# Z8000™ CPU

# Technical Manual

January 1983

Zilog

# Z8000 CPU
# Technical Manual

Zilog

# Table Of Contents

**Chapter 3. Address Spaces**

**Chapter 4. CPU Operation**

# Table Of Contents (Continued)

6

**Chapter 7. Exceptions**

7

**Chapter 8. Refresh**

8

**Chapter 9. External Interface**

9

9

**Chapter 10. Programming Techniques** . . . . . . . . . . . . . . . . . . . . . .

10

## Table of Contents (Continued)

**List of Illustrations**

# Table Of Contents (Continued)

**List of Tables**

# Chapter 1
# Z8000 Processor Overview

## 1.1 INTRODUCTION

This chapter provides a summary description of the advanced architecture of the Z8000 microprocessors with special attention given to those architectural features that set the Z8000 CPUs apart from their predecessors. A complete overview of the architecture is provided in Chapter 2, with detailed descriptions of the various aspects of the processor provided in succeeding chapters.

## 1.2 GENERAL ORGANIZATION

Zilog's Z8000 microprocessors have been designed to accommodate a wide range of applications, from the relatively simple to the large and complex. The Z8000 CPUs are offered in four versions: the Z8001, the Z8002, the Z8003, and the Z8004. The CPUs come with an entire family of support components: two memory management units, a DMA controller, serial and parallel I/O controllers, and extended processing units--all compatible with Zilog's Z-BUS. Together with other Z8000 Family components, the advanced CPU architecture in an LSI microprocessor design provides the flexibility and the features usually associated with a mini- or mainframe computer.

The major architectural features of the Z8000 CPU that enhance throughput and processing power are a general purpose register file, System and Normal modes of operation, multiple addressing spaces, a powerful instruction set, numerous addressing modes, multiple stacks, flexible interrupt structure, a rich set of data types, and separate I/O address spaces. In addition the Z8001 and Z8003 offer a large address space and segmented memory addressing. Both the Z8003 and Z8004 CPUs include provisions for the implementation of virtual memory systems, and enhanced test and set operations.

These architectural features combine to produce a powerful, versatile microprocessor. The benefits that result from these features are code density, compiler efficiency and support for typical operating system operations and complex data structures. These topics are treated in Section 1.3.

The Z8000 CPUs have been designed so that a powerful memory management system (discussed in Section 1.3.12) can be used to improve the utilization of the main memory, to implement a virtual memory system, and to provide protection capabilities for the system. Although memory management is an optional capability-the Z8000 CPU is a powerful processor without it-the CPU has explicit features to facilitate integrating an external memory management device into a Z8000 system configuration.

Finally, care has been taken to provide a general mechanism for extending the basic instruction set through the use of external devices (called Extended Processing Units--EPUs). In general, an EPU is dedicated to performing complex and time-consuming tasks so as to unburden the system's CPU. Typical tasks for specialized EPUs include floating-point arithmetic, data base search and maintenance operations, and network interfaces. This topic is treated in Section 1.5.

The overall design of the Z8000 CPUs provides the user with a powerful, low-cost, highly adaptable, CPU in a 40 or 48-pin package.

## 1.3 ARCHITECTURAL FEATURES

The architectural resources of a Z8000 CPU include sixteen 16-bit general-purpose registers, seven data types ranging from bits to 32-bit long words, to word and byte strings, eight user-selectable addressing modes, and an instruction set more powerful than that of most minicomputers. The 110 distinct instruction types combine with the various data types and addressing modes to form a set of 414 instructions. Moreover, the instruction set exhibits a high degree of regularity: more than 90% of the instructions can use any of five main addressing modes with 8-bit byte, 16-bit word, and 32-bit long-word data types.

The CPUs generate output status signals that indicate the nature of the bus transaction that is being attempted. These signals can be used to implement systems with multiple address spaces--memory areas dedicated to specific uses. The CPUs also have two operating modes, System and Normal, which can be used to separate operating system functions from normal applications processes. I/O operations have been separated from memory accesses, further enhancing the capability and integrity of Z8000-based systems, and a flexible interrupt structure facilitates the efficient operation of peripheral I/O devices. Moreover, the Extended Processing Unit (EPU) capability of the Z8000 allows CPUs to unload time-consuming tasks onto external devices.

Special features have been introduced to facilitate the implementation of multiple processor systems. In addition, the Z8001 and Z8003 CPUs have a large addressing capability that greatly extends the applicability of microprocessors to large system applications.

### 1.3.1 General-Purpose Register File

The heart of the Z8000 CPU architecture is a file of sixteen 16-bit general-purpose registers. These general-purpose registers give the Z8000 its power and flexibility and add to its regular structure.

General-purpose registers can be used as accumulators, memory pointers, or index registers. Their major advantage is that, as the needs of the program change, the particular use to which they are put can vary during the course of a program. Thus the general-purpose register file avoids the critical bottlenecks of an implied or dedicated register architecture, in which the contents of dedicated registers must be saved and restored when more registers of a particular type are needed than are supplied by the processor.

The Z8000 CPU register file can be addressed in several ways: as 16 byte registers (occupying the upper half of the file) or as 16 word registers, or, by using the register-pairing mechanism, as eight long-word (32-bit) registers or as four quadruple-word (64-bit) registers. Because of this register flexibility, registers can be used efficiently in the Z8000. For example, it is not necessary for a Z8000 user to dedicate a 32-bit register to hold a byte of data.

### 1.3.2 Instruction Set

A powerful instruction set is one of the distinguishing characteristics of the Z8000. The instruction set is one measure of the flexibility and versatility of a computer. Having a given operation implemented in hardware saves memory and improves speed. In addition, completeness of the available operations on a particular data type is frequently more important than additional instructions that are unlikely to affect performance significantly. The Z8000 CPU provides a full complement of arithmetic, logical, branch, I/O, shift, rotate, and string instructions. In addition, special instructions have been included to facilitate multiprocessing, multiple processor configurations, and typical high-level language and operating-system functions. The general philosophy of the instruction set is two-operand register-memory operations, which include, as a special subset, register-register operations. However, to improve code density, a few memory-memory operations are used for string manipulation. The two-address format reflects the most frequently occurring operations (such as A <-- A + B). Also, having one of the operands in a rapidly accessible general-purpose register facilitates the use of intermediate results generated during a calculation.

The majority of operations deal with byte, word, or long-word operands, thereby providing a high degree of regularity. Compact, one-word instructions for the most frequently used operations, such as branching short distances in a program, are also included in the instruction set.

The instruction set contains notable additions to the standard repertoire of earlier microprocessors. The Load and Exchange group of instructions has been expanded to support operating system functions and conversion of existing microprocessor programs. The usual arithmetic instructons can now deal with higher-precision operands, while hardware multiply and divide instructions have also been added. The Bit Manipulation instructions can use calculated values of assembled constants to specify the bit position within a byte or word. The Rotate and Shift instructions are considerably more flexible than those of previous microprocessors. The String instructions include one designed specifically for translating between different character codes. Multiple-processor configurations are supported by special instructions.

### 1.3.3 Data Types

Many data types are supported by the Z8000 architecture. A data type is supported when it has a hardware representation and instructions that directly apply to it. New data types can always be simulated in terms of basic data types, but hardware support provides faster and more convenient operations. The basic data type is the byte, which is also the basic addressable element. The architecture also supports the following data types: words (16 bits), long words (32 bits), byte strings, and word strings. In addition, bits are fully supported and addressed by number within a byte or word. BCD digits are supported and represented as two 4-bit digits in a byte. Arrays are supported by the Indexed addressing mode (see 1.3.4 and Chapter 5). Stacks are supported by the instruction set and by external devices (Memory Management Units) available to be used with the segmented Z8000 CPUs.

### 1.3.4 Addressing Modes

The addressing mode, which is the way an operand is specified in an instruction, determines how an address is generated. The Z8000 CPU offers eight addressing modes. Together with the large number of instructions and data types, they improve the processing power of the CPU. The addressing modes are Register, Immediate, Indirect Register, Direct Address, Index, Relative Address, Base Address, and Base Index. Several other addressing modes including autoincrement and autodecrement are implied by specific instructions, The first five modes listed above are the basic addressing modes that are used most frequently and apply to most instructions having more than one addressing mode. In the Z8002 and Z8004, Base Address and Index modes are identical.

### 1.3.5 Multiple Memory Address Spaces

The Z8000 CPU facilitates the use of multiple address spaces. When the Z8000 CPU generates a memory address, it also outputs signals indicating the particular internal activity that led to the memory request: instruction fetch, operand reference, or stack reference. This information can be used in two ways: to increase the memory space available to the processor (for example, by putting programs in one space and data in another), or to protect portions of the memory and allow only certain types of access (for example, by allowing only instruction fetches from an area designated to contain proprietary software). The Memory Management Units (MMUs) have been designed to provide precisely these kinds of protection features by using the CPU-generated status information.

### 1.3.6 System and Normal Modes of Operation

All Z8000 CPUs can run in either System mode or Normal mode. In System mode, all instructions can be executed and all CPU registers can be accessed. This mode is intended for use by programs performing operating system functions. In Normal mode, some instructions cannot be executed (e.g., I/O operations), and the control registers of the CPU are inaccessible. In general, this mode of operation is intended for use by application programs. This separation of CPU resources promotes the integrity of the system, since programs operating in Normal mode cannot access those aspects of the CPU that deal with time-dependent or system-interface events.

Programs that produce erroneous results when executing in Normal mode can usually reproduce those errors for debugging purposes simply by re-executing the program with its original data. Programs using facilities available only in System mode are more likely to encounter errors that are due to timing considerations (e.g., based upon the frequency of disk requests and disk arm-position). Such problems are difficult to debug because these errors are not easily reproduced. Thus, a preferred method of program development is to partition the task into a portion which can be performed without those resources accessible only in System mode (which will usually be the bulk of the task) and a portion requiring system mode resources. The classic example of this partitioning comes from current minicomputer and mainframe systems: the operating system runs in System mode and the individual users write their programs to run in Normal mode.

To support the System/Normal mode dichotomy, there are two copies of the stack pointer--one for a System mode stack and another for a Normal mode stack. These two stacks facilitate the task switching involved when interrupts or traps occur. To ensure that the Normal stack is free of system information, the information saved on the occurrence of interrupts or traps is always pushed onto the System stack before the new program status is loaded.

### 1.3.7 Separate I/O Address Spaces

The Z8000 architecture distinguishes between memory and I/O spaces by providing specific I/O instructions. This architectural separation allows better protection and has more potential for extension than "memory-mapped" I/O in which I/O and memory references share the same address space. There are two separate I/O address spaces: Standard I/O and Special I/O. The main purpose of these two spaces is to provide the Z8010 MMU, which is connected only to the high-order byte of the AD bus, with its own I/O address space.

Memory-mapped I/O is still possible at the implementor's option. It can be implemented simply by ignoring the I/O instructions.

### 1.3.8 Interrupt Structure

The flexible interrupt structure of the Z8000 allows the processor to continue performing useful work while waiting for peripheral events to occur. The elimination of periodic polling and idling loops (typically used to determine when a device is ready to transmit data) increases throughput.

The CPU supports three types of interrupt. A nonmaskable interrupt represents an event that requires immediate handling to preserve system integrity. In addition, there are two types of maskable interrupt: nonvectored interrupts and vectored interrupts. The latter provide an automatic call to interrupt processing routines, depending on the vector presented by the peripheral to the Z8000.

The Z8000 has a priority system for handling interrupts. Vectored interrupts have higher priority than non-vectored interrupts among devices attached to one of these interrupt lines; priority is determined by a daisy chain built into all Z-Bus peripherals. This priority scheme allows the efficient control of many peripheral devices in a Z8000 system.

An interrupt causes information relating to the currently executing program (program status) to be saved on a special system stack with a code describing the reason for the interrupt. This allows recursive task switches to occur while leaving the Normal mode stack undisturbed by system information. The address of the interrupt processing routine and the associated contents of the FCW Register (new program status) are loaded from a special area in memory, the program status area, designated by a pointer resident in the CPU.

The use of the stack and of a pointer to the program status area is a specific choice made to allow flexibility in system design and to allow architectural compatibility if new interrupts or traps are added to the architecture.

### 1.3.9 Multi-Processing

The Z8000 provides basic mechanisms that allow the sharing of address spaces among different microprocessors. Large segmented address spaces and the support for external memory management make this possible. Also, a resource request bus is provided which, in conjunction with software, provides the exclusive use of shared critical resources. A Test and Set instruction is also provided for the management of access to shared resources. This instruction and its associated output status code are used to prevent more than one processor from accessing a resource at the same time. These mechanisms, and peripherals such as the Z-FIO (FIFO Input/Output Interface Unit), have been designed to allow easy asynchronous communication between different CPUs.

### 1.3.10 Large Address Space for the Z8001 and Z8003

For many applications, a basic address space of 64K bytes is insufficient. A larger address space increases the range of applications of a system by permitting large, complex programs and data sets. A large address space simplifies program and data management. In addition, large address spaces and memories reduce the need for minimizing program size and permit the use of higher-level languages. The segmented versions of the Z8000 (Z8001 and Z8003) generate 23-bit addresses, for a basic address space of 8 megabytes (8M or 8,388,608 bytes).

Both the Z8003 and Z8004 CPUs also offer features that aid the implementation of virtual memory. The Z8003, in particular, when used with the Z8015 MMU which is designed for management of paged virtual memories, can implement an apparently unlimited amount of address space organized in fixed-sized (2K byte) pages. This paged virtual memory capability combines the benefits of a large virtual address space (ease of programming) with the benefits of a small physical memory (low cost).

### 1.3.11 Segmented Addressing

The segmented versions of the Z8000 CPU (i.e., Z8001 and Z8003) divide each 23-bit addresses into a 7-bit segment number and a 16-bit segment offset. The segment number serves as the logical name of a segment; it is not altered by the effective address calculation (by indexing, for example). This corresponds to the way memory is typically used by a program--one portion of the memory is set aside to hold instructions, another for data. In a segmented address space, the instructions could reside in one segment (or several different modules in different segments), and each data set could reside in a separate segment. One advantage of segmentation is that in systems that use external memory management, it speeds up address calculation and relocation. Thus, segmentation allows the use of slower memories than linear addressing schemes allow. In addition, segments provide a convenient way of partitioning memory so that each partition is given particular access attributes (for example, read-only). The Z8000 approach to segmentation (simultaneous direct access to the entire 8M byte address space) does not require the use of segment registers or other forms of addressing overhead.

### 1.3.12 Memory Management

Memory management is used primarily for the dynamic relocation, protection, and sharing of memory. It offers the following advantages: allowing a logical structure to the memory space that is independent of the actual physical location of data, protecting the user from mistakes, preventing unauthorized access to memory resources or data, and protecting the operating system from disruption by the users.

The addresses manipulated by the programmer, used by instructions, and output by the segmented Z8000 CPU are called logical addresses. The external memory management system takes the logical addresses and transforms them into the physical addresses required for accessing the memory. This address transformation process is called relocation. This process makes user software independent of the physical memory. Thus, the user is freed from specifying where information is actually located in the physical memory.

The segmented Z8000 CPUs support memory management both with segmented addressing and with program-status information. A segmented addressing space allows individual segments to be treated differently.

Program status information generated by the CPU permits an external memory management device to monitor the intended use of each memory access. Thus, illegal types of access can be suppressed and memory segments can be protected from unintended or unwanted modes of use. For example, system tables could be protected from direct user access. This added protection capability becomes more important as microprocessors are applied to large, complex tasks.

### 1.3.13 Virtual Memory Capability

Both the Z8003 and Z8004 CPUs are provided with features that support the use of a virtual memory system. A virtual memory system permits programs to reference an address space that exceeds the size of main (physical) memory.

In virtual memory systems, high-speed main memory is supported by medium and low-speed secondary storage devices such as hard disks or floppy disks. When the CPU in a virtual system issues an address that references a location that is not currently stored in main memory, the current operation must be aborted, a secondary storage access must be performed to retrieve and load into main memory block of memory containing the referenced location. The mainstream program must then be restarted at the point of interruption. The secondary storage access and restart operations are invisible to both the user and the executing program. The system, therefore, appears to have a memory that is not constrained by the physical size of main memory.

The maximum size of virtual memory is determined by the address structure used and by the capabilities of the memory management system used. Zilog provides a memory management chip (Z8015) designed specifically to implement a paged virtual memory system and a segmented MMU for segmented virtual memory (see Appendix B).

## 1.4 BENEFITS OF THE ARCHITECTURE

The features of the Z8000 Architecture combine to provide several significant benefits:

- high-density code
- efficient compilation
- operating system support
- structural data manipulations

### 1.4.1 Code Density

Code density affects both processor speed and memory utilization. Compaction of code saves memory space--an especially important factor in smaller systems--and improves processor speed by reducing the number of instruction words that must be fetched and decoded. The Z8000 offers several advantages with respect to code density. Code density is achieved in part by the use of special "short" formats for certain instructions that statistical analysis shows to be the most frequently used. Short formats decrease the amount of memory required to store instructions.

A "short offset" mechanism has been provided to allow a 2-word segmented address to be reduced to a single word; this format may be used by both assemblers and compilers.

The largest reduction in program size (and consequent increase in speed) results from the consistent and regular structure of the architecture and from the powerful Z8000 instruction set--factors that substantially reduce the number of instructions required for a task. The architecture is more regular than preceding microprocessors, because its registers, addressing modes, and data types can be used in an orderly fashion. Any general-purpose register can be specified as an accumulator, index register, or base register. With a few exceptions, all basic addressing modes can be used with all instructions, as can the various data types.

General-purpose registers do not have to be changed as often as special-purpose registers. This too reduces program size, since frequent load and store operations are not required.

### 1.4.2 Compiler Efficiency

For microprocessor users, the transition from assembly language to high-level languages allows greater freedom from architectural dependency and improves ease of programming. However, rather than adapting the architecture to a particular high-level language, the Z8000 was designed as a general-purpose microprocessor. (Tailoring a processor for efficiency in one language often leads to inefficiency in other languages.) For the Z8000, language support has been provided through the inclusion of features designed to minimize typical compilation and code-generation problems. Among these features is the regularity of the Z8000 addressing modes and data types. Access to arguments and local variables on a procedure stack is supported by the "Indexed With Short Offset" addressing mode, as well as the Based and Base Indexed addressing modes. In addition, address arithmetic is aided by the Increment and Decrement instructions.

Testing of data, logical evaluation, initialization, and comparision of data are made possible by the instructions Test, Test Condition Codes, Load Immediate Into Memory, and Compare Immediate With Memory. Since compilers and assemblers frequently manipulate character strings, the instructions Translate, Translate and Test, Block Compare, and Compare String all result in speed improvements over software simulations of these tasks. In addition, any register except register R0 can be used as a stack pointer by the Push and Pop instructions.

### 1.4.3 Operating System Support

Interrupt, task-switching, and memory-management and compiler-support features improve operating system implementation.

The interrupt structure has three levels: nonmaskable, nonvectored, and vectored. When an interrupt occurs, the program status is saved on the stack with an indication of the reason for this state-switching; then a new program status is loaded from a special area of memory. The program status consists of a flag register, the control bits, and the program counter. The reason for the occurrence is encoded in a 16-bit "vector" that is supplied by the interrupting device and read from the system bus and saved on the stack by the CPU. In the case of a vectored interrupt, one byte of the vector also indexes a table of interrupt processing routine addresses.

The inclusion of System and Normal modes improves operating system organization. In the System mode, all operations are allowed; in the Normal mode, certain instructions are prohibited. The System Call instruction allows a controlled switch of mode, and the Privileged instruction trap enforces these restrictions.

Traps result in the same type of program status-saving as interrupts: in both cases, the information saved is pushed onto the system stack leaving the normal stack undisturbed. The Load Multiple instruction allows the contents of registers to be saved efficiently in memory or on the stack when performing a task switch. Programs, during execution in System mode, can cause program status changes under direct software control by using the Load Program Status instruction.

Finally, process exclusion and serialization can be achieved with the Test And Set instruction which synchronizes asynchronous cooperating processes.

A new feature has been added to the Z8003 and Z8004 CPUs; when TSET is executed by these CPUs a special status code is generated. This code allows external circuitry to prevent access to the memory location (semaphore) addressed by TSET between TSET's read and write operation with that semaphore. This access protect function aids in the synchronization of CPU operations in multiprocessor systems.

### 1.4.4 Support for Many Types of Data Structures

A data structure is a logical organization of primitive elements (byte, word, etc.) whose format and access conventions are defined. Common data structures include arrays, lists, stacks, and strings. Since data structures are high-level constructs frequently used in programming, processor performance is enhanced if the CPU provides mechanisms for their efficient manipulations; such mechanisms are offered by Z8000 CPUs.

In many applications, one of the most frequently encountered data structures is the array. Arrays are supported in the Z8000 by the Index and the Base Index Addressing modes and by segmented addressing. The Base Index Addressing mode allows the use of pointers into an array (i.e., offsets from the array's starting address). Segmented addressing allows an array to be assigned to one segment, so that it can be referenced simply by segment number.

Lists occur more frequently than arrays in business applications and in general data processing. Lists are supported by the Indirect Register and Base Address Addressing modes. The Base Index Addressing mode is also useful for more complex lists.

Stacks are used in all applications for nesting of routines, block-structured languages, and interrupt handling. Stacks are supported by the Push and Pop instructions, and multiple stacks may be implemented based on the general-purpose registers of the Z8000. In addition, two hardware stack pointers are used to assign separate stacks to System and Normal operating modes, thereby further supporting the separation of the system and normal operating environments discussed earlier. The Z8010 and Z8015 MMUs provide special provisions to provide stack overflow and to allow dynamic expansion of stacks.

Byte strings are supported by the Translate and Translate And Test instructions. Decimal arithmetic on strings of BCD data, packed two characters per byte is supported by the Add/Subract Byte with Carry and Decimal Adjust instructions. The Rotate Digit instructions also manipulate 4-bit data.

### 1.4.5 Four CPU Versions: Differences

There are four versions of the Z8000 CPU: Z8001, Z8002, Z8003, and Z8004. The primary differences among these CPUs are summarized in Table 1-1; details of these differences are given throughout this manual.

A major consideration in selecting a Z8000 CPU is the amount of physical memory that can be addressed directly. For users who do not require a large address space, the nonsegmented versions (Z8002 and Z8004) of this CPU provide the capability of addressing up to 64K bytes of physical memory in each address space, or up to 256K of program/data and system/normal separations are used (plus two 64k byte I/O address spaces separate from memory).

For users who require larger amounts of memory, the segmented versions (Z8001 and Z8003) of this CPU provide the basic capability of addressing up to 8M bytes of physical memory in each address space.

For users who want a small amount of physical memory relative to the size of their data/program address space, the Z8003 and Z8004 CPUs can be used to implement virtual memory systems.

Features provided by the Z8000 CPUs enable them to be directly incorporated into multiprocessor configurations. The ability of segmented CPUs to

**Table 1.1. Z8000 CPUs, Summary of Differences**

| | Z8001 | Z8002 | Z8003 | Z8004 |
|---|---|---|---|---|
| Addressing Spaces | | | | |
| a. Segmented | Yes | No | Yes | No |
| b. Nonsegmented | Yes | Yes | Yes | Yes |
| Number of Output Address Bits | 23 | 16 | 23 | 16 |
| Virtual Memory Input Pin ($\overline{\text{ABORT}}$) | No | No | Yes | Yes |
| Separate External Interrupt Input Pin for Access Violation Signal from MMU | Yes | No | Yes | No |
| TSET Instruction Enhancement | No | No | Yes | Yes |
| Package Size (Pins) | 48 | 40 | 48 | 40 |

execute, without modification, code written for the nonsegmented CPUs enable several of the nonsegmented CPUs to be used with one segmented CPU to form a multiprocessor system.

## 1.5 EXTENDED INSTRUCTION FACILITY

The Z8000 architecture has a mechanism for extending the basic instruction set through the use of external devices. Special opcodes are used with this feature. When the CPU encounters an instruction with one of these opcodes in its instruction stream, it will perform any indicated address calculation and data transfer; otherwise, it will treat the "extended instruction" as being executed by the external device. Fields have been set aside in these extended instructions to be interpreted by external devices (Extended Processing Units--EPUs) as opcodes. Thus, by using appropriate EPUs, the instruction set of the Z8000 can be extended to include specialized instructions.

In general, an EPU is dedicated to performing complex and time-consuming tasks in order to unburden the CPU. Typical tasks suitable for specialized EPUs include floating-point arithmetic, data base search and maintenance operations, network interfaces, and graphics support operations.

## 1.6 SUMMARY

The architectural sophistication of the Z8000 microprocessor is on a level comparable with that of the minicomputer. Features of the Z8000 such as large address spaces, multiple memory spaces, segmented addresses, and support for virtual memory systems and multiple processors are beyond the capabilities of the traditional minicomputer. The benefits of this sophisticated architecture--code density, compiler support, and operating system support--greatly enhance the power and versatility of the Z8000 CPU. The CPU features that support external memory management systems also enhance the CPU's applicability to large system environments.

# Chapter 2
# Architecture

## 2.1 INTRODUCTION

This chapter provides an overview of the Z8000 CPU architecture. The basic hardware requirements, operating modes, and instruction set are all described. Differences among the versions of the Z8000 are noted where appropriate. Most of the subjects covered here are also treated in greater detail in later chapters of this manual.

## 2.2 GENERAL ORGANIZATION

Figure 2-1 contains a block diagram that shows the following major elements of the Z8000 CPU:

- A 16-bit internal data bus, which is used to move addresses and data within the CPU
- A Z-BUS interface, which controls the interaction of the CPU with the outside world
- Sixteen, 16-bit general-purpose registers, which are used to contain addresses and data
- Four special-purpose Program Status Registers, which control the CPU operation
- An Arithmetic and Logic Unit, which is used for manipulating data and generating addresses

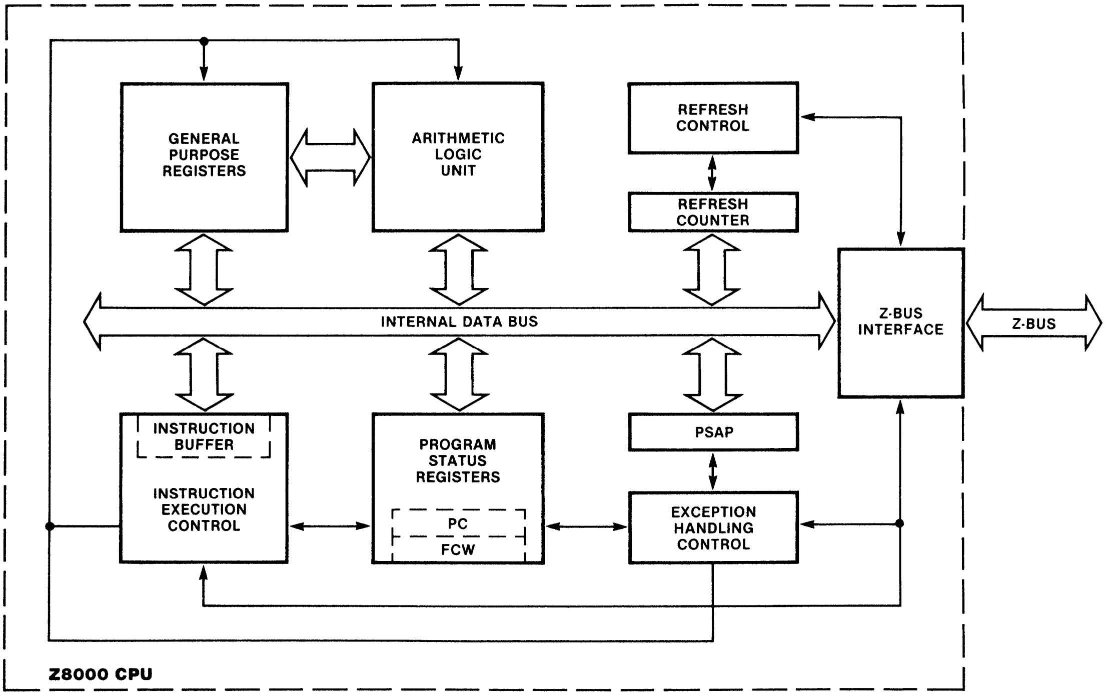


**Figure 2-1. Z8000 CPU Functional Block Diagram**

- An instruction execution controller, which fetches and executes Z8000 instructions
- An exception-handling controller, which processes interrupts and traps
- A refresh controller, which generates memory refresh cycles

Each of these elements is explained in the following sections. All of the elements are common to all of the Z8000 CPUs. The differences between the segmented and nonsegmented versions of the CPUs are derived from the number of bits in the addresses they generate. The Z8002 and Z8004 always generate 16-bit linear addresses, while the Z8001 and Z8003 always generate 23-bit segmented addresses (that is, an address composed of a 7-bit segment number and a 16-bit offset).

Figure 2-2 gives a system-level view of the Z8000. It is important to realize that the Z8000 CPU is part of a family of components that have been designed to allow the easy implementation of powerful systems. The major elements of such a system might include:

- The Z-BUS, a multiplexed, address/data bus that links the components of the system
- A Z8000 CPU
- One or more Extended Processing Units (EPUs), which are dedicated to performing specialized, tasks
- A memory sub-system, which in Z8001 or Z8003 systems can include one or more Memory Management Units (MMUs) which offer memory

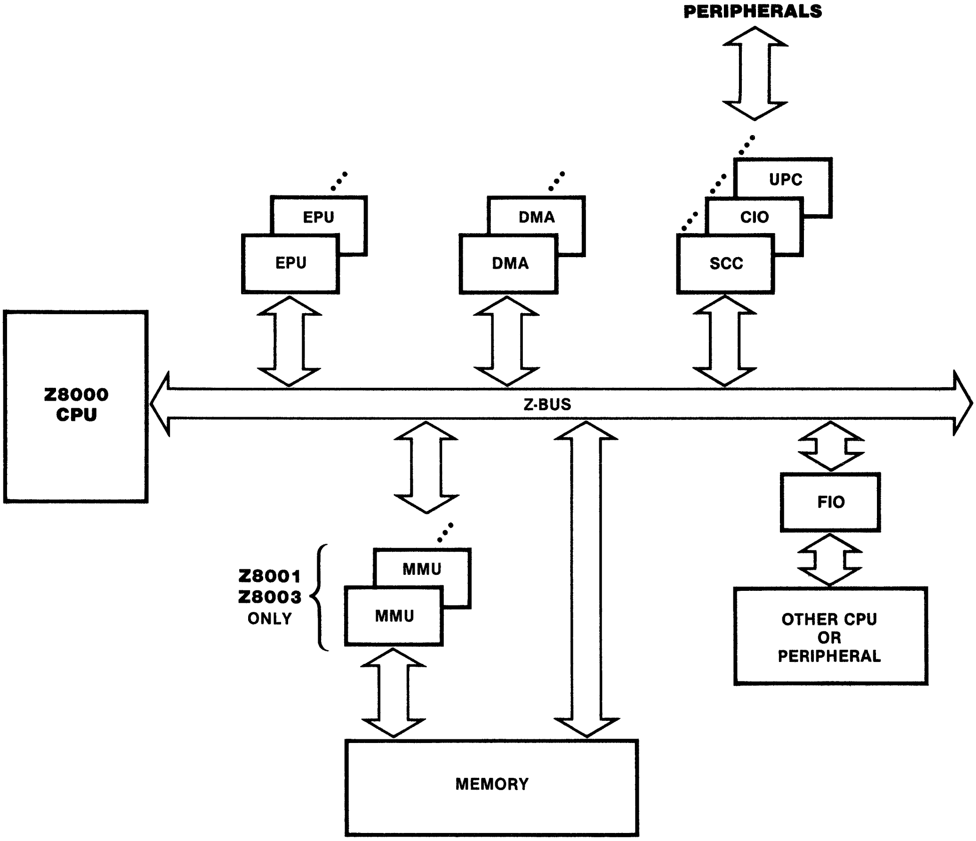


**Figure 2-2. Typical Z8000 System Configuration**

address translation and access protection features

- One or more Data Transfer Controllers (DTCs) for high-speed direct memory access (DMA) data transfers

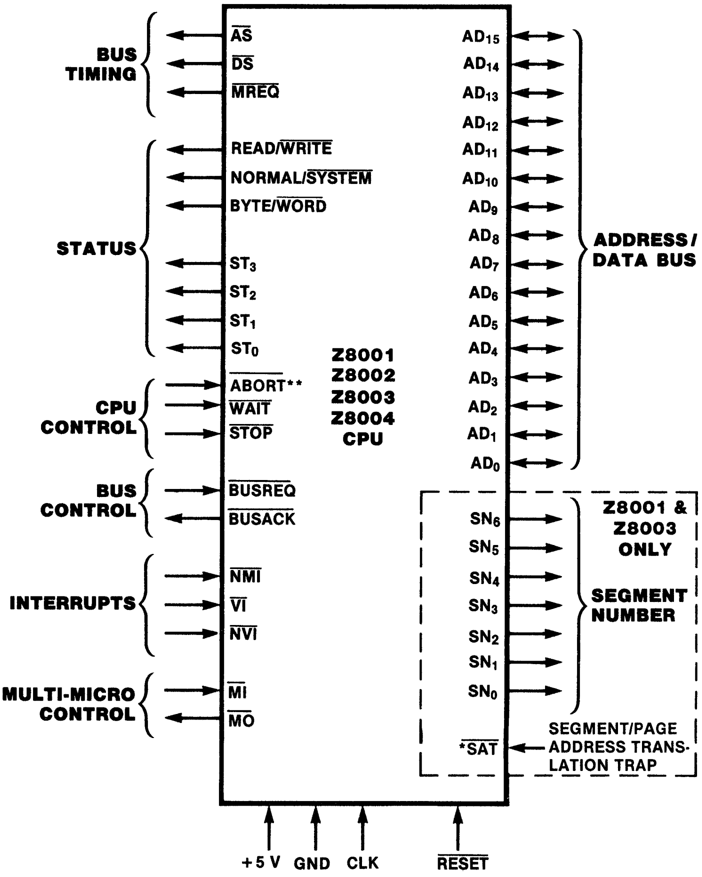
- A large number of possible peripheral devices interfaced to the Z-BUS through Universal Peripheral Controllers (Z-UPCs), Serial Communication Controllers (Z-SCCs), Counter-Timer and Parallel I/O Controllers (Z-CIOs) or other Z-BUS peripheral controllers

- One or more FIFO I/O Interface Units (FIOs) for elastic buffering between the CPU and another device, such as another CPU (not necessarily from the Z8000 family) in a distributed processing system

## 2.3 HARDWARE INTERFACE

Figure 2-3 shows the Z8000 pins grouped according to function. The Z8001 and Z8003 are packaged in 48-pin DIPs and the Z8002 and Z8004 are packaged in 40-pin DIPs. The eight additional pins on the Z8001 and Z8003 are the seven segment-number output lines and the address translation trap input. The address trap is designated as $\overline{\text{SEGT}}$ (Segment Trap) for the Z8001 and as $\overline{\text{SAT}}$ (Segment/Page Address Translation Trap) for the Z8003. Except for those eight pins and the Z8003 and Z8004 $\overline{\text{ABORT}}$ input (which corresponds to an unused pin of the Z8001 and Z8002 CPUs), all pins on the four CPU versions are identical.

The Z8000 is a Z-BUS CPU; thus, activity on its pins is governed by the Z-BUS protocol (see the "Z-BUS Component Interconnect Summary" document



* $\overline{\text{SAT}}$ for Z8003 CPU, $\overline{\text{SEGT}}$ for Z8001 CPU.
** $\overline{\text{ABORT}}$ used ın Z8003 and Z8004 CPUs only.

**Figure 2.3. Z8000 Pin Functions**

No. 00-2031-02). This protocol specifies two types of activities: transactions, which cover all data movement (such as memory references or I/O operations), and requests, which cover interrupts and requests for bus or resource control. The following is a brief overview of the Z8000 pin functions; complete descriptions are found in Chapter 9.

### 2.3.1 Address/Data Lines

These 16 lines alternately carry addresses or data. The addresses may be those of memory locations or I/O ports. The bus timing signal lines described below indicate what kind of information the Address/Data lines are carrying.

### 2.3.2 Segment Number (Z8001 and Z8003 only)

These seven lines encode the address of up to 128 relocatable memory segments. The segment signals become valid one CPU clock period before the address offset signals, thus permitting parallel processing of addresses to be performed by the memory management system and the CPU.

### 2.3.3 Bus Timing

The Address Strobe ($\overline{\text{AS}}$), Data Strobe ($\overline{\text{DS}}$) and Memory Request ($\overline{\text{MREQ}}$) lines are used to signal the beginning of a bus transaction and to determine when the multiplexed Address/Data Bus holds addresses or data. The Memory Request signal can be used to time the transmission of control signals to a memory system.

### 2.3.4 Status Lines

These output lines indicate the kind of transaction on the bus (see Table 2-1.), whether it is a read or a write (R/$\overline{\text{W}}$, High = Read, Low = Write), whether it is on byte or word data (B/$\overline{\text{W}}$, High = byte, Low = word), and whether the CPU is operating in Normal mode or System mode (N/$\overline{\text{S}}$, High = normal, Low = system). Status information defining the type of bus transaction is transmitted in advance of data transmission to allow required external hardware elements to be enabled before data is transferred. The status lines are enabled by the address strobe ($\overline{\text{AS}}$) a minimum of two CPU clock periods before the data ouput lines are sampled (strobed by data strobe $\overline{\text{DS}}$).

**Table 2-1. Status Line Codes**

| $ST_3$-$ST_0$ | Definition |
|---|---|
| 0 0 0 0 | Internal Operation |
| 0 0 0 1 | Memory Refresh |
| 0 0 1 0 | I/O Reference |
| 0 0 1 1 | Special I/O Reference |
| 0 1 0 0 | Segment Trap Acknowledge |
| 0 1 0 1 | Nonmaskable Interrupt Acknowledge |
| 0 1 1 0 | Nonvectored Interrupt Acknowledge |
| 0 1 1 1 | Vectored Interrupt Acknowledge |
| 1 0 0 0 | Data Memory Request |
| 1 0 0 1 | Stack Memory Request |
| 1 0 1 0 | Data Memory Request (EPU) |
| 1 0 1 1 | Stack Memory Request (EPU) |
| 1 1 0 0 | Instruction Space Access |
| 1 1 0 1 | Instruction Fetch, First Word |
| 1 1 1 0 | Transfer between EPU and CPU |
| 1 1 1 1 | Test and Set Data Access<br>(Z8003 and Z8004 only) |

### 2.3.5 CPU Control

These inputs allow external devices to delay the operation of the CPU. The $\overline{\text{WAIT}}$ line, when active (Low), causes the CPU to idle in the middle of a bus transaction, taking extra clock cycles until the $\overline{\text{WAIT}}$ line goes inactive; $\overline{\text{WAIT}}$ is typically input by memory or I/O peripherals that operate more slowly than the CPU. The Stop ($\overline{\text{STOP}}$) line halts internal CPU operation when the first word of an instruction has been fetched. This signal is used for single-step instruction execution during debugging operations and for enabling Extended Processing Units to halt the CPU temporarily. The $\overline{\text{ABORT}}$ line, when active (Low), indicates that external memory management circuitry has detected an address that does not correspond to a location in main memory; this condition causes the CPU to abort the currently executing instruction. When $\overline{\text{ABORT}}$ is enabled, the $\overline{\text{WAIT}}$ input must also be asserted for five CPU clock periods to permit the CPU internal control mechanism to perform the required abort instruction operations. This $\overline{\text{ABORT}}$ input is used in the Z8003 and Z8004 CPUs in the implementation of virtual memory systems.

### 2.3.6 Bus Control

These lines provide the means for other devices, such as direct memory access (DMA) controllers, to gain exclusive use of the signal lines that the CPU would not normally be using to conduct data transfers. The external device requesting control of the bus inputs a bus request ($\overline{\text{BUSREQ}}$); the CPU responds with a bus acknowledge ($\overline{\text{BUSACK}}$) after three-stating, or electrically neutralizing, the Address/Data Bus, Bus Timing lines, Status lines, and Control lines.

### 2.3.7 Interrupts

Three interrupt inputs are provided: nonmaskable interrupts ($\overline{\text{NMI}}$), vectored interrupts ($\overline{\text{VI}}$) and nonvectored interrupts ($\overline{\text{NVI}}$). These inputs permit external devices to cause the CPU to suspend execution of its current program and begin execution of an interrupt service routine.

### 2.3.8 Address Trap Request (Z8001 and Z8003 only)

This input, when used with the Z8001 CPU, is identified as $\overline{\text{SEGT}}$ (Segment Trap). It is used by external memory management circuitry to indicate, when active (Low), that an illegal memory access operation has been detected.

This input, when used with the Z8003 CPU, is identified as $\overline{\text{SAT}}$ (Segment/Page Address Translation Trap). It is used by external memory management circuitry to indicate, when active (Low), that either a referenced segment or page does not reside in memory or that an illegal method of access has been detected.

### 2.3.9 Multi-Micro Control

The Multi-Micro In ($\overline{\text{MI}}$) and Multi-Micro Out ($\overline{\text{MO}}$) lines are used in conjuction with a four-line resource bus and a set of four CPU instructions to coordinate multiple-CPU systems. They allow exclusive use by one CPU of a shared resource in a multiple-CPU system.

### 2.3.10 System Inputs

The four inputs shown at the bottom of Figure 2-3 are: +5 V power, ground, a single-phase clock signal and a CPU reset. The reset function is described in Chapter 7.

## 2.4 TIMING

Figure 2-4 shows the three basic timing periods of a Z8000 CPU: a clock cycle, a bus transaction, and a machine cycle. A clock cycle (sometimes called a T-state) is one cycle of the CPU clock, starting with a rising edge. A bus transaction covers a single data movement on the CPU bus and will last for three or more clock cycles, starting with a falling edge of $\overline{\text{AS}}$ and ending with a rising edge of $\overline{\text{DS}}$. A machine cycle covers one basic CPU

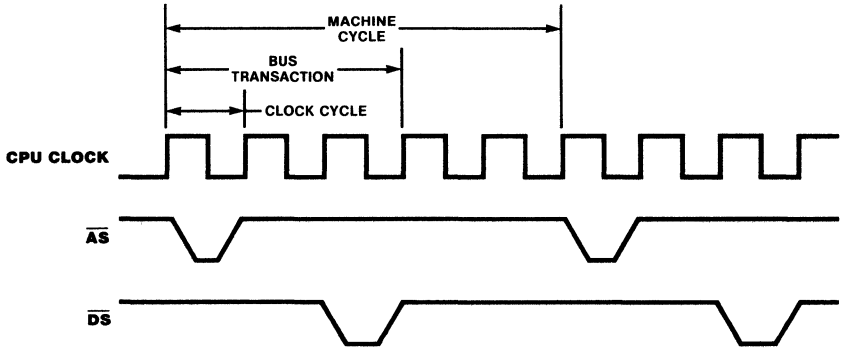


**Figure 2-4. Basic Timing Periods**

operation and always starts with a bus transaction. A machine cycle can extend beyond the end of a transaction by an unlimited number of clock cycles. For more information see Chapter 9.

## 2.5 ADDRESS SPACES

The Z8000 supports two main address spaces that correspond to the two kinds of location that can be addressed:

- **Memory Address Space.** This consists of the addresses of all locations in the main memory of the computer system.

- **I/O Address Space.** This consists of the addresses of all I/O ports through which peripheral devices are accessed.

See Chapter 3 for more information on address spaces.

### 2.5.1 Memory Address Space

Memory address space can be further subdivided, for both Normal and System modes, into Program Memory address space, Data Memory address space, and Stack Memory address space.

The particular space addressed is determined by the external circuitry from the code appearing at the CPU's output status pins ($ST_0$-$ST_3$) and the state of the Normal/System signal ($N/\overline{S}$ pin). Data memory reference and program memory reference each correspond to a different status code at the $ST_0$-$ST_3$ outputs, allowing two address spaces to be distinguished for each operating mode. Each of the address spaces has a range as great as the addressing ability of the processor. For the nonsegmented Z8000 CPUs, each address space can have up to 64K bytes of directly addressable memory. The segmented Z8000 CPUs provide up to 8M bytes of directly addressable memory in each address space.

Segmentation is a memory management technique in which memory is partitioned into variably-sized individually addressed segments. A variety of useful functions can be implemented in segmented memory; the following are examples of such functions:

- Protection mechanisms that prevent a user from referencing data belonging to others, from attempting to modify read-only data, or from

- Virtual memory, which permits a user to write functioning programs as if the system contained more physical memory than is actually available

- Dynamic relocation, which allows the placement blocks of data in physical memory independently of user addresses, allowing better management of the memory resources and sharing of data and programs

The control and status signals provided by segmented Z8000 CPUs assist in implementing these features. However, additional software and external circuitry (such as the Z8010 MMU or Z8015 PMMU ) is generally required to take full advantage of them. See Chapter 3 for an extensive discussion of segmentation.

### 2.5.2 Address Space in Segmented or Segmented/ Paged Virtual Memory Systems

The size of the address space in a virtual memory system is determined by the CPU's address structure and by the capabilities of the system's memory managment hardware and software. Memory Management circuitry external to the CPU can implement either a segmented or paged virtual memory system.

In a segmented system, information is transferred between main and secondary memory on a segment-by-segment basis. Variable length segments of up to 64K bytes in length can be used. Segmented virtual memory systems are supported by the Z8001 and Z8003 CPUs.

In a paged system, each segment is divided into fixed-size pages (standard size is 2048 bytes). Main memory is divided into page-sized "frames", and information is then transferred on a page-by-page basis between main and secondary memory. Paged virtual memory systems are supported by both the Z8003 and Z8004 CPUs.

### 2.5.3 I/O Address Space

I/O addresses are represented as 16-bit words for both the segmented and nonsegmented CPUs.

There are two I/O address spaces, Standard I/O and Special I/O, which are both separate from the memory address space. Each I/O space is accessed through a separate set of I/O instructions, which can be executed only when the CPU is operating in System mode. While these spaces are essentially identical, convention and future compatibility require that Standard I/O instructions transfer data between the CPU and peripherals and Special I/O instructions transfer data to or from external CPU support circuits such as the Z8010 and Z8015 MMUs. Access to Standard or Special I/O space is distinguished by the status lines ($ST_0$-$ST_3$).

## 2.6 GENERAL-PURPOSE REGISTERS

The Z8000 CPU contains 16 general-purpose registers, each 16 bits wide. Any general-purpose register can be used for any instruction operand (except for minor exceptions described at the beginning of Chapter 5).

Figure 2-5 shows these general-purpose registers. They allow data formats ranging from byte to quadruple words. The word registers are specified in assembly-language statements as R0-R15. Sixteen byte registers, RH0-RH7 and RL0-RL7, which can be used as accumulators, overlap the first eight word registers. Register grouping for larger operands produces eight double-word (32-bit) registers, RR0-RR14, which are used for segmented addresses and 32-bit arithmetic instructions, and eight 64-bit registers RQ4,RQ4,RQ8,RQ12, which are used by the Multiply, Divide and Extend Sign instructions.

As Figure 2-5 illustrates, the CPU has two hardware stack pointers, one dedicated to each of the two basic operating modes, System and Normal Segmented Z8000 CPUs (Figure 2-5A) used a two-word stack pointer for each mode (R14'/R15' or R14/R15), whereas the nonsegmented Z8000 CPUs (Figure 2-5B) used only one word for each mode (R15' or R15).

The system stack pointer is used for saving status information when an interrupt or trap occurs and for supporting subroutine calls in System mode. The normal stack pointer is used for subroutine calls in user programs. In Normal-mode operation only the normal stack pointer is accessible. In System-mode operation, the system stack pointer is directly accessed as a general-purpose register. The normal stack pointer can be accessed as a special control register.

## 2.7 SPECIAL-PURPOSE REGISTERS

In addition to the general-purpose registers, there are special-purpose registers. These include the Program Status registers, the Program Status Area Pointer, and the Refresh Counter; they are illustrated for both CPU types in Figure 2-6. Each register can be manipulated in software executing in System mode, and some are modified automatically by certain operations.

### 2.7.1 Program Status Registers

These registers include the Flag and Control Word (FCW) and the Program Counter (PC). They are used to keep track of the state of an executing program.

In the nonsegmented CPUs, Program Status consist of two words: one each for the FCW and the PC. In the segmented CPUs, Program Status consists of four words: one reserved word, one word for the FCW and two words for the segmented PC.

The low-order byte of the Flag and Control Word (FCW) contains the six status flags, from which the condition codes used for control of program looping and branching are derived. The six flags are:

**Carry (C),** which generally indicates a carry out of the high-order bit position of a register being used as an accumulator.

**Zero (Z),** which is generally used to indicate that the result of an operation is zero.

**Sign (S),** which is generally used to indicate that the result of an operation is a negative number.

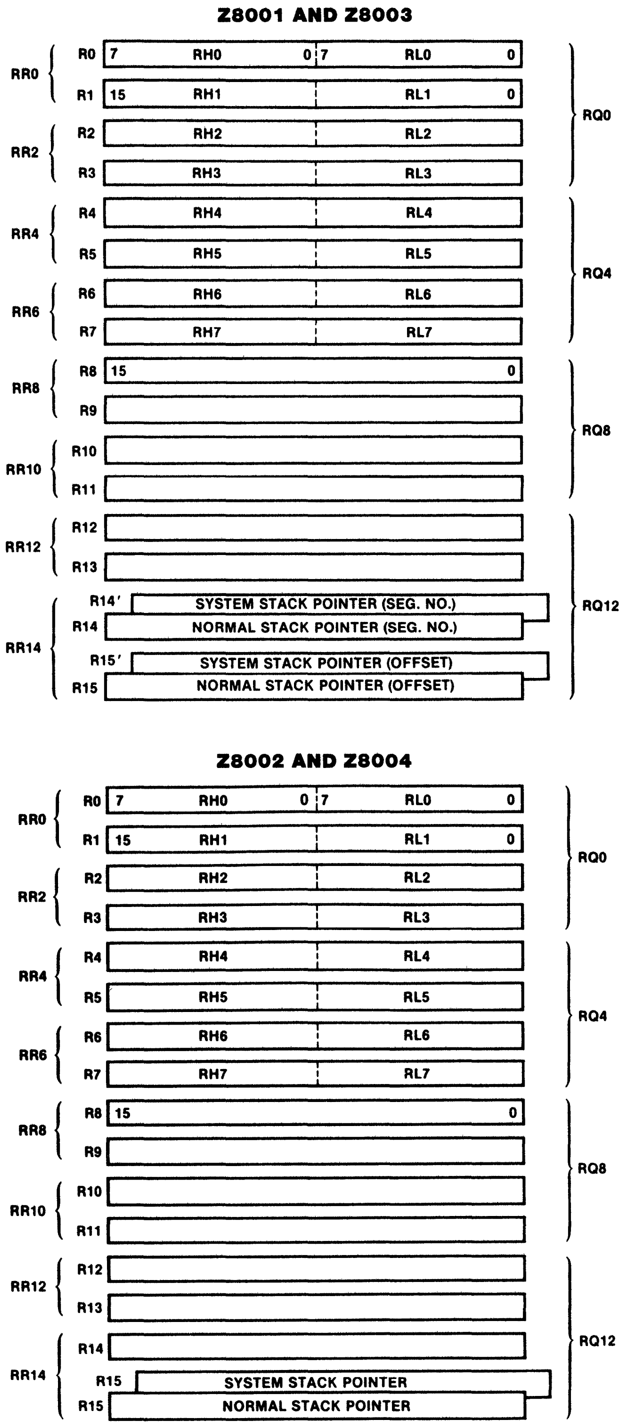
**Parity/Overflow (P/V),** which is generally used to indicate either parity (after logical operations on byte operands) or overflow (after arithmetic operations).

**Decimal-Adjust (D),** which is used in BCD arithmetic to indicate the type of instruction that was executed (addition or subtraction).

**Half Carry (H),** which is used to convert the result of a previous binary addition or subtraction of BCD numbers into the correct decimal result.

Section 6.3 provides more detail on these flags.

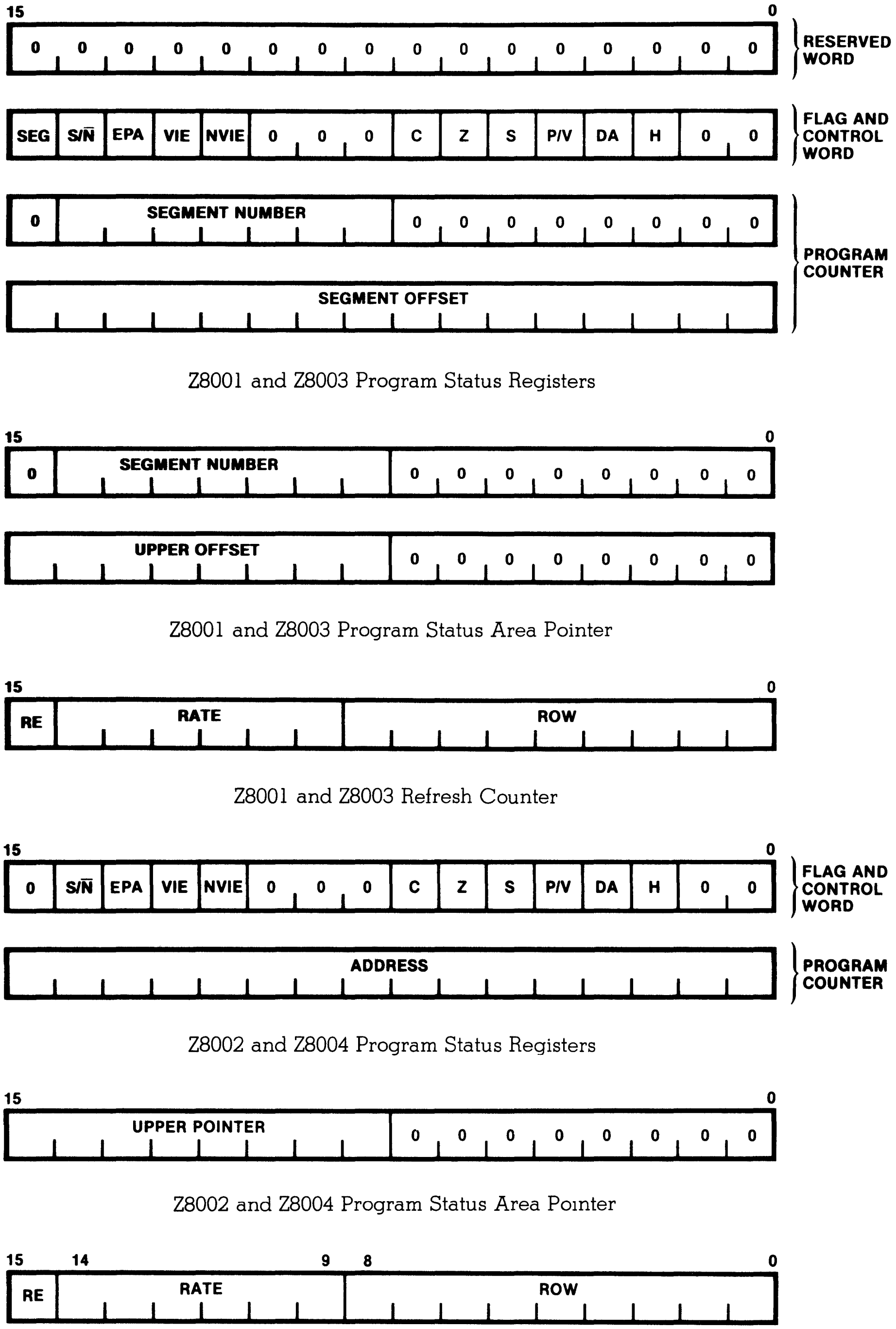
The control bits, which occupy the high-order byte of the FCW, are used to enable interrupts or to control CPU operating modes. The control bits are:



Z8001 AND Z8003

Z8002 AND Z8004

Figure 2-5. General Purpose Registers



Z8001 and Z8003 Program Status Registers

Z8001 and Z8003 Program Status Area Pointer

Z8001 and Z8003 Refresh Counter

Z8002 and Z8004 Program Status Registers

Z8002 and Z8004 Program Status Area Pointer

Z8002 and Z8004 Refresh Counter

**Figure 2-6. CPU Special Purpose Registers**

**Nonvectored Interrupt Enable (NVIE), Vectored Interrupt Enable (VIE).** These bits indicates whether or not the CPU will accept nonvectored or vectored interrupts (see Section 2.13.3).

**System/Normal Mode (S/$\overline{\text{N}}$).** When this bit is set to one, the CPU is operating in System mode; when cleared to zero, the CPU is in Normal mode (see Section 2.8). The CPU output status line (N/$\overline{\text{S}}$ pin) is the complement of this bit.

**Extended Processing Architecture (EPA) Enable.** When this bit is set to one, it indicates that the system contains Extended Processing Units, and hence extended instructions encountered in the CPU instruction stream are executed (see Section 2.12). When this bit is cleared to zero, extended instructions are trapped for software emulation.

**Segmentation Mode (SEG).** This bit is implemented only in the Z8001 and Z8003 CPUs; it is always cleared in the nonsegmented Z8002 and Z8004 CPUs. When this bit is set to one, the CPU is operating in segmented mode. When this bit is cleared to zero, the CPU is operating in nonsegmented mode (see Section 2.8).

### 2.7.2 Program Status Area Pointer (PSAP)

The Program Status Area Pointer points to an array of program status values (FCW and PC) in main memory called the Program Status Area. New FCW and PC values are fetched from this area when an interrupt or trap occurs. As shown in Figure 2-6, the PSAP consists of either one word (nonsegmented CPUs) or two words (segmented CPUs). For either configuration, the lower byte of the pointer must be zero. See Chapter 7 for details about the Program Status Area and its layout.

### 2.7.3 Refresh Register

The CPU contains a programmable counter that can be used to refresh dynamic memory automatically. The refresh register consists of a 9-bit row counter, a 6-bit rate counter and an enable bit (Figure 2-6). See Chapter 8 for details.

## 2.8 INSTRUCTION EXECUTION

**Running State.** In the usual course of events, the Z8000 CPU spends most of its time fetching instructions from memory and executing them. This process is called the running state of the CPU. The CPU also has two other states that it can enter.

**Stop/Refresh State.** This is really one state, although it can be entered either automatically for a periodic memory refresh, or when the $\overline{\text{STOP}}$ line is activated. In this state, program execution is temporarily suspended and the CPU makes use of the Refresh Register to generate refreshes. See Chapters 4 and 8 for more details.

**Bus-Disconnect State.** This is the state the CPU enters when a bus requester (such as DMA), takes over the bus. Program execution is suspended and the CPU disconnects itself from the bus. See Chapter 7 for more details.

While the CPU is in the running state, it can either be handling interrupts or executing instructions. If it is executing instructions, the Z8000 can be in the System or Normal execution mode. In System mode, privileged instructions (such as those that perform I/O) can be executed; in Normal mode they cannot. This dichotomy allows the creation of operating system software that controls CPU resources and is protected from application program access.

A CPU operates in either segmented or nonsegmented mode. In segmented mode, which is available only on the Z8001 and Z8003, the program uses 23-bit segmented addresses for memory accesses; in nonsegmented mode, which is available on all CPUs, the program uses 16-bit nonsegmented addreses for memory accesses.

While executing instructions, the mode of the CPU is controlled by bits in the FCW (Section 2.7). During the interrupt/trap response cycle, the CPU is always in System mode.

## 2.9 INSTRUCTIONS

The Z8000 instrution set contains over 400 different instructions which are formed by combining the 110 distinct instruction types (opcodes) with the various data types and addressing modes. The complete set is divided into the following groups:

**Load and Exchange** for register-to-register and register-to-memory operations, including stack management.

**Arithmetic** for arithmetic operations, including multipy and divide, on data in either registers or memory. Compare, increment, and decrement functions are included.

**Logical** for Boolean operations on data in registers or memory.

**Program Control** for program branching (conditional or unconditional), calls, and returns.

**Bit Manipulation** for setting, resetting and testing individual bits of bytes or words in registers or memory.

**Rotate and Shift** for bytes, words, or for shifts only long words within registers.

**Block Transfer and String Manipulation** for automatic memory-to-memory transfers of data blocks or strings, including compare and translate functions.

**Input/Output** for transfers of data between I/O ports and memory or registers.

**Extended** for operations involving Extended Processing Units.

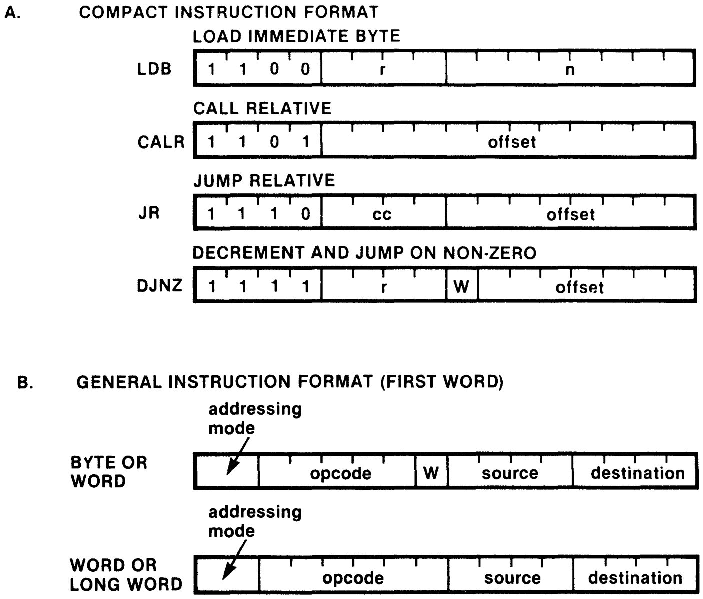
**CPU Control** for accessing special registers, controlling the CPU operating state, synchronizing multiple-processor operation, enabling/disabling interrupts, mode selection, and memory refresh.

Chapter 6 contains details on the full instruction set.

### 2.9.1 Instruction Formats

Formats of the instructions are shown in Figure 2-7. The two most significant bits (MSBs) in the instruction word determine whether the compact instruction format (Figure 2-7,A) or the general instruction format (Figure 2-7,B) is to be used.

When the two MSBs are both logic ones, the compact format is to be used. Compact formats enable the four most frequently used instructions to be encoded as single words, thereby saving on instruction-memory usage and increasing execution speed.



Note: W indicates Word (1) or Byte (0)

Figure 2-7. Instruction Formats

As when the two most significant bits are not both logic ones, the general format applies. In the general format, the two most significant bits in conjunction with the source-register field are sufficient for specifying any of the five main addressing modes. Source and destination fields are four bits wide for addressing the 16 general-purpose registers.

## 2.10 DATA TYPES

The Z8000 supports manipulation of eight data types. Five of these have fixed lengths; the other three have lengths that can vary dynamically. Each data type is supported by instructions that operate directly upon it. These data types are:

- Bit
- Signed byte-length, word-length, long word-length, or quadruple word-length binary integer
- Byte- or word-length logical value
- Word (nonsegmented or short segmented) or long word (segmented) address
- Unsigned byte-length decimal integer
- Dynamic-length string of byte data
- Dynamic-length string of word data
- Dynamic-length stack of word or long-word data

Bits can be manipulated in registers or memory. Binary and decimal integers and logical values can be manipulated in registers, although operands can be fetched directly from memory. Addresses are manipulated only in registers, but can be fetched from instruction or data memory. Strings and stacks can be manipulated only in memory.

## 2.11 ADDRESSING MODES

The information included in Z8000 instructions consists of the function to be performed, the type and size of data elements to be manipulated, and the locations of the data elements. For most two-operand instructions, one address mode is fixed, (usually Register Mode) and one of the following eight addressing modes designated as the other:

**Register Mode (R).** The data element is located in one of the 16 general-purpose registers.

**Immediate Mode (IM).** The data element is located in the instruction.

**Indirect Register Mode (IR).** The data element can be found in the location whose address is in one of the general purpose registers.

**Direct Address Mode (DA).** The data element can be found in the location whose address is in the instruction.

**Index Mode (X).** The data element can be found in the location whose address is the sum of the contents of an index value in one of the general purpose registers and an address in the instruction.

**Relative Address Mode (RA).** The data element can be found in the location whose address is the contents of the program counter offset by a displacement in the instruction.

**Base Address Mode (BA).** The data element can be found in the location whose address is the sum of a base address in one of the general purpose registers and a displacement in the instruction.

**Base Index Mode (BX).** The data element can be found in the location whose address is the sum of a base address and an index value each in separate general purpose registers.

Chapter 5 defines and illustrates the eight addressing modes.

## 2.12 EXTENDED PROCESSING ARCHITECTURE

An important feature of Z8000 CPU architecture is the Extended Processing Architecture (EPA). EPA permits the basic instruction set of the CPU to be extended via the use of external devices, called Extended Processing Unit (EPUs). A special set of instructions, called extended instructions, is used with each EPU. When the CPU encounters an extended instruction in its instruction stream, it either traps to a software trap handler to process the instruction or it performs the data transfer portion of the instruction (leaving the data manipulation part of the instruction to the EPU). Whether the CPU traps or transfers data depends on the setting of the EPA bit in the FCW.

The underlying philosophy of the EPA feature views the CPU as an instruction processor--the CPU fetches an instruction, fetches data associated with the instruction, performs the specified operations and stores the result. Extending the number of operations performed does not affect the instruction fetch and address calculation portion of the CPU acitvity. The extended instructions exploit this feature--the CPU fetches the instruction and performs any address calculation that may be needed. The CPU also generates the timing signals for the memory access if data must be transferred between memory and the EPU. But the actual data manipulation is handled by the EPU. The Extended Processing Architecture is explained more fully in Chapter 4.

## 2.13 EXCEPTIONS

Four events can alter the normal execution of a Z8000 program: a hardware interrupt which occurs when a peripheral device needs service, a synchronous software trap which occurs when an error condition arises, an instruction abort which occurs during virtual memory operations, and a system reset. Chapter 7 contains a detailed description of exceptions and how they are handled.

Interrupt requests and address translation trap requests are accepted on completion of the instruction execution cycle during which they were made. At the end of the instruction execution, a spurious instruction fetch transaction is usually performed before the interrupt acknowledge sequence but this fetch does not affect the Program Counter.

In virtual memory systems, the activation of the $\overline{\text{ABORT}}$ CPU input initiates a five-cycle abort interrupt operation. During this period, the CPU automatically saves information needed to restart the interrupted instruction execution operation. On completion of the abort interrupt operation, an interrupt service routine is initiated. This routine should locate and load the referenced information into main memory and then restart the mainstream program at its point of interruption.

### 2.13.1 Reset

A system reset overrides all other operating conditions. It puts the CPU in a known state and then causes a new program status to be fetched from a reserved area of memory to reinitialize the FCW and the PC.

### 2.13.2 Traps

Traps are synchronous events that are usually triggered by specific instructions and recur each time the instruction is executed with the same set of data and the same processor state. The four kinds of trap are:

**Extended instruction attempted in non-EPA mode.** The current instruction is an EPU instruction, but the system is not in EPA mode.

**Privileged instruction attempted in normal mode.** The current instruction is privileged (I/O for example), but the CPU is in Normal mode.

**System Call (SC) instruction.** This instruction provides a controlled access from Normal-mode to System-mode operation.

**Segmentation or addressing violation (supplied by external circuit).** This trap is intended for use by external memory managment circuitry. Only the segmented CPUs (Z8001 or Z8003) can initiate this type of trap.

### 2.13.3 Aborts

Both the Z8003 and Z8004 CPUs are provided with an $\overline{\text{ABORT}}$ input which is controlled by external memory management circuitry or devices. The detection of an asserted $\overline{\text{ABORT}}$ input by a CPU causes it to abort the currently executing instruction and to start saving status information which will be required to restart the interrupted instruction execution operation at the point of interruption. The external circuitry or device must supply five $\overline{\text{WAIT}}$ inputs to the CPU to provide time for the save operation. Status information is also presented to the external circuitry where it must also be saved for restart purposes.

### 2.13.4 Interrupts

Interrupts are asynchronous events typically triggered by peripheral devices needing attention. Three kinds of interrupt are provided, each with a separate input to the CPU. The interrupts are:

**Nonmaskable interrupts ($\overline{\text{NMI}}$).** These interrupts cannot be disabled and are usually reserved for external events that require immediate attention.

**Vectored interrupts ($\overline{\text{VI}}$).** These interrupts are maskable interrupts; eight bits of the vector output by the interrupting device are used to select the address to which the CPU will transfer to after status has been saved.

**Nonvectored interrupts ($\overline{\text{NVI}}$).** These interrupts are maskable interrupts; the vector output by the interrupting device does not affect the address to which the CPU transfers after status has been saved.

### 2.13.5 Trap and Interrupt Service Procedures

Interrupts and traps are handled similarly by the Z8000 CPUs. When the CPU begins to process an interrupt or trap, it immediately enters its system mode (segmented for Z8001 and Z8003) regardless of its mode at the time of the interrupt or trap. The CPU remains in its System mode until the new program status specified in the PSA has been established. The program status information in effect just prior to the interrupt or trap acknowledgment is pushed onto the system stack. An additional word, which serves as an identifier for the interrupt or trap, is also pushed onto the system stack, where it can be accessed by the interrupt or trap handler. The Program Status registers are loaded with new status information obtained from the Program Status Area of memory. Then control is transferred to the service procedure, whose address is now located in the Program Counter. For details of interrupt and trap handling, refer to Chapter 7.

# Chapter 3
# Address Spaces

## 3.1 INTRODUCTION

Programs and data may be located in the main memory of the computer system or in peripheral devices (that is, secondary memory). In either case, the location of the information must be specified by an address before that information can be accessed. A set of these addresses is called an address space.

The Z8000 supports two different types of address and thus two categories of address space:

- **Memory addresses,** which specify locations in main memory.
- **I/O addresses,** which specify the ports through which peripheral devices are accessed.

The CPU generates addresses during four types of operation:

- Instruction fetches, described in Chapter 4.
- Operand fetches and stores, described in Chapter 5.
- Exception processing, described in Chapter 7.
- Refreshes, described in Chapter 8.

Timing information concerning addresses is described in Chapter 9.

## 3.2 ADDRESS SPACES, SUBCATEGORIES

Within the two general types of address space (memory and I/O), it is possible to distinguish several subcategories. Figure 3-1 shows the address spaces that are available on the Z8000 CPUs.

The differences among the Z8000 CPUs lies not in the number and type of address spaces, but rather in the organization and maximum size of each space. For the Z8001 and Z8003, the addressable memory address space can be divided into 8M byte spaces. Each 8M byte address space is, in turn, divided into 128 64K byte segments. For the Z8002 and Z8004, each memory space is a homogeneous collection of 64K byte addresses. In both types of CPUs, the word I/O address spaces contain 64K port addresses and the byte I/O address space contains 64K port addresses. When an address is used to access data, the address spaces can be distingushed by the state of the status lines $ST_0$-$ST_3$ (which is determined by the way the address was generated) and by the value of the Normal/System line ($N/\overline{S}$) (which is determined by the state of the $S/\overline{N}$ bit in the FCW). The most frequently used options for specifying address spaces are:

- Instruction Space (status = 1100 or 1101), Normal mode ($N/\overline{S}$ = 1) or System mode ($N/\overline{S}$ = 0). These spaces typically address memory that contains user programs (Normal) or System programs (System).

- Data Spaces (status = 1000 or 1010), Normal mode ($N/\overline{S}$ = 1) or System mode ($N/\overline{S}$ = 0). These spaces may be used to address the data on which user or system programs operate.

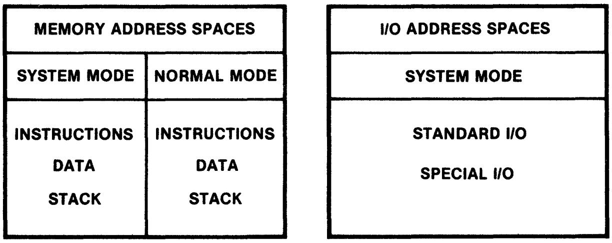


**Figure 3-1. Address Spaces on the Z8000 CPUs**

- Standard I/O Space (status = 0010). This space addresses all the I/O ports that are used for Z8000 peripherals.
- Special I/O space (status = 0011). This space addresses ports in CPU support chips (such as the Z8010 Memory Managment Unit).

## 3.3 I/O ADDRESS SPACE

All I/O addresses are represented by 16-bit words. Each of the ports addressed is either eight or 16 bits wide. Transfer to or from 16-bit ports always involves word data and, for 8-bit ports, byte data.

The address of a 16-bit port may be even or odd for both address spaces.

## 3.4 MEMORY ADDRESS SPACES

Each memory address space in the nonsegmented Z8000 CPUs, or each segment in each memory address space of the segmented Z8000 CPUs, can be viewed as addressing a string of 64K bytes numbered consecutively in ascending order. The 8-bit byte is the basic addressable element in Z8000 memory address spaces. However, there are three other addressable data elements:

- Bits, in either bytes or words
- 16-bit words
- 32-bit long words

### 3.4.1 Addressable Data Elements.

The nature of the data element being addressed depends on the instruction being executed. Different opcodes are used for addressing bytes, words, and long words; only certain instructions can address bits.

A bit can be addressed by specifying a byte or word address and the number of the bit within the byte (0-7) or word (0-15). Bits are numbered right-to-left, from the least to the most significant. This is consistent with the convention that bit n corresponds to $2^n$ in the conventional representation of positive binary numbers (see Figure 3-2).

The address of a data type longer than one byte (word or long word) is the same as the address of the byte with the lowest memory address within the word or long word (Figure 7-2). This is the leftmost, highest-order, or most significant byte of the word or long word.

Word or long word addresses are always even-numbered. Low bytes of words are stored at odd-numbered memory locations and high bytes at even-numbered locations. Byte addresses can be either even- or odd-numbered.

Only three words in memory are reserved; they are used for systems reset handling purposes (see Chapter 7).

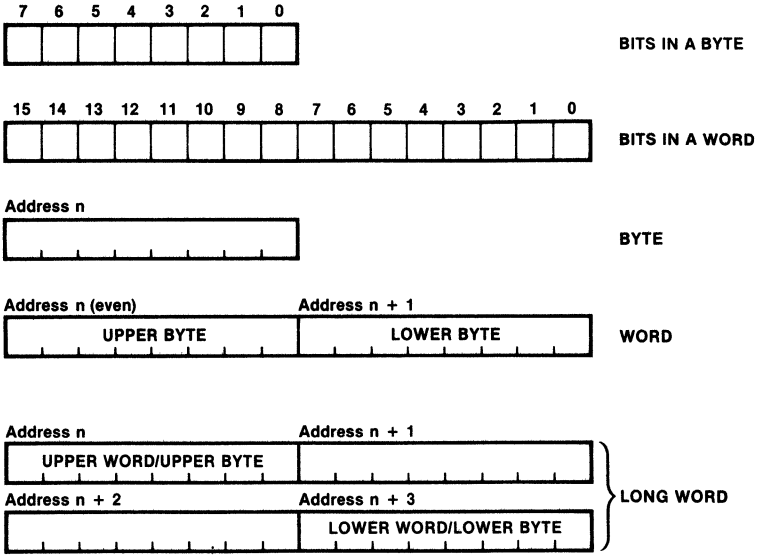


Figure 3-2. Addressable Data Elements

### 3.4.2 Segmented and Nonsegmented Addresses

The Z8002 and Z8004 generate 16-bit addresses which specify any location within a 64K byte address space. The Z8001 and Z8003 generate 23-bit segmented addresses. A segmented address consists of a 7-bit segment number, which specifies one of 128 segments, and a 16-bit offset, which specifies any one of up to 64K bytes in the specified segment. Each segment is an independent collection of bytes; thus, instructions and multiple byte data elements cannot cross segment boundaries unless explicit instructions for that purpose are included in the program. Examples of programs and data that cross segment boundaries are presented in Chapter 10. Some of the advantages of address segmentation are outlined in Section 3.4.3.

Figure 3-3 shows the formats of segmented and nonsegmented addresses. Nonsegmented addresses are 16 bits long and thus can be stored in word registers (Rn), or in memory as word-length addressable elements. The 23-bit segmented addresses are embedded in 32-bit long words and thus can be stored in register pairs (RRn) or long word memory elements.

When a segmented CPU (Z8001 or Z8003) is operating in the nonsegmented mode (Chapter 4), it still generates segmented addresses. The segment number portion of these addresses is supplied by the Segment Number portion of the Program Counter which remains unchanged during the nonsegmented mode of operation.

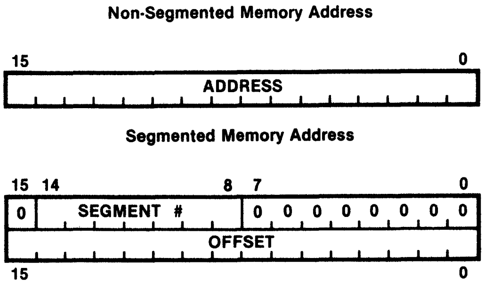


**Figure 3-3. Segmented and Nonsegmented Address Formats**

### 3.4.3 Segmentation and Paging Memory Management

Addresses manipulated by the programmer, used by instructions, and output by a segmented Z8000 CPU are called "logical addresses." An external memory-management circuit can translate logical addresses into physical (actual) memory addresses and perform certain checks to ensure that data and programs are properly accessed.

The Z8010 MMU performs a logical-to-physical address translation function for the segmented addresses produced by the Z8001 and Z8003 CPUs. A single MMU holds 64 segment descriptors. Each descriptor tells where the segment lies in physical memory, how long the segment is, and what kind of accesses can be made to the segment. The MMU uses these descriptors to translate logical segment numbers and offsets into 24-bit physical addresses (as shown in Figure 3-4). During translation, the MMU checks for errors such as an attempt to write into a read-only segment or a system segment being accessed by a nonsystem program. Z8010 MMUs are designed to be combined so that more than 64 descriptors can be supported in a system at any time.

The Z8015 Paged Memory Management Unit (PMMU) is designed to support a segmented paged virtual memory for the Z8003 CPU. The PMMU, however, can also be used to support the other Z8000 CPUs. Each PMMU can manage a memory area of 64 fixed-sized pages, with each page 2048 bytes in length. Other page sizes can be implemented and PMMUs can be combined to support more pages. The PMMUs translate the logical addresses output by the CPU into physical addresses. Each PMMU can manage a physical memory address area of 64 pages (128K bytes); PMMUs can be combined in groups of 8 to address any size of virtual memory. Each PMMU contains a table of 64 page descriptors. The information contained by a page descriptor enables the PMMU to determine the following:

1) whether or not the page containing the referenced, location is in main memory
2) the types of accesses permitted
3) whether or not the page had been previously addressed by the executing program
4) whether or not the original contents of the page had been changed
5) whether or not, the referenced area is to be used for stack operations; if it is, the PMMU should issue a write warning if fewer then 256 bytes remain in the page

If the referenced location (i.e. page) is not in main memory, the PMMU translating the logical address sends an instruction abort to the CPU. The CPU then aborts the current instruction and

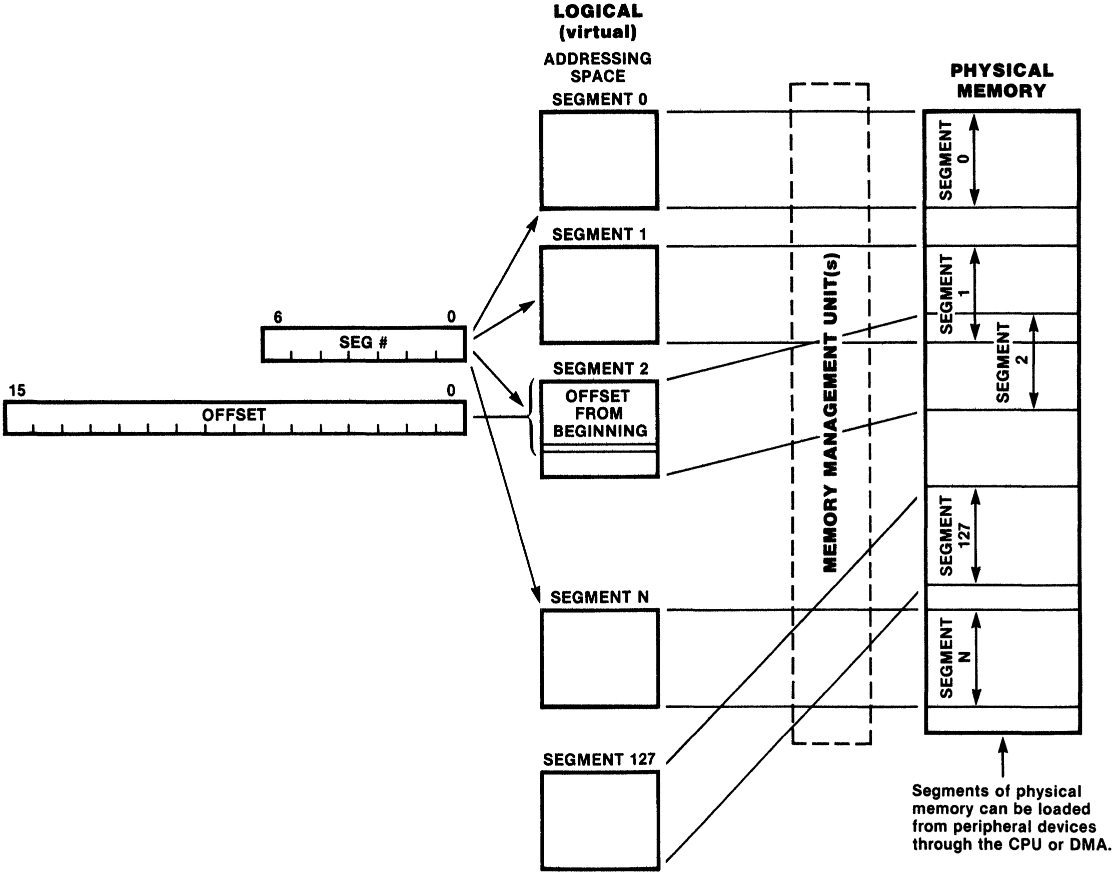


Figure 3-4. Segmented Address Translation

saves program restart information. The PMMU also sends a trap request to the CPU to initiate the execution of a service routine which locates and loads the desired page into main memory, and then restarts the mainstream program at the point of its abort interruption.

The Z8000 CPU does not require the use of Z8010 or Z8015 MMUs; the segment number can be used directly as part of a physical address. In this type of application, memory is regarded as being composed of 128, 64K byte banks with no need for explicit bank switching.

