at link or load time. Relocatable and absolute portions of a section can be specified in the same module or program.

If the $ABS assembler directive has been specified, the location counter reflects the absolute location of the current statement; if $REL has been specified, the counter reflects the relocatable offset of the statement. If neither is specified, the counter defaults to relocatable offset 0 at the beginning of a module. The location counter symbol, $, may be used in any expression, and represents the address of the first byte of the current instruction.

For the majority of programming tasks, one need not specify either $ABS or $REL in the source program. Program location can be carried out more appropriately at link time or load time. If one desires complete control over program location at assembly time, however, the $ABS directive will force that part of the program to reside at a specific location.

4.3.3 Modes of Arithmetic Expressions

All arithmetic expressions have one of three modes associated with them: absolute, relocatable, or external.

An absolute expression consists of one or more constants, constant identifiers, or absolute labels combined with arithmetic or logical operators. The difference between two relocatable expressions is also considered to be absolute.

```
    JP     IRQ_VECTOR              !Where IRQ_VECTOR is an absolute
                                    label!
    ADD    R1, #K*3                !Where K is a constant identifier!
```

A relocatable expression is exactly one identifier subject to relocation after assembly. The expression can be extended by adding or subtracting an absolute expression. Plus and minus are the only operators allowed, however.

```
    JP     Z, LOOP+2               !Where LOOP is a relocatable
                                    label!
```

An external expression is exactly one external identifier, possibly extended by adding or subtracting an absolute expression. An external identifier is used in the current module but defined in another module (Section 4.2.5). The value of an external identifier is not known until the modules are linked.

```
    LD     T1, timer_count         !External if timer_count was
                                    defined outside current module!
```

In the following summary, "AB" stands for an absolute value or expression, "RE" stands for a relocatable value or expression, and "EX" is an external identifier or expression. "operator" is one of the standard arithmetic or logical operators (+, -, *, LOR, LAND, etc.).

"AB" is defined as one of the following:

- an absolute identifier
- AB "operator" AB
- +AB, -AB, or LNOT AB
- RE - RE

"RE" is defined as one of the following:

- a relocatable identifier
- RE + AB
- AB + RE
- RE - AB
- +RE

"EX" is defined as one of the following:

- an external identifier
- EX + AB
- AB + EX
- EX - AB
- +EX

Certain mode combinations are not permitted in PLZ/ASM. For example, a relocatable expression that does not result in simple relocation is invalid. Simple relocation means that only a single relocation factor need be added to a relocatable value when the assembled program is relocated.

```
    JP    Z, LOOP1 + 8          !VALID -- relocation factor is
                                 added only once!

    JP    Z, LOOP1 + LOOP2      !INVALID -- relocation factor
                                 would have to be added twice!
```

In general, the second example would be invalid no matter which arithmetic or logical operator was used to combine LOOP1 and LOOP2. The one exception is subtraction. Suppose LOOP2 is the label of the first statement following a procedure named LOOP1. The statement

```
    LD    R5, #(LOOP2 - LOOP1)
```

could be used to find the length of the LOOP1 procedure. The difference between two relocatable labels that must be in the same section is always absolute, regardless of where the module is relocated.

Other invalid mode combinations are:

- A relocatable expression multiplied or divided by an absolute expression:

```
        JP    Z, LOOP1*4    !INVALID!
```

- A relocatable expression subtracted from an absolute expression (although the reverse is allowed):

```
        DJNZ  R5, LOOP1 - 8        !VALID!
        DJNZ  R5, 8 - LOOP1        !INVALID!
```

- An external expression combined with a relocatable expression, and vice-versa:

```
        ADD  R6, EXTERNAL_NUMB + LOOP1      !INVALID!
        JP   Z, LOOP1 - EXTERNAL_NUMB       !INVALID!
```

Section 5
PLZ/ASM High-Level Statements

5.1 Z8 Source Program Statements

The majority of code in a Z8 PLZ/ASM program will normally be the assembly-language instructions described in Section 3. Typically, the source program will also include some high-level statements and assembler directives.

High-level statements perform two basic functions:

- They introduce program structures (modules, procedures, DO loops, and IF statements);
- They declare and define data.

Assembler directives control the mode of assembly (absolute or relocatable), determine where object code is to be stored in memory, and specify the form of assembler output. These directives are embedded in the source program and are always preceded by a dollar sign ($) (see Appendix C).

In the descriptions of high-level statements in this section the following notational conventions are used:

- Keywords are shown as all capital letters: MODULE
- Parameters shown in lowercase letters represent items to be replaced by actual data or names: module_identifier
- Optional items are enclosed in square brackets: [local_declaration]
- Possible repetition of an item is indicated by appending a + (to signify one or more repetitions) or an * (to signify zero or more repetitions) to the item: declaration*
- Other special characters shown in statement and command formats such as :=, (), and [] will be enclosed in single quotes and must be written as shown. The special symbol := means "is defined as" or "is assigned".

For example:

```
        CONSTANT
          constant_identifier ':=' constant_expression

        RECORD  '[' identifier+ type ']'
```

5.2 Program Structuring Statements

5.2.1 Module Declaration

A Z8 PLZ/ASM program module consists of a sequence of data and procedure declarations. These declarations are bounded by the module declaration statement and the end-of-module statement. The format of a module declaration is:

```
        module_identifier MODULE
          declaration*
        END module_identifier
```

where

module_identifier	conforms to the rules for identifiers (Section 2.2.1).
declaration	is a data or procedure declaration (Sections 5.2.2 and 5.3).

Example:

```
        sine_computation MODULE
          .
          .
          .
        END sine_computation
```

5.2.2 Procedure Declaration

A procedure declaration defines an executable part of a program and associates an identifier with it so that it can be activated by the assembly-language CALL statement.

The procedure heading specifies the identifier naming the procedure. This identifier labels the first instruction in the procedure, and can be used as any other program label. The scope of the procedure identifier can be either GLOBAL, INTERNAL, or EXTERNAL. If the procedure is declared EXTERNAL, then only the procedure identifier is given, since the actual definition of the procedure occurs in some other module. A procedure declaration can also include local variable declarations. These variables are recognized only within the procedure in which they are declared.

The format of the procedure declaration is:

```
        procedure_identifier PROCEDURE
          [LOCAL
            [variable_identifier+ type]*]*
          [ENTRY
            action_statement*]
        END procedure_identifier
```

where

identifier	conforms to the rules for identifiers (Section 2.2.1).
type	is BYTE, SHORT_INTEGER, INTEGER, WORD, LONG, LONG_INTEGER, LABEL, ARRAY, RECORD or a user_defined type (Sections 5.3.2 through 5.3.4).
action_statement	is an assembly-language, DO, IF, REPEAT, or EXIT statement.

The keyword ENTRY is used to separate the local variable declarations from the executable part of the procedure, and must be used whenever any action statements are specified.

Example:

```
        EXTERNAL
          print PROCEDURE

        GLOBAL
          suml BYTE

          add_routine PROCEDURE
            LOCAL
              sum  BYTE
            ENTRY
              LD sum, addend
              ADD sum, augend
              SUB suml, sum
              CALL print
              RET
          END add_routine
```

Note that the RET instruction precedes the END statement. If RET is not present, control will fall through to the statement following the END statement, although future versions of the assembler will not necessarily support this.

5.2.3 DO Statement

DO loops provide a framework for performing actions repetitively. The statements between the DO and OD keywords are executed repeatedly until control is diverted through a loop control statement.

The only way the execution flow of a DO loop can be diverted is by encountering an assembly-language branch instruction (DJNZ, JP, JR, CALL, CALR, RET, or IRET) or an EXIT or REPEAT statement (described below).

The format of the DO loop is:

```
        [label]*
        DO
          action_statement*
        OD
```

where

label	conforms to the rules for labels (Sections 2.2.1 and 5.3.5) and is used to identify the DO block for use with multilevel EXIT and REPEAT statements.
action_statement	is an assembly-language, DO, IF, REPEAT, or EXIT statement.

The assembler automatically inserts a single unconditional jump instruction at the OD keyword which branches back to the DO keyword. Either a JR or JP is generated depending on the range of the loop; JR is used whenever possible.

The <u>EXIT statement</u> causes execution to continue at the first statement following the innermost DO...OD block containing the EXIT. The EXIT statement may be further qualified by a label indicating a specific DO...OD block from which to exit. Its format is:

```
        EXIT [FROM label]
```

where

label	conforms to the rules for labels (Sections 2.2.1 and 5.3.5).

The assembler automatically inserts a single unconditional jump to the instruction following the indicated OD keyword. Either a JR or JP is generated depending on the range of the EXIT, with a JR used whenever possible.

The REPEAT statement causes execution to continue at the first statement of the innermost DO...OD block containing the REPEAT. It can also be qualified by a label indicating a specific DO...OD block to which execution is to proceed. Its format is:

```
        REPEAT [FROM label]
```

where

label	conforms to the rules for labels (Sections 2.2.1 and 5.3.5).