Some of the benefits of the memory management features provided by an MMU are:

- Provision for flexible and efficient allocation of physical memory resources during the execution of programs

- Support for multiple, independently executing programs that can share access to common code and data

- Protection from unauthorized or unintentional access to data or programs

- Detection of obviously incorrect use of memory by an executing program

- Separation of user code from system code

Segmentation in the Z8001 and Z8003 helps support memory management in two ways:

- By allowing part of an address (the segment number) to be output by the CPU early in a memory transaction. This allows access to the address descriptor in the MMU without adding to the basic access time of the memory

- By providing a standard, variable-sized unit of memory for the protection, sharing, and movement of data

In addition, segmentation is a natural model for the support of modular programs and data in a multi-programming environment. It efficiently supports re-entrant programs by providing data relocation for different tasks using common code.

More information about the Z8010 MMU and memory management can be found in An Introduction to the Z8010 MMU Memory Management Unit (contained in Zilog's Data Book, document #00-2034-02), and the Z8010 MMU Technical Manual (document #00-2015-01). Information about the Z8015 PMMU can be found in the Z8015 PMMU Technical Manual (document #03-8223-01).

# Chapter 4
# CPU Operation

## 4.1 INTRODUCTION

This chapter gives a fundamental description of the operating states of the Z8000 CPU and the process of instruction execution. The details of instruction execution are described in Chapters 5 and 6. Other detailed aspects of Z8000 operation are given in Chapter 7 (Exceptions) and Chapter 8 (Refresh). Chapter 9 describes CPU operations as they are manifested by the external pins of the CPU.

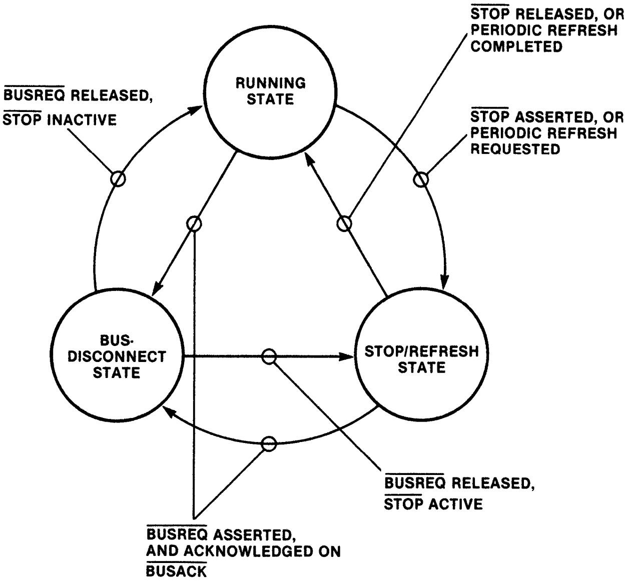
## 4.2 OPERATING 'STATES

The Z8000 CPU has three operating states: Running state, Stop/Refresh state, and Bus-Disconnect state. Running state is the usual state of the processor: the CPU is executing instructions or handling exceptions. Stop/Refresh state is entered when the $\overline{\text{STOP}}$ line is asserted or the refresh counter indicates that a periodic refresh should be performed. In this state, memory refresh transactions, if enabled, are generated continually (see Chapter 8). Bus-Disconnect state is entered when the CPU acknowledges a bus request and gives up control of the system bus. Figure 4-1 shows the three states and the conditions that cause state transitions.

### 4.2.1 Running State

While the CPU is in Running state, it is either executing instructions (as described in Section 4.3) or handling exceptions (as described in Chapter 7). The CPU is normally in Running state, but will leave this state in response to one of three conditions:

- The refresh mechanism indicates that a periodic refesh needs to be performed, in which case the CPU temporarily enters Stop/Refresh state.



$\overline{\text{STOP}}$ RELEASED, OR PERIODIC REFRESH COMPLETED
RUNNING STATE
$\overline{\text{BUSREQ}}$ RELEASED, $\overline{\text{STOP}}$ INACTIVE
$\overline{\text{STOP}}$ ASSERTED, OR PERIODIC REFRESH REQUESTED
BUS-DISCONNECT STATE
STOP/REFRESH STATE
$\overline{\text{BUSREQ}}$ RELEASED, $\overline{\text{STOP}}$ ACTIVE
$\overline{\text{BUSREQ}}$ ASSERTED, AND ACKNOWLEDGED ON $\overline{\text{BUSACK}}$

**Figure 4-1. Operating States and Transitions**

- An external bus request pushes the CPU into Bus-Disconnect state.

- An external stop request pushes the CPU into Stopped state.

### 4.2.2 Stop/Refresh State

While the CPU is in Stop/Refresh state, it generates a continuous stream of refresh cycles (as discussed in Chapter 8) and does not perform any other functions. This state provides for the generation of memory refreshes by the CPU and allows external devices to suspend CPU operation. This feature can be used to force single-step operation of the processor or to synchronize the CPU with an Extended Processing Unit (as described in Section 4.4).

The CPU enters Stop/Refresh state when the refresh mechanism needs to do a refresh or when the STOP line is activated. It leaves Stop/Refresh state when neither of these conditions holds or when a bus request causes the CPU to enter Bus-Disconnect state.

### 4.2.3 Bus-Disconnect State

A CPU enters a Bus-Disconnect state from either a Running state or a Stop/Refresh state when a bus request has been received on BUSREQ and is acknowledged on BUSACK (as described in Chapter 9). While in this state, it disconnects itself from the bus by 3-stating its output. It will leave Bus-Disconnect state when the external bus request has been received. The Bus-Disconnect state is highest in priority in that the presence of a bus request will force the CPU into this state, regardless of any other conditions indicating that a different state should be entered.

### 4.2.4 Effect of Reset

Activation of the CPU's RESET line puts the CPU into a nonoperational state within five clock cycles, regardless of its previous state or the states of its other inputs. The CPU will remain in this state until RESET is deactivated. When RESET is deactivated, the processor enters the running state for at least one machine cycle. Reset is more fully described in Chapters 7 and 9.

## 4.3 INSTRUCTION EXECUTION

While the CPU is in Running state and executing instructions, it is controlled by the Program Status registers (Figure 4-2). The Program Counter gives the address from which instructions are fetched, and the flags control branching (as described in Chapter 6). The control bits determine the CPU operating states (see section 4-2) and interrupt masking.

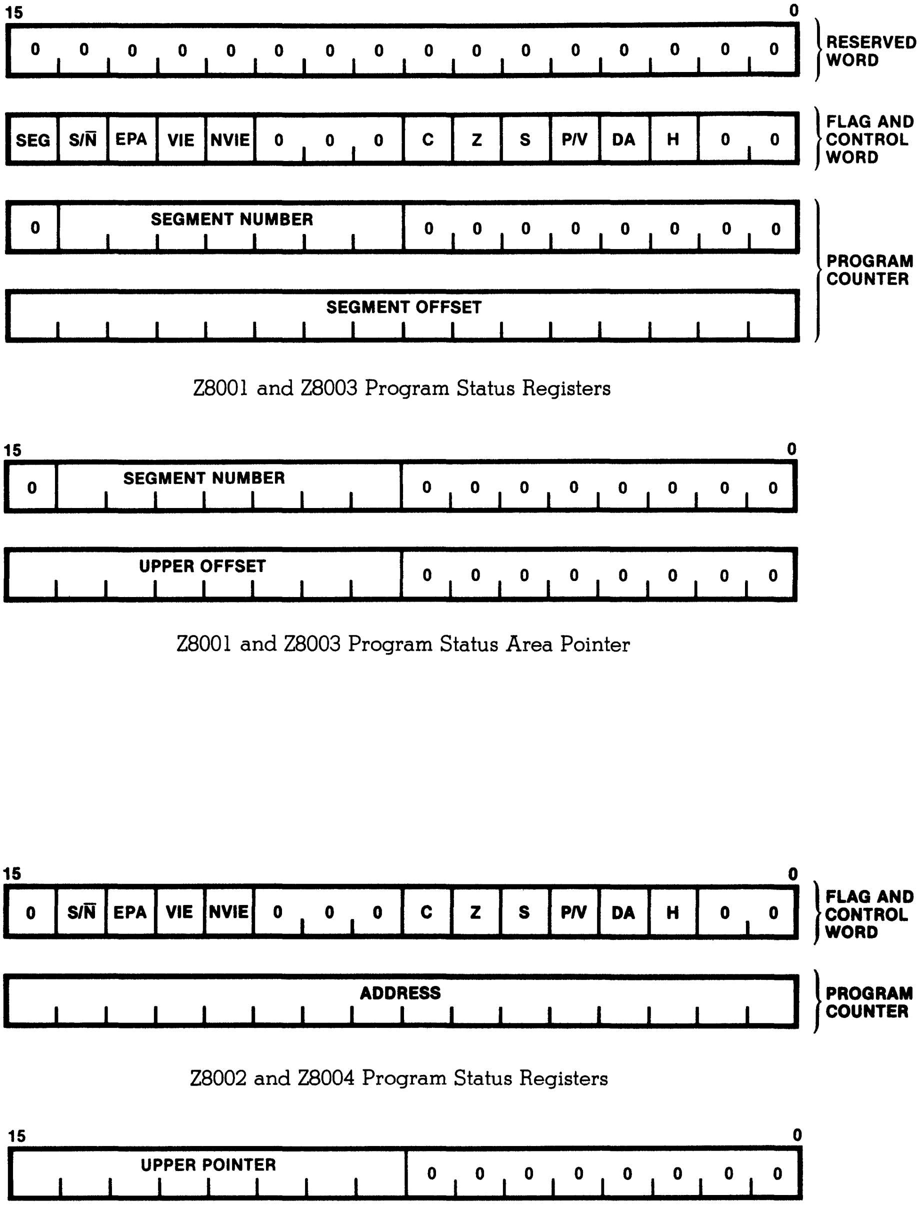
Instruction execution consists of the repeated application of two steps:

- Fetch one or more words comprising a single instruction from the program memory address space at the address specified by the Program Counter (PC).

- Perform the operation specified by the instruction and update the Program Counter and flags in the Program Status registers.

The operation performed by an instruction and the way the flags are updated depends on the particular instruction being executed. The instruction set is described in Chapter 6. For most instructions, the PC value is updated to point to the word immediately following the last word of the instruction. The effect of this is that instructions are fetched sequentially from memory. Exceptions to this are the Branch, Call, Interrupt Return, Load Program Status, System Call, Halt, Decrement And Jump If Non-Zero, and Return instructions, which cause the PC to be set to a value generated by the instruction. This causes a transfer of control with execution continuing at the new address in PC. The exact operation of these instructions is described in Chapter 6.

The Z8000 CPU is able to overlap the fetching of one instruction with the operations of the previous instruction. This facility, called Instruction Look-Ahead, is illustrated in Figure 4-3. This shows the execution of a series of memory-to-register instructions, such as a value in memory being added to the value in a general-purpose register. Part of each instruction is fetched while the previous instruction execution is being completed. This mechanism provides faster execution speed than fetching each instruction only after the prior instruction has completed execution.



Z8001 and Z8003 Program Status Registers

Z8001 and Z8003 Program Status Area Pointer

Z8002 and Z8004 Program Status Registers

Z8002 and Z8004 Program Status Area Pointer

**Figure 4-2. Program Status Registers**

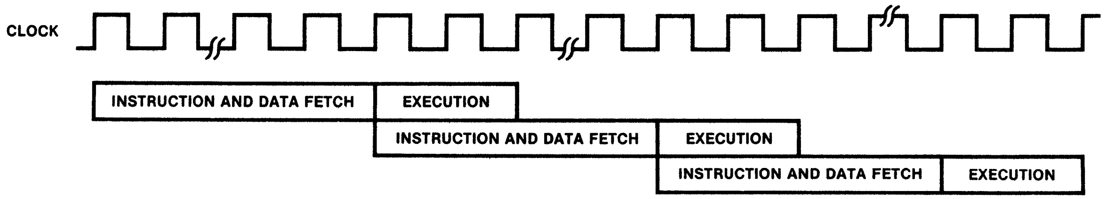

CLOCK
INSTRUCTION AND DATA FETCH
EXECUTION
INSTRUCTION AND DATA FETCH
EXECUTION
INSTRUCTION AND DATA FETCH
EXECUTION

**Figure 4-3. Instruction Look-Ahead**

After executing an instruction and in some cases (explained in Chapters 6 and 7) during an instruction's execution, the CPU checks to see if there are any traps or interrupts pending and not masked. If so, it temporarily suspends instruction execution and begins a standard exception-handling sequence. This sequence, which is described fully in Chapter 7, causes the value of the Program Status registers to be saved and a new value loaded. Instruction execution then continues with a new PC value and Flag and Control Word value. The effect is to switch the execution of the CPU from one program to another.

### 4.3.1 Running-State Mode

While the CPU is executing instructions, its mode is controlled by two control bits in the FCW: the System/Normal bit ($S/\overline{N}$) and the Segmentation Mode bit (SEG).

### 4.3.2 Segmented and Nonsegmented Modes

The segmentation mode of the CPU (segmented or nonsegmented) determines the size and format of addresses that are manipulated by programs. In segmented mode (SEG = 1), programs use 23-bit segmented addresses; in nonsegmented mode (SEG = 0), programs use 16-bit nonsegmented addresses. There are also the following differences in the address portions of instructions, which are due to the difference in address size:

- Indirect and Base Registers are 32-bit registers in segmented mode and 16-bit registers in nonsegmented mode.

- Addresses embedded in instructions are always 16-bit in nonsegmented mode. In segmented mode addresses consist of either one 16-bit word (7-bit segment number and an 8-bit offset) or two 16-bit words (7-bit segment number and a 16-bit offset).

Both the segmented and the nonsegmented modes are available on the Z8001 and Z8003. Only the nonsegmented mode is available on the Z8002 and Z8004. Since both addressing modes are supported on the Z8001 and Z8003, these CPUs can execute, without alteration, programs prepared for the Z8002 and Z8004. The reverse is not possible.

The Z8001 and Z8003 CPUs always generate segmented addresses, even when operating in nonsegmented mode. When a memory access is made in nonsegmented mode, the offset of the segmented address is the 16-bit address generated by the program, and the segment number is the value of the segment number field of the Program Counter.

### 4.3.3 Normal and System Modes

The operating mode of the CPU (System mode or Normal mode) determines which instructions can be executed and which Stack Pointer register is used.

In System mode ($S/\overline{N}$ = 1), all instructions can be executed. While in Normal mode, privileged instructions (such as I/O operations or changes to control registers) cannot be executed.

The second distinction between System and Normal mode is access to the system or normal Stack Pointer. As shown in Figure 4.4, there are two Stack Pointer registers (R14 and R15 in the Z8002 and Z8004, and RR14 in the Z8001 and Z8003): one for Normal mode and one for System mode. When in Normal mode, a reference by an instruction to the Stack Pointer register will access the Normal mode Stack Pointer. When in System mode, an access to the Stack Pointer register will normally reference the System mode Stack Pointer. If, however, the

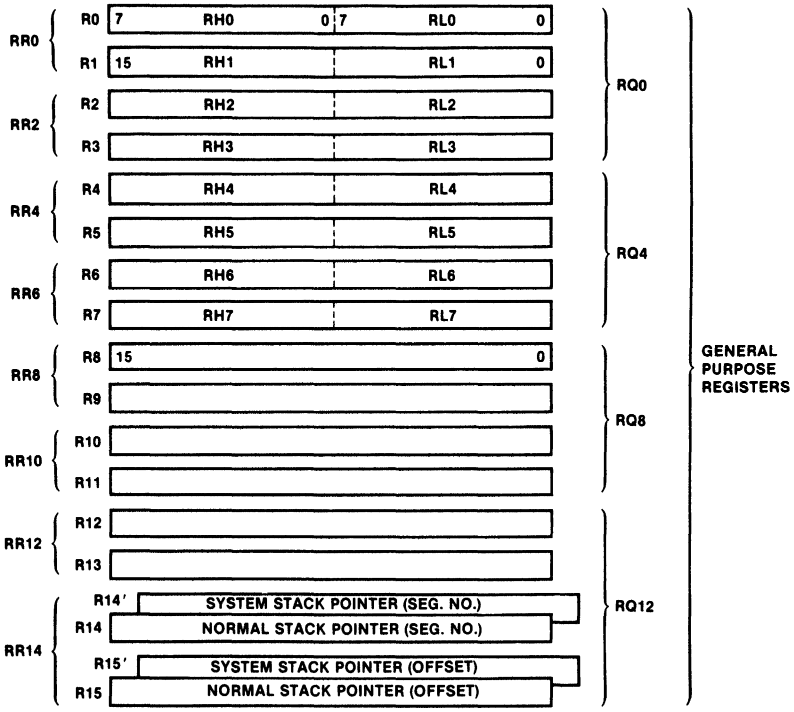

RR0
R0
7
RH0
0
7
RL0
0
R1
15
RH1
RL1
0
RQ0
RR2
R2
RH2
RL2
R3
RH3
RL3
RR4
R4
RH4
RL4
R5
RH5
RL5
RQ4
RR6
R6
RH6
RL6
R7
RH7
RL7
RR8
R8
15
0
R9
RQ8
GENERAL PURPOSE REGISTERS
RR10
R10
R11
RR12
R12
R13
RQ12
RR14
R14'
SYSTEM STACK POINTER (SEG. NO.)
R14
NORMAL STACK POINTER (SEG. NO.)
R15'
SYSTEM STACK POINTER (OFFSET)
R15
NORMAL STACK POINTER (OFFSET)

**Figure 4-4. General Purpose Registers**

CPU is either a Z8001 or Z8003 which is operating in a nonsegmented system mode, a reference to R14 will access the Normal mode Stack Pointer register R14 (see Table 4.1).

In Normal mode, the system Stack Pointer is not accessible; in System mode, the normal Stack Pointer is accessed by using a special Load Control Register instruction (described in Chapter 6).

The CPU switches modes whenever the Program Status Control bits change. This can happen when a privileged Load Control Register instruction is

**Table 4-1. Registers Accessed by Reference to R14 and R15**

| Register Referenced by Instruction | System Mode | | Normal Mode | |
|---|---|---|---|---|
| | Segmented | Nonsegmented | Segmented | Nonsegmented |
| R14 | System R14 | Normal R14 | Normal R14 | Normal R14 |
| R15 | System R15 | System R15 | Normal R15 | Normal R15 |
| RR14 | System R14<br>System R15 | Normal R14<br>System R15 | Normal R14<br>Normal R15 | Normal R14<br>Normal R15 |

NOTE: Z8002 and Z8004 always run in nonsegmented mode.

executed, or when an exception (interrupt, trap, or reset) occurs. The System Call instruction is used to generate a special trap that provides a controlled transition from Normal mode to System mode.

The distinction between Normal and System modes permits the operating system to run in System mode and control the system's resources, including the management of one or more application programs which run in Normal mode. Separate Normal and System modes, and memory protection, provide the basis for protecting an operating system from the malfunctions of application programs.

## 4.4 EXTENDED INSTRUCTIONS

The Z8000 CPU supports seven types of extended instructions, which can be executed cooperatively by the CPU and an external Extended Processing Unit (EPU). The execution of these instructions is controlled by the EPA control bit in the FCW.

The EPA bit specifies whether or not an EPU is in the system. When the EPA bit is zero, no EPU is in the system. The CPU will then trap when it encounters an extended instruction (as explained in Chapter 7). This allows the operation of the extended instruction to be simulated by software running on the CPU.

If the EPA bit is set, an EPU is in the system. The CPU will fetch the extended instruction and perform any address calculation required by that instruction. If the instruction specifies the transfer of data, the CPU will generate the timing signals for this transfer and will carry out its portion of the transfer. The CPU will then fetch and begin executing the next instruction in its instruction stream. The EPU is expected to monitor the CPU's activity, participate in extended instruction data transfers initiated by the CPU, and execute extended instructions. While the EPU is executing an instruction, the CPU can be fetching and executing further instructions. If the CPU fetches another extended instruction before the EPU is finished executing an instruction, the $\overline{\text{STOP}}$ line is used by the EPU to delay the CPU until the previous instruction is complete. This process is described more fully in Chapters 6 and 9.

# Chapter 5
# Addressing Modes

## 5.1 INTRODUCTION

This chapter describes the eight addressing modes used by instructions to access data in memory or CPU registers. Examples for the nonsegmented and segmented modes of operation are given at the end of the chapter.

An instruction is a consecutive list of one or more words aligned at even-numbered byte addresses in memory. Most instructions have operands in addition to an operation code (opcode). These operands can reside in CPU registers or memory locations. The modes by which references are made to operands are called "addressing modes." Figure 5-1 illustrates these modes. Not all instructions can use all addressing modes; some instructions can use only a few, and some instructions use none at all. In Figure 5.1, the term "operand" refers to the data to be operated upon.

## 5.2 USE OF CPU REGISTERS

The 16 general-purpose CPU registers can, with the exceptions noted below, be used in any of the following ways:

- As accumulators, where the data to be manipulated resides in the registers.

- As pointers, where the value in the register is the memory address of the operand, rather than the operand itself. In string and stack instructions, the pointers can be automatically stepped either forward or backward through memory locations.

- As index or base register, where the contents of the register and the word(s) following the instruction are combined to produce the address of the operand.

There are two exceptions to the above uses of general-purpose registers:

- Register R0 (or the double register RR0 in segmented mode) cannot be used as an indirect register, base register, index register, or software stack pointer.

- The System Mode stack register (R15 in the Z8002 or Z8004 or the double register RR14 in the Z8001 or Z8003) is used in acknowledging interrupts and therefore it cannot, in general, be used as an accumulator in System-mode operation.

In addition to the general-purpose use of Z8000 registers, the following registers are used for special purposes:

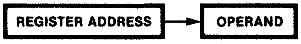
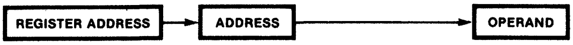
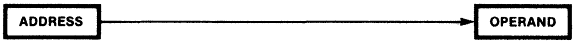
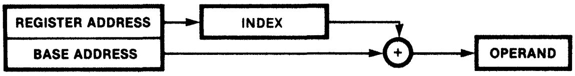
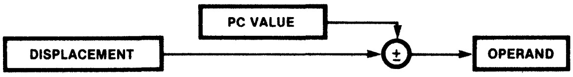
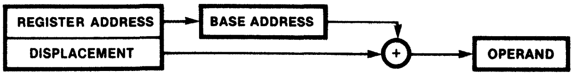
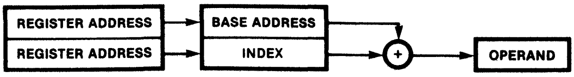
- Register R15 in nonsegmented operation or the double register RR14 in segmented operation is used as a stack pointer for subroutine calls and returns.

- The byte register RH1 is used in the translate instructions (TRDB, TRDRB, TRIB, TRIRB) and the translate and test instructions (TRTDB, TRTDRB, TRTIB, TRTIRB).

- Register R0 is used in extended instructions.

In the Relative Address (RA) mode, the program counter (PC) is used instead of a general-purpose CPU register to supply the base address for an effective address calculation.

The PC is normally used to keep track of the next instruction to be executed; whenever an instruction is fetched from memory, the PC is immediately incremented to point to the next instruction. This updated PC value is used in relative addressing as the base address for the effective address calculation. Operands specified by relative addressing reside in the program address space. That is, memory access bus transactions resulting from relative addressing operations are accompanied by status output

| Addressing Mode | Operand Addressing | | | Operand Value |
|---|---|---|---|---|
| | In the Instruction | In a Register | In Memory | |
| **R**<br>Register |  | | | The content of the register |
| **IM**<br>Immediate | | | | In the instruction |
| ***IR**<br>Indirect Register |  | | | The content of the location whose address is in the register |
| **DA**<br>Direct Address |  | | | The content of the location whose address is in the instruction |
| ***X**<br>Index |  | | | The content of the location whose address is the address in the instruction plus the content of the working register. |
| **RA**<br>Relative Address |  | | | The content of the location whose address is the content of the program counter, offset by the displacement in the instruction |
| ***BA**<br>Base Address |  REGISTER ADDRESS<br>DISPLACEMENT<br>BASE ADDRESS<br>+<br>OPERAND | | | The content of the location whose address is the address in the register, offset by the displacement in the instruction |
| ***BX**<br>Base Index |  REGISTER ADDRESS<br>REGISTER ADDRESS<br>BASE ADDRESS<br>INDEX<br>+<br>OPERAND | | | The content of the location whose address is the address in a register plus the index value in another register. |

*Do not use R0 or RR0 as indirect, index, or base registers.

**Figure 5-1. Addressing Modes**

($ST_3$-$ST_0$) 1100 which encodes "instruction space access".

## 5.3 SHORT ENCODING OF SEGMENTED ADDRESSES IN INSTRUCTIONS

Two of the addressing modes, Direct and Index, require a memory address as part of the instruction. Segmented addresses generated by the Z8001 and Z8003 are 23 bits long. Within an instruction, a segmented address is represented in either two words (16-bit long offset) or one word (8-bit short offset).

As Figure 5-2 illustrates, bit 7 of the segment number byte distinguishes between the two formats. When this bit is set, the long-offset representation is implied. When the bit is cleared, the short-offset address representation is implied. For a short-offset address, the 23-bit segmented address is reduced to 16 bits by omitting the eight most significant bits of the offset, which are assumed to be zero.

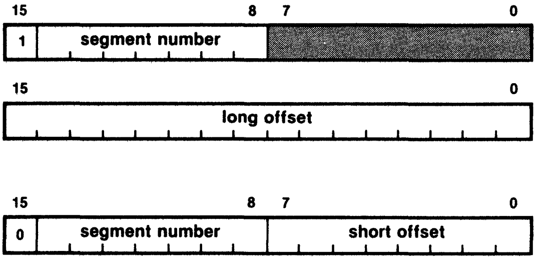


**Figure 5-2. Segmented Memory Address Within Instruction**

NOTE: Shaded area is reserved.

## 5.4 ADDRESSING MODE DESCRIPTIONS

The following pages contain descriptions of the addressing modes of the Z8000 CPUs. Each description:

- Explains how the operand address is calculated
- Indicates in which address space (Register, I/O Special I/O, Data memory, or Program memory) the operand is located
- Shows the assembly language format used to specify the addressing mode
- Works through an example

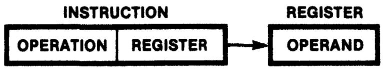
The descriptions are grouped into two sections--one for nonsegmented programs, the other for segmented programs.

## 5.5 DESCRIPTIONS AND EXAMPLES

The nonsegmented mode is described in this section. The information presented applies to the Z8002, Z8004, and to the Z8001 and Z8003 when operated in a nonsegmented mode.

### 5.5.1 Register (R)

In the Register Addressing mode the instruction operand is located in a specified general-purpose CPU register. Storing data in a register allows shorter instructions and faster execution than accessing memory.



THE OPERAND VALUE IS THE CONTENTS OF THE REGISTER.

The operand is always in a general-purpose CPU register. The register length (byte, word, register pair, or register quadruple) is implied by the instruction opcode.

**Assembler language format:**

| | |
|---|---|
| RHn,RLn | Byte register |
| Rn | Word register |
| RRn | Double-word register |
| RQn | Quadruple-word register |

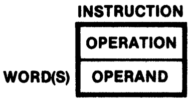
**Example of R mode:**

```
LD R2,R3     !Load the contents of
              R3 into R2!
```

| | *Before Execution* | | *After Execution* |
|---|---|---|---|
| R2 | A6B8 | R2 | 9A20 |
| R3 | 9A20 | R3 | 9A20 |

### 5.5.2 Immediate (IM)

The Immediate addressing mode does not indicate a register or memory address as the source operand. The data processed by the instruction is in the instruction. IM addressing can only be used to address source operands, never destination operands.



THE OPERAND VALUE IS IN THE INSTRUCTION.

Because an immediate operand is part of the instruction, it is always located in the program memory address space. Immediate mode is often used to initialize registers. The Z8000 is optimized for this function, providing several short immediate instructions to reduce the length of programs.

**Assembler language format (see also Chapter 6):**

#data

**Example of IM mode:**

```
LDB RH2,#%55        !Load hex 55 into RH2!
```

*Before Execution*

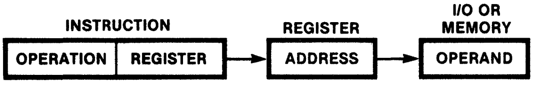
| | |
|---|---|
| R2 | 6789 |

*After Execution*

| | |
|---|---|
| R2 | 5589 |

### 5.5.3 Indirect Register (IR)

In the Indirect Register Addressing mode, the register holds the address of the operand.



THE OPERAND VALUE IS THE CONTENTS OF THE LOCATION WHOSE ADDRESS IS IN THE REGISTER.

A single word register is used to hold the address. Any general-purpose word register, except R0, can be used.

Depending on the instruction, the operand specified by IR mode will be located in Standard I/O address space (Standard I/O instructions), Special I/O address space (Special I/O instructions), or data memory address space, or stack memory address space. For non-I/O references, the status lines will indicate stack reference if the stack pointer (R15) is used as the indirect register; otherwise, the status lines will indicate data memory reference.

The Indirect Register mode may save space and reduce execution time when consecutive locations are referenced. This mode can also be used to simulate more complex addressing modes, since addresses can be computed before the data is accessed.

**Assembler language format (see also Chapter 6):**
@Rn

**Example of IR mode:**

```
LD R2,@R5          !Load R2 with the
                    data addressed by the
                    contents of R5!
```

*Before Execution*

| | |
|---|---|
| R2 | 030F |
| R3 | 0005 |
| R4 | 2000 |
| R5 | 170C |

*Memory*

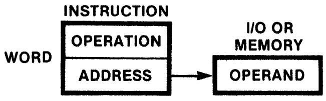
| | |
|---|---|
| | ⋮ |
| 170A | A023 |
| 170C | 0B0E |
| 170E | 10D0 |
| | ⋮ |

*After Execution*

| | |
|---|---|
| R2 | 0B0E |
| R3 | 0005 |
| R4 | 2000 |
| R5 | 170C |

### 5.5.4 Direct (DA)

In the Direct addressing mode, the address of the operand is in the instruction.



THE OPERAND VALUE IS IN THE CONTENTS OF THE LOCATION WHOSE ADDRESS IS IN THE REGISTER.

Depending upon the instruction, the operand specified by DA mode will be in Standard I/O space (Standard I/O instructions), in Special I/O space (Special I/O instructions), or in data memory space.

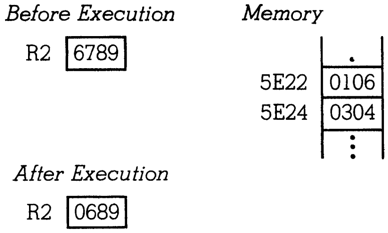
This mode is also used by Jump and Call instructions to specify the address of the next instruction to be executed in program memory. (Actually, the address serves as an immediate value that is loaded into the Program Counter.)

**Assembler language format (see also Chapter 6):**

address   either memory, I/O, or Special I/O

**Example of DA mode:**

```
LDB RH2,%5E23  !load RH2 with the
                data in location
                5E23!
```

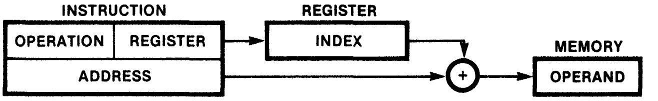


### 5.5.5 Index (X)

In the Index addressing mode, the operand address is computed by adding the address specified in the instruction to the contents of a word register, (called "the index register") which is also specified by the instruction. Indexed addressing allows random access to tables or other data structures when the address of the base of the table is known, but the index for a particular element must be computed by the program.

Any word register except R0 can be used as the index register.

Operands specified by X mode are always in data memory address space, except when Index Addressing is used with the Jump and Call instructions. In these cases, the destination, computed by adding the index register contents to the base address, is in program memory space.



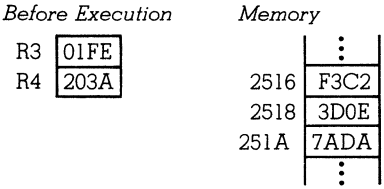
THE OPERAND VALUE IS THE CONTENTS OF THE LOCATION WHOSE ADDRESS IS THE ADDRESS IN THE INSTRUCTION, OFFSET BY THE CONTENTS OF THE REGISTER.

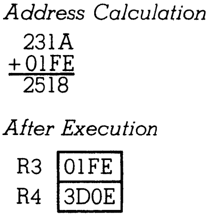
**Assembler language format (see also Chapter 6):**

address(Rn)

**Example of X mode:**

```
LD R4,%231A(R3)  !load into R4 the con-
                  tents of the memory
                  location whose
                  address is 231A +
                  the value in R3!
```



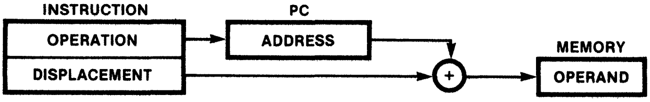


### 5.5.6 Relative Address (RA)

In the Relative Address mode the operand is found at an address relative to the address of the current instruction. The instruction specifies a two's complement displacement which is added to the 16-bit address in the Program Counter to form the target address. The Program Counter setting used is the address of the first instruction following the currently executing instruction.

An operand specified by RA mode is always in the program memory address space.

Relative addressing is used with the Jr, DJNZ, CALR, LDR and LDAR instructions.



THE OPERAND VALUE IS THE CONTENTS OF THE LOCATION WHOSE ADDRESS IS THE CONTENTS OF PC OFFSET BY THE DISPLACEMENT IN THE INSTRUCTION.

**Assembler language format (see also Chapter 6):**

address

**Example of RA mode:**

(Note that the symbol "$" is used for the value of the current program counter.)

```
LDR R2,$+%6        !Load into R2 the con-
                    tents of the memory
                    location whose
                    address is the address
                    of the given instruction
                    + hex 6!
```

Because the program counter is advanced to point to the next instruction before the address calculation is performed, the constant that occurs in the instruction is +2.

*Before Execution*

| | |
|---|---|
| R2 | A0F0 |
| PC | 0202 |

*Program Memory*

| | | |
|---|---|---|
| | ⋮ | |
| 0202 | 3102 | Instruction |
| 0204 | 0002 | |
| 0206 | E801 | |
| 0208 | FFFE | |
| | ⋮ | |

*Address Calculation*

```
 0206
+   2
 0208
```

*After Execution*

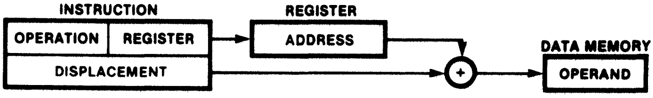
| | |
|---|---|
| R2 | FFFE |
| PC | 0206 |

### 5.5.7 Base Address (BA)

The Base Address mode is similar to the Index mode in that a base and an offset are combined to produce the effective address. In Base Addressing, however, a register contains the base address, and the displacement is expressed as a 16-bit value in the instruction. The two are added and the result is the address of the operand. This addressing mode can only be used with the Load and Load Address instructions. The Base Address mode allows random access to tables or other data structures for which the displacement of an element within the structure is known, but the base address of the particular structure must be computed by the program.

Any word register except R0 can be used for the base address

The status lines will indicate a stack reference if the base register is the stack pointer (R15) and will indicate data reference otherwise.



THE OPERAND VALUE IS THE CONTENTS OF THE LOCATION WHOSE ADDRESS IS THE ADDRESS IN THE INSTRUCTION, OFFSET BY THE CONTENTS OF THE REGISTER.

**Assembler language format (see also Chapter 6).**

Rn(#disp)

**Example of BA mode:**

```
LDL R5(#%18),RR2  !load the long word
                   in RR2 into the
                   memory location
                   whose address is
                   the value in R5 +
                   hex 18!
```

*Before Execution*

| | | |
|---|---|---|
| RR2 | R2 | 0A00 |
| | R3 | 1500 |
| | R4 | 3100 |
| | R5 | 20AA |

*Memory*

| | |
|---|---|
| | ⋮ |
| 20C0 | 0ABE |
| 20C2 | F50D |
| 20C4 | BADE |
| 20C6 | B0D1 |
| | ⋮ |

*Address Calculation*

$$\begin{array}{r} \text{20AA} \\ +\ \ \text{18} \\ \hline \text{20C2} \end{array}$$

*After Execution*

| | | |
|---|---|---|
| RR2 | R2 | 0A00 |
| | R3 | 1500 |
| | R4 | 3100 |
| | R5 | 20AA |

*Memory*

| | |
|---|---|
| | ⋮ |
| 20C0 | 0ABE |
| 20C2 | 0A00 |
| 20C4 | 1500 |
| 20C6 | B0D1 |
| | ⋮ |

### 5.5.8 Base Index (BX)

The Base Index addressing mode is an extension of the Base Addressing mode and can be used only with the Load and Load Address instructions. In this case, both the base address and the index (displacement) are held in registers. This mode allows access to memory locations whose addresses are computed at runtime and are not fully known at assembly time.

Any word register except R0 can be used for either the base address or the index.

The status lines will indicate a stack access if the base register is the stack pointer (R15), otherwise, they will indicate data access.

**Assembler language format (see also Chapter 6)**

Rn(Rm)

**Example of BX mode:**

```
LD R2,R5(R3)  !load into R2 the
               value whose address
               is the value in base
               register R5 + the
               value in index
               register R3!
```

*Before Execution*

| | |
|---|---|
| R2 | 1F3A |
| R3 | FFFE |
| R4 | 0300 |
| R5 | 1502 |

*Data Memory*

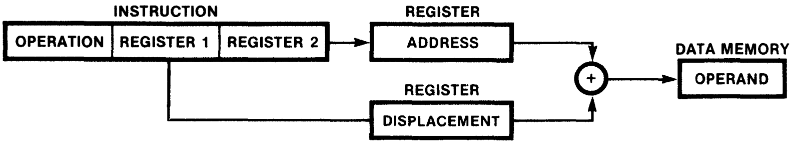
| | |
|---|---|
| | ⋮ |
| 14FE | 0101 |
| 1500 | B0DE |
| 1502 | F732 |
| | ⋮ |

*Address Calculation*

$$\begin{array}{r} \text{1502} \\ +\text{FFFE} \\ \hline \text{1500} \end{array}$$

*After Execution*

| | |
|---|---|
| R2 | B015 |
| R3 | FFFE |
| R4 | 0300 |
| R5 | 1502 |



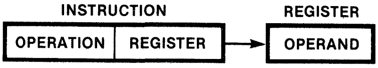
THE OPERAND VALUE IS THE CONTENTS OF THE LOCATION WHOSE ADDRESS IS THE CONTENTS OF REGISTER 2 OFFSET BY THE DISPLACEMENT IN REGISTER 1.

## 5.6 DESCRIPTIONS AND EXAMPLES (SEGMENTED Z8001 AND Z8003)

In this section, the notation <<nn>> is used to refer to segment number nn.

### 5.6.1 Register (R)

In the Register Addressing mode, the operand is taken from a specified general-purpose CPU register. Storing data in a register allows shorter instructions and faster execution than accessing memory.



THE OPERAND VALUE IS THE CONTENTS OF THE REGISTER.

The operand is always in a general-purpose CPU register. The register length (byte, word, register pair, or register quadruple) is specified by the instruction opcode.

**Assembler Language Formats (see also Chapter 6):**

| | |
|---|---|
| RHn, RLn | Byte register |
| Rn | Word register |
| RRn | Double-word register |
| RQn | Quadruple-word register |

**Example of R mode:**

```
LDL RR2,RR4       !Load the contents
                   of RR4 into RR2!
```

*Before Execution*

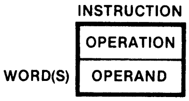
| | | |
|---|---|---|
| RR2 | R2 | A6B8 |
| | R3 | 9A20 |
| RR4 | R4 | 38A6 |
| | R5 | 745E |

*After Execution*

| | | |
|---|---|---|
| RR2 | R2 | 38A6 |
| | R3 | 745E |
| RR4 | R4 | 38A6 |
| | R5 | 745E |

### 5.6.2 Immediate (IM):

The Immediate Addressing mode does not indicate a register or memory address as the location of the source operand. The data processed by the instruction is in the instruction.



THE OPERAND VALUE IS IN THE INSTRUCTION.

Because an immediate operand is part of the instruction, it is always located in the program memory address space. Immediate mode is often used to initialize registers. The Z8000 is optimized for this function, providing several short immediate instructions to reduce the length of programs.

**Assembler language format (see also Chapter 6):**

#data

**Example of IM mode:**

```
LDB RH2, #%55   !load hex 55 into RH2!
```

*Before Execution*

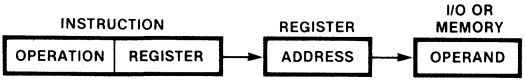
| | |
|---|---|
| R2 | 6789 |

*After Execution*

| | |
|---|---|
| R2 | 5589 |

### 5.6.3 Indirect Register (IR)

In the Indirect Register Addressing mode, the addressed register holds the address of the data.



THE OPERAND VALUE IS IN THE CONTENTS OF THE LOCATION WHOSE ADDRESS IS IN THE REGISTER.

Depending upon the instruction, the operand specified by IR mode will be located in either I/O address space (I/O instructions), Special I/O address space (Special I/O instructions), or data or stack memory address spaces. For non-I/O references, the status lines will indicate a stack access if the stack pointer (RR14) is used as the indirect register, otherwise they will indicate data reference.

A 16-bit register is used to hold an I/O or Special I/O address; a register pair is used to hold a memory address. Any general-purpose register, or register pair, except R0 or RR0 can be used.

The Indirect Register mode may save space and reduce execution time when consective locations are referenced. This mode can also be used to simulate more complex addressing modes, since addresses can be computed before the data is accessed.

**Assembler language formats (see also Chapter 6):**

| | |
|---|---|
| @Rn | Contains I/O or Special I/O address |
| @RRn | Contains memory address |

**Example of memory access using IR mode:**

```
LD R2, @RR4  !load into R2 the
               value in the memory
               location addressed
               by the contents of
               RR4!
```

*Before Execution*

| | | |
|---|---|---|
| RR2 | R2 | 030F |
| | R3 | 0005 |
| RR4 | R4 | 2000 |
| | R5 | 170C |

*Memory*

| | |
|---|---|
| | ⋮ |
| 170A* | A023 |
| 170C | 0B0E |
| 170E | 10D3 |
| | ⋮ |

* Segment Number 20

*After Execution*

| | | |
|---|---|---|
| RR2 | R2 | 0B0E |
| | R3 | 0005 |
| RR4 | R4 | 2000 |
| | R5 | 170C |

**Example of I/O using IR mode:**

```
OUTB @R1,RL0
```

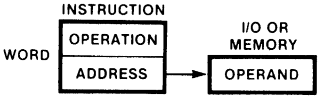
*Before Execution*

| | |
|---|---|
| R0 | 0A23 |
| R1 | 0011 |

Execution sends the data "23" to the I/O device addressed by "0011."

### 5.6.4 Direct Address (DA)

In the Direct Address mode, the operand address is specified in the instruction.



THE OPERAND VALUE IS THE CONTENTS OF THE LOCATION WHOSE ADDRESS IS IN THE REGISTER.

Depending upon the instruction, the operand specified by DA mode will be either in Standard I/O address space (Standard I/O instructions), Special I/O address space (Special I/O instructions), or in data memory space. I/O and Special I/O addresses are one word long; memory addresses can be either one or two words long, depending on whether the long or short format is used.

This mode is also used by Jump and Call instructions to specify the address of the next instruction to be executed. (Actually, the addresss serves as an immediate value that is loaded into the Program Counter.)

**Assembler language format (see also Chapter 6):**

| | |
|---|---|
| address | Either memory, I/O, or Special I/O, where double angle brackets "<<" and ">>" enclose the segment number, and vertical line delimiters "\|" and "\|" enclose short-form memory addresses. |

**Example of DA mode:**

```
LDB RH2,|<<15>>%23| !load RH2 with the
                     value in memory
                     segment 15, dis-
                     placement 23 (hex)!
```

*Before Execution*

R2 6789

*Memory*

```
           ⋮
<<15>> 0022 0206
       0024 0304
           ⋮
```

*After Execution*

R2 0689

### 5.6.5 Index (X).

In the Index Addressing mode, the address of the operand is computed by adding the contents of a word register (called the "index register") specified in the instruction to the address specified in the instruction.

The offset of the operand address is computed by adding the 16-bit index value to the offset portion of the segmented address specified in the instruction. The segment number of the operand address comes directly from the instruction. The segment number is unaffected by the offset computation; any overflow in the computation is ignored resulting in "wraparound". Indexed addressing allows random access to tables or other complex data structures where the address of the base of the table is known, but the particular element index must be computed by the program.

Any word register can be used as the index register except R0. The address in the instruction can be one or two words, depending on whether a long or short offset is used in the address.

Operands specified by X mode are always in the data memory address space.

**Assembler language format:**

Address(Rn)

**Example of X mode:**

```
LD R4, <<5>>%231A(R3)  !load into R4 the
                        contents of the
                        memory location
                        whose address is
                        segment 5,
                        displacement
                        231A + the
                        value in R3!
```

*Before Execution*

R3 01FE
R4 203A

*Memory*

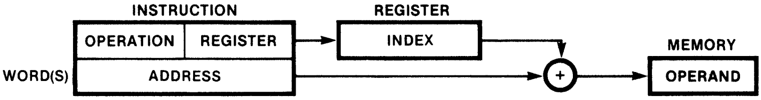
```
          ⋮
<<5>> 2516 F3C2
      2518 3D0E
      251A 7ADA
          ⋮
```

*Address Calculation*

```
<<5>>  %231A
+       01FE
-------------
<<5>>  %2518
```

*After Execution*

R3 01FE
R4 3D0E



THE OPERAND VALUE IS THE CONTENTS OF THE LOCATION WHOSE ADDRESS IS THE ADDRESS IN THE INSTRUCTION, OFFSET BY THE CONTENTS OF THE REGISTER.

### 5.6.6 Relative Address (RA)

In the Relative Address mode, the data processed is found at an address relative to the current instruction. The instruction specifies a two's complement displacement which is combined with the offset field of the Program Counter to form the target address. The Program Counter setting used is the address of the instruction following the currently executing instruction. (The assembler will take this into account in calculating the constant that is assembled into the instruction.)

An operand specified by RA mode is always in the program memory address space.

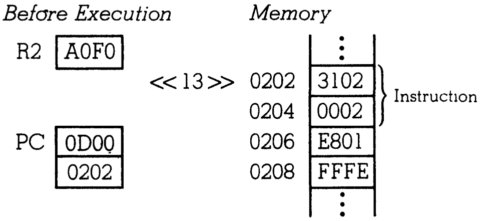
**Assembler language format (see also Chapter 6):**

Address

**Example of RA mode:**

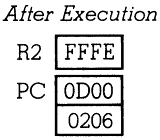
```
LDR R2, $+6  !load into R2 the con-
              tents of the memory
              location whose
              address is the
              current program
              counter + 6!
```

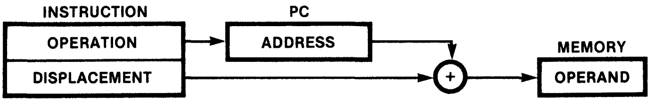
Because the program counter will be advanced to point to the next instruction before the address calculation is performed, the constant that occurs in the instruction is + 2.



*Address Calculation*

$$\begin{array}{l} \ll 13 \gg 0206 \\ \underline{+ \qquad\quad 2} \\ \ll 13 \gg 0208 \end{array}$$

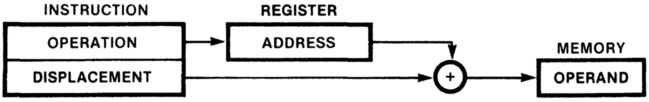




THE OPERAND VALUE IS THE CONTENTS OF THE LOCATION WHOSE ADDRESS IS THE CONTENTS OF PC OFFSET BY THE DISPLACEMENT IN THE INSTRUCTION.

### 5.6.7 Base Address (BA)

The Base address mode is similar to the Index mode in that a base and displacement are combined to produce the effective address. In Base addressing, a register pair contains the 23-bit segmented base address and the displacement is expressed as a 16-bit value in the instruction. The displacement is added to the offset of the base address, to obtain the operand address. (The segment number is not changed.) This addressing mode can only be used with the Load instructions. Base Addressing allows random access to records or other data structures where the displacement of an element within the structure is known, but the base of the structure must be computed by the program.



THE OPERAND VALUE IS THE CONTENTS OF THE LOCATION WHOSE ADDRESS IS THE ADDRESS IN THE REGISTER, OFFSET BY THE DISPLACEMENT IN THE INSTRUCTION.

Any double-word register except RR0 can be used for the base address. The Base Address mode allows access to locations whose segment numbers are not known at assembly time.

The status lines will indicate a stack access if the base register is the stack pointer (RR14) and will indicate a memory access otherwise.

If the segment number is known when the program is assembled (or loaded if the loader can resolve symbolic segment numbers), the Index Address mode may be used to simulate the Base Address mode. For example, if R2 is known to hold segment number 18, then the operand specified using the base address RR2 (#93) can also be referenced by the indexed address <<18>>%93(R3). The advantage of this simulation is that the Index mode is supported for most operations, whereas the Base mode is restricted to LOAD and LOAD ADDRESS.

**Assembler language format (see also Chapter 6):**

| | |
|---|---|
| RRn(#disp) | Add the immediate value to the contents of RRn; the result is the address of the operand. |

**Example of BA mode:**

```
LDL RR4(#%18), RR2   !load the long word
                      in RR2 into the
                      memory location
                      whose address is
                      the value of RR4
                      + hex 18!
```

*Before Execution*

| | | |
|---|---|---|
| RR2 | R2 | 0A00 |
| | R3 | 1500 |
| RR4 | R4 | 2500 |
| | R5 | 20AA |

*Data Memory*

| | | |
|---|---|---|
| | | ⋮ |
| <<31>> | 20C0 | 0ABE |
| | 20C2 | F50D |
| | 20C4 | BADE |
| | 20C6 | B0D1 |
| | | ⋮ |

*Address Calculation*

```
<<13>> 1502
+       FFEE
<<13>> 1500
```

*After Execution*

| | | |
|---|---|---|
| RR2 | R2 | 0A00 |
| | R3 | 1500 |
| RR4 | R4 | 2500 |
| | R5 | 20AA |

*Data Memory*

| | | |
|---|---|---|
| | | ⋮ |
| <<31>> | 20C0 | 0ABE |
| | 20C2 | 0A00 |
| | 20C4 | 1500 |
| | 20C6 | B0D1 |
| | | ⋮ |

### 5.6.8 Base Index (BX)

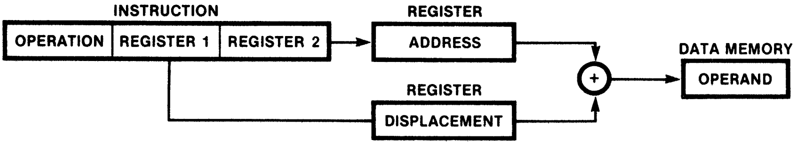
The Base Index addressing mode is an extension of the Base Addressing mode and can be used only with the Load and Load Address instructions. In this case, both the base address and index are held in registers. The index value is added to the offset of the base address to produce the offset of the operand address. The segment number of the operand address is the same as that of the base address. This mode allows access to memory locations whose addresses are computed at runtime and are not fully known at assembly time.

Any register pair can be used for the base address **except RR0.** Any word register **except R0** can be used for the index. Note that the Short Offset format for base addresses is not available in registers.

The status lines will indicate a stack access if the base register is the stack pointer (RR14) and will indicate a data access otherwise.

**Assembler language format (see also Chapter 6):**

RRn(Rn)



THE OPERAND VALUE IS THE CONTENTS OF THE LOCATION WHOSE ADDRESS IS THE CONTENTS OF REGISTER 2 OFFSET BY THE DISPLACEMENT IN REGISTER 1.

**Example of BX mode:**

```
LD R2, RR4(R3)  !load into R2 the value
                 whose address is the
                 contents of RR4 +
                 the contents of R3!
```

*Before Execution*

| | | |
|---|---|---|
| RR2 | R2 | 3535 |
| | R3 | FFFE |
| RR4 | R4 | 0D00 |
| | R5 | 1502 |

*Data Memory*

| | | |
|---|---|---|
| | | ⋮ |
| <<13>> | 14FE | 0101 |
| | 1500 | B0DE |
| | 1502 | F732 |
| | | ⋮ |

*Address Calculation*

```
<<13>> 1502
+      FFEE
<<13>> 1500
```

*After Execution*

| | | |
|---|---|---|
| RR2 | R2 | B0DE |
| | R3 | FFFE |
| RR4 | R4 | 0D00 |
| | R5 | 1502 |

*Data Memory*

| | | |
|---|---|---|
| | | ⋮ |
| <<13>> | 14FE | 0101 |
| | 1500 | B0DE |
| | 1502 | F732 |
| | | ⋮ |

# Chapter 6
# Instruction Set

## 6.1 INTRODUCTION

This chapter describes the instruction set of the Z8000 CPUs. An overview of the instruction set is presented first, in which the instructions are divided into ten functional groups. The instructions in each group are listed, followed by a summary description of the instructions. Significant characteristics shared by the instructions in the group, such as the available addressing modes, flags affected, or interruptibility, are described. Noteworthy instructions or features are mentioned.

Following the functional summary of the instruction set, flags and condition codes are discussed in relation to the instruction set. This is followed by a section discussing interruptibility of instructions and a description of traps. The last part of this chapter consists of a detailed description of each Z8000 instruction, listed in alphabetical order by mnemonic. This section is intended to be used as a reference by Z8000 programmers. The entry for each instruction includes a description of the instruction, addressing modes, assembly language mnemonics, instruction formats, execution times and simple examples illustrating the use of the instruction.

## 6.2 FUNCTIONAL SUMMARY

This section presents an overview of the Z8000 instructions. For this purpose, the instructions can be divided into ten functional groups:

- Load and Exchange
- Arithmetic
- Logical
- Program Control
- Bit Manipulation
- Rotate and Shift
- Block Transfer and String Manipulation
- Input/Output
- CPU Control
- Extended Instructions

### 6.2.1 Load and Exchange Instructions

| Instruction | Operand(s) | Name of Instruction |
|---|---|---|
| CLR<br>CLRB | dst | Clear |
| EX<br>EXB | dst,src | Exchange |
| LD<br>LDB<br>LDL | dst,src | Load |
| LDA | dst,src | Load Address |
| LDAR | dst,src | Load Address Relative |
| LDK | dst,src | Load Constant |
| LDM | dst,src,num | Load Multiple |
| LDR<br>LDRB<br>LDRL | dst,src | Load Relative |
| POP<br>POPL | dst,src | Pop |
| PUSH<br>PUSHL | dst,src | Push |

The Load and Exchange group includes a variety of instructions that provide for movement of data between registers, memory, and the program itself (e.g., immediate data). These instructions are supported with the widest range of addressing modes, including the Base (BA) and the Base

Index (BX) modes which are available only with the LD and LDA instructions. None of these instructions affect the CPU flags.

The Load and Load Relative instructions transfer a byte, word, or long word of data from the source operand to the destination operand. Special one-word instructions are also included to handle loading a small constant (0 to 15) into a register or an arbitrary constant into a byte register.

These instructions provide one of the following three functions:

- Load a register with data from a register or a memory location
- Load a memory location with data from a register
- Load a register or a memory location with immediate data

The memory location is specified using any of the addressing modes IR, DA, X, BA, BX, RA. The modes BA and BX are available with the LD and LDA instructions. The Relative addressing mode is used with the LDR and LDAR instructions.

The Clear and Clear byte instructions can be used to clear a register or memory location to zero. While this is functionally equivalent to a Load Immediate where the immediate data is zero, this operation occurs frequently enough to justify a special instruction that is more compact.

The Exchange instructions swap the source and destination operands.

The Load Multiple instruction provides for efficient saving and restoring of registers. This can lower the overhead of procedure calls and context switches such as those that occur at interrupts. The instruction allows any contiguous group of 1 to 16 registers to be transferred to or from a memory area, which can be designated using the DA, IR, or X addressing modes. (R0 is considered to follow R15, e.g. R9-R15 and R0-R3 can be saved with a single instruction.)

Stack operations are supported by the PUSH, PUSHL, POP, and POPL instructions. Any general-purpose register (or register pair in segmented mode) can be used as the stack pointer (except R0 and RR0). The source operand for the Push instructions and the destination operand for the Pop instructions can be a register or a memory location, specified by the DA, IR, or X addressing modes. Immediate data can also be pushed, one word at a time, onto a stack. Byte Push and Pop operations are not supported, and the stack pointer register must contain an even value when a stack instruction is executed. This is consistent with the general restriction of using even addresses for word and long word accesses.

The Load Address and Load Address Relative instructions compute the effective address for the DA, X, BA, BX, and RA modes and return the value in a register.

### 6.2.2 Arithmetic Instructions

| Instruction | Operand(s) | Name of Instruction |
|---|---|---|
| ADC<br>ADCB | dst,src | Add with<br>Carry |
| ADD<br>ADDB<br>ADDL | dst,src | Add |
| CP<br>CPB<br>CPL | dst,src | Compare |
| DAB | dst | Decimal<br>Adjust |
| DEC<br>DECB | dst,src | Decrement |
| DIV<br>DIVL | dst,src | Divide |
| EXTS<br>EXTSB<br>EXTSL | dst | Extend Sign |
| INC<br>INCB | dst,src | Increment |
| MULT<br>MULTL | dst,src | Multiply |
| NEG<br>NEGB | dst | Negate |
| SBC<br>SBCB | dst,src | Subtract with<br>Carry |
| SUB<br>SUBB<br>SUBL | dst,src | Subtract |

The Arithmetic group consists of instructions for performing integer arithmetic. The basic instructions use standard two's complement binary format and operations. Support is also provided for implementation of BCD arithmetic.

Most of the instructions in this group perform an operation between a register operand and a second operand designated by any of the five basic addressing modes (R, IR, DA, IM, X), and load the result into the register.

The arithmetic instructions, in general, alter the C, Z, S and P/V flags, which can then be tested by subsequent conditional jump instructions. The P/V flag is used to indicate arithmetic overflow for these instructions and it is referred to as the V (overflow) flag. The byte versions of these instructions generally alter the D and H flags as well.

The basic integer (binary) operations are performed on byte, word, or long word operands, although not all operand sizes are supported by all instructions. Multiple precision operations can be implemented in software using the Add with Carry, (ADC, ADCB), Subtract with Carry (SBC, SBCB) and Extend Sign (EXTS, EXTSB, EXTSL) instructions.

BCD operations are not provided directly, but can be implemented using a binary addition (ADDB, ADCB) or subtraction (SUBB, SBCB) followed by a decimal adjust instruction (DAB).

The Multiply and Divide instructions perform signed two's complement arithmetic on word or long word operands. The Multiply instruction (MULT) multiplies two 16-bit operands and produces a 32-bit result, which is loaded into the destination register pair. Multiply Long (MULTL) multiplies two 32-bit operands and produces a 64-bit result, which is loaded into the destination register quadruple. An overflow condition is never generated by a multiply, nor can a true carry be generated. The carry flag is used instead to indicate that the product has too many significant bits to be entirely contained in the low-order half of the destination.

The Divide instruction (DIV) divides a 32-bit number in the destination register pair by a 16-bit source operand and loads a 16-bit quotient into the low-order half of the destination register. A 16-bit remainder is loaded into the high-order half. Divide Long (DIVL) operates similarly with a 64-bit destination register quadruple and a 32-bit source. The overflow flag is set if the quotient is bigger than the low-order half of the destination, or if the source is zero.

### 6.2.3 Logical Instructions

| **Instruction** | **Operand(s)** | **Name of Instruction** |
|---|---|---|
| AND<br>ANDB | dst,src | And |
| COM<br>COMB | dst | Complement |
| OR<br>ORB | dst,src | Or |
| TEST<br>TESTB<br>TESTL | dst | Test |
| XOR<br>XORB | dst,src | Exclusive Or |

The instructions in this group perform logical operations on each of the bits of the operands. The operands may be bytes or words; logical operations on long words are not supported (except for TESTL) but are easily implemented with pairs of instructions.

The two-operand instructions, And (AND, ANDB), Or (OR, ORB) and Exclusive-Or (XOR, XORB) perform the appropriate logical operations on corresponding bits of the destination register and the source operands, which can be designated by any of the five basic addressing modes (R, IR, DA, IM, X). The result is loaded into the destination register.

Complement (COM, COMB) complements the bits of the destination operand. Finally, Test (TEST, TESTB, TESTL) performs the OR operation between the destination operand and zero and sets the flags accordingly. Test long (TESTL) allows 32-bit logical operations to be performed by two 16-bit logical operators followed by TESTL to set the flags. The Complement and Test instructions can use four basic addressing modes to specify the destination (IM mode is excluded).

The Logical instructions set the Z and S flags based on the result of the operation. The byte variants of these instructions also set the Parity Flag (P/V) if the parity of the result is even, while the word instructions leave this flag unchanged. The H and D flags are not affected by these instructions.

### 6.2.4 Program Control Instructions

| Instruction | Operand(s) | Name of Instruction |
|---|---|---|
| CALL | dst | Call Procedure |
| CALR | dst | Call Procedure Relative |
| DJNZ<br>DBJNZ | r,dst | Decrement and Jump if Not Zero |
| IRET | | Interrupt Return |
| JP | cc,dst | Jump |
| JR | cc,dst | Jump Relative |
| RET | cc | Return from Procedure |
| SC | src | System Call |

This group consists of the instructions that affect the Program counter (PC) and thereby control program flow. General-purpose registers and memory are not altered except for the processor stack pointer and the processor stack, which play a significant role in procedures and interrupts. An exception is Decrement And Jump If Not Zero (DJNZ), which uses a register as a loop counter. The flags are also preserved except for IRET which reloads the program status, including the flags, from the processor stack.

The Jump (JP) and Jump Relative (JR) instructions provide a conditional transfer of control to a new location if the processor flags satisfy the condition specified in the condition code field of the instruction. (See Section 6.4 for a description of condition codes.) Jump Relative is a one-word instruction that will jump to a specified address within an address range -254 to +256 bytes from the address of the JR instruction. Most conditional jumps in programs are made to locations only a few bytes away; the Jump Relative instruction exploits this fact to improve code compactness and efficiency.

Call (CALL) and Call Relative (CALR) instructions are used for calling procedures; the current contents of the PC are pushed onto the processor stack, and the effective address indicated by the instruction is loaded into the PC. The use of a procedure address stack in this manner allows straightforward implementation of nested and recursive procedures. Like Jump Relative, Call Relative provides a one-word instruction for calling nearby subroutines. However, a much larger range of address, -4092 to +4098 bytes for CALR instruction, is provided since subroutine calls exhibit less locality than normal control transfers.

Both Jump and Call instructions are available with the indirect register, indexed and relative address modes in addition to the direct address mode.

The Conditional Return instruction (RET) is a companion to the Call instruction; if the condition specified in the instruction is satisfied, it loads the PC from the stack and pops the stack.

A special instruction, Decrement And Jump if Not Zero (DJNZ, DBJNZ), implements the control part of the basic Pascal FOR loop in a one-word instruction.

System Call (SC) is used for controlled access to facilities provided by the operating system. It is implemented identically to a trap or interrupt: the current program status is pushed onto the system mode stack and a new program status is loaded from a dedicated part of the Program Status Area. An 8-bit immediate source field in the instruction can be retrieved from the stack by the software that handles system calls and interpreted as desired. For example the contents of this field can be used as an index into a dispatch table to implement a call to one of the services provided by the operating system.

Interrupt Return (IRET) is used for returning from interrupts and traps, including system calls, to the interrupted routines. This is a privileged instruction.

### 6.2.5 Bit Manipulation Instructions

| Instruction | Operand(s) | Name of Instruction |
|---|---|---|
| BIT<br>BITB | dst,src | Bit Test |
| RES<br>RESB | dst,src | Reset Bit |
| SET<br>SETB | dst,src | Set Bit |
| TSET<br>TSETB | dst | Test and Set |
| TCC<br>TCCB | cc,dst | Test Condition Code |

The instructions in this group are useful for manipulating individual bits in registers or memory.

The Bit Set (SET, SETB) and Bit Reset (RES, RESB) instructions set, or clear, a single bit in the destination byte or word, which can be in a register or in a memory location specified by any of the five basic addressing modes. The particular bit to be manipulated may be specified by a value (0 to 7 for byte, 0 to 15 for word) in the instruction itself or it may be specified by the contents of a register. In the latter case, the destination is restricted to a register. These instructions leave the flags unaffected.

The Bit Test instruction (BIT, BITB) tests a specified bit and sets the Z flag according to the state of the bit.

The Test and Set instruction (TSET, TSETB) can be used for implementing synchronization mechanisms such as semaphores between processes on the same or on different CPUs. When executed, by either a Z8003 or Z8004 CPU, the Test and Set instruction causes status code 1111 to be output during the instruction execution when a memory location is the operand. This code is used in a multiprocessor environment to ensure that only one processor at a time can access the access control semaphore of a shared resource.

Another instruction in this group, Test Condition Code (TCC, TCCB) sets the low order bit of the destination register if the state of the flags specified by the condition code in the instruction is true (See Section 6.6.B for a list of condition codes.) This may be used to control subsequent operation of the program after the flags have been changed by intervening instructions. It may also be used by language compilers for generating boolean values.

### 6.2.6 Rotate and Shift Instructions

| Instruction | Operand(s) | Name of Instruction |
|---|---|---|
| RL<br>RLB | dst,src | Rotate Left |
| RLC<br>RLCB | dst,src | Rotate Left through Carry |
| RLDB | dst,src | Rotate Left Digit |
| RR<br>RRB | dst,src | Rotate Right |
| RRC<br>RRCB | dst,src | Rotate Right through Carry |
| RRDB | dst,src | Rotate Right Digit |
| SDA<br>SDAB<br>SDAL | dst,src | Shift Dynamic Arithmetic |
| SDL<br>SDLB<br>SDLL | dst,src | Shift Dynamic Logical |
| SLA<br>SLAB<br>SLAL | dst,src | Shift Left Arithmetic |
| SLL<br>SLLB<br>SLLL | dst,src | Shift Left Logical |
| SRA<br>SRAB<br>SRAL | dst,src | Shift Right Arithmetic |
| SRL<br>SRLB<br>SRLB | dst,src | Shift Right Logical |

This group contains instructions for shifting and rotating the contents of data registers.

Shift Instructions are used to shift the contents of an operand arithmetically or logically in either direction. Three operand lengths are supported: 8, 16 and 32 bits. The amount of the shift, which may be any value up to the operand length, can be specified either by the contents of a field in the instruction or by the contents of a specified register.

Rotate instructions are used to rotate the contents of a specified byte or word register in either direction by either one or two bit positions; the carry bit can be included in the rotation. A pair of digit rotation instructions (RLDB, RRDB) are useful in manipulating BCD data.

This group includes instructions that provide a full complement of string comparision, string translation, and block transfer functions. Using these instructions, byte or word blocks of any length up to 64K bytes can be moved in memory, a byte or word string can be searched until a given value is found, two byte or word strings can be compared, and a byte string can be translated by using the value of each byte as the address of its own replacement in a translation table. Translate and Test instructions skip over a class of bytes specified by a translation table, detecting bytes with values of special interest.

All the operations can proceed through the data in either direction. Furthermore, the operations can be repeated automatically while decrementing a length counter until it is zero, or they can

### 6.2.7 Block Transfer and String Manipulation Instructions

| Instruction | Operand(s) | Name of Instruction |
|---|---|---|
| CPD<br>CPDB | dst,src,r,cc | Compare and Decrement |
| CPDR<br>CPDRB | dst,src,r,cc | Compare, Decrement, and Repeat |
| CPI<br>CPIB | dst,src,r,cc | Compare and Increment |
| CPIR<br>CPIRB | dst,src,r,cc | Compare, Increment, and Repeat |
| CPSD<br>CPSDB | dst,src,r,cc | Compare String, and Decrement |
| CPSDR<br>CPSDRB | dst,src,r,cc | Compare String, Decrement and Repeat |
| CPSI<br>CPSIB | dst,src,r,cc | Compare String, and Increment |
| CPSIR<br>CPSIRB | dst,src,r,cc | Compare String, Increment and Repeat |
| LDD<br>LDDB | dst,src,r | Load and Decrement |
| LDDR<br>LDDRB | dst,src,r | Load, Decrement, and Repeat |
| LDI<br>LDIB | dst,src,r | Load and Increment |
| LDIR<br>LDIRB | dst,src,r | Load, Increment, and Repeat |
| TRDB | dst,src,r | Translate and Decrement |
| TRDRB | dst,src,r | Translate, Decrement and Repeat |
| TRIB | dst,src,r | Translate and Increment |
| TRIRB | dst,src,r | Translate, Increment and Repeat |
| TRTDB | src1,src2,r | Translate, Test, and Decrement |
| TRTDRB | src1,src2,r | Translate, Test, Decrement, and Repeat |
| TRTIB | src1,src2,r | Translate, Test, and Increment |
| TRTIRB | src1,src2,r | Translate, Test, Increment, and Repeat |

operate on one storage unit per execution with the length counter decremented by one, and the source and destination pointer registers properly adjusted. The latter form is useful for adding other instructions within a loop containing the block instructions.

Any word register can be used as a length counter in most cases. If the execution of the instruction causes this register to be decremented to zero, the P/V flag is set. In most cases the auto-repeat forms of these instructions always leave this flag set.

The D and H flags are not affected by these instructions. The C anc S flags are preserved by all but the compare instructions.

These instructions use the Indirect Register (IR) addressing mode: the source and destination operands are addressed by the contents of general-purpose registers (word registers in nonsegmented mode and register pairs in segmented mode). In the segmented mode, only the low-order half of the register pair (the offset) is incremented or decremented. The segment number is never changed in Z8000 address arithmetic.

The repetitive forms of these instructions are interruptible. This is essential since the repetition count can be as high as 65,536 and the instructions can take 9 to 14 cycles for each iteration after the first one. The instruction can be interrupted after any iteration. If the instruction is not finished, the address of the instruction itself is saved on the stack together with the contents of the operand pointer registers and the repetition counter. The instruction can then be simply reissued after returning from the interrupt.

### 6.2.8 Input/Output Instructions

| Instruction | Operand(s) | Name of Instruction | Instruction | Operand(s) | Name of Instruction |
|---|---|---|---|---|---|
| IN<br>INB | dst,src | Input | SIN<br>SINB | dst,src | Special Input |
| IND<br>INDB | dst,src,r | Input and Decrement | SIND<br>SINDB | dst,src,r | Special Input and Decrement |
| INDR<br>INDRB | dst,src,r | Input, Decrement, and Repeat | SINDR<br>SINDRB | dst,src,r | Special Input, Decrement, and Repeat |
| INI<br>INIB | dst,src,r | Input and Increment | SINI<br>SINIB | dst,src,r | Special Input and Increment |
| INIR<br>INIRB | dst,src,r | Input, Increment, and Repeat | SINIR<br>SINIRB | dst,src,r | Special Input, Increment, and Repeat |
| OTDR<br>OTDRB | dst,src,r | Output, Decrement, and Repeat | SOTDR<br>SOTDRB | dst,src,r | Special Output, Decrement, and Repeat |
| OTIR<br>OTIRB | dst,src,r | Output, Increment, and Repeat | SOTIR<br>SOTIRB | dst,src,r | Special Output, Increment, and Repeat |
| OUT<br>OUTB | dst,src | Output | SOUT<br>SOUTB | dst,src | Special Output |
| OUTD<br>OUTDB | dst,src,r | Output and Decrement | SOUTD<br>SOUTDB | dst,src,r | Special Output and Decrement |
| OUTI<br>OUTIB | dst,src,r | Output and Increment | SOUTI<br>SOUTIB | dst,src,r | Special Output and Increment |

This group consists of instructions for transferring a byte, word, or block of data between peripheral devices and the CPU registers or memory. Two separate I/O address spaces with 16-bit addresses are recognized: a Standard I/O address space and a Special I/O address space. The latter is intended for use with special Z8000 Family devices, typically a Memory Management Unit (MMU). Instructions that operate on the Special I/O address space are prefixed with the word "special". Standard I/O and Special I/O instructions generate different codes on the CPU status lines but are otherwise identical. Normal 8-bit peripherals are connected to bus lines $AD_0$-$AD_7$. Standard I/O byte instructions use odd addresses only. Special 8-bit peripherals such as the MMU or the Paged Memory Management Unit (PMMU), which are used with Special I/O instructions, are connected to bus lines $AD_8$-$AD_{15}$. Special I/O byte instructions use even addresses only.

The instructions for transferring a single byte or word (IN, INB, OUT, OUTB, SIN, SINB, SOUT, SOUTB) can transfer data between any general-purpose register and any port in either the indicated I/O address space. For the Standard I/O instructions, the port number can be specified in the instruction or by the contents of the CPU register. For the Special I/O instructions the port number is specified statically.

The remaining instructions in this group transfer blocks of data between I/O ports and memory. The operation of these instructions is similar to that of the block move instructions described earlier, but one operand is always an I/O port which remains unchanged while the address of the other operand (a memory location) is incremented or decremented. These instructions are also interruptible.

All I/O instructions are privileged, i.e., they can be executed only in system mode. The single byte/word I/O instructions do not alter any flags. The block I/O instructions, including the single iteration variants, alter the Z and P/V flags. The latter is set when the repetition counter is decremented to zero.

The instructions in this group relate to the CPU control and status registers (FCW, PSAP, REFRESH, etc.), or perform functions that do not fit into any of the other groups, such as instructions that support multimicroprocessor operation. All of these instructions are privileged, with the exception of NOP and the instructions operating on the flags (SETFLG, RESFLG, COMFLG, LDCTLB).

### 6.2.9 CPU Control Instructions

| Instruction | Operand(s) | Name of Instruction |
|---|---|---|
| COMFLG | flag | Complement Flag |
| DI | int | Disable Interrupt |
| EI | int | Enable Interrupt |
| HALT | | Halt |
| LDCTL<br>LDCTLB | dst,src | Load Control Register |
| LDPS | src | Load Program Status |
| MBIT | | Multi-Micro Bit Test |
| MREQ | dst | Multi-Micro Request |
| MRES | | Multi-Micro Reset |
| MSET | | Multi-Micro Set |
| NOP | | No Operation |
| RESFLG | flag | Reset Flag |
| SETFLG | flag | Set Flag |

### 6.2.10 Extended Instructions

The Z8000 architecture includes a mechanism for extending the basic instruction set through the use of external devices known as Extended Processing Units (EPUs). (See Section 2.12 for a more comprehensive presentation of the Extended Processing Architecture.) Six opcodes, OE, OF, 4E, 4F, 8E and 8F (in hexadecimal), are dedicated to the implementation of extended instructions using this facility. Four addressing modes (R, IR, DA, and X) can be used by extended instructions for accessing data for the EPUs.

There are four types of extended instruction in the Z8000 CPU instruction repertoire: EPU internal operations, data transfers between memory and EPU, data transfers between EPU and CPU, and data transfers between EPU flag registers and the CPU flag and control word. The last type is useful when the program must branch based on conditions determined by the EPU.

Upon encountering extended instructions, the CPU's action is dependent upon the EPA control bit in the CPU's FCW. When this bit is set, the instruction is executed by the EPU. If this bit is clear, the CPU traps (an extended instruction trap) so that a trap handler can emulate the desired operation in software.

### 6.2.11 Privileged Instructions

The following list presents the names and mnemonics of the Z8000 Privileged Instructions:

Disable Interrupt (DI)
Enable Interrupt (EI)
Halt (HALT)
Input (IN)
Special Input (SIN)
Input and Decrement (IND)
Special Input and Decrement (SIND)
Input, Decrement and Repeat (INDR)
Special Input, Decrement and Repeat (SINDR)
Input and Increment (INI)
Special Input and Increment (SINI)
Input, Increment and Repeat (INIR)
Special Input, Increment and Repeat (SINIR)
Interrupt Return (IRET)
Load Control (LDCTL)
Load Program Status (LDPS)
Multi Micro Bit Test (MBIT)
Multi Micro Request (MREQ)
Multi Micro Reset (MRES)
Multi Micro Set (MSET)
Output, Decrement and Repeat (OTDR)
Special Output, Decrement and Repeat (SOTDR)
Output, Increment and Repeat (OTIR)
Special Output, Increment and Repeat (SOTIR)
Output (OUT)
Special Output (SOUT)
Output and Decrement (OUTD)
Special Output and Decrement (SOUTD)
Output and Increment (OUTI)
Special Output and Increment (SOUTI)

## 6.3 PROCESSOR FLAGS

The processor flags are part of the program status (Section 2.7.1). They provide a link between sequentially executed instructions in the sense that the result of executing one instruction may alter the flags, and the resulting value of the flags can be used to determine the operation of a subsequent instruction, typically a conditional jump instruction. For example, the use of a flag when a Test is followed by a Conditional Jump:

```
TEST R1         !sets Z FLAG if R1 = 0!
JR Z, DONE      !go to done if Z flag is set!
  .
  .
  .
DONE:
```

The program branches to DONE if the TEST sets the Z flag, i.e., if R1 contains zero.

The program status has six flags:

- Carry (C)
- Zero (Z)
- Sign (S)
- Parity/Overflow (P/V)
- Decimal Adjust (D)
- Half Carry (H)

Appendix C lists the instructions and the flags they affect. In addition, there are Z8000 CPU control instructions that allow the programmer to set, reset (clear), or complement any or all of the first four flags. The Half-Carry and Decimal-Adjust flags are used by the Z8000 processor for BCD arithmetic corrections.

The Flags register can be separately loaded by the Load Control Register (LDCTLB) instruction without disturbing the control bits in the other byte of the FCW. In fact, access to the Flags register is not a privileged operation, while access to the control bits is privileged. The contents of the Flag registers can also be saved in a register or memory.

The Carry (C) flag, when set, generally indicates a carry out of or a borrow into the high-order bit position of a register being used as an accumulator. For example, adding the 8-bit numbers 225 and 64 causes a carry out of bit 7 and sets the Carry flag:

| | | | | 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 plays an important role in the implementation of multiple-precision arithmetic (see the ADC, SBC instructions). It is also involved in the Rotate Left Through Carry (RLC) and Rotate Right Through Carry (RRC) instructions. These instructions are used to implement rotation, or shifting, of long strings of bits.

The Zero (Z) flag is set when the result register's contents are zero following certain operations. This is often useful for determining when a counter reaches zero. In addition, the block compare instructions use the Z flag to indicate that the specified comparison condition is satisfied.

The Sign (S) flag is set to one when the most significant bit of a result register contains a one (a negative number in two's complement notation) following certain operations.

The Overflow (V) flag, when set, indicates that a two's complement number in a result register has exceeded the largest number, or is less than the smallest number, that can be represented in a two's complement notation. This flag is set as the result of an arithmetic operation. Consider the following 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 |
| | 1 = Overflow flag set | | | | | | | |

The result in this case (-31 in two's complement notation) is incorrect, thus the overflow flag is set.

The same bit acts as a Parity (P) flag following logical instructions on byte operands. The number of one bits in the register is counted; if the total is even, parity is said to be even and the flag P is set (i.e. P=1). If the total is odd; parity is odd and the flag P is reset (i.e. P=0). This flag is referred to as the P/V flag.

The Block Move and String instructions and the Block I/O instructions use the P/V flag to indicate that the repetition counter has decremented to 0.

The Decimal-Adjust (D) flag is used for BCD arithmetic. Since the algorithm for correcting BCD operations is different for addition and subtraction, this flag is used to record whether an add or subtract instruction was executed so that the subsequent Decimal Adjust (DAB) instruction can perform its function correctly. See the DAB instruction for further discussion on the use of this flag.

The Half-Carry (H) flag indicates a carry out of bit 3 or a borrow into bit 3 as the result of adding or subtracting bytes containing two BCD digits each. This flag is used by the DAB instruction to convert the binary result of a previous decimal addition or subtraction into the correct decimal (BCD) result.

Neither the Decimal-Adjust nor the Half-Carry flag is used in condition codes and neither is affected by the SETFLG, RESFLG, or COMFLG instructions. The only access to these flags is through the LDCTL instruction.

## 6.4 CONDITION CODES

The first four flags, C, Z, S, and P/V, are used to control the operation of certain "conditional" instructions such as the Conditional Jump. The operation of these instructions is a function of whether a specified boolean condition on the four flags is satisfied or not. Sixteen of the flag settings are encoded in a 4-bit field called the condition code, which forms a part of all conditional instructions.

The condition codes and the flag settings they represent are listed in Section 6.6.

Although there are sixteen unique condition codes, more than sixteen mnemonics are used for the conditional codes. Some of the flag settings have more than one meaning for the programmer, depending on the context (PE & OV, Z & EQ, C & ULT, etc.).

## 6.5 INSTRUCTION INTERRUPTS AND TRAPS

Interrupts are discussed in detail in Chapter 7. This section looks at the relationship between instructions and interrupts.

When the CPU receives an interrupt request, and it is enabled for interrupts of that class, the interrupt is normally processed at the end of the current instruction. There are two exceptions: when an abort instruction interrupt occurs, the executing instruction is aborted immediately, and instructions which are designed to be interruptible so as to minimize the length of time it takes the CPU to respond to an interrupt. The latter type instructions are the iterative versions of the String and Block instructions and the Block I/O instructions. When an interrupt request is received during the execution of the iterative version of the string/block instructions, the instruction is suspended after the current iteration. The address of the instruction itself is saved on the stack, so that the same instruction is executed again when the interrupt handler executes an IRET. The contents of the repetition counter and the registers that index into the block operands are such that when the instruction is reissued upon returning from an interrupt, the effect is the same as if the instruction were not interrupted. The interrupt handler preserved the contents of the registers.

The longest noninterruptible instruction that can be used in normal mode is Divide Long (728 cycles in the worst case). Multi-Micro-Request, a privileged instruction, can take longer depending on the programmable propagation delay constant.

Traps are synchronous events that result from the execution of the previous instruction. The action of the CPU in response to a trap is similar to the response of an interrupt (see Chapter 7). Traps are nonmaskable.

The Z8000 CPUs implement four kinds of trap:

- Extended Instruction
- Privileged Instruction in Normal mode
- Addressing violation (segmentation trap in Z8001, and Segment/Address Translation Trap in Z8003)
- System Call

The Extended Instruction trap occurs when an Extended Instruction is encountered, but the EPA bit in the FCW is zero. This allows the same software to be run on Z8000 system configurations with or without EPUs. On systems without EPUs, the desired extended instructions can be emulated by software which is invoked by the Extended Instruction trap handler.

The System Call instruction always causes a trap. It is used to transfer control to system mode software in a controlled way, typically to request supervisor services.

The Privileged Instruction trap serves to protect the integrity of a system from erroneous or unauthorized actions of normal mode processes. Certain instructions, called privileged instructions, can only be executed in system mode. An attempt to execute one of these instructions in normal mode causes a Privileged Instruction trap. All the I/O instructions and the instructions that operate on the control portion of the FCW, such as instructions HALT and IRET, are privileged.

Address Violation traps are initiated by events external to the CPU such as the detection of an address violation by an MMU. This type of trap selects and initiates the routines needed to service or correct the detected violation. This type of trap is useful in enforcing access protection rules. Traps of this type are used on the Z8003 and Z8004 CPU for the implementation of virtual memory.

## 6.6 NOTATION AND BINARY ENCODING

The rest of this chapter consists of detailed descriptions of the instruction set, with the instructions listed in alphabetical order by mnemonic. This section describes the notational conventions used in the instruction descriptions and the binary encoding for some of the common instruction fields (e.g., register designation fields).

The description of each instruction begins on a new page. The instruction mnemonic(s) and name is printed in large bold letters at the top of each page to enable the reader to easily locate a desired description. The term "Privileged Instruction" is also printed at the top of each page which contains a description of this type of instruction.

The assembler language syntax is then given in a single generic form that covers all the variants of the instruction, along with a list of applicable addressing modes.

Example:

| | |
|---|---|
| AND dst,src | dst:R |
| ANDB | src:R, IM, IR, DA,X |

The description normally contains the following items in the given sequence:

1. The operations performed by the instruction.
2. A discussion of the overall operational aspects of the functions performed by the instruction.
3. The effect the instruction has on each of the processor flags is given.
4. A table that illustrates all of the variants of the instruction for each applicable addressing mode and operand size. The following information is presented for each of the variants.

**A. Instruction Mnemonics.** An instruction specification is shown for each applicable operand size (byte, word, or long). The instruction mnemonic code is given in upper case characters; lower case characters represent the variable part of the instruction specification for which suitable values are to be substituted. For example,

ADD Rd,#data

represents a statement of the form

ADD R3,#35.

The following notation is used for register operands:

| | |
|---|---|
| Rd, Rs: | a word register in the range R0-R15 |
| Rbd, Rbs: | a byte register RHn or RLn where n is within the range of 0 - 7 |
| RRd, RRs: | a register pair RR0, RR2, ...,RR14 |
| RQd: | a register quadruple RR0, RR4, RR8, or RR12 |

The "s" or "d" represents a source or destination operand. To simplify presentation, the terms source and destination are used when the instruction arguments referred to are not strictly source or destination operands. For example:

EX dst,src or MREQ dst

Address registers used in the Indirect, Base, and Base Index addressing modes represent word registers in nonsegmented mode and register pairs in segmented mode; this situation is flagged and an explanation is given in a footnote.

**B. Instruction Format.** The binary encoding of the instruction is given in each case for both the nonsegmented and segmented modes. Where applicable, both the short and long forms of the segmented version are given (SS and SL).

The instruction formats for byte and word versions of an instruction are usually combined. A single bit, labeled, "w", distinguishes them: a one indicates a word instruction, while a zero indicates a byte instruction.

Fields specifying register operands are identified with the same symbols (Rs, RRd, etc.) as described in item A. In some cases, only nonzero values are permitted for certain registers, such as index registers. This is indicated by a notation of the form "Rs ≠ 0."

The binary encoding for register fields is as follows:

| Register | | | | Binary | Hex |
|---|---|---|---|---|---|
| RQ0 | RR0 | R0 | RH0 | 0000 | 0 |
| | | R1 | RH1 | 0001 | 1 |
| | RR2 | R2 | RH2 | 0010 | 2 |
| | | R3 | RH3 | 0011 | 3 |
| RQ4 | RR4 | R4 | RH4 | 0100 | 4 |
| | | R5 | RH5 | 0101 | 5 |
| | RR6 | R6 | RH6 | 0110 | 6 |
| | | R7 | RH7 | 0111 | 7 |
| RQ8 | RR8 | R8 | RL0 | 1000 | 8 |
| | | R9 | RL1 | 1001 | 9 |
| | RR10 | R10 | RL2 | 1010 | A |
| | | R11 | RL3 | 1011 | B |
| RQ12 | RR12 | R12 | RL4 | 1100 | C |
| | | R13 | RL5 | 1101 | D |
| | RR14 | R14 | RL6 | 1110 | E |
| | | R15 | RL7 | 1111 | F |

In the case of relative addresses, the instruction format contains a "displacement". The actual value of this argument is dependent on the value of the PC at the time the instruction is executed.

A condition code is indicated by "cc" in the instruction formats. The condition codes, the flag settings they represent, and the binary encoding in the instructions are as follows:

| **Code** | **Meaning** | **Flag Setting** | **Binary** |
|---|---|---|---|
| F | Always false | - | 0000 |
| | 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 |
| PE | Parity even | P = 1 | 0100 |
| PO | Parity odd | P = 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 |
| UGE | Unsigned greater than or equal | C = 0 | 1111 |
| ULT | Unsigned less than | C = 1 | 0111 |
| UGT | Unsigned greater than | ((C = 0) AND (Z = 0)) = 1 | 1011 |
| ULE | Unsigned less than or equal | (C OR Z) = 1 | 0011 |

Notes: 1. Some of the condition codes correspond to identical flag settings: i.e., Z-EQ, NZ-NE, NC-UGE, PE-OV, PO-NOV.

**C. Cycles.** This line gives the execution time of the instructions in CPU cycles.

**D. Example.** An instruction specification example is given.

## 6.7 Z8000 INSTRUCTION DESCRIPTIONS AND FORMATS

The remainder of this chapter consists of individual descriptions of each Z8000 instruction. These descriptions are arranged in alphabetical order for ease in reference. Bold running heads are provided to enable the reader to easily locate a specific description.

# ADC
## Add With Carry

**ADC** dst, src    dst: R
**ADCB**    src: R

**Operation:** dst ◄— dst + src + c

The source operand, along with the setting of the carry flag, is added to the destination operand and the sum is stored in the destination. The contents of the source are not affected. Two's complement addition is performed. In multiple precision arithmetic, this instruction permits the carry from the addition of low-order operands to be carried into the addition of high-order operands.

**Flags:**

**C:** Set if there is a carry from the most significant bit of the result; cleared otherwise
**Z:** Set if the result is zero; cleared otherwise
**S:** Set if the result is negative; cleared otherwise
**V:** Set if arithmetic overflow occurs, that is, if both operands were of the same sign and the result is of the opposite sign; cleared otherwise
**D:** ADC—unaffected; ADCB—cleared
**H:** ADC—unaffected; ADCB—set if there is a carry from the most significant bit of the low-order four bits of the result; cleared otherwise

| Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles | Instruction Format | Cycles |
| R: | ADC Rd, Rs<br>ADCB Rbd, Rbs | 10 \| 11010 \| W \| Rs \| Rd | 5 | 10 \| 11010 \| W \| Rs \| Rd | 5 |

**Example:** Long addition can be done with the following instruction sequence, assuming R0, R1 contain one operand and R2, R3 contain the other operand:

```
ADD  R1,R3        !add low-order words!
ADC  R0,R2        !add carry and high-order words!
```

If R0 contains %0000, R1 contains %FFFF, R2 contains %4320 and R3 contains %0001, then the above two instructions leave the value %4321 in R0 and %0000 in R1.

# ADD
## Add

**ADD** dst, src
**ADDB**
**ADDL**

dst: R
src: R, IM, IR, DA, X

**Operation:** dst ← dst + src

The source operand is added to the destination operand and the sum is stored in the destination. The contents of the source are not affected. Two's complement addition is performed.

**Flags:**

**C:** Set if there is a carry from the most significant bit of the result; cleared otherwise
**Z:** Set if the result is zero; cleared otherwise
**S:** Set if the result is negative; cleared otherwise
**V:** Set if arithmetic overflow occurs, that is, if both operands were of the same sign and the result is of the opposite sign; cleared otherwise
**D:** ADD, ADDL—unaffected; ADDB—cleared
**H:** ADD, ADDL—unaffected; ADDB—set if there is a carry from the most significant bit of the low-order four bits of the result; cleared otherwise

| Source Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles | Instruction Format | Cycles |
| R: | ADD Rd, Rs<br>ADDB Rbd, Rbs | 10 00000 W Rs Rd | 4 | 10 00000 W Rs Rd | 4 |
| | ADDL RRd, RRs | 10 010110 RRs RRd | 8 | 10 010110 RRs RRd | 8 |
| IM: | ADD Rd, #data | 00 000001 0000 Rd<br>data | 7 | 00 000001 0000 Rd<br>data | 7 |
| | ADDB Rbd, #data | 00 000000 0000 Rbd<br>data data | 7 | 00 000000 0000 Rbd<br>data data | 7 |
| | ADDL RRd, #data | 00 010110 0000 RRd<br>31 data (high) 16<br>15 data (low) 0 | 14 | 00 010110 0000 RRd<br>31 data (high) 16<br>15 data (low) 0 | 14 |
| IR: | ADD Rd, @Rs[1]<br>ADDB Rbd, @Rs[1] | 00 00000 W Rs≠0 Rd | 7 | 00 00000 W RRs≠0 Rd | 7 |
| | ADDL RRd, @Rs[1] | 00 010110 Rs≠0 RRd | 14 | 00 010110 RRs≠0 RRd | 14 |

| Source Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | | |
|---|---|---|---|---|---|---|
| | | Instruction Format | Cycles | | Instruction Format | Cycles |
| **DA:** | ADD Rd, address<br>ADDB Rbd, address | 01 00000 W 0000 Rd<br>address | 9 | SS | 01 00000 W 0000 Rd<br>0 segment offset | 10 |
| | | | | SL | 01 00000 W 0000 Rd<br>1 segment 00000000<br>offset | 12 |
| | ADDL RRd, address | 01 010110 0000 RRd<br>address | 15 | SS | 01 010110 0000 RRd<br>0 segment offset | 16 |
| | | | | SL | 01 010110 0000 RRd<br>1 segment 00000000<br>offset | 18 |
| **X:** | ADD Rd, addr(Rs)<br>ADDB Rbd, addr(Rs) | 01 00000 W Rs≠0 Rd<br>address | 10 | SS | 01 00000 W Rs≠0 Rd<br>0 segment offset | 10 |
| | | | | SL | 01 00000 W Rs≠0 Rd<br>1 segment 00000000<br>offset | 13 |
| | ADDL RRd, addr(Rs) | 01 010110 Rs≠0 RRd<br>address | 16 | SS | 01 010110 Rs≠0 RRd<br>0 segment offset | 16 |
| | | | | SL | 01 010110 Rs≠0 RRd<br>1 segment 00000000<br>offset | 19 |

**Example:** ADD R2, AUGEND !augend A located at %1254!

Before instruction execution:

| Memory | | R2 | Flags |
|---|---|---|---|
| 1252 | | B D 2 1 | C Z S P/V D H |
| 1254 | 0 6 4 4 | | c z s p d h |
| 1256 | | | |

After instruction execution:

| Memory | | R2 | Flags |
|---|---|---|---|
| 1252 | | C 3 6 5 | C Z S P/V D H |
| 1254 | 0 6 4 4 | | 0 0 1 0 d h |
| 1256 | | | |

Note 1: Word register in nonsegmented mode, register pair in segmented mode

# AND
## And

**AND** dst, src dst: R
**ANDB** src: R, IM, IR, DA, X

**Operation:** dst ◄– dst AND src

A logical AND operation is performed between the corresponding bits of the source and destination operands, and the result is stored in the destination. A one bit is stored wherever the corresponding bits in the two operands are both ones; otherwise a zero bit is stored. The source contents are not affected.

**Flags:**
**C:** Unaffected
**Z:** Set if the result is zero; cleared otherwise
**S:** Set if the most significant bit of the result is set; cleared otherwise
**P:** AND — unaffected; ANDB — set if parity of the result is even; cleared otherwise
**D:** Unaffected
**H:** Unaffected

| Source Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | | |
|---|---|---|---|---|---|---|
| | | Instruction Format | Cycles | | Instruction Format | Cycles |
| R: | AND Rd, Rs<br>ANDB Rbd, Rs | 10 00011 W Rs Rd | 4 | | 10 00011 W Rs Rd | 4 |
| IM: | AND Rd, #data | 00 000111 0000 Rd<br>data | 7 | | 00 000111 0000 Rd<br>data | 7 |
| | ANDB Rbd, #data | 00 000110 0000 Rbd<br>data data | 7 | | 00 000110 0000 Rbd<br>data data | 7 |
| IR: | AND Rd, @Rs[1]<br>ANDB Rbd, @Rs[1] | 00 00011 W Rs≠0 Rd | 7 | | 00 00011 W RRs≠0 Rd | 7 |
| DA: | AND Rd, address<br>ANDB Rbd, address | 01 00011 W 0000 Rd<br>address | 9 | SS | 01 00011 W 0000 Rd<br>0 segment offset | 10 |
| | | | | SL | 01 00011 W 0000 Rd<br>1 segment 0000 0000<br>offset | 12 |
| X: | AND Rd, addr(Rs)<br>ANDB Rbd, addr(Rs) | 01 00011 W Rs≠0 Rd<br>address | 10 | SS | 01 00011 W Rs≠0 Rd<br>0 segment offset | 10 |
| | | | | SL | 01 00011 W Rs≠0 Rd<br>1 segment 0000 0000<br>offset | 13 |

**Example:** ANDB RL3, # %CE

Before instruction execution:

| RL3 |
|---|
| 1 1 1 0 0 1 1 1 |

| Flags |
|---|
| C Z S P/V D H |
| c z s p d h |

After instruction execution:

| RL3 |
|---|
| 1 1 0 0 0 1 1 0 |

| Flags |
|---|
| C Z S P/V D H |
| c 0 1 1 d h |

Note 1: Word register in nonsegmented mode, register pair in segmented mode.

# BIT
## Bit Test

| | |
|---|---|
| **BIT** dst, src<br>**BITB** | dst: R, IR, DA, X<br>src: IM<br>or<br>dst: R<br>src: R |

**Operation:** Z ← NOT dst (src)

The specified bit within the destination operand is tested, and the Z flag is set to one if the specified bit is zero; otherwise the Z flag is cleared to zero. The contents of the destination are not affected. The bit number (the source) can be specified statically as an immediate value, or dynamically as a word register whose contents are the bit number. In the dynamic case, the destination operand must be a register, and the source operand must be R0 through R7 for BITB, or R0 through R15 for BIT. The bit number is a value from 0 to 7 for BITB, or 0 to 15 for BIT, with 0 indicating the least significant bit. Note that only the lower four bits of the source operand are used to specify the bit number for BIT, while only the lower three bits of the source operand are used for BITB.

**Flags:**

**C:** Unaffected
**Z:** Set if specified bit is zero; cleared otherwise
**S:** Unaffected
**V:** Unaffected
**D:** Unaffected
**H:** Unaffected

## Bit Test Static

| Destination Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | | |
|---|---|---|---|---|---|---|
| | | Instruction Format | Cycles | | Instruction Format | Cycles |
| R: | BIT Rd, #b<br>BITB Rbd, #b | 10 \| 10011 \| W \| Rd \| b | 4 | | 10 \| 10011 \| W \| Rd \| b | 4 |
| IR: | BIT @ Rd[1], #b<br>BITB @ Rd[1], #b | 00 \| 10011 \| W \| Rd≠0 \| b | 8 | | 00 \| 10011 \| W \| RRd≠0 \| b | 8 |
| DA: | BIT address, #b<br>BITB address, #b | 01 \| 10011 \| W \| 0000 \| b<br>address | 10 | SS | 01 \| 10011 \| W \| 0000 \| b<br>0 \| segment \| offset | 11 |
| | | | | SL | 01 \| 10011 \| W \| 0000 \| b<br>1 \| segment \| 0000 0000<br>offset | 13 |
| X: | BIT addr(Rd), #b<br>BITB addr(Rd), #b | 01 \| 10011 \| W \| Rd≠0 \| b<br>address | 11 | SS | 01 \| 10011 \| W \| Rd≠0 \| b<br>0 \| segment \| offset | 11 |
| | | | | SL | 01 \| 10011 \| W \| Rd≠0 \| b<br>1 \| segment \| 0000 0000<br>offset | 14 |

## Bit Test Dynamic

| Source Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles | Instruction Format | Cycles |
| R: | BIT Rd, Rs<br>BITB Rbd, Rs[2] | 00 10011 W 0000 Rs<br>0000 Rd 0000 0000 | 10 | 00 10011 W 0000 Rs<br>0000 Rd 0000 0000 | 10 |

**Example:** If register RH2 contains %B2 (10110010), the instruction

BITB RH2, #0

will leave the Z flag set to 1.

Note 1: Word register in nonsegmented mode, register pair in segmented mode.
Note 2: Word registers 0-7 only.

# CALL
## Call

**CALL** dst        dst: IR, DA, X

**Operation:**

| Nonsegmented | Segmented |
|---|---|
| SP ← SP − 2 | SP ← SP − 4 |
| @SP ← PC | @SP ← PC |
| PC ← dst | PC ← dst |

The current contents of the program counter (PC) are pushed onto the top of the processor stack. The stack pointer used is R15 in nonsegmented mode, or RR14 in segmented mode. (The program counter value used is the address of the first instruction following the CALL instruction.) The specified destination address is then loaded into the PC and points to the first instruction of the called procedure.
At the end of the procedure a RET instruction can be used to return to original program. RET pops the top of the processor stack back into the PC.

**Flags:** No flags affected

| Destination Addressing Mode | Assembler Language Syntax | Nonsegmented Mode |  | Segmented Mode |  |
|---|---|---|---|---|---|
|  |  | Instruction Format | Cycles | Instruction Format | Cycles |
| IR: | CALL @Rd[1] | 00 011111 Rd≠0 0000 | 10 | 00 011111 RRD≠0 0000 | 15 |
| DA: | CALL address | 01 011111 0000 0000<br>address | 12 | SS 01 011111 0000 0000<br>0 segment offset | 18 |
|  |  |  |  | SL 01 011111 0000 0000<br>1 segment 0000 0000<br>offset | 20 |
| X: | CALL addr(Rd) | 01 011111 Rd≠0 0000<br>address | 13 | SS 01 011111 Rd≠0 0000<br>0 segment offset | 18 |
|  |  |  |  | SL 01 011111 Rd≠0 0000<br>1 segment 0000 0000<br>offset | 21 |

**Example:** In nonsegmented mode, if the contents of the program counter are %1000 and the contents of the stack pointer (R15) are %3002, the instruction

CALL   %2520

causes the stack pointer to be decremented to %3000, the value %1004 (the address following the CALL instruction with direct address mode specified) to be loaded into the word at location %3000, and the program counter to be loaded with the value %2520. The program counter now points to the address of the first instruction in the procedure to be executed.

Note 1: Word register in nonsegmented mode, register pair in segmented mode.

# CALR
## Call Relative

**CALR** dst dst: RA

**Operation:**

| Nonsegmented | Segmented |
|---|---|
| SP ← SP − 2 | SP ← SP − 4 |
| @SP ← PC | @SP ← PC |
| PC ← PC + (2 × displacement) | PC ← PC+(2 × displacement) |

The current contents of the program counter (PC) are pushed onto the top of the processor stack. The stack pointer used is R15 in nonsegmented mode, or RR14 in segmented mode. (The program counter value used is the address of the first instruction following the CALR instruction.) The destination address is calculated and then loaded into the PC and points to the first instruction of a procedure.

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

The destination address is the sum of twice the displacement in the instruction and the current value of the PC. The displacement is a 12-bit signed value in the range -2048 to +2047. Thus, the destination address must be in the range -4094 to +4096 bytes from the start of the CALR instruction. In segmented mode, the PC segment number is not affected. The assembler automatically calculates the displacement by subtracting the PC value of the following instruction from the address given by the programmer and dividing the result by 2.

**Flags:** No flags affected

| Destination Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles | Instruction Format | Cycles |
| **RA:** | CALR address | 1101 displacement | 10 | 1101 displacement | 15 |

**Example:** In nonsegmented mode, if the contents of the program counter are %1000 and the contents of the stack pointer (R15) are %3002, the instruction

```
CALR PROC
```

causes the stack pointer to be decremented to %3000, the value %1004 (the address following the CALR instruction) to be loaded into the word location %3000, and the program counter to be loaded with the address of the first instruction in procedure PROC.

# CLR
## Clear

**CLR** dst dst: R, IR, DA, X
**CLRB**

**Operation:** dst ← 0

The destination is cleared to zero.

**Flags:** No flags affected.

| Destination Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | | |
|---|---|---|---|---|---|---|
| | | Instruction Format | Cycles | | Instruction Format | Cycles |
| R: | CLR Rd<br>CLRB Rbd | 10 00110 W Rd 1000 | 7 | | 10 00110 W Rd 1000 | 7 |
| IR: | CLR @Rd[1]<br>CLRB @Rd[1] | 00 00110 W Rd ≠ 0 1000 | 8 | | 00 00110 W RRd≠0 1000 | 8 |
| DA: | CLR address<br>CLRB address | 01 00110 W 0000 1000<br>address | 11 | SS | 01 00110 W 0000 1000<br>0 segment offset | 12 |
| | | | | SL | 01 00110 W 0000 1000<br>1 segment 0000 0000<br>offset | 14 |
| X: | CLR addr(Rd)<br>CLRB addr(Rd) | 01 00110 W Rd≠0 1000<br>address | 12 | SS | 01 00110 W Rd≠0 1000<br>0 segment offset | 12 |
| | | | | SL | 01 00110 W Rd≠0 1000<br>1 segment 0000 0000<br>offset | 15 |

**Example:** If the word at location %ABBA contains 13, the statement

CLR %ABBA

will leave the value 0 in the word at location %ABBA.

[1] Note 1: Word register in nonsegmented mode, register pair in segmented mode.

# COM
## Complement

**COM** dst dst: R, IR, DA, X
**COMB**

**Operation:** (dst ← NOT dst)

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

**Flags:**
**C:** Unaffected
**Z:** Set if the result is zero; cleared otherwise
**S:** Set if the most significant bit of the result is set; cleared otherwise
**P:** COM—unaffected; COMB—set if parity of the result is even; cleared otherwise
**D:** Unaffected
**H:** Unaffected

| Destination Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | | |
|---|---|---|---|---|---|---|
| | | Instruction Format | Cycles | | Instruction Format | Cycles |
| R: | COM Rd<br>COMB Rbd | 10 00110 W Rd 0000 | 7 | | 10 00110 W Rd 0000 | 7 |
| IR: | COM @Rd[1]<br>COMB @Rd[1] | 00 00110 W Rd ≠ 0 0000 | 12 | | 00 00110 W RRd≠0 0000 | 12 |
| DA: | COM address<br>COMB address | 01 00110 W 0000 0000<br>address | 15 | SS | 01 00110 W 0000 0000<br>0 segment offset | 16 |
| | | | | SL | 01 00110 W 0000 0000<br>1 segment 0000 0000<br>offset | 18 |
| X: | COM addr(Rd)<br>COMB addr(Rd) | 01 00110 W Rd≠0 0000<br>address | 16 | SS | 01 00110 W Rd≠0 0000<br>0 segment offset | 16 |
| | | | | SL | 01 00110 W Rd≠0 0000<br>1 segment 0000 0000<br>offset | 19 |

**Example:** If register R1 contains %2552 (0010010101010010), the statement

COM R1

will leave the value %DAAD (1101101010101101) in R1.

[1] Note 1 Word register in nonsegmented mode, register pair in segmented mode

# COMFLG
## Complement Flag

**COMFLG** flag  Flag: C, Z, S, P, V
FLAGS (4:7) ◄– FLAGS (4:7) XOR instruction (4:7)

**Operation:** Any combination of the C, Z, S, P or V flags is complemented (each one bit is changed to zero, and vice-versa). The flags to be complemented are encoded in a field in the instruction. If the bit in the field is one, the corresponding flag is complemented; if the bit is zero, the flag is left unchanged. Note that the P and V flags are represented by the same bit. There may be one, two, three or four operands in the assembly language statement, in any order.

**Flags:**
**C:** Complemented if specified; unaffected otherwise
**Z:** Complemented if specified; unaffected otherwise
**S:** Complemented if specified; unaffected otherwise
**P/V:** Complemented if specified; unaffected otherwise
**D:** Unaffected
**H:** Undefined

| | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | **Instruction Format** | **Cycles** | **Instruction Format** | **Cycles** |
| | COMFLG flags | 10001101 \| CZSP/V \| 0101 | 7 | 10001101 \| CZSP/V \| 0101 | 7 |

**Example:** If the C, Z, and S flags are all clear ( = 0), and the P flag is set ( = 1), the statement

COMFLG P, S, Z, C

will leave the C, Z, and S flags set ( = 1), and the P flag cleared ( = 0).

# CP
## Compare

| | |
|---|---|
| **CP** dst, src | dst: R |
| **CPB** | src: R, IM, IR, DA, X |
| **CPL** | or |
| | dst: IR, DA, X |
| | src: IM |

**Operation:** dst – src

The source operand is compared to (subtracted from) the destination operand, and the appropriate flags set accordingly, which may then be used for arithmetic and logical conditional jumps. Both operands are unaffected, with the only action being the setting of the flags. Subtraction is performed by adding the two's complement of the source operand to the destination operand. There are two variants of this instruction: Compare Register compares the contents of a register against an operand specified by any of the five basic addressing modes; Compare Immediate performs a comparison between an operand in memory and an immediate value.

**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
**S:** Set if the result is negative; cleared otherwise
**V:** Set if arithmetic overflow occurs, that is, if both operands were of opposite signs and the sign of the result is the same as the sign of the source; cleared otherwise
**D:** Unaffected
**H:** Unaffected

## Compare Register

| Source Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles | Instruction Format | Cycles |
| **R:** | CP Rd, Rs<br>CPB Rbd, Rbs | 10 00101 W Rs Rd | 4 | 10 00101 W Rs Rd | 4 |
| | CPL RRd, RRs | 10 010000 RRs RRd | 8 | 10 010000 RRs RRd | 8 |
| **IM:** | CP Rd, #data | 00 001011 0000 Rd<br>data | 7 | 00 001011 0000 Rd<br>data | 7 |
| | CPB Rbd, #data | 00 001010 0000 Rbd<br>data data | 7 | 00 001010 0000 Rbd<br>data data | 7 |
| | CPL RRd, #data | 00 010000 0000 RRd<br>31 data (high) 16<br>15 data (low) 0 | 14 | 00 010000 0000 RRd<br>31 data (high) 16<br>15 data (low) 0 | 14 |
| **IR:** | CP Rd, @Rs[1]<br>CPB Rbd, @Rs[1] | 00 00101 W Rs≠0 Rd | 7 | 00 00101 W Rs≠0 Rd | 7 |
| | CPL RRd, @Rs[1] | 00 010000 Rs≠0 RRd | 14 | 00 010000 Rs≠0 RRd | 14 |

| Source Addressing Mode | Assembler Language Syntax | Nonsegmented Mode |  | Segmented Mode |  |
|---|---|---|---|---|---|
|  |  | Instruction Format | Cycles | Instruction Format | Cycles |
| DA: | CP Rd, address<br>CPB Rbd, address | 01 00101 W 0000 Rd<br>address | 9 | SS 01 00101 W 0000 Rd<br>0 segment offset | 10 |
|  |  |  |  | SL 01 00101 W 0000 Rd<br>1 segment 0000 0000<br>offset | 12 |
|  | CPL RRd, address | 01 010000 0000 RRd<br>address | 15 | SS 01 010000 0000 RRd<br>0 segment offset | 16 |
|  |  |  |  | SL 01 010000 0000 Rd<br>1 segment 0000 0000<br>offset | 18 |
| X: | CP Rd, addr(Rs)<br>CPB Rbd, addr(Rbs) | 01 00101 W Rs≠0 Rd<br>address | 10 | SS 01 00101 W Rs≠0 RRd<br>0 segment offset | 10 |
|  |  |  |  | SL 01 00101 W Rs≠0 Rd<br>1 segment 0000 0000<br>offset | 13 |
|  | CPL RRd, addr(Rs) | 01 010000 Rs ≠ 0 RRd<br>address | 16 | SS 01 010000 Rs≠0 RRd<br>0 segment offset | 16 |
|  |  |  |  | SL 01 010000 Rs≠0 RRd<br>1 segment 0000 0000<br>offset | 19 |

## Compare Immediate

| Destination Addressing Mode | Assembler Language Syntax | Nonsegmented Mode |  | Segmented Mode |  |
|---|---|---|---|---|---|
|  |  | Instruction Format | Cycles | Instruction Format | Cycles |
| IR: | CP @Rd[1], #data | 00 00110 W Rd ≠ 0 0001<br>data | 11 | 00 00110 W RRd≠0 0001<br>data | 11 |
|  | CPB @Rd[1], #data | 00 00110 W Rd≠0 0001<br>data data | 11 | 00 00110 W RRd≠0 0001<br>data data | 11 |

| Destination Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles | Instruction Format | Cycles |
| DA: | CP address, #data | 01 00110 W 0000 0001<br>address<br>data | 14 | SS 01 00110 W 0000 0001<br>0 segment offset<br>data | 15 |
| | | | | SL 01 00110 W 0000 0001<br>1 segment 0000 0000<br>offset<br>data | 17 |
| | CPB address, #data | 01 00110 W 0000 0001<br>address<br>data data | 14 | SS 01 00110 W 0000 0001<br>0 segment offset<br>data data | 15 |
| | | | | SL 01 00110 W 0000 0001<br>1 segment 0000 0000<br>offset<br>data data | 17 |
| X: | CP addr(Rd), #data | 01 00110 W Rd ≠ 0 0001<br>address<br>data | 15 | SS 01 00110 W Rd ≠ 0 0001<br>0 segment offset<br>data | 15 |
| | | | | SL 01 00110 W Rd≠ 0 0001<br>1 segment 0000 0000<br>offset<br>data | 18 |
| | CPB addr(Rd), #data | 01 00110 W Rd≠0 0001<br>address<br>data data | 15 | SS 01 00110 W Rd≠0 0001<br>0 segment offset<br>data data | 15 |
| | | | | SL 01 00110 W Rd≠0 0001<br>1 segment 0000 0000<br>offset<br>data data | 18 |

**Example:** If register R5 contains %0400, the byte at location %0400 contains 2, and the source operand is the immediate value 3, the statement

CPB @R5,#3

will leave the C flag set, indicating a borrow, the S flag set, and the Z and V flags cleared.

Note 1. Word register in nonsegmented mode, register pair in segmented mode

# CPD
## Compare and Decrement

**CPD** dst, src, r, cc  
**CPDB**

dst: R  
src: IR

**Operation:**

dst − src  
AUTODECREMENT src (by 1 if byte, by 2 if word)  
r ← r − 1

This instruction is used to search a string of data for an element meeting the specified condition. The contents of the location addressed by the source register are compared to (subtracted from) the destination operand, and the Z flag is set if the condition code specified by "cc" would be set by the comparison; otherwise the Z flag is cleared. See Section 6.6 for a list of condition codes. Both operands are unaffected.

The source register is then decremented by one if CPDB, or by two if CPD, thus moving the pointer to the previous element in the string. The word register specified by "r" (used as a counter) is then decremented by one. The source, destination, and count registers must be separate and non-overlapping registers.

**Flags:**

**C:** Undefined  
**Z:** Set if the condition code generated by the comparison matches cc; cleared otherwise  
**S:** Undefined  
**V:** Set if the result of decrementing r is zero; cleared otherwise  
**D:** Unaffected  
**H:** Unaffected

| Source Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles | Instruction Format | Cycles |
| IR: | CPD Rd, @Rs[1], r, cc<br>CPDB Rbd, @Rs[1], r, cc | 1011101 \| W \| Rs ≠ 0 \| 1000<br>0000 \| r \| Rd \| cc | 20 | 1011101 \| W \| RRs≠0 \| 1000<br>0000 \| r \| Rd \| cc | 20 |

**Example:**

If register RH0 contains %FF, register R1 contains %4001, the byte at location %4001 contains %00, and register R3 contains 5, the instruction

```
CPDB  RH0, @R1, R3, EQ
```

will leave the Z flag cleared since the condition code would not have been "equal." Register R1 will contain the value %4000 and R3 will contain 4. For segmented mode, R1 must be replaced by a register pair.

Note 1: Word register in nonsegmented mode, register pair in segmented mode.

# CPDR
## Compare, Decrement and Repeat

**CPDR** dst, src, r, cc　　dst: R
**CPDRB**　　src: IR

**Operation:**

dst − src
AUTODECREMENT src (by 1 if byte; by 2 if word)
r ← r − 1
repeat until cc is true or R = 0

This instruction is used to search a string of data for an element meeting the specified condition. The contents of the location addressed by the source register are compared to (subtracted from) the destination operand, and the Z flag is set if the condition code specified by "cc" would be set by the comparison; otherwise the Z flag is cleared. See Section 6.6 for a list of condition codes. Both operands are unaffected.

The source register is then decremented by one if CPDRB, or by two if CPDR, thus moving the pointer to the previous element in the string. The word register specified by "r" (used as a counter) is decremented by one. The entire operation is repeated until either the condition is met or the result of decrementing r is zero. This instruction can search a string from 1 to 65536 bytes or 32768 words long (the value of r must not be greater than 32768 for CPDR). The source, destination, and count registers must be separate and non-overlapping registers.

This instruction can be interrupted after each execution of the basic operation. The program counter value of the start of this instruction is saved before the interrupt request is accepted, so that the instruction can be properly resumed. Seven more cycles should be added to this instruction's execution time for each interrupt request that is accepted.

**Flags:**

**C:** Undefined
**Z:** Set if the condition code generated by the comparison matches cc; cleared otherwise
**S:** Undefined
**V:** Set if the result of decrementing r is zero; cleared otherwise
**D:** Unaffected
**H:** Unaffected

| Source Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles$^2$ | Instruction Format | Cycles$^2$ |
| **IR:** | CPDR Rd, @Rs[1], r, cc<br>CPDRB Rbd, @Rs[1], r, cc | 1011101 W Rs ≠ 0 1100<br>0000 r Rd cc | 11 + 9n | 1011101 W RRs≠0 1100<br>0000 r Rd cc | 11 + 9n |

**Example:**

If the string of words starting at location %2000 contains the values 0, 2, 4, 6 and 8, register R2 contains %2008, R3 contains 5, and R8 contains 8, the instruction

```
CPDR  R3, @R2, R8, GT
```

will leave the Z flag set indicating the condition was met. Register R2 will contain the value %2002, R3 will still contain 5, and R8 will contain 2. For segmented mode, a register pair would be used instead of R2.

Note 1: Word register in nonsegmented mode, register pair in segmented mode.
Note 2: n = number of data elements compared.

# CPI
## Compare and Increment

**CPI** dst, src, r, cc  dst: IR
**CPIB**  src: IR

**Operation:**

dst − src
AUTOINCREMENT src (by 1 if byte; by 2 if word)
r ← r − 1

This instruction is used to search a string of data for an element meeting the specified condition. The contents of the location addressed by the source register are compared to (subtracted from) the destination operand and the Z flag is set if the condition code specified by "cc" would be set by the comparison; otherwise the Z flag is cleared. See Section 6.6 for a list of condition codes. Both operands are unaffected.

The source register is then incremented by one if CPIB, or by two if CPI, thus moving the pointer to the next element in the string. The word register specified by "r" (used as a counter) is then decremented by one. The source, destination, and counter registers must be separate and non-overlapping registers.

**Flags:**

**C:** Undefined
**Z:** Set if the condition code generated by the comparison matches cc; cleared otherwise
**S:** Undefined
**V:** Set if the result of decrementing r is zero; cleared otherwise
**D:** Unaffected
**H:** Unaffected

| Source Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles | Instruction Format | Cycles |
| **IR:** | CPI Rd, @Rs[1], r, cc<br>CPIB Rbd, @Rs[1], r, cc | 1011101 W Rs ≠ 0 0000<br>0000 r Rd cc | 20 | 1011101 W RRs≠0 0000<br>0000 r Rd cc | 20 |

**Example:** This instruction can be used in a "loop" of instructions that searches a string of data for an element meeting the specified condition, but an intermediate operation on each data element is required. The following sequence of instructions (to be executed in non-segmented mode) "scans while numeric," that is, a string is searched until either an ASCII character not in the range "0" to "9" (see Appendix C) is found, or the end of the string is reached. This involves a range check on each character (byte) in the string. For segmented mode, R1 must be changed to a register pair.

```
           LD         R3, #STRLEN             !initialize counter!
           LDA        R1, STRSTART            !load start address!
           LDB        RL0,#‘9’                !largest numeric char!
LOOP:
           CPB        @R1,#‘0’                !test char < ‘0’!
           JR         ULT,NONNUMERIC
           CPIB       RL0, @RI, R3, ULE       !test char ≤ 9!
           JR         NZ, NONNUMERIC
           JR         NOV, LOOP               !repeat until counter = 0!
DONE:
             .
             .
             .
NONNUMERIC:                                   !handle non-numeric char!
```

Note 1. Word register in nonsegmented mode, register pair in segmented mode

# CPIR
## Compare, Increment and Repeat

| | |
|---|---|
| **CPIR** dst, src, r, cc | dst: R |
| **CPIRB** | src: IR |

**Operation:**

```
dst − src
AUTOINCREMENT src (by 1 if byte; by 2 if word)
r ← r − 1
repeat until cc is true or R = 0
```

This instruction is used to search a string of data for an element meeting the specified condition. The contents of the location addressed by the source register are compared to (subtracted from) the destination operand, and the Z flag is set if the condition code specified by "cc" would be set by the comparison; otherwise the Z flag is cleared. See Section 6.6 for a list of condition codes. Both operands are unaffected.

The source register is then incremented by one if CPIRB, or by two if CPIR, thus moving the pointer to the next element in the string. The word register specified by "r" (used as a counter) is then decremented by one. The entire operation is repeated until either the condition is met or the result of decrementing r is zero. This instruction can search a string from 1 to 65536 bytes or 32768 words long (the value of r must not be greater than 32768 for CPIR). The source, destination, and counter registers must be separate and non-overlapping registers.

This instruction can be interrupted after each execution of the basic operation. The program counter value of the start of this instruction is saved before the interrupt request is accepted, so that the instruction can be properly resumed. Seven more cycles should be added to this instruction's execution time for each interrupt request that is accepted.

**Flags:**

**C:** Undefined
**Z:** Set if the condition code generated by the comparison matches cc; cleared otherwise
**S:** Undefined
**V:** Set if the result of decrementing r is zero; cleared otherwise
**D:** Unaffected
**H:** Unaffected

| Source Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles[2] | Instruction Format | Cycles[2] |
| **IR:** | CPIR Rd, @Rs[1], r, cc<br>CPIRB Rbd,@Rs[1], r, cc | 1011101 W Rs ≠ 0 0100<br>0000 r Rd cc | 11 + 9n | 1011101 W RRs 0100<br>0000 r Rd cc | 11 + 9n |

**Example:** The following sequence of instructions (to be executed in nonsegmented mode) can be used to search a string for an ASCII return character. The pointer to the start of the string is set, the string length is set, the character (byte) to be searched for is set, and then the search is accomplished. Testing the Z flag determines whether the character was found. For segmented mode, R1 must be changed to a register pair.

```
LDA      R1, STRSTART
LD       R3, #STRLEN
LDB      RL0, #% D           !hex code for return is D!
CPIRB    RL0, @R1, R3, EQ
JR       Z, FOUND
```

Note 1: Word register in nonsegmented mode, register pair in segmented mode.

Note 2: n = number of data elements compared.

# CPSD
## Compare String and Decrement

**CPSD** dst, src, r, cc　　dst: IR
**CPSDB**　　src: IR

**Operation:**

dst − src
AUTODECREMENT dst and src (by 1 if byte; by 2 if word)
r ← r − 1

This instruction can be used to compare two strings of data until the specified condition is true. The contents of the location addressed by the source register are compared to (subtracted from) the contents of the location addressed by the destination register. The Z flag is set if the condition code specified by "cc" would be set by the comparison; otherwise the Z flag is cleared. See Section 6.6 for a list of condition codes. Both operands are unaffected.
The source and destination registers are then decremented by one if CPSDB, or by two if CPSD, thus moving the pointers to the previous elements in the strings. The word register specified by "r" (used as a counter) is then decremented by one.
The source, destination, and count registers must be separate and non-overlapping registers.

**Flags:**

**C:** Cleared if there is a carry from the most significant bit of the result of the comparison; set otherwise, indicating a "borrow". Thus this flag will be set if the destination is less than the source when viewed as unsigned integers.
**Z:** Set if the condition code generated by the comparison matches cc; cleared otherwise
**S:** Set is the result of the comparison is negative; cleared otherwise
**V:** Set if the result of decrementing r is zero; cleared otherwise
**D:** Unaffected
**H:** Unaffected

| Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles | Instruction Format | Cycles |
| IR: | CPSD @Rd[1], @Rs[1], r, cc<br>CPSDB @Rd[1],@Rs[1],r,cc | 1011101 W Rs ≠ 0 1010<br>0000 r Rd ≠ 0 cc | 25 | 1011101 W RRs ≠0 1010<br>0000 r RRd ≠0 cc | 25 |

**Example:**

If register R2 contains %2000, the byte at location %2000 contains %FF, register R3 contains %3000, the byte at location %3000 contains %00, and register R4 contains 1, the instruction (executed in nonsegmented mode)

CPSDB @R2, @R3, R4, UGE

will leave the Z flag set to 1 since the condition code would have been "unsigned greater than or equal", and the V flag will be set to 1 to indicate that the counter R4 now contains 0. R2 will contain %1FFF, and R3 will contain %2FFF. For segmented mode, R2 and R3 must be changed to register pairs.

Note 1 Word register in nonsegmented mode, register pair in segmented mode.

# CPSDR
## Compare String, Decrement and Repeat

**CPSDR** dst, src,r, cc  dst: IR
**CPSDRB**  src: IR

**Operation:**

dst — src
AUTODECREMENT dst and src (by 1 if byte; by 2 if word)
r ← r − 1
repeat until cc is true or r = 0

This instruction is used to compare two strings of data until the specified condition is true. The contents of the location addressed by the source register are compared to (subtracted from) the contents of the location addressed by the destination register. The Z flag is set if the condition code specified by "cc" would be set by the comparison; otherwise the Z flag is cleared. See Section 6.6 for a list of condition codes. Both operands are unaffected.

The source and destination registers are then decremented by one if CPSDRB, or by two if CPSDR, thus moving the pointers to the previous elements in the strings. The word register specified by "r" (used as a counter) is then decremented by one. The entire operation is repeated until either the condition is met or the result of decrementing r is zero. This instruction can compare strings from 1 to 65536 bytes or from 1 to 32768 words long (the value of r must not be greater than 32768 for CPSDR). The source, destination, and count registers must be separate and non-overlapping registers.

This instruction can be interrupted after each execution of the basic operation. The program counter of the start of this instruction is saved before the interrupt request is accepted, so that the instruction can be properly resumed. Seven more cycles should be added to this instruction's execution time for each interrupt request that is accepted.

**Flags:**

**C:** Cleared if there is a carry from the most significant bit of the result of the comparison; set otherwise, indicating a "borrow". Thus this flag will be set if the destination is less than the source when viewed as unsigned integers
**Z:** Set if the conditon code generated by the comparison matches cc; cleared otherwise
**S:** Set if the result of the comparison is negative; cleared otherwise
**V:** Set if the result of decrementing r is zero; cleared otherwise
**D:** Unaffected
**H:** Unaffected

| Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles | Instruction Format | Cycles |
| IR: | CPSDR @Rd[1],@Rs[1],r,cc<br>CPSDRB @Rd[1],@Rs[1],r,cc | 1011101 W Rs≠0 1110<br>0000 r Rd≠0 cc | 11 + 14n | 1011101 W RRs≠0 1110<br>0000 r RRd≠0 cc | 11 + 14n |

**Example:**

If the words from location %1000 to %1006 contain the values 0, 2, 4, and 6, the words from location %2000 to %2006 contain the values 0, 1, 1, 0, register R13 contains %1006, register R14 contains %2006, and register R0 contains 4, the instruction (executed in nonsegmented mode)

```
CPSDR   @R13, @R14, R0, EQ
```

leaves the Z flag set to 1 since the condition code would have been "equal" (locations %1000 and %2000 both contain the value 0). The V flag will be set to 1 indicating R0 was decremented to 0. R13 will contain %0FFE, R14 will contain %IFFE, and R0 will contain 0. For segmented mode, R13 and R14 must be changed to register pairs.

Note 1: Word register in nonsegmented mode, register pair in segmented mode.

Note 2: n = number of data elements compared.

# CPSI
## Compare String and Increment

**CPSI** dst, src, r, cc    dst: IR
**CPSIB**    src: IR

**Operation:**

dst − src
AUTOINCREMENT dst and src (by 1 if byte, by 2 if word)
r ← r − 1

This instruction can be used to compare two strings of data until the specified condition is true. The contents of the location addressed by the source register are compared to (subtracted from) the contents of the location addressed by the destination register. The Z flag is set if the condition code specified by "cc" would be set by the comparison; otherwise the Z flag is cleared. See Section 6.6 for a list of condition codes. Both operands are unaffected.

The source and destination registers are then incremented by one if CPSIB, or by two if CPSI, thus moving the pointers to the next elements in the strings. The word register specified by "r" (used as a counter) is then decremented by one.
The source, destination, and count registers must be separate and non-overlapping registers.

**Flags:**

**C:** Cleared if there is a carry from the most significant bit of the result of the comparison; set otherwise, indicating a "borrow". Thus this flag will be set if the destination is less than the source when viewed as unsigned integers
**Z:** Set if the condition code generated by the comp arison matches cc; cleared otherwise
**S:** Set is the result of the comparison is negative; cleared otherwise
**V:** Set if the result of decrementing r is zero; cleared otherwise
**D:** Unaffected
**H:** Unaffected

| Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles | Instruction Format | Cycles |
| IR: | CPSI @Rd$^1$,@Rs$^1$,r,cc<br>CPSIB @Rd$^1$,@Rs$^1$,r,cc | 1011101 \| W \| Rs ≠ 0 \| 0010<br>0000 \| r \| Rd ≠ 0 \| cc | 25 | 1011101 \| W \| RRs≠0 \| 0010<br>0000 \| r \| RRd≠0 \| cc | 25 |

**Example:**

This instruction can be used in a "loop" of instructions which compares two strings until the specified condition is true, but where an intermediate operation on each data element is required. The following sequence of instructions, to be executed in nonsegmented mode, attempts to match a given source string to the destination string which is known to contain all upper-case characters. The match should succeed even if the source string contains some lower-case characters. This involves a forced conversion of the source string to upper-case (only ASCII alphabetic letters are assumed, see Appendix C) by resetting bit 5 of each character (byte) to 0 before comparison.

```
          LDA       R1, SRCSTART        !load start addresses!
          LDA       R2, DSTSTART
          LD        R3, #STRLEN         !initialize counter!
LOOP:
          RESB      @R1,#5              !force upper-case!
          CPSIB     @R1,@R2, R3, NE     !compare until not equal!
          JR        Z, NOTEQUAL         !exit loop if match fails!
          JR        NOV, LOOP           !repeat until counter = 0!
DONE:                   .               !match succeeds!
                        .
                        .
NOTEQUAL:               .               !match fails!
```

In segmented mode, R1 and R2 must both be register pairs.

Note 1 Word register in nonsegmented mode, register pair in segmented mode

# CPSIR

## Compare String, Increment and Repeat

**CPSIR** dst,src,r,cc dst: IR
**CPSIRB** src: IR

**Operation:**

```
dst - src
AUTOINCREMENT dst and src (by 1 if byte, by 2 if word)
r ← r - 1
repeat until cc is true or r = 0
```

This instruction is used to compare two strings of data until the specified condition is true. The contents of the location addressed by the source register are compared to (subtracted from) the contents of the location addressed by the destination register. The Z flag is set if the condition code specified by "cc" would be set by the comparison; otherwise the Z flag is cleared. See Section 6.6 for a list of condition codes. Both operands are unaffected. The source and destination registers are then incremented by one if CPSIRB, or by two if CPSIR, thus moving the pointers to the next elements in the strings. The word register specified by "r" (used as a counter) is then decremented by one. The entire operation is repeated until either the condition is met or the result of decrementing r is zero. This instruction can compare strings from 1 to 65536 bytes or from 1 to 32768 words long (the value of r must not be greater than 32768 for CPSIR). The source, destination, and counter registers must be separate and non-overlapping registers.

This instruction can be interrupted after each execution of the basic operation. The program counter value of the start of this instruction is saved before the interrupt request is accepted, so that the instruction can be properly resumed. Seven cycles should be added to this instruction's execution time for each interrupt request that is accepted.

**Flags:**

**C:** Cleared if there is a carry from the most significant bit of the result of the last comparison made; set otherwise, indicating a "borrow". Thus this flag will be set if the last destination element is less than the last source element when viewed as unsigned integers.
**Z:** Set if the condition code generated by the comparison matches cc; cleared otherwise
**S:** Set if the result of the last comparison made is negative; cleared otherwise
**V:** Set if the result of decrementing r is zero; cleared otherwise
**D:** Unaffected
**H:** Unaffected

| Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles[2] | Instruction Format | Cycles[2] |
| IR: | CPSIR @Rd[1],@Rs[1],r,cc<br>CPSIRB @Rd[1],@Rs[1],r,cc | 1011101 W Rs ≠ 0 0110<br>0000 r Rd ≠ 0 cc | 11 + 14n | 1011101 W RRs ≠ 0 0110<br>0000 r RRd ≠ 0 cc | 11 + 14n |

**Example:** The CPSIR instruction can be used to compare text strings for lexicographic order. (For most common character encodings—for example, ASCII and EBCDIC—lexicographic order is the same as alphabetic order for alphabetic text strings that do not contain blanks.)

Let S1 and S2 be text strings of lengths L1 and L2. According to lexicographic ordering, S1 is said to be "less than" or "before" S2 if either of the following is true:

- At the first character position at which S1 and S2 contain different characters, the character code for the S1 character is less than the character code for the S2 character.
- S1 is shorter than S2 and is equal, character for character, to an initial substring of S2.

For example, using the ASCII character code, the following strings are in ascending lexicographic order:

A<br>
A A<br>
A B C<br>
A B C D<br>
A B D

Let us assume that the address of S1 is in RR2, the address of S2 is in RR4, the lengths L1 and L2 of S1 and S2 are in R0 and R1, and the shorter of L1 and L2 is in R6. The the following sequence of instructions will determine whether S1 is less than S2 in lexicographic order:

```
     CPSIRB  @RR2, @RR4, R6, NE     !Scan to first unequal character!
                                    !The following flags settings are possible:
                                     Z = 0, V = 1: Strings are equal through L1
                                     character (Z = 0, V = 0 cannot occur).
                                     Z = 1, V = 0 or 1: A character position was
                                     found at which the strings are unequal.
                                     C = 1 (S = 0 or 1): The character in the RR2
                                     string was less (viewed as numbers from 0 to
                                     255, not as numbers from -128 to +127).
                                     C = 0 (S = 0 or 1): The character in the RR2
                                     string was not less!
     JR Z,CHAR_COMPARE              !If Z=1, compare the characters!
     CP R0,R1                       !Otherwise, compare string lengths!
     JR LT, S1_IS_LESS
     JR S1_NOT_LESS
CHAR_COMPARE:
     JR ULT, S1_IS_LESS             !ULT is another name for C=1!
S1_NOT LESS:
               •
               •
               •
S1_IS_LESS:
```

# DAB
## Decimal Adjust

**DAB** dst        dst: R

**Operation:** dst ← DA dst

The destination byte is adjusted to form two 4-bit BCD digits following a binary addition or subtraction operation on two BCD encoded bytes. For addition (ADDB, ADCB) or subtraction (SUBB, SBCB), the following table indicates the operation performed:

| Instruction | Carry Before DAB | Bits 4-7 Value (Hex) | H Flag Before DAB | Bits 0-3 Value (Hex) | Number Added To Byte | Carry After DAB |
|---|---|---|---|---|---|---|
| | 0 | 0-9 | 0 | 0-9 | 00 | 0 |
| | 0 | 0-8 | 0 | A-F | 06 | 0 |
| ADDB | 0 | 0-9 | 1 | 0-3 | 06 | 0 |
| ADCB | 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 |
| | 1 | 0-3 | 1 | 0-3 | 66 | 1 |
| SUBB | 0 | 0-9 | 0 | 0-9 | 00 | 0 |
| SBCB | 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 binary addition or subtraction of BCD digits.

**Flags:**

**C:** Set or cleared according to the table above
**Z:** Set if the result is zero; cleared otherwise
**S:** Set if the most significant bit of the result is set; cleared otherwise
**V:** Unaffected
**D:** Unaffected
**H:** Unaffected

| Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles | Instruction Format | Cycles |
| **R:** | DAB Rbd | 10 110000 Rbd 0000 | 5 | 10 110000 Rbd 0000 | 5 |

**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 DAB instruction adjusts this result so that the correct BCD representation is obtained.

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

# DEC
## Decrement

**DEC** dst, src  dst: R, IR, DA, X
**DECB**  src: IM

**Operation:** dst ◄– dst – src (where src = 1 to 16)

The source operand (a value from 1 to 16) is subtracted from the destination operand and the result is stored in the destination. Subtraction is performed by adding the two's complement of the source operand to the destination operand. The source operand may be omitted from the assembly language statement and defaults to the value 1.

The value of the source field in the instruction is one less than the actual value of the source operand. Thus, the coding in the instruction for the source ranges from 0 to 15, which corresponds to the source values 1 to 16.

**Flags:**

**C:** Unaffected
**Z:** Set if the result is zero; cleared otherwise
**S:** Set if the result is negative; cleared otherwise
**V:** Set if arithmetic overflow occurs, that is, if the operands were of opposite signs, and the sign of the result is the same as the sign of the source; cleared otherwise
**D:** Unaffected
**H:** Unaffected

| Destination Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | | |
|---|---|---|---|---|---|---|
| | | Instruction Format | Cycles | | Instruction Format | Cycles |
| **R:** | DEC Rd, #n<br>DECB Rbd, #n | 10 10101 W Rd n – 1 | 4 | | 10 10101 W Rd n – 1 | 4 |
| **IR:** | DEC @Rd[1], #n<br>DECB @Rd[1], #n | 00 10101 W Rd≠0 n – 1 | 11 | | 00 10101 W RRd≠0 n – 1 | 11 |
| **DA:** | DEC address, #n<br>DECB address, #n | 01 10101 W 0000 n – 1<br>address | 13 | SS | 01 10101 W 0000 n – 1<br>0 segment offset | 14 |
| | | | | SL | 01 10101 W 0000 n – 1<br>1 segment 0000 0000<br>offset | 16 |
| **X:** | DEC addr(Rd), #n<br>DECB addr(Rd), #n | 01 10101 W Rd≠0 n – 1<br>address | 14 | SS | 01 10101 W Rd≠0 n – 1<br>0 segment offset | 14 |
| | | | | SL | 01 10101 W Rd≠0 n – 1<br>1 segment 0000 0000<br>offset | 17 |

**Example:** If register R10 contains %002A, the statement

DEC R10

will leave the value %0029 in R10.

Note 1· Word register in nonsegmented mode, register pair in segmented mode.

# DI
## Disable Interrupt

## Privileged Instruction

**DI** Int                Int: VI, NVI

**Operation:**

If instruction (0) = 0 then NVI ← 0
If instruction (1) = 0 then VI ← 0

Any combination of the Vectored Interrupt (VI) or Non-Vectored Interrupt (NVI) control bits in the Flags and Control Word (FCW) are cleared to zero if the corresponding bit in the instruction is zero, thus disabling the appropriate type of interrupt. If the corresponding bit in the instruction is one, the control bit will not be affected. All other bits in the FCW are not affected. There may be one or two operands in the assembly language statement, in either order.

**Flags:** No flags affected.

| | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles | Instruction Format | Cycles |
| | DI ınt | 01111100 \| 000000 \| VI \| NVI | 7 | 01111100 \| 000000 \| VI \| NVI | 7 |

**Example:** If the NVI and VI control bits are set (1) ın the FCW, the instruction:

DI VI

will leave the NVI control bit in the FCW set (1) and will leave the VI control bit in the FCW cleared (0).

# DIV
## Divide

| | |
|---|---|
| **DIV** dst, src | dst: R |
| **DIVL** | src: R, IM, IR, DA, X |

**Operation:**

Word: (dst is register pair, src is word):
dst (0:31) is divided by src (0:15)
(dst (0:31) = quotient × src (0:15) + remainder)
dst (0:15) ← quotient
dst (16:31) ← remainder

Long: (dst register quadruple, src is long word or register pair):
dst (0:63) is divided by src (0:31)
(dst (0:63) = quotient × src (0:31) + remainder)
dst (0:31) ← quotient
dst (32:63) ← remainder

The destination operand (dividend) is divided by the source operand (divisor), the quotient is stored in the low-order half of the destination and the remainder is stored in the high-order half of the destination. The contents of the source are not affected. Both operands are treated as signed, two's complement integers and division is performed so that the remainder is of the same sign as the dividend. For DIV, the destination is a register pair and the source is a word value; for DIVL, the destination is a register quadruple and the source is a long word value.

There a four possible outcomes of the Divide instruction, depending on the division, and the resulting quotient:

CASE 1. If the quotient is within the range $-2^{15}$ to $2^{15} - 1$ inclusive for DIV or $-2^{31}$ to $2^{31} - 1$ inclusive for DIVL, then the quotient and remainder are left in the destination register as defined above, the overflow and carry flags are cleared to zero, and the sign and zero flags are set according to the value of the quotient.

CASE 2. If the divisor is zero, the destination register remains unchanged, the overflow and zero flags are set to one and the carry and sign flags are cleared to zero.

CASE 3. If the quotient is outside the range $-2^{16}$ to $2^{16} - 1$ inclusive for DIV or $-2^{32}$ to $2^{32} - 1$ inclusive for DIVL, the destination register contains an undefined value, the overflow flag is set to one, the carry and zero flags are cleared to zero, and the sign flag is undefined.

CASE 4. If the quotient is inside the range of case 3 but outside the range of case 1, then all but the sign bit of the quotient and all of the remainder are left in the destination register, the overflow and carry flags are set to one, and the sign and zero flags are set according to the value of the quotient. In this case, the sign flag can be replicated by subsequent instruction into the high-order half of the destination to produce the two's complement representation of the quotient in the same precision as the original dividend.

**Flags:**

**C:** Set if V is set and the quotient lies in the range from $-2^{16}$ to $2^{16} - 1$ inclusive for DIV or in the range from $-2^{32}$ to $2^{32} - 1$ inclusive for DIVL; cleared otherwise
**Z:** Set if the quotient or divisor is zero; cleared otherwise
**S:** Undefined if V is set and C is clear (overflow); otherwise set if the quotient is negative, cleared if the quotient is non-negative.
**V:** Set if the divisor is zero or if the computed quotient lies outside the range from $-2^{15}$ to $2^{15} - 1$ inclusive for DIV or outside range from $-2^{31}$ to $2^{31} - 1$ inclusive for DIVL; cleared otherwise
**D:** Unaffected
**H:** Unaffected

| Source Addressing Mode | Assembler Language Syntax | Nonsegmented Mode |  | Segmented Mode |  |
|---|---|---|---|---|---|
|  |  | Instruction Format | Cycles[2] | Instruction Format | Cycles[2] |
| R: | DIV RRd, Rs | 10 011011 Rs RRd | 107 | 10 011011 Rs RRd | 107 |
|  | DIVL RQd, RRs | 10 011010 RRs RQd | 744 | 10 011010 RRs RQd | 744 |
| IM: | DIV RRd, #data | 00 011011 0000 RRd<br>data | 107 | 00 011011 0000 RRd<br>data | 107 |
|  | DIVL RQd, #data | 00 011010 0000 RQd<br>31 data (high) 16<br>15 data (low) 0 | 744 | 00 011010 0000 RQd<br>31 data (high) 16<br>15 data (low) 0 | 744 |
| IR: | DIV RRd, @Rs[1] | 00 011011 Rs≠0 RRd | 107 | 00 011011 RRs≠0 RRd | 107 |
|  | DIVL RQd, @Rs[1] | 00 011010 Rs≠0 RQd |  | 00 011010 RRs≠0 RQd |  |
| DA: | DIV RRd, address | 01 011011 0000 RRd<br>address | 108 | SS 01 011011 0000 RRd<br>0 segment offset | 109 |
|  |  |  |  | SL 01 011011 0000 RRd<br>1 segment 0000 0000<br>offset | 111 |
|  | DIVL RQD, address | 01 011010 0000 RQd<br>address | 445 | SS 01 011010 0000 RQd<br>0 segment offset | 746 |
|  |  |  |  | SL 01 011010 0000 RQd<br>1 segment 0000 0000<br>offset | 748 |
| X: | DIV RRd, addr(Rs) | 01 011011 Rs≠0 RRd<br>address | 109 | SS 01 011011 Rs≠0 RRd<br>0 segment offset | 109 |
|  |  |  |  | SL 01 011011 Rs≠0 RRd<br>1 segment 0000 0000<br>offset | 112 |
|  | DIVL RQd, addr(Rs) | 01 011010 Rs≠0 RQd<br>address | 746 | SS 01 011010 Rs≠0 RQd<br>0 segment offset | 746 |
|  |  |  |  | SL 01 011010 Rs≠0 RQd<br>1 segment 0000 0000<br>offset | 749 |

**Example:** If register RR0 (composed of word register R0 and R1) contains %00000022 and register R3 contains 6, the statement

```
DIV   RR0,R3
```

will leave the value %00040005 in RR0 (R1 contains the quotient 5 and R0 contains the remainder 4).

Note 1: Word register in nonsegmented mode, register pair in segmented mode.

Note 2: The execution time for the instruction will be lower by 94 cycles for word, 714 for long word than indicated for divide by zero and by 82 for word, 693 for long word for overflow conditions.

# DJNZ

## Decrement and Jump if Not Zero

**DJNZ** R, dst
**DBJNZ** dst: RA

**Operation:**

R ← R − 1
If R ≠ 0 then PC ← PC − (2 × displacement)

The register being used as a counter is decremented. If the contents of the register are not zero after decrementing, the destination address is calculated and then loaded into the program counter (PC). Control will then pass to the instruction whose address is pointed to by the PC. When the register counter reaches zero, control falls through to the instruction following DJNZ or DBJNZ. This instruction provides a simple method of loop control.

The relative addressing mode is calculated by doubling the displacement in the instruction, then subtracting this value from the updated value of the PC to derive the destination address. The updated PC value is taken to be the address of the instruction following the DJNZ or DBJNZ instruction, while the displacement is a 7-bit positive value in the range 0 to 127. Thus, the destination address must be in the range -252 to 2 bytes from the start of the DJNZ or DBJNZ instruction. In the segmented mode, the PC segment number is not affected. The assembler automatically calculates the displacement by subtracting the PC value of the following instruction from the address given by the programmer and dividing the result by 2. Note that DJNZ or DBJNZ cannot be used to transfer control in the forward direction, nor to another segment in segmented mode operation.

**Flags:** No flags affected

| Destination Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles | Instruction Format | Cycles |
| RA: | DJNZ R, address<br>DBJNZ Rb, address | 1111 r W disp | 11 | 1111 r W disp | 11 |

**Example:**

DJNZ and DBJNZ are typically used to control a "loop" of instructions. In this example for nonsegmented mode, 100 bytes are moved from one buffer area to another, and the sign bit of each byte is cleared to zero. Register RH0 is used as the counter.

```
        LDB      RH0,#100         !initalize counter!
        LDA      R1, SRCBUF       !load start address!
        LDA      R2, DSTBUF
LOOP:
        LDB      RL0,@R1          !load source byte!
        RESB     RL0,#7           !mask off sign bit!
        LDB      @R2, RL0         !store into destination!
        INC      R1               !advance pointers!
        INC      R2
        DBJNZ    RH0, LOOP        !repeat until counter = 0!
 NEXT:
```

For segmented mode, R1 and R2 must be changed to register pairs.

## Privileged Instruction

# EI
## Enable Interrupts

**EI** int Int: VI, NVI

**Operation:**

If instruction (0) = 0 then NVI ← 1
If instruction (1) = 0 then VI ← 1

Any combination of the Vectored Interrupt (VI) or Non-Vetored Interrupt (NVI) control bits in the Flags and Control Word (FCW) are set to one if the corresponding bit in the instruction is zero, thus enabling the appropriate type of interrupt. If the corresponding bit in the instruction is one, the control bit will not be affected. All other bits in the FCW are not affected. There may be one or two operands in the assembly language statement, in either order.

**Flags:** No flags affected

| | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles | Instruction Format | Cycles |
| | EI int | 01111100 000001 VI NVI | 7 | 01111100 000001 VI NVI | 7 |

**Example:**

If the NVI contol bit is set (1) in the FCW, and the VI control bit is clear (0), the instruction

EI VI

will leave both the NVI and VI control bits in the FCW set (1)

# EX
## Exchange

**EX** dst, src  dst: R
**EXB**  src: R, IR, DA, X

**Operation:**

tmp ← src (tmp is a temporary internal register)
src ← dst
dst ← tmp

The contents of the source operand are exchanged with the contents of the destination operand.

**Flags:** No flags affected

| Source Addressing Mode | Assembler Language Syntax | Nonsegmented Mode |  | Segmented Mode |  |
|---|---|---|---|---|---|
|  |  | Instruction Format | Cycles | Instruction Format | Cycles |
| R: | EX Rd, Rs<br>EXB Rbd, Rbs | 10 10110 W Rs Rd | 6 | 10 10110 W Rs Rd | 6 |
| IR: | EX Rd, @Rs[1]<br>EXB Rbd, @Rs[1] | 00 10110 W Rs≠0 Rd | 12 | 00 10110 W Rs≠0 Rd | 12 |
| DA: | EX Rd, address<br>EXB Rbd, address | 01 10110 W 0000 Rd<br>address | 15 | SS<br>01 10110 W 0000 Rd<br>0 segment offset | 16 |
|  |  |  |  | SL<br>01 10110 W 0000 Rd<br>1 segment 0000 0000<br>offset | 18 |
| X: | EX Rd, addr(Rs)<br>EXB Rbd, addr(Rs) | 01 10110 W Rs≠0 Rd<br>address | 16 | SS<br>01 10110 W Rs≠0 Rd<br>0 segment offset | 16 |
|  |  |  |  | SL<br>01 10110 W Rs≠0 Rd<br>1 segment 0000 0000<br>offset | 19 |

**Example:**

If register R0 contains 8 and register R5 contains 9, the statement

```
EX   R0,R5
```

will leave the values 9 in R0 and 8 in R5. The flags will be left unchanged.

[1] Note 1: Word register in nonsegmented mode, register pair in segmented mode.

**EXTSB** dst dst: R
**EXTS**
**EXTSL**

**Operation:**

```
Byte
if dst (7) = 0   then dst (8:15) ← 000...000
                 else dst (8:15) ← 111...111
Word
if dst (15) = 0  then dst (16:31) ← 000...000
                 else dst (16:31) ← 111...111
Long
if dst (31) = 0  then dst (32:63) ← 000...000
                 else dst (32:63) ← 111...111
```

The sign bit of the low-order half of the destination operand is copied into all bit positions of the high-order half of the destination. For EXTSB, the destination is a word; for EXTS, the destination is a register pair; for EXTSL, the destination is a register quadruple.

This instruction is useful in multiple precision arithmetic or for conversion of small signed operands to larger signed operands (as, for example, before a divide).

**Flags:** No flags affected

| Destination Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles | Instruction Format | Cycles |
| **R:** | EXTSB Rd | 10 \| 110001 \| Rd \| 0000 | 11 | 10 \| 110001 \| Rd \| 0000 | 11 |
| | EXTS RRD | 10 \| 110001 \| RRd \| 1010 | 11 | 10 \| 110001 \| RRd \| 1010 | 11 |
| | EXTSL RQd | 10 \| 110001 \| RQd \| 0111 | 11 | 10 \| 110001 \| RQd \| 0111 | 11 |

**Example:** If register pair RR2 (composed of word registers R2 and R3) contains %12345678, the statement

EXTS RR2

will leave the value %00005678 in RR2 (because the sign bit of R3 was 0).

# HALT
## Halt

## Privileged Instruction

**HALT**

**Operation:** The CPU operation is suspended until an interrupt or reset request is received. This instruction is used to synchronize the Z8000 with external events, preserving its state until an interrupt or reset request is honored. After an interrupt is serviced, the instruction following HALT is executed. While halted, memory refresh cycles will still occur, and $\overline{\text{BUSREQ}}$ will be honored.

**Flags:** No flags affected

| | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles[1] | Instruction Format | Cycles[1] |
| | HALT | 01111010 \| 00000000 | 8+3n | 01111010 \| 00000000 | 8+3n |

Note 1 Interrupts are recognized at the end of each 3-cycle period, thus n = number of periods without interruption

## Privileged Instruction

# IN (SIN)

## (Special) Input

| | |
|---|---|
| **IN** dst, src<br>**INB** | dst: R<br>src: IR, DA |
| **SIN** dst, src<br>**SINB** | dst: R<br>src: DA |

**Operation** dst ← src

The contents of the source operand, an Input or Special Input port, are loaded into the destination register. IN and INB are used for Standard I/O operation; SIN and SINB are used for Special I/O operation.

**Flags:** No flags affected

| Source Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format[1] | Cycles | Instruction Format[1] | Cycles |
| **IR:** | IN Rd, @Rs<br>INB Rbd, @Rs | 0 0 \| 1 1 1 1 0 \| W \| Rs≠0 \| Rd | 10 | 0 0 \| 1 1 1 1 0 \| W \| Rs≠0 \| Rd | 10 |
| **DA:** | IN Rd, port<br>INB Rbd, port<br>SIN Rd, port<br>SINB Rbd, port | 0 0 \| 1 1 1 0 1 \| W \| Rd \| 0 1 0 S<br>port | 12 | 0 0 \| 1 1 1 0 1 \| W \| Rd \| 0 1 0 S<br>port | 12 |

**Example:** If register R6 contains the I/O port address %0123 and the port %0123 contains %FF, the statement

```
INB   RH2, @R6
```

will leave the value %FF in register RH2.

Note 1. For SIN, S = 1; otherwise S = 0

# INC
## Increment

**INC** dst, src　　dst: R, IR, DA, X
**INCB**　　src: IM

**Operation:** dst ← dst + src (src = 1 to 16)

The source operand (a value from 1 to 16) is added to the destination operand and the sum is stored in the destination. Two's complement addition is performed. The source operand may be omitted from the assembly language statement and defaults to the value 1.

The value of the source field in the instruction is one less than the actual value of the source operand. Thus, the coding in the instruction for the source ranges from 0 to 15, which corresponds to the source values 1 to 16.

**Flags:**

**C:** Unaffected
**Z:** Set if the result is zero; cleared otherwise
**S:** Set if the result is negative; cleared otherwise
**V:** Set if arithmetic overflow occurs, that is, if both operands were of the same sign and the result is of the opposite sign; cleared otherwise
**D:** Unaffected
**H:** Unaffected

| Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles | Instruction Format | Cycles |
| R: | INC Rd, #n<br>INCB Rbd, #n | 10 10100 W Rd n − 1 | 4 | 10 10100 W Rd n − 1 | 4 |
| IR: | INC @Rd[1], #n<br>INCB @Rd[1], #n | 00 10100 W Rd≠0 n − 1 | 11 | 00 10100 W RRd≠0 n − 1 | 11 |
| DA: | INC address, #n<br>INCB address, #n | 01 10100 W 0000 n − 1<br>address | 13 | SS 01 10100 W 0000 n − 1<br>0 segment offset | 14 |
| | | | | SL 01 10100 W 0000 n − 1<br>1 segment 0000 0000<br>offset | 16 |
| X: | INC addr(Rd), #n<br>INCB addr(Rd), #n | 01 10100 W Rd≠0 n − 1<br>address | 14 | SS 01 10100 W Rd≠0 n − 1<br>0 segment offset | 14 |
| | | | | SL 01 10100 W Rd≠0 n − 1<br>1 segment 0000 0000<br>offset | 17 |

**Example:** If register RH2 contains %21, the statement

INCB RH2,#6

will leave the value %27 in RH2.

[1] Note 1: Word register in nonsegmented mode, register pair in segmented mode.

## Privileged Instruction

# IND (SIND)

## (Special) Input and Decrement

| | |
|---|---|
| **IND** dst, src, r | dst: IR |
| **INDB** | src: IR |
| **SIND** | |
| **SINDB** | |

**Operation:**

dst ◄– src
AUTODECREMENT dst (by 1 byte, by 2 if word)
r ◄– r – 1

This instruction is used for block input of strings of data. IND and INDB are used for normal I/O operation; SIND and SINDB are used for special I/O operation. The contents of the I/O port addressed by the source word register are loaded into the memory location addressed by the destination register. I/O port addresses are 16 bits. The destination register is then decremented by one if a byte instruction or by two if a word instruction, thus moving the pointer to the previous element of the string in memory. The word register specified by "r" (used as a counter) is then decremented by one. The address of the I/O port in the source register is unchanged.

**Flags:**

**C:** Unaffected
**Z:** Unaffected
**S:** Unaffected
**V:** Set if the result of decrementing r is zero; cleared otherwise
**D:** Unaffected
**H:** Unaffected

| Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles | Instruction Format | Cycles |
| IR: | IND @Rd[1], @Rs, r<br>INDB @Rd[1], @Rs, r<br>SIND @Rd[1], @Rs, r<br>SINDB @Rd[1], @Rs, r | 0011101 \| W \| Rs ≠ 0 \| 100S<br>0000 \| r \| Rd ≠ 0 \| 1000 | 21 | 0011101 \| W \| Rs ≠ 0 \| 100S<br>0000 \| r \| Rd ≠ 0 \| 1000 | 21 |

**Example:**

In segmented mode, if register RR4 contains %02004000 (segment 2, offset %4000), register R6 contains the I/O port address %0228, the port %0228 contains %05B9, and register R0 contains %0016, the instruction

```
IND   @RR4, @R6, R0
```

will leave the value %05B9 in location %02004000, the value %02003FFE in RR4, and the value %0015 in R0. The V flag will be cleared. Register R6 still contains the value %0228. In nonsegmented mode, a word register would be used instead of RR4.

[1] Word register in nonsegmented mode, register pair in segmented mode.

# INDR (SINDR)

## Privileged Instruction

## (Special) Input, Decrement and Repeat

| | |
|---|---|
| **INDR** dst, src, r | dst: IR |
| **INDRB** | src: IR |
| **SINDR** | |
| **SINDRB** | |

**Operation:**

```
dst ← src
AUTODECREMENT dst (by 1 if byte, by 2 if word)
r ← r − 1
repeat until r = 0
```

This instruction is used for block input of strings of data. INDR and INDRB are used for normal I/O operation; SINDR and SINDRB are used for special I/O operation. The contents of the I/O port addressed by the source word register are loaded into the memory location addressed by the destination register. I/O port addresses are 16 bits. The destination register is then decremented by one if a byte instruction, or by two if a word instruction, thus moving the pointer to the previous element of the string in memory. The word register specified by "r" (used as a counter) is then decremented by one. The address of the I/O port in the source register is unchanged. The entire operation is repeated until the result of decrementing r is zero. This instruction can input from 1 to 65536 bytes or 32768 words (the value for r must not be greater than 32768 for INDR or SINDR).

This instruction can be interrupted after each execution of the basic operation. The program counter value of the start of this instruction is saved before the interrupt request is accepted, so that the instruction can be properly resumed. Seven more cycles should be added to this instruction's execution time for each interrupt request that is accepted.

**Flags:**

**C:** Unaffected
**Z:** Undefined
**S:** Unaffected
**V:** Set
**D:** Unaffected
**H:** Unaffected

| Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles[2] | Instruction Format | Cycles[2] |
| IR: | INDR @Rd[1], @Rs, r<br>INDRB @Rd[1], @Rs, r<br>SINDR @Rd[1], @Rs, r<br>SINDRB @Rd[1], @Rs, r | 0011101 W Rs ≠ 0 100S<br>0000 r Rd ≠ 0 0000 | 11 + 10n | 0011101 W Rs ≠ 0 100S<br>0000 r Rd ≠ 0 0000 | 11 + 10n |

**Example:** If register R1 contains %202A, register R2 contains the Special I/O address %0AFC, and register R3 contains 8, the instruction

```
SINDRB  @R1, @R2, R3
```

will input 8 bytes from the Special I/O port 0AFC and leave them in descending order from %202A to %2023. Register R1 will contain %2022, and R3 will contain 0. R2 will not be affected. The V flag will be set. This example assumes nonsegmented mode; in segmented mode, R1 would be replaced by a register pair.

Note 1: Word register in nonsegmented mode, register pair in segmented mode
Note 2: n = number of data elements transferred
Note 3· For SINDR, S = 1, otherwise S = 0.

# INI (SINI)

## Privileged Instruction

## (Special) Input and Increment

**INI** dst, src, r<br>
**INIB**<br>
**SINI**<br>
**SINIB**

dst: IR<br>
src: IR

**Operation:**

dst ← src<br>
AUTOINCREMENT dst (by 1 if byte, by 2 if word)<br>
r ← r − 1

This instruction is used for block input of strings of data. INI, INIB are used for normal I/O operation; SINI, SINIB are used for special I/O operation. The contents of the I/O port addressed by the source word register are loaded into the memory location addressed by the destination register. I/O port addresses are 16 bits. The destination register is then incremented by one if a byte instruction, or by two if a word instruction, thus moving the pointer to the next element of the string in memory. The word register specified by "r" (used as a counter) is then decremented by one. The address of the I/O port in the source register is unchanged.

**Flags:**

**C:** Unaffected<br>
**Z:** Undefined<br>
**S:** Unaffected<br>
**V:** Set if the result of decrementing r is zero; cleared otherwise<br>
**D:** Unaffected<br>
**H:** Unaffected

| Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles | Instruction Format | Cycles |
| IR: | INI @Rd[1], @Rs, r<br>INIB @Rd[1], @Rs, r<br>SINI @Rd[1], @Rs, r<br>SINIB @Rd[1], @Rs, r | 0011101 \| W \| Rs ≠ 0 \| 000S<br>0000 \| r \| Rd ≠ 0 \| 1000 | 21 | 0011101 \| W \| Rs ≠ 0 \| 000S<br>0000 \| r \| Rd ≠ 0 \| 1000 | 21 |

**Example:**

In nonsegmented mode, if register R4 contains %4000, register R6 contains the I/O port address %0229, the port %0229 contains %B9, and register R0 contains %0016, the instruction

```
INIB  @R4, @R6, R0
```

will leave the value %B9 in location %4000, the value %4001 in R4, and the value %0015 in R0. Register R6 still contains the value %0229. The V flag is cleared. In segmented mode, R4 would be replaced by a register pair.

[1] Note 1: Word register in nonsegmented mode, register pair in segmented mode.

# INIR (SINIR)

## (Special) Input, Increment and Repeat

| | |
|---|---|
| **INIR** dst, src, r<br>**INIRB**<br>**SINIR**<br>**SINIRB** | dst: IR<br>src: IR |

**Operation:**

```
dst ◄- src
AUTOINCREMENT dst (by 1 if byte, by 2 if word)
r ◄- r - 1
repeat until r = 0
```

This instruction is used for block input of strings of data. INIR and INIRB are used for Standard I/O operation; SINIR and SINIRB are used for Special I/O operation. The contents of the I/O port addressed by the source word register are loaded into the memory location addressed by the destination register. I/O port addresses are 16 bits. The destination register is then incremented by one if a byte instruction, or by two if a word instruction, thus moving the pointer to the next element in the string. The word register specified by "r" (used as a counter) is then decremented by one. The address of the I/O port in the source register is unchanged. The entire operation is repeated until the result of decrementing r is zero. This instruction can input from 1 to 65536 bytes or 32768 words (the value for r must not be greater than 32768 for INIR or SINIR).

This instruction can be interrupted after each execution of the basic operation. The program counter value of the start of this instruction is saved before the interrupt request is accepted, so that the instruction can be properly resumed. Seven more cycles should be added to this instruction's execution time for each interrupt request that is accepted. The source, destination, and count registers must be separate and non-overlapping registers.

**Flags:**

**C:** Unaffected
**Z:** Undefined
**S:** Unaffected
**V:** Set
**D:** Unaffected
**H:** Unaffected

| Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format[3] | Cycles[2] | Instruction Format[3] | Cycles[2] |
| **IR:** | INIR @Rd[1], @Rs, r<br>INIRB @Rd[1], @Rs, r<br>SINIR @Rd[1], @Rs, r<br>SINIRB @Rd[1], @Rs, r | 0011101 W Rs ≠ 0 000S<br>0000 r Rd ≠ 0 0000 | 11 + 10n | 0011101 W Rs ≠ 0 000S<br>0000 r RRd≠0 0000 | 11 + 10n |

**Example:** In nonsegmented mode, if register R1 contains %2023, register R2 contains the I/O port address %0551, and register R3 contains 8, the statement

```
INIRB   @R1, @R2, R3
```

will input 8 bytes from port %0551 and leave them in ascending order from %2023 to %202A. Register R1 will contain %202B, and R3 will contain 0. R2 will not be affected. The V flag will be set. In segmented mode, a register pair must be used instead of R1.

Note 1 Word register in nonsegmented mode, register pair in segmented mode

Note 2 n = number of data elements transferred

Note 3· For SINIR, S = 1, otherwise S = 0

## Privileged Instruction

# IRET
## Interrupt Return

**IRET**

**Operation:**

| Nonsegmented | Segmented |
|---|---|
| SP ← SP + 2 (Pop "identifier") | SP ← SP + 2 (Pop "identifier") |
| PS ← @SP | PS ← @SP |
| SP ← SP + 4 | SP ← SP + 6 |

This instruction is used to return to a previously executed procedure at the end of a procedure entered by an interrupt or trap (including a System Call instruction). First, the "identifier" word associated with the interrupt or trap is popped from the system stack and discarded. Then the contents of the location addressed by the system stack pointer are popped into the program status (PS), loading the Flags and Control Word (FCW) and the program counter (PC). The new value of the FCW is not effective until the next instruction, so that the status pins will not be affected by the new control bits until after the IRET instruction execution is completed. The next instruction executed is that addressed by the new contents of the PC. The system stack pointer (R15 if nonsegmented, or RR14 if segmented) is used to access memory. When using a Z8001 or Z8003, the operation of IRET in nonsegmented mode is undefined. A Z8001/3 must be in segmented mode when an IRET instruction is performed.