The assembler automatically inserts a single unconditional jump instruction at the REPEAT statement which branches to the indicated DO keyword. Either a JR or JP is generated depending on the range of the REPEAT, with a JR used whenever possible.

Example:

```
        LOOP1:  DO
                  ADD RO, @R1
                  INC R5
                  CP  R5, #limit1
                  IF EQ THEN EXIT FI
                  DO
                    ADD R2, @R3
                    INC R6
                    CP  R6, #limit2
                    IF GT THEN REPEAT FROM LOOP1 FI
                  OD
                OD
```

5.2.4 IF Statement

The IF statement specifies that the statements between the keywords THEN and ELSE (or between THEN and FI if the ELSE clause is omitted) are to be executed if the specified condition code is true. If the condition is false and the ELSE clause is present, the statements between ELSE and FI are executed. If the condition is false and the ELSE clause is omitted, execution continues with the statement following FI.

The format of the IF statement is:

```
IF   condition_code
  THEN  action1_statement*
 [ELSE  action2_statement*]
FI
```

where

condition_code	is F, Z, NZ, C, NC, PL, MI, EQ, NE, OV, NOV, PE, PO, LE, LT, GT, GE, ULE, ULT, UGT, or UGE (Section 3.2.1).
action1_statement	is performed if condition_code is true and consists of zero or more assembly-language, DO, IF, REPEAT, or EXIT statements or a combination of these statements.
action2_statement	is performed if condition_code is false and consists of zero or more assembly-language, DO, IF, REPEAT, or EXIT statements or a combination of these statements.

The assembler automatically inserts a single conditional jump instruction just before the THEN keyword, which branches to either the ELSE clause, if present, or to the FI keyword if not. The conditional jump has an opposite sense from the condition code given; for instance, "IF OV THEN" generates a "JP NOV" instruction. The opposing condition pairs are: Z-NZ, C-NC, PL-MI, EQ-NE, OV-NOV, PE-PO, LE-GT, LT-GE, ULE-UGT, and ULT-UGE. If the ELSE clause is present, a single unconditional jump is inserted just before the ELSE clause which branches to the FI keyword.

For each of the jump instructions, either a JR or JP is generated depending on the range of the IF statement, with a JR used whenever possible.

Example:

```
IF NZ THEN
  LD  counter, #1
FI

IF GT
  THEN SCF RET
  ELSE RCF RET
FI
```

5.2.5 IF-CASE Statement

The IF-CASE statement is an extension of the IF statement. It allows the user to select from a series of actions depending on the contents of a selector register. The case whose list contains a match with the contents of the selector register is performed. An ELSE clause can be used to specify alternative statements to be executed if no match occurs. If no ELSE is specified and no match occurs, the statement following the FI keyword is executed next.

The IF-CASE statement has the format:

```
        IF   selector_register
                [CASE  expression+  THEN action_statement*]+
                [ELSE  action_statement*]
        FI
```

where

selector_register	is the designator for a register (Section 2.4.2).
expression	is any expression which is valid as an operand in a Compare instruction.
action_statement	is an assembly language, DO, IF, REPEAT, or EXIT statement.

The assembler automatically inserts a Compare instruction and a conditional jump for each list element. For the last expression in a CASE clause, a "jump NE" to the next CASE is generated (or to the ELSE clause, if present, or if not present, to the FI keyword for the last CASE). Either a JR or JP instruction is used depending on the range of the CASE clause, with a JR used whenever possible. When there is more than one expression in a single CASE clause, all but the last expression generate a "JR EQ" to the start of the action statements associated with the CASE clause. Therefore, the number and size of expressions for a single CASE clause must not exceed the range of the first JR instruction for that clause.

Example:

```
        IF R5
          CASE #1, R4 THEN CALL control_1
          CASE #2, @R3 THEN CALL control_gt_1
          ELSE CALL control_gt_4
        FI
```

5.2.6 Jump Optimization

All jumps produced by PLZ/ASM high level control structures are optimized whenever possible. Jump optimization is explicitly provided to the user via a JPR control instruction. The Jump Relative optimization (JPR) control instruction has the following form:

```
        JPR [cc] <jpr_expr>
```

where:

cc	is any condition code that can be used with a JP or JR instruction.
<jpr_expr>	is a simple expression.

A jump relative optimization (<jpr_expr>) expression is a relocatable expression containing exactly one relocatable value (<label>) and has the following form:

```
        <label> [('+'|'-') <const_expr>]
```

where:

<const_expr>	is a constant expression as defined in Section 2.3.2.
<label>	is a procedure name or program statement label.

Example:

```
        L1: LD R0, R1
             or
        L1 PROCEDURE
              ENTRY
            .
            .
            .
        END L1
```

The destination of a JPR must be a program label with an optional constant added or subtracted to/from it. However, one particular form of <label> $\pm$ <const_expr> cannot be optimized. This form is best explained by the following example:

```
        L1:    JPR   L2-300
                .
                .
                .
        L3:    JPR   L99
                .
                .
                .
        L2:
```

If L2-L1 is less than 300 bytes, the JPR at L1 is actually a backward jump. The destination becomes further away if the JPR at L3 is optimized. This case is very expensive to handle and is rare enough not be be optimized.

JPRs are optimized upon the occurrence of one of the following source conditions. If the target label has not occurred before such a source condition, the JPR will produce a jump.

Source Conditions:

- A $ABS or $REL directive.

- Certain constant expressions cause jump optimization. If a constant expression contains the difference between two relocatable values (<label> - <label>) and if there is a JPR that lies between the two labels, then jump optimization will occur.

There are three conditions that must be met for Jump Relative optimization (JPR) to produce a Jump Relative.

- The JPR and its destination must be in the same section.

- The jump and its target label must be in the same module.

- The target label has to be in the same addressing mode (absolute or relocatable) as the jump. These addresses canr have mixed modes.

5.3 Defining Data

Data (constants and variables) must be defined or declared so that it can be referenced accordingly. In general, data definition associates an identifier with a fixed value or type. Data declaration introduces an identifier as the name of a variable and associates a scope and type with it. The following three statements are used to define and declare data:

- The constant-definition statement (CONSTANT), that associates a constant identifier with a fixed value;

- The type-definition statement (TYPE), that associates a type identifier with a fixed type;

- The variable declaration that associates a variable identifier with a scope, type, and (optionally) an initial value.

Constant identifiers are assumed to have INTERNAL scope. Constants have no explicit type and are represented as 32-bit values.

Type identifiers appearing in type-definitions are assumed to have INTERNAL scope. No scope or initial value can be specified in type-definition statements. They are used primarily to categorize data or provide a template for structured data.

The variable declaration allows a variable identifier to be associated with any specific scope, type, or initial value (within the limits of the variable's scope).

5.3.1 Constant Definition

A constant definition associates an identifier with a constant expression. Since this value must be determinable at assembly time, any identifier appearing in the expression must be previously defined.

Constant identifiers have no explicit type and are always represented as 32-bit values. Constants are defined at the module level, and therefore have a scope of the module in which they are defined (INTERNAL). Constants to be used in other modules must be redefined in those modules.

The format of the constant-definition statement is:

```
        CONSTANT
          [constant_identifier ':=' constant_expression]*
```

where

constant_identifier	conforms to the rules for identifiers (Section 2.2.1).
constant_expression	conforms to the rules for constant expressions (Section 2.3.2).

Example:

```
        CONSTANT
          minus        :=  -1
          count        :=  10
          NEG_COUNT    :=  minus*count
          A_SYMB       :=  'A'
          MODULUS      :=  256
```

5.3.2 Data Types

Data types are associated with variables either to indicate the size of the values the variables can represent or to identify a name as a label. Simple data types indicate whether a variable can hold an 8-bit or a 16-bit value; structured data types provide a template of storage for collections of simple variables. The label type is used to declare the scope of labels explicitly.

Data types can be directly associated with variable identifiers in variable declarations (Section 5.3.4), or they may be associated with variables indirectly using a user-defined type identifier to specify the type. The latter is an identifier that has been previously associated with a type in a TYPE statement (Section 5.3.3). In the following example, the variable CHAR is associated with the simple type BYTE, allowing CHAR to be used as a type identifier in subsequent type definitions:

```
        TYPE
          CHAR BYTE
          letter CHAR
          digit CHAR
```

Simple Types. The basic data type is either a standard simple data type or a simple data type defined by the user in a type definition (like CHAR in the example just given). The standard data types are:

SHORT_INTEGER or BYTE	An 8-bit quantity whose value can be signed (-128 to +127) or unsigned (0 to 255). This value may also represent a single character from the ASCII character set.
INTEGER or WORD	A 16-bit quantity whose value can be signed (-32768 to +32767) or unsigned (0 to 65535).

The values of simple variables (variables defined with simple types) are interpreted as signed or unsigned depending on their use in assembly-language instructions.

```
        TYPE
          CHAR  BYTE
          small_value  BYTE
          large_value  INTEGER
          letter  CHAR
```

Structured Types. Structured types are defined by indicating the structuring method to be used and the types of all elements within the selected structure. Two structuring methods are available: ARRAY and RECORD.

Note that if an array or record has not been previously defined in a program, only the array or record name can be used in an instruction. Array subscripting and record field accessing are not allowed unless the array or record has been previously defined.

Array Structures. An array structure is a collection of variable elements, each of which has the same type. When referenced, the identifier associated with the ARRAY type refers to the entire array structure. Arrays with N elements are indexed from 0 to N-1; for example, a 10-element array has index 0 as the first element and index 9 as the last element.

Individual elements within an array can be accessed in several different ways. A particular element's address can be calculated at run time, for instance, by specifying an indexed address mode. At assembly time, a particular element's address can be specified by an expression containing the array identifier and a fixed offset, or by an array identifier followed by one or more constant expressions enclosed within square brackets. In the latter case, each constant expression represents an index for the particular dimension of the array, and the assembler's calculation of the desired element's address can involve an implicit multiplication by the size of each dimension or by the size (in bytes) of the element type (see Section 5.3.4).

Example:

```
        TYPE
           STRING ARRAY [26 BYTE]
        INTERNAL
           alpha STRING
        !The array identifier "alpha" is defined as a 26-byte
         array!
                .
                .
                .
        LD   R5, #0
        LD  alpha(R5), #'A'
        LD  alpha+1, #'B'
        LD  alpha[2], #'C'
        !The first element of array alpha now contains 'A', the
         second element contains 'B', and the third element
         contains 'C'!
```

Array definition and initialization are explained in detail in Section 5.3.4.

Record Structures. A record structure is a collection of named fields. Unlike array elements, record fields are not required to have the same type. For example, a record structure named strobe might have a BYTE field named pin and a SHORT_INTEGER field named voltage.

```
        TYPE
            strobe RECORD [pin BYTE  voltage SHORT_INTEGER]
        INTERNAL
            s1  strobe
```

As this example indicates, a RECORD type definition specifies an identifier and a type for each of its fields, as well as an identifier for the record structure itself. Individual fields can be referenced subsequently by specifying a record variable name followed by a period (.) and the field name.

```
        LDB  RL5, s1.voltage
```

The scope of a record identifier is specified in the variable declaration in which it is introduced (or it is implicitly INTERNAL if introduced in a TYPE statement). The scope of record field identifiers is the module or procedure in which they are introduced; in other words, all field identifiers must be unique in their entire scope. Record definition is explained in detail in Section 5.3.4.

Label Type. An identifier with type LABEL can only be used as a statement label. The LABEL type declaration is used primarily to explicitly specify a label's scope. The scope of a label is assumed to be the module in that it appears (that is, INTERNAL). Therefore, statement labels which are to be accessible throughout the module need not be declared, although they can be explicitly declared INTERNAL for documentation purposes. If a label is to have GLOBAL, EXTERNAL, or LOCAL scope, however, it must be declared explicitly in a label declaration statement.

```
        GLOBAL
           TRIG_FUNCTION LABEL
```

Label declaration is explained in detail in Section 5.3.5.

5.3.3 Type Definition

The type-definition statement associates an identifier with a fixed type. These identifiers are assumed to have a scope of the current module (that is, INTERNAL scope).

Types are used primarily to categorize data and informally associate attributes or properties with the value of the data. Types ARRAY and RECORD provide an abbreviated template for structured data storage. Z8 PLZ/ASM allows arbitrary association of types and data values with no type-compatibility restrictions.

The format of the type definition statement is:

```
        TYPE
          [type_identifier type]*
```

where

type_identifier	conforms to the rules for identifiers (Section 2.2.1).
type	is BYTE, SHORT_INTEGER, INTEGER, WORD, a previously defined type identifier, or is an ARRAY or RECORD type definition.

An EXTERNAL variable declaration cannot include an initial value. The type of the variable can be any simple or structured type, or a previously-specified type identifier. Consequently, the format of the variable declaration in this case reduces to:

```
          variable_identifier+ type
```

Example:

```
        EXTERNAL
          counter WORD
          input, output ARRAY [72 BYTE]
          customer_name STRING
```

Simple Variable Declaration. Simple variables are variables whose type is BYTE, SHORT_INTEGER, WORD, INTEGER, or a previously specified simple type identifier (Section 5.3.2). Simple variables that are GLOBAL, INTERNAL, or LOCAL can be given an initial value.

Simple variables are initialized in one of two ways--with a single constant expression, or with a list of constant expressions enclosed in square brackets. In the first case, only one variable can appear in the declaration and is initialized to the constant value. If a list is supplied, the variables are initialized in left-to-right order from the initial-value list. The initial-value list can have fewer items than the variable-identifier list, but an error results if the initial-value list is longer.

Example:

```
        INTERNAL
          HUE  BYTE
          limit WORD := %FFFF
          total, subtotal BYTE := [0...]
          A, B, G  BYTE  := ['A', 'B', 'G']
          D, E, F  BYTE  := [0, 1]
        !D=0, E=1, F is still undefined!
```

Array Variable Declaration. An array variable is a variable with the type ARRAY. An array variable declaration has the format:

```
        array_identifier+  ARRAY '[' dimension+ type ']'
                ['=' initial_value]
```

or:

```
        array_identifier+ array_type [':=' initial_value]
```

where

array_identifier	conforms to the rules for identifiers (Section 2.2.1).
dimension	specifies the number of dimensions in the array structure and the number of elements in each dimension. The dimension(s) must be one or more constant expression(s) (Section 2.3.2) or a single asterisk (*), as detailed below.
type	is BYTE, SHORT_INTEGER, INTEGER, WORD, a previously defined type identifier, or is an ARRAY, or RECORD type definition.
array_type	is a previously-defined ARRAY type identifier.
initial_value	is a bracketed list of constant expressions, or a character sequence as detailed below.

Several array identifiers can appear in a single declaration, and, optionally, can be initialized if they are declared GLOBAL, INTERNAL, or LOCAL.

Array structures are initialized in left-to-right order from the initial values supplied and in row-major sequence (that is, in the sequence of ascending memory addresses). For example, a 3x3 matrix would be ordered in the following sequence:

```
[0,0] [0,1] [0,2] [1,0] [1,1] [1,2] [2,0] [2,1] [2,2]
```

Array structures are initialized by a bracketed list of constant expressions. If the initial-value list contains N items, the first N elements of the array structure are initialized (in row-major sequence). The number of constants supplied cannot exceed the number of elements in the array structure.

Example:

```
INTERNAL
  matrix ARRAY [10 10 SHORT_INTEGER]
  list   ARRAY [10 BYTE]  := [0,1,0,0,1]
  TABLE  ARRAY [4 BYTE]   := ['T', 'O', 'D', 'S']
  ONEDIM1, ONEDIM2  ARRAY [2 BYTE] := [[1...][2...]]
```

Normally, each dimension specified in the array declaration must be a constant expression so that variable upper bounds are prohibited. The sole exception is an array declaration initializing a one-dimensional array structure. In this case, * is specified as the dimension and the length of the list is determined by the number of items in the initialization list. When the * feature is used, array structures can be initialized in two ways: with a bracketed list of constants as described above, or with a character sequence (enclosed in single quote marks). In the latter instance, the array elements must be type BYTE, SHORT_INTEGER, or a user-defined 8-bit type and each byte is initialized to a single character value.

Example:

```
        INTERNAL
          list ARRAY [* BYTE]  := [0,1,0,0,1]
        !This array is only 5 bytes;  "list" array in last
         example is 10 bytes, although only 5 are initialized!

          TABLE ARRAY [* BYTE]  := 'TODS'
        !Compare to TABLE array in last example!
```

Record Variable Declaration. A record variable is a variable whose type is RECORD. A record variable declaration specifies an identifier for the record as a whole and an identifier and type for each field within the record. A field type can be ARRAY or RECORD, as well as a simple type.

A record variable declaration has the format:

```
        record_identifier+ RECORD '[' [field_identifier+ type]+ ']
                ['':=' initial_value]
```

or:

```
        record_identifier+ record_type
                [':=' initial_value]
```

where:

identifier	conforms to the rules for identifiers (Section 2.2.1).
type	is BYTE, SHORT_INTEGER, WORD, INTEGER, a previously-defined type identifier, or is an ARRAY or RECORD type definition.
record_type	is a previously-defined RECORD type identifier.
initial_value	is a bracketed list of constant expressions.

Several record identifiers can appear in a single declaration, and, opionally, can be initialized if declared GLOBAL, INTERNAL, or LOCAL. Each record field is initialized in left-to-right order from the values given in the initial list. A list of N values enclosed by square brackets may be given, with the first N fields being initialized. Having more constants than the total number of record fields is flagged as an error.