**Flags:**

**C:** Loaded from system stack
**Z:** Loaded from system stack
**S:** Loaded from system stack
**P/V:** Loaded from system stack
**D:** Loaded from system stack
**H:** Loaded from system stack

| Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles | Instruction Format | Cycles |
| | IRET | 01111011 \| 00000000 | 13 | 01111011 \| 00000000 | 16 |

**Example:**

In the nonsegmented Z8002 version, if the program counter contains %2550, the system stack pointer (R15) contains %3000, and locations %3000, %3002 and %3004 contain %7F03, a saved FCW value, and %1004, respectively, the instruction

```
IRET
```

will leave the value %3006 in the system stack pointer and the program counter will contain %1004, the address of the next instruction to be executed. The program status will be determined by the saved FCW value.

# JP
## Jump

**JP** cc, dst        dst: IR, DA, X

**Operation:** If cc is satisfied, then PC ← dst

A conditional jump transfers program control to the destination address if the condition specified by "cc" is satisfied by the flags in the FCW. See section 6.6 for a list of condition codes. If the condition is satisfied, the program counter (PC) is loaded with the designated address; otherwise, the instruction following the JP instruction is executed.

**Flags:** No flags affected

| Destination Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | **Instruction Format** | **Cycles[2]** | **Instruction Format** | **Cycles[2]** |
| **IR:** | JP cc, @Rd[1] | 00 011110 Rd≠0 cc | 10/7 | 00 011110 RRd≠0 cc | 15/7 |
| **DA:** | JP cc, address | 01 011110 0000 cc<br>address | 7/7 | SS 01 011110 0000 cc<br>0 segment offset | 8/8 |
| | | | | SL 01 011110 0000 cc<br>1 segment 0000 0000<br>offset | 10/10 |
| **X:** | JP cc, addr(Rd) | 01 011110 Rd≠0 cc<br>address | 8/8 | SS 01 011110 Rd≠0 cc<br>0 segment offset | 8/8 |
| | | | | SL 01 011110 Rd≠0 cc<br>1 segment 0000 0000<br>offset | 11/11 |

**Example:** If the carry flag is set, the statement

JP C, %1520

replaces the contents of the program counter with %1520, thus transferring control to that location.

[1] Note 1 Word register in nonsegmented mode, register pair in segmented mode

[2] Note 2. The two values correspond to jump taken and jump not taken

# JR
## Jump Relative

**JR** cc, dst dst: RA

**Operation:** if cc is satisfied then PC ← PC + (2 × displacement)

A conditional jump transfers program control to the destination address if the condition specified by "cc" is satisfied by the flags in the FCW. See Section 6.6 for a list of condition codes. If the condition is satisfied, the program counter (PC) is loaded with the designated address; otherwise, the instruction following the JR instruction is executed. The destination address is calculated by doubling the displacement in the instruction, then adding this value to the updated value of the PC. The updated PC value is taken to be the address of the instruction following the JR instruction, while the displacement is an 8-bit signed value in the range −128 to +127. Thus, the destination address must be in the range −254 to +256 bytes from the start of the JR instruction. In the segmented mode, the PC segment number is not affected.

The assembler automatically calculates the displacement by subtracting the PC value of the following instruction from the address given by the programmer.

**Flags:** No flags affected

| Destination Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles | Instruction Format | Cycles |
| RA: | JR cc, address | 1110 cc displacement | 6 | 1110 cc displacement | 6 |

**Example:** If the result of the last arithmetic operation executed is negative, the next four instructions (which occupy a total of twelve bytes) are to be skipped. This can be accomplished with the instruction

JR MI, $ +14

If the S flag is not set, execution continues with the instruction following the JR.

A byte-saving form of a jump to the label LAB is

JR LAB

where LAB must be within the allowed range. The condition code is "blank" in this case, and indicates that the jump is always taken.

# LD
## Load

|  |  |
|---|---|
| **LD** dst, src<br>**LDB**<br>**LDL** | dst: R<br>src: R, IR, DA, X, BA, BX<br>or<br>dst: IR, DA, X, BA, BX<br>src: R<br>or<br>dst: R, IR, DA, X<br>src: IM |

**Operation:** dst ← src

The contents of the source are loaded into the destination. The contents of the source are not affected.

There are three versions of the Load instruction: Load into a register, load into memory and load an immediate value.

**Flags:** No flags affected

## Load Register

| Source Addressing Mode | Assembler Language Syntax | Nonsegmented Mode |  | Segmented Mode |  |  |
|---|---|---|---|---|---|---|
|  |  | Instruction Format | Cycles |  | Instruction Format | Cycles |
| **R:** | LD Rd, Rs<br>LDB Rbd, Rbs | 10 10000 W Rs Rd | 3 |  | 10 10000 W Rs Rd | 3 |
|  | LDL RRd, RRs | 10 010100 RRs RRd | 5 |  | 10 010100 RRs RRd | 5 |
| **IR:** | LD Rd, @Rs[1]<br>LDB Rbd, @Rs[1] | 00 10000 W Rs≠0 Rd | 7 |  | 00 10000 W RRs≠0 Rd | 7 |
|  | LDL RRd, @Rs[1] | 00 010100 Rs≠0 RRd | 11 |  | 00 010100 RRS≠0 RRd | 11 |
| **DA:** | LD Rd, address<br>LDB Rbd, address | 01 10000 W 0000 Rd<br>address | 9 | SS | 01 10000 W 0000 Rd<br>0 segment offset | 10 |
|  |  |  |  | SL | 01 10000 W 0000 Rd<br>1 segment 0000 0000<br>offset | 12 |
|  | LDL RRd, address | 01 010100 0000 RRd<br>address | 12 | SS | 01 010100 0000 RRd<br>0 segment offset | 13 |
|  |  |  |  | SL | 01 010100 0000 RRd<br>1 segment 0000 0000<br>offset | 15 |

Note 1: Word register in nonsegmented mode, register pair in segmented mode.

## Load Register (Continued)

| Source Addressing Mode | Assembler Language Syntax | Nonsegmented Mode |  | Segmented Mode |  |
|---|---|---|---|---|---|
|  |  | Instruction Format | Cycles | Instruction Format | Cycles |
| X: | LD Rd, addr(Rs)<br>LDB Rbd, addr(Rs) | 01 10000 W Rs≠0 Rd<br>address | 10 | SS 01 10000 W Rs≠0 Rd<br>0 segment offset | 10 |
|  |  |  |  | SL 01 10000 W Rs≠0 Rd<br>1 segment 0000 0000<br>offset | 13 |
|  | LDL RRd, addr(Rs) | 01 010100 Rs≠0 RRd<br>address | 13 | SS 01 010100 Rs≠0 RRd<br>0 segment offset | 13 |
|  |  |  |  | SL 01 010100 Rs≠0 RRd<br>1 segment 0000 0000<br>offset | 16 |
| BA: | LD Rd, Rs[1] (#disp)<br>LDB Rbd, Rs[1] (#disp) | 00 11000 W Rs≠0 Rd<br>displacement | 14 | 00 11000 W RRs≠0 RRd<br>displacement | 14 |
|  | LDL RRd, Rs[1] (#disp) | 00 110101 Rs≠0 RRd<br>displacement | 17 | 00 110101 RRs≠0 RRd<br>displacement | 17 |
| BX: | LD Rd, Rs[1] (Rx)<br>LDB Rbd, Rs[1] (Rx) | 01 11000 W Rs≠0 Rd<br>0000 Rx 0000 0000 | 14 | 01 11000 W RRs≠0 Rd<br>0000 Rx 0000 0000 | 14 |
|  | LDL RRd, Rs[1] (Rx) | 01 11010 1 Rs≠0 RRd<br>0000 Rx 0000 0000 | 17 | 01 11010 1 RRs≠0 RRd<br>0000 Rx 0000 0000 | 17 |

## Load Memory

| Destination Addressing Mode | Assembler Language Syntax | Nonsegmented Mode |  | Segmented Mode |  |
|---|---|---|---|---|---|
|  |  | Instruction Format | Cycles | Instruction Format | Cycles |
| IR: | LD @Rd[1], Rs<br>LDB @Rd[1], Rbs | 00 10111 W Rd≠0 Rs | 8 | 00 10111 W RRd≠0 Rs | 8 |
|  | LDL @Rd[1], RRs | 00 011101 Rd≠0 RRs | 11 | 00 011101 RRd≠0 RRs | 11 |
| DA: | LD address, Rs<br>LDB address, Rbs | 01 10111 W 0000 Rs<br>address | 11 | SS 01 10111 W 0000 Rs<br>0 segment offset | 12 |
|  |  |  |  | SL 01 10111 W 0000 Rs<br>1 segment 0000 0000<br>offset | 14 |

Note 1 Word register in nonsegmented mode, register pair in segmented mode

# Load Memory (Continued)

| Destination Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles | Instruction Format | Cycles |
| **DA:** | LDL address, RRs | 01 011101 0000 RRs<br>address | 14 | SS 01 011101 0000 RRs<br>0 segment offset | 15 |
| | | | | SL 01 011101 0000 RRs<br>1 segment 0000 0000<br>offset | 17 |
| **X:** | LD addr(Rd), Rs<br>LDB addr(Rd), Rbs | 01 10111 W Rd≠0 Rs<br>address | 12 | SS 01 10111 W Rd≠0 Rs<br>0 segment offset | 12 |
| | | | | SL 01 10111 W Rd≠0 Rs<br>1 segment 0000 0000<br>offset | 15 |
| | LDL addr(Rd), RRs | 01 011101 Rd≠0 RRs<br>address | 15 | SS 01 011101 Rd≠0 RRs<br>0 segment offset | 15 |
| | | | | SL 01 011101 Rd≠0 RRs<br>1 segment 0000 0000<br>offset | 18 |
| **BA:** | LD Rd[1](#disp), Rs<br>LDB Rd[1](#disp), Rbs | 00 11001 W Rd≠0 Rs<br>displacement | 14 | 00 11001 W RRd≠0 Rs<br>displacement | 14 |
| | LDL Rd[1](#disp), RRs | 00 110111 Rd≠0 RRs<br>displacement | 17 | 00 110111 RRd≠0 RRs<br>displacement | 17 |
| **BX:** | LD Rd[1](Rx), Rs<br>LDB Rd[1](Rx), Rbs | 01 11001 W Rd≠0 Rs<br>0000 Rx 0000 0000 | 14 | 01 11001 W RRd≠0 Rs<br>0000 Rx 0000 0000 | 14 |
| | LDL Rd[1](Rx), RRs | 01 110111 Rd≠0 RRs<br>0000 Rx 0000 0000 | 17 | 01 110111 RRd≠0 RRs<br>0000 Rx 0000 0000 | 17 |

Note 1· Word register in nonsegmented mode, register pair in segmented mode

# Load Immediate Value

| Destination Addressing Mode | Assembler Language Syntax | Nonsegmented Mode |  | Segmented Mode |  |
|---|---|---|---|---|---|
|  |  | Instruction Format | Cycles | Instruction Format | Cycles |
| R: | LD Rd, #data | 00 100001 0000 Rd<br>data | 7 | 00 100001 0000 Rd<br>data | 7 |
|  | LDB Rbd, #data[2] | 00 100000 0000 Rbd<br>data data | 7 | 00 100000 0000 Rbd<br>data data | 7 |
|  |  | 1100 Rd data | 5 | 1100 Rd data | 5 |
|  | LDL RRd, #data | 00 010100 0000 RRd<br>31 data (high) 16<br>15 data (low) 0 | 11 | 00 010100 0000 RRd<br>31 data (high) 16<br>15 data (low) 0 | 11 |
| IR: | LD @Rd[1], #data | 00 001101 Rd ≠ 0 0101<br>data | 11 | 00 001101 RRd≠0 0101<br>data | 11 |
|  | LDB @Rd[1], #data | 00 001100 Rd ≠ 0 0101<br>data data | 11 | 00 001100 RRd≠0 0101<br>data data | 11 |
| DA: | LD address, #data | 01 001101 0000 0101<br>address<br>data | 14 | SS 01 001101 0000 0101<br>0 segment offset<br>data | 15 |
|  |  |  |  | SL 01 001101 0000 0101<br>1 segment 0000 0000<br>offset<br>data | 17 |
|  | LDB address, #data | 01 001100 0000 0101<br>address<br>data data | 14 | SS 01 001100 0000 0101<br>0 segment offset<br>data data | 15 |
|  |  |  |  | SL 01 001100 0000 0101<br>1 segment 0000 0000<br>offset<br>data data | 17 |

Note 1: Word register in nonsegmented mode, register pair in segmented mode.

Note 2: Although two formats exist for "LDB R, IM", the assembler always uses the short format. In this case, the "src field" in the instruction format encoding contains the source operand.

# Load Immediate Value (Continued)

| Destination Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | | |
|---|---|---|---|---|---|---|
| | | Instruction Format | Cycles | | Instruction Format | Cycles |
| X: | LD addr(Rd), #data | 01 001101 Rd≠0 0101<br>address<br>data | 15 | SS | 01 001101 Rd≠0 0101<br>0 segment offset<br>data | 15 |
| | | | | SL | 01 001101 Rd≠0 0101<br>1 segment 0000 0000<br>offset<br>data | 18 |
| | LDB addr(Rd), #data | 01 001100 Rd≠0 0101<br>address<br>data data | 15 | SS | 01 001100 Rd≠0 0101<br>0 segment offset<br>data data | 15 |
| | | | | SL | 01 001100 Rd≠0 0101<br>1 segment 0000 0000<br>offset<br>data data | 18 |

**Example:** Several examples of the use of the Load instruction are treated in detail in Chapter 5 under addressing modes.

# LDA
## Load Address

**LDA** dst, src

dst: R
src: DA, X, BA, BX

**Operation:** dst ← address (src)

The address of the source operand is computed and loaded into the destination. The contents of the source are not affected. The address computation follows the rules for address arithmetic. The destination is a word register in nonsegmented mode, and a register pair in segmented mode.

In segmented mode, the address loaded into the destination has an undefined value in all reserved bits (bits 16-23 and bit 31). However, this address may be used by subsequent instructions in the indirect, base, or base-index addressing modes without any modification to the reserved bits.

**Flags:** No flags affected

| Source Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | | |
|---|---|---|---|---|---|---|
| | | Instruction Format | Cycles | | Instruction Format | Cycles |
| **DA:** | LDA Rd[1], address | 01 110110 0000 Rd<br>address | 12 | SS | 01 110110 0000 RRd<br>0 segment offset | 13 |
| | | | | SL | 01 110110 0000 RRd<br>1 segment 0000 0000<br>offset | 15 |
| **X:** | LDA Rd[1], addr(Rs) | 01 110110 Rs≠0 Rd<br>address | 13 | SS | 01 110110 Rs≠0 RRd<br>0 segment offset | 13 |
| | | | | SL | 01 110110 Rs≠0 RRd<br>1 segment 0000 0000<br>offset | 16 |
| **BA:** | LDA Rd[1], Rs[1] (#disp) | 00110100 Rs≠0 Rd<br>displacement | 15 | | 00110100 RRs≠0 RRd<br>displacement | 15 |
| **BX:** | LDA Rd[1], Rs[1] (Rx) | 01110100 Rs≠0 Rd<br>0000 Rx 0000 0000 | 15 | | 01110100 RRs≠0 RRd<br>0000 Rx 0000 0000 | 15 |

Note 1: Word register in nonsegmented mode, register pair in segmented mode.

**Examples:**

```
LDA  R4,STRUCT           !in nonsegmented mode, register R4 is loaded!
                         !with the nonsegmented address of the location!
                         !named STRUCT!
LDA  RR2, <<3>> 8(R4)    !in segmented mode, if index register R4!
                         !contains %20, then register RR2 is loaded!
                         !with the segmented address (segment 3, offset %28)!
LDA  RR2,RR4(#8)         !in segmented mode, if base register RR4!
                         !contains %01000020, then register RR2 is loaded!
                         !with the segment address << 1 >> %28!
                         !(segment 1, offset %28)!
```

Note 1 Word register in nonsegmented mode, register pair in segmented mode.

# LDAR
## Load Address Relative

**LDAR** dst, src    dst: R
src: RA

**Operation:** dst ◄– address (src)

The address of the source operand is computed and loaded into the destination. The contents of the source are not affected. The destination is a word register in nonsegmented mode, and a register pair in segmented mode. In segmented mode, the address loaded into the destination has all "reserved" bits (bits 16-23 and bit 31) cleared to zero.

The relative addressing mode is calculated by adding the displacement in the instruction to the updated value of the program counter (PC) to derive the address. The updated PC value is taken to be the address of the instruction following the LDAR instruction, while the displacement is a 16-bit signed value in the range -32768 to +32767. The addition is performed following the rules of address arithmetic, with no modifications to the segment number in segmented mode. Thus in segmented mode, the source operand must be in the same segment as the LDAR instruction.

The assembler automatically calculates the displacement by subtracting the PC value of the following instruction from the address given by the programmer.

**Flags:** No flags affected

| Source Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles | Instruction Format | Cycles |
| **RA:** | LDAR Rd[1], address | 00110100 \| 0000 \| Rd<br>displacement | 15 | 00110100 \| 0000 \| RRd<br>displacement | 15 |

**Example:**

```
LDAR    R2, TABLE      !in nonsegmented mode, register R2 is loaded!
                       !with the address of TABLE!
LDAR    RR4, TABLE     !in segmented mode, register pair RR4 is!
                       !loaded with the segmented address of TABLE,!
                       !which must be in the same segment as the program!
```

Note 1. Word register in nonsegmented mode, register pair in segmented mode

# LDCTL
## Load Control

## Privileged Instruction

| | |
|---|---|
| **LDCTL** dst, src | dst: CTLR<br>src: R<br>or<br>dst: R<br>src: CTLR |

**Operation:** dst ← src

This instruction loads the contents of a general purpose register into a control register, or loads the contents of a control register into a general-purpose register. The control register may be one of the following CPU registers:

| | |
|---|---|
| FCW | Flag and Control Word |
| REFRESH | Refresh Control |
| PSAPSEG | Program Status Area Pointer - segment number |
| PSAPOFF | Program Status Area Pointer - offset |
| NSPSEG | Normal Stack Pointer - segment number |
| NSPOFF | Normal Stack Pointer - offset |

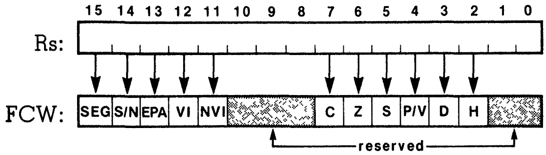
The operation of each of the variants of the instruction is detailed below, The ones which load data into a control register are described first, followed by the variants which load data from a control register into a general purpose register. Whenever bits are marked reserved, the corresponding bit in the source register must be either 0 or the value returned by a previous load from the same control register. For compatibility with future CPUs, programs should not assume that memory copies of control registers contain 0s, nor should they store data in reserved fields of memory copies of control registers.

## Load Into Control Register

LDCTL FCW, Rs

**Operation:**

FCW (2:7) ← Rs (2:7)
FCW (11:15) ← Rs (11:15)



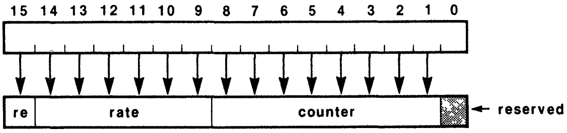
LDCTL REFRESH, Rs

**Operation:**

REFRESH (1:15) ← Rs (1:15)

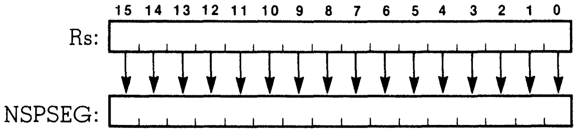
Rs:

REFRESH:



LDCTL NSPSEG, Rs

**Operation:** NSPSEG (0:15) ← Rs (0:15)

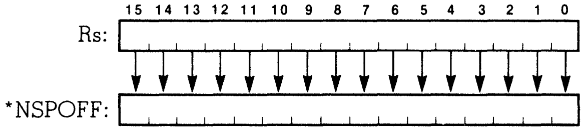


In segmented mode, the NSPSEG register is the normal mode R14 and contains the segment number of the normal mode processor stack pointer which is otherwise inaccessible for system mode.

In nonsegmented mode, R14 is not used as part of the normal processor stack pointer. This instruction may not be used in nonsegmented mode.

LDCTL NSPOFF, Rs
NSP, Rs

**Operation:** NSPOFF (0:15) ← Rs (0:15)



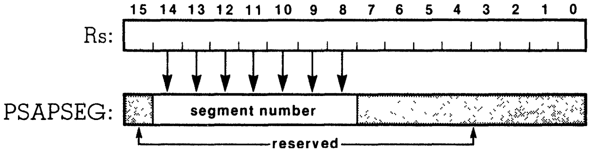
***NSP in nonsegmented mode**

In segmented mode, the NSPOFF register is R15 in normal mode and contains the offset part of the normal processor stack pointer. In nonsegmented mode, R15 is the entire normal processor stack pointer.

In nonsegmented mode, the mnemonic "NSP" should be used in the assembly language statement, and indicates the same control register as the mnemonic "NSPOFF".

LDCTL PSAPSEG, Rs

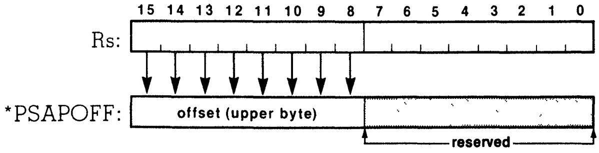
**Operation:** PSAPSEG (8:14) ← Rs (8:14)



The PSAPSEG register may not be used in nonsegmented operation. In segmented Z8000s, care must be exercised when changing the two PSAP register values so that an interrupt occurring between the changing of PSAPSEG and PSAPOFF is handled correctly. This is typically accomplished by first disabling interrupts before changing PSAPSEG and PSAPOFF.

LDCTL PSAPOFF, Rs
PSAP, Rs

**Operation:** PSAPOFF (8:15) ← Rs (8:15)



***PSAP in nonsegmented mode**

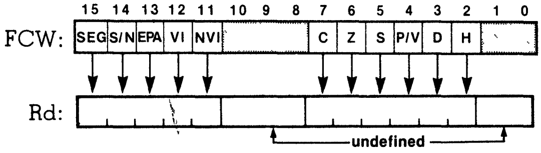
In the nonsegmented Z8000s, the mnemonic "PSAP" should be used in the assembly language statement and indicates the same control register as the mnemonic "PSAPOFF". In the segmented Z8000s, care must be exercised when changing the two PSAP register values so that an interrupt occurring between the changing of PSAPSEG and PSAPOFF is handled correctly. This is typically accomplished by first disabling interrupts before changing PSAPSEG and PSAPOFF. The low order byte of PSAPOFF should be 0.

## Load From Control Register

LDCTL Rd, FCW

**Operation:**

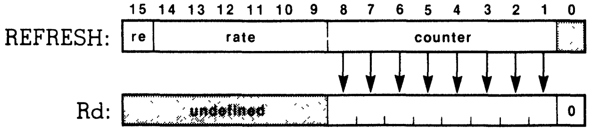
Rd (2:7) ← FCW (2:7)
Rd (11:15) ← FCW (11:15) (Z8001 only)
Rd (11:14) ← FCW (11:14) (Z8002 only)
Rd (0:1) ← UNDEFINED
Rd (8:10) ← UNDEFINED
Rd (15) ← 0 (Z8002 only)



LDCTL Rd, REFRESH

**Operation:**

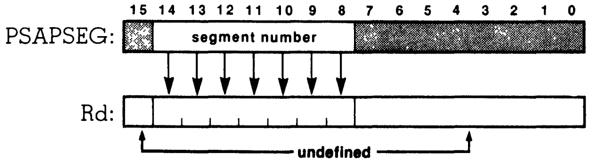
Rd (1:8) ← REFRESH (1:8)
Rd (0) ← UNDEFINED
Rd (9:15) ← UNDEFINED



LDCTL Rd, PSAPSEG

**Operation:**

Rd (8:14) ← PSAPSEG (8:14)
Rd (0:7) ← UNDEFINED
Rd (15) ← UNDEFINED

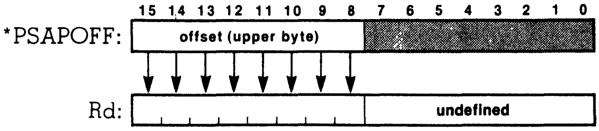


This instruction may not be used in nonsegmented mode.

LDCTL Rd, PSAPOFF
Rd, PSAP

**Operation:**

Rd (8:15) ← PSAPOFF (8:15)
Rd (0:7) ← UNDEFINED



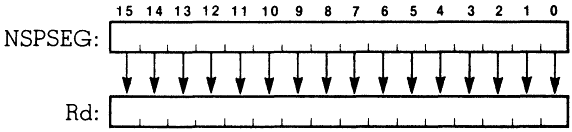
***PSAP in nonsegmented mode**

In nonsegmented mode, the mnemonic PSAP should be used in the assembly language statement, and it indicates the same control register as the mnemonic PSAPOFF.

LDCTL Rd, NSPSEG

**Operation:**

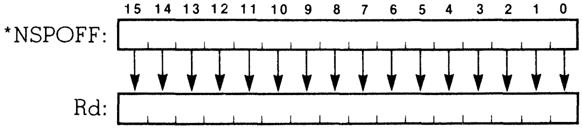
Rd (0:15) ← NSPSEG (0:15)



This instruction is not available in nonsegmented mode.

LDCTL Rd, NSPOFF
      Rd, NSP

**Operation:** Rd (0:15) ◄– NSPOFF (0:15)



***NSP in nonsegmented mode**

In nonsegmented mode, the mnemonic NSP should be used in the assembly language statement, and it indicates the same control register as the mnemonic NSPOFF.

**Flags:** No flags affected, except when the destination is the Flag and Control Word (LDCTL FCW, Rs), in which case all the flags are loaded from the source register.

| Source Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles | Instruction Format | Cycles |
| | LDCTL FCW, Rs | 01111101 Rs 1010 | 7 | 01111101 Rs 1010 | 7 |
| | LDCTL REFRESH, Rs | 01111101 Rs 1011 | 7 | 01111101 Rs 1011 | 7 |
| | LDCTL PSAPSEG, Rs | | | 01111101 Rs 1100 | 7 |
| | LDCTL PSAPOFF, Rs<br>PSAP, Rs | 01111101 Rs 1101 | 7 | 01111101 Rs 1101 | 7 |
| | LDCTL NSPSEG, Rs | | | 01111101 Rs 1110 | 7 |
| | LDCTL NSPOFF, Rs<br>NSP, Rs | 01111101 Rs 1111 | 7 | 01111101 Rs 1111 | 7 |

| Destination Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles | Instruction Format | Cycles |
| | LDCTL Rd, FCW | 01111101 Rd 0010 | 7 | 01111101 Rd 0010 | 7 |
| | LDCTL Rd, REFRESH | 01111101 Rd 0011 | 7 | 01111101 Rd 0011 | 7 |
| | LDCTL Rd, PSAPSEG | | | 01111101 Rd 0100 | 7 |
| | LDCTL Rd, PSAPOFF<br>LDCTL Rd, PSAP | 01111101 Rd 0101 | 7 | 01111101 Rd 0101 | 7 |
| | LDCTL Rd, NSPSEG | | | 01111101 Rd 0110 | 7 |
| | LDCTL Rd, NSPOFF<br>Rd, NSP | 01111101 Rd 0111 | 7 | 01111101 Rd 0111 | 7 |

# LDCTLB
## Load Control Byte

| | |
|---|---|
| **LDCTLB** dst, src | dst: FLAGS<br>src: R<br>or<br>dst: R<br>src: FLAGS |

**Operation:** dst ← src

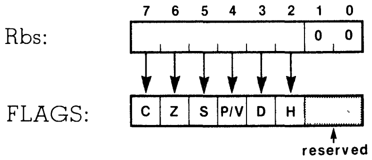
This instruction is used to load the FLAGS register or to transfer its contents into a general-purpose register. Note that this is not a privileged instruction.

## Load Into FLAGS Register

LDCTLB FLAGS, Rbs

FLAGS (2:7) ← src (2:7)

The contents of the source (a byte register) are loaded into the FLAGS register. The lower two bits of the FLAGS register and the entire source register are unaffected.

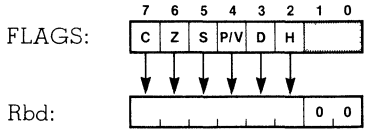


## Load From FLAGS Register

LDCTLB Rbd, FLAGS

dst (2:7) ← FLAGS (2:7)
dst (0:1) ← 0

The contents of the upper six bits of the FLAGS register are loaded into the destination (a byte register). The lower two bits of the destination register are cleared to zero. The FLAGS register is unaffected.



**Flags:** When the FLAGS register is the destination, all the flags are loaded from the source. When the FLAGS register is the source, none of the flags are affected.

|  | Assembler Language Syntax | Nonsegmented Mode |  | Segmented Mode |  |
|---|---|---|---|---|---|
|  |  | Instruction Format | Cycles | Instruction Format | Cycles |
|  | LDCTLB FLAGS, Rbs | 10001100 Rbs 1001 | 7 | 10001100 Rbs 1001 | 7 |
|  | LDCTLB Rbd, FLAGS | 10001100 Rbd 0001 | 7 | 10001100 Rbd 0001 | 7 |

# LDD
## Load and Decrement

**LDD** dst, src, r  dst: IR
**LDDB**  src: IR

**Operation:**

dst ← src
AUTODECREMENT dst and src (by 1 if byte, by 2 if word)
r ← r − 1

This instruction is used for block transfers of strings of data. The contents of the location addressed by the source register are loaded into the location addressed by the destination register. The source and destination registers are then decremented by one if LDDB, or by two if LDD, thus moving the pointers to the previous elements in the strings. The word register specified by "r" (used as a counter) is then decremented by one. The source, destination, and counter registers must be separate and non-overlapping registers.

**Flags:**

**C:** Unaffected
**Z:** Undefined
**S:** Unaffected
**V:** Set if the result of decrementing r is zero; cleared otherwise
**D:** Unaffected
**H:** Unaffected

| Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles | Instruction Format | Cycles |
| IR: | LDD @Rs[1], @Rd[1], r<br>LDDB @Rs[1], @Rd[1], r | 1011101 W Rs ≠ 0 1001<br>0000 r Rd ≠ 0 1000 | 20 | 1011101 W RRs ≠ 0 1001<br>0000 r RRd ≠ 0 1000 | 20 |

**Example:**

In nonsegmented mode, if register R1 contains %202A, register R2 contains %404A, the word at location %404A contains %FFFF, and register R3 contains 5, the instruction

LDD @R1, @R2, R3

will leave the value %FFFF at location %202A, the value %2028 in R1, the value %4048 in R2, and the value 4 in R3. The V flag will be cleared. In segmented mode, register pairs would be used instead of R1 and R2.

[1] Note 1: Word register in nonsegmented mode, register pair in segmented mode

# LDDR

## Load, Decrement and Repeat

| | |
|---|---|
| **LDDR** dst, src, r | dst: IR |
| **LDDRB** | src: IR |

**Operation:**

dst ← src<br>
AUTODECREMENT dst and src (by 1 if byte, by 2 if word)<br>
r ← r − 1<br>
repeat until r = 0

This instruction is used for block transfers of strings of data. The contents of the location addressed by the source register are loaded into the location addressed by the destination register. The source and destination registers are then decremented by one if LDDRB, or by two if LDDR, thus moving the pointers to the previous elements in the strings. The word register specified by "r" (used as a counter) is then decremented by one. The entire operation is repeated until the result of decrementing r is zero. The source, destination, and counter registers must be separate and non-overlapping registers. This instruction can transfer from 1 to 65536 bytes or from 1 to 32768 words (the value for r must not be greater than 32768 for LDDR).

The effect of decrementing the pointers during the transfer is important if the source and destination strings overlap with the source string starting at a lower memory address. Placing the pointers at the highest address of the strings and decrementing the pointers ensures that the source string will be copied without destroying the overlapping area.

This instruction can be interrupted after each execution of the basic operation. The program counter value of the start of this instruction is saved before the interrupt request is accepted, so that the instruction can be properly resumed. Seven cycles should be added to this instruction's execution time for each interrupt request that is accepted.

**Flags:**

**C:** Unaffected<br>
**Z:** Undefined<br>
**S:** Unaffected<br>
**V:** Set<br>
**D:** Unaffected<br>
**H:** Unaffected

| Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles$^2$ | Instruction Format | Cycles$^2$ |
| **IR:** | LDDR @Rd$^1$, @Rs$^1$, r<br>LDDRB @Rd$^1$, @Rs$^1$, r | 1011101 W Rs ≠ 0 1001<br>0000 r Rd ≠ 0 0000 | 11 + 9n | 1011101 W RRs≠0 1001<br>0000 r RRd≠0 0000 | 11 + 9n |

**Example:**

In nonsegmented mode, if register R1 contains %202A, register R2 contains %404A, the words at locations %4040 through %404A all contain %FFFF, and register R3 contains 6, the instruction

```
LDDR   @R1, @R2, R3
```

will leave the value %FFFF in the words at locations %2020 through %202A, the value %201E in R1, the value %403E in R2, and 0 in R3. The V flag will be set. In segmented mode, register pairs would be used instead of R1 and R2.

Note 1 Word register in nonsegmented mode, register pair in segmented mode.

Note 2· n = number of data elements transferred.

# LDI
## Load and Increment

| | |
|---|---|
| **LDI** dst, src, r | dst: IR |
| **LDIB** | src: IR |

**Operation:**

dst ← src<br>
AUTOINCREMENT dst and src (by 1 if byte, by 2 if word)<br>
r ← r − 1

This instruction is used for block transfers of strings of data. The contents of the location addressed by the source register are loaded into the location addressed by the destination register. The source and destination registers are then incremented by one if LDIB, or by two if LDI, thus moving the pointers to the next elements in the strings. The word register specified by "r" (used as a counter) is then decremented by one. The source, destination, and counter registers must be separate and non-overlapping registers.

**Flags:**

**C:** Unaffected<br>
**Z:** Undefined<br>
**S:** Unaffected<br>
**V:** Set if the result of decrementing r is zero, cleared otherwise<br>
**D:** Unaffected<br>
**H:** Unaffected

| Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles | Instruction Format | Cycles |
| IR: | LDI @Rd[1], @Rs[1], r<br>LDIB @Rd[1], @Rs[1], r | 1011101 \| W \| Rs ≠ 0 \| 0001<br>0000 \| r \| Rd ≠ 0 \| 1000 | 20 | 1011101 \| W \| RRs≠0 \| 0001<br>0000 \| r \| RRd≠0 \| 1000 | 20 |

**Example:**

This instruction can be used in a "loop" of instructions which transfers a string of data from one location to another, but an intermediate operation on each data element is required. The following sequence transfers a string of 80 bytes, but tests for a special value (%0D, an ASCII return character) which terminates the loop if found. This example assumes nonsegmented mode. In segmented mode, register pairs would be used instead of R1 and R2.

```
         LD      R3, #80          !initialize counter!
         LDA     R1, DSTBUF       !load start addresses!
         LDA     R², SRCBUF
LOOP:
         CPB     @R2, #%0D        !check for return character!
         JR      EQ, DONE         !exit loop if found!
         LDIB    @R1, @R2, R3     !transfer next byte!
         JR      NOV, LOOP        !repeat until counter = 0!
DONE:
```

[1] Note 1: Word register in nonsegmented mode, register pair in segmented mode.

# LDIR
## Load, Increment and Repeat

| | |
|---|---|
| **LDIR** dst, src, r<br>**LDIRB** | dst: IR<br>src: IR |

**Operation:**

dst ← src
AUTOINCREMENT dst and src (by 1 if byte; by two if word)
r ← r − 1
repeat until R = 0

This instruction is used for block transfers of strings of data. The contents of the location addressed by the source register are loaded into the location addressed by the destination register. The source and destination registers are then incremented by one if LDIRB, or by two if LDIR, thus moving the pointers to the next elements in the strings. The word register specified by "r" (used as a counter) is then decremented by one. The entire operation is repeated until the result of decrementing r is zero. The source, destination, and counter registers must be separate and non-overlapping registers. This instruction can transfer from 1 to 65536 bytes or from 1 to 32768 words (the value for r must not be greater than 32768 for LDIR).

The effect of incrementing the pointers during the transfer is important if the source and destination strings overlap with the source string starting at a higher memory address. Placing the pointers at the lowest address of the strings and incrementing the pointers ensures that the source string will be copied without destroying the overlapping area.

This instruction can be interrupted after each execution of the basic operation. The program counter value of the start of this instruction is saved before the interrupt request is accepted, so that the instruction can be properly resumed. Seven cycles should be added to this instruction's execution time for each interrupt request that is accepted.

**Flags:**

**C:** Unaffected
**Z:** Undefined
**S:** Unaffected
**V:** Set
**D:** Unaffected
**H:** Unaffected

| Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles[2] | Instruction Format | Cycles[2] |
| IR: | LDIR @Rd[1], @Rs[1], r<br>LDIRB @Rd[1], @Rs[1], r | 1011101 W Rs ≠ 0 0001<br>0000 r Rd ≠ 0 0000 | 11 + 9n | 1011101 W RRs≠0 0001<br>0000 r RRd≠0 0000 | 11 + 9 n |

**Example:** The following sequence of instructions can be used in nonsegmented mode to copy a buffer of 512 words (1024 bytes) from one area to another. The pointers to the start of the source and destination are set, the number of words to transfer is set, and then the transfer takes place.

```
LDA   R1, DSTBUF
LDA   R2, SRCBUF
LD    R3, #512
LDIR  @R1, @R2, R3
```

In segmented mode, R1 and R2 must be replaced by register pairs.

Note 1 Word register in nonsegmented mode, register pair in segmented mode

Note 2 n = number of data elements transferred

# LDK
## Load Constant

**LDK** dst, src

dst: R
src: IM

**Operation:** dst ← src (src = 0 to 15)

The source operand (a constant value specified in the src field) is loaded into the destination register. The source operand is a value from 0 to 15. It is loaded into the four low-order bits of the destination register, while the high-order 12 bits are cleared to zero.

**Flags:** No flags affected

| Destination Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles | Instruction Format | Cycles |
| **R:** | LDK Rd, #data | 10 111101 Rd data | 5 | 10 111101 Rd data | 5 |

**Example:** To load register R3 with the constant 9:

```
LDK   R3,#9
```

# LDM
## Load Multiple

|  |  |
|---|---|
| **LDM** dst, src, n | dst: R<br>src: IR, DA, X<br>or<br>dst: IR, DA, X<br>src: R |

**Operation:** dst ◄- src(n words)

The contents of n source words are loaded into the destination. The contents of the source are not affected. The value of n lies between 1 and 16, inclusive. This instruction moves information between memory and registers; registers are accessed in increasing order starting with the specified register; R0 follows R15. The instruction can be used either to load multiple registers into memory (e.g. to save the contents of registers upon subroutine entry) or to load multiple registers from memory (e.g. to restore the contents of registers upon subroutine exit).

The instruction encoding contains values from 0 to 15 in the "num" field corresponding to values of 1 to 16 for n, the number of registers to be loaded or saved.

The starting address is computed once at the start of execution, and incremented by two for each register loaded. If the original address computation involved a register, the register's value will not be affected by the address incrementation during execution. Similarly, modifying that register during a load from memory will not affect the address used by this instruction.

**Flags:** No flags affected

## Load Multiple - Registers From Memory

| Source Addressing Mode | Assembler Language Syntax | Nonsegmented Mode |  | Segmented Mode |  |
|---|---|---|---|---|---|
|  |  | Instruction Format | Cycles[2] | Instruction Format | Cycles[2] |
| **IR:** | LDM Rd, @Rs[1], #n | 00 \| 011100 \| Rs≠0 \| 0001<br>0000 \| Rd \| 0000 \| n−1 | 11 + 3n | 00 \| 011100 \| RRs≠0 \| 0001<br>0000 \| Rd \| 0000 \| n−1 | 11 + 3n |
| **DA:** | LDM Rd, address, #n | 01 \| 011100 \| 0000 \| 0001<br>0000 \| Rd \| 0000 \| n−1<br>address | 14 + 3n | **SS**<br>01 \| 011100 \| 0000 \| 0001<br>0000 \| Rd \| 0000 \| n−1<br>0 \| segment \| offset | 15 + 3n |
|  |  |  |  | **SL**<br>01 \| 011100 \| 0000 \| 0001<br>0000 \| Rd \| 0000 \| n−1<br>1 \| segment \| 0000 0000<br>offset | 17 + 3n |

## Load Multiple - Registers From Memory (Continued)

| Source Addressing Mode | Assembler Language Syntax | Nonsegmented Mode |  | Segmented Mode |  |
|---|---|---|---|---|---|
|  |  | Instruction Format | Cycles[2] | Instruction Format | Cycles[2] |
| X: | LDM Rd, addr(Rs), #n | 01 011100 Rs≠0 0001<br>0000 Rd 0000 n−1<br>address | 15+3n | SS 01 011100 Rs≠0 0001<br>0000 Rd 0000 n−1<br>0 segment offset | 15+3n |
|  |  |  |  | SL 01 011100 Rs≠0 0001<br>0000 Rd 0000 n−1<br>1 segment 0000 0000<br>offset | 18+3n |

## Load Multiple - Memory From Registers

| Destination Addressing Mode | Assembler Language Syntax | Nonsegmented Mode |  | Segmented Mode |  |
|---|---|---|---|---|---|
|  |  | Instruction Format | Cycles[2] | Instruction Format | Cycles[2] |
| IR: | LDM@Rd[1], Rs, #n | 00 011100 Rd≠0 1001<br>0000 Rs 0000 n−1 | 11+3n | 00 011100 RRd≠0 1001<br>0000 Rs 0000 n−1 | 11+3n |
| DA: | LDM address, Rs, #n | 01 011100 0000 1001<br>0000 Rs 0000 n−1<br>address | 14+3n | SS 01 011100 0000 1001<br>0000 Rs 0000 n−1<br>0 segment offset | 15+3n |
|  |  |  |  | SL 01 011100 0000 1001<br>0000 Rs 0000 n−1<br>1 segment 0000 0000<br>offset | 17+3n |
| X: | LDM addr(Rd), Rs, #n | 01 011100 Rd≠0 1001<br>0000 Rs 0000 n−1<br>address | 15+3n | SS 01 011100 Rd≠0 1001<br>0000 Rs 0000 n−1<br>0 segment offset | 15+3n |
|  |  |  |  | SL 01 011100 Rd≠0 1001<br>0000 Rs 0000 n−1<br>1 segment 0000 0000<br>offset | 18+3n |

**Example:** In nonsegmented mode, if register R5 contains 5, R6 contains %0100, and R7 contains 7, the statement

LDM @R6, R5, #3

will leave the values 5, %0100, and 7 at word locations %0100, %0102, and %0104, respectively, and none of the registers will be affected. In segmented mode, a register pair would be used instead of R6.

[1] Word register in nonsegmented mode, register pair in segmented mode

[2] n = number of registers

# LDPS
## Load Program Status

**Privileged Instruction**

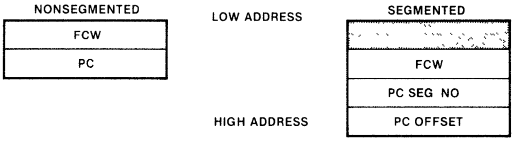
**LDPS** src     src: IR, DA, X

**Operation:** PS ◄– src

The contents of the source operand are loaded into the Program Status (PS), loading the Flags and Control Word (FCW) and the program counter (PC). The new value of the FCW does not become effective until the next instruction, so that the status pins will not be affected by the new control bits until after the LDPS instruction execution is completed. The next instruction executed is that addressed by the new contents of the PC. The contents of the source are not affected.

This instruction is used to set the Program Status of a program and is particularly useful for setting the System/Normal mode of a program to Normal mode, or for running a nonsegmented program in segmented Z8000s. The PC segment number is not affected by the LDPS instruction in nonsegmented mode.

The format of the source operand (Program Status block) depends on the current Segmentation mode (not on the version of the Z8000) and is illustrated in the following figure:



(shaded area is reserved—must be zero)

**Flags:** All flags are loaded from the source operand.

| Source Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | | |
|---|---|---|---|---|---|---|
| | | Instruction Format | Cycles | | Instruction Format | Cycles |
| **IR:** | LDPS @Rs[1] | 00 111001 Rs≠0 0000 | 12 | | 00 111001 RRs≠0 0000 | 16 |
| **DA:** | LDPS address | 01 111001 0000 0000<br>address | 16 | SS | 01 111001 0000 0000<br>0 segment offset | 20 |
| | | | | SL | 01 111001 0000 0000<br>1 segment 0000 0000<br>offset | 22 |
| **X:** | LDPS addr(Rs) | 01 111001 Rs≠0 0000<br>address | 17 | SS | 01 111001 Rs≠0 0000<br>0 segment offset | 20 |
| | | | | SL | 01 111001 Rs≠0 0000<br>1 segment 0000 0000<br>offset | 23 |

Note 1. Word register is used in nonsegmented mode, register pair in segmented mode.

**Example:**

In nonsegmented Z8000s, if the program counter contains %2550, register R3 contains %5000, location %5000 contains %1800, and location %5002 contains %A000, the instruction

```
LDPS  @R3
```

will leave the value %A000 in the program counter, and the FCW value will be %1800 (indicating Normal Mode, interrupts enabled, and all flags cleared.) In the segmented mode, a register pair is used instead of R3.

# LDR
## Load Relative

| | |
|---|---|
| **LDR** dst, src<br>**LDRB**<br>**LDRL** | dst: R<br>src: RA<br>or<br>dst: RA<br>src: R |

**Operation:** dst ◄– src

The contents of the source operand are loaded into the destination. The contents of the source are not affected. The relative address is calculated by adding the displacement in the instruction to the updated value of the program counter (PC) to derive the operand's address. In segmented mode, the segmented number of the computed address is the same as the segment number of the PC. The updated PC value is taken to be the address of the instruction following the LDR, LDRB, or LDRL instruction, while the displacement is a 16-bit signed value in the range -32768 to +32767.

Status pin information during the access to memory for the data operand will be Program Reference, (1100) instead of Data Memory request (1000).

The assembler automatically calculates the displacement by subtracting the PC value of the following instruction from the address given by the programmer.

This instruction must be used to modify memory locations containing program information, such as the Program Status Area, if program and data space are separated by the memory system.

**Flags:** No flags affected

## Load Relative Register

| Source Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles | Instruction Format | Cycles |
| **RA:** | LDR Rd, address<br>LDRB Rbd, address | 0011000 W 0000 Rd<br>displacement | 14 | 0011000 W 0000 Rd<br>displacement | 14 |
| | LDRL RRd, address | 00110101 0000 RRd<br>displacement | 17 | 00110101 0000 RRd<br>displacement | 17 |

## Load Relative Memory

| Destination Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles | Instruction Format | Cycles |
| **RA:** | LDR address, Rs<br>LDRB address, Rbs | 0011001 W 0000 Rs<br>displacement | 14 | 0011001 W 0000 Rs<br>displacement | 14 |
| | LDRL address, RRs | 00110111 0000 RRs<br>displacement | 17 | 00110111 0000 RRs<br>displacement | 17 |

**Example:** LDR R2, DATA !register R2 is loaded with the value in the!
!location named DATA!

## Privileged Instruction

# MBIT

## Multi-Micro Bit Test

**MBIT**

**Operation:** S ← 1 if $\overline{MI}$ high (inactive); 0 otherwise

This instruction is used to synchronize multiple processors' exclusive access to shared hardware resources. The multi-micro input pin ($\overline{MI}$) is tested, and the S flag is cleared if the pin is low (active); otherwise, the S flag is set, indicating that the pin is high (inactive).

After the MBIT instruction is executed, the S flag can be used to determine whether a requested resource is available or not. If the S flag is clear, then the resource is not available; if the S flag is set, then the resource is available for use by this CPU.

**Flags:**

**C:** Unaffected
**Z:** Undefined
**S:** Set if $\overline{MI}$ is high; cleared otherwise
**V:** Unaffected
**D:** Unaffected
**H:** Unaffected

| | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles | Instruction Format | Cycles |
| | MBIT | 0111101100001010 | 7 | 0111101100001010 | 7 |

**Example:** The following sequence of instructions can be used to wait for the availability of a resource.

```
        LOOP:
                 MBIT            !test multi-micro input!
                 JR    PL,LOOP   !repeat until resource is available!
   AVAILABLE:
```

## Privileged Instruction

# MREQ

## Multi-Micro Request

**MREQ** dst          dst: R

**Operation:**

```
Z ← 0
if $\overline{\text{MI}}$ low (active)  then S ← 0
                                MO forced high (inactive)
                        else $\overline{\text{MO}}$ forced low (active)
                             repeat dst ← dst $-$ 1 until dst $=$ 0
                             if $\overline{\text{MI}}$ low (active)  then S ← 1
                                                     else S ← 0
                                                          $\overline{\text{MO}}$ forced high (inactive)
                             Z ← 1
```

This instruction is used to synchronize multiple processors' exclusive access to shared hardware resources. A request for a resource is signalled through the multimicro input and output pins ($\overline{\text{MI}}$ and $\overline{\text{MO}}$), with the S and Z flags indicating the availability of the resource after the MREQ instruction has been executed.

First, the Z flag is cleared. Then the $\overline{\text{MI}}$ pin is tested. If the $\overline{\text{MI}}$ pin is low (active), the S flag is cleared and the $\overline{\text{MO}}$ pin is forced high (inactive),thus indicating that the resource is not available and removing any previous request by the CPU from the $\overline{\text{MO}}$ line.

If the $\overline{\text{MI}}$ pin is high (inactive), indicating that the resource may be available, a sequence of machine operations occurs. First, the $\overline{\text{MO}}$ pin is forced low (active), signalling a request by the CPU for the resource. Next, a finite delay to allow for propagation of the signal to other processors is accomplished by repeatedly decrementing the contents of the destination (a word register) until its value is zero. The original value of the counter must be greater than 2. Then the $\overline{\text{MI}}$ pin is tested to determine whether the request for the resource was acknowledged. If the $\overline{\text{MI}}$ pin is low (active), the S flag is set to one, indicating that the resource is available and access is granted. If the $\overline{\text{MI}}$ pin is still high (inactive), the S flag is cleared to zero, and the $\overline{\text{MO}}$ pin is forced high (inactive), indicating that the request was not granted and removing the request signal for the $\overline{\text{MO}}$. Finally, in either case, the Z flag is set to one, indicating that the original test of the $\overline{\text{MI}}$ pin caused a request to be made. External hardware should inhibit bus request while $\overline{\text{M0}}$ is active to ensure and upper bound on request timing.

| S flag | Z flag | $\overline{\text{MO}}$ | Indicates |
|---|---|---|---|
| 0 | 0 | high | Request not signalled<br>(resource not available) |
| 0 | 1 | high | Request not granted<br>(resource not available) |
| 1 | 1 | low | Request granted<br>(resource available) |

**Flags:**

**C:** Unaffected
**Z:** Set if request was signalled; cleared otherwise
**S:** Set if request was signalled and granted; cleared otherwise
**V:** Unaffected
**D:** Unaffected
**H:** Unaffected

| Destination Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles[1] | Instruction Format | Cycles[1] |
| **R:** | MREQ Rd | 01 \| 111011 \| Rd \| 1101 | 12 + 7n | 01 \| 111011 \| Rd \| 1101 | 12 + 7n |

**Example:**

```
TRY:
                LD      R0,#5            !allow for propagation delay!
                MREQ    R0               !multi-micro request with delay!
                                         !in register R0!
                JR    MI,AVAILABLE
                JR      Z,NOT_GRANTED
NOT_AVAILABLE:          .                !resource not available!
                        .
                        .
NOT_GRANTED:            .                !request not granted!
                        .
                        .
                JR    TRY                !try again after awhile!
AVAILABLE:              .                !use resource!
                        .
                        .
                MRES                     !release resource!
```

Note 1 If the request is made, n = number of times the destination is decremented If the request is not made, n = 0

# MRES
## Multi-Micro Reset

**MRES**

**Operation:** $\overline{MO}$ is forced high (inactive)

This instruction is used to synchronize multiple processors' exclusive access to shared hardware resources. The multi-micro output pin $\overline{MO}$ is forced high (inactive). Forcing $\overline{MO}$ high (inactive) indicates that a resource controlled by the CPU is available for use by other processors.

**Flags:** No flags affected.

| Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|
| | **Instruction Format** | **Cycles** | **Instruction Format** | **Cycles** |
| MRES | 01111011 \| 00001001 | 5 | 01111011 \| 00001001 | 5 |

**Example:**

```
MRES                    !signal that resource controlled by this CPU!
                        !is available to other processors!
```

# MSET

## Multi-Micro Set

## Privileged Instruction

**MSET**

**Operation:** $\overline{MO}$ is forced low (active)

This instruction is used to synchronize multiple processors' exclusive access to shared hardware resources. The multi-micro output pin $\overline{MO}$ is forced low (active). Forcing $\overline{MO}$ low (active) is used either to indicate that a resource controlled by the CPU is not available to other processors, or to signal a request for a resource controlled by some other processor.

**Flags:** No flags affected.

| | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles | Instruction Format | Cycles |
| | MSET | 01111011 \| 00001000 | 5 | 01111011 \| 00001000 | 5 |

**Example:** MSET !CPU controlled resource not available!

# MULT
## Multiply

**MULT** dst, src　　　dst: R
**MULTL**　　　　　　src: R, IM, IR, DA, X

**Operation:**

Word
dst (0:31) ← dst (0:15) × src (0:15)
Long
dst (0:63) ← dst (0:31) × src (0:31)

The low-order half of the destination operand (multiplicand) is multiplied by the source operand (multiplier) and the product is stored in the destination. The contents of the source are not affected. Both operands are treated as signed, two's complement integers. For MULT, the destination is a register pair and the source is a word value; for MULTL, the destination is a register quadruple and the source is a long word value.

For proper instruction execution, the "dst field" in the instruction format encoding must be even for MULT and must be a multiple of 4 (0, 4, 8, 12) for MULTL. If the source operand in MULTL is a register, the "src field" must be even.

The initial contents of the high-order half of the destination register do not affect the operation of this instruction and are overwritten by the result. The carry flag is set to indicate that the upper half of the destination register is required to represent the result; if the carry flag is clear, the product can be correctly represented in the same precision as the multiplicand and the upper half of the destination merely holds a sign extension.

The following table gives execution times for word and long word operands in each possible addressing mode.

| src | Word | | | Long Word | | |
|---|---|---|---|---|---|---|
| | NS | SS | SL | NS | SS | SL |
| R | 70 | 70 | 70 | 282+7*n | 282+7*n | 282+7*n |
| IM | 70 | 70 | 70 | 282+7*n | 282+7*n | 282+7*n |
| IR | 70 | 70 | 70 | 282+7*n | 282+7*n | 282+7*n |
| DA | 71 | 72 | 74 | 283+7*n | 284+7*n | 286+7*n |
| X | 72 | 72 | 75 | 284+7*n | 284+7*n | 287+7*n |

(n = number of bits equal to one in the absolute value of the low-order 16 bits of the destination operand)

When the multiplier is zero, the execution time of Multiply is reduced to the following times:

| src | Word | | | Long Word | | |
|---|---|---|---|---|---|---|
| | NS | SS | SL | NS | SS | SL |
| R | 18 | 18 | 18 | 30 | 30 | 30 |
| IM | 18 | 18 | 18 | 30 | 30 | 30 |
| IR | 18 | 18 | 18 | 30 | 30 | 30 |
| DA | 19 | 20 | 22 | 31 | 32 | 34 |
| X | 20 | 20 | 23 | 32 | 32 | 35 |

**Flags:**

**C:** MULT—set if product is less than $-2^{15}$ or greater than or equal to $2^{15}$; cleared otherwise; MULTL—set if product is less than $-2^{31}$ or greater than or equal to $2^{31}$; cleared otherwise
**Z:** Set if the result is zero; cleared otherwise
**S:** Set if the result is negative; cleared otherwise
**V:** Cleared
**D:** Unaffected
**H:** Unaffected

| Source Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | | Segmented Mode | |
|---|---|---|---|---|---|---|
| | | Instruction Format | Cycles[2] | | Instruction Format | Cycles[2] |
| R: | MULT RRd, Rs | 10 011001 Rs RRd | | | 10 011001 Rs RRd | |
| | MULTL RQd, RRs | 10 011000 Rs RRd | | | 10 011000 Rs RRd | |
| IM: | MULT RRd, #data | 00 011001 0000 RRd<br>data | | | 00 011001 0000 RRd<br>data | |
| | MULTL RQd, #data | 00 011000 0000 RQd<br>31 data (high) 16<br>15 data (low) 0 | | | 00 011000 0000 RQd<br>31 data (high) 16<br>15 data (low) 0 | |
| IR: | MULT RRd, @Rs[1] | 00 011001 Rs≠0 RRd | | | 00 011001 RRs≠0 RRd | |
| | MULTL RQd, @Rs[1] | 00 011000 Rs≠0 RQd | | | 00 011000 RRs≠0 RQd | |
| DA: | MULT RRd, address | 01 011001 0000 RRd<br>address | | SS | 01 011001 0000 RRd<br>0 segment offset | |
| | | | | SL | 01 011001 0000 RRd<br>1 segment 0000 0000<br>offset | |
| | MULTL RQd, address | 01 011000 0000 RQd<br>address | | SS | 01 011000 0000 RQd<br>0 segment offset | |
| | | | | SL | 01 011000 0000 RQd<br>1 segment 0000 0000<br>offset | |
| X: | MULT RRd, addr(Rs) | 01 011001 Rs≠0 RRd<br>address | | SS | 01 011001 Rs≠0 RRd<br>0 segment offset | |
| | | | | SL | 01 011001 Rs≠0 RRd<br>1 segment 0000 0000<br>offset | |
| | MULTL RQd, addr(Rs) | 01 011000 Rs≠0 RQd<br>address | | SS | 01 011000 Rs≠0 RQd<br>0 segment offset | |
| | | | | SL | 01 011000 Rs≠0 RQd<br>1 segment 0000 0000<br>offset | |

**Example:** If register RQ0 (composed of register pairs RR0 and RR2) contains %2222222200000031 (RR2 contains decimal 49), the statement

```
MULTL RQ0,#10
```

will leave the value %00000000000001EA (decimal 490) in RQ0.

Note 1 Word register in nonsegmented mode, register pair in segmented mode

Note 2 Execution times for each instruction are given in the preceding tables

# NEG
## Negate

**NEG** dst  dst: R, IR, DA, X
**NEGB**

**Operation:** dst ◄– –dst

The contents of the destination are negated, that is, replaced by its two's complement value. Note that %8000 for NEG and %80 for NEGB are replaced by themselves since in two's complement representation the negative number with greatest magnitude has no positive counterpart; for these two cases, the V flag is set.

**Flags:**
**C:** Cleared if the result is zero; set otherwise, which indicates a "borrow"
**Z:** Set if the result is zero; cleared otherwise
**S:** Set if the result is negative; cleared otherwise
**V:** Set if the result is %8000 for NEG, or %80 for NEGB: cleared otherwise
**D:** Unaffected
**H:** Unaffected

| Destination Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | | |
|---|---|---|---|---|---|---|
| | | Instruction Format | Cycles | | Instruction Format | Cycles |
| R: | NEG Rd<br>NEGB Rbd | 10 00110 W Rd 0010 | 7 | | 10 00110 W Rd 0010 | 7 |
| IR: | NEG @Rd[1]<br>NEGB @Rd[1] | 00 00110 W Rd≠0 0010 | 12 | | 00 00110 W RRd≠0 0010 | 12 |
| DA: | NEG address<br>NEGB address | 01 00110 W 0000 0010<br>address | 15 | SS | 01 00110 W 0000 0010<br>0 segment offset | 16 |
| | | | | SL | 01 00110 W 0000 0010<br>1 segment 0000 0000<br>offset | 18 |
| X: | NEG addr(Rd)<br>NEGB addr(Rd) | 01 00110 W Rd≠0 0010<br>address | 16 | SS | 01 00110 W Rd≠0 0010<br>0 segment offset | 16 |
| | | | | SL | 01 00110 W Rd≠0 0010<br>1 segment 0000 0000<br>offset | 19 |

**Example:** If register R8 contains %051F, the statement

NEG R8

will leave the value %FAE1 in R8.

Note 1 Word register in nonsegmented mode, register pair in segmented mode

# NOP
## No Operation

**NOP**

**Operation:** No operation is performed.

**Flags:** No flags affected

| | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles | Instruction Format | Cycles |
| | NOP | 10001101 \| 00000111 | 7 | 10001101 \| 00000111 | 7 |

# OR
## Or

**OR** dst, src    dst: R
**ORB**    src: R, IM, IR, DA, X

**Operation:** dst ← dst OR src

The source operand is logically ORed with the destination operand and the result is stored in the destination. The contents of the source are not affected. The OR operation results in a one bit being stored whenever either of the corresponding bits in the two operands is one; otherwise a zero bit is stored.

**Flags:**
**C:** Unaffected
**Z:** Set if the result is zero; cleared otherwise
**S:** Set if the most significant bit of the result is set; cleared otherwise
**P:** OR—unaffected; ORB—set if parity of the result is even; cleared otherwise
**D:** Unaffected
**H:** Unaffected

| Source Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles | Instruction Format | Cycles |
| R: | OR Rd, Rs<br>ORB Rbd, Rbs | 10 00010 W Rs Rd | 4 | 10 00010 W Rs Rd | 4 |
| IM: | OR Rd, #data | 00 000101 0000 Rd<br>data | 7 | 00 000101 0000 Rd<br>data | 7 |
| | ORB Rbd, #data | 00 000100 0000 Rd<br>data data | 7 | 00 000100 0000 Rd<br>data data | 7 |
| IR: | OR Rd, @Rs[1]<br>ORB Rbd, @Rs[1] | 00 00010 W Rs≠0 Rd | 7 | 00 00010 W RRs≠0 Rd | 7 |
| DA: | OR Rd, address<br>ORB Rbd, address | 01 00010 W 0000 Rd<br>address | 9 | SS 01 00010 W 0000 Rd<br>0 segment offset | 10 |
| | | | | SL 01 00010 W 0000 Rd<br>1 segment 0000 0000<br>offset | 12 |
| X: | OR Rd, addr(Rs)<br>ORB Rbd, addr(Rs) | 01 00010 W Rs≠0 Rd<br>address | 10 | SS 01 00010 W Rs≠0 Rd<br>0 segment offset | 10 |
| | | | | SL 01 00010 W Rs≠0 Rd<br>1 segment 0000 0000<br>address | 13 |

**Example:** If register RL3 contains %C3 (11000011) and the source operand is the immediate value %7B (01111011), the statement

```
ORB   RL3,#%7B
```

wıll leave the value %FB (11111011) in RL3.

Note 1 Word regıster ın nonsegmented mode, regıster paır ın segmented mode

# OTDR (SOTDR)

## Privileged Instruction

## (Special) Output, Decrement and Repeat

| | |
|---|---|
| **OTDR** dst, src, r | dst: IR |
| **OTDRB** | src: IR |
| **SOTDR** | |
| **SOTDRB** | |

**Operation:**

```
dst ← src
AUTODECREMENT src (by 1 if byte, by 2 if word)
r ← r - 1
repeat until r = 0
```

This instruction is used for block output of strings of data. OTDR and OTDRB are used for Standard I/O operation; SOTDR and SOTDRB are used for Special I/O operation. The contents of the memory location addressed by the source register are loaded into the I/O port addresses by the destination word register. I/O port addresses are 16 bits. The source register is then decremented by one if a byte instruction, or by two if a word instruction, thus moving the pointer to the previous element of the string in memory. The word register specified by "r" (used as a counter) is then decremented by one. The address of I/O port in the destination register is unchanged. The entire operation is repeated until the result of decrementing r is zero. This instruction can output from 1 to 65536 bytes or 32768 word (the value for r must not be greater than 32768 for OTDR or SOTDR).

This instruction can be interrupted after each execution of the basic operation. The program counter value of the start of this instruction is saved before the interrupt request is accepted, so that the instruction can be properly resumed. Seven more cycles should be added to this instruction's execution time for each interrupt request that is accepted.

**Flags:**

**C:** Unaffected
**Z:** Undefined
**S:** Unaffected
**V:** Set
**D:** Unaffected
**H:** Unaffected

| Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format[3] | Cycles[2] | Instruction Format[3] | Cycles[2] |
| IR: | OTDR @Rd,@Rs[1], r<br>OTDRB @Rd,@Rs[1], r<br>SOTDR @Rd,@Rs[1], r<br>SOTDRB @Rd,@Rs[1], r | 0011101 W Rs ≠ 0 101S<br>0000 r Rd ≠ 0 0000 | 11 + 10n | 0011101 W RRs≠0 101S<br>0000 r Rd ≠ 0 0000 | 11 + 10n |

**Example:**

In nonsegmented mode, if register R11 contains %0FFF, register R12 contains %B006, and R13 contains 6, the instruction

```
OTDR  @R11, @R12, R13
```

will output the string of words from locations %B006 to %AFFC (in descending order of address) to port %0FFF. R12 will contain %AFFA, and R13 will contain 0. R11 will not be affected. The V flag will be set. In segmented mode, R12 would be replaced by a register pair.

Note 1 Word register in nonsegmented mode, register pair in segmented mode

Note 2 n = number of data elements transferred.

Note 3. For SOTDR, S = 1, otherwise S = 0

# OTIR (SOTIR)

**Privileged Instruction**

## (Special) Output, Increment and Repeat

**OTIR** dst, src, r — dst: IR<br>
**OTIRB** — src: IR<br>
**SOTIR**<br>
**SOTIRB**

| | |
|---|---|
| **OTIR** dst, src, r | dst: IR |
| **OTIRB** | src: IR |
| **SOTIR** | |
| **SOTIRB** | |

**Operation:**

```
dst ◄- src
AUTOINCREMENT src (by 1 if byte, by 2 if word)
r ◄- r - 1
repeat until r = 0
```

This instruction is used for block output of strings of data. OTIR and OTIRB are used for Standard I/O operation; SOTIR and SOTIRB are used for Special I/O operation. The contents of the memory location addressed by the source register are loaded into the I/O port addressed by the destination word register. I/O port addresses are 16 bits. The source register is then incremented by one if a byte instruction, or by two if a word instruction, thus moving the pointer to the next element of the string in memory. The word register specified by "r" (used as a counter) is then decremented by one. The address of I/O port in the destination register is unchanged. The entire operation is repeated until the result of decrementing r is zero. This instruction can output from 1 to 65536 bytes or 32768 words (the value for r must not be greater than 32768 for OTIR or SOTIR).

This instruction can be interrupted after each execution of the basic operation. The program counter value of the start of this instruction is saved before the interrupt request is accepted, so that the instruction can be properly resumed. Seven more cycles should be added to this instruction's execution time for each interrupt request that is accepted.

**Flags:**

**C:** Unaffected<br>
**Z:** Undefined<br>
**S:** Unaffected<br>
**V:** Set<br>
**D:** Unaffected<br>
**H:** Unaffected

| Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format[3] | Cycles | Instruction Format[3] | Cycles |
| **IR:** | OTIR @Rd, @Rs[1], r<br>OTIRB @Rd, @Rs[1], r<br>SOTIR @Rd, @Rs[1], r<br>SOTIRB @Rd, @Rs[1], r | 0011101 W Rs ≠ 0 001S<br>0000 r Rd ≠ 0 0000 | 11 + 10n | 0011101 W RRs≠0 001S<br>0000 r Rd ≠ 0 0000 | 11 + 10n |

node, the following sequence of instructions can be used to output the specified I/O port. The pointers to the I/O port and the start g are set, the number of bytes to output is set, and then the output

```
#PORT
SRCBUF
#LENGTH
., @R2, R3
```

le, a register pair would be used instead of R2.

nonsegmented mode, register pair in segmented mode

data elements transferred

1, otherwise S = 0.

# OUT (SOUT)

## (Special) Output

**Privileged Instruction**

| | | |
|---|---|---|
| **OUT** dst, src<br>**OUTB** | dst: IR, DA<br>src: R |
| **SOUT** dst, src<br>**SOUTB** | dst: DA<br>src: R |

**Operation:** dst ◄– src

The contents of the source register are loaded into Special Output port. OUT and OUTB are used for and SOUTB are used for Special I/O operation.

**Flags:** No flags affected.

| Destination Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | |
|---|---|---|---|
| | | Instruction Format[1] | Cycles |
| **IR:** | OUT @Rd, Rs<br>OUTB @Rd, Rbs | 0011111 W Rd ≠ 0 Rs | 10 |
| **DA:** | OUT port, Rs<br>OUTB port, Rbs<br>SOUT port, Rs<br>SOUTB port, Rbs | 0011101 W Rs 011S<br>port | 12 |

**Example:** If register R6 contains %5252, the instruction

OUT %1120, R6

will output the value %5252 to the port %1120.

Note 1 For SOUT, S = 1, otherwise S = 0

**Example:** In nonsegmented mode, the following sequence of instructions can be used to output a string of bytes to the specified I/O port. The pointers to the I/O port and the start of the source string are set, the number of bytes to output is set, and then the output is accomplished.

```
LD       R1, #PORT
LDA      R2, SRCBUF
LD       R3, #LENGTH
OTIRB    @R1, @R2, R3
```

In segmented mode, a register pair would be used instead of R2.

Note 1 Word register in nonsegmented mode, register pair in segmented mode

Note 2 n = number of data elements transferred

Note 3 For SOTIR, S = 1; otherwise S = 0.

# OUT (SOUT) (Special) Output

## Privileged Instruction

| | |
|---|---|
| **OUT** dst, src<br>**OUTB** | dst: IR, DA<br>src: R |
| **SOUT** dst, src<br>**SOUTB** | dst: DA<br>src: R |

**Operation:** dst ← src

The contents of the source register are loaded into the destination, an Output or Special Output port. OUT and OUTB are used for Standard I/O operation; SOUT and SOUTB are used for Special I/O operation.

**Flags:** No flags affected.

| Destination Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format[1] | Cycles | Instruction Format[1] | Cycles |
| IR: | OUT @Rd, Rs<br>OUTB @Rd, Rbs | 0011111 W Rd ≠ 0 Rs | 10 | 0011111 W Rd ≠ 0 Rs | 10 |
| DA: | OUT port, Rs<br>OUTB port, Rbs<br>SOUT port, Rs<br>SOUTB port, Rbs | 0011101 W Rs 011S<br>port | 12 | 0011101 W Rs 011S<br>port | 12 |

**Example:** If register R6 contains %5252, the instruction

```
OUT    %1120, R6
```

will output the value %5252 to the port %1120.

[1] Note 1· For SOUT, S = 1, otherwise S = 0.

Privileged Instruction

# OUTD (SOUTD)

## (Special) Output and Decrement

| | |
|---|---|
| **OUTD** dst, src, r | dst: IR |
| **OUTDB** | src: IR |
| **SOUTD** | |
| **SOUTDB** | |

**Operation:**

dst ← src
AUTODECREMENT src (by 1 if byte, by 2 if word)
r ← r − 1

This instruction is used for block output of strings of data. OUTD and OUTDB are used for Standard I/O operation; SOUTD and SOUTDB are used for Special I/O operation. The contents of the memory location addressed by the source register are loaded into the I/O port addressed by the destination word register. I/O port addresses are 16 bits. The source register is then decremented by one if a byte instruction, or by two if a word instruction, thus moving the pointer to the previous element of the string in memory. The word register specified by "r" (used as a counter) is then decremented by one. The address of the I/O port in the destination register is unchanged.

**Flags:**

**C:** Unaffected
**Z:** Undefined
**S:** Unaffected
**V:** Set if the result of decrementing r is zero; cleared otherwise
**D:** Unaffected
**H:** Unaffected

| Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format[2] | Cycles | Instruction Format[2] | Cycles |
| IR: | OUTD @Rd, @Rs[1], r<br>OUTDB @Rd, @Rs[1], r<br>SOUTD @Rd, @Rs[1], r<br>SOUTDB @Rd, @Rs[1], r | 0011101 \| W \| Rs ≠ 0 \| 101S<br>0000 \| r \| Rd ≠ 0 \| 1000 | 21 | 0011101 \| W \| RRs≠0 \| 101S<br>0000 \| r \| Rd ≠ 0 \| 1000 | 21 |

**Example:**

In segmented mode, if register R2 contains the I/O port address %0030, register RR6 contains %12005552 (segment %12, offset %5552), the word at memory location %12005552 contains %1234, and register R8 contains %1001, the instruction

OUTD @R2, @RR6, R8

will output the value %1234 to port %0030 and leave the value %12005550 in RR6, and %1000 in R8. Register R2 will not be affected. The V flag will be cleared. In nonsegmented mode, a word register would be used instead of RR6.

[1] Note 1 Word register in nonsegmented mode, register pair in segmented mode

[2] Note 2 For SOUTD, S = 1, otherwise S = 0.

# OUTI (SOUTI)

## Privileged Instruction

## (Special) Output and Increment

| | |
|---|---|
| **OUTI** dst, src, r | dst: IR |
| **OUTIB** | src: IR |
| **SOUTI** | |
| **SOUTIB** | |

**Operation:**

dst ← src
AUTOINCREMENT src (by 1 if byte, by 2 if word)
r ← r − 1

This instruction is used for block output of strings of data. OUTI and OUTIB are used for Standard I/O operation; SOUTI and SOUTIB are used for Special I/O operation. The contents of the memory location addressed by the source register are loaded into the I/O port addressed by the destination word register. I/O port addresses are 16-bit. The source register is then incremented by one if a byte instruction, or by two if a word instruction, thus moving the pointer to the next element of the string in memory. The word register specified by "r" (used as a counter) is then decremented by one. The address of the I/O port in the destination register is unchanged.

**Flags:**

**C:** Unaffected
**Z:** Undefined
**S:** Unaffected
**V:** Set if the result of decrementing r is zero; cleared otherwise
**D:** Unaffected
**H:** Unaffected

| Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format[2] | Cycles | Instruction Format[2] | Cycles |
| IR: | OUTI @Rd, @Rs[1], r<br>OUTIB @Rd, @Rs[1], r<br>SOUTI @Rd, @Rs[1], r<br>SOUTIB @Rd, @Rs[1], r | 0011101 W Rs ≠ 0 001S<br>0000 r Rd ≠ 0 1000 | 21 | 0011101 W RRs ≠ 0 001S<br>0000 r Rd ≠ 0 1000 | 21 |

**Example:** This instruction can be used in a "loop" of instructions which outputs a string of data, but an intermediate operation on each element is required. The following sequence outputs a string of 80 ASCII characters (bytes) with the most significant bit of each byte set or reset to provide even parity for the entire byte. Bit 7 of each character is initially zero. This example assumes nonsegmented mode. In segmented mode, R2 would be replaced with a register pair.

```
        LD      R1, #PORT       !load I/O address!
        LDA     R2, SRCSTART    !load start of string!
        LD      R3, #80         !initialize counter!
LOOP:
        TESTB   @R2             !test byte parity!
        JR      PE, EVEN
        SETB    @R2, #7         !force even parity!
EVEN:
        OUTIB   @R1, @R2, R3    !output next byte!
        JR      NOV, LOOP       !repeat until counter = 0!
DONE:
```

Note 1. Word register in nonsegmented mode, register pair in segmented mode.

Note 2· For SOUTI, S = 1; otherwise S = 0

# POP
## Pop

**POP** dst, src  dst: R, IR, DA, X
**POPL**  src: IR

**Operation:**

dst ◄– src
AUTOINCREMENT src (by 2 ıf word, by 4 if long)

The contents of the location addressed by the source register (a stack pointer) are loaded into the destinatıon. The source register is then incremented by a value which equals the size in bytes of the destinatıon operand, thus removing the top element of the stack by changıng the stack poınter. Any regıster except R0 (or RR0 in segmented mode) can be used as a stack poınter.

The same register cannot be used in both the source and destination addressing fields.

**Flags:** No flags affected

| Destination Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles | Instruction Format | Cycles |
| R: | POP Rd, @Rs[1] | 10 010111 Rs ≠ 0 Rd | 8 | 10 010111 RRs≠0 Rd | 8 |
| | POPL RRd, @Rs[1] | 10 010101 Rs ≠ 0 RRd | 12 | 10 010101 RRs≠0 RRd | 12 |
| IR: | POP @Rd[1], @Rs[1] | 00 010111 Rs≠0 Rd ≠ 0 | 12 | 00 010111 RRs≠0 RRd≠0 | 12 |
| | POPL @Rd[1], @Rs[1] | 00 010101 Rs≠0 Rd ≠0 | 19 | 00 010101 RRs≠0 RRd≠0 | 19 |
| DA: | POP address, @Rs[1] | 01 010111 Rs≠0 0000<br>address | 16 | SS 01 010111 RRs≠0 0000<br>0 segment offset | 16 |
| | | | | SL 01 010111 RRs≠0 0000<br>1 segment 0000 0000<br>offset | 18 |
| | POPL address, @Rs[1] | 01 010101 Rs≠0 0000<br>address | 23 | SS 01 010101 RRs≠0 0000<br>0 segment offset | 23 |
| | | | | SL 01 010101 RRs≠0 0000<br>1 segment 0000 0000<br>offset | 25 |

| Destination Addressing Mode | Assembler Language Syntax | Nonsegmented Mode |  | Segmented Mode |  |
|---|---|---|---|---|---|
|  |  | Instruction Format | Cycles | Instruction Format | Cycles |
| X: | POP addr(Rd), @Rs[1] | 01 010111 Rs≠0 Rd≠0<br>address | 16 | SS 01 010111 RRs≠0 Rd≠0<br>0 segment offset | 16 |
|  |  |  |  | SL 01 010111 RRs≠0 Rd≠0<br>1 segment 0000 0000<br>offset | 19 |
|  | POPL addr(Rd), @Rs[1] | 01 010101 Rs≠0 Rd≠0<br>address | 23 | SS 01 010101 RRs≠0 Rd≠0<br>0 segment offset | 23 |
|  |  |  |  | SL 01 010101 RRs≠0 Rd≠0<br>1 segment 0000 0000<br>offset | 26 |

**Example:** In nonsegmented mode, if register R12 (a stack pointer) contains %1000, the word at location %1000 contains %0055, and register R3 contains %0022, the instruction

```
POP   R3, @R12
```

will leave the value %0055 in R3 and the value %1002 in R12. In segmented mode, a register pair must be used as the stack pointer instead of R12.

[1] Word register in nonsegmented mode, register pair in segmented mode

# PUSH
## Push

**PUSH** dst, src
**PUSHL**

dst: IR
src: R, IM, IR, DA, X

**Operation:**

AUTODECREMENT dst (by 2 if word, by 4 if long)
dst ◄— src

The contents of the destination register (a stack pointer) are decremented by a value which equals the size in bytes of the source operand. Then the source operand is loaded into the location addressed by the updated destination register, thus adding a new element to the top of the stack by changing the stack pointer. Any register except R0 (or RR0 in segmented mode) can be used as a stack pointer.

With PUSHL, the same register cannot be used for both the source and destination addressing fields.

**Flags:** No flags affected

| Source Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles | Instruction Format | Cycles |
| | PUSH @Rd$^1$, Rs | 10 010011 Rd≠0 Rs | 9 | 10 010011 Rd≠0 Rs | 9 |
| | PUSHL @Rd$^1$, RRs | 10 010001 Rd≠0 Rs | 12 | 10 010001 Rd≠0 Rs | |
| IM: | PUSH @Rd$^1$, #data | 00 001101 Rd≠0 1001<br>data | 12 | 00 001101 Rd≠0 1001<br>data | 12 |
| IR: | PUSH @Rd$^1$, @Rs$^1$ | 00 010011 Rd≠0 Rs ≠ 0 | 13 | 00 010011 Rd≠0 Rs ≠ 0 | 13 |
| | PUSHL @Rd$^1$, @Rs$^1$ | 00 010001 Rd≠0 Rs ≠ 0 | 20 | 00 010001 Rd≠0 Rs ≠ 0 | 20 |
| DA: | PUSH @Rd$^1$, address | 01 010011 Rd≠0 0000<br>address | 14 | SS 01 010011 Rd≠0 0000<br>0 segment offset | 14 |
| | | | | SL 01 010011 Rd≠0 0000<br>1 segment 0000 0000<br>offset | 17 |
| | PUSHL@Rd$^1$, address | 01 010001 Rd≠0 0000<br>address | 21 | SS 01 010001 Rd≠0 0000<br>0 segment offset | 13 |
| | | | | SL 01 010001 Rd≠0 0000<br>1 segment 0000 0000<br>offset | 24 |

| Source Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles | Instruction Format | Cycles |
| **X:** | PUSH @Rd[1], addr(Rs) | 01 010011 Rd≠0 Rs≠0<br>address | 14 | SS 01 010011 RRd≠0 Rs≠0<br>0 segment offset | 14 |
| | | | | SL 01 010011 RRd≠0 Rs≠0<br>1 segment 0000 0000<br>offset | 17 |
| | PUSHL @Rd[1], addr(Rs) | 01 010001 Rd≠0 Rs≠0<br>address | 21 | SS 01 010001 RRd≠0 Rs≠0<br>0 segment offset | 21 |
| | | | | SL 01 010001 RRd≠0 Rs≠0<br>1 segment 0000 0000<br>offset | 24 |

**Example:** In nonsegmented mode, if register R12 (a stack pointer) contains %1002, the word at location %1000 contains %0055, and register R3 contains %0022, the instruction

PUSH @R12, R3

will leave the value %0022 in location %1000 and the value %1000 in R12. In segmented mode, a register pair must be used as the stack pointer instead of R12.

Note 1: Word register is used in nonsegmented mode, register pair in segmented mode.

# RES
## Reset Bit

| | | |
|---|---|---|
| | **RES** dst, src<br>**RESB** | dst: R, IR, DA, X<br>src: IM<br>or<br>dst: R<br>src: R |

**Operation:** dst(src) ← 0

This instruction clears the specified bit within the destination operand without affecting any other bits in the destination. The source (the bit number) can be specified as either an immediate value (Static), or as a word register which contains the value (Dynamic). In the second case, the destination operand must be a register, and the source operand must be R0 through R7 for RESB, or R0 through R15 for RES. The bit number is a value from 0 to 7 for RESB, or 0 to 15 for RES, with 0 indicating the least significant bit.

Only the lower four bits of the source operand are used to specify the bit number for RES, while only the lower three bits of the source operand are used with RESB. When the source operand is an immediate value, the "src field" in the instruction format encoding contains the bit number in the lowest four bits for RES, or the lowest three bits for RESB.