Example:

```
   GLOBAL
     person  RECORD     [age, height, weight  BYTE
                         birth RECORD [day, month, year BYTE]
                         salary WORD]

     MSG  RECORD        [length BYTE char ARRAY [50 BYTE]] :=[0[0]]
                         !length field and first byte of char array
                         are initialized to zero!
```

If an array or record appears within another array or record, then this nesting is represented by enclosing each level of initialization values within square brackets. Note that in this case, the last record fields or array elements at each level do not have to be specified. Furthermore, if more than one structured variable identifier appears in a single declaration, then the part of the initial value list corresponding to each structured variable must also be enclosed by square brackets.

```
        TYPE
            PATIENT RECORD  [ROOM WORD
                             BIRTH RECORD [DAY,MO,YR BYTE]
                             SEX  BYTE]
        INTERNAL
            FEMALE ARRAY [100 PATIENT] :=[[?,[],'F']...]
               !only the SEX field of each record is initialized!
```

Memory Section Allocation. Variables can be assigned to the DATA, REGISTER, or PROGRAM memory sections using the $SECTION assembler directive. If $SECTION is not specified, variables are assigned to REGISTER memory by default. REGISTER memory cannot be initialized, however. Variables declared when the default is in effect cannot be assigned initial values.

5.3.5 Label Declaration

The label declaration statement specifies that an identifier is used in the program as a statement label. It cannot be used for any other purpose within its defined scope. The format of the label declaration is:

```
        label_declaration+ LABEL
```

where

label_identifier	conforms to the rules for identifiers in Section 2.2.1.

Note that the colon (:) that follows a label identifier when it appears in an executable statement (Section 2.2.1) is not included when the label is identified in a label declaration statement. Note also that labels cannot be given an initial value (that is, the label declaration cannot be used to assign absolute addresses).

A label can have GLOBAL, EXTERNAL, INTERNAL, or LOCAL scope (Section 4.2.5). If a label is used in an executable statement without being declared in a label declaration statement, it is assumed to be INTERNAL to the module in which it appears. This default scope can be overridden by explicitly declaring the scope of the label.

Note that a label with LOCAL scope must be declared in the procedure declaration before the ENTRY keyword; that is, before the label is used or defined, whereas a label with GLOBAL, EXTERNAL, or INTERNAL scope can be used before it is either declared or defined. To allow the programmer to avoid pre-declaring LOCAL labels, Z8 PLZ/ASM provides a special form of statement labels which are always of LOCAL scope and cannot be declared explicitly. The form of a special label is a dollar sign ($) followed immediately by any valid decimal number, and it can be used in the same manner as a regular statement label identifier, except that its scope is always limited to the procedure in which it is defined.

Example:

```
        GLOBAL
          STEP3   LABEL

           process10  PROCEDURE       !Procedure has GLOBAL scope!
              LOCAL
                a,b,c  BYTE
                STEP2  LABEL
              ENTRY
                 STEP1: ...           !STEP1 has INTERNAL scope!
                   .
                   .
                   .
                 STEP2: ...           !STEP2 is LOCAL to
                   .                   "process10"!
                   .
                   .
                 STEP3: ...           !STEP3 has GLOBAL scope!
                   .
                   .
                   .
                 $1: ...              !$1 is LOCAL to "process10"!
                   .
                   .
                   .
        END process10
```

5.3.6 SIZEOF Operator

Z8 PLZ/ASM includes a special unary operator, SIZEOF, which operates on type identifiers, type reserved words, and variable identifiers for structure types to determine the size (in bytes) of a variable field. The SIZEOF operator does not work on variable identifiers for simple types. In the following example, the identifier to which the SIZEOF operator is being applied to, has already been defined.

```
TYPE
  char     BYTE
  digit    char
  matrix   ARRAY [10 10 WORD]
  patient  RECORD [height weight BYTE
                        room WORD]
    .
    .
    .

  LD  R0, #SIZEOF digit                     !R0 contains 1!
  LD  R1, #SIZEOF matrix                    !R1 contains 200!
  LD  R1, #(SIZEOF matrix/
              SIZEOF WORD)                  !R1 contains 100!
  LD  R2, #SIZEOF patient                   !R2 contains 4!
  LD  R3, #SIZEOF patient.weight            !R3 contains 1!
```

The SIZEOF operator can also be used on identifiers that are defined later in the program:

```
MSGLEN BYTE := SIZEOF MSG
MSG ARRAY[* BYTE] := 'Hello%r'
```

Below is a description of the restrictions on the SIZEOF operand when it is not predefined.

Forward References

In PLZ/ASM, forward references can only be made to an identifier. The identifier can represent a label, a data variable, or a procedure name. Because array dimensions and record fields are constant values, forward references cannot be made to subscripted array variables or fields of record variables. For example:

```
        .
        .
LD R0, A[1]
LD R0, REC.F1
        .
        .
```

are illegal unless the record or array is predefined.

The restriction applies to the SIZEOF operator as well. The SIZEOF operator works on identifiers only and not on subscripted array variables and fields of record variables, unless the array or record is predefined. For example:

```
               .
               .
LD R0, #SIZEOF A       !LEGAL!
LD R0, #SIZEOF REC     !LEGAL!
LD R0, #SIZEOF A[1]    !ILLEGAL!
LD R0, #SIZEOF REC.F1  !ILLEGAL!

A ARRAY [10 BYTE]
REC RECORD [F1 BYTE, F2 WORD]

LD R0, #SIZEOF A[1]    !LEGAL!
LD R0, #SIZEOF REC.F1  !LEGAL!
               .
               .
```

Appendix A
Assembly Language Instruction Summary

This appendix provides a quick-reference summary of the Z8 PLZ/ASM assembly-language instruction set. For an expanded explanation of any of these instructions, see Section 3.

In the instruction summary, addressing modes and status flags are represented by the following notational shorthand:

Addressing Modes

Symbol	Meaning
R	Register or working-register address
r	Working-register address only
IR	Indirect-register or indirect working-register address
Ir	Indirect working-register address only
RR	Register pair or working-register pair address
IRR	Indirect register pair or indirect working-register pair address
Irr	Indirect working-register pair only
X	Indexed
DA	Direct Address
RA	Relative address
IM	Immediate

Status Flags

Symbol	Meaning	Symbol	Meaning
C	Carry flag	V	Overflow flag
Z	Zero flag	D	Decimal Adjust flag
S	Sign flag	H	Half Carry flag

Additional symbols used are:

Symbol	Meaning
dst	Destination operand
src	Source operand
cc	Condition code (see list below)
@	Indirect address prefix
SP	Stack pointer
PC	Program counter
FLAGS	Flag register (control register 252)
RP	Register pointer (control register 253)
IMR	Interrupt mask register (control register 251)

Condition Codes

Code	Meaning	Flags Set
F	Always false	-
(blank)	Always true	-
C	Carry	C = 1
NC	No carry	C = 0
Z	Zero	Z = 1
NZ	Not zero	Z = 0
PL	Plus	S = 0
MI	Minus	S = 1
OV	Overflow	V = 1
NOV	No overflow	V = 0
EQ	Equal	Z = 1
NE	Not equal	Z = 0
GE	Greater than or equal	(S XOR V) = 0
LT	Less than	(S XOR V) = 1
GT	Greater than	(Z OR (S XOR V)) = 0
LE	Less than or equal	(Z OR (S XOR V)) = 1
UGE	Unsigned greater than or equal	C = 0
ULT	Unsigned less than	C = 1
UGT	Unsigned greater than	((C=0) and (Z=0)) = 1
ULE	Unsigned less than or equal	(C or Z) = 1

The flags affected by each instruction are indicated by:

0:	cleared to zero
1:	set to one
*:	set or cleared according to operation
-:	unaffected
X:	undefined

Assignment of a value is indicated by the symbol "<-". For example,

```
dst <- dst + src
```

indicates that the source data is added to the destination data and the result is stored in the destination location. The notation "addr(n)" is used to refer to bit "n" of a given location. For example,

```
dst(7)
```

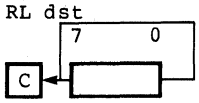
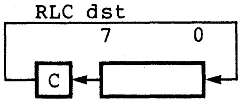
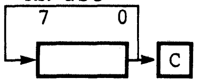
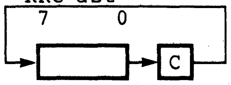
refers to bit 7 of the destination operand.