**Flags:** No flags affected

## Reset Bit Static

| Destination Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | | |
|---|---|---|---|---|---|---|
| | | Instruction Format | Cycles | | Instruction Format | Cycles |
| **R:** | RES Rd, #b<br>RESB Rbd, #b | 10 10001 W Rd b | 4 | | 10 10001 W Rd b | 4 |
| **IR:** | RES @Rd1, #b<br>RESB @Rd1, #b | 00 10001 W Rd≠0 b | 11 | | 00 10001 W RRd≠0 b | 11 |
| **DA:** | RES address, #b<br>RESB address, #b | 01 10001 W 0000 b<br>address | 13 | SS | 01 10001 W 0000 b<br>0 segment offset | 14 |
| | | | | SL | 01 10001 W 0000 b<br>1 segment 0000 0000<br>offset | 16 |
| **X:** | 'RES addr(Rd), #b<br>RESB addr(Rd), #b | 01 10001 W Rd≠0 b<br>address | 14 | SS | 01 10001 W Rd≠0 b<br>0 segment offset | 14 |
| | | | | SL | 01 10001 W Rd≠0 b<br>1 segment 0000 0000<br>offset | 17 |

## Reset Bit Dynamic

| Source Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles | Instruction Format | Cycles |
| R: | RES Rd, Rs<br>RESB Rbd, Rs[2] | 00 10001 W 0000 Rs<br>0000 Rd 0000 0000 | 10 | 00 10001 W 0000 Rs<br>0000 Rd 0000 0000 | 10 |

**Example:** If register RL3 contains %B2 (10110010), the instruction

RESB RL3, #1

will leave the value %B0 (10110000) in RL3.

Note 1 Word register in nonsegmented mode, register pair in segmented mode.
Note 2· Word register 0-7 only

# RESFLG
## Reset Flag

**RESFLG** flag　　　　flag: C, Z, S, P, V

**Operation:** FLAGS (4:7) ← FLAGS (4:7) AND NOT instruction (4:7)

Any combination of the C, Z, S, P or V flags are cleared to zero if the corresponding bits in the instruction are one. If the bit in the instruction corresponding to a flag is zero, the flag will not be affected. All other bits in the FLAGS register are unaffected. Note that the P and V flags are represented by the same bit.

There may be one, two, three, or four operands in the assembly language statement, in any order.

**Flags:**

**C:** Cleared if specified, unaffected otherwise
**Z:** Cleared if specified, unaffected otherwise
**S:** Cleared if specified, unaffected otherwise
**P/V:** Cleared if specified, unaffected otherwise
**D:** Unaffected
**H:** Unaffected

| | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles | Instruction Format | Cycles |
| | RESFLG flags | 10 001101 CZSP/V 0011 | 7 | 10 001101 CZSP/V 0011 | 7 |

**Example:** If the C, S, and V flags are set (1) and the Z flag is clear (0), the statement

RESFLG C, V

will leave the S flag set (1), and the C, Z, and V flags cleared (0).

# RET
## Return

**RET** cc

**Operation:**

| Nonsegmented | Segmented |
|---|---|
| if cc is true then | if cc is true then |
| PC ◄– @SP | PC ◄– @SP |
| SP ◄– SP + 2 | SP ◄– SP + 4 |

This instruction is used to return to a previously executed procedure at the end of a procedure entered by a CALL or CALR instruction. If the condition specified by "cc" is satisfied by the flags in the FCW, then the contents of the location addressed by the processor stack pointer are popped into the program counter (PC). The next instruction executed is that addressed by the new contents of the PC. See section 6.6 for a list of condition codes. The stack pointer used is R15 in nonsegmented mode, or RR14 in segmented mode. If the condition is not satisfied, then the instruction following the RET instruction is executed. If no condition is specified, the return is taken regardless of the flag settings.

**Flags:** No flags affected

| Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles[1] | Instruction Format | Cycles[1] |
| | RET cc | 10 011110 0000 cc | 10/7 | 10 011110 0000 cc | 13/7 |

**Example:** In nonsegmented mode, if the program counter contains %2550, the stack pointer (R15) contains %3000, location %3000 contains %1004, and the Z flag is clear, then the instruction

RET NZ

will leave the value %3002 in the stack pointer and the program counter will contain %1004 (the address of the next instruction to be executed).

Note 1 The two values correspond to return taken and return not taken

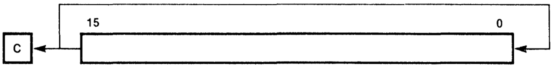
# RL
## Rotate Left

**RL** dst, src　　dst: R
**RLB**　　src: IM

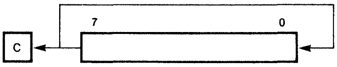
**Operation:**

```
Do src times: (src = 1 or 2)
        tmp ← dst
        c ← tmp (msb)
        dst(0) ← tmp (msb)
        dst (n + 1) ← tmp (n) (for n = 0 to msb − 1)
```

Word:



Byte:



The contents of the destination operand are rotated left one bit position if the source operand is 1, or two bit positions if the source operand is 2. The most significant bit (msb) of the destination operand is moved to the bit 0 position and also replaces the C flag.

The source operand may be omitted from the assembly language statement and thus defaults to the value 1.

**Flags:**

**C:** Set if the last bit rotated from the most significant bit position was 1; cleared otherwise
**Z:** Set if the result is zero; cleared otherwise
**S:** Set if the most significant bit of the result is set; cleared otherwise
**V:** Set if arithmetic overflow occurs, that is, if the sign of the destination changed during rotation; cleared otherwise
**D:** Unaffected
**H:** Unaffected

| Destination Addressing Mode | Assembler Language Syntax[1] | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format[2] | Cycles[3] | Instruction Format[2] | Cycles[3] |
| R: | RL Rd, #n<br>RLB Rbd, #n | 1 0 \| 1 1 0 0 1 \| W \| Rd \| 0 0 \| S \| 0 | 6/7 | 1 0 \| 1 1 0 0 1 \| W \| Rd \| 0 0 \| S \| 0 | 6/7 |

**Example:**

If register RH5 contains %88 (10001000), the statement

RLB　RH5

will leave the value %11 (00010001) in RH5 and the Carry flag will be set to one.

Note 1　n = source operand
Note 2　s = 0 for rotation by 1 bit, s = 1 for rotation by 2 bits
Note 3　The given execution times are for rotation by 1 and 2 bits respectively

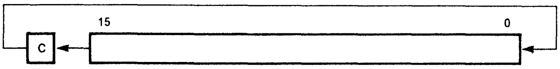
# RLC
## Rotate Left through Carry

**RLC** dst, src dst: R
**RLCB** src: IM

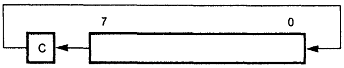
**Operation:**

```
Do src times: (src = 1 or 2)
        tmp ← c
        c ← dst (msb)
        dst (n + 1) ← dst (n) (for n = msb – 1 to 0)
        dst (0) ← tmp
```

Word:



Byte:



The contents of the destination operand with the C flag are rotated left one bit position if the source operand is 1, or two bit positions if the source operand is 2. The most significant bit (msb) of the destination operand replaces the C flag and the previous value of the C flag is moved to the bit 0 position of the destination during each rotation.

The source operand may be omitted from the assembly language statement and thus defaults to the value 1.

**Flags:**

**C:** Set if the last bit rotated from the most significant bit position was 1; cleared otherwise
**Z:** Set if the result is zero; cleared otherwise
**S:** Set if the most significant bit of the result is set; cleared otherwise
**V:** Set if arithmetic overflow occurs, that is, if the sign of the destination changed during rotation; cleared otherwise
**D:** Unaffected
**H:** Unaffected

| Destination Addressing Mode | Assembler Language Syntax[1] | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format[2] | Cycles[3] | Instruction Format[2] | Cycles[3] |
| **R:** | RLC Rd, #n<br>RLCB Rbd, #n | 10 \| 11001 \| W \| Rd \| 10 \| S \| 0 | 6/7 | 10 \| 11001 \| W \| Rd \| 10 \| S \| 0 | 6/7 |

**Example:**

If the Carry flag is clear (= 0) and register R0 contains %800F (1000000000001111), the statement

RLC R0,#2

will leave the value %003D (0000000000111101) in R0 and clear the Carry flag.

Note 1. n = source operand.
Note 2 s = 0 for rotation by 1 bit, s = 1 for rotation by 2 bits.
Note 3· The given execution times are for rotation by 1 and 2 bits respectively

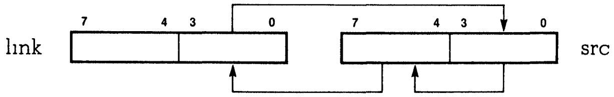
# RLDB
## Rotate Left Digit

**RLDB** link, src

src: R
link: R

**Operation:**

tmp (0:3) ← link (0:3)
link (0:3) ← src (4:7)
src (4:7) ← src (0:3)
src (0:3) ← tmp (0:3)



The low digit of the link byte register is logically concatenated to the source byte register. The resulting three-digit quantity is rotated to the left by one BCD digit (four bits). The lower digit of the source is moved to the upper digit of the source; the upper digit of the source is moved to the lower digit of the link, and the lower digit of the link is moved to the lower digit of the source. The upper digit of the link is unaffected. In multiple-digit BCD arithmetic, this instruction can be used to shift to the left a string of BCD digits, thus multiplying it by a power of ten. The link serves to transfer digits between successive bytes of the string. This is analogous to the use of the Carry flag in multiple precision shifting using the RLC instruction.

The same byte register must not be used as both the source and the link.

**Flags:**

**C:** Unaffected
**Z:** Set if the link is zero after the operation; cleared otherwise
**S:** Undefined
**V:** Unaffected
**D:** Unaffected
**H:** Unaffected

| Destination Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles | Instruction Format | Cycles |
| **R:** | RLDB Rbl, Rbs | 10 111110 Rbs Rbl | 9 | 10 111110 Rbs Rbl | 9 |

**Example:** If location 100 contains the BCD digits 0,1 (00000001), location 101 contains 2,3 (00100011), and location 102 contains 4,5 (01000101)

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 100 | 0 | 1 | 101 | 2 | 3 | 102 | 4 | 5 |

the sequence of statements

```
        LD       R3,#3         !set loop counter for 3 bytes!
                               !(6 digits)!
        LD       R2,#102       !set pointer to low-order digits!
        CLRB     RH1           !zero-fill low-order digit!
LOOP:
        LDB      RL1,@R2       !get next two digits!
        RLDB     RH1,RL1       !shift digits left one position!
        LDB      @R2,RL1       !replace shifted digits!
        DEC      R2            !advance pointer!
        DJNZ     R3, LOOP      !repeat until counter is zero!
```

will leave the digits 1,2 (00010010) in location 100, the digits 3,4 (00110100) in location 101, and the digits 5,0 (01010000) in location 102.

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 100 | 1 | 2 | 101 | 3 | 4 | 102 | 5 | 0 |

In segmented mode, R2 would be replaced by a register pair.

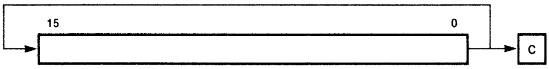
# RR
## Rotate Right

**RR** dst, src dst: R
**RRB** src: IM

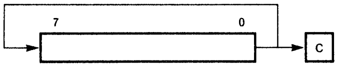
**Operation:**

```
Do src times: (src = 1 or 2)
        tmp ← dst
        c ← tmp (0)
        dst (msb) ← tmp (0)
        dst (n − 1) ← tmp (n) (for n = 1 to msb)
```

Word:



Byte:



The contents of the destination operand are rotated right one bit position if the source operand is 1, or two bit positions if the source operand is 2. The least significant bit of the destination operand is moved to the most significant bit (msb) and also replaces the C flag.

The source operand may be omitted from the assembly language statement and thus defaults to the value 1.

**Flags:**

**C:** Set if the last bit rotated from the least significant position was 1; cleared otherwise
**Z:** Set if the result is zero; cleared otherwise
**S:** Set if the most significant bit of the result is set; cleared otherwise
**V:** Set if arithmetic overflow occurs, that is, if the sign of the destination changed during rotation; cleared otherwise
**D:** Unaffected
**H:** Unaffected

| Destination Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format[1] | Cycles[2] | Instruction Format[1] | Cycles[2] |
| R: | RR Rd, #n<br>RRB Rbd, #n | 10 11001 W Rd 01 S 0 | 6/7 | 10 11001 W Rd 01 S 0 | 6/7 |

**Example:**

If register RL6 contains %31 (00110001), the statement

RRB RL6

will leave the value %98 (10011000) in RL6 and the Carry flag will be set to one.

Note 1 s = 0 for rotation by 1 bit, s = 1 for rotation by 2 bits.
Note 2· The given execution times are for rotation by 1 and 2 bits respectively

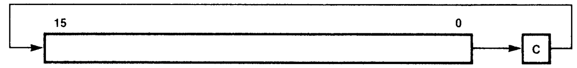
# RRC
## Rotate Right through Carry

**RRC** dst, src     dst: R
**RRCB**     src: IM

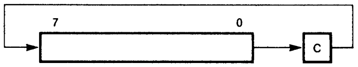
**Operation:**

```
Do src tımes: (src = 1 or 2)
        tmp ◄- c
        c ◄- dst (0)
        dst (n) ◄- dst (n + 1) (for n = 0 to msb - 1)
        dst (msb) ◄- tmp
```

Word:



Byte:



The contents of the destinatıon operand wıth the C flag are rotated one bıt posıtıon ıf the source operand ıs 1, or two bıt posıtıons ıf the source operand ıs 2. The least sıgnıfıcant bıt of the destınatıon operand replaces the C flag and the prevıous value of the C flag ıs moved to the most sıgnıfıcant bıt (msb) posıtıon of the destınatıon durıng each rotatıon.

The source operand may be omıtted from the assembly language statement and thus defaults to the value 1.

**Flags:**

**C:** Set ıf the last bıt rotated from the least sıgnıfıcant bıt posıtıon was 1; cleared otherwıse
**Z:** Set ıf the result ıs zero; cleared otherwıse
**S:** Set ıf the most sıgnıfıcant bıt of the result ıs set; cleared otherwıse
**V:** Set ıf arıthmetıc overflow occurs, that ıs, ıf the sıgn of the destınatıon changed durıng rotatıon; cleared otherwıse
**D:** Unaffected
**H:** Unaffected

| Destination Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format¹ | Cycles² | Instruction Format¹ | Cycles² |
| | RRC Rd, #n<br>RRCB Rbd, #n | 1 0 \| 1 1 0 0 1 \| W \| Rd \| 1 1 \| S \| 0 | 6/7 | 1 0 \| 1 1 0 0 1 \| W \| Rd \| 1 1 \| S \| 0 | 6/7 |

**Example:**

If the Carry flag is clear ( = 0) and the register R0 contains %00DD (0000000011011101), the statement

```
RRC   R0,#2
```

will leave the value %8037 (10000000110111) in R0 and clear the Carry flag.

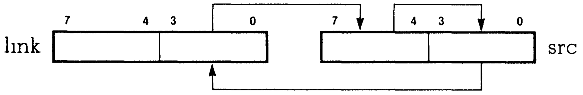
Note 1 s = 0 for rotatıon by 1 bıt, s = 1 for rotatıon by 2 bıts
Note 2 The gıven executıon tımes are for rotatıon by 1 and 2 bıts respectıvely

# RRDB
## Rotate Right Digit

**RRDB** link, src          src: R
                                  link: R

**Operation:**

```
tmp (0:3) ◄- link (0:3)
link (0:3) ◄- src (0:3)
src (0:3) ◄- src (4:7)
src (4:7) ◄- tmp (0:3)
```



The low digit of the link byte register is logically concatenated to the source byte register. The resulting three-digit quantity is rotated to the right by one BCD digit (four bits).

The lower digit of the source is moved to the lower digit of the link; the upper digit of the source is moved to the lower digit of the source and the lower digit of the link is moved to the upper digit of the source.

The upper digit of the link is unaffected. In multiple-digit BCD arithmetic, this instruction can be used to shift to the right a string of BCD digits, thus dividing it by a power of ten. The link serves to transfer digits between successive bytes of the string. This is analogous to the use of the carry flag in multiple precision shifting using the RRC instruction.

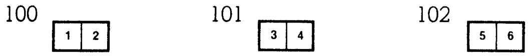
The same byte register must not be used as both the source and the link.

**Flags:**

**C:** Unaffected
**Z:** Set if the link is zero after the operation; cleared otherwise
**S:** Undefined
**V:** Unaffected
**D:** Unaffected
**H:** Unaffected

| Destination Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles | Instruction Format | Cycles |
| R: | RRDB Rbl, Rbs | 10 111100 Rbs Rbl | 9 | 10 111100 Rbs Rbl | 9 |

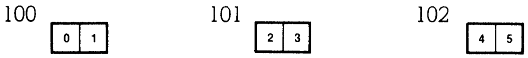
**Example:** If location 100 contains the BCD digits 1,2 (00010010), location 101 contains 3,4 (00110100), and location 102 contains 5,6 (01010110)



the sequence of statements

```
        LD      R3,#3           !set loop counter for 3 bytes (6
                                 digits)!
        LD      R2,#100         !set pointer to high-order digits!
        CLRB    RH1             !zero-fill high-order digit!
LOOP:
        LDB     RL1,@R2         !get next two digits!
        RRDB    RH1,RL1         !shift digits right one position!
        LDB     @R2,RL1         !replace shifted digits!
        INC     R2              !advance pointer!
        DJNZ    R3,LOOP         !repeat until counter is zero!
```

will leave the digits 0,1 (00000001) in location 100, the digits 2,3 (00100011) in location 101, and the digits 4,5 (01000101) in location 102. RH1 will contain 6, the remainder from dividing the string by 10.



In segmented mode, R2 would be replaced by a register pair.

# SBC
## Subtract with Carry

**SBC** dst, src    dst: R
**SBCB**    src: R

**Operation:** dst ◄– dst – src – C

The source operand, along with the setting of the carry flag, is subtracted from the destination operand and the result is stored in the destination. The contents of the source are not affected. Subtraction is performed by adding the two's complement of the source operand to the destination operand. In multiple precision arithmetic, this instruction permits the carry ("borrow") from the subtraction of low-order operands to be subtracted from the subtraction of high-order operands.

**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
**S:** Set if the result is negative; cleared otherwise
**V:** Set if arithmetic overflow occurs, that is, if the operands were of opposite signs and the sign of the result is the same as the sign of the source; cleared otherwise
**D:** SBC—unaffected; SBCB—set
**H:** SBC—unaffected; SBCB—cleared if there is a carry from the most significant bit of the low-order four bits of the result; set otherwise, indicating a "borrow"

| Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles | Instruction Format | Cycles |
| R: | SBC Rd, Rs<br>SBCB Rbd, Rbs | 10 11011 W Rs Rd | 5 | 10 11011 W Rs Rd | 5 |

**Example:** Long subtraction may be done with the following instruction sequence, assuming R0, R1 contain one operand and R2, R3 contain the other operand:

```
SUB R1,R3        !subtract low-order words!
SBC R0,R2        !subtract carry and high-order words!
```

If R0 contains %0038, R1 contains %4000, R2 contains %000A and R3 contains %F000, then the above two instructions leave the value %002D in R0 and %5000 in R1.

# SC
## System Call

**SC** src    src: IM

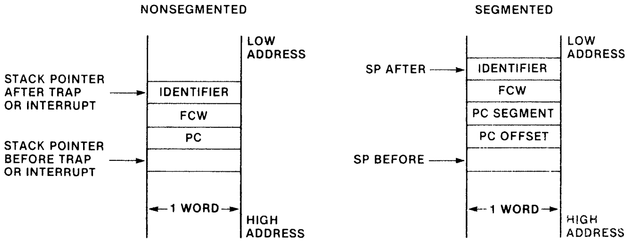
**Operation:**

| Nonsegmented | Segmented |
|---|---|
| SP ← SP − 4 | SP ← SP − 6 |
| @SP ← PS | @SP ← PS |
| SP ← SP − 2 | SP ← SP − 2 |
| @SP ← instruction | @SP ← instruction |
| PS ← System Call PS | PS ← System Call PS |

This instruction is used for controlled access to operating system software in a manner similar to a trap or interrupt. The current program status (PS) is pushed on the system processor stack, and then the instruction itself, which includes the source operand (an 8-bit value) is pushed. The PS includes the Flag and Control Word (FCW), and the updated program counter (PC). (The updated program counter value used is the address of the first instruction following the SC instruction.)

The system stack pointer is always used (R15 in nonsegmented CPUs, or RR14 in segmented CPUs), regardless of whether system or normal mode is in effect The new PS is then loaded from the Program Status block associated with the System Call trap (see section 6.2.4), and control is passed to the procedure whose address is the program counter value contained in the new PS. This procedure may inspect the source operand on the top of the stack to determine the particular software service desired.

The following figure illustrates the format of the saved program status in the system stack:



The segmented Z8000s always execute the segmented mode of the System Call instruction, regardless of the current mode, and set the Segmentation Mode bit (SEG) to segmented mode ( = 1) at the start of the SC instruction execution. All Z8000s set the System/Normal Mode bit (S/N) to system mode ( = 1) at the start of the SC instruction execution. The status pins reflect the setting of these control bits during the execution of the SC instruction. However, the setting of SEG and S/N does not affect the value of these bits in the old FCW pushed onto the stack. The new value of the FCW is not effective until the next instruction, so that the status pins will not be affected by the new control bits until after the SC instruction execution is completed.

The "src field" in the instruction format encoding contains the source operand. The "src field" values range from 0 to 255 corresponding to the source values 0 to 255.

**Flags:** Flags loaded from Program Status Area

| Source Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles | Instruction Format | Cycles |
| **IM:** | SC #src | 01111111 \| src | 33 | 01111111 \| src | 39 |

**Example:** In the nonsegmented Z8002, if the contents of the program counter are %1000, the contents of the system stack pointer (R15) are %3006, and the Program Counter and FCW values associated with the System Call trap in the Program Status Area are %2000 and %5800, respectively, the instruction

```
SC  #3                    !system call, request code = 3!
```

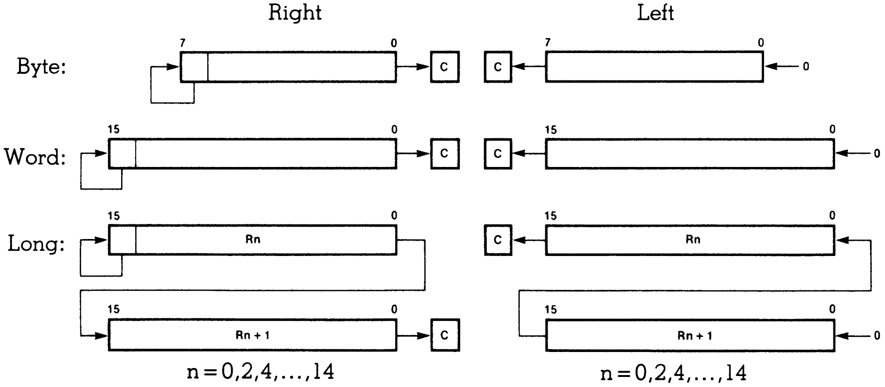
causes the system stack pointer to be decremented to %3000. Location %3000 contains %7F03 (the SC instruction). Location %3002 contains the old FCW, and location %3004 contains %1002 (the address of the instruction following the SC instruction). System mode is in effect, and the Program Counter contains the value %2000, which is the start of a System Call trap handler, and the FCW contains %5800.

# SDA
## Shift Dynamic Arithmetic

| | |
|---|---|
| **SDA** dst, src | dst: R |
| **SDAB** | src: R |
| **SDAL** | |

**Operation:**

```
Right (src negative)
Do – src times:
   c ← dst (0)
   dst (n) ← dst (n + 1) (for n = 0 to msb – 1)
   dst (msb) ← dst (msb)
Left (src positive)
Do src times:
   c ← dst (msb)
   dst (n + 1) ← dst (n) (for n = msb – 1 to 0)
   dst (0) ← 0
```



The destination operand is shifted arithmetically left or right by the number of bit positions specified by the contents of the source operand, a word register.

The shift count ranges from -8 to +8 for SDAB, from -16 to +16 for SDA and from -32 to +32 for SDAL. If the value is outside the specified range, the operation is undefined. The source operand is represented as a 16-bit two's complement value. Positive values specify a left shift, while negative values specify a right shift. A shift of zero positions does not affect the destination; however, the flags are set according to the destination value. The sign bit is replicated in shifts to the right, and the C flag is loaded from bit 0 of the destination. The least significant bit is filled with 0 in shifts to the left, and the C flag is loaded from the most significant bit (msb) of the destination. The setting of the carry bit is undefined for zero shift.

**Flags:**

**C:** Set if the last bit shifted from the destination was 1, undefined for zero shift; cleared otherwise
**Z:** Set if the result is zero; cleared otherwise
**S:** Set if the result is negative; cleared otherwise
**V:** Set if arithmetic overflow occurs, that is, if the sign of the destination changed during shifting; cleared otherwise
**D:** Unaffected
**H:** Unaffected

| Destination Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles[1] | Instruction Format | Cycles[1] |
| **R:** | SDA Rd, Rs | 10 110011 Rd 1011<br>0000 Rs 0000 0000 | 15+3n | 10 110011 Rd 1011<br>0000 Rs 0000 0000 | 15+3n |
| | SDAB Rbd, Rs | 10 110010 Rbd 1011<br>0000 Rs 0000 0000 | 15+3n | 10 110010 Rbd 1011<br>0000 Rs 0000 0000 | 15+3n |
| | SDAL RRd, Rs | 10 110011 RRd 1111<br>0000 Rs 0000 0000 | 15+3n | 10 110011 RRd 1111<br>0000 Rs 0000 0000 | 15+3n |

**Example:** If register R5 contains %C705 (1100011100000101) and register R1 contains −2 (%FFFE or 1111111111111110), the statement

SDA R5,R1

performs an arithmetic right shift of two bit positions, leaves the value %F1C1 (1111000111000001) in R5, and clears the Carry flag.

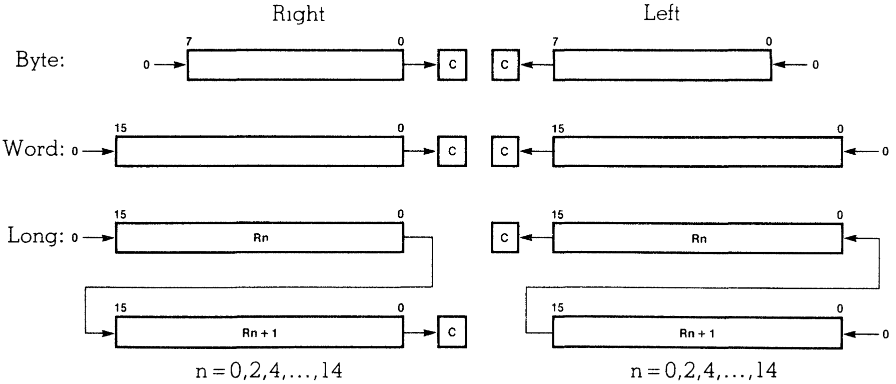
[1] Note 1. n = number of bit positions, the execution time for n =0 is the same as for n = 1

# SDL
## Shift Dynamic Logical

| | |
|---|---|
| **SDL** dst, src | dst: R |
| **SDLB** | src: R |
| **SDLL** | |

**Operation:**

```
Right (src negative)
Do −src times
   c ← dst (0)
   dst (n) ← dst (n + 1) (for n = 0 to msb − 1)
   dst (msb) ← 0
Left (src positive)
Do src times
   c ← dst (msb)
   dst (n + 1) ← dst (n) (for n = msb − 1 to 0)
   dst (0) ←
```



The destination operand is shifted logically left or right by the number of bit positions specified by the contents of the source operand, a word register. The shift count ranges from −8 to +8 for SDLB, from −16 to +16 for SDL and from −32 to +32 for SDLL. If the value is outside the specified range, the operation is undefined. The source operand is represented as a 16-bit two's complement value. Positive values specify a left shift, while negative values specify a right shift. A shift of zero positions does not affect the destination; however, the flags are set according to the destination value. The most significant bit (msb) is filled with 0 in shifts to the right, and the C flag is loaded from bit 0 of the destination. The least significant bit is filled with 0 in shifts to the left, and the C flag is loaded from the most significant bit of the destination. The setting of the carry bit is undefined for zero shift.

**Flags:**

**C:** Set if the last bit shifted from the destination was 1, undefined for zero shift; cleared otherwise
**Z:** Set if the result is zero; cleared otherwise
**S:** Set if the most significant bit of the result is set; cleared otherwise
**V:** Undefined
**D:** Unaffected
**H:** Unaffected

| Destination Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles | Instruction Format | Cycles |
| R: | SDL Rd, Rs | 10 \| 110011 \| Rd \| 0011<br>0000 \| Rs \| 0000 0000 | 15 + 3n | 10 \| 110011 \| Rd \| 0011<br>0000 \| Rs \| 0000 0000 | 15 + 3n |
| | SDLB Rbd, Rs | 10 \| 110010 \| Rbd \| 0011<br>0000 \| Rs \| 0000 0000 | 15 + 3n | 10 \| 110010 \| Rbd \| 0011<br>0000 \| Rs \| 0000 0000 | 15 + 3n |
| | SDLL RRd, Rs | 10 \| 110011 \| RRd \| 0111<br>0000 \| Rs \| 0000 0000 | 15 + 3n | 10 \| 110011 \| RRd \| 0111<br>0000 \| Rs \| 0000 0000 | 15 + 3n |

**Example:** If register RL5 contains %B3 (10110011) and register R1 contains 4 (0000000000000100), the statement

SDLB RL5,R1

performs a logical left shift of four bit positions, leaves the value %30 (00110000) in RL5, and sets the Carry flag.

Note 1 n = number of bit positions, the execution time for n = 0 is the same as for n = 1

# SET
## Set Bit

**SET** dst, src
**SETB**

dst: R, IR, DA, X
src: IM
or
dst: R
src: R

**Operation:** dst(src) ← 1

Sets the specified bit within the destination operand without affecting any other bits in the destination. The source (the bit number) can be specified as either an immediate value (Static), or as a word register which contains the value (Dynamic). In the second case, the destination operand must be a register, and the source operand must be R0 through R7 for SETB, or R0 through R15 for SET. The bit number is a value from 0 to 7 for SETB or 0 to 15 for SET, with 0 indicating the least significant bit.

Only the lower four bits of the source operand are used to specify the bit number for SET, while only the lower three bits of the source operand are used with SETB. When the source operand is an immediate value, the "src field" in the instruction format encoding contains the bit number in the lowest four bits for SET, or the lowest three bits for SETB.

**Flags:** No flags affected

## Set Bit Static

| Destination Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | | |
|---|---|---|---|---|---|---|
| | | Instruction Format | Cycles | | Instruction Format | Cycles |
| R: | SET Rd, #b<br>SETB Rbd, #b | 10 10010 W Rd b | 4 | | 10 10010 W Rd b | 4 |
| IR: | SET @Rd[1], #b<br>SETB @Rd[1], #b | 00 10010 W Rd≠0 b | 11 | | 00 10010 W RRd≠0 b | 11 |
| DA: | SET address, #b<br>SETB address, #b | 01 10010 W 0000 b<br>address | 13 | SS | 01 10010 W 0000 b<br>0 segment offset | 14 |
| | | | | SL | 01 10010 W 0000 b<br>1 segment 0000 0000<br>offset | 16 |
| X: | SET addr(Rd), #b<br>SETB addr(Rd), #b | 01 10010 W Rd≠0 b<br>address | 14 | SS | 01 10010 W Rd ≠ 0 b<br>0 segment offset | 14 |
| | | | | SL | 01 10010 W Rd ≠ 0 b<br>1 segment 0000 0000<br>offset | 17 |

Note 1: Word register in nonsegmented mode, register pair in segmented mode.

# Set Bit Dynamic

| Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles | Instruction Format | Cycles |
| R: | SET Rd, Rs<br>SETB Rbd, Rs[2] | 00 10010 W 0000 Rs<br>0000 Rd 0000 0000 | 10 | 00 10010 W 0000 Rs<br>0000 Rd 0000 0000 | 10 |

**Example:** If register RL3 contains %B2 (10110010) and register R2 contains the value 6, the instruction

SETB RL3, R2

will leave the value %F2 (11110010) in RL3.

Note 2: Word registers 0-7 only.

# SETFLG
## Set Flag

**SETFLG** flag    Flag: C, Z, S, P, V

**Operation:** FLAGS (4:7) ← FLAGS (4:7) OR instruction (4:7)

Any combination of the C, Z, S, P or V flags are set to one if the corresponding bits in the instruction are one. If the bit in the instruction corresponding to a flag is zero, the flag will not be affected. All other bits in the FLAGS register are unaffected. Note that the P and V flags are represented by the same bit.

There may be one, two, three, or four operands in the assembly language statement, in any order.

**Flags:**

**C:** Set if specified; unaffected otherwise
**Z:** Set if specified; unaffected otherwise
**S:** Set if specified; unaffected otherwise
**P/V:** Set if specified; unaffected otherwise
**D:** Unaffected
**H:** Unaffected

| | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles | Instruction Format | Cycles |
| | SETFLG flags | 10001101 \| CZSP/V \| 0001 | 7 | 10001101 \| CZSP/V \| 0001 | 7 |

**Example:** If the C, Z, and S flags are all clear (0), and the P flag is set (1), the statement

```
SETFLG  C
```

will leave the C and P flags set (1), and the Z and S flags cleared (0).

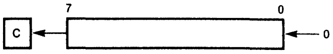
# SLA
## Shift Left Arithmetic

| | |
|---|---|
| **SLA** dst, src | dst: R |
| **SLAB** | src: IM |
| **SLAL** | |

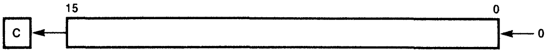
**Operation:**

```
Do src times:
    c ← dst (msb)
    dst (n + 1) ← dst (n) (for n = msb − 1 to 0)
    dst (0) ← 0
```

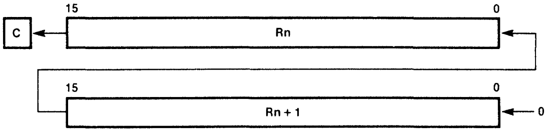
Byte:



Word:



Long:



n = 0, 2, 4, ..., 14

The destination operand is shifted arithmetically left the number of bit positions specified by the source operand. For SLAB, the source is in the range 0 to 8; for SLA, the source is in the range 0 to 16; for SLAL, the source is in the range 0 to 32. The least significant bit of the destination is filled with 0, and the C flag is loaded from the sign bit of the destination. The operation is the equivalent of a multiplication of the destination by a power of two with overflow indication. A shift of zero positions does not affect the destination; however, the flags are set according to the destination value with the C flag undefined.

The src field is encoded in the instruction format as the 8- or 16-bit two's complement positive value of the source operand. For each operand size, the operation is undefined if the source operand is not in the specified range.

The source operand may be omitted from the assembly language statement and thus defaults to the value 1.

**Flags:**

**C:** Set if the last bit shifted from the destination was 1, undefined for zero shift; cleared otherwise
**Z:** Set if the result is zero; cleared otherwise
**S:** Set if the result is negative; cleared otherwise
**V:** Set if arithmetic overflow occurs, that is, if the sign of the destination changed during shifting; cleared otherwise
**D:** Unaffected
**H:** Unaffected

| Destination Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles[1] | Instruction Format | Cycles[1] |
| R: | SLA Rd, #b | 10 110011 Rd 1001<br>b | 13 + 3b | 10 110011 Rd 1001<br>b | 13 + 3b |
| | SLAB Rbd, #b | 10 110010 Rbd 1001<br>0 b | 13 + 3b | 10 110010 Rbd 1001<br>0 b | 13 + 3b |
| | SLAL RRd, #b | 10 110011 RRd 1101<br>b | 13 + 3b | 10 110011 RRd 1101<br>b | 13 + 3b |

**Example:** If register pair RR2 contains %1234ABCD, the statement

SLAL RR2,#8

will leave the value %34ABCD00 in RR2 and clear the Carry flag.

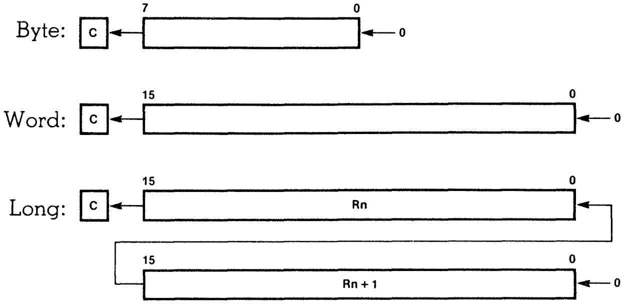
Note 1 b = number of bit positions, the execution time for b = 0 is the same as for b = 1

# SLL
## Shift Left Logical

| | |
|---|---|
| **SLL** dst, src | dst: R |
| **SLLB** | src: IM |
| **SLLL** | |

**Operation:**

```
Do src times:
        c ← dst (msb)
        dst (n + 1) ← dst (n) (for n = msb − 1 to 0)
        dst (0) ← 0
```



n = 0, 2, 4, ..., 14

The destination operand is shifted logically left by the number of bit positions specified by the source operand. For SLLB, the source is in the range 0 to 8; for SLL, the source is in the range 0 to 16; for SLLL, the source is in the range 0 to 32. The least significant bit of the destination is filled with 0, and the C flag is loaded from the most significant bit (msb) of the destination. This instruction performs an unsigned multiplication of the destination by a power of two. A shift of zero positions does not affect the destination; however, the flags are set according to the destination value. The setting of the carry bit is undefined for zero shift.

The src field is encoded in the instruction format as the 8- or 16-bit positive value of the source operand. For each operand size, the operation is undefined if the source operand is not in the specified range.

The source operand may be omitted from the assembly language statement and thus defaults to the value 1.

**Flags:**

**C:** Set if the last bit shifted from the destination was 1, undefined for zero shift; cleared otherwise
**Z:** Set if the result is zero; cleared otherwise
**S:** Set if the most significant bit of the result is set; cleared otherwise
**V:** Undefined
**D:** Unaffected
**H:** Unaffected

| Destination Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles[1] | Instruction Format | Cycles[1] |
| **R:** | SLL Rd, #b | 10 110011 Rd 0001<br>b | 13 + 3b | 10 110011 Rd 0001<br>b | 13 + 3 b |
| | SLLB Rbd, #b | 10 110010 Rbd 0001<br>0 b | 13 + 3b | 10 110010 Rbd 0001<br>0 b | 13 + 3 b |
| | SLLL RRd, #b | 10 110011 RRd 0101<br>b | 13 + 3b | 10 110011 RRd 0101<br>b | 13 + 3 b |

**Example:** If register R3 contains %4321 (0100001100100001), the statement

SLL R3,#1

will leave the value %8642 (1000011001000010) in R3 and clear the carry flag.

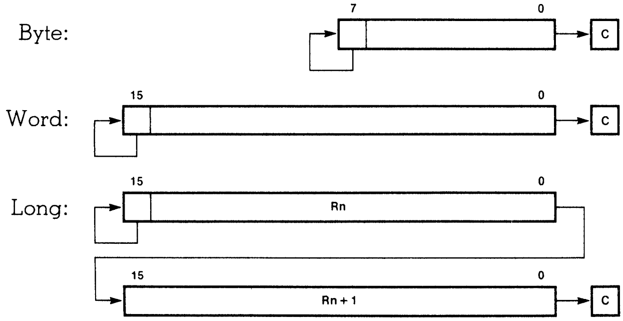
[1] b = number of bit positions, the execution time for b = 0 is the same as for b = 1

# SRA
## Shift Right Arithmetic

**SRA** dst, src　　dst: R
**SRAB**　　src: IM
**SRAL**

**Operation:**

```
Do src times:
        c ← dst (0)
        dst (n) ← dst (n + 1)(for n = 0 to msb − 1)
        dst (msb) ← dst (msb)
```



n = 0, 2, 4, ..., 14

The destination operand is shifted arithmetically right by the number of bit positions specified by the source operand. For SRAB, the source is in the range 1 to 8; for SRA, the source is in the range 1 to 16; for SRAL, the source is in the range 1 to 32. A right shift of zero for SRA is not possible. The most significant bit (msb) of the destination is replicated, and the C flag is loaded from bit 0 of the destination, this instruction performs a signed division of the destination by a power of two.

The src field is encoded in the instruction format as the 8- or 16-bit two's complement negative of the source operand. For each operand size, the operation is undefined if the source operand is not in the specified range.

The source operand may be omitted from the assembly language statement and thus defaults to the value 1.

**Flags:**

**C:** Set if the last bit shifted from the destination was 1; cleared otherwise
**Z:** Set if the result is zero; cleared otherwise
**S:** Set if the result is negative; cleared otherwise
**V:** Cleared
**D:** Unaffected
**H:** Unaffected

| Destination Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles[1] | Instruction Format | Cycles[1] |
| R: | SRA Rd, #b | 10 110011 Rd 1001<br>−b | 13+3b | 10 110011 Rd 1001<br>−b | 13+3b |
| | SRAB Rbd, #b | 10 110010 Rbd 1001<br>0 −b | 13+3b | 10 110010 Rbd 1001<br>0 −b | 13+3b |
| | SRAL RRd, #b | 10 110011 RRd 1101<br>−b | 13+3b | 10 110011 RRd 1101<br>−b | 13+3b |

**Example:** If register RH6 contains %3B (00111011), the statement

SRAB RH6,#2

will leave the value %0E (00001110) in RH6 and set the carry flag.

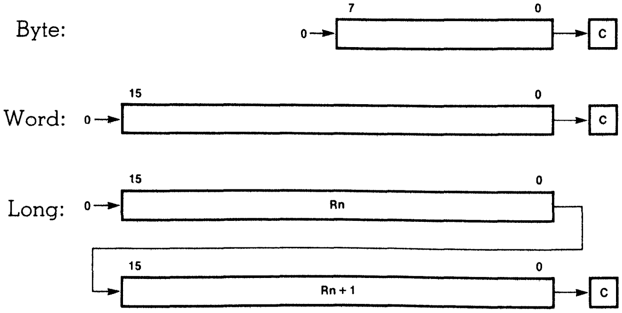
[1] b = number of bit positions, the execution time for b = 0 is the same as for b = 1

# SRL
## Shift Right Logical

| | |
|---|---|
| **SRL** dst, src | dst: R |
| **SRLB** | src: IM |
| **SRLL** | |

**Operation:**

```
Do src times:
        c ◄— dst (0)
        dst (n) ◄— dst (n + 1)(for n = 0 to msb – 1)
        dst (msb) ◄— 0
```



n = 0, 2, 4, ..., 14

The destination operand is shifted logically right by the number of bit positions specified by the source operand. For SRLB, the source operand is in the range 1 to 8; for SRL, the source is in the range 1 to 16; for SRLL, the source is in the range 1 to 32. A right shift of zero for SRL is not possible. The most significant bit (msb) of the destination is filled with 0, and the C flag is loaded from bit 0 of the destination. This instruction performs an unsigned division of the destination by a power of two.

The src field is encoded in the instruction format as the 8- or 16-bit negative value of the source operand in two's complement notation. For each operand size, the operation is undefined if the source operand is not in the range specified above.

The source operand may be omitted from the assembly language statement and thus defaults to the value of 1.

**Flags:**

**C:** Set if the last bit shifted from the destination was 1; cleared otherwise
**Z:** Set if the result is zero; cleared otherwise
**S:** Set if the most significant bit of the result is one; cleared otherwise
**V:** Undefined
**D:** Unaffected
**H:** Unaffected

| Destination Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles[1] | Instruction Format | Cycles[1] |
| R: | SRL Rd, #b | 10 110011 Rd 0001<br>−b | 13 + 3b | 10 110011 Rd 0001<br>−b | 13 + 3b |
| | SRLB Rbd, #b | 10 110010 Rbd 0001<br>0 −b | 13 + 3b | 10 110010 Rbd 0001<br>0 −b | 13 + 3b |
| | SRLL RRd, #b | 10 110011 RRd 0101<br>−b | 13 + 3b | 10 110011 RRd 0101<br>−b | 13 + 3b |

**Example:** If register R0 contains %1111 (0001000100010001), the statement

SRL R0,#6

will leave the value %0044 (0000000001000100) in R0 and clear the carry flag.

[1] Note 1 b = number of bit positions, the execution time for b = 0 is the same as for b = 1

# SUB
## Subtract

| | |
|---|---|
| **SUB** dst, src<br>**SUBB**<br>**SUBL** | dst: R<br>src: R, IM, IR, DA, X |

**Operation:** dst ← dst − src

The source operand is subtracted from the destination operand and the result is stored in the destination. The contents of the source are not affected. Subtraction is performed by adding the two's complement of the source operand to the destination operand.

**Flags:**

**C:** Cleared if there is a carry from the most significant bit; set otherwise, indicating a "borrow"
**Z:** Set if the result is zero; cleared otherwise
**S:** Set if the result is negative; cleared otherwise
**V:** Set if arithmetic overflow occurs, that is, if the operands were of opposite signs and the sign of the result is the same as the sign of the source; cleared otherwise
**D:** SUB, SUBL—unaffected; SUBB—set
**H:** SUB, SUBL—unaffected; SUBB—cleared if there is a carry from the most significant bit of the low-order four bits of the result; set otherwise, indicating a "borrow"

| Source Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles | Instruction Format | Cycles |
| **R:** | SUB Rd, Rs<br>SUBB Rbd, Rbs | 10 00001 W Rs Rd | 4 | 10 00001 W Rs Rd | 4 |
| | SUBL RRd, RRs | 10 010010 RRs RRd | 8 | 10 010010 RRs RRd | 8 |
| **IM:** | SUB Rd, #data | 00 000010 0000 Rd<br>data | 7 | 00 000010 0000 Rd<br>data | 7 |
| | SUBB Rbd, #data | 00 000011 0000 Rbd<br>data data | 7 | 00 000011 0000 Rbd<br>data data | 7 |
| | SUBL RRd, #data | 00 010010 0000 RRd<br>31 data (high) 16<br>15 data (low) 0 | 14 | 00 010010 0000 RRd<br>31 data (high) 16<br>15 data (low) 0 | 14 |
| **IR:** | SUB Rd, @Rs[1]<br>SUBB Rbd, @Rs[1] | 00 00001 W Rs≠0 Rd | 7 | 00 00001 W RRs≠0 Rd | 7 |
| | SUBL RRd, @Rs[1] | 00 010010 Rs≠0 RRd | 14 | 00 010010 RRs≠0 RRd | 14 |

| Source Addressing Mode | Assembler Language Syntax | Nonsegmented Mode |  | Segmented Mode |  |
|---|---|---|---|---|---|
|  |  | Instruction Format | Cycles | Instruction Format | Cycles |
| **DA:** | SUB Rd, address<br>SUBB Rbd, address | 01 00001 W 0000 Rd<br>address | 9 | SS 01 00001 W 0000 Rd<br>0 segment offset | 10 |
|  |  |  |  | SL 01 00001 W 0000 Rd<br>1 segment 0000 0000<br>offset | 12 |
|  | SUBL RRd, address | 01 010010 0000 RRd<br>address | 15 | SS 01 010010 0000 RRd<br>0 segment offset | 16 |
|  |  |  |  | SL 01 010010 0000 RRd<br>1 segment 0000 0000<br>offset | 18 |
| **X:** | SUB Rd, addr(Rs)<br>SUBB Rbd, addr(Rs) | 01 00001 W Rs≠0 Rd<br>address | 10 | SS 01 00001 W Rs≠0 Rd<br>0 segment offset | 10 |
|  |  |  |  | SL 01 00001 W Rs≠0 Rd<br>1 segment 0000 0000<br>offset | 13 |
|  | SUBL RRD, addr(Rs) | 01 010010 Rs≠0 RRd<br>address | 16 | SS 01 010010 Rs≠0 RRd<br>0 segment offset | 16 |
|  |  |  |  | SL 01 010010 Rs≠0 RRd<br>1 segment 0000 0000<br>offset | 19 |

**Example:** If register R0 contains %0344, the statement

SUB R0,#%AA

will leave the value %029A in R0.

Note 1 Word register in nonsegmented mode, register pair in segmented mode

# TCC
## Test Condition Code

**TCC** cc, dst  dst: R
**TCCB**

**Operation:**

```
if cc is satisfied then
    dst (0) ← 1
```

This instruction is used to create a Boolean data value based on the flags set by a previous operation. The flags in the FCW are tested to see if the condition specified by "cc" is satisfied. If the condition is satisfied, then the least significant bit of the destination is set. If the condition is not satisfied, bit zero of the destination is not cleared but retains its previous value. All other bits in the destination are unaffected by this instruction.

**Flags:** No flags affected

| Destination Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles | Instruction Format | Cycles |
| R: | TCC cc, Rd<br>TCCB cc, Rbd | 10 10111 W Rd cc | 5 | 10 10111 W Rd cc | 5 |

**Example:** If register R1 contains 0, and the Z flag is set, the statement

```
TCC   EQ,R1
```

will leave the value 1 in R1.

# TEST
## Test

**TEST** dst    dst: R, IR, DA, X
**TESTB**
**TESTL**

**Operation:** dst OR 0

The destination operand is tested (logically ORed with zero), and the Z, S and P flags are set to reflect the attributes of the result. The flags may then be used for logical conditional jumps. The contents of the destination are not affected.

**Flags:**
**C:** Unaffected
**Z:** Set if the result is zero; cleared otherwise
**S:** Set if the most significant bit of the result is set; cleared otherwise
**P:** TEST—unaffected; TESTL—undefined; TESTB—set if parity of the result is even; cleared otherwise
**D:** Unaffected
**H:** Unaffected

| Destination Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | | |
|---|---|---|---|---|---|---|
| | | Instruction Format | Cycles | | Instruction Format | Cycles |
| **R:** | TEST Rd<br>TESTB Rbd | 10 00110 W Rd 0100 | 7 | | 10 00110 W Rd 0100 | 7 |
| | TESTL RRd | 10 011100 RRd 1000 | 13 | | 10 011100 RRd 1000 | 13 |
| **IR:** | TEST @Rd[1]<br>TESTB @Rd[1] | 00 00110 W Rd ≠ 0 0100 | 8 | | 00 00110 W RRd≠0 0100 | 8 |
| | TESTL @Rd[1] | 00 011100 Rd ≠ 0 1000 | 13 | | 00 011100 RRd≠0 1000 | 13 |
| **DA:** | TEST address<br>TESTB address | 01 00110 W 0000 0100<br>address | 11 | SS | 01 00110 W 0000 0100<br>0 segment offset | 12 |
| | | | | SL | 01 00110 W 0000 0100<br>1 segment 0000 0000<br>address | 14 |
| | TESTL address | 01 011100 0000 1000<br>address | 16 | SS | 01 011100 0000 1000<br>0 segment offset | 17 |
| | | | | SL | 01 011100 0000 1000<br>1 segment 0000 0000<br>offset | 19 |

| Destination Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | | |
|---|---|---|---|---|---|---|
| | | Instruction Format | Cycles | | Instruction Format | Cycles |
| X: | TEST addr(Rd)<br>TESTB addr(Rd) | 01 00110 W Rd≠0 0100<br>address | 12 | SS | 01 00110 W Rd≠0 0100<br>0 segment offset | 12 |
| | | | | SL | 01 00110 W Rd≠0 0100<br>1 segment 0000 0000<br>offset | 15 |
| | | 01 011100 Rd≠0 1000<br>address | 17 | SS | 01 011100 Rd≠0 1000<br>0 segment offset | 17 |
| | | | | SL | 01 011100 Rd≠0 1000<br>1 segment 0000 0000<br>offset | 20 |

**Example:** If register R5 contains %FFFF (1111111111111111), the statement

TEST R5

will set the S flag, clear the Z flag, and leave the other flags unaffected.

Note 1 Word register in nonsegmented mode, register pair in segmented mode

# TRDB
## Translate and Decrement

**TRDB** dst, src, r

dst: IR
src: IR

**Operation:**

dst ← src[dst]
AUTODECREMENT dst by 1
r ← r − 1

This instruction is used to translate a string of bytes from one code to another code. The contents of the location addressed by the destination register (the "target byte") are used as an index into a table of translation values whose lowest address is contained in the source register. The index is computed by adding the target byte to the address contained in the source register. The addition is performed following the rule for address arithmetic, with the target byte treated as an unsigned 8-bit value extended with high-order zeros. The sum is used as the address of an 8-bit translation value within the table which replaces the original contents of the location addressed by the destination register.

The destination register is then decremented by one, thus moving the pointer to the previous element in the string. The word register specified by "r" (used as a counter) is then decremented by one. The original contents of register RH1 are lost and are replaced by an undefined value. R0 and R1 in nonsegmented mode, or RR0 in segmented mode, must not be used as a source or destination pointer, and R1 should not be used as a counter. The source, destination, and counter registers must be separate and non-overlapping registers.

Because the 8-bit target byte is added to the source register to obtain the address of a translation value, the table may contain 256 bytes. A smaller table size may be used where it is known that not all possible 8-bit target byte values will occur. The source register is unchanged.

**Flags:**

**C:** Unaffected
**Z:** Undefined
**S:** Unaffected
**V:** Set if the result of decrementing r is zero; cleared otherwise
**D:** Unaffected
**H:** Unaffected

| Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles | Instruction Format | Cycles |
| IR: | TRDB @Rd[1], @Rs[1], r | 10 111000 Rd ≠ 0 1000<br>0000 r Rs ≠ 0 0000 | 25 | 10 111000 RRd≠0 1000<br>0000 r RRs≠0 0000 | 25 |

**Example:**

In nonsegmented mode, if register R6 contains %4001, the byte at location %4001 contains 3, register R9 contains %1000, the byte at location %1003 contains %AA, and register R12 contains 2, the instruction

TRDB @R6, @R9, R12

will leave the value %AA in location %4001, the value %4000 in R6, and the value 1 in R12. R9 will not be affected. The V flag will be cleared. RH1 will be set to an undefined value. In segmented mode, R6 and R9 would be replaced with register pairs.

[1] Word register in nonsegmented mode, register pair in segmented mode

# TRDRB

## Translate, Decrement and Repeat

**TRDRB** dst, src, r — dst: IR<br>src: IR

| | |
|---|---|
| **TRDRB** dst, src, r | dst: IR<br>src: IR |

**Operation:**

```
dst ◄- src [dst]
AUTODECREMENT dst by 1
r ◄- r - 1
repeat until r = 0
```

This instruction is used to translate a string of bytes from one code to another code. The contents of the location addressed by the destination register (the "target byte") are used as an index into a table of translation values whose lowest address is contained in the source register. The index is computed by adding the target byte to the address contained in the source register. The addition is performed following the rules for address arithmetic, with the target byte treated as an unsigned 8-bit value extended with high-order zeros. The sum is used as the address of an 8-bit translation value within the table that replaces the original contents of the location addressed by the destination register.

The destination register is then decremented by one, thus moving the pointer to the previous element in the string. The word register specified by "r" (used as a counter) is then decremented by one. The entire operation is repeated until the result of decrementing r is zero. This instruction can translate from 1 to 65536 bytes. The original contents of register RH1 are lost and are replaced by an undefined value. The source register is unchanged. The source, destination, and counter registers must be separate and non-overlapping registers.

Because the 8-bit target byte is added to the source register to obtain the address of a translation value, the table may contain 256 bytes. A smaller table size may be used where it is known that not all possible 8-bit target byte values will occur.

This instruction can be interrupted after each execution of the basic operation. The program counter of the start of this instruction is saved before the interrupt request is accepted, so the instruction can be properly resumed. Seven cycles should be added to this instruction's execution time for each interrupt request that is accepted.

**Flags:**

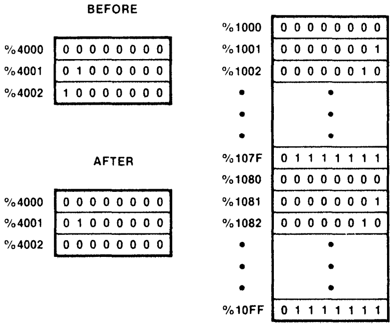
**C:** Unaffected
**Z:** Undefined
**S:** Unaffected
**V:** Set
**D:** Unaffected
**H:** Unaffected

| Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles[2] | Instruction Format | Cycles[2] |
| IR: | TRDRB @Rd[1], @Rs[1], r | 10 \| 111000 \| Rd ≠ 0 \| 1100<br>0000 \| r \| Rs ≠ 0 \| 0000 | 11 + 14n | 10 \| 111000 \| RRd≠0 \| 1100<br>0000 \| r \| RRs≠0 \| 0000 | 11 + 14n |

**Example:** In nonsegmented mode, if register R6 contains %4002, the bytes at locations %4000 through %4002 contain the values %00, %40, %80, respectively, register R9 contains %1000, the translation table from location %1000 through %10FF contains 0, 1, 2, ..., %7F, 0, 1, 2, ..., %7F (the second zero is located at %1080), and register R12 contains 3, the instruction

```
TRDRB   @R6, @R9, R12
```

will leave the values %00, %40, %00 in byte locations %4000 through %4002, respectively. Register R6 will contain %3FFF, and R12 will contain 0. R9 will not be affected. The V flag will be set, and the contents of RH1 will be replaced by an undefined value. In segmented mode, R6 and R9 would be replaced by register pairs.



Note 1· Word register in nonsegmented mode, register pair in segmented mode.

Note 2. n = number of data elements translated.

# TRIB
## Translate and Increment

**TRIB** dst, src, r

dst: IR
src: IR

**Operation:**

dst ← src[dst]
AUTOINCREMENT dst by 1
r ← r − 1

This instruction is used to translate a string of bytes from one code to another code. The contents of the location addressed by the destination register (the "target byte") are used as an index into a table of translation values whose lowest address is contained in the source register. The index is computed by adding the target byte to the address contained in the source register. The addition is performed following the rules for address arithmetic, with the target byte treated as an unsigned 8-bit value extended with high-order zeros. The sum is used as the address of an 8-bit translation value within the table which replaces the original contents of the location addressed by the destination register. The destination register is then incremented by one, thus moving the pointer to the next element in the string. The word register specified by "r" (used as a counter) is then decremented by one. The original contents of register RH1 are lost and are replaced by an undefined value. The source register is unchanged. The source, destination, and counter registers must be separate and non-overlapping registers.

Because the 8-bit target byte is added to the source register to obtain the address of a translation value, the table may contain 256 bytes. A smaller table size may be used where it is known that not all possible 8-bit target byte values will occur.

**Flags:**

**C:** Unaffected
**Z:** Undefined
**S:** Unaffected
**V:** Set if the result of decrementing r is zero; cleared otherwise
**D:** Unaffected
**H:** Unaffected

| Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles | Instruction Format | Cycles |
| | TRIB @Rd[1], @Rs[1], r | 10 111000 Rd ≠ 0 0000<br>0000 r Rs ≠ 0 0000 | 25 | 10 111000 RRd≠0 0000<br>0000 r RRs≠0 0000 | 25 |

**Example:** This instruction can be used in a "loop" of instructions which translate a string of data from one code to any other desired code, but an intermediate operation on each data element is required. The following sequence translates a string of 1000 bytes to the same string of bytes, with all ASCII "control characters" (values less than 32, see Appendix C) translated to the "blank" character (value = 32). A test, however, is made for the special character "return" (value = 13) which terminates the loop. The translation table contains 256 bytes. The first 33 (0-32) entries all contain the value 32, and all other entries contain their own index in the table, counting from zero. This example assumes nonsegmented mode. In segmented mode, R4 and R5 would be replaced by register pairs.

```
         LD      R3, #1000        !initialize counter!
         LDA     R4, STRING       !load start addresses!
         LDA     R5, TABLE
LOOP:
         CPB     @R4, #13         !check for return character!
         JR      EQ, DONE         !exit loop if found!
         TRIB    @R4, @R5, R3     !translate next byte!
         JR      NOV, LOOP        !repeat until counter = 0!
DONE:
```

| | |
|---|---|
| TABLE + 0 | 0 0 1 0 0 0 0 0 |
| TABLE + 1 | 0 0 1 0 0 0 0 0 |
| TABLE + 2 | 0 0 1 0 0 0 0 0 |
| •<br>•<br>• | •<br>•<br>• |
| TABLE + 32 | 0 0 1 0 0 0 0 0 |
| TABLE + 33 | 0 0 1 0 0 0 0 1 |
| TABLE + 34 | 0 0 1 0 0 0 1 0 |
| •<br>•<br>• | •<br>•<br>• |
| TABLE + 255 | 1 1 1 1 1 1 1 1 |

Note 1 Word register in nonsegmented mode, register pair in segmented mode

# TRIRB
## Translate, Increment and Repeat

**TRIRB** dst, src, r

dst: IR
src: IR

**Operation:**

```
dst ← src[dst]
AUTOINCREMENT dst by 1
r ← r − 1
repeat until r = 0
```

This instruction is used to translate a string of bytes from one code to another code. The contents of the location addressed by the destination register (the "target byte") are used as an index into a table of translation values whose lowest address is contained in the source register. The index is computed by adding the target byte to the address contained in the source register. The addition is performed following the rules for address arithmetic, with the target byte treated as an unsigned 8-bit value extended with high-order zeros. The sum is used as the address of an 8-bit translation value within the table which replaces the original contents of the location addressed by the destination register. The destination register is then incremented by one, thus moving the pointer to the next element in the string. The word register specified by "r" (used as a counter) is then decremented by one. The entire operation is repeated until the result of decrementing r is zero. This instruction can translate from 1 to 65536 bytes. The original contents of register RH1 are lost and are replaced by an undefined value. The source register is unaffected. The source, destination, and counter registers must be separate and non-overlapping registers.

Because the 8-bit target byte is added to the source register to obtain the address of a translation value, the table may contain 256 bytes. A smaller table size may be used where it is known that not all possible 8-bit target byte values will occur.

This instruction can be interrupted after each execution of the basic operation. The program counter of the start of this instruction is saved before the interrupt request is accepted so that the instruction can be properly resumed. Seven cycles should be added to this instruction's execution time for each interrupt request that is accepted.

**Flags:**

**C:** Unaffected
**Z:** Undefined
**S:** Unaffected
**V:** Set
**D:** Unaffected
**H:** Unaffected

| Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles[2] | Instruction Format | Cycles[2] |
| **IR:** | TRIRB @Rd[1], @Rs[1], r | 10 111000 Rd ≠ 0 0100<br>0000 r Rs ≠ 0 0000 | 11 + 14n | 10 111000 RRd≠0 0100<br>0000 r RRs≠0 0000 | 11 + 14n |

**Example:**

The following sequence of instructions can be used to translate a string of 80 bytes from one code to another. The pointers to the string and the translation table are set, the number of bytes to translate is set, and then the translation is accomplished. After executing the last instruction, the V flag is set and the contents of RH1 are lost. The example assumes nonsegmented mode. In segmented mode, R4 and R5 would be replaced by register pairs.

```
LDA    R4, STRING
LDA    R5, TABLE
LD     R3, #80
TRIRB  @R4, @R5, R3
```

Note 1 Word register in nonsegmented mode, register pair in segmented mode

Note 2 n = number of data elements translated

# TRTDB

## Translate, Test and Decrement

**TRTDB** srcl, src2, r

src 1: IR
src 2: IR

**Operation:**

```
RH1 ← src2[srcl]
AUTODECREMENT srcl by 1
r ← r − 1
```

This instruction is used to scan a string of bytes testing for bytes with special meaning. The contents of the location addressed by the first source register (the "target byte") are used as an index into a table of translation values whose lowest address is contained in the second source register. The index is computed by adding the target byte to the address contained in the second source register. The addition is performed following the rules for address arithmetic, with the target byte treated as an unsigned 8-bit value extended with high-order zeros. The sum is used as the address of an 8-bit value within the table which is loaded into register RH1. The Z flag is set if the value loaded into RH1 is zero; otherwise the Z flag is cleared. The contents of the locations addressed by the source registers are not affected. The first source register is then decremented by one, thus moving the pointer to the previous element in the string. The word register specified by "r" (used as a counter) is then decremented by one. The second source register is unaffected. The source and counter registers must be separate and non-overlapping registers.

Because the 8-bit target byte is added to the second source register to obtain the address of a translation value, the table may contain 256 bytes. A smaller table size may be used where it is known that not all possible 8-bit target byte values will occur.

**Flags:**

**C:** Unaffected
**Z:** Set if the translation value loaded into RH1 is zero; cleared otherwise
**S:** Unaffected
**V:** Set if the result of decrementing r is zero; cleared otherwise
**D:** Unaffected
**H:** Unaffected

| Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles | Instruction Format | Cycles |
| IR: | TRTDB @Rsl[1], @Rs2[1], r | 10 \| 111000 \| Rs1 ≠ 0 \| 1010<br>0000 \| r \| Rs2 ≠ 0 \| 0000 | 25 | 10 \| 111000 \| RRs1≠0 \| 1010<br>0000 \| r \| RRs2≠0 \| 0000 | 25 |

**Example:**

In nonsegmented mode, if register R6 contains %4001, the byte at location %4001 contains 3, register R9 contains %1000, the byte at location %1003 contains %AA, and register R12 contains 2, the instruction

```
TRTDB  @R6, @R9, R12
```

Will leave the value %AA in RH1, the value %4000 in R6, and the value 1 in R12. Location %4001 and register R9 will not be affected. The Z and V flags will be cleared. In segmented mode, register pairs must be used instead of R6 and R9.

[1] Note 1 Word register in nonsegmented mode, register pair in segmented mode

# TRTDRB

## Translate, Test, Decrement and Repeat

**TRTDRB** src1, src2, r      src1: IR
src2: IR

**Operation:**

```
RH1 ← src 2[src1]
AUTODECREMENT src1 by 1
r ← r − 1
repeat until RH1 ≠ 0 or r = 0
```

This instruction is used to scan a string of bytes testing for bytes with special meaning. The contents of the location addressed by the first source register (the "target byte") are used as an index into a table of translation values whose lowest address is contained in the second source register. The index is computed by adding the target byte to the address contained in the second source register. The addition is performed following the rules for address arithmetic, with the target byte treated as an unsigned 8-bit value extended with high-order zeros. The sum is used as the address of an 8-bit value within the table which is loaded into register RH1. The Z flag is set if the value loaded into RH1 is zero; otherwise the Z flag is cleared. The contents of the locations addressed by the source registers are not affected. The first source register is then decremented by one, thus moving the pointer to the previous element in the string. The word register specified by "r" (used as a counter) is then decremented by one. The entire operation is repeated until either the Z flag is clear, indicating that a non-zero translation value was loaded into RH1, or until the result of decrementing r is zero. This instruction can translate and test from 1 to 65536 bytes. The source and counter registers must be separate and non-overlapping registers.

Target byte values which have corresponding zero translation-table entry values are to be scanned over, while target byte values which have corresponding non-zero translation-table entry values are to be detected. Because the 8-bit target byte is added to the second source register to obtain the address of a translation value, the table may contain 256 bytes. A smaller table size may be used where it is known that not all possible 8-bit target byte values will occur.

This instruction can be interrupted after each execution of the basic operation. The program counter of the start of this instruction is saved before the interrupt request is accepted so that the instruction can be properly resumed. Seven cycles should be added to this instruction's execution time for each interrupt request that is accepted.

**Flags:**

**C:** Unaffected
**Z:** Set if the translation value loaded into RH1 is zero; cleared otherwise
**S:** Unaffected
**V:** Set if the result of decrementing r is zero; cleared otherwise
**D:** Unaffected
**H:** Unaffected

| Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles[2] | Instruction Format | Cycles[2] |
| IR: | TRTDRB@Rs1[1],@Rs2[1],r | 10 111000 Rs1 ≠ 0 1110<br>0000 r Rs2 ≠ 0 1110 | 11 + 14n | 10 111000 RRs1≠0 1110<br>0000 r RRs2≠0 1110 | 11 + 14n |

**Example:** In nonsegmented mode, if register R6 contains %4002, the bytes at locations %4000 through %4002 contain the values %00, %40, %80, repectively, register R9 contains %1000, the translation table from location %1000 through %10FF contains 0, 1, 2, ..., %7F, 0, 1, 2, ..., %7F (the second zero is located at %1080), and register R12 contains 3, the instruction

```
TRTDRB  @R6, @R9, R12
```

will leave the value %40 in RH1 (which was loaded from location %1040). Register R6 will contain %4000, and R12 will contain 1. R9 will not be affected. The Z and V flags will be cleared. In segmented mode, register pairs are used instead of R6 and R9.

| | |
|---|---|
| %4000 | 0 0 0 0 0 0 0 0 |
| %4001 | 0 1 0 0 0 0 0 0 |
| %4002 | 1 0 0 0 0 0 0 0 |

| | |
|---|---|
| %1000 | 0 0 0 0 0 0 0 0 |
| %1001 | 0 0 0 0 0 0 0 1 |
| %1002 | 0 0 0 0 0 0 1 0 |
| • | • |
| • | • |
| • | • |
| %107F | 0 1 1 1 1 1 1 1 |
| %1080 | 0 0 0 0 0 0 0 0 |
| %1081 | 0 0 0 0 0 0 0 1 |
| %1082 | 0 0 0 0 0 0 1 0 |
| • | • |
| • | • |
| • | • |
| %10FF | 0 1 1 1 1 1 1 1 |

Note 1. Word register in nonsegmented mode, register pair in segmented mode.

Note 2. n = number of data elements translated

# TRTIB
## Translate, Test and Increment

**TRTIB** src1, src2, r

src1: IR
src2: IR

**Operation:**

```
RH1 ← src2[src1]
AUTOINCREMENT src1 by 1
r ← r − 1
```

This instruction is used to scan a string of bytes testing for bytes with special meaning. The contents of the location addressed by the first source register (the "target byte") are used as an index into a table of translation values whose lowest address is contained in the second source register. The index is computed by adding the target byte to the address contained in the second source register. The addition is performed following the rules for address arithmetic, with the target byte treated as an unsigned 8-bit value extended with high-order zeros. The sum is used as the address of an 8-bit value within the table which is loaded into register RH1. The Z flag is set if the value loaded into RH1 is zero; otherwise the Z flag is cleared. The contents of the locations addressed by the source registers are not affected. The first source register is then incremented by one, thus moving the pointer to the next element in the string. The word register specified by "r" (used as a counter) is then decremented by one. The second source register is unaffected. The source and counter registers must be separate and non-overlapping registers.

Because the 8-bit target byte is added to the second source register to obtain the address of a translation value, the table may contain 256 bytes. A smaller table size may be used where it is known that not all possible 8-bit target byte values will occur.

**Flags:**

**C:** Unaffected
**Z:** Set if the translation value loaded into RH1 is zero; cleared otherwise
**S:** Unaffected
**V:** Set if the result of decrementing r is zero; cleared otherwise
**D:** Unaffected
**H:** Unaffected

| Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles | Instruction Format | Cycles |
| IR: | TRTIB @Rs1[1], @Rs2[1], r | 10 \| 111000 \| Rs1 ≠ 0 \| 0010<br>0000 \| r \| Rs2 ≠ 0 \| 0000 | 25 | 10 \| 111000 \| RRs1≠0 \| 0010<br>0000 \| r \| RRs2≠0 \| 0000 | 25 |

**Example:**

This instruction can be used in a "loop" of instructions which translate and test a string of data, but an intermediate operation on each data element is required. The following sequence outputs a string of 72 bytes, with each byte of the original string translated from its 7-bit ASCII code to an 8-bit value with odd parity. Lower case characters are translated to upper case, and any embedded control characters are skipped over. The translation table contains 128 bytes, which assumes that the most significant bit of each byte in the string to be translated is always zero. The first 32 entries and the 128th entry are zero, so that ASCII control characters and the "delete" character (%7F) are suppressed. The given instruction sequence is for nonsegmented mode. In segmented mode, register pairs would be used instead of R3 and R4.

```
        LD      R5, #72           !initialize counter!
        LDA     R3, STRING        !load start address!
        LDA     R4, TABLE
LOOP:
        TRTIB   @R3, @R4, R5      !translate and test next byte!
        JR      Z, LOOP           !skip control character!
        OUTB    PORTn, RH1        !output characters!
        JR      NOV, LOOP         !repeat until counter = 0!
DONE:
```

Note 1 Word register in nonsegmented mode, register pair in segmented mode

# TRTIRB

## Translate, Test, Increment and Repeat

**TRTIRB** src1, src2, r

src1: IR
src2: IR

**Operation:**

```
RH1 ← src2[src1]
AUTOINCREMENT src1 by 1
r ← r − 1
repeat until RH1 ≠ 0 or r = 0
```

This instruction is used to scan a string of bytes, testing for bytes with special meaning. The contents of the location addressed by the first source register (the "target byte") are used as an index into a table of translation values whose lowest address is contained in the second source register. The index is computed by adding the target byte to the address contained in the second source register. The addition is performed following the rules for address arithmetic, with the target byte treated as an unsigned 8-bit value extended with high-order zeros. The sum is used as the address of an 8-bit value within the table which is loaded into register RH1. The Z flag is set if the value loaded into RH1 is zero; otherwise the Z flag is cleared. The contents of the locations addressed by the source registers are not affected.

The first source register is then incremented by one, thus moving the pointer to the next element in the string. The word register specified by "r" (used as a counter) is then decremented by one. The entire operation is repeated until either the Z flag is clear, indicating that a non-zero translation value was loaded into RH1, or until the result of decrementing r is zero. This instruction can translate and test from 1 to 65536 bytes. The source and counter registers must be separate and non-overlapping registers.

Target byte values which have corresponding zero translation table entry values are scanned over, while target byte values which have corresponding non-zero translation table entry values are detected and terminate the scan. Because the 8-bit target byte is added to the second source register to obtain the address of a translation value, the table may contain 256 bytes. A smaller table size may be used where it is known that not all possible 8-bit target byte values will occur.

This instruction can be interrupted after each execution of the basic operation. The program counter of the start of this instruction is saved before the interrupt request is accepted, so that the instruction can be properly resumed. Seven cycles should be added to this instruction's execution time for each interrupt request that is accepted.

**Flags:**

**C:** Unaffected
**Z:** Set if the translation value loaded into RH1 is zero; cleared otherwise
**S:** Unaffected
**V:** Set if the result of decrementing r is zero; cleared otherwise
**D:** Unaffected
**H:** Unaffected

| Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles[2] | Instruction Format | Cycles[2] |
| **IR:** | TRTIRB @Rs1[1], @Rs2[1], r | 10 \| 111000 \| Rs1 ≠ 0 \| 0110<br>0000 \| r \| Rs2 ≠ 0 \| 1110 | 11 + 14n | 10 \| 111000 \| RRs1≠0 \| 0110<br>0000 \| r \| RRs2≠0 \| 1110 | 11 + 14n |

**Example:**

The following sequence of instructions can be used in nonsegmented mode to scan a string of 80 bytes, testing for special characters as defined by corresponding non-zero translation table entry values. The pointers to the string and translation table are set, the number of bytes to scan is set, and then the translation and testing is done. The Z and V flags can be tested after the operation to determine if a special character was found and whether the end of the string has been reached. The translation value loaded into RH1 might then be used to index another table, or to select one of a set of sequences of instructions to execute next. In segmented mode, R4 and R5 must be replaced with register pairs.

```
                LDA       R4, STRING
                LDA       R5, TABLE
                LD        R6, #80
                TRTIRB    @R4, @R5, R6
                JR        NZ, SPECIAL
END__OF__STRING:           .
                          .
                          .
SPECIAL:
                JR        OV,LAST__CHAR__SPECIAL
                          .
                          .
                          .
LAST__CHAR__SPECIAL:
```

Note 1 Word register in nonsegmented mode, register pair in segmented mode

Note 2 n = number of data elements translated

**TSET** dst dst: R, IR, DA, X
**TSETB**

**Operation:**

S ← dst(msb)
dst(0:msb) ← 111...111

This instruction tests the most significant bit of the destination operand, copying its value into the S flag, then sets the entire destination to all 1 bits. It provides a locking mechanism which can be used to synchronize software processes which require exclusive access to certain data or instructions at one time.

During the execution of this instruction, $\overline{\text{BUSREQ}}$ is not honored in the time between loading the destination from memory and storing the destination to memory. For systems with one processor, this ensures that the testing and setting of the destination will be completed without any intervening accesses. To synchronize software processes residing on separate processors where the destination is a shared memory location, TSET should be used with a Z8003 CPU.

**Flags:**

**C:** Unaffected
**Z:** Unaffected
**S:** Set if the most significant bit of the destination was 1; cleared otherwise
**V:** Unaffected
**D:** Unaffected
**H:** Unaffected

| Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles | Instruction Format | Cycles |
| **R:** | TSET Rd<br>TSETB Rbd | 10 00110 W Rd 0110 | 7 | 10 00110 W Rd 0110 | 7 |
| **IR:** | TSET @Rd[1]<br>TSETB @Rd[1] | 00 00110 W Rd≠0 0110 | 11 | 00 00110 W RRd≠0 0110 | 11 |
| **DA:** | TSET address<br>TSETB address | 01 00110 W 0000 0110<br>address | 14 | SS 01 00110 W 0000 0110<br>0 segment offset | 15 |
| | | | | SL 01 00110 W 0000 0110<br>1 segment 0000 0000<br>offset | 17 |
| **X:** | TSET addr(Rd)<br>TSETB addr(Rd) | 01 00110 W Rd≠0 0110<br>address | 15 | SS 01 00110 W Rd≠0 0110<br>0 segment offset | 15 |
| | | | | SL 01 00110 W Rd≠0 0110<br>1 segment 0000 0000<br>offset | 18 |

**Example:** A simple mutually-exclusive critical region may be implemented by the following sequence of statements:

```
ENTER:
TSET      SEMAPHORE
JR        MI,ENTER            !loop until resource con-!
              .               !trolled by SEMAPHORE!
              .               !is available!
              .
!Critical Region—only one software process!
!executes this code at a time!
              .
              .
              .
CLR       SEMAPHORE           !release resource controlled!
                              !by SEMAPHORE!
```

# XOR
## Exclusive Or

**XOR** dst, src  
**XORB**

dst: R  
src: R, IM, IR, DA, X

**Operation:** dst ◄– dst XOR src

The source operand is logically EXCLUSIVE ORed with the destination operand and the result is stored in the destination. The contents of the source are not affected. The EXCLUSIVE OR operation results in a one bit being stored whenever the corresponding bits in the two operands are different; otherwise, a zero bit is stored.

**Flags:**

**C:** Unaffected  
**Z:** Set if the result is zero; cleared otherwise  
**S:** Set if the most significant bit of the result is set; cleared otherwise  
**P:** XOR—unaffected; XORB—set if parity of the result is even; cleared otherwise  
**D:** Unaffected  
**H:** Unaffected

| Source Addressing Mode | Assembler Language Syntax | Nonsegmented Mode | | Segmented Mode | | |
|---|---|---|---|---|---|---|
| | | Instruction Format | Cycles | | Instruction Format | Cycles |
| **R:** | XOR Rd, Rs<br>XORB Rbd, Rbs | 10 00100 W Rs Rd | 4 | | 10 00100 W Rs Rd | 4 |
| **IM:** | XOR Rd, #data | 00 001 001 0000 Rd<br>data | 7 | | 00 001 001 0000 Rd<br>data | 7 |
| | XORB Rbd, #data | 00 001000 0000 Rbd<br>data data | 7 | | 00 001000 0000 Rbd<br>data data | 7 |
| **IR:** | XOR Rd, @Rs[1]<br>XORB Rbd, @Rs[1] | 00 00100 W Rs≠0 Rd | 7 | | 00 00100 W Rs≠0 RRd | 7 |
| **DA:** | XOR Rd, address<br>XORB Rbd, address | 01 00100 W 0000 Rd<br>address | 9 | SS | 01 00100 W 0000 Rd<br>0 segment offset | 10 |
| | | | | SL | 01 00100 W 0000 Rd<br>1 segment 0000 0000<br>offset | 12 |
| **X:** | XOR Rd, addr(Rs)<br>XORB Rbd, addr(Rs) | 01 00100 W Rs≠0 Rd<br>address | 10 | SS | 01 00100 W Rs≠0 Rd<br>0 segment offset | 10 |
| | | | | SL | 01 00100 W Rs≠0 Rd<br>1 segment 0000 0000<br>offset | 13 |

**Example:** If register RL3 contains %C3 (11000011) and the source operand is the immediate value %7B (01111011), the statement

```
XORB  RL3,#%7B
```

will leave the value %B8 (10111000) in RL3.

Note 1 Word register in nonsegmented mode, register pair in segmented mode

## 6.8 EPA Instruction Templates

There are seven "templates" for EPA instructions. These templates correspond to EPA instructions, which combine EPU operations with possible transfers between memory and an EPU, between CPU registers and EPU registers, and between the Flag byte of the CPU's FCW and the EPU. Each of these templates is described on the following pages. The description assumes that the EPA control bit in the CPU's FCW has been set to 1. In addition, the description is from the point of view of the CPU--that is, only CPU activities are described; the operation of the EPU is implied, but the full specification of the instruction depends upon the implementation of the EPU and is beyond the scope of this manual.

Fields ignored by the CPU are shaded in the diagrams of the templates. The 2-bit field in bit positions 0 and 1 of the first word of each template would normally be used as an identification field for selecting one of up to four EPUs in a multiple EPU system configuration. Other shaded fields would typically contain opcodes for instructing an EPU as to the operation it is to perform in addition to the data transfer specified by the template.

# Extended Instruction
## Load Memory from EPU

**Operation:** Memory ← EPU

The CPU performs the indicated address calculation and generates n EPU memory write transactions. The *n* words are supplied by an EPU and are stored in *n* consecutive memory locations starting with the effective address and increasing in address.

**Flags/Registers:** No flags or CPU registers are affected by this instruction.

| Source Addressing Mode | Operation | Nonsegmented Mode |  | Segmented Mode |  |  |
|---|---|---|---|---|---|---|
|  |  | Instruction Format | Cycles |  | Instruction Format | Cycles |
| **IR:** | @ Rd ← EPU | 00 \| 001111 \| Rd≠0 \| 11 \| ▨<br>▨ \| n-1 | 11 + 3n |  | 00 \| 00 1111 \| RRd≠0 \| 11 \| ▨<br>▨ \| n-1 | 11 + 3n |
| **DA:** | EPU ← address | 01 \| 001111 \| 0000 \| 11 \| ▨<br>▨ \| n-1<br>address | 14 + 3n | SS | 01 \| 001111 \| 0000 \| 11 \| ▨<br>▨ \| n-1<br>0 \| segment \| offset | 15 + 3n |
|  |  |  |  | SL | 01 \| 001111 \| 0000 \| 11 \| ▨<br>▨ \| n-1<br>1 \| segment \| 00000000<br>offset | 17 + 3n |
| **X:** | EPU ← addr (Rs) | 01 \| 001111 \| Rd≠0 \| 11 \| ▨<br>▨ \| n-1<br>address | 15 + 3n | SS | 01 \| 001111 \| Rd≠0 \| 11 \| ▨<br>▨ \| n-1<br>0 \| segment \| offset | 15 + 3n |
|  |  |  |  | SL | 01 \| 001111 \| Rd≠0 \| 11 \| ▨<br>▨ \| n-1<br>1 \| segment \| 00000000<br>offset | 18 + 3n |

# Extended Instruction
## Load EPU from Memory

**Operation:** EPU ← Memory

The CPU performs the indicated address calculation and generates *n* EPU memory read transactions. The *n* consecutive words are fetched from the memory locations starting with the effective address. The data is read by an EPU and operated upon according to the EPA instruction encoded into the shaded fields.

**Flags/Registers:** No flags or CPU registers are affected by this instruction.

| Source Addressing Mode | Operation | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles | Instruction Format | Cycles |
| IR: | EPU ← @ Rs | 00 001111 Rs≠0 01 ▨<br>▨ n-1 | 11 + 3n | 00 00 1111 RRs≠0 01 ▨<br>▨ n-1 | 11 + 3n |
| DA: | EPU ← address | 01 001111 0000 01 ▨<br>▨ n-1<br>address | 14 + 3n | SS<br>01 001111 0000 01 ▨<br>▨ n-1<br>0 segment offset | 15 + 3n |
| | | | | SL<br>01 001111 0000 01 ▨<br>▨ n-1<br>1 segment 00000000<br>offset | 17 + 3n |
| X: | EPU ← addr (Rs) | 01 001111 Rs≠0 01 ▨<br>▨ n-1<br>address | 15 + 3n | SS<br>01 001111 Rs≠0 01 ▨<br>▨ n-1<br>0 segment offset | 15 + 3n |
| | | | | SL<br>01 001111 Rs≠0 01 ▨<br>▨ n-1<br>1 segment 00000000<br>offset | 18 + 3n |

# Extended Instruction
## Load CPU from EPU

**Operation:** CPU ← EPU registers

The contents of *n* words are transferred from an EPU to consecutive CPU registers starting with register dst. CPU registers are loaded consecutively, with register 0 following register 15.

**Flags/Registers:** No flags are affected by this instruction.

| Source Addressing Mode | Operation | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles | Instruction Format | Cycles |
| **R:** | Rd ← EPU | 10 \| 001111 \| 0 \| ▨ \| 00 \| ▨<br>▨ \| dst \| ▨ \| n-1 | 11 + 4n | 10 \| 001111 \| 0 \| ▨ \| 00 \| ▨<br>▨ \| dst \| ▨ \| n-1 | 11 + 4n |

# Extended Instruction
## Load EPU from CPU

**Operation:** EPU ← CPU registers

The contents of *n* words are transferred to an EPU from consecutive CPU registers starting with register src. CPU registers are transferred consecutively, with register 0 following register 15.

**Flags/Registers:** No flags are affected by this instruction.

| Source Addressing Mode | Operation | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles | Instruction Format | Cycles |
| **R:** | EPU ← Rs | 10 \| 001111 \| 0 \| ▨ \| 10 \| ▨<br>▨ \| src \| ▨ \| n-1 | 11 + 4n | 10 \| 001111 \| 0 \| ▨ \| 10 \| ▨<br>▨ \| src \| ▨ \| n-1 | 11 + 4n |

# Extended Instruction
## Load FCW from EPU

**Operation:** Flags ← EPU

The Flags in the CPU's Flag and Control Word are loaded with information from an EPU on AD lines $AD_0$-$AD_7$.

**Flags/Registers:** The contents of CPU register 0 are undefined after the execution of this instruction.

| Source Addressing Mode | Operation | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles | Instruction Format | Cycles |
| R: | FCW ← EPU | 10 001110 ▨ 00 ▨<br>▨ 0000 ▨ 0000 | 15 | 10 001111 ▨ 00 ▨<br>▨ 0000 ▨ 0000 | 15 |

# Extended Instruction
## Load EPU from FCW

**Operation:** EPU ← Flags

The Flags in the CPU's Flag and Control Word are transferred to an EPU on AD lines $AD_0$-$AD_7$.

**Flags/Registers:** The flags in the FCW are unaffected by this instruction.

| Source Addressing Mode | Operation | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles | Instruction Format | Cycles |
| R: | EPU ← FCW | 10 001110 ▨ 10 ▨<br>▨ 0000 ▨ 0000 | 15 | 10 001110 ▨ 10 ▨<br>▨ 0000 ▨ 0000 | 15 |

# Extended Instruction
## Internal EPU Operation

**Operation:** Internal EPU Operation

The CPU treats this template as a No Op. It is typically used to initiate an internal EPU operation.

**Flags/Registers:** No flags or registers are affected.

| Source Addressing Mode | Operation | Nonsegmented Mode | | Segmented Mode | |
|---|---|---|---|---|---|
| | | Instruction Format | Cycles | Instruction Format | Cycles |
| | EPU INTERNAL OPERATION | 10 \| 001110 \| ▨ \| 01 \| ▨<br>▨ \| ▨ \| n-1 | 11 + 4n | 10 \| 001110 \| ▨ \| 01 \| ▨<br>▨ \| ▨ \| n-1 | 11 + 4n |

# Chapter 7
# Exceptions

## 7.1 INTRODUCTION

Exceptions are conditions that can alter the normal flow of program execution. The Z8000 CPU supports four types of exception:

- interrupts
- traps
- abort (Z8003 and Z8004 only)
- reset

Interrupts are triggered by peripheral devices that need attention. They cause the processor to suspend program execution in order to service the requesting device. Traps are responses by the CPU to certain events detected during the attempted execution of an instruction. Thus, the difference between traps and interrupts is their origin. A trap condition is always reproducible by re-executing the program that created the traps, whereas an interrupt is generally independent of the currently executing task. Abort is a special trap-like exception that is used in the implementation of virtual memory systems. An abort exception is controlled by external memory management circuitry or peripheral device such as the Z8015 PMMU. If the CPU in a virtual system outputs an address which does not correspond to any location in main memory, an Abort Instruction function is initiated in the CPU (see Section 7.4). A reset overrides all other conditions, including all interrupts and traps. It occurs when the $\overline{\text{RESET}}$ line is activated, and it causes certain control registers to be initialized.

## 7.2 INTERRUPTS

Three kinds of interrupt are activated by three different pins on the Z8000 CPU. (Interrupt handling for all interrupts is discussed in Section 7.6).

### 7.2.1 Nonmaskable Interrupt ($\overline{\text{NMI}}$)

This type of interrupt cannot be disabled (masked) by software. It is typically reserved for external events that require immediate attention.

### 7.2.2 Vectored Interrupt ($\overline{\text{VI}}$)

One result of any interrupt or trap is that a 16-bit identifier word is pushed onto the system stack (see Section 7.6.2). This word can be used to identify the source of the interrupt or trap. For vectored and non-vectored interrupts, this identifier is supplied by the interrupting device and read from the bus by the CPU during the interrupt acknowledge cycle. For vectored interrupts, the low order byte of this identifier is used by the CPU hardware as an index to a table of interrupt service routine addresses. These interrupts can be disabled by clearing the VIE bit in the FCW.

### 7.2.3 Nonvectored Interrupts ($\overline{\text{NVI}}$)

These interrupts also result in an identifier word being pushed onto the system stack, however, the CPU does not use the identifier as a vector to select a service routine; all nonvectored interrupts are serviced by the same routine. They can be disabled by clearing the NVIE bit in the FCW.

## 7.3 TRAPS

All Z8000 CPUs support three traps generated internally. The Z8001 and Z8003 CPUs support a fourth trap, which is controlled externally (but synchronously) by either a Zilog MMU or external memory management circuitry. This fourth trap is intended for use in virtual memory systems to report access violations and page faults to the CPU.

Traps cannot be disabled. (Trap handling operations are discussed in Section 7.7).

### 7.3.1 Extended Instruction Trap

This trap occurs when the CPU encounters an extended instruction (see Section 6.2.10) while the EPA bit in the FCW is cleared. One of the major uses of this trap is to allow the program to simulate the operations of the EPU when none is present in the system.

### 7.3.2 Privileged Instruction Trap

This trap occurs whenever an attempt is made to execute a privileged instruction while the CPU is in System mode ($S/\overline{N}$ bit in the FCW is cleared).

### 7.3.3 System Call Trap

This trap occurs whenever a System Call (SC) instruction is executed. It allows an orderly transition to be made from Normal mode to System mode.

### 7.3.4 Segment/Address Violation Trap (Z8001 and Z8003 only)

There are two types of address violation traps both of which are controlled by an input (or inputs) from memory management hardware external to the CPU.

A Segment Trap is controlled by CPU input $\overline{\mathrm{SEGT}}$ for Z8001 and $\overline{\mathrm{SAT}}$ for Z8003. This trap is initiated by the external memory management circuitry. Violations which enable this trap include the detection of an address offset value which is larger than the length of the assigned segment, a write warning (a write into the lowest 256 byte section of a stack was detected), and violations of segment or page attributes (refer to MMU and PMMU descriptions in Appendix B).

### 7.3.5 Abort Trap (Z8003 and Z8004 only).

An Abort trap is initiated in the CPU by the assertion of the CPU $\overline{\mathrm{ABORT}}$ input. This input is controlled by external memory management circuitry in the implementation of a virtual memory system. When $\overline{\mathrm{ABORT}}$ is asserted with inputs $\overline{\mathrm{WAIT}}$ and $\overline{\mathrm{SAT}}$, an Abort Instruction function followed by a $\overline{\mathrm{SAT}}$ trap operation are initiated in the CPU.

**Note:** The Z8004 does not have a $\overline{\mathrm{SAT}}$ input; either the $\overline{\mathrm{NVI}}$, $\overline{\mathrm{VI}}$, or the $\overline{\mathrm{NMI}}$ input can be used in its place.

## 7.4 ABORT INSTRUCTION FUNCTION

In a virtual memory system, the detection of a CPU output address that references a location that is not in main memory will cause the memory management circuitry to activate a CPU abort exception (CPU input $\overline{\mathrm{ABORT}}$ is asserted). This initiates an Abort Instruction Function in the CPU which aborts the current instruction execution and saves all of the information that is needed to restart the instruction at the point of interruption. The Abort Instruction Function must be followed by an externally controlled trap ($\overline{\mathrm{SAT}}$ for the Z8003 and any interrupt input for the Z8004). The trap must initiate the execution of a user-prepared routine that will bring the segment or page containing the referenced location into main memory and will perform the functions needed to restart the aborted instruction at the point of interruption. A maskable interrupt should be used for the Z8004 trap function since this type of interrupt has a higher priority than non-maskable interrupts. Details of the abort operations are given in Chapter 9.

## 7.5 RESET

A reset initializes selected control registers of the CPU to system specifiable values. A reset can occur at the end of any clock cycle, provided the $\overline{\mathrm{RESET}}$ line is Low.

A system reset overrides all other considerations, including interrupts, traps, bus requests, and stop requests. A reset must be used to initialize a system as part of the power-up sequence.

Within five clock cycles of the $\overline{\text{RESET}}$ becoming Low $AD_0$-$AD_{15}$ are 3-stated; $\overline{\text{AS}}$, $\overline{\text{DS}}$, $\overline{\text{MREQ}}$, $\overline{\text{BUSACK}}$, and $\overline{\text{MO}}$ are forced Low. The R/$\overline{\text{W}}$, B/$\overline{\text{W}}$, and N/$\overline{\text{S}}$ lines are undefined. $\overline{\text{RESET}}$ must be held Low five clock cycles to reset the CPU.

Three clock cycles after $\overline{\text{RESET}}$ has returned to High, consecutive memory read cycles are executed in system mode to initialize the Program Status registers. In the Z8001 and Z8003, the first cycle reads the FCW from Location 0002 in segment number 0. The second cycle reads the PC segment number value from locations 0004 and the third cycle reads the PC offset value from location 0006 in segment number 0.

Each of these fetches is made with the instruction memory access code (binary 1100) on status lines $ST_3$-$ST_0$. The next initial instruction cycle starts the mainstream program.

In the Z8002 and the Z8004, the first cycle reads the FCW from memory location 2 and the second cycle reads the PC value from location 4. Each of these fetches is made with the instruction memory access code (binary 1100) on status lines $ST_3$-$ST_0$. The next initial instruction cycle starts the program.

## 7.6 INTERRUPT DISABLING

Vectored and nonvectored interrupts can be enabled or disabled independently by setting or clearing appropriate control bits in the Flag and Control Word (FCW). Two control bits in the FCW control the maskable interrupts: VIE and NVIE. Any control bit can be changed when a new FCW is lodded from the PSA during an interrupt or trap acknowledge sequence and will be restored to its previous setting by an Interrupt Return (IRET) instruction. When VIE is 1, vectored interrupts are enabled; when NVIE is 1, nonvectored interrupts are enabled. These two flags can be set or cleared either together or separately. In addition, these control bits are set when the FCW is loaded using either the LDPS or LDCTL FCW instruction.

When any type of interrupt has been disabled, the CPU ignores any interrupt request on the corresponding input pin. Because masked interrupt requests are not retained by the CPU, the request signal must be asserted until the CPU acknowledges the request.

## 7.7 INTERRUPT AND TRAP HANDLING

The CPU response to a trap or interrupt request consists of four steps: acknowledging the external request (for interrupts and segment or address traps), saving the current program status information, loading a new program status, and transferring to the service routine. Returning to the interrupted task at the end of the service routine is accomplished by executing the IRET instruction which removes the saved information from the stack and restores the status. Interrupt timing is shown in Chapter 9.

### 7.7.1 Acknowledge Cycle

An external acknowledge cycle is required only for externally generated requests. As described in Chapter 9, the main effect of such a cycle is to enable the CPU to receive from the external device a 16-bit identifier word, which will be saved with the current program status. Before the acknowledge cycle, the CPU enters its System mode. The N/$\overline{\text{S}}$ line is asserted to indicate that a transition has been made to System mode. The saved FCW is not affected by this change in mode. The CPU remains in System mode until it begins to execute the exception service routine, at which time its mode is dictated by the FCW.

### 7.7 2 Status Saving

The current program status information is saved on the system stack in the following order of entry: the Program Counter, the Flag and Control Word, and finally, the interrupt/trap identifier word. The identifier word contains the reason or source of the trap or interrupt. For internal traps, the identifier is the first word of the trapped instruction. For segment or address trap or interrupts, the identifier is the value on the data bus read by the CPU during the interrupt-acknowledge or trap-acknowledge cycle. The format of the saved program status in the system stack is illustrated in Figure 7-1.

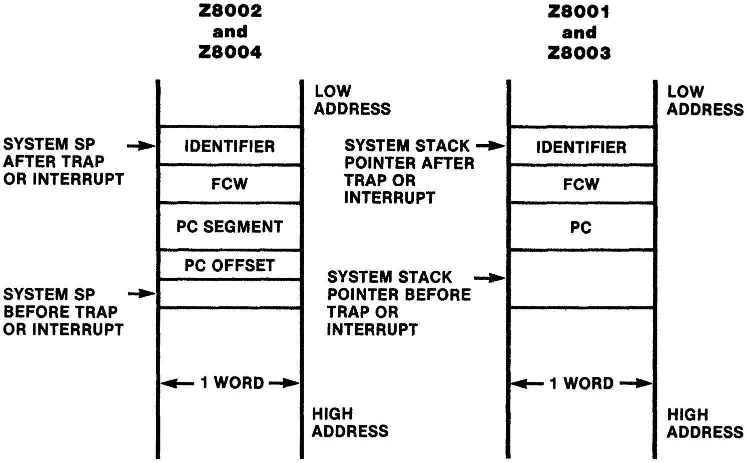

Z8002
and
Z8004
Z8001
and
Z8003
LOW ADDRESS
SYSTEM SP AFTER TRAP OR INTERRUPT
IDENTIFIER
FCW
PC SEGMENT
PC OFFSET
SYSTEM SP BEFORE TRAP OR INTERRUPT
1 WORD
HIGH ADDRESS
LOW ADDRESS
SYSTEM STACK POINTER AFTER TRAP OR INTERRUPT
IDENTIFIER
FCW
PC
SYSTEM STACK POINTER BEFORE TRAP OR INTERRUPT
1 WORD
HIGH ADDRESS

**Figure 7-1. Format of Saved Program Status in the System Stack**

Table 7-1 shows the PC value that is pushed onto the stack for each type of interrupt and trap.

### 7.7.3 Loading New Program Status

After saving the current program status, the new program status (PC and FCW) is automatically loaded from the Program Status Area in system program memory (i.e. status outputs $ST_3$-$ST_0$ indicate $IF_N$, and $N/\overline{S}$ indicates System mode). The particular status words fetched from the Program Status Area are a function of the type of trap or interrupt and (for vectored interrupt) of the interrupt vector. Figure 7-2 shows the format of the Program Status Area.

For each kind of interrupt or trap other than a vectored interrupt, there is a single program status block that is automatically loaded into the Flag and Control Word and the Program Counter.

The size of each program status block depends on the version of the Z8000 (two words for the nonsegmented CPUs (Z8002, Z8004) and four words for the segmented CPUs (Z8001, Z8003).

For all vectored interrupts, the same Flag and Control Word (FCW) is loaded from the corresponding program status block. However, the appropriate Program Counter (PC) value is selected from up to 256 (Z8002, Z8004) or 128 (Z8001,Z8003) different values in the Program Status Area.

**Table 7-1. PC Value Pushed For Each Interrupt or Trap**

| Exception | PC Value is Address Of: |
|---|---|
| Extended Instruction Trap | Second word of instruction |
| Privileged Instruction Trap | Second word of instruction |
| System Call Trap | Next instruction |
| Address Violation Trap | Depends on external circuitry |
| All Interrupts | Next instruction* |

* If an interruptible instruction (e.g., LDIR) is executing but not completed, then the next instruction is the current instruction.

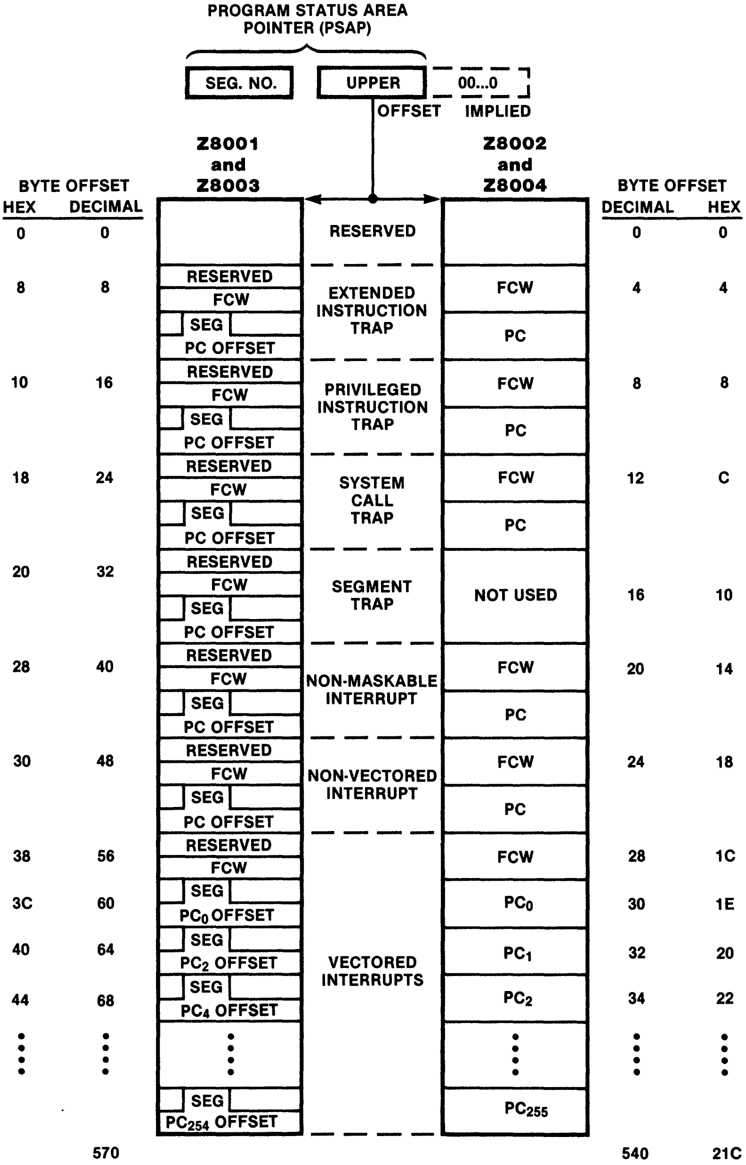


Figure 7-2. Program Status Area

The low-order eight bits of the identifier placed on the data bus by the interrupting device is multiplied by two and used as an offset into the Program Status Area following the FCW for vectored interrupts. On the Z8002 and Z8004, the identifier value 0 selects the first PC value, the value 1 selects the second PC, and so on up to the identifier value 255. On the Z8001 and Z8003, the identifier value 0 selects the first PC value, the value 2 selects the second PC, and so on up to identifier value 254, which selects the 128th PC value. Odd identifier values cannot normally be used with the Z8001 or Z8003 CPUs.

The Program Status Area is addressed by a special control register, the Program Status Area Pointer, or PSAP. This pointer is one word for the nonsegmented CPUs and two words for the segmented CPUs. As shown in Figure 7-2, the pointer contains a segment number (if applicable) and the high-order byte of a 16-bit offset address. The low-order byte is assumed to contain zeros; thus the Program Status Area must start on a 256-byte address boundary. The programmer accesses the PSAP using the Load Control Register instruction (LDCTL).

### 7.7.4 Executing the Service Routine

Loading the new program status automatically initializes the Program Counter to the starting address of the service routine required by the interrupt or trap causing this routine to be executed. Because a new FCW is loaded, the maskable interrupts can be disabled for the initial processing of the service routine by a suitable choice of FCW. This allows critical information to be stored before subsequent interrupts are handled. Service routines that enable interrupts permit interrupts to be handled in a nested fashion.

### 7.7.5 Returning from an Interrupt or Trap

Upon completion, the service routine can execute an Interrupt Return instruction (IRET) to cause execution to continue at the point where the interrupt or trap occurred. IRET causes information to be popped from the system stack in the following order: the identifier is discarded, the saved FCW and PC are restored. The newly loaded FCW takes effect with the next fetched instruction, which is determined by the restored Program Counter.

On Z8001 and Z8003 CPUs, IRET can be executed only in segmented mode; in nonsegmented mode the operation is undefined.

In Virtual Memory Systems, the instruction interrupted by an Abort condition may have been interrupted after it had modified a CPU register(s) but before it had completed execution. When such a condition occurs, special software must ensure that a correct restart environment is established. The software requirements for restarting an aborted instruction are described in Appendix D.

The specific point in the instruction execution cycle at which an abort can occur is determined by the external memory management circuitry used. The instruction restart algorithm used will then depend on the operation of the memory management circuitry used. If a Zilog MMU or PMMU is used, the needed restart information is presented in their respective technical manual, document number 03-8070-01 for the Z8010 MMU and document number 03-822301 for the Z8015 PMMU.

## 7.8 PRIORITY

Because it is possible for several exceptions to occur simultaneously, the CPU enforces a priority scheme for deciding which event will be honored first. The following gives the descending priority order:

- Reset
- Internal Trap (i.e., privileged instruction, system call, extended instruction)
- Nonmaskable Interrupt
- Addressing Violation or Segment/Address Translation Trap (Z8001 and Z8003 only)
- Vectored Interrupt
- Nonvectored Interrupt

The priority system works as follows:

1. When a reset is requested, it is performed immediately.

2. If several non-reset exceptions occur simultaneously, the one that has the highest priority and is also enabled (traps and nonmaskable interrupts are always enabled) is acknowledged, current status is saved, and new status is loaded. The new status consists of the starting address of the service routine (PC) and a new FCW that may disable vectored or nonvectored interrupts.

3. If any enabled exceptions remain after the new FCW is loaded, the highest-priority exception is acknowledged immediately (see Step 2). Note that in this case, the current status is the PC and FCW of the first exception's service routine.

4. The process described in steps 2 and 3 is repeated until no enabled exceptions remain. At that point, the current PC and FCW will contain the status values for the lowest priority exception that was acknowledged.

5. The execution of the service routine now proceeds in the reverse of the order in which the exceptions were acknowledged. After all the exceptions have been serviced, the original status is restored and execution resumes.

Within each of the classes above, there can be several interrupt sources. The internal traps are mutually exclusive and therefore need no priority resolution within that class. The other types arise from external sources; thus when several devices share the same request line, the possibliity arises that two or more devices may request service from the CPU simultaneously. Either all the interrupt sources must be serviced simultaneously (as with MMU), or competing requests must be resolved external to the CPU. The Z-BUS definitions provide for a daisy-chain interrupt structure; all Z-BUS compatible peripherals have input and output pins (IEI and IEO) to implement this type of priority interrupt system.

An external priority interrupt controller can also be used. The Z-CIO, for example, is designed to be used in this manner with the Z8000 CPUs.

# Chapter 8
# Refresh

## 8.1 INTRODUCTION

All Z8000 CPUs have an internal mechanism for refreshing dynamic memory. This mechanism can be activated in two ways:

- When the Refresh Enable (RE) bit in the CPU Refresh control register is set to one (Figure 8-1), a memory refresh cycle is performed periodically at a rate specified by the RATE field in the counter. (See Section 8.3).
- When the $\overline{\text{STOP}}$ line is activated, the CPU generates memory refreshes cycles continuously. (See Section 8.4.)

## 8.2 REFRESH CYCLES

The refresh mechanism is a way of generating a special kind of bus transaction called a refresh cycle, which is described in Chapter 9. A refresh cycle is three clock cycles long and when due, it will be inserted immediately after the last clock cycle of the current bus transaction.

During a refresh cycle, status lines $ST_3$-$ST_0$ are set to 0001, address lines $AD_0$-$AD_8$ contain the memory ROW value, and address lines $A_9$-$A_{15}$ are undefined. The ROW value determines the memory row that is being refreshed on this cycle. Since memory is word-organized, $AD_0$ is always zero. After the refresh cycle is complete, the ROW field is incremented by two, thus stepping through 256 rows numbered 0,2,4...,510.

## 8.3 PERIODIC REFRESH

The Refresh Enable (RE) bit controls only Periodic Refresh; refresh cycles can be generated using the $\overline{\text{STOP}}$ line, regardless of the state of RE. When RE is set to one, the value of the 6-bit RATE field determines the time between successive refreshes (the refresh period). When RATE = 0, the refresh period is 256 clock cycles; when RATE = n, the refresh period is 4n clock cycles. For example, if there is a 4 MHz clock, the refresh period is between 1 µs and 64 µs, while with a 10MHz clock, the refresh period will be between 400ns and 25.6 µs).

The LDCTL instruction is used to set the refresh rate, to set or clear RE, or to initialize the ROW field. (See Section 6.6 for a detailed discussion of this instruction.)

The refresh cycle is generated as soon as possible after the refresh period has elapsed, usually, after the last clock cycle of the current transaction. If the CPU receives a trap or an interrupt simultaneously with a Periodic Refresh request, the refresh operation is performed first.

When the CPU does not have control of the bus (that is, while $\overline{\text{BUSACK}}$ is asserted and the CPU enters Bus-Disconnect state or while the $\overline{\text{WAIT}}$ line is activated), the CPU cannot issue refresh cycles. To deal with this situation, all Z8000 CPUs have internal circuitry that counts and

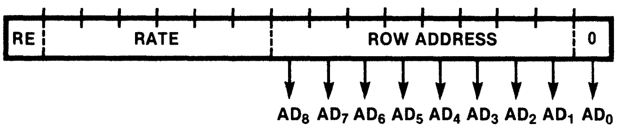


**Figure 8-1. Refresh Control Register**

remembers skipped refresh cycles. When the CPU regains control of the bus, or when the $\overline{\text{WAIT}}$ line is deactivated, it immediately issues the skipped refresh cycles. The internal circuitry can record up to two skipped refresh operations.

After a reset operation, Periodic Refresh is disabled (RE is cleared) and the internal circuitry that counts skipped refreshes is cleared.

## 8.4 STOP-STATE REFRESH

The CPU has three internal operating states: Run, Stop and Bus-Disconnect (see Section 2.8). The Stop state is entered each time the $\overline{\text{STOP}}$ line is activated; while the CPU is in this state, it continuously generates three-clock-cycle refresh transations. When $\overline{\text{STOP}}$ goes High again, one more refresh cycle is performed, then the CPU enters the Run state.

# Chapter 9
# External Interface

## 9.1 INTRODUCTION

This chapter covers the external manifestations (i.e., the activity on the CPU pins) that result from the operations described in Chapters 2 through 8. Since the pins are connected to the system bus (see Figure 2.3 in Chapter 2), much of the discussion will center on the bus and bus operations. The Z8000 CPU is designed to be compatible with the Zilog Z-BUS protocols, which are described in the "Component Interconnect Z-BUS Summary," document number 00-2031-01. In the sections that follow, the interface between the Z8000 CPU and its environment is described in detail.

## 9.2 BUS OPERATIONS

Two kinds of operation can occur on the system bus: transactions and requests. At any given time, one device (either the CPU or a bus requester, such as the Z8016 DMA Controller) has control of the bus and is known as the bus master. A transaction is initiated by the bus master and is responded to by some other device on the bus. Only one transaction can proceed at a time; six kinds of transaction can occur:

- **Memory transaction.** Transfers eight or 16 bits of data to or from a memory location (Section 9.4.2).

- **I/O transaction.** Transfers eight or 16 bits of data to or from a peripheral or CPU support component, such as an MMU (Section 9.4.3).

- **EPU transfer.** Transfers 16 bits of data between the CPU and an EPU (Section 9.4.4).

- **Interrupt/Trap Acknowledge transaction.** Acknowledges an interrupt or trap and transfers an identification/status word (or vector) from the interrupting or trapping device (Section 9.4.5).

- **Refresh transaction.** Refreshes dynamic memory only, does not transfer data (Section 9.4.7).

- **Internal operation transaction.** Data is not transferred. Indicates that the CPU is performing an operation that does not require data to be transferred on the bus (Section 9.4.7).

Only the bus master can initiate transactions. A request, however, can be initiated by a component that does not have control of the bus. Five types of request can occur:

- **Interrupt request.** Requests the attention of the CPU (Section 9.6.1).

- **Bus request.** Requests control of the bus to initiate transactions (Section 9.6.2).

- **Resource request.** Requests control of a particular system resource (Section 9.6.3).

- **Abort request.** Terminates instruction execution (Section 9.6.4).

- **Stop request.** Suspends CPU instruction exection (Section 9.6.5).

When an interrupt or bus request is made, it is answered by the CPU according to its type: for an interrupt request, an interrupt acknowledge transaction is initiated; for a bus request, the CPU enters Bus-Disconnect state, relinquishes the bus, and activates an acknowledge signal; for a stop request, the CPU stops execution and enters Stop/Refresh state. A resource request does not require CPU action, since it occurs on a separate bus that uses external logic to link CPU pins $\overline{MI}$ and $\overline{MO}$ in a daisy chain. A CPU generates a Resource Request when it executes a multi-micro request instruction. An Abort request causes the CPU to terminate execution of the current instruction immediately.

## 9.3 CPU PINS

The CPU pins can be grouped into five categories according to their functions (Figure 9-1).

### 9.3.1 Transaction Pins

These signals provide timing, control, and data transfer for Z-BUS transactions.

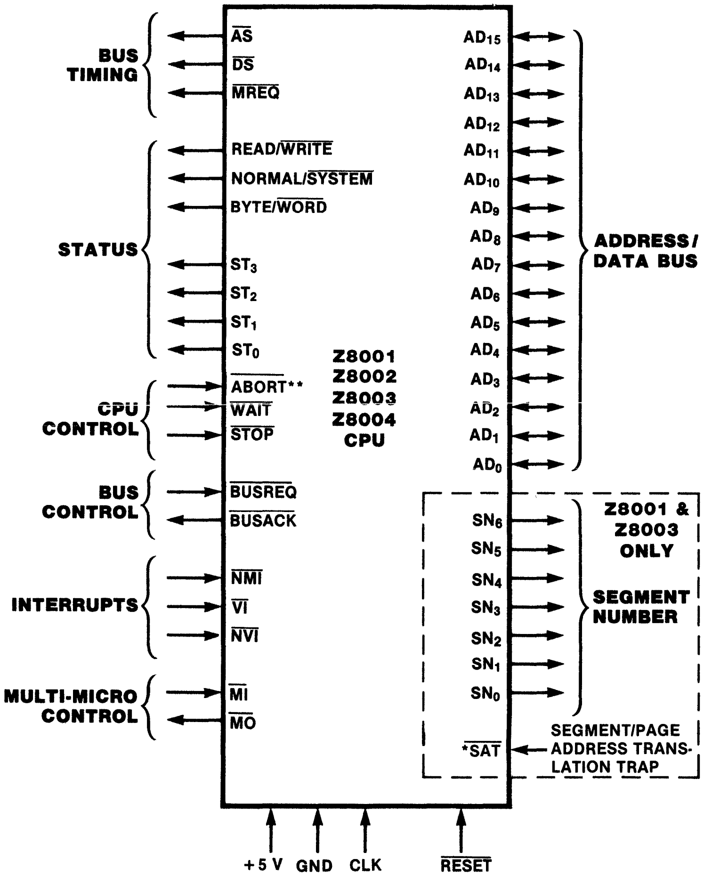
**$AD_0$-$AD_{15}$.** Address/Data (Bidirectional, active High, 3-state). These multiplexed data and address lines carry I/O addresses, memory addresses, and data during Z-BUS transactions. For the Z8001 and Z8003, only the offset portion of memory addresses is carried on these lines.

**$SN_0$-$SN_7$.** Segment Number (Z8001/3 only, output, active High, 3-state). These lines contain the segment number portion of a memory address.

**$ST_0$-$ST_3$.** (Output, active High, 3-state). These lines indicate the kind of transaction occurring on the bus and give additional information about the transaction (such as the address space for memory transactions).

**$\overline{AS}$.** Address Strobe (Output, active Low, 3-state). The rising edge of $\overline{AS}$ indicates the beginning of a transaction and shows that the $AD_0$-$AD_{15}$, $ST_0$-$ST_3$, $N/\overline{S}$, $R/\overline{W}$, and $B/\overline{W}$ signals are valid.

**$\overline{DS}$.** Data Strobe (Output, active Low, 3-state). $\overline{DS}$ provides timing for data movement to or from the CPU.



* $\overline{SAT}$ for Z8003 CPU, $\overline{SEGT}$ for Z8001 CPU.
** $\overline{ABORT}$ used in Z8003 and Z8004 CPUs only.

**Figure 9-1. Pin Functions**

**R/$\overline{W}$.** Read/Write (Output, Low = Write, 3-state). This signal determines the direction of data transfer for memory, I/O, or CPU transfer transactions.

**B/$\overline{W}$.** Byte/Word (Output, Low = Word, 3-state). This signal indicates whether a byte or word of data is to be transmitted during a transaction.

**N/$\overline{S}$.** Normal/System mode (Output, Low = System mode, 3-state). This output indicates whether the CPU is in Normal or System operating mode.

**$\overline{\text{WAIT}}$.** (Input, active Low). A Low on this line causes the CPU to extend the duration of a bus cycle by inserting additional clock cycles.

**$\overline{\text{MREQ}}$.** Memory Request (Output, active Low, 3-state). A falling edge on this line indicates that the address/data bus is holding a memory address.

### 9.3.2 Bus Control Pins

These pins carry signals for requesting and obtaining control of the bus from the CPU.

**$\overline{\text{BUSREQ}}$.** Bus Request (Input, active Low). A Low indicates that a bus requester has obtained, or is trying to obtain, control of the bus.

**$\overline{\text{BUSACK}}$.** Bus Acknowledge (Output, active Low). A Low on this line indicates that the CPU has relinquished control of the bus in response to a bus request.

### 9.3.3 Interrupt/Trap Pins

These pins convey interrupt and external trap requests to the CPU.

**$\overline{\text{ABORT}}$.** Abort Request (Z8003/4 only, input, active Low). When $\overline{\text{ABORT}}$ is asserted it initiates an Instruction Abort in the CPU. This input must be followed immediately by an interrupt.

**$\overline{\text{NMI}}$.** Nonmaskable Interrupt. (Input, active Low). A High-to-Low transition on NMI requests a nonmaskable interrupt.

**$\overline{\text{NVI}}$.** Nonvectored Interrupt (Input, active Low). A Low on this line requests a nonvectored interrupt.

**$\overline{\text{VI}}$.** Vectored Interrupt (Input, active Low). A Low on this input requests a vectored interrupt.

**$\overline{\text{SAT}}$.** Segment/Address Translation Trap (Z8003 only, input, active Low). A Low on this input requests a Segment/Address Translation trap.

**$\overline{\text{SEGT}}$.** Segment Trap (Z8001 only, input, active Low). A Low on this line requests a segment trap.

### 9.3.4 Multi-Micro Pins

These pins are the Z8000's interface to the Z-BUS resource request lines.

**$\overline{\text{MI}}$.** Multi-Micro In (Input, active Low). This input is used to sample the state of the resource request lines.

**$\overline{\text{MO}}$.** Multi-Micro Out (Output, active Low). This line is used by the CPU to make resource requests.

### 9.3.5 CPU Control

These pins carry signals that control the overall operation of the CPU.

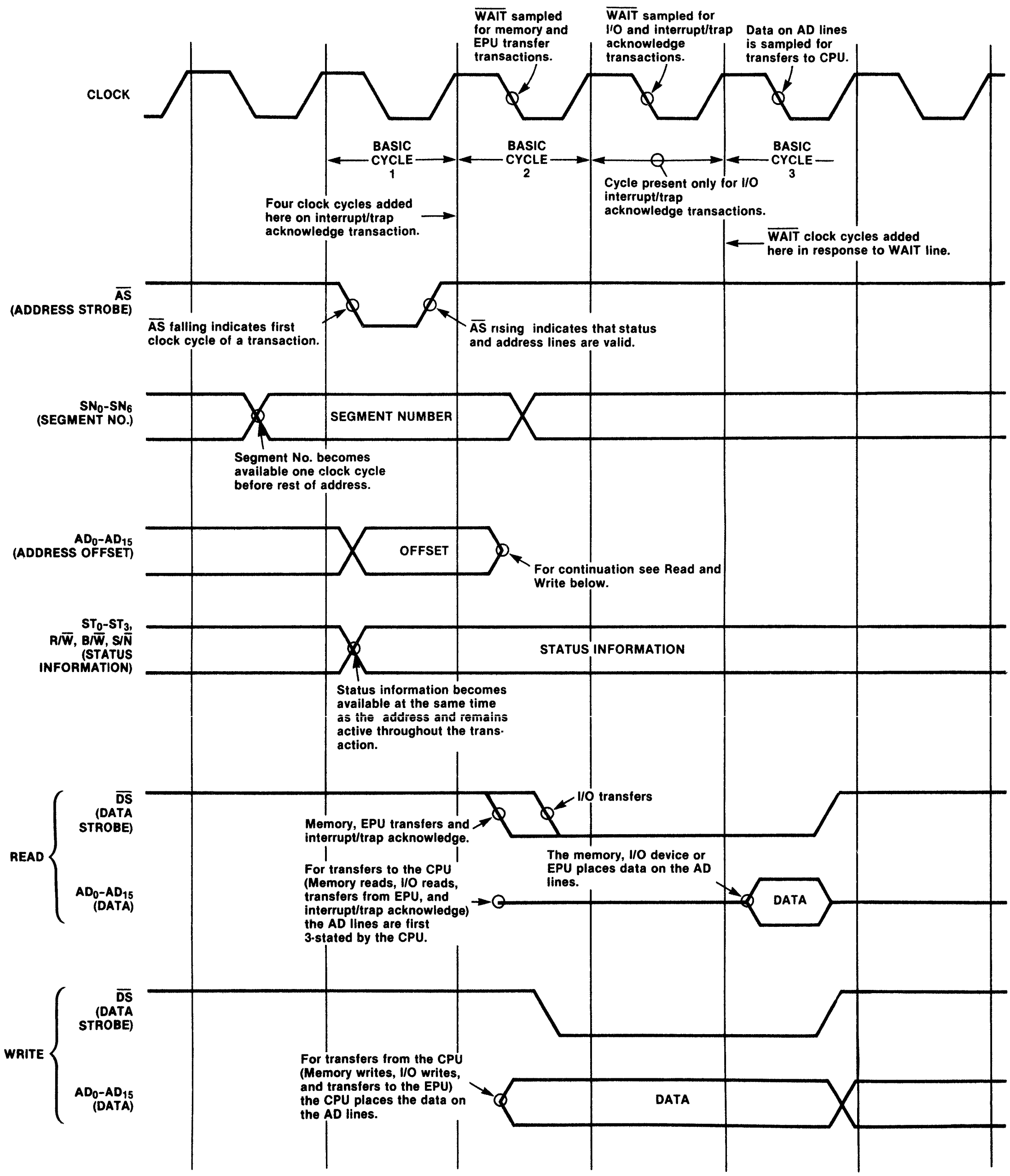
**$\overline{\text{STOP}}$.** (Input, active Low). When asserted this line suspends CPU operation either after the fetch of the first word of an instruction or during an EPU instruction if the EPU is busy.

**$\overline{\text{RESET}}$.** (Input, active Low). A Low on this line resets the CPU.

## 9.4 TRANSACTIONS

Data transfers to and from the CPU are accomplished through the use of transactions.

All transactions start with Address Strobe ($\overline{\text{AS}}$) being driven Low and then raised High by the CPU. On the rising edge of $\overline{\text{AS}}$, the status lines $ST_0$-$ST_3$ are valid; these lines indicate the type of transaction being initiated (see Table 9-1). The six types of transactions are discussed in section 9.4. Associated with the status lines are three other lines that become valid at this time. These are Normal/System (N/$\overline{S}$), Read/Write (R/$\overline{W}$), and Byte/Word (B/$\overline{W}$). Except where indicated below,



WAIT sampled for memory and EPU transfer transactions.
WAIT sampled for I/O and interrupt/trap acknowledge transactions.
Data on AD lines is sampled for transfers to CPU.
CLOCK
BASIC CYCLE 1
BASIC CYCLE 2
BASIC CYCLE 3
Cycle present only for I/O interrupt/trap acknowledge transactions.
Four clock cycles added here on interrupt/trap acknowledge transaction.
WAIT clock cycles added here in response to WAIT line.
AS (ADDRESS STROBE)
AS falling indicates first clock cycle of a transaction.
AS rising indicates that status and address lines are valid.
SN0-SN6 (SEGMENT NO.)
SEGMENT NUMBER
Segment No. becomes available one clock cycle before rest of address.
AD0-AD15 (ADDRESS OFFSET)
OFFSET
For continuation see Read and Write below.
ST0-ST3, R/W, B/W, S/N (STATUS INFORMATION)
STATUS INFORMATION
Status information becomes available at the same time as the address and remains active throughout the transaction.
READ
DS (DATA STROBE)
I/O transfers
Memory, EPU transfers and interrupt/trap acknowledge.
The memory, I/O device or EPU places data on the AD lines.
AD0-AD15 (DATA)
For transfers to the CPU (Memory reads, I/O reads, transfers from EPU, and interrupt/trap acknowledge) the AD lines are first 3-stated by the CPU.
DATA
WRITE
DS (DATA STROBE)
AD0-AD15 (DATA)
For transfers from the CPU (Memory writes, I/O writes, and transfers to the EPU) the CPU places the data on the AD lines.
DATA

Figure 9-2. Transaction Timing

**Table 9-1. Status Codes**

| Kind of Transaction | $ST_3$-$ST_0$ | Additional Information |
|---|---|---|
| Internal Operation | 0000 | |
| Refresh | 0001 | |
| I/O Transaction | 0010 | Standard I/O |
| | 0011 | Special I/O |
| Interrupt Acknowledge Transaction | 0100 | Segment/Address Translation Trap |
| | 0101 | Nonmaskable Interrupt |
| | 0110 | Nonvectored Interrupt |
| | 0111 | Vectored Interrupt |
| Memory Transaction | 1000 | Data Address Space |
| | 1001 | Stack Address Space |
| | 1010* | Data Address Space, EPU Transfer |
| | 1011* | Stack Address Space, EPU Transfer |
| | 1100 | Program Address Space |
| | 1101 | Program Address Space, First Word of Instruction |
| EPU/CPU Transfer | 1110 | |
| Locked Memory Transaction | 1111 | Test and Set Instruction |

*On earlier Z8000 CPUs status codes 1000 and 1001, rather than 1010 and 1011, indicate that the EPU is to capture or supply the data.

$N/\overline{S}$ designates the operating mode of the CPU, $R/\overline{W}$ designates the direction of data transfer (read to the CPU, write from the CPU), and $B/\overline{W}$ designates the size of the data item being transferred.

If the transaction requires an address, it too is valid on the rising edge of $\overline{AS}$. No address is required for interrupt acknowledge, EPU transfer, or internal operation transactions. (In the Z8001 and Z8003, the segment number lines $SN_0$-$SN_6$ are valid one clock cycle before the rising edge of $\overline{AS}$ to allow external memory management hardware to perform some of its functions in parallel with the CPUs address computation. See Chapter 2 for more information.)

The CPU uses Data Strobe ($\overline{DS}$) to time the actual data transfer. In refresh and internal operation transactions the CPU does not transfer any data and thus does not activate $\overline{DS}$. For write operations ($R/\overline{W}$ = Low), a Low on $\overline{DS}$ indicates that valid data from the bus master is on the $AD_0$-$AD_{15}$ lines. For read operations ($R/\overline{W}$ = High), the bus master places lines $AD_0$-$AD_{15}$ into the high impedance state before driving $\overline{DS}$ Low so that the addressed device can put its data on the bus. The bus master samples this data on the falling clock edge that precedes the rising edge of $\overline{DS}$.

### 9.4.1 $\overline{\text{WAIT}}$

As shown in Figure 9-2, $\overline{\text{WAIT}}$ is sampled on the falling clock edge of the last clock cycle before data is sampled by the CPU (Read) or $\overline{DS}$ rises (Read or Write). If $\overline{\text{WAIT}}$ is Low, another cycle is added to the transaction before data is sampled or $\overline{DS}$ rises. In this added cycle, and in all subsequent added WAIT cycles input $\overline{\text{WAIT}}$ is again sampled on the falling edge and, if it is Low, another cycle is added to the transaction. In this way, the transaction can be extended to an arbitrary length to accommodate slow memories or peripherals.

All WAIT inputs to the CPU must be synchronized with the CPU clock.

### 9.4.2 Memory Transactions

Memory transactions move data to or from memory when the CPU makes a memory access. Thus, they are generated during program execution to fetch instructions from memory or to fetch or store memory data. They are also generated to store the current program status and to fetch new program status during interrupt, trap, or reset handling.

As shown in Figure 9-3, a memory transaction is three clock cycles long unless extended by inserted WAIT cycles. The status pins, besides indicating a memory transaction, give the following information:

- Whether the memory access is to the data (1000, 1010), stack (1001,1011), or program (1101,1101) address space (Chapter 3).

- Whether the first word of an instruction (1101) or another program element (1100) is being fetched.

- Whether the data for the access is to be supplied (write code 1010) or captured (read code 1011) by an Extended Processing Unit.

For the Z8002, the full memory address will be on $AD_0$-$AD_{15}$ when AS rises. For the Z8001, the offset portion of the segmented address will be on $AD_0$-$AD_{15}$ and the segment number portion will be on $SN_0$-$SN_6$ when AS rises. The segment portion will become valid on $SN_0$-$SN_6$ approximately one cycle before $AD_0$-$AD_{15}$ is valid.

During the transfer of bytes into the CPU, the input byte can be read from either the high or low half of the bus depending on the state of bus line $AD_0$. If $AD_0$ is set to 1, input is taken from lines $AD_7$-$AD_0$; if $AD_0$ is set to 0, input is taken from lines $A_{15}$-$AD_8$. During the transfer of bytes from the CPU, the contents of each output byte are repeated on both halves of the bus. Figure 9-4 illustrates the manner in which memory is organized. For byte reads (B/W High, R/W High) the CPU uses only the byte whose address it outputs. For byte writes (B/W High, R/W Low), the memory should store only the byte whose address was output by the CPU. For word transfers, (B/W = Low), all 16 bits are captured by the CPU (Read: R/W = High) or stored by the memory (Write: R/W = Low).

As explained more fully in Section 9.5, a Z8000 CPU and an Extended Processing Unit act like a single CPU with the CPU providing addresses, status and timing information and the EPU providing, or capturing, data.

### 9.4.3 I/O Transactions

I/O transactions move data to or from peripherals or CPU support devices (e.g., MMUs). They are generated during the execution of I/O instructions.

As shown in Figure 9-5, I/O transactions are four clock cycles long at a minimum, and they may be lengthened by the addition of WAIT cycles. The extra clock cycles allow for slower peripheral operation.

I/O cycles are like memory cycles, except for the extra Wait state and the different $ST_3$-$ST_0$ status and MREQ values. Peripherals whose speeds are better matched to the 3-cycle memory transaction can be interfaced using memory-mapped I/O.

Peripherals designed to be used with the Z8000 CPUs can generally be interfaced directly to the CPU using the I/O transaction mechanism without the need for special hardware to generate Wait states.

The status lines indicate whether the access is to the Standard I/O (0010) or Special I/O (0011) address space. The N/S line is always Low, indicating System mode. The I/O address is found on $AD_0$-$AD_{15}$ when AS rises. Since the I/O address is always 16 bits long, the segment number lines are undefined on Z8001 and Z8003 CPUs.

Word data (B/W = Low) to or from the CPU is transmitted on $AD_0$-$AD_{15}$. As stated in Section 9.3.3, byte data is sent to the CPU on either half of the bus and byte data and address outputs of the CPU are repeated on both halves of the bus. This allows peripheral devices or CPU support devices to attach to only eight of the 16 $AD_0$-$AD_{15}$

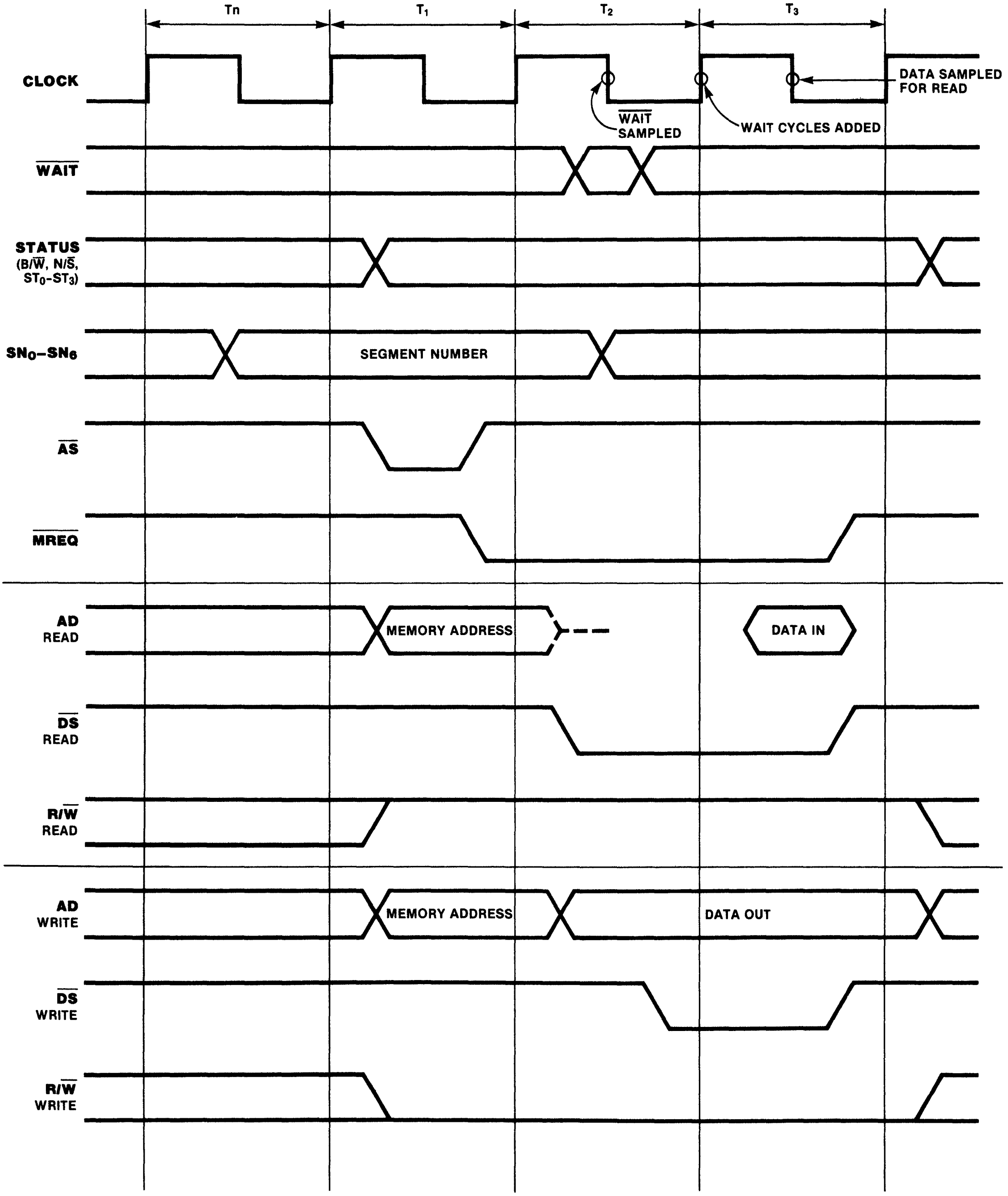


Figure 9-3. Memory Read and Write Transaction

lines. The Read/Write line ($R/\overline{W}$) indicates the direction of the data transfer: peripheral-to-CPU (Read: $R/\overline{W}$ = High) or CPU-to-peripheral (Write: $R/\overline{W}$ = Low).

### 9.4.4 EPU Transfer Transactions

EPU transfer transactions move data between the CPU and an Extended Processing Unit (EPU) or between an EPU and memory. During EPU/CPU transfers, the CPU can transfer data either to or from an EPU, it can either read from, or write to, the EPU's status registers, and can perform transfers between the EPU and memory. EPU transfer transactions are performed during the execution of EPA instructions.

EPU memory transfer transactions are the same as memory transactions (Figure 9-3). No address is generated, and status codes 1010 and 1011 are used. In a multiple EPU system, the EPU that is to participate in a transaction is selected implicitly, as described in Section 9.5, rather than by an address. EPU-CPU transactions have the same timing relationships as I/O transactions (Figure 9-5).

The data transferred is 16-bit words ($B/\overline{W}$ = Low), except for transfer between the Flags byte of the FCW and an EPU. In this case, a byte of data is transferred on $AD_0$-$AD_7$ ($B/\overline{W}$ = High). The Read/Write line ($R/\overline{W}$) indicates the direction of the data transfer. The $N/\overline{S}$ line indicates either System mode (Low) or Normal mode (High).

### 9.4.5 Interrupt/Trap Acknowledge Transactions

Acknowledge transactions acknowledge an interrupt or trap and read a 16-bit identifier word from the device that generated the interrupt or trap. The transactions are generated by the CPU when an interrupt or segment trap is detected.

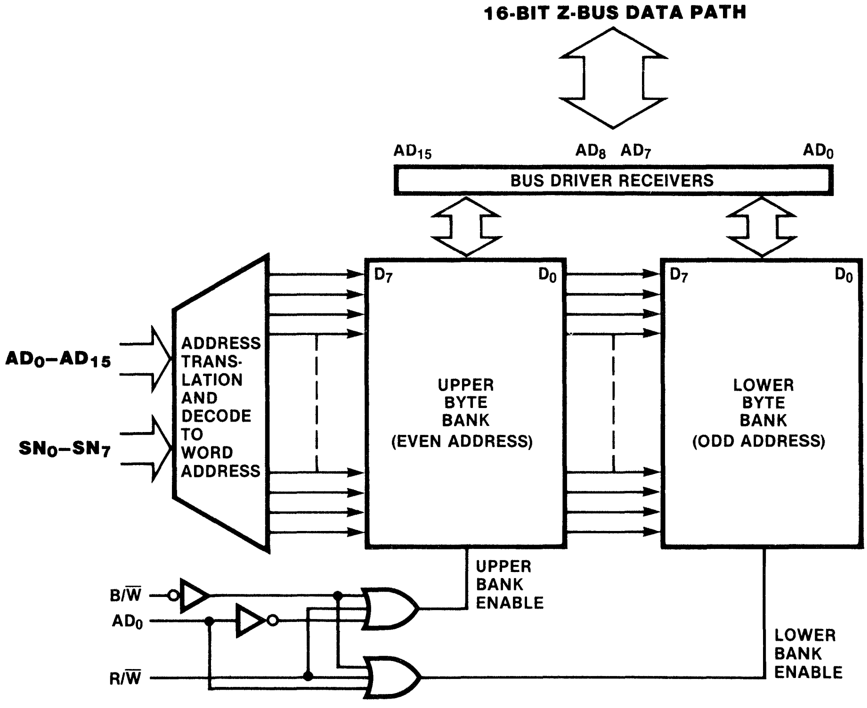


Figure 9-4. Memory Organization

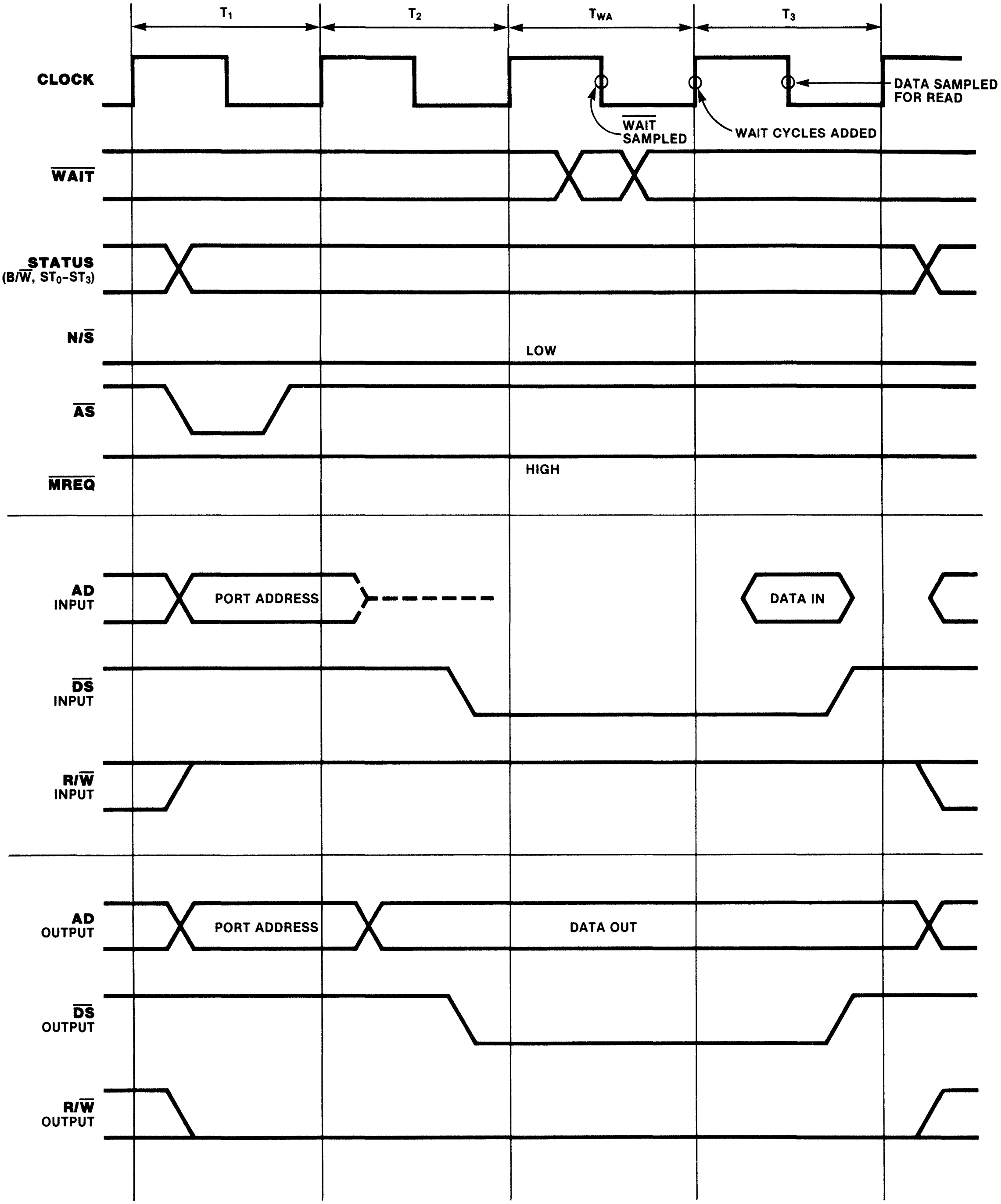


Figure 9-5. Input/Output Transaction

Acknowledge transactions are ten clock cycles long at a minimum (as shown in Figure 9-6), having five automatic Wait cycles. The Wait cycles are used to give the interrupt priority daisy chain (or other priority resolution device) time to settle before the identifier word is read. (Consult the "Z-BUS Component Interconnect Summary," document number 00-2031-01 for more information on the operation of the priority daisy-chain).

The status lines identify the type of exception that is being acknowledged. The possibilities are Segment Trap or Segment/Address Translation Trap (0100), Nonmaskable Interrupt (0101), Nonvectored Interrupt (0110), and Vectored Interrupt (0111). No address is generated. The $N/\overline{S}$ line indicates System mode (Low), the $R/\overline{W}$ line indicates Read (High), and the $B/\overline{W}$ line indicates Word (Low).

The only item of data transferred is the identifier word, which is always 16 bits long and is captured from the $AD_0$-$AD_{15}$ lines on the falling clock edge as $\overline{DS}$ goes High.

As shown in Figure 9-6, there are two places where $\overline{WAIT}$ is sampled, and thus WAIT cycles can be inserted at two points in acknowledge transactions. The first serves to delay the falling edge of $\overline{DS}$ to allow the daisy chain a longer time to settle, and the second serves to delay the point at which data is read.

### 9.4.6 Interrupt/Trap Request and Acknowledge Operations

The following paragraphs describe the operations required to initiate a trap interrupt and acknowledge function. Refer to Figure 9-6 for timing.

The $\overline{VI}$, $\overline{NVI}$, and $\overline{SAT}$ inputs, as well as the state of the internal $\overline{NMI}$ latch, are sampled at the beginning of $T_3$.

In response to an interrupt or trap, the subsequent $IF_1$ cycle is exercised. The Program Counter, however, is not updated, but the system Stack Pointer is decremented.

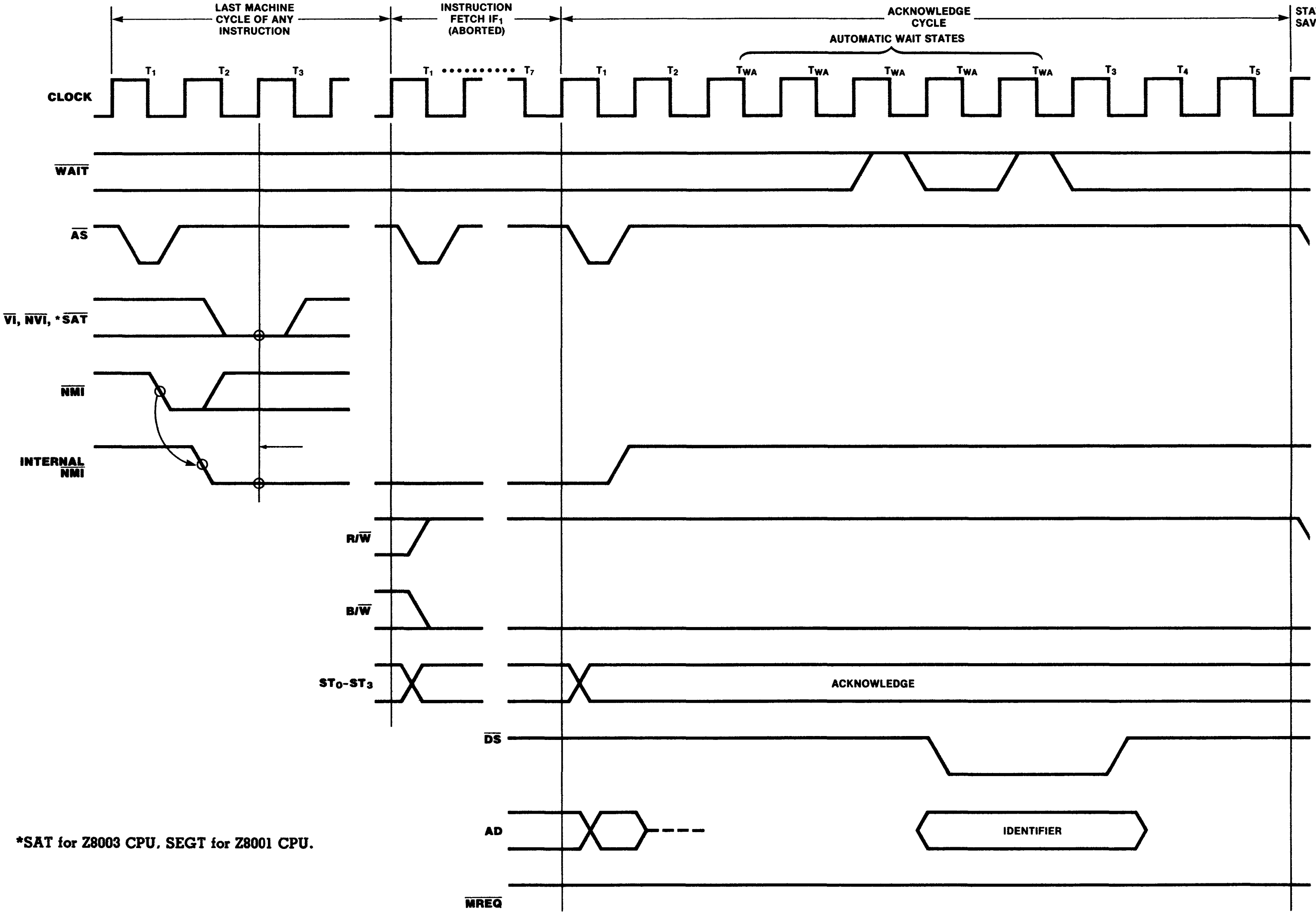
The next machine cycle is the interrupt acknowledge cycle. This cycle has five automatic Wait states, and additional Wait states can be inserted. After the last Wait state, the CPU reads the information on $AD_0$-$AD_{15}$ and stores it temporarily, to be saved on the stack later in the acknowledge sequence. This word identifies the source of the interrupt or trap. For internal traps, the identifier is the first word of the trapped instruction. For external events, the identifier is the contents of the data bus as sampled during $T_3$ of the acknowledge cycle. During nonvectored and nonmaskable interrupts and Segment/Address Translation Trap, all 16-bits can represent peripheral device status information. For the vectored interrupt, the low byte is an index to the array of PC values in the PSA, and the high byte can be used for extra status. During and after the acknowledge cycle, the $N/\overline{S}$ output indicates the automatic change to System mode.

### 9.4.7 Internal Operations and Refresh Transactions

There are two kinds of bus transaction that do not transfer data: internal operations and memory refresh. Both look like a memory transaction, except that Data Strobe remains High and no data is transferred.

For internal operation transactions (shown in Figure 9-7), the Address and Segment Number lines contain arbitrary data when the Address Strobe goes High. The $R/\overline{W}$ line indicates Read (High), the $B/\overline{W}$ line is undefined, and $N/\overline{S}$ is the same as for the immediately preceding transaction. This transaction is initiated to maintain a minimum transaction rate while the CPU is doing a long internal operation.

A memory refresh transaction (shown in Figure 9-8) is generated by the Z8000 CPU's refresh mechanism as described in Chapter 8 and can come immediately after the final clock cycles of any other transaction. The Refresh register's 9-bit ROW field is output on $AD_0$-$AD_8$ during the normal time for addresses. Refresh transactions can be used to



*SAT for Z8003 CPU, SEGT for Z8001 CPU.

Figure 9-6. Segment/Address Translation Trap Interrupt Request and Acknowledge

generate refreshes for dynamic RAMs. The value of N/$\overline{S}$, R/$\overline{W}$, and B/$\overline{W}$ during a refresh transaction is the same as for the immediately preceeding transaction.

$\overline{WAIT}$ is not sampled during internal operations or refresh cycles.

## 9.5 CPU AND EXTENDED PROCESSING UNIT (EPU) INTERACTION

A Z8000 CPU and one or more EPUs work together like a single CPU component, with the CPU providing address, status and timing signals and the EPU supplying and capturing data. The EPU

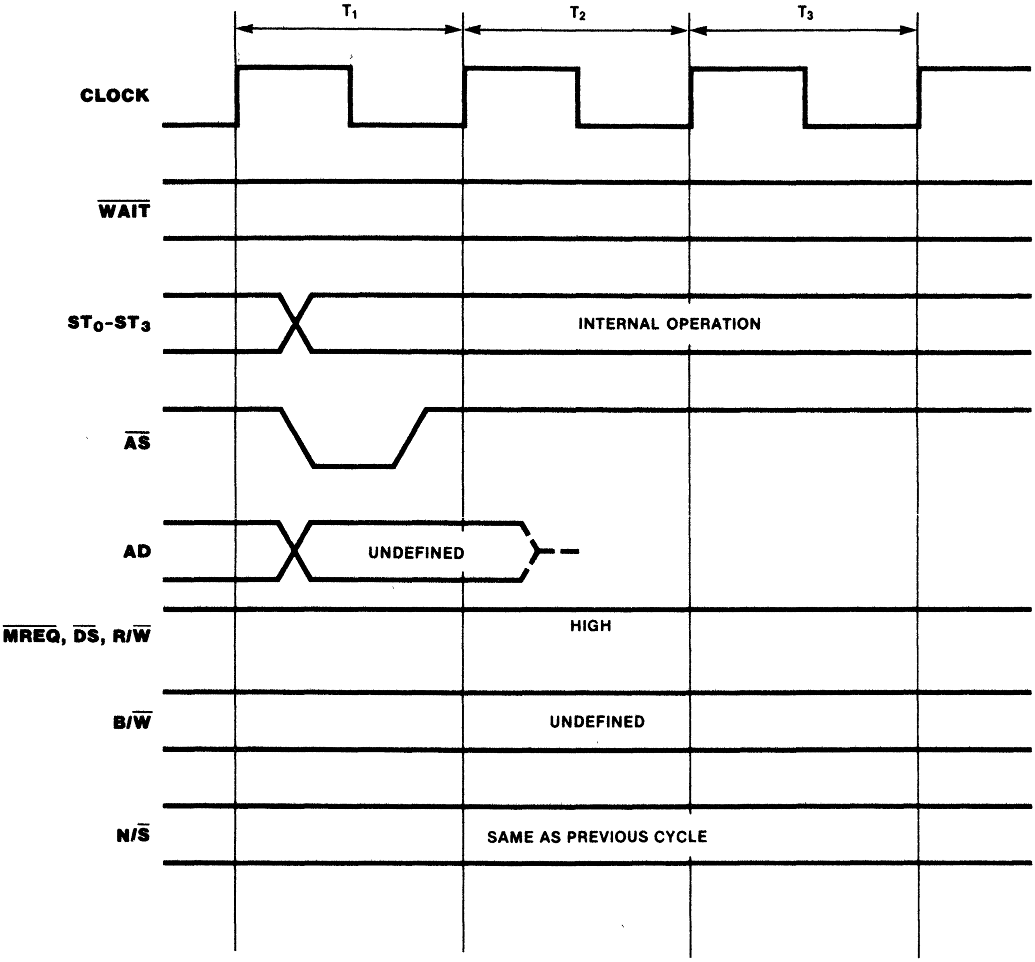


Figure 9-7. Internal Operation Timing

monitors the instructions fetched and the status and timing signals output by the CPU so that it will know when to participate in a memory or EPU transfer transaction. When the EPU is to participate in a memory transaction, the CPU places its AD lines into the high impedance state while $\overline{DS}$ is Low, so that the EPU can use them.

In order to know which transaction it is to participate in, the EPU must track the following sequence of events:

1. When the CPU fetches the first word of an instruction ($ST_3$-$ST_0$ = 1101), the EPU must also capture the intruction returned by memory. If the instruction is an extended instruction, it will have an ID field which indicates whether or not the EPU is to execute the instruction.

2. If the instruction is to be executed by the EPU, the next non-refresh transaction by the CPU will fetch the second word of the instruction ($ST_3$-$ST_0$ = 1100). The EPU must also capture this word.

3. If the instruction involves a read or write to memory, there will be zero or more program fetches by the CPU ($ST_3$-$ST_0$ = 1100) to obtain the address portion of the extended instruction. The next one to 16 non-refresh transactions by the CPU will transfer data between memory and the EPU. (See Table 9-1 for codes.)The EPU must supply the data (Write, R/$\overline{W}$ Low) or capture the data (Read, R/$\overline{W}$ High) for each transaction, just as if it were part of the CPU. In both cases, the CPU will 3-state its AD lines while data is being transferred

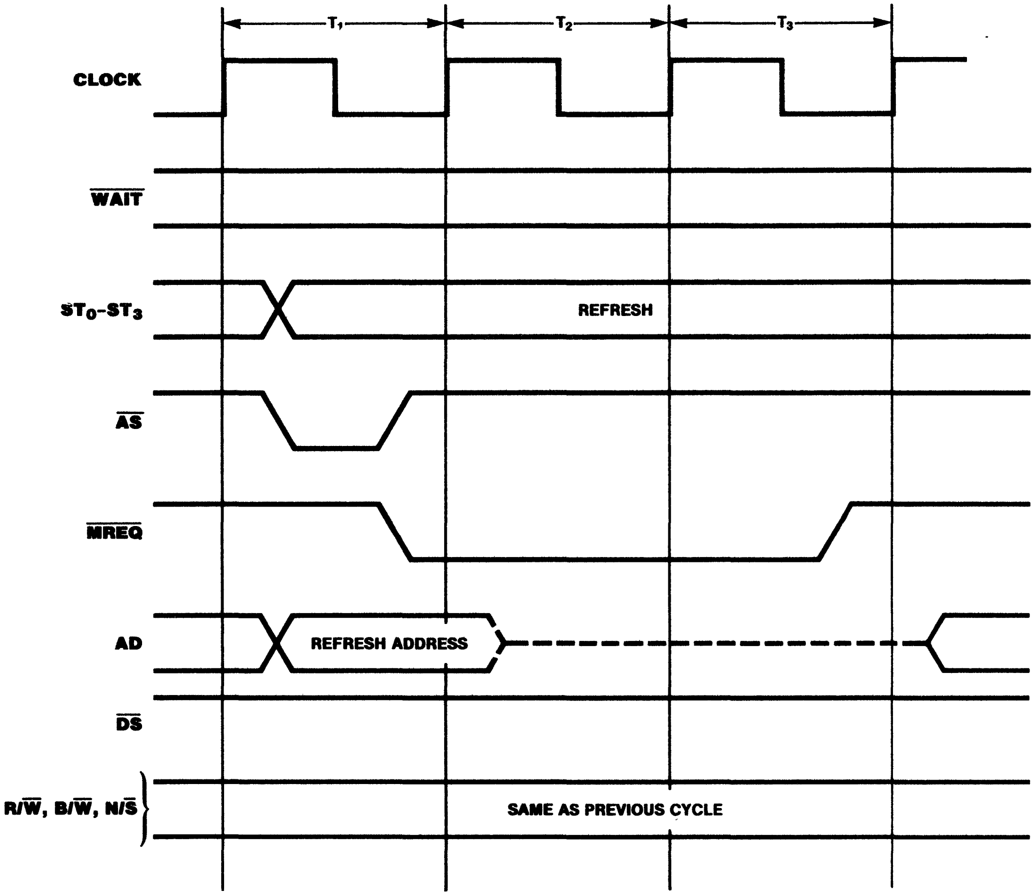


Figure 9-8. Memory Refresh Timing

($\overline{\text{DS}}$ Low). EPU memory transfers are always word-oriented (B/$\overline{\text{W}}$ Low).

4. If the instruction involves a transfer between the CPU and EPU, the next one to 16 non-refresh transactions by the CPU will transfer data between the EPU and CPU ($ST_3$-$ST_0$ = 1110).

To follow the above sequence, an EPU has to monitor the $\overline{\text{BUSACK}}$ line to verify that the transaction it is monitoring on the bus was generated by the CPU. It should also be noted that in a multiple EPU system, there is no indication on the bus as to which EPU is cooperating with the CPU at any given time. This must be determined from the opcodes and ID fields of the extended instructions the EPU captures.

A final aspect of CPU-EPU interaction is the use of the CPU's $\overline{\text{STOP}}$ pin. When an EPU begins to execute an extended instruction, the CPU can continue fetching and executing instructions. If the CPU fetches another extended instruction before the first one has completed execution, the EPU must activate the CPU's $\overline{\text{STOP}}$ pin to stop the CPU (as described in Section 9.6.5) until execution of the previous EPU instruction is completed.

Besides determining whether or not to participate in the execution of an EPA instruction, the EPU must determine from the first two instruction words:

- Whether or not a memory access will be made.
- The number of words of data to be transferred for memory or EPU-CPU transfers.
- The operation to be performed on the data.

## 9.6 REQUESTS

There are four kinds of request that the Z-BUS supports and in which the Z8000 CPU participates. These are

- Interrupt/Trap requests, which another device Initiates and the CPU accepts and acknowledges.

- Bus requests, which another potential bus master initiates and the CPU accepts and acknowledges.

- Resource requests, which any device capable of implementing the request protocol (usually the CPU) can request. No component has control by default of the resource controlled by the resource bus.

The CPU supports an additional request beyond those of the Z-BUS:

- Stop request, which another device initiates and the CPU accepts.

When a request is made, it is answered according to its type: for bus requests, an acknowledge signal is sent (Sections 9.6.2 and 9.6.3); for Stop request, the CPU enters the Stop/Refresh state. In all cases except Stop, the Z-BUS provides for a daisy-chain priority mechanism to arbitrate between simultaneous requests.

### 9.6.1 Interrupt/Trap Request

The Z8000 CPU supports three interrupts and one external trap (segment or segment/address translation trap) as shown in Figure 9-6. The Interrupt Request line ($\overline{\text{INT}}$) of a Z-BUS peripheral that is capable of generating an interrupt may be tied to any of the three Z8000 interrupt pins ($\overline{\text{NMI}}$, $\overline{\text{NVI}}$, $\overline{\text{VI}}$). Several devices can be connected to one pin, in which case the devices must be arranged in a priority daisy chain using the IEI and IEO pins available on all Z-BUS peripherals. The segment trap or segment/address translation trap pin ($\overline{\text{SEGT}}$ for Z8001, $\overline{\text{SAT}}$ for Z8003) is activated by the memory management hardware. The CPU uses the same protocol for handling requests on any of these pins. Here is the sequence of events that is followed:

- Any High-to-Low transition on the $\overline{\text{NMI}}$ input is asynchronously edge-detected, and the internal $\overline{\text{NMI}}$ latch is set. At the beginning of the last clock cycle in the last machine cycle of any instruction, the $\overline{\text{VI}}$, $\overline{\text{NVI}}$, and $\overline{\text{SEGT}}$ inputs are sampled along with the state of the internal $\overline{\text{NMI}}$ latch.

- If an interrupt or trap is detected, the subsequent initial instruction fetch cycle is exercised, but nullified.

- The next machine cycle is the interrupt acknowledge transaction (see Section 9.4.5) which results in an identifier word from the highest-priority interrupting device being read from the AD lines.

- This word, along with the program status information, is stored on the System mode stack, and new status information is loaded (see Chapter 7).

For more information about the interrupt structure, consult the "Component Interconnect Z-BUS summary."*

Interrupt requests are sampled during the penultimate clock cycle of each instruction; however, the decision to accept the request is made at the start of the next instruction and the instruction fetch is aborted if the pending interrupt is enabled. Thus if an interrupt request is pending during the execution of Enable Interrupt (or LDPS, LD FCW, SC or IRET which would enable the interrupt), the pending interrupt will be acknowledged after the execution of that instruction. For example, if a vectored interrupt is pending and the instruction sequence to be executed is

```
EI  VI
DI  VI
```

then the vectored interrupt will be acknowledged between the execution of the EI and DI instruction. Note that a 7 cycle aborted initial instruction fetch is inserted between the execution of the EI instruction and the interrupt acknowledge sequence.

### 9.6.2 Bus Request

To generate transactions on the bus, a potential bus master (such as a DMA Controller) must gain control of the bus by making a bus request (shown in Figure 9-9). A bus request is initiated by pulling $\overline{\text{BUSREQ}}$ Low. Several bus requesters may be wired to the $\overline{\text{BUSREQ}}$ pin; priorities are resolved externally to the CPU, usually by a priority daisy chain (see the "Component Interconnect Z-BUS Summary"*).

The asynchronous $\overline{\text{BUSREQ}}$ signal generates an internal $\overline{\text{BUSREQ}}$, which is synchronous. If the external $\overline{\text{BUSREQ}}$ is Low at the beginning of any machine cycle, the internal $\overline{\text{BUSREQ}}$ will cause the bus acknowledge line ($\overline{\text{BUSACK}}$) to be asserted after the current machine cycle is completed. The CPU then enters Bus-Disconnect state and gives up control of the bus. All CPU output pins, except $\overline{\text{BUSACK}}$ and $\overline{\text{MO}}$, are 3-stated.

The CPU regains control of the bus two clock cycles after $\overline{\text{BUSREQ}}$ rises. Thus any device desiring control of the bus must wait at least two cycles after $\overline{\text{BUSREQ}}$ has risen before pulling it down again.

### 9.6.3 Resource Request

The CPU generates resource requests by executing the Multi-Micro Request (MREQ) instruction. The CPU tests the availability of the shared resource by examining $\overline{\text{MI}}$. If $\overline{\text{MI}}$ is High, the resource is available, otherwise the CPU must try again later. The $\overline{\text{MO}}$ pin is used to make the resource request. $\overline{\text{MO}}$ is pulled Low, then, after a delay for arbitration of priority, $\overline{\text{MI}}$ is tested again. If it is Low, the CPU has control of the resource; if it is still High, the request was not granted and $\overline{\text{MO}}$ must be deactivated. If the request was granted $\overline{\text{MO}}$ must be kept active until the CPU is ready to release the resource, whereupon $\overline{\text{MO}}$ is deactivated by an MRES instruction.

The "Component Interconnect Z-BUS Summary"* describes an arbitration scheme that is implemented with a resource request daisy chain.