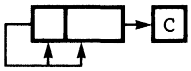
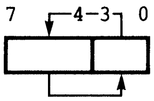
Instruction and Operation	Addr Modes dst	Addr Modes src	Hex Opcode	Bytes	Cycles	Flags Affected C Z S V D H
ADC dst,src	r	r	12	2	6	* * * * 0 *
dst<-dst+src+C	r	Ir	13	2	6	
	R	R	14	3	10	
	R	IR	15	3	10	
	R	IM	16	3	10	
	IR	IM	17	3	10	
ADD dst,src	r	r	02	2	6	* * * * 0 *
dst<-dst+src	r	Ir	03	2	6	
	R	R	04	3	10	
	R	IR	05	3	10	
	R	IM	06	3	10	
	IR	IM	07	3	10	
AND dst,src						
dst<-dst AND src	r	r	52	2	6	- * * 0 - -
	r	Ir	53	2	6	
	R	R	54	3	10	
	R	IR	55	3	10	
	R	IM	56	3	10	
	IR	IM	57	3	10	
CALL dst	DA		D6	3	20	- - - - - -
SP<-SP-2	IRR		D4	2	20	
@SP<-PC						
PC<-dst						
CCF			EF	1	6	* - - - - -
C<-NOT C						
CLR dst	R		B0	2	6	- - - - - -
dst<-0	IR		B1	2	6	

Instruction and Operation	Addr Modes dst	src	Hex Opcode	Bytes	Cycles	Flags Affected C X S V D H
COM dst	R		60	2	6	- * * 0 - -
dst<-NOT dst	IR		61	2	6	
CP dst,src	r	r	A2	2	6	* * * * - -
dst-src	r	IR	A3	2	6	
	R	R	A4	3	10	
	R	IR	A5	3	10	
	R	IM	A6	3	10	
	IR	IM	A7	3	10	
DA dst	R		40	2	8	* * * X - -
dst<-DA dst	IR		41	2	8	
DEC dst	R		00	2	6	- * * * - -
dst<-dst-1	IR		01	2	6	
DECW dst	RR		80	2	10	- * * * - -
dst<-dst-1	IR		81	2	10	
DI			8F	1	6	- - - - - -
IMR(7)<-0						
DJNZ r,dst	RA		rA	2	12/10	- - - - - -
r<-r-1			r=0-F		(taken/	
if r<>0					not taken)	
PC<-PC+dst						
Range: +127,-128						
EI			9F	1	6	- - - - - -
IMR(7)<-1						

Instruction and Operation	Addr Modes dst	src	Hex Opcode	Bytes	Cycles	Flags Affected C X S V D H
INC dst	r		rE	1	6	- * * * - -
dst<-dst+1			r=0-F			
	R		20	2	6	
	IR		21	2	6	
INCW dst	RR		A0	2	10	- * * * - -
dst<-dst+1	IR		A1	2	10	
IRET			BF	1	16	* * * * * *
FLAGS<-@SP						
SP<-SP+1						
PC<-@SP						
SP<-SP+2						
IMR(7)<-1						
JP cc,dst	DA		cD	3	12/10	- - - - - -
if cc is true,			c=0-F		(taken/	
PC<-dst					not taken)	
	IRR		30	2	8	
JR cc,dst	RA		cB	2	12/10	- - - - - -
if cc is true,			c=0-F		(taken/	
PC<-PC+dst					not taken)	
Range: +128,-127						
LD dst,src	r	IM	rC	2	6	- - - - - -
dst<-src	r	R	r8	2	6	
	R	r	r9	2	6	
			r=0-F			
	r	X	C7	3	10	
	X	r	D7	3	10	
	r	Ir	E3	2	6	
	Ir	r	F3	2	6	
	R	R	E4	3	10	
	R	IR	E5	3	10	
	R	IM	E6	3	10	
	IR	IM	E7	3	10	
	IR	R	F5	3	10	

Instruction and Operation	Addr Modes dst	src	Hex Opcode	Bytes	Cycles	Flags Affected C X S V D H
LDC dst,src	r	Irr	C2	2	12	- - - - - -
dst<-src	Irr	r	D2	2	12	
LDCI dst,src	Ir	Irr	C3	2	18	- - - - - -
dst<-src	Irr	Ir	D3	2	18	
r<-r+1						
rr<-rr+1						
LDE dst,src	r	Irr	82	2	12	- - - - - -
dst<-src	Irr	r	92	2	12	
LDEI dst,src	Ir	Irr	83	2	18	- - - - - -
dst<-src	Irr	Ir	93	2	18	
r<-r+1						
rr<-rr+1						
NOP				FF	6	- - - - - -
OR dst,src	r	r	42	2	6	- * * 0 - -
dst<-dst OR src	r	Ir	43	2	6	
	R	R	44	3	10	
	R	IR	45	3	10	
	R	IM	46	3	10	
	IR	IM	47	3	10	
POP dst	R		50	2	10	- - - - - -
dst<-@SP	IR		51	2	10	
SP<-SP+1						
PUSH src		R	70	2	10/12	- - - - - -
SP<-SP-1					(int/ext stack)	
@SP<-src		IR	71	2	12/14	
					(int/ext stack)	

Instruction and Operation	Addr Modes dst	src	Hex Opcode	Bytes	Cycles	Flags Affected C X S V D H
RCF C<-0			CF	1	6	0 - - - - -
RET PC<-@SP SP<-SP+2			AF	1	14	- - - - - -
RL dst  	R		90	2	6	* * * * - -
	IR		91	2	6	
RLC dst  	R		10	2	6	* * * * - -
	IR		11	2	6	
RR dst  	R		E0	2	6	* * * * - -
	IR		E1	2	6	
RRC dst  	R		C0	2	6	* * * * - -
	IR		C1	2	6	
SBC dst,src dst<-dst-src-C	r	r	32	2	6	* * * * 1 *
	r	Ir	33	2	6	
	R	R	34	3	10	
	R	IR	35	3	10	
	R	IM	36	3	10	
	IR	IM	37	3	10	

Instruction and Operation	Addr Modes dst	src	Hex Opcode	Bytes	Cycles	Flags Affected C X S V D H
SCF C<-1			DF	1	6	1 - - - - -
SRA dst 7 0 	R		D0	2	6	* * * 0 - -
	IR		D1	2	6	
SRP src RP<-src		IM	31	2	6	- - - - - -
SUB dst,src dst<-dst-src	r	r	22	2	6	* * * * 1 *
	r	Ir	23	2	6	
	R	R	24	3	10	
	R	IR	25	3	10	
	R	IM	26	3	10	
	IR	IM	27	3	10	
SWAP dst 	R		F0	2	8	X * * X - -
	IR		F1	2	8	
TCM dst,src (NOT dst)AND src	r	r	62	2	6	- * * 0 - -
	r	Ir	63	2	6	
	R	R	64	3	10	
	R	IR	65	3	10	
	R	IM	66	3	10	
	IR	IM	67	3	10	
TM dst,src dst AND src	r	r	72	2	6	- * * 0 - -
	r	Ir	73	2	6	
	R	R	74	3	10	
	R	IR	75	3	10	
	R	IM	76	3	10	
	IR	IM	77	3	10	

Instruction and Operation	Addr Modes dst	src	Hex Opcode	Bytes	Cycles	Flags Affected C X S V D H
XOR dst,src	r	r	B2	2	6	- * * 0 - -
dst<-dst XOR src	r	Ir	B3	2	6	
	R	R	B4	3	10	
	R	IR	B5	3	10	
	R	IM	B6	3	10	
	IR	IM	B7	3	10	

Appendix B
High-Level Statement Summary

The following summarizes the high-level keywords and their uses.

ARRAY	One of the two structured variable types. Used in type definition and variable declaration statements.
BYTE	One of the simple variable types. Used in type definition and variable declaration statements.
CASE	Used in IF conditional-execution statement. Instructions following CASE definition are executed if one of the specified values matches selector register.
CONSTANT	Introduces constant definition(s).
DO	Introduces DO loop.
ELSE	Used in IF conditional-execution statement. Statements between ELSE and FI are executed if the specified condition is false.
END	Module or procedure terminator.
ENTRY	Marks beginning of action-statement part of a procedure.
EXIT	Loop control statement used to control execution flow of a DO loop.
EXTERNAL	Specifies that variables and/or procedures defined as GLOBAL in another module will be used in the current module.
FI	IF statement terminator.

FROM	Used in conjunction with EXIT and REPEAT loop control statements.
GLOBAL	Declares variables and/or procedures to have a scope of the entire executable program.
IF	Introduces IF statement. Code following IF-THEN is executed if the specified condition is true.
INTEGER	One of the simple variable types. Equivalent to WORD.
INTERNAL	Declares variables and/or procedures to have a scope of the current module only.
JPR	A control instruction for which the assembler will produce a jump relative instruction whenever possible.
LABEL	Used to declare statement label scope explicitly.
LOCAL	Declares variables to have a scope of the current procedure only.
MODULE	Introduces a module.
OD	DO loop terminator.
PROCEDURE	Introduces a procedure.
RECORD	One of the two structured variable types. Used in type definition and variable declaration statements.
REPEAT	Loop control statement used to control execution flow of a DO loop.
SHORT_INTEGER	One of the simple variable types. Equivalent to BYTE.
THEN	Used in IF conditional-execution block. The statements between THEN and ELSE (or THEN and FI if ELSE is omitted) are executed if the specified condition is true.

TYPE	Introduces type definitions.
WORD	One of the simple variable types. Used in type definition and variable declaration statements.