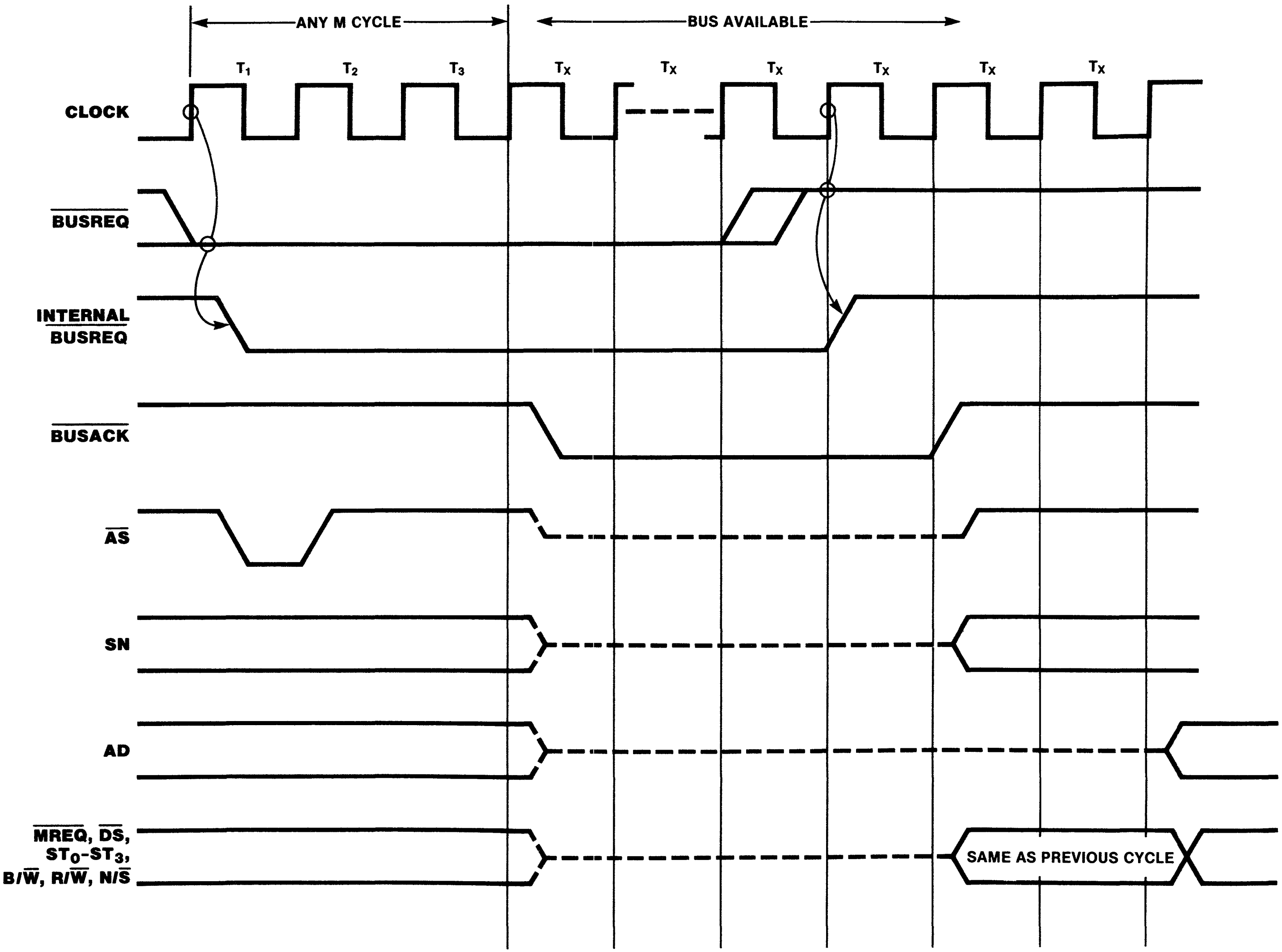
### 9.6.4 Stop Request

As shown in Figure 9-10, the $\overline{\text{STOP}}$ pin is normally sampled on the falling clock edge immediately preceding an initial instruction fetch cycle. If $\overline{\text{STOP}}$ is found Low, the CPU enters Stop/Refresh state and a stream of memory refresh cycles is inserted after the third clock cycle in the instruction fetch. The ROW field in the Refresh Register is incremented by two after every refresh cycle.

When $\overline{\text{STOP}}$ is found High again, the next refresh cycle is completed; then the original instruction fetch continues.

If the EPA bit in the FCW is set (indicating an EPU is in the system), the $\overline{\text{STOP}}$ line is also sampled on the falling clock edge immediately preceding the second word of an instruction fetch--if the first word indicates an extended instruction. Thus, the $\overline{\text{STOP}}$ line can be used by an EPU to deactivate the CPU whenever the CPU fetches an extended instruction before the EPU has finished processing an earlier one. The $\overline{\text{STOP}}$ line can also be used to single-step the CPU.

*Document number 00-2031-01



2045-012

Figure 9-9. Bus Request/Acknowledge Timing

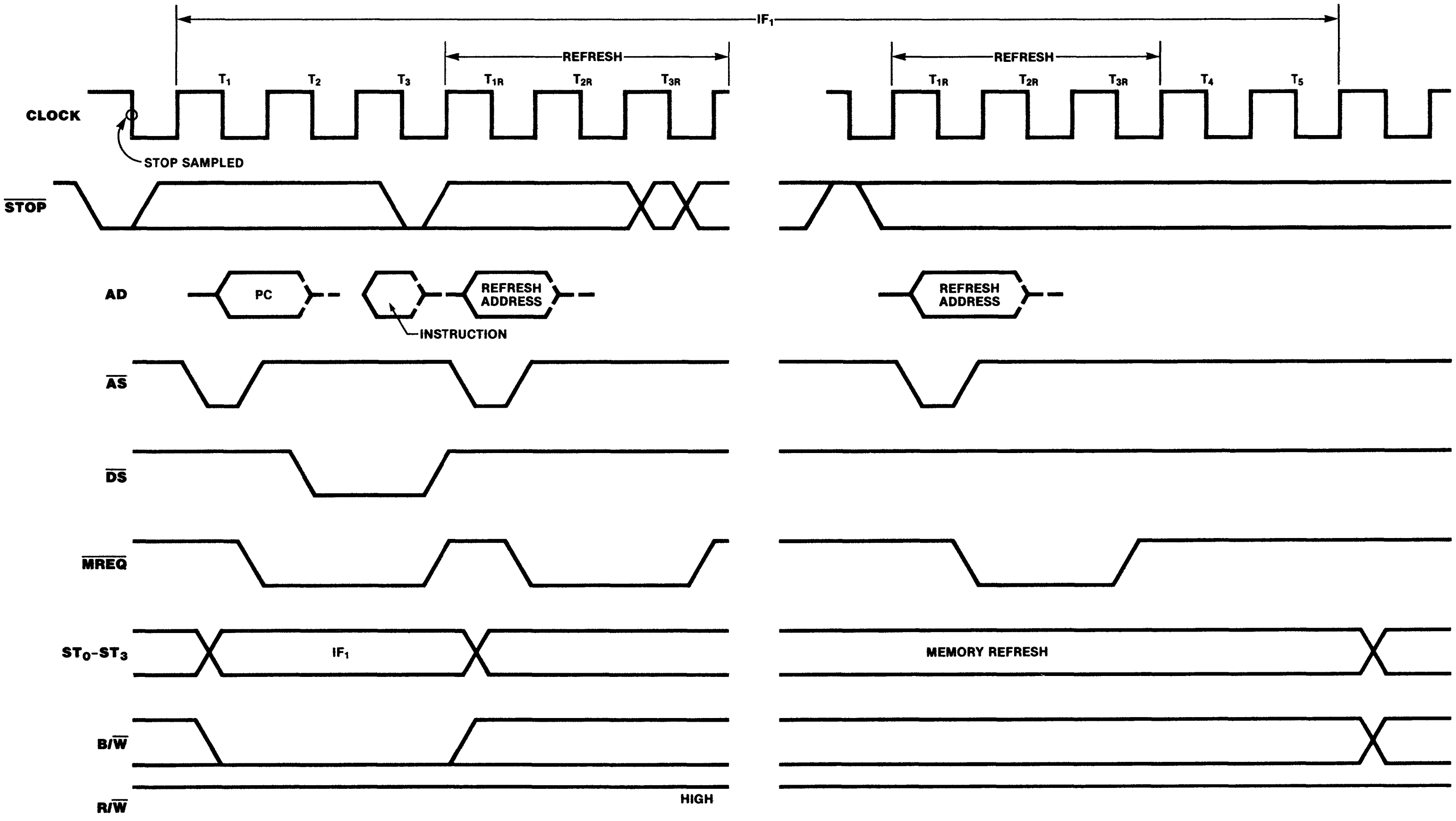


Figure 9-10. Stop Timing

## 9.7 ABORT REQUEST

The timing for an Instruction Abort operation is shown in Figure 9-11. As shown, the CPU (Z8003 or Z8004) monitors its $\overline{\text{ABORT}}$ input during each bus transaction that is generated. If the $\overline{\text{ABORT}}$ input is asserted during clock cycle $T_2$, then the currently executing instruction is aborted. If no abort is indicated, but input $\overline{\text{WAIT}}$ is asserted, the $\overline{\text{ABORT}}$ input is tested during each added $\overline{\text{WAIT}}$ cycle ($T_2$). When input $\overline{\text{ABORT}}$ is asserted, the $\overline{\text{WAIT}}$ input must also be asserted for five cycles to permit the CPU internal control mechanism to abort the current instruction. When the $\overline{\text{WAIT}}$ input is deasserted, the CPU will acknowledge any pending interrupt request. The memory management circuit that caused the instruction abort must also initiate an interrupt. Input $\overline{\text{SAT}}$ is provided for this purpose on the Z8003, but any interrupt input can be used with either the Z8003 or the Z8004. This interrupt will initiate the software routine that will bring into memory the required information and restart the interrupted mainstream program at the exact point of interruption. Care must be taken in the selection and use of the interrupt associated with an abort to prevent a higher-priority interrupt from occurring and being processed between the abort instruction function and the processing of the interrupt associated with the abort.

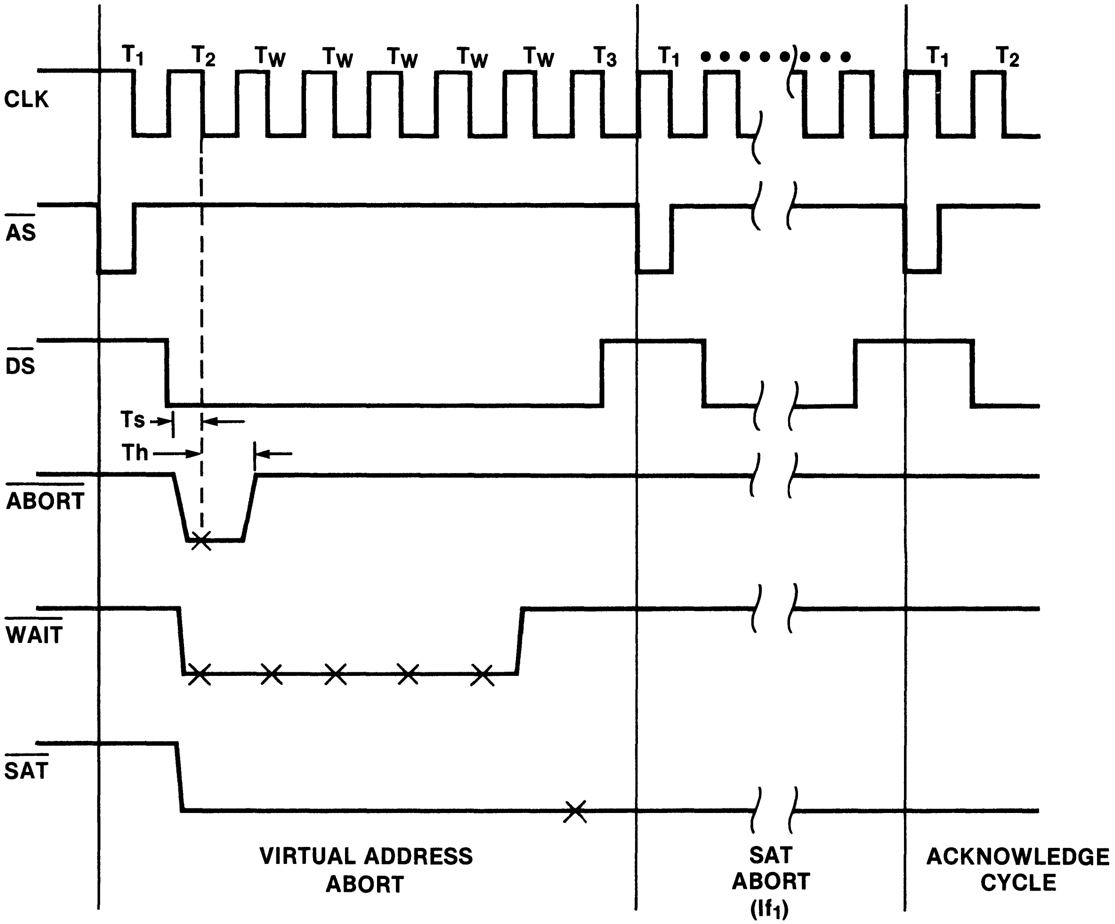
## 9.8 RESET

A hardware reset puts the Z8000 into a known state and initializes selected control registers of the CPU to system specifiable values (as described in Chapter 7). A reset will begin at the end of any clock cycle, if the $\overline{\text{RESET}}$ line is low.

A system reset overrides all other operations of the CPU, including interrupts, traps, bus requests, and stop requests. A reset should be used to initialize a system as part of the power-up sequence.

Within five clock cycles of the $\overline{\text{RESET}}$ line becoming Low (Figure 9-12) $AD_0$-$AD_{15}$ are 3-stated, $\overline{\text{AS}}$, $\overline{\text{DS}}$, $\overline{\text{MREQ}}$, $\overline{\text{BUSACK}}$, $\overline{\text{MO}}$, and $ST_0$-$ST_3$ are forced High, and $SN_0$-$SN_6$ are forced Low. The R/$\overline{\text{W}}$, B/$\overline{\text{W}}$ and N/$\overline{\text{S}}$ lines are undefined. Reset must be held Low for at least five clock cycles.

After $\overline{\text{RESET}}$ has returned High for three clock cycles, consecutive memory-read transactions are executed in System mode to intitalize the Program Status registers. These correspond to the memory accesses described in Chapter 7.



NOTE * = Clock Sample Points

Figure 9-11. Abort Request Timing

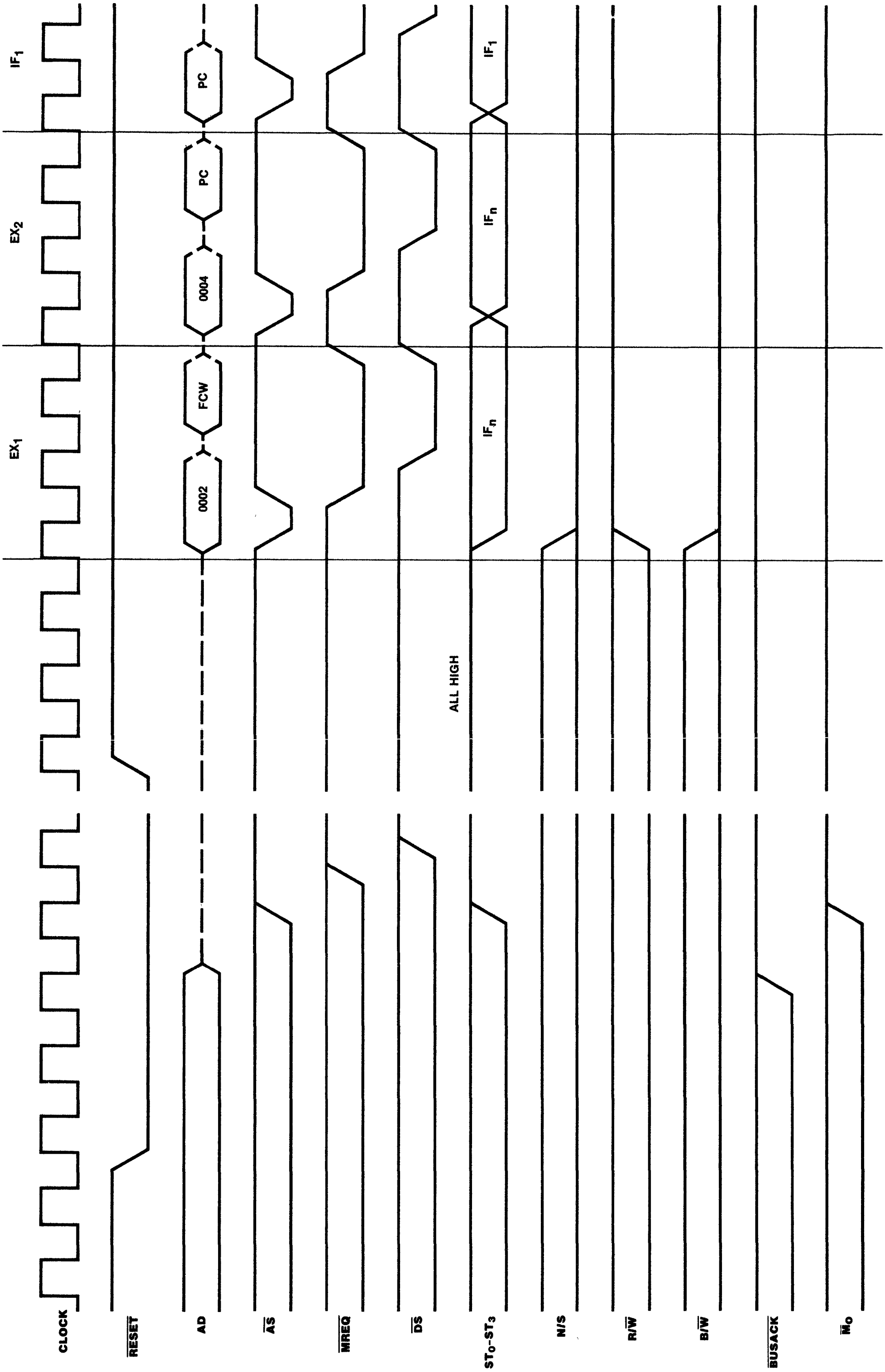


**Figure 9-12. Reset Timing**

# Chapter 10
# Programming Techniques

## 10.1 INTRODUCTION

The purpose of this chapter is to demonstrate how the features of the Z8000 CPU can be used to solve typical software problems. The first half focuses on specific programming techniques (through Section 10.12). In the second half, fully worked-out programs are presented for several important or illustrative problems.

A goal of programming is to allow computer users to deal with the high-level operations of their applications and to escape from the details of machine design and behavior. Many programming techniques have been designed with this goal in mind. This section introduces some widely used programming techniques and shows how they are implemented using the Z8000 architecture and instruction set.

## 10.2 DATA TYPES

All computer applications are based upon the interpretation of collections of bits--as numbers, text, logical flags, and so forth. The term data type refers to a bit collection of specified size and interpretation.

Every computer provides direct support for some data types, and the programmer provides programs to support the manipulation of other desired data types. The Z8000 architecture provides direct support for several frequently used data types and the instructions for performing the operations associated with them. These are described below.

**Bits.** A two-valued logical flag is the simplest useful interpretation of a bit collection, and its natural size is one bit. Unlike many earlier computers, the Z8000 has instructions that allow any bit in memory or in any general-purpose register to be set, tested, or cleared. Thus, any bit can be used as a logical flag, and flags can be packed into words or bytes without undue increase in processing overhead. An important application of this idea is a bit table, an array of 1-bit logical flags stored in consecutive bits of consecutive bytes of memory.

**Digits.** An important bit collection is a number, and an important special case of numbers is a decimal or hexadecimal digit. These are most conveniently represented by collections of four bits (occasionally referred to as nibbles). The Z8000 supports digits with the RRDB, RLDB, and DAB instructions, and the D and H bits in the Flags register.

**Bytes.** A collection of eight bits is called a byte. Almost all Z8000 instructions that take arguments have byte versions. (The Push, Pop, Multiply, and Divide instructions are the only important exceptions.) The two principal interpretations of bytes are as signed whole numbers and as codes for text characters. These interpretations are not enforced by the hardware, but some Z8000 features are designed with one or the other interpretation in mind. For example, the Translate and Test instruction and the P (parity) bit in the Flags register support the text data type, while the arithmetic instructions support signed whole numbers. The Z8000 has 16 byte registers.

**Words.** A collection of 16 bits is called a word. Almost all argument-taking instructions have word versions. (The Block Translate and Test instructions and the Decimal Arithmetic Support instructions are the only exceptions.) The principal interpretations of words are as signed and unsigned whole numbers, Z8000 instructions, index values, and nonsegmented addresses. The Z8000 provides 16 word registers.

**Long Words.** A collection of 32 bits is called a long word. The principal interpretations of long words are as segmented addresses and as signed and unsigned whole numbers. The Z8000 provides long-word versions of its Load, Push, and Pop instructions and supports 32-bit signed whole numbers with long-word versions of its four main arithmetic operations: add, subtract, multiply, and divide. The Z8000 provides eight long-word registers.

**Quadruple Words.** The Long Multiply and Long Divide instructions involve the use of 64-bit signed whole numbers. Four quadruple-word registers are provided for this purpose.

In addition to these data types, several other collections of bits are manipulated by certain Z8000 instructions.

**Addresses.** The LDA and LDAR instructions generate and save addresses. Addresses are words or long words, depending upon the segmentation mode of the CPU at the time of execution.

**Register Sets.** The LDM instruction manipulates register sets during the movement of information between general-purpose registers and memory. A register set consists of from 1 to 16 words stored in contiguous memory locations or in consecutive word registers.

**Data blocks.** The Z8000 block instructions manipulate data blocks, which can be from 1 to 65,536 words or bytes stored in contiguous memory locations. An important special case of a data block is a text string.

As subsequent examples illustrate, this large selection of data types offers Z8000 programmers simple approaches to solving a wide variety of programming problems.

## 10.3 ADDRESSING MODES

The Z8000 addressing modes were chosen and designed with the programmer's needs in mind. Here is a brief summary of the ideas behind these modes.

**Direct Addressing.** With Direct addressing, the actual memory address of the argument is contained in the instruction. This is especially useful in programs assembled by hand and in "patches."

**Register Addressing.** This addressing mode allows fast access to intermediate results. Almost all two-operand instructions require the use of register addressing for one of the operands.

**Immediate Addressing.** Immediate addressing is similar to direct addressing, but the actual value of the argument rather than its address is contained in the instruction. Immediate addressing can only be used for source arguments.

**Indirect Register Addressing.** In this mode the address of the argument is in an address register (a word or long-word Register, depending upon the segmentation mode). Its variants, the Autoincrement and Autodecrement modes, are used with the Push and Pop instructions to implement stacks, and with the block instructions to effect operations on sets of contiguous words or bytes in memory. Indirect Register addressing is used when addresses are passed as arguments to subroutines and to implement more elaborate access techniques, such as linked lists. Figure 10-1 is a simple

```
LDA RR2,X               !RR2 = address of text array!
LOOP: LDB RHO,@RR2      !Fetch next character!
      TESTB RHO
        JR Z,ENDLP      !Done when NUL reached!
                        !(Modify the character)!
      LDB @RR2,RHO      !Replace character by modified character!
      INC R3            !Point at next character!
      JR LOOP
ENDLP: . . .
```

**Figure 10-1. Example of Indirect Register Addressing**

example of its use--a loop to read successive bytes of memory until a zero terminator is found and to replace each byte with a modified value.

In this example, RR2 is used as an address register to point at (that is, contain the address of) successive bytes of a text string. Notice that the instruction

```
INC R3
```

is used to point to the next byte. This takes advantage of the way segmented addresses are stored in registers but assumes that the text string does not extend outside of the memory segment. A later example deals with arrays that extend beyond one segment.

Notice also that the instruction

```
LDA RR2,X
```

is used to set the contents of the address register RR2. An alternative instruction is

```
LDL RR2,#X
```

but it should be avoided, because it needlessly ties the code to a specific segmentation mode.

**Index Addressing.** In the Index addressing mode, a fixed address is stored in the instruction and a displacement is stored in a register. This is required when an array is being processed using a varying index. For example, consider the following FORTRAN instructions:

```
   DO 13 I = N1,N2
13 TABLE(I) = TABLE(I)+I
```

This can be implemented using Index addressing as shown in Figure 10-2.

Assume that the registers have been set:

```
R0 contains N2
R1 contains N1  (R1 will be I)
```

Two-dimensional arrays can be handled easily by a program that computes the offset associated with an index pair. For example, suppose that the M x N array of bytes TABLE is stored consecutively in memory as follows:

```
TABLE(1,1), TABLE(2,1),...,TABLE (M,1),
TABLE(1,2),...
```

Each column is a one-dimensional array, and these one-dimensional arrays are stored end to end in contiguous bytes of memory. (This format is standard in FORTRAN.) A two-dimensional array can be viewed as a one-dimensional array of dimension MN, and the element TABLE(I,J) of the two-dimensional array is the element TABLE([J-1]*M+I) in the one-dimensional array. If R1 contains I-1, R2 contains J-1, and R3 contains M, then the following code loads TABLE(I,J) into RH0:

```
LD R5,R2                !RR4 = (xxx,J-1)!
MULT RR4,R3             !RR4 = (0,[J-1]*M)!
ADD R5,R1               !R5  = [J-1]*M+[I-1]!
LDB RH0,TABLE(R5)
```

This code assumes that MN ≤ 65,536. If this is not true, then Index addressing cannot be used directly and the assumption that the columns of TABLE are stored end-to-end cannot be made. Instead, J is used as an index to a table of memory addresses (called a "dope vector"), and

```
!
     LD R3,R1              !Use R3 for actual offset!
     SLA R3                !Assume two-byte entries!
LOOP:CP R1,R0              !Is I > N2 yet?!
       JR GT,DONE          !Done if so!
     LD R2,TABLE-2(R3)     !TABLE(I) - FORTRAN arrays start at 1!
       ADD R2,R1           !TABLE(I)+I!
       LD TABLE-2(R3),R2   !Replace original TABLE(I) value!
       INC R1              !Increment I!
       INC R3,#2
       JR LOOP
DONE:     . . .
```

**Figure 10-2. Use of Index Addressing**

each of these addresses is the start of the corresponding column. If R1 contains I-1, R2 contains J-1 and the table of column base addresses is at an address contained in RR4, then the following code loads TABLE (I,J) into RH0:

```
LD R3,R2
SLA R3,#2           !R3 = 4*(J-1)!
LDL RR6,RR4(R3)     !RR6 = address of Jth column!
LDB RH0,RR6(R1)
```

This code uses Base Index addressing (see below). It is so efficient that it can be used even when MN $\leq$ 65,536.

For nonsegmented operation, Index addressing can be used to simulate Base addressing (see below), since addresses and offsets are both 16 bits. For example,

```
ADD R0,8(R15)
```

adds the fifth word of the stack to R0. (NOTE: If separate data and stack spaces are used, this technique does not work. When R15 is used in the Index addressing mode, the status outputs $ST_3$-$ST_0$ reflect data reference, not stack reference.)

In segmented mode, the same technique can be used if the segment number is known when the program is assembled. For example, if the stack has been assigned to segment 12 (that is, R14 contains 0C00), then

```
ADD R0,<<12>>8(R15)
```

adds the fifth word on the stack to R0.

Use of Index addressing to simulate Base addressing is helpful because Base addressing is available only with the Load instruction.

**Base Addressing.** Base addressing specifies the address of an argument as the sum of a displacement contained in the instruction and a base address contained in an address register. For example,

```
LD R0,RR14(#8)
```

can be used in segmented mode to access the fifth word of the stack.

The Base addressing mode is the key to a subroutine argument-passing convention that uses a stack (see Section 10.4).

Base addressing is useful in accessing items in records or more general data structures of predefined format, especially when the address of the record in memory is not known in advance. For example, if a number of 80-character records have been read into memory end to end starting at a location specified in RR2, then the code shown in Figure 10-3 steps through the records until one is found in which the seventy-third character is equal to 41H.

**Base Index Addressing.** Base Index addressing takes both the base address and the displacement from registers. One example of it was shown above in the code to handle large two-dimensional arrays. Other examples are shown in this chapter.

Base Index addressing is also useful in a generalization of the record or data structure example given in Figure 10-3. For example, if the termination condition were the presence of 41H in any of positions 73 through 80, the code of Figure 10-3 would appear as shown in Figure 10-4.

**Relative Addressing.** This is a variant of Base addressing in which the base register is always the Program Counter. It helps the programmer produce position-independent code (see Section 10.6), and it leads to more compact code in many cases. Also, if separate data and instruction memories are used, the LDR instruction is the only

```
LOOP:   LDB RH0,RR2(#72)     !Get 73rd character!
        CPB RH0,#%41         !Compare with %41!
          JR EQ,ENDLP        !Done if equal!
        ADD R3,#80           !Otherwise, point at next record!
        JR LOOP
ENDLP:  . . .
```

**Figure 10-3. Use of Base Addressing**

```
LOOP:    LD R4,#72              !Set to 73rd position!
LOOP1:   LDB RH0,RR2(R4)        !Get R4th character!
         CPB RH0,#%41           !Compare with %41
           JR EQ,ENDLP          !Done if equal!
         INC R4                 !Otherwise, give R4 next index!
         CP R4,#80              !Compare position with last!
           JR LT,LOOP1          !If not past last, try next position!
         ADD R3,#80             !Otherwise, point at next record!
           JR LOOP
ENDLP:   . . .
```

**Figure 10-4. Use of Base Index Addressing**

way to refer to a constant that is assembled as part of the program (except immediate data in instructions).

Further examples using the Z8000 addressing modes are given in the following sections.

## 10.4 STACKS

A stack is a last-in, first-out (LIFO) buffer of finite but unspecified size. It is like a stack of plates on a table in a room: plates can only be added to or removed from the top and while there is no preset maximum number of plates, the room does have a ceiling. Sometimes the metaphor used is a stack of plates on a spring in a well (as at a steam table); this accounts for the names PUSH and POP used for the operations of adding or removing items, but in the usual computer implementations the items stay fixed like plates on a table.

In the Z8000, stacks are implemented as arrays of declared fixed sizes, but an external memory-mapping facility allows stacks to be open ended, with additional memory allocations made as needed. The Push and Pop instructions are designed to work with stacks that grow downward; that is, the first item on the stack occupies the highest-numbered memory location. Programs, on the other hand, grow upward; that is, as each instruction is added to the program or as program modules are linked together, higher and higher-numbered addresses are used. This provides an efficient way for a program and a stack to share a given block of memory. The program can begin at the lowest-numbered address and grow upward as developments increase its size; the stack can begin at the highest-numbered location and grow downward as the program is executed. This is the most flexible and efficient use of the space. If there is room for both the program and the stack in memory, then memory is automatically allocated successfully.

A stack in the Z8000 uses an address register to keep track of the location of the top item (the lowest-numbered item). The stack register always contains the address of the top item because of the way PUSH and POP work. PUSH first decrements the stack register by 2 or 4, causing it to point at the next free word or long-word location and then stores its argument at that location. POP first fetches the item pointed to by the stack register, then increments the stack register.

Reference to items on a stack can be made using the Base or Base Index addressing mode. For example, if RR4 is a stack register, then RR4(#0), RR4(#2), and RR4(#4) refer to the top, second, and third words on the stack, respectively. Also, as previously explained, Index addressing can be used to refer to stack items when the stack's segment number is known at assembly time. Reference to stack items is illustrated in Section 10.7, Subroutines.

The most common use of stacks is for dynamic allocation of temporary storage space. The two pieces of code in Figure 10-5 show how a program can accumulate words for future processing. The first uses fixed temporary storage; the second uses a stack.

```
  !Accumulating words in a fixed buffer!

       CLR R4          !Word counter!
       LDA RR2, BUF    !RR2 always points at next free location!
  LP:  CALL GETWD      !Get next word!
         JR C,DONE     !If C set, no more to get!
       LD @RR2,R0      !Store word, increment pointer!
       INC R3,#2
       INC R4          !Count the word!
       JR LP

DONE:   . . .

  !Accumulating words on a stack!

       CLR R4          !Word counter!
  LP:  CALL GETWD      !Get next word!
         JR C,DONE     !If C set, no more to get!
       PUSH @RR2,R0    !Store word, increment pointer!
       INC R4          !Count the word!
       JR LP

DONE:  . . .
```

**Figure 10-5. Accumulation Of Words Within A Fixed Buffer And On A Stack**

In the first piece of code, a buffer called BUF is allocated to the program at assembly time. Each time this code is executed, words are stored in this buffer, starting at the beginning of the buffer. The second piece of code has no storage of its own; every time it is executed it stores words on the stack controlled by RR2. It is assumed that the system initializes this stack before the process including this code begins running.

Using a stack in this way has several advantages:

- The total amount of space needed by the stack is usually less than the amount required by fixed allocation.

- Storage mangement is separate from the implementation of the function. This tends to simplify the implementation of functions.

- Program functions can be encoded in ROM more easily and management of RAM can be localized.

- It is easier to make program functions shareable (see below); in the preceding example, several different sets of words might have been accumulated in different parts of the stack by different calls on the code. This would not be possible with the fixed-buffer accumulation.

There are also some disadvantages to using stacks in this way. In general, programs that use a stack must leave it exactly as they found it; every item pushed onto the stack must be popped off before completion of the program. This is because the same stack used by the program that calls the given program is also used by programs called by the given program. For example, consider the following code:

```
PUSH @RR4,R0
CALL SUBR
POP R0,@RR4
```

This is a common means of saving a value, in this case R0, that would otherwise be destroyed by the intermediate operation, in this case CALL SUBR. But this procedure fails if the SUBR routine does not leave the stack controlled by RR4 exactly as it was found.

The requirement that each program regulate its stack use can make checkout difficult, since a subroutine's failure in stack management can lead to anomalies in the behavior of the calling program. The symptom and cause can be in seemingly unrelated portions of the program. Also, there is a dedicated stack register used for subroutine calling; failure in its management can cause symptoms that are difficult to recognize and usually interferes with the standard checkout procedures.

Dynamic allocation of temporary storage leads to another checkout problem: it is difficult to examine memory after the fact to look for the causes of anomalous behavior. A desired piece of information may have been overwritten, and it is difficult to determine where a given program stored its intermediate or temporary data.

In general, stack use is not as flexible as the use of dedicated storage. For example, in the preceding code, once the words are accumulated in BUF, they are processed any way the programmer desires. Index addressing of the form

```
ADD R1,BUF(R2)
```

makes the fixed buffer a random access memory. With a stack, on the other hand, only the top item is easily available. Other items can be accessed using Base or Base Index addressing of the form

```
LD R1,RR6(#2)
LD R2,RR6(R3)
```

If the stack segment number is known when the program is assembled, the Index addressing mode can be used, as in the preceding ADD example. For example:

```
ADD R1, <<seg no>>2(R7)
```

adds the next-to-last word received to R1.

The stack addressing methods described allow items in memory to be examined without giving up their places (as happens with POP), but the offsets (#2 or R3 in the above lines) are measured from the top of the stack, that is, from the last item placed there. To process the items in a first-in, first-out (FIFO) order requires a complicated computation that can lead to errors. For example, referring to the sample code of Figure 10-5 for accumulating words on a stack, Figure 10-6 shows the code at DONE that allows the words to be examined in the order received.

Stack initialization is straightforward. The stack register must be set to the address one word above (that is, at a higher-numbered address than) the first word to be used by the stack. This works regardless of whether words or long words are used. (In fact, there is no problem with mixing words and long words on a stack, as long as any item pushed with a PUSHL instruction is popped with a POPL instruction.) So, for example, if a stack uses locations F000-FFFF of segment 6, the first word used by the stack is at location FFFE. The stack register should be initialized to segment 6, offset zero.

Boundary protection has two aspects: overflow and underflow. Overflow occurs when all locations assigned to a stack have been filled and another push is attempted. Underflow results from an attempt to pop items from an empty stack. The Push and Pop instructions provide no direct support for boundary protection. This is achieved in software by using push and pop subroutines that check for overflow or underflow before pushing or popping. An external memory management facility can also help detect stack overflow.

The preceding discussion applies to all stacks in the Z8000. The Z8000 automatically uses stacks

```
DONE:     SLA R4          !Multiply by # bytes/word!
           JR Z,FINIS     !No words to examine!
GETNXT:   DEC R4,#2       !Convert count to offset!
          LD R0,RR2(R4)   !Fetch the word from the RR2 stack!
          !(Process the word)!
          TEST R4         !R4 contains # of bytes remaining!
          JR NZ,GETNXT

FINIS:    . . .
```

**Figure 10-6. Examination Of Words In The Order Received**

for subroutine calling and for saving CPU status on traps and interrupts, and for these purposes an implicit stack register is used. The implicit stack register is R15 for nonsegmented operation and RR14 for segmented operation. Furthermore, there are two copies of the implicit stack register, one for system mode operation and one for Normal mode. In ordinary operation, each is referred to as R15 or RR14, but when referring to the Normal mode stack register while operating in System mode, the LDCTL instruction is used with the argument NSP (in nonsegmented operation) or the arguments NSPSEG and NSPOFF (in segmented operation). It is not possible to refer to the System mode stack register while operating in normal mode.

There are several points about this implicit stack register that are important to understand:

- When the implicit stack register is used as an address register (that is, in a Push or Pop instruction in the Indirect Register mode) or as a base register in the Base or Base Index modes, the status lines $ST_3$-$ST_0$ reflect stack reference status rather than data reference status.

- An interrupt can occur between the execution of any two Z8000 instructions (or even between repetitions in the block instructions). The System mode implicit stack register is used for saving the CPU status, so it must never contain a higher-numbered address than that of any location containing stack data.

- The Normal mode implicit stack register is not involved in the processing of interrupts, but it is used for saving subroutine return addresses in Normal mode. Therefore, whenever a subroutine call is made while operating in Normal mode, the Normal mode implicit stack register must not contain a higher-numbered address than that of any location containing stack data.

Although the significance of these points may not be immediately obvious, they need to be considered when the stack is used other than as a last-in, first-out (LIFO) buffer accessed only with Push and Pop instructions.

One approach to processing stack items in an order other than last-in, first-out is to alter the value of the stack register temporarily. For example, after pushing five words onto the stack, one might wish to increment the stack register by 10 and step through the words in the order received, decrementing the stack register by two before each access. At the end of this process, the stack register returns to its correct value. This works with any other stack (assuming no pushes or pops are done on it during the processing), but with the System mode implicit stack register, any trap or interrupt causes CPU status to overwrite a portion of the five words being processed. This technique can be used with the Normal mode implicit stack register provided that no subroutine calls are executed in the course of processing.

One approach to processing stack items that avoids these problems is to move the stack register contents into some other address register and then treat the stack data in question as an array (or other data structure) addressed by the new address register. Additional pushes and pops on the stack (such as those caused by traps, interrupts, or subroutine calls) are then handled correctly without affecting the processing of the stack elements. There are two potential problems with this approach:

- When the contents of the implicit stack register are moved into another address register and the other register is used for referring to the stack items, the status outputs $ST_3$-$ST_0$ will show data reference. Thus, this technique cannot be used without modification if the status outputs are used for directing references to separate data and stack memories.

- The programmer must be careful in using addresses that point into the stack. Since the stack storage is allocated dynamically, the same stack memory locations can be used in other ways that change their contents. Naturally, a change to the stack location contents before they are completely processed can only occur as the result of a programming error, but this sort of error is easy to make, especially if a stack management scheme is being used. Furthermore, there is no way to determine by examination of the saved stack address whether the contents are still valid.

A similar technique, subject to the same potential problems, is to use the stack for temporary storage of an array, character string, or other data structure and to pass the address of that structure to a utility subroutine for processing.

The called program generally does not use the implicit stack register as an address register for processing the structure.

Since the Z8000 architecture does not allow words to be stored at odd addresses, and since an interrupt can occur at any time, the system mode implicit stack register must never contain an odd address. For this reason, pushes and pops of bytes cannot be allowed on the system mode implicit stack register. This is most easily done by providing for no byte Push and Pop instructions.

Saving byte registers can be accomplished by saving the entire word register. Restoring byte registers without disturbing the other half of the word register must be simulated. For example, if

```
PUSH @RR8,R0
```

is used to simulate PUSHB @RR8,RL0, then POPB RL0,@RR8 can be simulated by

```
LDB RL0,RR8(#1)
INC R9,#2
```

## 10.5 CONDITION CODES

Condition codes are names for logical combinations of flags bits. There are eight such combinations and an opposite for each, for a total of 16 condition codes. Of the eight, one is "always true"; four are single-bit combinations (C = 0, V = 0, S = 0, Z = 0), and three are multi-bit combinations [S XOR V = 0, Z OR (S XOR V) = 0, C OR Z = 0].

Because the condition codes are designed for use in a variety of Z8000 applications, some of these combinations have more than one name. Following are some typical applications and the condition code names associated with them.

**Arithmetic Result Testing.** An arithmetic operation (for example, ADD R0,R1) is performed and the result is used for conditional control (for example, a branch).

| **Code** | **Meaning** | **Opposite Code** | |
|---|---|---|---|
| Z | Result is Zero | NZ | Non-Zero |
| MI | Result is negative (MInus) | PL | Plus |
| C | Carry (or borrow) occurred | NC | No Carry |
| OV | OVerflow occurred | NOV | No OVerflow |

**Logical Result Testing.** A logical operation (for example, AND R0, R1) is performed and the result is used for conditional control.

| **Code** | **Meaning** | **Opposite Code** | |
|---|---|---|---|
| Z | Result is Zero | NZ | Non-Zero |
| PE | Parity is Even (byte op only) | PO | Parity Odd |

**Arithmetic Comparison.** Two arithmetic values are compared by CP a,b (for example, CP R0,R1). The relationship between the values is to be determined.

| **Code** | **Meaning** | **Opposite Code** | |
|---|---|---|---|
| EQ | a = b Equal | NE | Not Equal |
| LT | a < b Less Than | GE | Greater or Equal |
| LE | a ≤ b Less than or Equal | GT | Greater Than |

**Unsigned Arithmetic Comparison.** Two unsigned values (for example, addresses) are compared by CP a,b (for example, CP R0,R1). The relationship between the values is to be determined.

| **Code** | **Meaning** | **Opposite Code** | |
|---|---|---|---|
| EQ | a = b Equal | NE | Not Equal |
| ULT | a < b Unsigned Less Than | UGE | Unsigned Greater or Equal |
| ULE | a ≤ b Unsigned Less than or Equal | UGT | Unsigned Greater Than |

**Miscellaneous Situations.** There are many Z8000 instructions (for example, $\overline{\text{MREQ}}$, shift instructions, block instructions) that set specific flags bits in other ways. Also, the programmer can use the flags bits for passing information between routines. SETFLG and RESFLG are provided for this purpose and any of the 16 combinations can be tested using any of the available names.

| **Code** | **Meaning** | **Opposite Code** |
|---|---|---|
| LT | S XOR V = 1 | GE |
| LE | (S XOR V) OR Z = 1 | GT |
| ULE | C OR Z = 1 | UGT |
| OV,PE,V* | V = 1 | NOV,PO,NV* |
| MI,S* | S = 1 | PL,NS* |
| Z,EQ | Z = 1 | NZ,NE |
| C,ULT | C = 1 | NC,UGE |

(*V, NV, S, NS not recognized by all assemblers)

It is important to understand the operation of the Test instruction. TEST sets S and Z to reflect the value of its argument; that is, S is set if the high-order bit of the argument is set, and Z is set if the value of the argument is not set. The only other bit set is P/V. For byte arguments it is set to reflect the parity, for long-word arguments it is undefined, and for word arguments it is unaffected. C is always unaffected by TEST.

MI and EQ are the only condition codes solely dependent upon Z and S, so there is no easy way to determine whether the tested argument is less than or equal to zero. There are several ways around this:

- CP a, #0 can be used instead of TEST a; C, Z, S, and V will be set according to their arithmetic meanings. This works for byte, word, or long-word arguments.

- For word arguments only, if V is clear, TEST a can be used; if a ≤ 0, then LE is true.

- TEST a can be followed by two tests:

```
       TEST a
         JR LT,X
         JR EQ,X
       (come here if a > 0)
       .
       .
       .
    X: (come here if a ≤ 0)
```

This works for byte, word, or long-word arguments.

It is often desirable to postpone the testing of a condition until after the execution of instructions that must be performed regardless of the outcome of the test. For this reason, Z8000 instructions do not change the settings of the flag bits except to report the outcomes of their operations. In particular, the transfer instructions (CALL, CALR, JP, JR, RET) and the data-moving instructions (CLR, LD, EX, SET, TCC, etc.) do not affect the flags bits. For example, in the code of Figure 10-7, the result of the addition is stored via the pointer RR4, regardless of the values of the flag bits.

```
ADD R0,R1
LD @RR4,R0
  JR OV,W
  JR Z,X
  JR MI,Y
!(otherwise come here)!
```

**Figure 10-7. Test Instructions**

If the LD @RR4,R0 instruction affected the flags bits, it could not be placed before the tests. Instead, a LD @RR4,R0 instruction would have to appear at each of the four locations to which control might pass as a result of the testing, and the code would take the form shown in Figure 10-8.

```
      ADD R0,R1
        JR OV,W
        JR Z,X
        JR MI,Y
      LD @RR4,R0
      .
      .
      .
W:    LD @RR4,R0
      .
      .
      .
X:    LD @RR4,R0
      .
      .
      .
Y:    LD @RR4,R0
      .
      .
      .
```

**Figure 10-8. Example With LD@RR4,R0 Instruction**

If, however the example in Figure 10-7 required the unconditional execution of

```
INC R5,#2
```

after the LD instruction (to point RR4 at the next word of storage), the INC instruction could not have been placed before the conditional JR instructions, since INC affects Z, S, and V. (However, POP R0,@RR4 would solve that difficulty.)

To avoid duplicating the increment instruction at each of four locations in the program, the Flags register can be saved and restored as follows:

```
LDCTLB  RH6,FLAGS
INC R5,#2
LDCTLB  FLAGS,RH6
```

The saving and restoring of the Flags register is not a privileged operation.

One important use of flags bits is based upon the ability to postpone testing: passing information back from subroutines. For example, consider the routine in Figure 10-9.

This routine might be called in a sequence like

```
LP:    CALL GETCH        !Get next char into RL0!
       CALL TSTHEX       !Is it hex?!
         JR C,X          !C=1 means "no"!
      !(Code for the case:  char is hex)!
       JR LP
X:    !(Code for the case:  char not hex)!
       JR LP
```

There are several advantages in using condition codes this way:

- Registers are undisturbed. The flag bits are usually available, since they cannot be used for long-term storage. If registers are used to pass this kind of information, additional instructions are necessary for saving and restoring previous register values.
- The calling routine can ignore the information if it is irrelevant to the specific case. This is in contrast to the commonly used technique of signaling different conditions by returning to different locations (for example, to the first or second word after the call).

```
!Test the ASCII character in RL0 to see whether it is a hex digit.

  CALL TSTHEX; RL0 = the character
  Return with registers unchanged and C=0 if a digit, C=1 if not.
!
ASCZER=%30; ASC9=ASCZER+9    !0-9 range!
ASCA=%41;   ASCF=ASCA+5      !A-F range!

TSTHEX:  CPB RL0,#ASCZER     !Compare with "zero"!
           JR ULT,NOTHEX     !All digits are > "zero"!
         CPB RL0,#ASC9       !In "0" to "9" range?!
           JR ULE,ISHEX      !Yes--success!
         CPB RL0,#ASCA       !Now try "A" (> "zero")
           JR ULT,NOTHEX     !Between "9" and "A"--fail!
         CPB RL0,#ASCF       !In "A" to "F" range?!
           JR ULE,ISHEX      !Yes--success!
NOTHEX:  SETFLG C            !Return C=1!
         RET
ISHEX:   RESFLG C            !Return C=0!
         RET
```

**Figure 10-9. Example, Testing an ASCII Character**

This difference is especially important if the return of an error condition is being added to an existing routine. In this case, existing calls do not need to be modified immediately.

- The use of flag bits takes advantage of the Z8000's conditional instructions. Any scheme other than "returns to different locations" has to be followed by a testing procedure, which would involve the use of flag bits anyway.

The technique of using flag bits to return information from subroutines can be adapted for use with "system call" routines as well, so a sequence such as the following is possible:

```
SC #HXTEST
JR C,X
```

This sequence cannot be accomplished by using the SETFLG and RESFLG instructions in the system routine. System routines called through the SC mechanism behave like interrupt routines: CPU status (including flags) is saved on the R15 or RR14 stack when the SC is executed, and it is restored from the stack when the IRET is executed. Therefore, the copy of flags saved on the stack must be modified to reflect the desired returned settings. Modification of stack locations by called programs is tricky. For example, when the SC trap first occurs, the saved FCW is the second word on the stack; it can be accessed as R15(#2) or RR14(#2). If the SC handling program then calls the subroutine corresponding to the given index (#HXTEST in the example above), the subroutine return is stored on the stack. Access to the saved FCW is then done as R15(#4) or RR14(#6). If the called subroutine begins by saving registers, the offset changes again. For example, after a

```
PUSHL @15,RR0
```

or a

```
PUSHL @RR14,RR0
```

the new offsets become R15(#8) and RR14(#10). Similarly, each time the processing routine calls a subroutine or uses the stack for temporary storage, the situation changes.

Not only is changing the FCW value saved on the stack potentially error prone, but the type of error that can occur is serious. Thus, change to the saved FCW value is better done by the SC-dispatch routine, the routine whose address appears in the program status area entry corresponding to the SC trap. An SC-dispatch routine to accomplish this is shown in Figure 10-10.

Many variations on this dispatch mechanism are possible, depending on the system in which it functions. This example illustrates the use of condition codes, but is not a model SC dispatcher.

## 10.6 POSITION-INDEPENDENT PROGRAMS

A position-independent program is one that can be moved to different locations in memory without changing its behavior. The instructions and program constants are in a fixed order, but their

```
SCDISP: EX R13,@RR14           !Save RR12, get "reason" into R13!
        PUSH @RR14,R12
        PUSH @RR14,R0          !Use RL0 to pass saved FLAGS!
        LDB RL0,RR14(#7)       !(Offset of FLAGS is 7 after above
                                saves)!
        !(Code to compute processing subroutine
          address from "reason" and leave it in RR12)!
        CALL @RR12             !Call processing routine!
          JR C,NOMSG           !Dont use updated FLAGS!
        LDB RR14(#7),RL0       !Update flags on stack!
NOMSG:  POP R0,@RR14           !Restore R0!
        POP R12,@RR14          !Restore RR12!
        LD R13,@RR14
        IRET
```

**Figure 10-10. Example of Using An SC-Dispatch Routine To Change To A Saved FCW**

behavior does not depend upon the actual addresses of the memory locations where they are stored.

An example of a position-independent program is the subroutine TSTHEX of Figure 10-9. Figure 10-11 contains an assembled version of this subroutine starting at location 1000H.

```
1000    0A08 3030       TSTHEX:  CPB RLO,#ASCZER
1004    E709                      JR ULT,NOTHEX
1006    0A08 3939                CPB RLO,#ASC9
100A    E308                      JR ULE,ISHEX
100C    0A08 4141                CPB RLO,#ASCA
1010    E703                      JR ULT, NOTHEX
1012    0A08 4646                CPB RLO,#ASCF
1016    E302                      JR ULE,ISHEX
1018    8D81            NOTHEX:  SETFLG C
101A    9E08                     RET
101C    8D83            ISHEX:   RESFLG C
101E    9E08                     RET
```

**Figure 10-11. Assembled Version Of Subroutine TSTHEX**

Because of Relative addressing, the hex values of the instructions remain the same wherever the program is assembled. This is true despite the fact that the symbols NOTHEX (at location 1018) and ISHEX (at location 101C) are referred to by instructions in the program. To understand this, consider the two instances of the instruction

JR ULT,NOTHEX

The hex values corresponding to these two instances are not the same, because NOTHEX is used in these two instructions simply as a convenience to the programmer. They are actually two different instructions:

JR ULT,$+%14

JR ULT,$+%8

In other words, these instructions do not rely on the fact that NOTHEX is at 1018H. Instead they require the destination to be 14 or 8 locations after the location containing the instruction.

Position-independent programs contribute in several ways to achieving modularity. One way is by using "silicon software." Imagine a set of programs, each available on a ROM, that provide a variety of software tools, such as a debugger, an editor, and a text-formatting program. If each of these programs is position-independent, the system designer can select from among these ROMs and assign a set of memory addresses to each, thus building a custom-tailored system. A variation of this idea is a "demand loading" memory system that loads position-independent programs from secondary storage into any available RAM area whenever calls are made on them.

As another example, consider a debugging program that can be loaded into RAM wherever space is available. For example, it could reside in a buffer area while the initialization code was executing and then move to overlay the initialization code while the program used the buffers.

These examples show some of the uses of position-independent programs. When writing position-independent programs, the main rule is, "Don't use addresses in instructions." Addresses in instructions are generally used in the Direct and Index addressing modes and as immediate arguments. Direct and Index addressing cannot be used in position-independent programs except when Index addressing is used as previously described to simulate Base addressing. The use of addresses as immediate arguments should be avoided. The same result can be achieved with the LDA and LDAR instructions.

Relative addressing--the CALR, JR, LDR, and LDAR instructions--is the principal tool available to the programmer writing position-independent programs. Another important tool is the use of fixed-location utilities called from position-independent programs. For example, in a demand-loading scheme, segment zero might be dedicated to routines that are always resident. If so, the first 256 bytes of segment zero can consist of subroutine entry points, and calls can be made on these subroutines by using Direct or Index addressing from position-independent programs. (The first 256 bytes of each segment can be addressed by using a short segmented address.) The system call trap can also be used to access system routines from position-independent programs.

Many variations on these ideas are possible, depending on what is to be fixed and what is to be position-independent. Use of the stack for temporary storage automatically achieves position independence of the data. If the stack is not used, position independence of data can be achieved using the LDAR instruction, the Indirect Register, or the Base and Base Index addressing modes.

The kind of position independence discussed here is an independence from the particular range of addresses assigned to the program. Another kind of position independence is provided by an external memory-mapping facility, which allows a given address range to correspond to different physical memory locations.

## 10.7 SUBROUTINES

The principal property of Z8000 subroutines is that they use RET as an exit so that they can be called from more than one place. Invocation of subroutines is accomplished with the CALL (or CALR) instruction. CALL and RET perform complementary functions. When a CALL (or CALR) instruction is executed, the address of the following memory location is saved on the RR14 or R15 stack. Then transfer is made to the address specified in the CALL instruction. When a RET instruction is executed, the address on top of the RR14 or R15 stack is popped into the PC; that is, it is removed from the stack and a transfer to that address is made.

In this way, the programmer can encode commonly used functions in one place and then make use of them by CALLs whenever they are needed. The CALL of the given subroutine is like another instruction added to the CPU's instruction set. This is the most important tool of the assembly language programmer; it allows instructions to be used that are relevant to the application at hand, thereby simplifying and clarifying assembly language programs.

The CALL and RET instructions provide the subroutine calling mechanism but do not dictate a specific means of argument passing. For example, if a subroutine is needed to compute the square root of a number, the programmer must decide how to specify that number to the subroutine. The programmer must also decide how the subroutine will report the answer.

There are three commonly used methods for argument passing:

- In a register
- On a stack
- In the program, in locations following the call

Each of these methods can be used to pass actual arguments or to pass the address of an argument table.

The return of answers to the calling program has four commonly used options:

- In a register
- On a stack
- By returning to addresses at varying offsets from the CALL
- By manipulating flag bits

The use of registers for subroutine argument passing and result returning is the most popular and most efficient option. For example, to implement the FORTRAN statement Y=SQRT(X) the following code can be used:

```
LDL RR0,X          !Get X!
CALL SQRT          !Compute square root!
LDL Y,RR0          !Store in Y!
```

Here the subroutine SQRT takes its argument in RR0 and returns the answer in RR0.

The code for a SQRT routine that takes arguments and returns results on a stack might be:

```
PUSHL @RR6,X
CALL SQRT
POPL Y,@RR6
```

This assumes that a stack controlled by RR6 is available for use in argument passing.

There are times when passing arguments on a stack is preferable to using registers. There might be more arguments than can be accommodated in the registers, or it might be desirable to make the subroutine re-entrant (see Section 10.8). When a stack is used for passing arguments, the subroutine usually uses the Base or Base Index addressing modes to refer to them. For example, suppose that the subroutine BIGSQRT accepts an array of 14 numbers on the RR6 stack and replaces each with its square root. The code might look like that of Figure 10-12.

```
BIGSQRT: LDK R2,#14         !Set argument Counter!
         CLR R3             !Initialize index!
LOOP:    LDL RR0,RR6(R3)    !Get next arg!
         CALL SQRT          !Compute square root!
         LDL RR6(R3),RR0    !Store it back!
         INC R3,#4          !Arguments are 4 bytes!
         DJNZ R2,LOOP       !Loop if more arguments!
         RET
```

**Figure 10-12. Example Using a Stack For Passing Arguments**

In nonsegmented operation or in segmented operation when the stack segment number is known at assembly time, Index addressing can also be used to refer to stack items. The passing of arguments by including them in the program following the CALL, and the return of status information by returning to addresses at varying offsets from the CALL, are illustrated in the following code:

```
CALL SQRT          !Compute square root!
X                  !Adr of argument!
Y                  !Adr at which to store result!
JR NEGX            !Error return: X was negative!
(Execution resumes here if no error)
```

The subroutine SQRT used with this sort of call might look like the one in Figure 10-13.

The code makes it apparent that this is an awkward means of passing information. It was originally developed for computers that had few registers and no multiple-word instructions and that stored their return addresses in the subroutines rather than on a stack. It is not well suited to the Z8000.

Often it is convenient to use an argument table whose address is passed to the subroutine. The subroutine refers to the table elements as it would to arguments on a stack--it uses Base or Base Index addressing. An example of such a table is given in Section 10.13.4

The flag bits provide a convenient means of passing error or status information back from a subroutine. Since RET does not affect any flag bits, a condition can be set in a subroutine and tested in the calling program. For example, the SQRT routine might use C to indicate that an error condition prevented it from computing a square root. The calling program might look like this:

```
LDL RR0,X          !Get the argument!
CALL SQRT          !Compute the square root!
 JR C,ERREX        !C set if error!
LDL Y,RR0          !Store the result!
```

```
SQRT:    LDL RR12,@RR14      !Get saved return!
         LDL RR2,@RR12       !Get address of X!
         LDL RR0,@RR2        !Then get X itself!
         INC R13,#4          !Step over adr of X!
         LDL RR2,@RR12       !Get address of Y!
         INC R13,#4          !Step over adr of Y!
         TESTL RR0           !Test X!
           JR MI,ERREX       !Error if X ≤ 0!
         INC R13,#2          !Step over error exit!
        !(Compute square root)!
         LDL @RR2,RR0        !Store in Y!
ERREX:   LDL @RR14,RR12      !Put updated Return adr on stack!
         RET
```

**Figure 10-13. Example, Subroutine SQRT**

## 10.8 RE-ENTRANT PROGRAMS

Often in computer systems, two or more distinct processes seem to be running simultaneously. Actually, the computer alternates between these processes, dropping each one in turn, then picking it up at the point at which it was dropped. Since the CPU's most fundamental resources are generally not duplicated, the two processes share them. For example, the values of the FCW and the PC being used for one process must be saved before they are set to the values appropriate for the next process. There are other resources that may need to be saved, such as the general-purpose registers and memory. The context of the processes is the total set of registers and memory that needs to be saved for each process when it is suspended and later restored. The operation of saving one context and restoring another is called context switching.

A re-entrant program is a program that can be used simultaneously by two or more processes. A program is re-entrant if, and only if, it refers only to registers and memory locations that are included in the process contexts.

One example of concurrent processes arises when interrupts are used. In this case, the CPU provides for the automatic saving of the PC and FCW. Let us assume that we are working with a system in which every interrupt-processing routine saves and restores RR0 and RR2. Figure 10-14 shows three pieces of code that form the basis of an extended illustration of how re-entrancy is achieved.

The routine MULTEN is re-entrant, since it refers only to registers and memory locations in the context assumed above. The references to RR14(#4) are to a location in the context. This is because the contents of RR14 (or R15 in the nonsegmented

```
Calling Program (in segment 6)

    100 93E3                       PUSH @RR14,R3    !Put argument on stack!
    102 5F00 0600 2000             CALL MULTEN      !Multiply it by 10!
    108 97E3                       POP R3,@RR14     !Return argument to R3!
    10A                            -------------

MULTEN Program (in segment 6)

   2000 31E1 0004        MULTEN:   LD R1,RR14(#4)   !Get argument!
   2004 BD2A                       LDK R2,#10       !Constant Multiplier!
   2006 9920                       MULT RR0,R2      !10 x argument!
   2008 33E1 0004                  LD RR14(#4),R1   !Replace arg with result!
   200C 9E08                       RET

Interrupt-Processing Program (in segment 8)

    600 91E0             IROUT:    PUSHL @RR14,RR0  !Save!
    602 91E2                       PUSHL @RR14,RR2  !  registers!
    604 31E0 0008                  LD R0,RR14(#8)   !Get "reason"!
    608 93E0                       PUSH @RR14,R0    !Compute!
    60A 5F00 0600 2000             CALL MULTEN      !  10 x "reason"!
    610 97E0                       POP R0,@RR14     !R0 gets 10 x "reason"!
    ------------------            !(Perform other tasks)!
    630 95E2                       POPL RR2,@RR14   !Restore!
    632 95E0                       POPL RR0,@RR14   !  registers!
    634 7B00                       IRET
```

**Figure 10-14. Re-entrant MULTEN Routine, a Calling Program, and an Interrupt-Processing Program**

case) are implicitly saved and restored in switching to and from interrupt processing, and all memory locations at a positive offset from the base defined by RR14 are, in effect, separate copies of that portion of the context.

The example shows how the execution of MULTEN, called from the code starting at 100, is interrupted to allow the interrupt-processing routine IROUT to run. In turn, IROUT calls MULTEN, so MULTEN must work simultaneously in two contexts.

This example follows the changing contents of the registers R0, R1, R2, R3, RR14, PC, and FCW, and shows the section of the stack used during execution of this portion of the program. Figure 10-15 lists the assumed initial values.

| **Registers** | | **Stack** | |
|---|---|---|---|
| **Name** | **Contents** | **Address** | **Contents** |
| R0 | 0000 | (none used yet) | |
| R1 | 1111 | | |
| R2 | 2222 | | |
| R3 | 0003 | | |
| RR14 | (0400,009A) | | |
| PC | (0600,0100) | | |
| FCW | D880 | | |

**Figure 10-15. Initial Values For Registers, PC and FCW For Example**

As the first instruction, at 100 of segment 6, is executed, the stack register value changes to (0400,0098) and stack location 98 contains 0003, the contents of R3. The PC is incremented to (0600,0102), and everything else is unchanged. The next instruction is the call to MULTEN. Figure 10-16 shows the status following that call.

| **Register** | | **Stack** | | |
|---|---|---|---|---|
| **Name** | **Contents** | **Address** | **Contents** | |
| R0 | 0000 | 98 | 0003 | argument |
| R1 | 1111 | 96 | 0108 | saved PC |
| R2 | 2222 | 94 | 0600 | |
| R3 | 0003 | | | |
| RR14 | (0400,0094) | | | |
| PC | (0600,2000) | | | |
| FCW | D880 | | | |

**Figure 10-16. Values After Call To MULTEN From 102**

Figure 10-17 shows the situation after the first two instructions of Multen have been executed. Suppose at this point that an interrupt occurs and that IROUT is the processing routine. Figure 10-18 shows the status immediately following the interrupt. The first two instructions of IROUT push RR0 and RR2 onto the stack. Then the "reason" is fetched and pushed onto the stack as an argument for the call to MULTEN. MULTEN is called, and after the first two instructions of MULTEN have been executed, we are exactly where we were before the interrupt. Figure 10-19 shows the new status.

| **Register** | | **Stack** | |
|---|---|---|---|
| **Name** | **Contents** | **Address** | **Contents** |
| R0 | 0000 | 98 | 0003 |
| R1 | 0003 | 96 | 0108 |
| R2 | 000A (10 = %A) | 94 | 0600 |
| R3 | 0003 | | |
| RR14 | (0400,0094) | | |
| PC | (0600,2006) | | |
| FCW | D880 | | |

**Figure 10-17 Status Before the Interrupt**

| **Registers** | | **Stack** | | |
|---|---|---|---|---|
| **Name** | **Contents** | **Address** | **Contents** | |
| R0 | 0000 | 98 | 0003 | |
| R1 | 0003 | 96 | 0108 | |
| R2 | 000A | 94 | 0600 | |
| R3 | 0003 | 92 | 2006 | saved PC |
| RR14 | (0400,008C) | 90 | 0600 | |
| PC | (0800,0600) | 8E | D880 | saved FCW |
| FCW | D800* | 8C | 0005 | "reason"* |

*To make the example concrete, assume a value of 0005 for "reason" and an FCW value of D800 associated with the interrupt.

**Figure 10-18. Status Immediately Following the Interrupt**

The stack locations 7E, 80, and 82 in Figure 10-19 play the same role as did 94, 96, and 98 in Figure 10-17. If the contents of stack locations 84 thru 98 in Figure 10-19 are covered up, there would be no essential difference between the two figures. The only record of the first execution of MULTEN is stored in these stack locations. Conversely, in Figure 10-17, if the portion of the stack with

| Register | | Stack | | |
|---|---|---|---|---|
| Name | Contents | Address | Contents | |
| R0 | 0000 | 98 | 0003 | Argument & return address for first (interrupted) execution of |
| R1 | 0005 | 96 | 0108 | MULTEN |
| R2 | 000A | 94 | 0600 | |
| R3 | 0003 | 92 | 2006 | CPU status & "reason" pushed automatically when the interrupt |
| RR14 | (0400,007E) | 90 | 0600 | occurred |
| PC | (0600,2006) | 8E | D880 | |
| FCW | D800 | 8C | 0005 | |
| | | 8A | 0003 | RR0,RR2 values saved by IROUT (contain register values set |
| | | 88 | 0000 | during first execution of MULTEN) |
| | | 86 | 0003 | |
| | | 84 | 000A | |
| | | 82 | 0005 | Argument & return address for second execution of MULTEN |
| | | 80 | 0610 | |
| | | 7E | 0800 | |

Execution of MULTEN is at the exact point reached before the interrupt (Figure 10-17). Every value in Figure 10-17 is somewhere on the stack in this figure.

**Figure 10-19 Current and Saved Contexts for MULTEN**

addresses 100 through 114 were shown (nothing tells us where the stack originally started), the context of a previous execution of MULTEN might be found.

Assume that execution proceeds without further interrupts. MULTEN computes 5 x A and stores the result at stack location 82 [at RR14(#4)]. Its RET causes the contents of 7E and 80 to be popped into the PC and execution resumes in IROUT, where the 0032 (5 x A) is popped into R0 and presumably is used in the "perform other tasks" section of IROUT.

When execution in IROUT reaches 630, the RR2 and RR0 values are restored from the stack. At this point, status is exactly as shown in Figure 10-18, except that the PC (and possibly the FCW) has a different value. The execution of the IRET restores the saved values of PC and FCW, leaving the status originally shown in Figure 10-17.

Execution of MULTEN proceeds at 2006 as if there had never been an interrupt. The result of 3 x A (1E) is stored in stack location 98 [R14(#4)]. The RET at 200C causes the saved PC to be restored from stack locations 94 and 96. Execution of the original program then resumes at 108 of segment 6, where the result of the multiplication is popped into R3. The status at this point is shown in Figure 10-20. All of the values here are exactly as they would have been if the execution of MULTEN had not been interrupted.

This example also illustrates how the definition of re-entrancy depends upon the properties of the surrounding system. If RR4 and RR6 instead of RR0 and RR2 had been preserved by interrupt-processing routines, then MULTEN would not be re-entrant and it could not be called from interrupt-processing routines.

| Register | | | Stack | |
|---|---|---|---|---|
| Name | Contents | | Address | Contents |
| R0 | 0000 | | (none still in use) | |
| R1 | 001E | Result of MULT RR0,R2 | | |
| R2 | 000A | Result of LDK R2,#10 | | |
| R3 | 001E | Result of POP R3,@RR14 | | |
| RR14 | (004,009A) | | | |
| PC | (0600,010A) | | | |
| FCW | D800 | FLAGS set by MULT RR0,R2 | | |

**Figure 10-20. Final Values for MULTEN Routine**

The MULTEN example illustrates context switching triggered by interrupts. Another instance of re-entry, for which it is harder to provide a simple illustration, is a program shared by a number of concurrent processes, each doing approximately the same thing. For example, a BASIC or Pascal timesharing system might have one copy of the interpreter that works on the user's programs "concurrently," switching from one to the next either at the expiration of a "time slice" or when the user's program pauses for I/O. Each user would have an interpretable program and a temporary storage stack. These would be in the user's private memory and would be addressed using a base register and an offset (pseudo-PC) register for the interpretable program and a stack register for the stack. These registers and the other general-purpose registers used by the interpreter constitute the context to be switched. The re-entry of the interpreter depends upon its reference to the user's memory areas only through the use of the registers making up the context.

## 10.9 CONTEXT SWITCHING

In Section 10.7, we defined the context of a process to be the values of all registers and memory locations that need to be saved before another process running "at the same time" can have its turn at using them. In general, the context of a process consists of the entire register and memory contents, but in most applications measures are taken to keep the size of the context to a minimum. Fixed storage locations can be avoided, and the times at which context switches occur can be controlled.

Fixed storage locations must become part of the context of a process if some other process can change the contents between the time its value is set and the time it is no longer needed. On the other hand, a process that "ties up the loose ends" before another process can run can have a small context, even though it may use and abandon many registers and locations during the period in which other processes cannot run. The recursive subroutine QUICK presented in Section 10.13.6 is an example of this phenomenon.

In most context-switching schemes, the stack is used for storage of all or part of saved process contexts, as illustrated in Section 10.8. Saving registers on the stack is accomplished efficiently by using the LDM instruction. For example,

```
DEC R15,#16          !Can't decrement by 28!
DEC R15,#12          ! all at once!
LDM @RR14,R0,#14
```

causes registers R0 through R13 to be saved on the RR14 stack.

Saving control registers, if necessary, is accomplished by loading them into registers and then saving the registers. If it is necessary to save the FCW explicitly, care must be taken that the saving operations do not affect the flag bits before they are saved or after they are restored. For example, the DEC instruction affects V, Z, and S, so after the above instructions have been used to save the registers, it is too late to save flags. A variation on the preceding code that saves flags is:

```
PUSHL @RR14,RR12     !Make room to work!
LDCTL R12,FCW        !Get FCW into R12!
SUB R15,#24          !Finish saving registers!
LDM @RR14,R0,#12
PUSH @RR14,R12       !Save FCW!
```

Of the control registers, the Normal Stack Pointer is the one most likely to be part of a process context in a multi-processing system. To save it, the following instructions are added to the above:

```
LDCTL R12,NSPSEG
LDCTL R13,NSPOFF
PUSHL @RR14,RR12
```

If fixed locations are part of the process context, their contents also must be saved. In the code shown in Figure 10-21, assume that RR2 contains the address of a list of fixed word locations whose contents must be saved. Assume that the list is terminated by a double word, -1. This code causes the contents of these locations to be saved on the stack, each accompanied by the corresponding address.

```
LOOP: LDL RR4,@RR2      !Get next item!
      CPL RR4,#-1       !Test for terminator!
        JR EQ,DONE      !Done if -1 encountered!
      LD R0,@RR4        !Get contents!
      PUSH @RR14,R0     !Save both!
      PUSHL @RR14,RR4
      INC R3,#4         !Increment list pointer!
        JR LOOP
DONE:   -----
```

**Figure 10-21. Example, Saving Contents of Fixed Locations Onto Stack**

## 10.10 INTERRUPTS

An interrupt forces a context switch. Since there is almost no control of the time a switch to the interrupt context occurs, interrupt routines must save and restore the values of any registers, control registers, or memory locations they use. (An exception is a memory location purposely changed by the interrupt routine, such as a flag indicating that output of a given line of text is finished.)

Before interrupts can be used, the linkage between the interrupt and the processing routine must be established. This is done using the Program Status Area (PSA) and the Program Status Area Pointer (PSAP). The format of the Program Status Area is described in Chapter 2.7.2. In the PSA, a CPU status (FCW and PC) is specified for every allowed interrupt type. In contrast with machines that used fixed memory locations for such interrupt response definition, the PSA of the Z8000 can be anywhere in program memory so long as it is on a 256-byte block boundary (that is, the last eight bits of its address are zero). This means that the PSA can be assembled with the program without conflicting with the loader's use of the interrupt facility. The only thing remaining to be done at initialization time is to set up the PSAP and then to enable interrupts. If the PSA begins at PSALOC, setting up the PSAP can be done by:

```
LDA RRO,PSALOC
LDCTL PSAPSEG,RO
LDCTL PSAPOFF,R1
```

The use of interrupts for input or output of data requires communication between the program requesting the input or output and the associated interrupt-processing routines. Furthermore, an interrupt-processing routine must communicate with itself; that is, whenever an interrupt occurs, it must know exactly what it is doing and how far along it is.

The solution to providing communication between interrupt and application routines and to providing temporary storage for the interrupt routines is a set of fixed memory locations (often called a process status block or context block) containing pointers, counters, flags, etc.

When the application routine needs to perform an I/O operation, it calls on an initiator routine. For example, it may need to send the ASCII characters for "HELLO" to a CRT screen. The initiator program sets a pointer in the context block to the zero-terminated string of ASCII characters for "HELLO" provided by the application program, and it sets a flag in the context block to "BUSY." Then it does whatever is necessary to assure that output interrupts for the CRT screen begin to occur. As each interrupt occurs, the processing routine transmits another character of the string and advances the pointer in the context block. When the pointer reaches the terminating zero, the interrupt routine sets the flag in the context block to "DONE." Meanwhile, the application program can be doing other things. If it needs to output another string, it waits for the flag to change from "BUSY" to "DONE." It can enter a loop in which all it does is test the flag, or it can do other things while the output proceeds.

This sort of communication between tasks proceeding under interrupt and application programs is sometimes used to implement an event-driven timesharing system. Instead of entering a loop to wait for the flag to change from "BUSY" to "DONE," the program defers to other tasks, allowing them to execute until they too are held up waiting for an I/O operation to be completed.

## 10.11 INITIALIZATION

For the programmer responsible for the entire CPU instead of simply providing programs to run under some system, the sequence of operations following a cold start (reset) is important.

Execution begins when the CPU fetches its CPU status (FCW and PC) from instruction memory addresses beginning at segment 0, offset 2. The FCW is at offset 2, the PC at offset 4. If an external memory-mapping unit is in use, it must be capable of dealing properly with these initial fetches, even before any code is executed to establish memory-mapping parameters.

The PC value at location 4 is the address of the first instruction to be executed. The FCW value should leave the CPU in System mode and segmented operation (unless the CPU is a Z8002) with all maskable interrupts disabled. The nonmaskable interrupt ($\overline{\text{NMI}}$) should be disabled at this point also, but that is impossible, so the system must be designed so that the $\overline{\text{NMI}}$ cannot occur immediately after a reset.

The initialization code first sets the PSAP to point at the previously assembled PSA. The implicit stack register (R15 or RR14) must then be set. If an external memory mapping facility is used, its parameters are set up as soon as possible. Until then, it must continue to handle all instruction, data and stack references properly. Once the stack register and the PSAP are properly initialized, interrupts can be enabled. If the Refresh register is to be used, it is initialized during this sequence.

## 10.12 PROGRAMMING FOR BOTH SEGMENTATION MODES

It is important for Z8000 programmers to know how to write programs for operation in one segmentation mode that can be adapted for use in the other segmentation mode with minimal alterations. The only way two modes differ is in the format of addresses--in instructions, in general-purpose registers, in the PC, in control registers, and on the stack after subroutine calls, traps, or interrupts. Therefore, the solution to this lies in finding mode-independent ways of handling addresses. Addresses are manipulated by programs in many ways. The most common are:

- Loading them into registers
- Performing arithmetic on them
- Using them in the Indirect Register, Base and Base Index addressing modes
- Moving them out of registers and into memory or onto the stack

The two program fragments shown in Figure 10-22 are segmented and nonsegmented versions of the same algorithm. If symbolic definitions are given for the address registers, the code takes the form shown in Figure 10-23.

| **Non-Segmented** | **Segmented** |
|---|---|
| LDA R2,XYZ | LDA RR2,XYZ |
| LD R0,@R2 | LD R0,@RR2 |
| INC R2,#2 | INC R3,#2 |
| LD R1,@R2 | LD R1,@RR2 |
| PUSH @R15,R2 | PUSHL @RR14,RR2 |
| LD R4,R2 | LDL RR4,RR2 |

**Figure 10-22. Example of Segmented vs. Nonsegmented Code**

With the symbolic definitions, the two pieces of code are very similar. The remaining problem is the "L" in the mnemonics. If there were an assembler that recognized the perfectly unambiguous source statements

```
LD RR4,RR2
PUSH @RR14,RR2
```

and generated the long-word versions of the instructions, then at the source code level the segmented and nonsegmented programs would be identical. Without such an assembler, the only other possibility is conditional assembly. Except for very small programs, this is unlikely to be workable, unless the conditional instructions are built into a set of address-manipulation macros.

```
ADREG = R2; ADOFF = R2          ADREG = RR2; ADOFF = R3
SAVREG = R4                     SAVREG = RR4
SR = R15                        SR = RR14

LDA ADREG,XYZ                   LDA ADREG,XYZ
LD R0,@ADREG                    LD R0,@ADREG
INC ADOFF,#2                    INC ADOFF, #2
LD R1,@ADREG                    LD R1,@ADREG
PUSH @SR,ADREG                  PUSHL @SR,ADREG
LD SAVREG,ADREG                 LDL SAVREG,ADREG
```

**Figure 10-23. Assembled Code For Segmented & Nonsegmented Examples**

For example (following no particular macro syntax), an address pushing macro could be defined as follows:

```
        APUSH x,y =
              if y is an RR, then
                        "PUSHL @x,y"
              else
                        "PUSH @x,y"
Then,
                   APUSH SR,ADREG
```

is the next-to-last line of either of the programs in Figure 10-23.

## 10.13 PROGRAMMING EXAMPLES

Sections 10.1 through 10.12 showed how specific features of the Z8000 are related to standard programming techniques. The paragraphs within this section present some complete examples to give a clearer picture of how Z8000 instructions and features are used.

### 10.13.1 Adding An Array Of Numbers

**Problem:** To find the sum of an array of 16-bit signed numbers.

**Solution:** The items are added one at a time to an initially cleared accumulator. Any occurrence of a V indication following any of the additions is registered, and V is set upon completion if overflow occurs at any point during the operation.

Notice that the sum can be correct even if overflow occurs; for example, let the array be (32,765, 8, -25). The first sum, 32,765 + 8, yields -32,763 and an overflow indication. The second sum, (-32,763) + (-25) yields 32,748 and another overflow indication. The final answer is correct:

32,748 = 32,765 + 8 + (-25).

An overflow indication is set upon completion of the addition, and the programmer can choose to take action. Alternatively, the addition program might take action on overflow (such as by terminating the process), but the programmer calling the function has more information about the intended use of the sum and the nature of the data. The code for this appears in Figure 10-24.

```
!Addition subroutine
   CALL SUM with RR2 = array address
                 R0  = number of words in the array (0 to 32,767)
   Returns sum in R1; V is set on return if an arithmetic overflow
   occurred in any of the addition operations used in forming the
   sum.
   The contents of R0, R2, R3 and R4 are lost.
!
SUM:    CLR R1          !Initialize sum to zero!
        CLR R4          !R4 saves any V's!
LOOP:   CP R0,#0        !Done when R0 no longer!
          JR LE,ENDLP   ! greater than zero!
        ADD R1,@RR2     !Add in the next!
        TCC OV,R4       !Save overflow indication!
        INC R3,#2       !Increment array pointer!
        DEC R0          !Decrement loop counter!
        JR LOOP
ENDLP:  RESFLG V
        TEST R4         !V=0 if no overflow!
          RET Z
        SETFLG V        !Otherwise V=1!
        RET
```

**Figure 10-24. Example, Addition Subroutine**

**Notes to Figure 10-24:**

1. Notice that the test for the loop termination condition is done first; this allows the program to behave properly if the initial value of R0 is zero--it returns a sum of zero. Also notice that the test is for LE instead of for EQ. This is simply a precaution. If the count becomes negative, then there is a programming error somewhere, and it is best to stop immediately.

   Given that we wish to test for a counter value less than or equal to zero, we use

   ```
   CP R0,#0
   ```

   instead of

   ```
   TEST R0
   ```

   Because the TEST instruction leaves the V bit unaffected, while the definition of LE is Z OR (S XOR V).

   An alternative to

   ```
   CP R0,#0
   ```

   is the sequence

   ```
   RESFLG V; TEST R0
   ```

   but this sequence does not work with the TESTB instruction, because TESTB uses V to report the parity of the byte (since P = V), and this is unrelated to the sign.

2. Notice the use of the TCC instruction. Initially R4 is cleared; if V is clear (no overflow occurred in the ADD), then

   ```
   TCC OV,R4
   ```

   leaves R4 unaffected. If V is set (overflow did occur), then

   ```
   TCC OV,R4
   ```

   causes the low-order bit to be set. This means that if overflow ever occurs, R4 will be non-zero for the remainder of the time, since the only instruction affecting R4 after it is initially cleared is the TCC, which either sets it or leaves it unaffected.

   It is important to note that the TCC instruction does not set the destination value to zero if the specified condition code is false.

3. Notice the use of

   ```
   INC R3,#2
   ```

   to increment the array pointer RR2. This is done because the segmented address arithmetic is done separately on the segment and offset portions of the address.

   As written, the program wraps around the end of a segment, treating the word at offset zero as the successor to the word at offset 65,534 (FFFE). If this is to be treated as an error, a test can be made for this condition. Z can be used, but this does not necessarily work with larger increments; if long words are being added, for example, INC R3,#4 might change R3 from FFEE to 2. Another approach is to test for this condition on entry, using the initial values of RR2 and R0.

### 10.13.2 Determining The Parity Of A Byte String

**Problem:** To find the parity of an arbitrarily long byte string and to set P (PE true) if the total number of bits in the string of bytes is even, to clear P (PO true) if the total number of bits is odd.

**Solution:** The parity of a byte string is, by definition, the sum of its bits modulo 2. Since addition is associative (i.e., the sum is the same if the items are grouped, subtotals computed for the groups, and the subtotals added), the parity of the byte string is the sum of the parities of its bytes.