The remainder of this appendix contains the complete grammar for Z8 PLZ/ASM. In this grammar:

a. Keywords are shown as all capital letters;

b. Required special characters are enclosed in single quotes;

c. Optional items are enclosed in square brackets;

d. Possible repetition of an item is shown by appending a "+" (to signify one or more repetitions) or "*" (to signify zero or more repetitions) to the item;

e. Parentheses are used to group items to be repeated;

f. A vertical bar "|" signifies an alternative follows.

```
module                     => module_identifier MODULE
                                 declarations*
                              END module_identifier

declarations               => constants
                           => types
                           => globals
                           => internals
                           => externals

constants                  => CONSTANT
                                 constant_definition*

types                      => TYPE
                                 type_definition*

globals                    => GLOBAL
                                 var_or_proc_declaration*

internals                  => INTERNAL
                                 var_or_proc_declaration*

externals                  => EXTERNAL
                                 restricted_var_or_proc_declaration*

constant_definition        => constant_identifier
                                 ':=' expression

expression                 => arithmetic_expression
                                 [rel_op arithmetic_expression]

arithmetic_expression      => term [add_op term]*

term                       => factor [mult_op factor]*

factor                     => unary_operator factor
                           => '(' expression ')'
                           => SIZEOF type_identifier
                           => constant_identifier
                           => label
                           => variable
                           => number
                           => character_constant

character_constant         => character_sequence

type_definition            => type_identifier type
```

```
type                       => simple_type
                           => structured_type

simple_type                => BYTE
                           => SHORT_INTEGER
                           => WORD
                           => INTEGER
                           => simple_type_identifier

structured_type            => array_type
                           => record_type

array_type                 => ARRAY '[' expression+ type ']'
                           => array_type_identifier

record_type                => RECORD '[' field_declaration+ ']'
                           => record_type_identifier

field_declaration          => field_identifier+ type

var_or_proc_declaration => variable_declaration
                           => procedure_declaration

variable_declaration       => variable_noinitial_declaration
                           => variable_initial_declaration
                           => label_declaration

restricted_var_or_proc_declaration

                           => identifier+ type
                           => procedure_identifier PROCEDURE

variable_noinitial_declaration

                           => identifier+ type

variable_initial_declaration

                           => identifier simple_type
                                 ':=' initial_value
                           => identifier identifier+ simple_type
                                 ':=' '[' initial_value* ['...'] ']'
                           => identifier structured_type
                                 ':=' constructor
                           => identifier identifier+ structured_type
                                 ':=' '[' constructor* ['...'] ']'
                           => identifier ARRAY '[' '*' simple_type ']
                                 ':=' '[' initial_value+ ']'
                           => identifier ARRAY '[' '*' simple_type ']
                                 ':=' character_sequence
```

```
constructor              => '['initial_component* ['...'] ']'

initial_component        => initial_value
                         => constructor

initial_value            => expression
                         => '?'

label_declaration        => label_identifier+ LABEL

variable                 => identifier
                         => array_variable
                         => record_variable

array_variable           => array_designator '[' expression+ ']'

array_designator         => array_identifier
                         => record_variable
                         => array_variable

record_variable          => record_designator '.' field_identifier

record_designator        => record_identifier
                         => array_variable
                         => record_variable

procedure_declaration    => procedure_identifier
                            PROCEDURE
                               locals*
                            [ENTRY
                               statement*]
                            END procedure_identifier

locals                   => LOCAL
                               variable_declaration*

statement                => [label ':'] statement
                         => loop_statement
                         => exit_statement
                         => repeat_statement
                         => if_statement
                         => select_statement
                         => assembler_instruction

loop_statement           => DO
                               statement*
                            OD

label                    => label_identifier
                         => '$' decimal_constant
```

```
exit_statement            => EXIT [FROM label]

repeat_statement          => REPEAT [FROM label]

if_statement              => IF condition_code THEN statement*
                                  [ELSE statement*] FI

select_statement          => IF selector_register
                                  select_element+
                                  [ELSE statement*] FI

selector_register         => register
                          => indirect_register

select_element            => CASE expression+
                                  THEN statement*

rel_op                    => '=' | '<>' | '<' | '>' | '<=' | '>='

add_op                    => '+' | '-' | LOR | LXOR

mult_op                   => '*' | '/' | LAND | MOD | SHL | SHR

unary_operator            => '+' | '-' | LNOT

assembler_instruction     => operation operand*

operation                 => instruction
                          => pseudo_instruction

operand                   => register
                          => indirect_register
                          => immediate
                          => index
                          => direct_address
                          => relative_address
                          => condition_code

register                  => address_designator
                          => numbered_register

numbered_register         => single_register
                          => double_register

single_register           => R0 | R1 | R2 | R3 | R4 | R5 |
                             R6 | R7 | R8 | R9 | R10 | R11 |
                             R12 | R13 | R14 | R15

double_register           => RR0 | RR2 | RR4 | RR6 |
                             RR8 | RR10 | RR12 | RR14

indirect_register         => '@' register
```

```
immediate                   => '#' expression
                            => '#' HI expression
                            => '#' LO expression

index                       => address_designator '(' single_register ')'

direct_address              => address_designator

relative_address            => address_designator

address_designator          => expression

condition_code              => F | Z | NZ | EQ | NE | MI | PL | C |
                               NC | OV | NOV | LT | GE | LE |
                               GT | ULT | UGE | ULE | UGT

instruction                 => ADC | ADD | AND | CALL | CCF | CLR |
                               COM | CP | DA | DEL | DELW | DI |
                               DJNZ | EJ | INC | INCW | IRET | JP |
                               JR | LD | LDC | LDCI | LDE | LDEI |
                               NOP | OR | POP | PUSH | RCF | RET |
                               RL | RLC | RR | RRC | SBC | SCF |
                               SRA | SRP | SWAP | SUB | TCM | TM | XOR

pseudo_instruction          => ADCW | ADDW | ANDW | COMW |
                               CPW | DWJNZ | JPR | LDCIW | LDCW |
                               LDEIW | LDEW | ORW | POPW | PUSHW |
                               SBCW | SUBW | XORW
```

PLZ/ASM GRAMMAR LEXICAL SYNTAX

```
PLZ_text                    => separator* [id_constant_text]
                                 (separator+ id_constant_text)*

id_constant_text            => identifier
                            => PLZ_word_symbol
                            => Z8_word_symbol
                            => Z8_extended_word_symbol
                            => register_word_symbol
                            => condition_word_symbol
                            => constant
                            => control_reg_word_symbol

separator                   => delimiter_text
                            => special_symbol

module_identifier           => identifier

constant_identifier         => identifier

type_identifier             => identifier

simple_type_identifier      => identifier

array_type_identifier       => identifier

record_type_identifier      => identifier

field_identifier            => identifier

procedure_identifier        => identifier

array_identifier            => identifier

record_identifier           => identifier

label_identifier            => identifier

identifier                  => letter (letter  |  digit  |  '_')*

constant                    => number
                            => character_sequence

delimiter_text              => delimiter
                            => comment
```

```
number                   => decimal_constant
                         => hex_constant
                         => octal_constant
                         => binary_constant

decimal_constant         => digit+

hex_constant             => '%' hex_digit+

octal_constant           => '%(8)' oct_digit+

binary_constant          => '%(2)' bin_digit+

character_sequence       => ''' string_text+ '''

string_text              => string_char
                         => special_string_text

string_char              => any_character_except_%_or_'

special_string_text      => '%' special_string_char
                         => '%' hex_digit hex_digit

special_string_char      => 'R' | 'L' | 'T' | 'P' | 'Q' | '%' |
                            'r' | 'l' | 't' | 'p' | 'q'

comment                  => '!' any_character_except_! '!'
                         => '//' any_character_except_carriage_return
                            carriage return

letter                   => 'A' | 'B' | ... | 'Z' |
                            'a' | 'b' | ... | 'z'

bin_digit                => '0' | '1'

oct_digit                => '0' | '1' | '2' | '3' |
                            '4' | '5' | '6' | '7'

digit                    => '0' | '1' | '2' | '3' | '4' |
                            '5' | '6' | '7' | '8' | '9'

hex_digit                => '0' | '1' | '2' | '3' | '4' |
                            '5' | '6' | '7' | '8' | '9' |
                            'A' | 'B' | 'C' | 'D' | 'E' | 'F' |
                            'a' | 'b' | 'c' | 'd' | 'e' | 'f'

special_symbol           => '+' | '-' | '*' | '/' | ':=' |
                            ':' | '.' | '%' | '#' | '@' |
                            '[' | ']' | '(' | ')' | '$' |
                            '=' | '<' | '>' | '<=' | '>='
```

```
PLZ_word_symbol           => ARRAY | BYTE | CASE | CONSTANT |
                             DATA | DO | ELSE | END | ENTRY |
                             EXIT | EXTERNAL | FI | FROM |
                             GLOBAL | IF | INCLUDE | INTEGER |
                             INTERNAL | LAND | LNOT | LOCAL |
                             LOR | LOW | LXOR | MOD | MODULE |
                             OD | PROCEDURE | PROGRAM | RECORD |
                             REGISTER | REPEAT | SHL |
                             SHORT_INTEGER | SHR | THEN | TYPE |
                             WORD

Z8_word_symbol            => ADC | ADD | AND | CALL | CCF | CLR |
                             COM | CP | DA | DEL | DELW | DI |
                             DJNZ | EJ | INC | INCW | IRET | JP |
                             JR | LD | LDC | LDCI | LDE | LDEI |
                             NOP | OR | POP | PUSH | RCF | RET |
                             RL | RLC | RR | RRC | SBC | SCF |
                             SRA | SRP | SWAP | SUB | TCM | TM |
                             XOR

Z8_extended_word_symbol => ADCW | ADDW | ANDW | BVAL | CLRW |
                             COMW | CPW | DWJNZ | JPR | LDCIW |
                             LDCW | LDEIW | LDEW | ORW | POPW |
                             PUSHW | SBCW | SUBW | WVAL | XORW

register_word_symbol      => R0 | R1 | R2 | R3 | R4 | R5 | R6 |
                             R7 | R8 | R9 | R10 | R11 | R12 |
                             R13 | R14 | R15 | RR0 | RR2 | RR4 |
                             RR6 | RR8 | RR10 | RR12 | RR14

control_reg_word_symbol => SIO |TMR |T1 | PRE1 | T0
                             |PRE0 | P2M | P3M | P01M
                             |IPR | IRQ | IMR | FLAGS
                             |RP | SP | SPH | SPL

condition_word_symbol     => F | Z | NZ | EQ | NE | MI | PL | C |
                             NC | OV | NOV | LT | GE | LE |
                             GT | ULT | UGE | ULE | UGT

delimiter                 => ';' | space | ',' |
                             tab | formfeed | linefeed  |
                             carriage_return
```

Appendix C
Assembler Directives and Pseudo Instructions

C.1 Assembler Directives

The following is a summary of some of the assembler directives used to control the operation of the Z8 assembler.

These directives can be embedded in the source program, and always start with a dollar sign ($), immediately followed by the particular directive and then any operands. For example, a programmer may want to fix the address for a reset procedure:

```
        $ABS 12
```

The assembler directives are:

$ABS [location]	Specifies that the object code produced is to be absolute. If location is specified, the location counter will begin assigning absolute addresses from that location. Location must be a constant expression. $ABS remains in effect until a $REL directive is encountered.
$LISTON [option] $LISTOFF [option]	These two directives control listing format. When used without options they turn source line listing on or off. The default listing mode is $LISTON. Note that for every $LISTOFF an accompanying $LISTON directive must be used to resume listing source lines. For example, two $LISTOFFs require two $LISTONs to begin. Several options can be used with these directives. $TTY directs the assembler to produce a narrower listing format suitable for 80-column paper or CRT screens. $LISTOFF $TTY is the default and the standard full width line is listed. $BEX directs the format of the object code. Turning this option off ($LISTOFF

$LISTON [option] $LISTOFF [option] (cont.)	$BEX) causes the listing to contain only one line of object code for every input by suppressing any subsequent lines of object code that may be produced. Turning the $BEX option on, which is the default, prints all the object code produced. Note that all listing directives appear in the listing itself.
$PAGE	Causes the listing to space to the next top-of-page. The directive itself does not appear in the listing.
$REL [location] or $REL [$ ± value]	Specifies that the object code produced is to be relocatable. If location is specified, it is an offset from the beginning of the current section. Location must be an absolute value or an offset from the current location counter. $REL remains in effect until the $ABS directive is encountered. $REL 0 is the default at the start of module assembly.
$EVEN	Aligns the current location of the assembler to an even boundary.
$SDEFAULT	Cancels the effect of the $SECTION assembler directive and restores default memory assignment. Data declarations reside in the REGISTER area and procedure declarations reside in the PROGRAM area.

$SECTION [identifier] area

This directive causes the object code produced to be associated with a symbolic identifier and allocated to one of the three memory spaces of the Z8, Program, Register or Data. By default there are three sections, one for each memory space, called PROGRAM, REGISTER, and DATA. Procedure declarations reside in the PROGRAM section and data declarations reside in the REGISTER section. The DATA section is not used unless specified by the user. These defaults can be overridden by specifying $SECTION Area, which forces default sections to contain different portions of a program.

$SECTION [identifier] area (cont.)	User-defined sections can be created by specifying an identifier to be associated with a section that is to be allocated to a particular memory space. At link time sections with the same name from different modules are combined and errors occur if they are not allocated to the same memory space. The $SDEFAULT directive causes default sectioning to resume.
$DATA and $CODE	$DATA instructs the assembler to allow PLZ type data declarations inside of a procedure. Inside a procedure, everything between the $DATA and $CODE is called a $DATA block. $DATA blocks are only allowed inside procedures and are terminated by a $CODE directive, or the procedure END statement. The $CODE directive causes the assembler to return to processing instructions. Procedure definitions are forbidden inside $DATA blocks. Conversely, the $CODE directive can be used to instruct the assembler to allow machine instructions without declaring a procedure.

$CODE blocks are allowed only in data areas, not inside procedures. They can consist only of instructions, high level PLZ contructs or assembler directives. Procedure or data declarations are not allowed inside $CODE blocks. Any $CODE directive occurring within a procedure has no effect and is ignored by the assembler. Any procedure definition occurring within a $CODE block causes an assembler error. All $CODE blocks must be terminated by a $DATA directive or the end of module statement.

Neither type of block causes any automatic section changing. However, $SECTION directives are allowed within the blocks. $SECTION directives within a block only apply to the statements within that block. When a block is terminated, the section reverts back to the section in effect before the $CODE or $DATA directive was encountered.

A $DATA block must have a declaration scope specified. That is, the first statement (excluding assembler directives) after the $DATA must be one of GLOBAL, INTERNAL, EXTERNAL, TYPE, or CONSTANT. Note that when the block is terminated any data or procedure declarations that follow will have the declaration scope in effect before the block was entered. Local scope is not ıllowed within a $DATA block. However, an INTERNAL or CONSTANT leclaration whose identifier is of the form used to denote local .abels, will be local to the procedure that data block was leclared in.

:he following example produces a table of values within a ›rocedure:

```
        GLOBAL test PROCEDURE
        Entry
              .

              .
        $DATA
        INTERNAL
        t ARRAY [3 WORD] := [L1, L2, L3]   !Any data declaration here !
        y byte := 0
        $CODE                        !Return to instruction processing !
```

.2 Pseudo Instructions

he following two pseudo instructions can be placed within a rocedure as a one-operand assembly-language instruction and roduce an arbitrary byte or word value.

VAL expression	Defines a byte value to be located at the current location counter. Can be used to create Z8 instructions or data.
VAL expression	Defines a word value to be located at the current location counter. Can be used to create Z8 instructions or data.

.3 Conditional Assembly

›nditional assembly allows the programmer to inhibit the ssembly of portions of the source text provided certain ›nditions are satisfied. Conditional assembly is particularly seful when a program requires similar code sequences for Lightly different applications.

ıther than generating a multitude of programs to handle each ›plication, the application-dependent sections of code can be ıclosed by the conditional pseudo-ops within a single program. / changing the values of several symbols used to control the ›nditional assembly, the user can generate different object ›dules from subsequent assemblies of the same source.

The assembler directives that comprise conditional assembly are:

$IF <id>	Begin conditional.
$FI	End conditional.
$ELSE	Reverse sense of conditional.
$THEN	Optional, may follow "$IF <id>".

An example of correct usage of directives is as follows:

```
        $IF <id> $THEN
        .
        .
        .
        $ELSE     ! optional !
        .
        .
        .
        $FI
```

<id>, explained fully below, determines whether the condition is true or false. If the condition is true, the statements between $THEN and $ELSE assemble, and those between $ELSE and $FI are ignored. If the condition is false, the opposite occurs.

<id> can be a single identifier or a numeric constant. Expressions are not allowed. <id> can be: 1) a constant; 2) another identifier, such as a data item or label; or 3) an undefined symbol.

The conditional is true for a nonzero constant or a defined identifier; it is false for a zero constant or an undefined symbol.

An example of the usage of the directive $IF <id> is as follows:

```
         PGM MODULE
             CONSTANT
                 Z80PS := 1      ! Assemble for the Z8 version !

             .
             .
             .
             $IF Z80PS $THEN
             CALL Z8PRINT
             $ELSE
             CALL Z80PRINT
             $FI Z80PRINT
             .
             .
             .
       END      PGM
```

n this case, the first call would assemble and the other would e ignored.

NOTE

These directives should not be confused with the IF/THEN/ELSE/FI higher level runtime statements. Also, more than one $ELSE is allowed for each $IF. Each $ELSE simply reverses the meaning. Also note that nesting of conditionals to any level is allowed.

Appendix D
Reserved Words and Special Characters

D.1 Reserved Words

Certain special symbols are reserved for Z8 PLZ/ASM and can not be redefined as symbols by the programmer. These are the names of operators, condition codes, control register symbols, assembly language instructions, high-level statement key, and assembler directives. The specific reserved words are listed below.

NAMES OF OPERATORS:

HI	LXOR
LAND	MOD
LNOT	SHL
LO	SHR
LOR	SIZEOF

CONDITION CODES:

C	MI	UGE
EQ	NC	UGT
F	NE	ULE
GE	NOV	ULT
GT	NZ	Z
LE	OV	
LT	PL	

CONTROL REGISTER SYMBOLS:

FLAGS	PRE1	RP	SPL
IMR	P01M	SIO	TMR
IPR	P2M	SP	T0
IRQ	P3M	SPH	T1
PRE0			

ASSEMBLY-LANGUAGE INSTRUCTIONS:

ADC	DEC	JR	PUSH	SCF
ADD	DECW	LD	RCF	SRA
AND	DI	LDC	RET	SRP
CALL	DJNZ	LDCI	RL	SWAP
CCF	EI	LDE	RLC	SUB
CLR	INC	LDEI	RR	TCM
COM	INCW	NOP	RRC	TM
CP	IRET	OR	SBC	XOR
DA	JP	POP		

When defining symbols, users should also avoid the forms Rn and RRn, where n is a number from 0 to 15.

HIGH-LEVEL STATEMENT KEYWORDS:

ARRAY	END	GLOBAL	LOCAL	REPEAT
3YTE	ENTRY	IF	MODULE	SHORT_INTEGER
:ASE	EXIT	INTEGER	OD	THEN
:ONSTANT	EXTERNAL	INTERNAL	PROCEDURE	TYPE
DO	FI	JPR	RECORD	WORD
:LSE	FROM	LABEL		

3EGMENT NAMES AND NUMBERS:

)ATA	2
?ROGRAM	0
?EGISTER	1

?SEUDO INSTRUCTIONS:

3VAL
/VAL

/ORDS RESERVED FOR FUTURE EXTENSIONS:

\DCW	CPW	LDEW	PUSHW
DDW	DWJNZ	LDW	SBCW
NDW	LDCIW	ORW	SUBW
:LRW	LDCW	POPW	XORW
:OMW	LDEIW		

).2 Special Characters

'he list of special characters below includes delimiters and pecial symbols. The difference between them is that delimiters ave no semantic significance (for example, two PLZ/ASM tokens 'an have any number of blanks separating them), whereas special ymbols do have semantic meaning (for example, # is used to ndicate an immediate value).

'he class of delimiters includes the space (blank), tab, form eed, line feed, carriage return, semicolon (;), and comma (,). 'he comment construct enclosed in exclamation points (!) is also onsidered a delimiter.

The special symbols and their uses are as follows:

+	Binary addition; unary plus.
-	Binary subtraction; unary minus.
*	Unsigned multiplication; dimension specifier for list (one-dimensional array) initialization.
/	Unsigned division.
:	Label terminator.
:=	Constant and variable initialization.
%	Nondecimal number base specifier; special character specifier within quoted character sequence.
#	Immediate data specifier.
@	Indirect address specifier.
$	Current contents of location counter; specifies special LOCAL statement labels; precedes assembler directives.
[]	Enclose components of ARRAY or RECORD definition; enclose index part of ARRAY reference; enclose initialization values.
()	Enclose expressions selectively; enclose octal or binary number base indicator; enclose index part of indexed address.
.	Separates RECORD name from field name in RECORD field reference.
<	"Less than" operator
>	"Greater than" operator.
=	"Equal" operator.
<>	"Not equal" operator.
<=	"Less than or equal" operator.
>=	"Greater than or equal" operator.

Appendix E
ASCII Character Set

Hex	Dec	Char	Hex	Dec	Char	Hex	Dec	Char	Hex	Dec	Char
00	0	NUL	20	32	SP	40	64	@	60	96	`
01	1	SOH	21	33	!	41	65	A	61	97	a
02	2	STX	22	34	"	42	66	B	62	98	b
03	3	ETX	23	35	#	43	67	C	63	99	c
04	4	EOT	24	36	$	44	68	D	64	100	d
05	5	ENQ	25	37	%	45	69	E	65	101	e
06	6	ACK	26	38	&	46	70	F	66	102	f
07	7	BEL	27	39	'	47	71	G	67	103	g
08	8	BS	28	40	(	48	72	H	68	104	h
09	9	HT	29	41	)	49	73	I	69	105	i
0A	10	LF	2A	42	*	4A	74	J	6A	106	j
0B	11	VT	2B	43	+	4B	75	K	6B	107	k
0C	12	FF	2C	44	,	4C	76	L	6C	108	l
0D	13	CR	2D	45	-	4D	77	M	6D	109	m
0E	14	SO	2E	46	.	4E	78	N	6E	110	n
0F	15	SI	2F	47	/	4F	79	O	6F	111	o
10	16	DLE	30	48	0	50	80	P	70	112	p
11	17	DC1	31	49	1	51	81	Q	71	113	q
12	18	DC2	32	50	2	52	82	R	72	114	r
13	19	DC3	33	51	3	53	83	S	73	115	s
14	20	DC4	34	52	4	54	84	T	74	116	t
15	21	NAK	35	53	5	55	85	U	75	117	u
16	22	SYN	36	54	6	56	86	V	76	118	v
17	23	ETB	37	55	7	57	87	W	77	119	w
18	24	CAN	38	56	8	58	88	X	78	120	x
19	25	EM	39	57	9	59	89	Y	79	121	y
1A	26	SUB	3A	58	:	5A	90	Z	7A	122	z
1B	27	ESC	3B	59	;	5B	91	[	7B	123	{
1C	28	FS	3C	60	<	5C	92	\	7C	124	|
1D	29	GS	3D	61	=	5D	93	]	7D	125	}
1E	30	RS	3E	62	>	5E	94	^	7E	126	~
1F	31	US	3F	63	?	5F	95	_	7F	127	DEL

Index

Zilog

READER COMMENTS

Your comments concerning this publication are important to us. Please take the time to complete this questionnaire and return it to Zilog.

Title of Publication: ____________________

Document Number: ____________________

Your Hardware Model and Memory Size: ____________________

Describe your likes/dislikes concerning this document:

Technical Information: ____________________

Supporting Diagrams: ____________________

Ease of Use: ____________________

Your Name: ____________________

Company and Address: ____________________

Your Position/Department: ____________________

No Postage Necessary If Mailed In The United States

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 35, CAMPBELL, CA.

POSTAGE WILL BE PAID BY:

Zilog

1315 Dell Ave.
Campbell, California 95008
ATTENTION: Corporate Publications

Zilog Sales Offices and Technical Centers

West

Sales & Technical Center
Zilog, Incorporated
1315 Dell Avenue
Campbell, CA 95008
Phone: (408) 370-8120
TWX: 910-338-7621

Sales & Technical Center
Zilog, Incorporated
18023 Sky Park Circle
Suite J
Irvine, CA 92714
Phone: (714) 549-2891
TWX: 910-595-2803

Sales & Technical Center
Zilog, Incorporated
15643 Sherman Way
Suite 430
Van Nuys, CA 91406
Phone: (213) 989-7485
TWX: 910-495-1765

Sales & Technical Center
Zilog, Incorporated
1750 112th Ave. N.E.
Suite D161
Bellevue, WA 98004
Phone: (206) 454-5597

Midwest

Sales & Technical Center
Zilog, Incorporated
951 North Plum Grove Road
Suite F
Schaumburg, IL 60195
Phone: (312) 885-8080
TWX: 910-291-1064

Sales & Technical Center
Zilog, Incorporated
28349 Chagrin Blvd.
Suite 109
Woodmere, OH 44122
Phone: (216) 831-7040
FAX: 216-831-2957

South

Sales & Technical Center
Zilog, Incorporated
4851 Keller Springs Road,
Suite 211
Dallas, TX 75248
Phone: (214) 931-9090
TWX: 910-860-5850

Zilog, Incorporated
7113 Burnet Rd.
Suite 207
Austin, TX 78757
Phone: (512) 453-3216

East

Sales & Technical Center
Zilog, Incorporated
Corporate Place
99 South Bedford St.
Burlington, MA 01803
Phone: (617) 273-4222
TWX: 710-332-1726

Sales & Technical Center
Zilog, Incorporated
240 Cedar Knolls Rd.
Cedar Knolls, NJ 07927
Phone: (201) 540-1671

Technical Center
Zilog, Incorporated
3300 Buckeye Rd.
Suite 401
Atlanta, GA 30341
Phone: (404) 451-8425

Sales & Technical Center
Zilog, Incorporated
1442 U.S. Hwy 19 South
Suite 135
Clearwater, FL 33516
Phone: (813) 535-5571

Zilog, Incorporated
613-B Pitt St.
Cornwall, Ontario
Canada K6J 3R8
Phone: (613) 938-1121

United Kingdom

Zilog (U.K.) Limited
Zilog House
43-53 Moorbridge Road
Maidenhead
Berkshire, SL6 8PL England
Phone: 0628-39200
Telex: 848609

France

Zilog, Incorporated
Cedex 31
92098 Paris La Defense
France
Phone: (1) 334-60-09
TWX: 611445F

West Germany

Zilog GmbH
Eschenstrasse 8
D-8028 TAUFKIRCHEN
Munich, West Germany
Phone: 89-612-6046
Telex: 529110 Zilog d.

Japan

Zilog, Japan K.K.
Konparu Bldg. 5F
2-8 Akasaka 4-Chome
Minato-Ku, Tokyo 107
Japan
Phone: (81) (03) 587-0528
Telex: 2422024 A/B: Zilog J

Zilog, Inc. 1315 Dell Ave., Campbell, California 95008 Telephone (408)370-8000 TWX 910-338-7621

03-3023-03 Printed in USA