Furthermore, if a and b are binary numbers (in particular, if they are bytes), then the parity of (a XOR b) equals the parity of a plus the parity of b. (It suffices to prove this for one-bit arguments a and b, since the parity of an n-bit binary number is the sum of the parities of its n bits. The proof for one-bit arguments is accomplished by considering the four possible bit combinations.) Therefore, the total parity can be determined as follows:

- Initialize a register to zero.
- For each byte of the string, compute the XOR of the byte with the current contents of the register.
- Test the parity of the final contents of the register.

The code for this appears in Figure 10-25.

```
!Subroutine to test the parity of an arbitrarily long
 byte string
        CALL BIGPAR with RR2 = address of the byte string
                          R1 = number of bytes (0 to 32,767)
        Returns with P set (PE true) if parity is even, P
        clear (PO true) if parity is odd.  Contents of R0,
        R1, R2 are lost.
!
BIGPAR: CLRB RL0        !Accumulate parity in RL0!
LOOP:   CP R1,#0        !All tested yet?!
          JR LE,ENDLP   !If so, determine final parity!
        XORB RL0,@RR2   !XOR this byte with RL0!
        INC R3          ! and set up!
        DEC R1          !  for next byte!
        JR LOOP
ENDLP:  TESTB RL0       !Final parity!
        RET
```

**Figure 10-25. Example of the Determination of Parity of a Byte String**

**Notes to Figure 10-25:**

1. Notice that by initially clearing RL0, we assure that a zero-length string has even parity.

2. If we wish to allow for from 0 to 65,535 bytes in the string, we replace

   JR LE,ENDLP

   with

   JR EQ,ENDLP

   In this case we are using the contents of R1 as an unsigned number in the range 0 to 216-1 instead of as a signed number in the range -215 to 215-1.

3. If we wish to allow for from 1 to 65,536 bytes in the string, we remove the instructions

```
CP R1,#0
JR LE,ENDLP
```

and move the label LOOP down to the XORB instruction. The instructions

```
DEC R1
JR LOOP
```

become

```
DJNZ R1,LOOP
```

and the label ENDLP is no longer be needed.

4. For long byte strings, the efficiency of this routine can be increased by using the XOR instruction to process whole words at a time. Special tests have to be included to handle strings that begin at an odd byte or end at an even byte.

### 10.13.3 Accesing An Array Larger Than 65,536 Bytes

**Problem:** To manage a one-dimensional array that is too large to fit within one memory segment and has too many elements to be indexed by a 16-bit word.

**Solution:** Two solutions to this problem are presented. One provides high efficiency but little flexibility, and the other provides great flexibility, but substantial cost in processing overhead.

The high-efficiency scheme uses an arbitrary segmented address as the address of the first array element and assumes that the array is stored contiguously in memory. Segmented address (N+1, 0) is assumed to follow address (N,65,535); that is, consecutively numbered segments are treated as contiguous pieces of the address space. If the segment number bits were bits 6 through 0 of the high-order segmented address byte, this interpretation would be achieved automatically simply by treating segmented addresses as 32-bit unsigned integers. Since this is not the case, the addition of an offset to the starting address of the array must include an operation that takes bits 6-0 of the high-order word of the result and adds them to the segment number field, which is in bits 14-8.

A subroutine is provided to take the base segmented address in one long-word register and an offset in another long-word register. The offset must be less than 223. The algorithm used causes a wraparound from segment 127 to segment 0, so the full 223 bytes of segmented address space are used, regardless of the base segmented address. The code for this version appears in Figure 10-26.

The high-flexibility scheme also uses a 23-bit offset, or virtual address, but instead of a starting segmented address and a contiguous array, it uses a virtual-to-segmented address mapping scheme that works as follows (see Figure 10-27):

- An array of "virtual" addresses in ascending numerical order ($V_1$, $V_2$,...,$V_n$ is provided. A segmented address ($S_0$, $S_1$,...,$S_{n-1}$) is associated with each. Virtual addresses 0 through $V_1$-1 are mapped into a contiguous block of segmented addresses, starting at $S_0$. $V_1$ through $V_2$-1 are mapped into a block at $S_1$, and so on.

- The given virtual address, v, is compared with each of $V_1$, $V_2$,..., $V_n$ until the first $V_i$ is found for which $v < V_i$. If $v \geq V_n$, an error indication is returned.

- The segmented address $S_{i-1} + (v-V_{i-1})$ is returned ($V_0$ is assumed to be zero).

```
!Address-mapping subroutine (high-efficiency version)
   CALL ADMAP with RR2 = virtual address (23 bits)
                   RR4 = starting segmented address
   Returns with RR2 = segmented address corresponding
 to the given virtual address and RR4 preserved.
!
ADMAP:  ADD   R3,R5
        ADCB  RL2,RH4
        EXB   RH2,RL2
        RET
```

**NOTE:** The EXB instruction is unnecessary if the result is returned in RR4. The code for this is:

```
ADMAP:  ADD R5,R3
        ADCB RH4,RL2
        RET
```

The longer version given above allows the array base to be maintained in RR4 at all times.

**Figure 10-26. Example of Accessing Arrays Larger Than 64K Bytes**

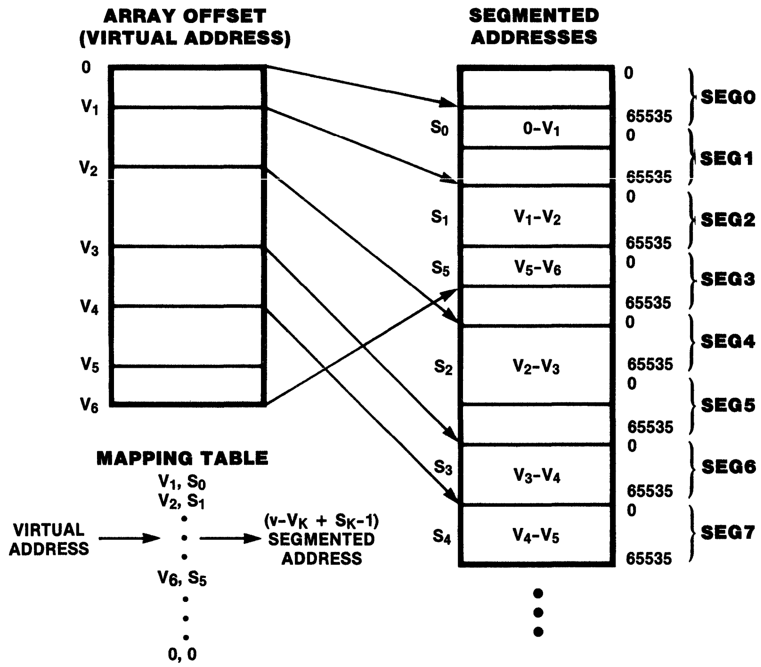


**Figure 10-27. Memory Mapping**

The address calculation $S_{i-1} + (v-V_{i-1})$ is performed as for the high-efficiency scheme above, so that consecutively numbered segments are treated as contiguous, and wraparound occurs from segment 127 to segment 0.

As an example, suppose we have an array of 200,000 bytes that we wish to store in memory in three sections:

```
      0-84,999 starting at segment 6,
               offset 30,000

 85,000-131,071 starting at segment 14,
               offset 0

131,072-199,999 starting at segment 19,
               offset 45,000
```

The subroutine is called with the address of the virtual-to-segmented address mapping table in one double-word register and the virtual address in another. For this example, the mapping table takes the form shown in Figure 10-28. The means of expressing the 32-bit constants and segmented addresses depends upon the specific assembler used. Simpler ways are possible with some assemblers.

In this example, suppose that RR2 contains a virtual address, that is, an index between 0 and 199,999. It can be translated into a segmented address with the following code:

```
LDA RR4,MAPTAB
CALL ADMAP
```

The code for the high-flexibility solution appears in Figure 10-29.

```
MAPTAB:     0;     0    !$V_0$=0!
         %600;30000     !$S_0$=(6,30000)!
            1;19464     !$V_1$=85,000 (= $2^{16}$ + 19464)!
         %E00;     0    !$S_1$=(14,0)!
            2;     0    !$V_2$=131,072!
        %1300;-20536    !$S_2$=(19,45000)!
            3; 3392     !$V_3$=200,000 (= 3 x $2^{16}$ + 3392)!
            0;     0    !Two 32-bit zeros terminate!
            0;     0
```

**Figure 10-28. Example, Memory Mapping Subroutine**

```
!Address-mapping subroutine (high-flexibility version)
   CALL ADMAP with RR2 = virtual address (23 bits)
                   RR4 = address of mapping table
   Returns with C=0 and RR2 = segmented address; or
                C=1 if virtual address out of range
   The contents of RR4 are lost.
!
ADMAP:   INC R5,#8        !Step to next entry!
         TESTL @RR4       !Terminator?!
           JR Z,ERREX     ! Yes-out of range!
         CPL RR2,@RR4     !Compare v with $V_i$!
           JR GE,ADMAP    ! If v $\geq$ $V_i$, try next!
FIND:    DEC R5,#8        !Back up to $V_{i-1}$!
         SUBL RR2,@RR4    !RR2 = v-$V_{i-1}$!
         INC R5,#4        !Step to $S_{i-1}$!
         ADDL RR2,@RR4    !RR2 = $S_{i-1}$ + (v-$V_{i-1}$)!
         ADDB RH2,RL2     !Carry overflow to segment field!
         CLRB RL2         !Clear "reserved" bits!
         RESFLG C; RET    !C=0 for success return!
ERREX:   SETFLG C; RET    !C=1 for out-of-range return!
```

**Figure 10-29. Example, Flexible Memory Mapping Subroutine**

**Notes to Figure 10-29:**

1. The algorithms here are designed for random access. A loop to step through a byte array addressed for the high-efficiency version uses the following sort of address computation:

```
              LDL RR2,RR4     !Start at the beginning!
       LOOP:  !If at end of array, exit!
              !Perform operation on the array element!
              INC R3          !Step to next address!
                JR NZ,LOOP    !Still in the segment!
              INCB RH2        !New segment!
              JR LOOP
```

2. Notice the use of the V0 entry in the MAPTAB table. Even though V0 can only be zero, the program is simplified by including an entry for it in the table.

3. There is no error checking performed in either routine. Several errors can occur: RR2 can contain a virtual address of greater than 23 bits, or MAPTAB can be incorrectly formed or can define an array that overlaps itself.

   The checking of RR2 in either version must be done dynamically. The checking of MAPTAB can be done once when the table is created or each time it is changed. A special routine can be provided for this purpose.

4. The DEC R5,#8 and INC R5,#4 instructions in the mapping computation are required because the Based Addressing mode cannot be used with the ADD and SUB instructions. If it could, the code at FIND might be

```
       FIND:  SUBL RR2,RR4(#-8)
              ADDL RR2,RR4(#-4)
```

   In the nonsegmented mode, indexed addressing can be used to simulate based addressing (see Section 10.2 Addressing Modes), but of course, this program would not be used in nonsegmented operations.

5. Many applications using large arrays do not need to have the entire array in memory at all times. The high-flexibility version of address mapping can be used to implement a demand-loading scheme. For this, the code at "FIND:" must recognize a special value for the base segmented address Si-1 that signifies that the array section in question is not currently present in main memory. (Si-1=231-1 is a good value for this purpose.) At this point a call can be made on a demand-loading routine that loads the section in question and passes back its actual segmented address for storage in the address-mapping table.

### 10.13.4 Removing Trailing Blanks

**Problem:** To replace a fixed-length array of text (such as a card image) by a possibly shorter array containing the initial segment of the original array up to and including the last non-blank character. This type of operation is useful when a set of fixed-length arrays (for example, a card deck) is to be read into memory. Elimination of trailing blanks allows more records to fit into a buffer of given size.

**Solution:** The Z8000 block instructions that use the autodecrement mode are designed to handle this sort of problem. The array is scanned backward until the first non-blank character is found. The code for this appears in Figure 10-30.

```
!Subroutine to remove trailing blanks
   CALL STRIP with RR2 = address of the array
                    R1 = # of bytes in the array (1 to 65,536)
   Returns with R0 = number of bytes in stripped array.
   The contents of R0, R1 and RR2 are lost.
!
BLANK=32         !ASCII Code for blank!

STRIP:           LDB RL0,#BLANK          !Comparison character!
                 ADD R3,R1               !Set RR2 to point!
                 DEC R3                  ! at end of array!
                 CPDRB RL0,@RR2,R1,NE    !Scan backward to non-blank!
                 LD R0,R1                !Remaining count (Z not affected)!
                   RET NZ                !If all-blank return R0=0!
                 INC R0                  !Count the final non-blank!
                 RET
```

**Figure 10-30. Example, Removing Trailing Blanks**

**Notes to Figure 10-30:**

1. Notice the computation to set RR2 to point at the last byte of the array. R3 is the offset portion of the address in RR2. Adding R1 (the number of bytes in the array) to R3 leaves RR2 pointing at the first byte following the array. DEC R3 brings RR2 back to its array. (R1 = 0 means 65,536 bytes.)

2. The Block Compare instruction terminates when the count in R1 reaches zero or when one of the CPB RL0,@RR2 operations causes the NE condition to be true. R1 is decremented for each comparison, whether or not there is a match. Therefore, if a match occurs (which the block compare instruction signals by setting Z), the count remaining in R1 is one less than the number of bytes in the stripped array. If no match occurs, R1 is decremented to zero, which is equal to the number of bytes in the stripped array.

### 10.13.5 Determining Whether A 16-Bit Word Is A Bit Palindrome

**Problem:** To determine whether or not a given 16-bit word satisfies the condition

Bit n = Bit (15-n)

for n = 0, 1, 2, ..., 15. A word meeting this condition is called a bit palindrome, since it reads the same frontwards and backwards.

**Solution:** This problem illustrates the use of the Z8000 bit-testing instructions that allow the bit number to be specified in a register. The solution given here is the straightforward one: comparing bit n with bit (15-n) for n = 0, 1, 2, ..., 7. The code appears in Figure 10-31.

```
!Subroutine to test for bit palindromes
 CALL BITPAL with R0 = 16-bit word to be tested.
 Returns with C=1 if not a bit palindrome, C=0 if it is.
 Register use: R1 = n; R2 = 15-n; RH3 = loop count; RL3 = scratch.
!
BITPAL: CLR R1          !Set n = 0!
        LDK R2,#15      ! and 15-n = 15!
        LDB RH3,#8      !Set loop counter!
LOOP:   CLRB RL3
        BIT R0,R1       !Test Bit n and!
        TCCB NZ,RL3     ! move it into RL3!
        RLB RL3         !Make room for Bit 15-n!
        BIT R0,R2       !Test Bit 15-n and!
        TCCB NZ,RL3     ! move it into RL3!
        TESTB RL3       !Bit n = Bit 15-n if and!
          JR PO,NOTPAL  ! only if parity of RL3 is even!
        INC R1          !Increment n!
        DEC R2          !Decrement 15-n!
        DBJNZ RH3,LOOP  !Loop until count exhausted!
        RESFLG C; RET   !Success: set C=0 and return!
NOTPAL: SETFLG C; RET   !Failure: set C=1 and return!
```

**Figure 10-31 Example, Test for Bit Palindromes Subroutine**

**Notes to Figure 10-31:**

1. This example illustrates the operation of the Bit Test instruction. A more efficient solution to the problem involves a direct comparison of the two bytes of R0 after reversing one of them with a loop like:

```
        LDK R2,#8
LOOP:   RLCB RL0
        RRCB RL1
        DJNZ R2,LOOP
        RLCB RL0
```

2. The condition code NZ is used in the TCC instructions. BIT sets Z if the bit is clear and clears Z if the bit is set.

3. TCC instructions are used to save the bit values, and TEST is used to compare them by testing the parity of the byte into which they have been stored. Both simplify the flow of control. Not using these techniques results in the sort of jumping around shown in Figure 10-32.

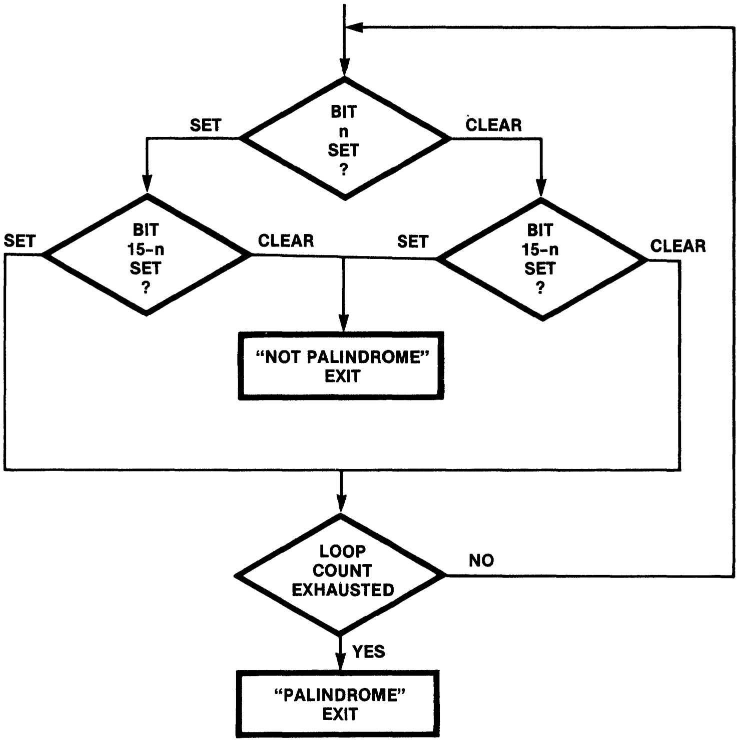


**Figure 10-32. A Poor Alternative to the Use of TCC**

### 10.13.6 Sorting

**Problem:** Given an array A of "items" and an order relation "$\le$", rearrange the items of A in such a way that for integers i and j and items ai and aj, ai $\le$ aj whenever i $\le$ j. The items of A can be integers, floating point numbers, character strings, or any other data type. The order relation can be any ordering appropriate to the given data type, for example, dictionary order for character strings.

**Solution:** An adaptation of the "quicksort" algorithm of C.A.R. Hoare is used. A program is written to sort an array of 16-bit 2's complement integers in ascending numerical order. The organization of the program into subroutines indicates how other items and orderings can be used.

Assume that A is an array indexed from 0 to N. Quicksort is a recursive procedure that begins by arbitrarily selecting one of the items of A as the "pivot" value. Then a preliminary rearrangement of A is made as follows: For some i, $0 \le i \le N$, ai is the pivot value and $a_k \le a_i$ if $0 \le k \le i$, $a_k \ge a_i$ if $i < k \le N$. That is, all items less than, or equal to, the pivot are moved into the "left half" of the array and all those greater than, or equal to, the pivot are moved into the "right half."

Once this is done, the same process is performed on each of the two array segments $a_0$ to $a_{i-1}$ and $a_{i+1}$ to $a_N$. These segments are usually not of equal size. Implementation of the algorithm requires a minimum of stack storage if at each stage the smaller segment is sorted first.

In this example assume that array offsets are 23-bit numbers in the range of 0 to 8,388,607 and that the array elements are 16-bit signed integers. A base segmented address and an address computation similar to that of the high-efficiency version of ADMAP (Section 10.23.3) are used. The generalization to other types of element is straightforward. The code for this appears in Figures 10-33 through 10-38.

```
!Subroutine Quicksort
 CALL QUICK with RR12 = array address
                 RR10 = U (offset of upper limit)
                 RR8  = L (offset of lower limit)
Returns with array elements at offsets between L and U (inclusive)
sorted.  L and U are 23-bit integers in the range 0 to 8,388,607.
 Register use:
        RR14: Stack Register
        RR12: Always contains starting segmented address of array
         RQ8: (L,U) on call; shorter (L,U) range returned by SHORT
         RQ4: longer (L,U) range returned by SHORT
         RQ0: used by subroutines of QUICK
!
 QUICK: CPL RR8,RR10     !Compare L,U!
          RET GE         !Return if L ≥ U!
        CALR PART        !Partition: RQ4, RQ8 get ranges!
        CALR SHORT       !Put shorter range in RQ8, longer in RQ4!
        DEC R15,#8       !Save RQ4 - longer (L,U) range!
        LDM @RR14,R4,#4
        CALR QUICK       !Recursive call to sort the shorter range!
        LDM R8,@RR14,#4  !Restore longer range - into RQ8!
        INC R15,#8
        CALR QUICK       !Recursive call to sort the longer range!
        RET
```

**Figure 10-33. Example, Sort Subroutine Quicksort Initialization**

```
!Subroutine of QUICK to put shorter range first
 CALL SHORT with RQ4 = one (L,U) range
                 RQ8 = another (L,U) range
 Returns with shorter range in RQ8, longer in RQ4
 Register use: as for QUICK.  RR0 contents are lost.
!
SHORT:  LDL RR0,RR6      !RR0 = U-L for RQ4!
        SUBL RR0,RR4
        PUSHL @RR14,RR0  !Save first U-L!
        LDL RR0,RR10     !RR0 = U-L for RQ8!
        SUBL RR0,RR8
        CPL RR0,@RR14    !Compare lengths!
        POPL RR0,@RR14   !Clear the stack!
          RET LE         !Return if RQ8 length ≤ RQ4 length!
        EX R4,R8         !Exchange RQ4 & RQ8!
        EX R5,R9
        EX R6,R10
        EX R7,R11
        RET
```

**Figure 10-34. Quicksort Subroutine to Position Shorter Range First**

```
!Partitioning subroutine of QUICK
 CALL PART with registers as for QUICK
 Returns with array segment between L and U partitioned
 around a pivot element with index I.  Returns the two
 ranges to be sorted: (L,I-1) in RQ8 & (I+1,U) in RQ4.
 Register use: RQ8 = (L,U); RQ4 = (I,J).  On return,
 RQ4,RQ8 are new ranges.  RQ0 is used by subroutines.
!
PART:   CALR SETPIV     !Choose pivot; initialize pivot routines!
        LDL RR4,RR8     !Set I = L!
        LDL RR6,RR10    !Set J = U!
        CALR DECI       !Decrement I: I=L-1!
LPI:    CALR UPI        !Increment I until a_I ≥ pivot value!
        CALR DOWNJ      !Decrement J until a_J ≤ pivot or J ≤ I!
          JR C,MOVPIV   !J ≤ I: only pivot remains to be moved!
        CALR EXCHIJ     !Exchange a_I and a_J values!
        JR LPI
MOVPIV: CALR EXCHIP     !Exchange a_I and pivot values!
        LDL RR6,RR10    !Move I to end of RQ4 (where J was)!
        LDL RR10,RR4    !Move I to end of RQ8 (where U was)!
        CALR DECI       !Decrement I: RR4 = I-1!
        EX R4,R10       !Exchange RR4,RR10:        !
        EX R5,R11       !Now RQ8 = (L,I-1); RR4 = I!
        CALR INCI       !Increment I: Now RQ4 = (I+1,U)!
        RET
```

**Figure 10-35. Quicksort Partitioning Subroutine**

```
!Subroutines of PART for moving I and J
 CALL UPI: returns with I incremented until a_I ≥ pivot value
 CALL DOWNJ: returns with J decremented until a_J ≤ pivot
 or J ≤ I; returns C=1 if J ≤ I, otherwise C=0
 Register use: As for PART.
!
UPI:    CALR INCI       !Increment I!
        CALR CPPI       !Compare pivot value with a_I!
           RET LE       !Return if pivot value ≤ a_I!
        JR UPI          !Otherwise keep incrementing!

DOWNJ:  CALR DECJ       !Decrement J!
        CPL RR4,RR6     !Compare I,J!
          JR LT,DJ1     !I < J: proceed!
        SETFLG C; RET   !J ≤ I: return C=1!
DJ1:    CALR CPPJ       !Compare pivot with a_J!
          JR LT,DOWNJ   !Keep decrementing if pivot value ≤ a_J!
        RESFLG C; RET   !Otherwise return with C=0!

!Routines to increment or decrement I or J.
ESIZE = 2               !Entries are words: two bytes!
INCI:   ADDL RR4,#ESIZE
        RET
DECJ:   SUBL RR6,#ESIZE
        RET
DECI:   SUBL RR4,#ESIZE
        RET
```

**Figure 10-36. Quicksort Subroutine For Moving I and J**

```
!Pivot Setting and Comparison Subroutines
   CALL SETPIV - chooses pivot & saves its value in a
 register
 CALL CPPI - compare pivot value, $a_I$. Set FLAGS.
 CALL CPPJ - compare pivot value, $a_J$. Set FLAGS.
 Register use: as for PART. R0 = temp. R1 = saved pivot
 value.  RR2 = calling argument and actual address
 returned by ADCOMP
!
SETPIV: LDL RR2,RR10     !RR2 = U!
        CALR ADCOMP      !RR2 = actual address of $a_U$!
        LD R1,@RR2       !Choose $a_U$ as pivot value!
        RET

CPPI:   LDL RR2,RR4      !RR2 = I!
        JR IJM
CPPJ:   LDL RR2,RR6      !RR2 = J!
IJM:    CALR ADCOMP      !RR2 = adr of item to be compared!
        CP R1,@RR2
        RET
```

**Figure 10-37. Quicksort Subroutines for Pivot Setting and Comparison**

```
!Exchange Subroutines
 CALL EXCHI - exchange $a_I$ and pivot values.
 CALL EXCHIJ - exchange $a_I$ and $a_J$ values.
 Register use: as for PART. R0 = temp. R1 = saved pivot
 value.  RR2 = calling argument and actual address
 returned by ADCOMP
!
EXCHIJ: LDL RR2,RR4      !RR2 = I!
        CALR ADCOMP      !RR2 = actual address of $a_I$!
        LD R0,@RR2       !R0 = $a_I$!
        PUSHL @RR14,RR2  !Save address of $a_I$!
        LDL RR2,RR6      !RR2 = J!
        CALR ADCOMP      !RR2 = actual addresss of $a_J$!
        EX R0,@RR2       !Exchange: R0=$a_J$, $a_J$ replaced by $a_I$!
        POPL RR2,@RR14   !Restore $a_I$ address!
        LD @RR2,R0       !Replace $a_I$ by $a_J$!
        RET

EXCHIP: LDL RR2,RR4      !RR2 = I!
        CALR ADCOMP      !RR2 = actual address of $a_I$!
        EX R1,@RR2       !Exchange $a_I$ with saved pivot value!
        LDL RR2,RR10     !RR2 = U (offset of pivot element)!
        CALR ADCOMP      !RR2 = actual address of $a_U$!
        LD @RR2,R1       !Replace $a_U$ by $a_I$!
        RET

ADCOMP: ADDL RR2,RR12    !Add array base to offset!
        ADDB RH2,RL2     !Carry overflow into segment field!
        CLRB RL2         !Clear reserved bits!
        RET
```

**Figure 10-38. Quicksort Exchange Subroutines**

**Notes to Figure 10-38**

1. This code falls into two principal categories: the code to implement the algorithms and the code to manipulate the indices and data items. The algorithm is implemented by the routines QUICK, PART, SHORT, UPI, DOWNJ and SETPIV. The manipulation and comparison of data items and the arithmetic on array indices occur in the routines INCI, DECI, DECJ, CPPI, CPPJ, EXCHIP, EXCHIJ, and SETPIV. The mapping of array offsets into actual memory addresses occurs in ADCOMP.

   The organization used here facilitates the alteration of QUICK for other applications. For example, a nonsegmented version can be produced simply by changing all instances of @RR2 to @R3 and keeping the nonsegmented array address in R13 with a zero in R12. All references to RR14 also have to be changed to refer to R15. The resulting code is less efficient than a tailor-made nonsegmented version, but this does not matter in many applications.

   As another example, QUICK could be changed so that it sorts bytes by redefining the symbol ESIZE to take the value 1. Instead of using R0 as a temporary location and R1 for the saved pivot value, the routines SETPIV, CPPI, CPPJ, EXCHIP, and EXCHIJ need byte registers. Then the four LD instructions, the CP instruction, and the two EX instructions in those routines must be changed to byte versions.

   Sorting on the basis of other ordering relations is facilitated by this program orgnization. For example, decreasing numerical order could be used simply by replacing the instruction CP R1,@RR2 with:

   ```
   LD R0,@RR2
   CP R0,R1
   ```

   in the CPPI/CPPJ routine (CP @RR2,R1 is not a legal instruction). The program could have byte constants representing the various flags combinations it wishes to return. For example, the less than condition can be returned by the following sequence of instructions at the end of the subroutine:

   ```
   LDB RH0,#LTVAL
   LDCTLB FLAGS,RH0
   RET
   ```

   The symbol LTVAL might have the value %20, corresponding to C = 0, Z = 0, S = 1, V = 0, D = 0, and H = 0.

2. The CPPI and CPPJ routines illustrate the useful programming technique of multiple entry points. An alternative organization is

   ```
   CPPI: LDL RR2,RR4
         CALR IJM
         RET
   CPPJ: LDL RR2,RR6
         CALR IJM
         RET
   ```

   The code at IJM in both organizations is shared. The objective of this is not principally to save memory space but rather to assure that these two related functions are carried out according to a common algorithm.

3. The SETPIV routine is mainly concerned with data manipulation, but it also implicitly embodies a part of the quicksort algorithm, the choice of a pivot element. Use of aU for the pivot is inefficient if the array is already sorted. Other algorithms can be used to make the choice.

4. The use of 23-bit indices stored in long-word registers simplifies index comparisons such as those that occur in QUICK and SHORT. To use the same code for one-word registers, the index values would have to be restricted to 15 bits. If 16-bit indices are used, the comparisons must be the unsigned versions. In that case, special tests must be made for the case L > U, in both SHORT and QUICK. In particular, the case U = -1, L = 0, a termination condition for QUICK, needs further special handling.

### 10.13.7 Polynomial Evaluation

**Problem:** Given a set of coefficients $a_0$, $a_1,\ldots,a_n$ and a variable x, compute

$$f(x) = a_0 + a_1x + a_2x^2 + \ldots a_nx^n.$$

**Solution:** The coefficients $a_0,\ldots,a_n$, the variable x, all of the products $a_kx^k$, the intermediate sums, and the final sum are assumed to be within the range of 32-bit signed integers, $-2^{31}$ to $2^{31}-1$.

A subroutine (code shown in Figure 10-39) is provided that accepts as its arguments the variable x and the address of a parameter table describing the array. The table has the following format:

n (1 word)
$a_0$ (2 words)
.
.
.
$a_n$ (2 words)

```
!Subroutine to perform polynomial evaluations!
   CALL POLY with RR0 = x
                  RR2 = adr. of table (n, a0, ..., an)
   Returns with   RR4 = f(x), contents of RR2 and R6-R13 lost
                  V = 0 if all values in bounds, 1 otherwise.
   Register use:  RR0, RR2 -- calling arguments
     RR4 -- running sum        R12 -- coefficient counter
     RR6 -- x^k (k=0,1,...,n)  R13 -- error flag
     RQ8 -- scratch
!
POLY:    POP R12,@RR2      !Get n from table to set counter!
         LDL RR6,#1        !Initialize: x^k = 1 (i.e., k = 0) !
         LDL RR4,#0        !            f(x) = 0             !
         CLR R13           !            Error flag = 0       !
LOOP:    POPL RR10,@RR2    !Get a^k from the table!
         CALR MULCH        !RR10 = a_kx^k!
         ADDL RR4,RR10     !f(x) = f(x) + a_kx^k!
         TCC OV,R13        !Remember overflow, if any!
         DEC R12           !Decrement coefficient counter!
           JR MI,POLEX     !  Done if < 0!
         LDL RR10,RR0      !Get x!
         CALR MULCH        !RR10 = x^(k+1)!
         LDL RR6,RR10      !Replace x^k by x^(k+1) (i.e., increment k)!
         JR LOOP           !Perform computation for new k!
POLEX:   RESFLG V
         TEST R13          !Were there any overflows?!
           RET Z           !  No -- return with V = 0!
         SETFLG V; RET     !  Yes -- return with V = 1!
```

**Figure 10-39. Example, Subroutine To Perform Polynominal Evaluations**

The subroutine returns the value f(x). In addition, the results of computations are checked at each stage to verify that they remain within the stated bounds. If the bounds are exceeded at any stage, V is set when the subroutine returns its final result.

The code is arranged so that multiplications are required at two places. In each case, the arguments are manipulated in the registers so that the required instruction is

```
MULTL RQ8,RR6
```

A subroutine is provided to execute this instruction and to verify that the result fits into RR10, the low-order half of RQ8. If not, a bit is set in an error-flag register that is initially cleared to zero by the main routine. The code for the multiply and check routine is shown in Figure 10-40.

```
!Multiply and check subroutine!
MULCH: MULTL RQ8,RR6    !Perform the multiplication!
       PUSHL @RR14,RR8  !Save high-order 32 bits!
       EXTSL RQ8        !Set high-order 32 bits to proper value!
       CPL RR8,@RR14    !Was it already OK?!
       TCC NE,R13       !If not, then overflow occurred!
       INC R15,#4       !Discard saved RR8!
       RET
```

**Figure 10-40. Example, Multiply and Check Subroutine**

**Notes to Figure 40:**

1. Notice the structure of the loop in POLY. There is no test at the beginning, so the loop is always executed at least once. The effect of this is that tables with negative values of n will be treated as if they had n = 0.

   There is also no test at the end of the loop. Instead, the decrement of the coefficient counter and the test for termination appear immediately following the latest update of the running sum and before the computation of $x^{k+1}$. The overall length of the program can be shortened by moving this test to the end of the loop, but then $x^{n+1}$ is always computed unnecessarily. In addition to the wasted computation, this leads to an erroneous overflow indication if $x^{n+1}$ exceeds the 32-bit limitation.

2. The subroutine MULCH illustrates the use of the multiplication and sign extension instructions. The instruction

   ```
   MULTL RQ8,RR6
   ```

   causes the contents RR10 (the low-order half of RQ8) to be multiplied by the contents of RR6 and the resulting value to be stored in RQ8. The original contents of RQ8 (the high-order half of RQ8) are irrelevant. The instruction

   ```
   EXTSL RQ8
   ```

   causes the contents of RQ8 to be replaced by a number whose value is the same as that of RR10 but which has twice as many bits. Assuming that all results are within the range of signed 32-bit numbers, the EXTSL instruction should cause no change to RR8. This explains the test performed in MULCH.

3. The use of the TCC instruction to remember the occurrence of overflows is similar to its use in Section 2.1.

### 10.13.8 PSEUDO-RANDOM NUMBER GENERATION

**Problem:** To provide a subroutine that returns an "unpredictable" 16-bit number.

**Solution:** The solution presented is sometimes referred to as the power residue method. A large positive number N with few prime factors is chosen. The values returned by the function RND on successive calls 1,2,... are defined as follows:

$$RND_1 = N^2 \qquad \pmod{2^{16}}$$

$$RND_k = (RND_{k-1} \text{ AND } (2^{15}-1)) \text{ X } N \pmod{2^{16}}$$

for k = 2,3,...

The algorithm used requires that the routine know at each stage the value it returned when last called. The storage space for remembering this value is provided by the caller in a table whose address is passed to the routine each time it is called. An initializing routine is provided for setting up the table. Figure 10-41 shows the code for the initializing routine and the pseudo-random number generator.

```
!Random-number routines
     CALL INRAND with RR2 = address of 2-word temp storage table.
     Returns with table "initialized," R1 lost, and RO = N.

     CALL RAND with RR2 = address of the table.
     Returns with RO = "random" number & table updated.

     Register use:
       RRO: Dest for multiplication; RO returns the random number.
       RR2: address of table.
!
N = 15419               !N = 17*907!

RAND:    LD R1,RR2(#2)    !R1 = RNDk-1!
         RES R1,#15       !R1 = RNDk-1  AND 2^15-1!
         MULT RRO, @RR2   !RRO = (RNDk-1 AND (2^15-1))*N!
         LD RO,R1         !RO = RNDk!
         LD RR2(#2),RO    !Remember RNDk for next call!
         RET

INRAND: LD RO,#N
         LD @RR2,RO       !Save N in table!
         LD RR2(#2),RO    !RND0 = N!
         RET
```

**Figure 10-41. Example, Random Number Generator**

**Notes to Figure 10-41:**

1. This is a quick and dirty pseudo-random number generator. For a thorough discussion of random-number theory and algorithms, refer to Chapter 3 of "The Art of Complete Programming, Volume 2: Seminumerical Algorithms," by Donald E. Knuth.

2. Similar routines can be used for 32-bit random numbers. In fact, RAND could be generalized to take its argument size from the table. The desired size could be passed to INRAND, which would set up the table accordingly.

3. The choice of the number N could be made by the caller and passed, possibly as an option, to INRAND.

4. Note the use of the instruction

   ```
   RES R1,#15
   ```

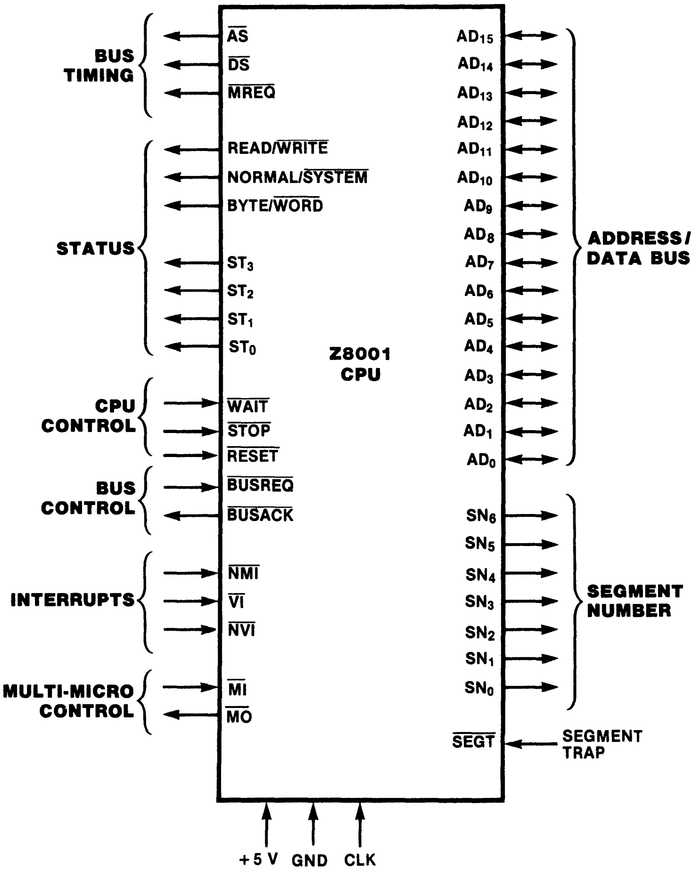
   as an alternative to

   ```
   AND R1,#%7FFF.
   ```

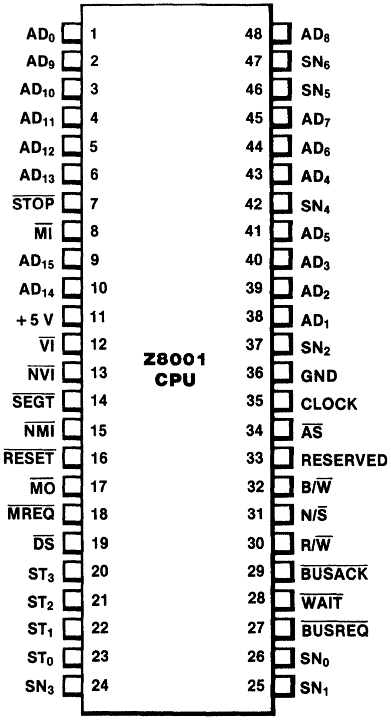
5. Note that the use of an argument table makes RAND a re-entrant routine.

A

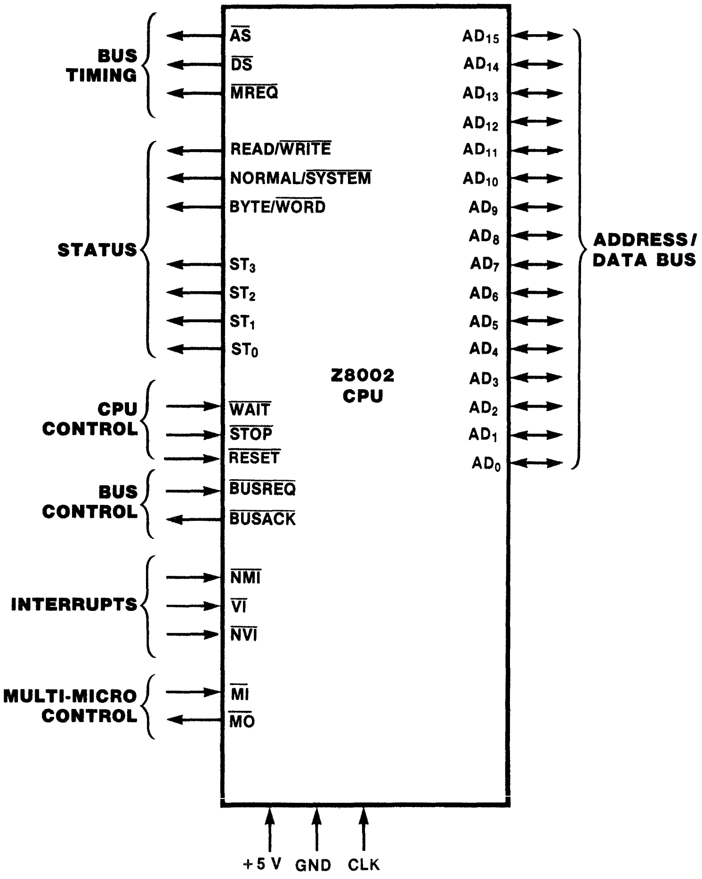
Zilog



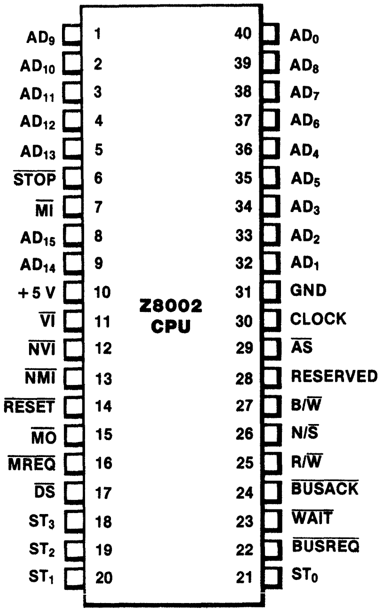
Z8001 CPU Pin Functions



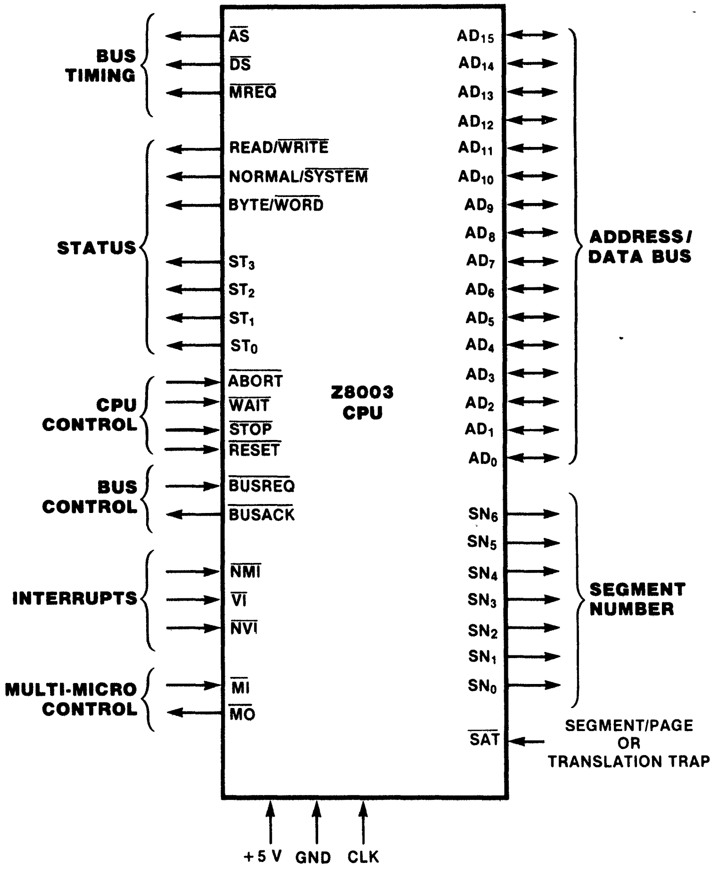
Z8001 Pin Assignments



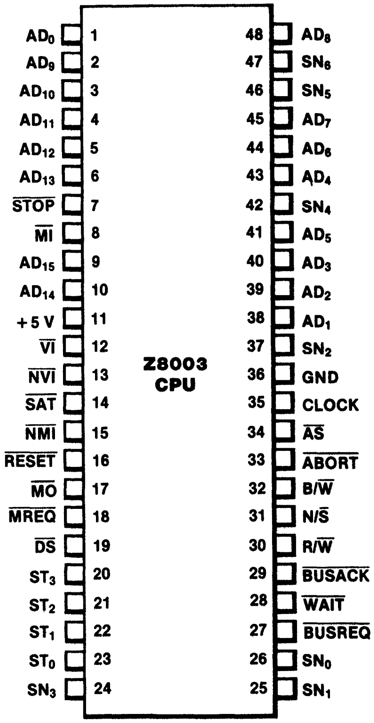
Z8002 CPU Pin Functions



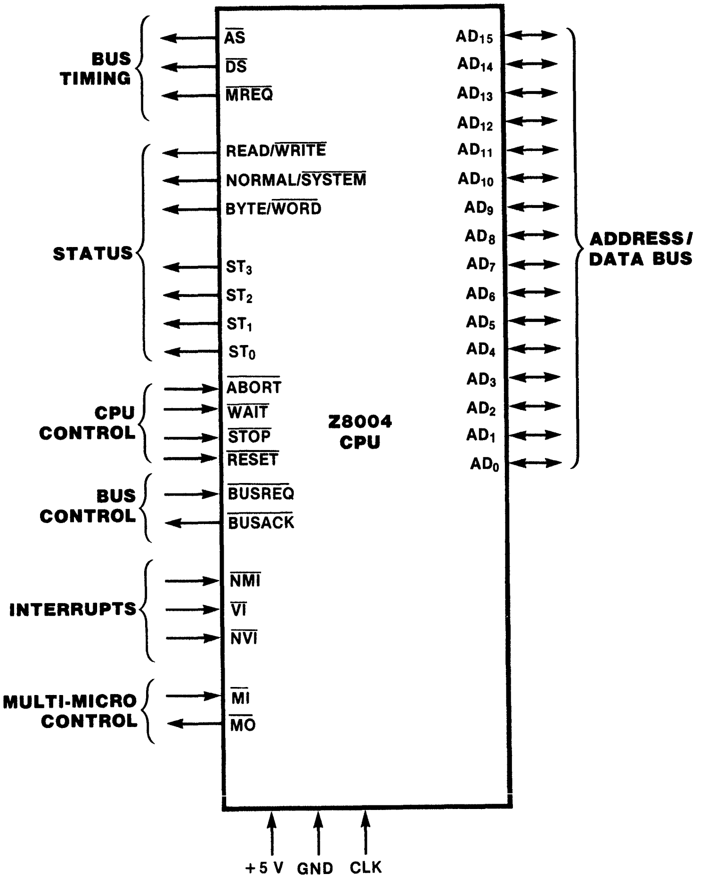
Z8002 Pin Assignments



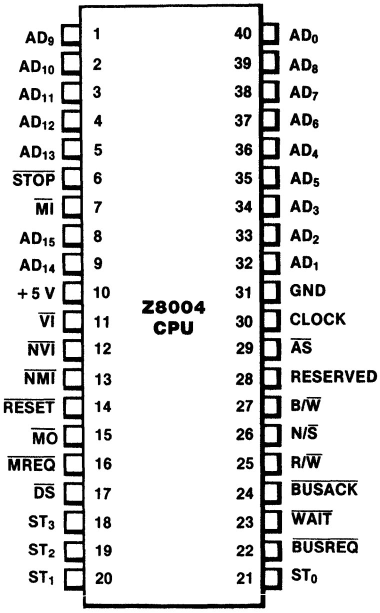
Z8003 Pin Functions



Z8003 Pin Assignments



Z8004 Pin Functions



Z8004 Pin Assignments

B

Zilog

# Z8003/4 Z8000™ VMPU Virtual Memory Processing Unit

Zilog

# Product Brief

June 1982

## Features

- Binary, function, and pin compatibility with the Z8001/2 microprocessors.
- Designed-in compatibility with present and future Zilog Memory Management Units (MMUs).
- Operates with up to a 10 MHz clock.
- Status lines indicate the read/write phase of the Test and Set instruction for use in multiprocessor systems.
- 23-bit segmented addresses for Z8003.
- 16-bit non-segmented addresses for Z8004.

## Description

The Z8003/4 Virtual Memory Processor Unit (VMPU), a 16-bit MOS microprocessor, offers integral provisions for operation in a virtual memory environment, in addition to the features of the Z8001 CPU. The Z8003 VMPU generates 23-bit addresses. The address space is organized into 128 segments, each up to 64K bytes in length. The Z8004 generates 16-bit addresses. The Z8003/4 VMPU addressing scheme distinguishes between memory space for program, data, and stack in each of two modes, System and Normal.

For use in shared-memory multiprocessor systems, the Z8003/4 VMPU provides an output on the status lines ($ST_0$-$ST_3$), indicating the read/write phase of the Test and Set (TSET) instruction. This status output can be used externally for arbitration of bus control.

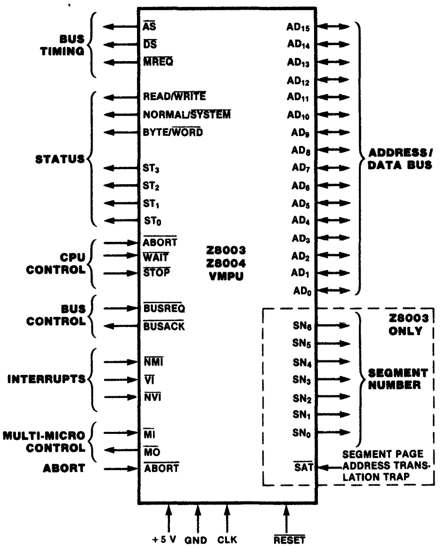


**Figure 1. Pin Assignments**

In a virtual memory environment, the programs and data being operated on need not reside simultaneously in main memory. Thus, provision must be made for retrieving parts of a program or data located in "secondary" storage (such as a disk). Attempts by the microprocessor to access instructions or data not in main memory are called "accesses to nonresident data." When this is done, the transaction accessing the nonresident data must be interrupted, the state of the microprocessor saved, the program or data in secondary storage moved to main memory, the state of the microprocessor restored, and the interrupted instruction restarted.

The Z8003/4 VMPU provides an external abort pin to permit the interruption of instruc-

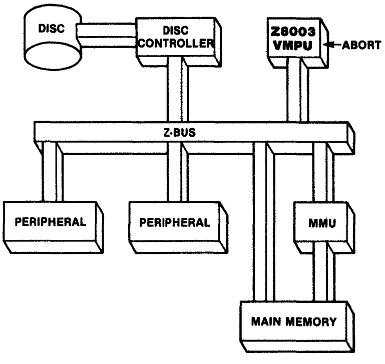


**Figure 2. Virtual Memory Environment**

## Description (Continued)

tion execution before the instruction completes.

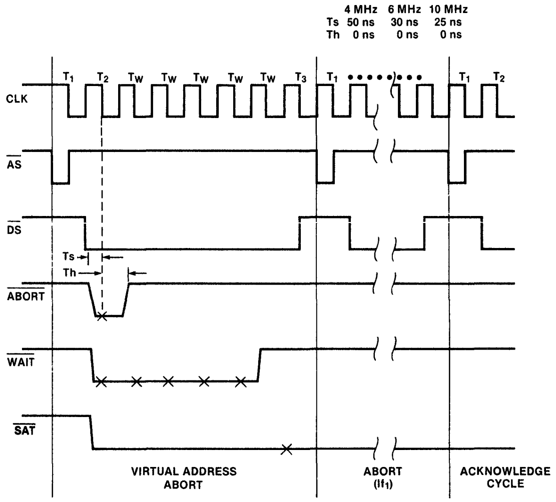
When the Z8003/4 VMPU is used in a multiprocessor system, there may be dual-ported memories used by the processors. In this type of system, a resources manager arbitrates simultaneously attempted accesses to shared resources. When a processor tests to see if a resource is in use, the read/write portion of the test transaction must not be interrupted or the probability of a collision increases greatly. The Z8003/4 VMPU provides features that help to avoid collisions during accesses to shared resources via the enhanced TSET instruction.

## Functional Description

The Z8003/4 VMPU can operate in a virtual memory environment. The virtual memory capability is provided by an instruction abort function on pin 33 of the Z8003 and on pin 28 of the Z8004. When this pin WAIT, and SAT are activated at the same time, an instruction abort sequence begins. This abort sequence leaves the VMPU in a well-defined state, allowing a software recovery. To make this recovery smoothly, the software must know which instruction was aborted and how much of the instruction was executed. Figure 3 shows the timing sequence for the abort function. Figure 4 shows the sequence of hardware and software events that occurs when an instruction is aborted.

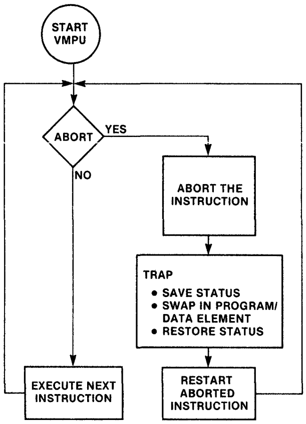
During the read phase of the TSET instruction on the Z8003/4 VMPU, the status lines $ST_0$-$ST_3$ are all set to 1s. On the Z8001/2 all 1s on the status lines is a reserved status encoding.

The Z8003/4 VMPU is compatible with the Z8000 Family of microprocessor and peripheral devices. Instruction set and bus transaction protocols of the VMPU can be found in the *Z8000 CPU Technical Manual* (document number 00-2010-C). The VMPU enhancements are described in the *VMPU Product Specification.*



NOTES: * = Clock Sample Points

**Figure 3. Instruction Abort Timing**



NOTE: The abort sequence is initiated when ABORT, SAT, and WAIT are activated.

**Figure 4. Instruction Abort Function Flow**

## Summary

The Zilog VMPU is the first 16-bit microprocessor that offers integral provision for operation in a virtual memory environment. The upward compatibility of the VMPU with the Z8001/2 CPU means that applications software developed for a Z8001/2 CPU will execute directly on the VMPU, preserving investments in software and development tools.

# Z8010
# Z8000™ Z-MMU Memory Management Unit

Zilog

## Product Brief

March 1982

### Features

- Dynamic segment relocation makes software addresses independent of physical memory addresses.
- Access validation to protect memory areas from unauthorized or unintentional access.
  - □ Overflow warning and expansion provision for stack segments.
- 64 variable-sized segments from 256 to 65,536 bytes can be mapped into a total physical address space of 16M bytes; all 64 segments are randomly accessible.
- Can be used with either the Z8001 or Z8003 CPU.
- Multiple MMUs can support several translation tables for each Z8001/3 address space.
- MMU architecture supports multi-programming systems and virtual memory implementations.

### Description

The Z8010 Memory Management Unit (MMU) manages the large 8M byte address spaces of the Z8001 or Z8003 CPU. The MMU provides dynamic segment relocation as well as numerous memory protection features.

Dynamic segment relocation makes user software addresses independent of the physical memory addresses, thereby freeing the user from specifying where information is actually located in the physical memory. It also provides a flexible, efficient method for supporting multi-programming systems. The MMU uses a translation table to transform the 23-bit logical address output from the Z8001/3 CPU into a 24-bit address for the physical memory. (Only logical memory addresses go to an MMU for translation; I/O addresses and data bypass this component.)

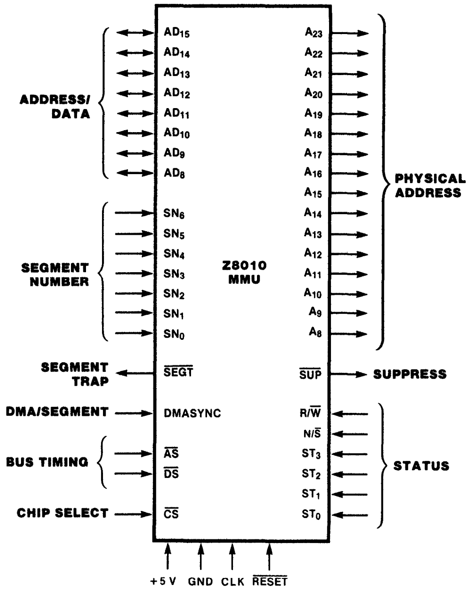


**Figure 1. Pin Functions**

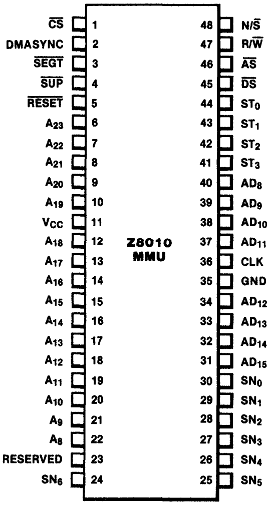


**Figure 2. Pin Assignments**

## Description (Continued)

Memory segments are variable in size from 256 bytes to 64K bytes, in increments of 256 bytes. Pairs of MMUs support the 128 segment numbers available for a Z8001/3 CPU address space. Within an address space, any number of MMUs can be used to accommodate multiple translation tables for System and Normal operating modes, or to support more sophisticated memory-management systems.

MMU memory-protection features safeguard memory areas from unauthorized or unintended access by associating special access restrictions with each segment. A segment is assigned a number of attributes when its descriptor enters into the MMU. When a memory reference is made, these attributes are checked against the status information supplied by the CPU. If a mismatch occurs, a trap is generated and the CPU is interrupted. The CPU can then check the status registers of the MMU to determine the cause.

Segments are protected by modes of permitted use, such as read only, system only, execute only and CPU-access only. Other segment management features include a write-warning zone useful for stack operations and status flags that record read or write accesses to each segment.

The MMU is controlled via 22 Special I/O instructions from the Z8001/3 CPU in System mode. With these instructions, system software can assign program segments to arbitrary memory locations, restrict the use of segments and monitor whether segments have been read or written.

## Segmented Addressing

A segmented address space—compared with linear addressing—is closer to the way a programmer uses memory because each procedure and data set can reside in its own segment.

The 8M byte Z8001/3 addressing spaces are divided into 128 relocatable segments of up to 64K bytes each. A 23-bit segmented address uses a 7-bit segment number to point to the segment, and a 16-bit offset to address any byte relative to the beginning of the segment. The two parts of the segmented address are manipulated separately.

The MMU divides the physical memory into 256-byte blocks. Segments consist of physically contiguous blocks. Certain segments may be so designated that writes into the last block generate a warning trap. If such a segment is used as a stack, this warning can be used to increase the segment size and prevent a stack overflow error.

The addresses manipulated by the programmer, used by instructions and output by the CPU are called *logical addresses.* The MMU takes the logical addresses and transforms them into the *physical addresses* required for accessing the memory (Figure 3). This address transformation process is called *relocation.*

The relocation process is transparent to user software. A translation table in the MMU associates the 7-bit segment number with the base address of the physical memory segment. The 16-bit logical address offset is added to the physical base address to obtain the actual physical memory location. Because a base address always has a low byte equal to zero, only the high-order 16 bits are stored in the MMU and used in the addition. Thus the low-order byte of the physical memory location is the same as the low-order byte of the logical address offset. This low-order byte therefore bypasses the MMU, thus reducing the number of pins required.

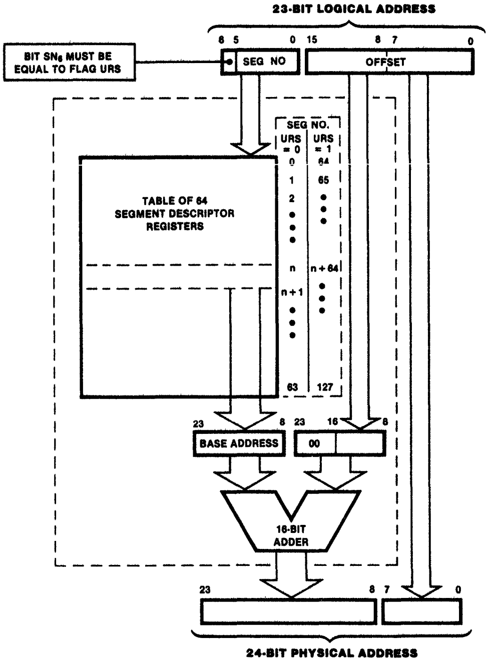


**Figure 3. Logical-to-Physical Address Translation**

# Z8015 Z8000™ PMMU Paged Memory Management Unit

Zilog

## Product Brief

June 1982

### Features

- PMMU architecture supports paged, virtual memory systems for the Z8003 VMPU.
- Dynamic page relocation makes software addresses independent of physical memory addresses.
- Memory-management features provide access validation to protect memory areas from unauthorized or unintentional access, and a write-warning indicator to prevent stack overflow.
- 64 pages, each 2048 bytes in length, can be mapped into a total physical address space of 16 megabytes.
- PMMU can be used to implement systems with larger or smaller page sizes.
- The number of accessible pages can be increased by using multiple PMMUs to support separate translation tables for each Z8003 VMPU address space.

### Description

The Z8015 Paged Memory Management Unit (PMMU) is designed to support a paged virtual memory system for the Z8003 Virtual Memory Processor Unit (VMPU). Although designed primarily for the Z8003, the PMMU can also be used to support the other CPUs in the Z8000 Family. Memory-management features allow access validation for memory protection and a write-warning to prevent stack overflow. An instruction abort for accesses to pages not in main memory allows restarting of instructions in the Z8003 VMPU. Each PMMU can manage a basic memory area of sixty-four 2048-byte, fixed-size pages. The VMPU's 8M byte logical address space is translated by the PMMU into a 16M byte physical address space. Page size can be easily changed and multiple PMMUs can be combined to support more pages. The PMMU is produced in a 64-pin package.

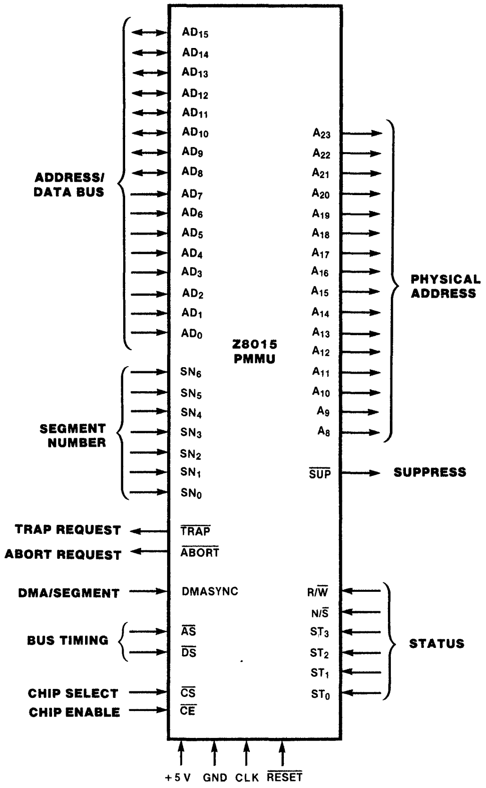


**Figure 1. Pin Functions**

## Functional Description

The Z8015 Paged Memory Management Unit (PMMU) manages the 8M byte addressing spaces of the Z8003 VMPU. The PMMU provides dynamic page relocation as well as numerous memory protection features.

Dynamic page relocation makes user software addresses independent of the physical memory addresses, thereby freeing the user from specifying where information is located in the physical memory. It also provides a flexible, efficient method for supporting multiprogramming systems. The PMMU uses a content-addressable translation table to transform the 23-bit logical address output from the VMPU into a 24-bit address for the physical memory. (Only logical memory addresses go to a PMMU for translation; I/O addresses and data bypass this component.)

The PMMU is designed to use a memory page 2048 bytes in length. Multiple PMMUs can be used to support more than 64 pages within a given address space. In addition, PMMUs can be used to accommodate separate translation tables for system and normal operating modes. The basic page length of 2048 bytes can be increased or decreased using a minimal amount of external circuitry.

The PMMU is designed to implement a paged virtual memory using the Z8003 VMPU. The PMMU saves sufficient information to recover from an instruction abort due to a page fault. The instruction can be restarted after the required information has been placed in primary memory and the PMMU's descrip tors updated to allow address translation to the selected primary memory locations.

As an aid in implementing efficient paging algorithms, the PMMU provides Changed and Referenced flags for each page. The Changed flag indicates that a page has been altered and hence must be copied to secondary storage before that physical memory can be used for another page. The Referenced flag can be used to determine which pages have not been accessed by an executing program. This information is useful in a variety of memory-management algorithms.

PMMU memory protection features safeguard memory areas from unauthorized or unintended access by associating special access restirctions with each page. A page is assigned a number of attributes when its descriptor is initially entered into the PMMU. Pages are protected by modes of permitted use, such as read only, system only, and execute only. The Valid flag indicates whether or not a descriptor has been initialized. When a memory reference is made, these attributes are checked against the status information supplied by the VMPU. If a mismatch occurs, the instruction is aborted, a Trap Request signal is generated and the VMPU is interrupted. The VMPU then checks the status registers of the PMMU to determine the cause of the abort.

The PMMU is controlled by 20 special I/O instructions, which can be issued from the VMPU in system mode only. With these instructions, system software can assign program pages to arbitrary memory locations, restrict the use of pages, and monitor whether pages have been read or written.

The PMMU has two operating modes: an address translation mode in which addresses are translated automatically as they are received, and a command mode, during which specific registers in the PMMU are accessed using special I/O commands.

## Segmented Addressing and Address Translation

The addresses manipulated by the programmer, used by instructions, and output by the VMPU are called logical addresses. The PMMU translates logical addresses into the physical addresses required for accessing the memory.

The 23-bit logical addresses output by the VMPU divide an 8M byte addressing space into 128 segments of up to 64K bytes each. A 23-bit segmented address consists of a 7-bit segment number and a 16-bit offset used to address any byte relative to the beginning of the segment. The two parts of the segmented address (segment number and offset) can be manipulated separately.

The PMMU divides physical memory into 2048-byte pages. Pages are assumed to be allocated in memory on 2048-byte boundaries so that the 11 low-order bits of the starting location of each page are always equal to zero. Segments in a virtual memory system can consist of pages that need not be in physical storage. Those segment pages in main memory need not be contiguous. Segments can have a variable number of pages. Any page can be designated so that writes into the lowest numbered 128 bytes generate a warning trap without an instruction abort. If such a page is used as the last page of the system stack, the warning trap can be used to allocate another page to the stack segment and prevent a stack overflow error.

# Z8030 Z8000™ Z-SCC Serial Communications Controller

Zilog

# Product Brief

March 1982

## Features

- Two independent, 0 to 1M bit/second, full-duplex channels, each with a separate crystal oscillator, baud rate generator, and Digital Phase-Locked Loop for clock recovery.
- Multi-protocol operation under program control; programmable for NRZ, NRZI, or FM data encoding.
- Asynchronous mode with five to eight bits and one, one and one-half, or two stop bits per character; programmable clock factor; break detection and generation; parity, overrun, and framing error detection.
- Synchronous mode with internal or external character synchronization on one or two sync characters and CRC generation and checking with CRC-16 or CRC-CCITT preset to either 1s or 0s.
- SDLC/HDLC mode with comprehensive frame-level control, automatic zero insertion and deletion, I-field residue handling, abort generation and detection, CRC generation and checking, and loop mode operation.
- Local loopback and auto-echo modes.

## Description

The Z-SCC Serial Communication Controller is a dual-channel, multi-protocol data communication peripheral for Z-BUS use. It is software-configured to satisfy a wide variety of serial communication applications. Its basic function is serial-to-parallel and parallel-to-serial conversion. In addition, the Z-SCC has internal functions that minimize the need for external random logic on the circuit card.

The Z-SCC handles asynchronous formats, synchronous byte-oriented protocols such as IBM Bisync, and synchronous bit-oriented protocols such as HDLC and IBM SDLC. It also supports virtually any other serial data transfer application (cassette or diskette interface, for example).

The device can generate and check CRC codes in any synchronous mode and can be

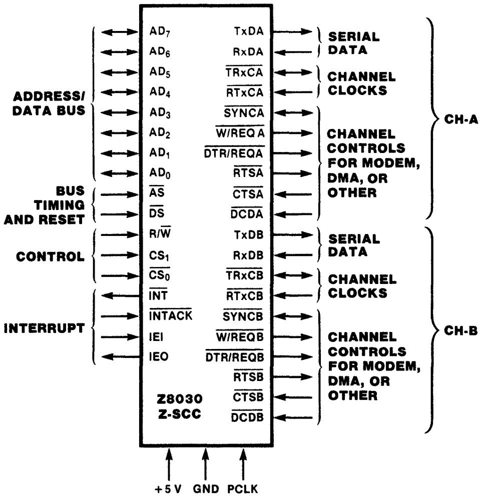


**Figure 1. Pin Functions**

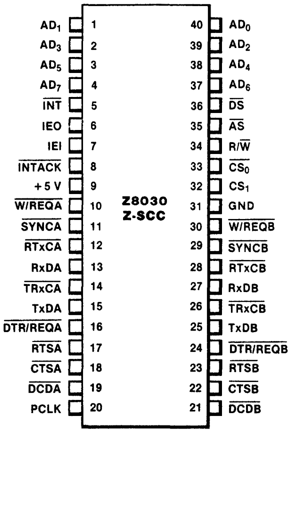


**Figure 2. Pin Assignments**

## Description (Continued)

programmed to check data integrity in various modes. It also has facilities for modem controls in both channels. In applications where these controls are not needed, the modem controls can be used for general-purpose I/O.

As is standard among Zilog peripheral components, the Z-BUS daisy-chain interrupt heirarchy is supported.

The Z-SCC contains the necessary multiplexed address/data bus interface with strobe and chip select lines to function as a Z-BUS peripheral. It includes internal control and interrupt logic, two full-duplex channels and two baud-rate generators. Associated with each channel are several read and write registers for mode control as well as the logic necessary to interface to modems or other external devices.

The read and write register group for each channel includes eight control registers, two sync-character registers, and four status registers. Each baud rate generator has two read/write registers for holding the time constant that determines baud rate. Associated with the interrupt logic is a write register for interrupt vector and three read registers: vector with status, vector without status, and interrupt pending status.

The logic for each channel provides formatting, synchronization and validation for data transferred to and from the channel interface. The modem control inputs are monitored by the control logic under program control. All of the modem control signals are general purpose in nature and optionally can be used for functions other than modem control.

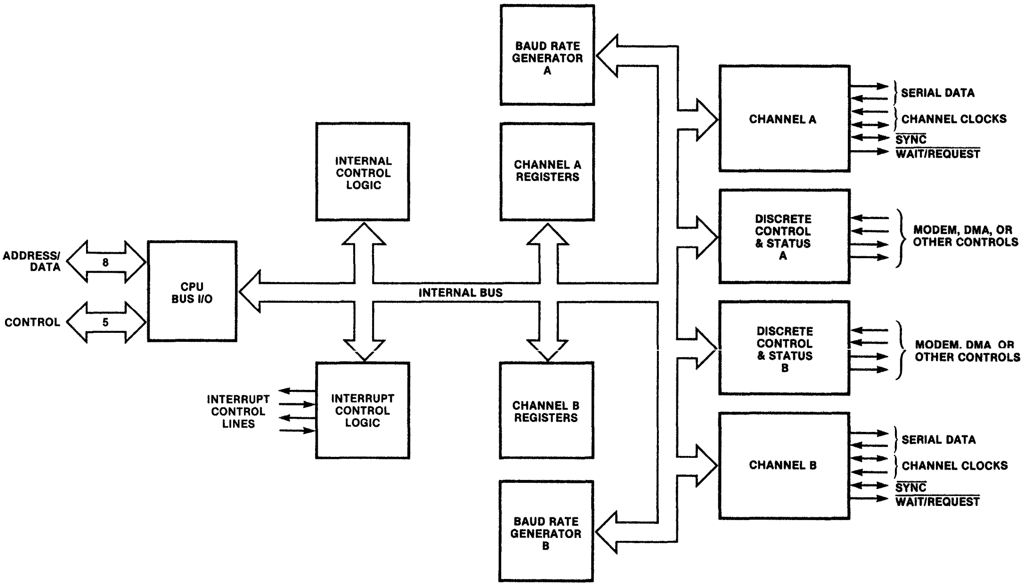


**Figure 3. Functional Block Diagram**

## Typical Applications

Figure 4 shows how a Z-SCC can be connected with Channel A programmed for the Synchronous Data Link Control (SDLC) Loop mode, functioning as a secondary station. If NRZI or FM coding is used, no clock lines are required because the clock can be recovered from the received data, using the Z-SCC's on-chip Digital Phase Locked Loop (DPLL). Another Z-SCC (not shown), programmed for the SDLC mode, would be the controlling station, polling the loop for traffic. The figure shows a typical, asynchronous serial port being serviced by Channel B of the Z-SCC. It could just as well support another synchronous data link, or even a high-speed link, transferring data via a DMA controller.

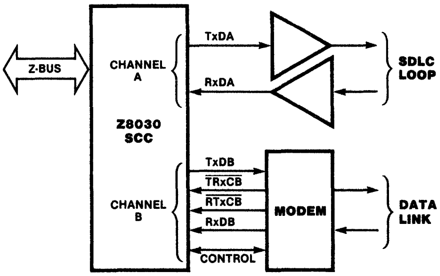


**Figure 4. Loop Secondary Station and Serial Port**

# Z8036 Z8000™ Z-CIO Counter/Timer and Parallel I/O Unit

Zilog

## Product Brief

March 1982

### Features

- Two independent 8-bit, double-buffered, bidirectional I/O ports plus a 4-bit special-purpose I/O port. I/O ports feature programmable polarity, programmable direction (Bit mode), "pulse catchers," and programmable open-drain outputs.
- Four handshake modes, including 3-Wire (like the IEEE-488).
- REQUEST/$\overline{\text{WAIT}}$ signal for high-speed data transfer.
- Flexible pattern-recognition logic, programmable as a 16-vector priority interrupt controller.
- Three 16-bit counter/timers with up to four external access lines per counter/timer (count input, output, gate, and trigger), and three output duty cycles (pulsed, one-shot, and square-wave), programmable as retriggerable or nonretriggerable.
- Easy to use since all registers are read/write and directly addressable.

### Description

The Z8036 Z-CIO Counter/Timer and Parallel I/O element is a general-purpose peripheral circuit, satisfying most counter/timer and parallel I/O needs encountered in system designs. This versatile device contains three I/O ports and three counter/timers. Many programmable options tailor its configuration to specific applications. The use of the device is simplified by making all internal registers (command, status, and data) readable and (except for status bits) writable. In addition, each register is given its own unique address so that it can be accessed directly—no special sequential operations are required. The Z-CIO is directly Z-BUS compatible.

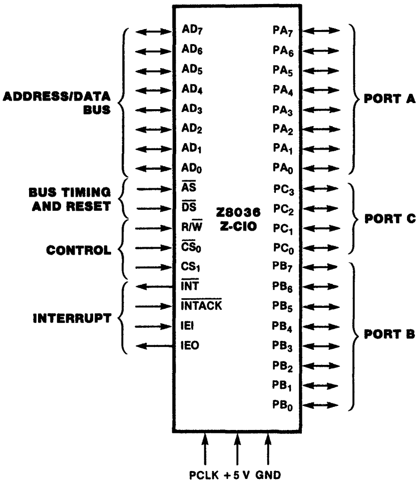

ADDRESS/DATA BUS
AD$_7$ AD$_6$ AD$_5$ AD$_4$ AD$_3$ AD$_2$ AD$_1$ AD$_0$
BUS TIMING AND RESET
$\overline{AS}$ $\overline{DS}$
CONTROL
R/$\overline{W}$ $\overline{CS_0}$ CS$_1$
INTERRUPT
$\overline{INT}$ $\overline{INTACK}$ IEI IEO
Z8036 Z-CIO
PA$_7$ PA$_6$ PA$_5$ PA$_4$ PA$_3$ PA$_2$ PA$_1$ PA$_0$
PORT A
PC$_3$ PC$_2$ PC$_1$ PC$_0$
PORT C
PB$_7$ PB$_6$ PB$_5$ PB$_4$ PB$_3$ PB$_2$ PB$_1$ PB$_0$
PORT B
PCLK +5 V GND

**Figure 1. Pin Functions**

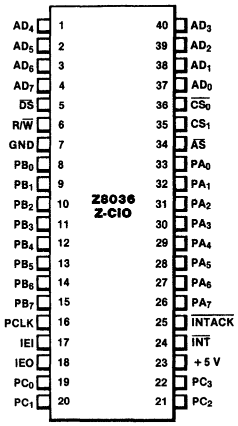


**Figure 2. Pin Assignments**

## Architecture

The Z8036 Z-CIO consists of a Z-BUS interface, three I/O ports (Ports A and B are general-purpose 8-bit ports linkable into a 16-bit port; Port C is a special-purpose 4-bit port), three 16-bit Counter/Timers (C/T 1, C/T 2, C/T 3), an interrupt control logic block, and an internal control logic block. Ports A and B are identical; B also is able to provide external access to C/T 1 and C/T 2. Either port can be specified as a handshake-driven, double-buffered port (input, output, or bidirectional) or a control-type port with programmable individual bit direction. Pattern recognition interrupt generation on match is provided; one mode facilitates implementing a priority interrupt controller.

Ports A and B each contain 12 registers. The Data Path registers are the Input, Output, and Buffer registers. The Mode Specification and Handshake Specification registers define the mode of operation of the ports. The reference pattern (for pattern match) is specified by the Pattern Polarity, Pattern Transition, and Pattern Mask registers. Detailed characteristics of the bit paths are controlled by the Data Path Polarity, Data Direction, and Special I/O Control registers. The Command and Status register contains the primary control and status bits. Registers associated with unused capabilities do not need initialization.

Port C provides handshake lines for Ports A and B as needed. Unused lines can provide external access to C/T 3 or to bit I/O. Port C has five registers. The Data Path registers are the Input and Output registers. The bit path definition registers are the Data Path Polarity, Data Direction, and Special I/O Control registers.

The three identical Counter/Timers each consist of a 16-bit down-counter, a 16-bit Time Constant register (which holds the initial down-counter value), a 16-bit Current Count register (for reading the down-counter contents), and C/T Mode Specification and C/T Command and Status registers. Counter input, gate input, trigger input, and C/T output lines are optionally available, as are the pulse, one-shot, or square-wave C/T output duty cycles. Each C/T can be programmed as retriggerable or not.

The interrupt control logic provides standard Z-BUS interrupt capabilities. There are five registers (Master Interrupt Control register, three Interrupt Vector registers, and the Current Vector register) associated with the interrupt logic. In addition, the ports' Command and Status registers and the counter/timers' Command and Status registers include bits associated with the interrupt logic. Each of these registers contains three bits for interrupt control and status: Interrupt Pending (IP), Interrupt Under Service (IUS), and Interrupt Enable (IE).

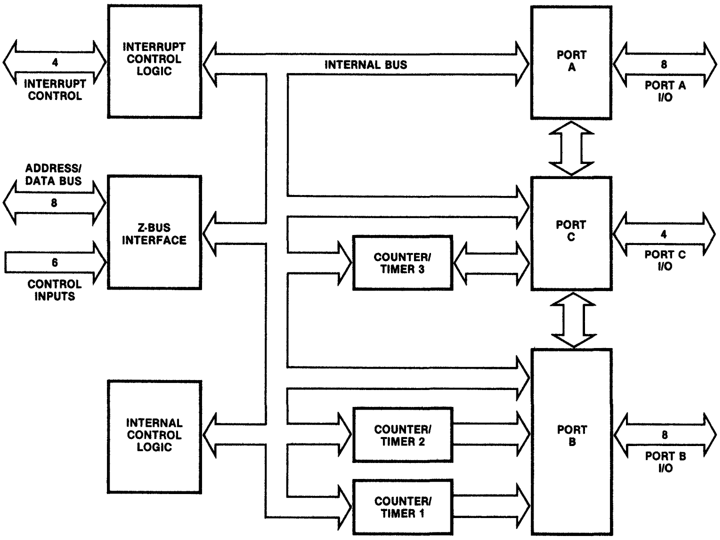


**Figure 3. Block Diagram**

# Z8038 Z8000™ Z-FIO FIFO Input/ Output Interface Unit

Zilog

## Product Brief

March 1982

### Features

- 128-byte FIFO buffer provides asynchronous bidirectional CPU/CPU or CPU/peripheral interface, expandable to any width in byte increments by use of multiple FIOs.
- Interlocked 2-Wire or 3-Wire Handshake logic port mode; Z-BUS or non-Z-BUS interface.
- Pattern-recognition logic stops DMA transfers and/or interrupts CPU; preset byte count can initiate variable-length DMA transfers.
- Seven sources of vectored/nonvectored interrupt which include pattern-match, byte count, empty or full buffer status; a dedicated "mailbox" register with interrupt capability provides CPU/CPU communication.
- $\overline{\text{REQUEST}}/\overline{\text{WAIT}}$ lines control high-speed data transfers.
- All functions are software controlled via directly addressable read/write registers.

### Description

The Z8038 FIO provides an asynchronous 128-byte FIFO buffer between two CPUs or between a CPU and a peripheral device. This buffer interface expands to a 16-bit or wider data path and expands in depth to add as many Z8060 FIFOs as are needed.

The FIO manages data transfers by assuming Z-BUS, non-Z-BUS microprocessor (a generalized microprocessor interface), Interlocked 2-Wire Handshake, and 3-Wire Handshake operating modes. These modes interface dissimilar CPUs or CPUs and peripherals running under differing speeds or protocols, allowing asynchronous data transactions and improving I/O overhead by as much as two orders of magnitude. Figures 1 and 2 show how the signals controlling these operating modes are mapped to the FIO pins.

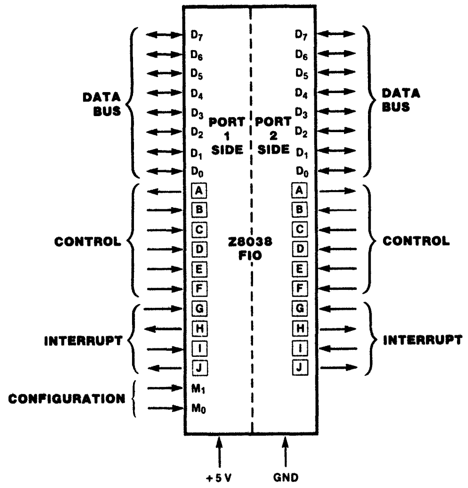


Figure 1. Pin Functions

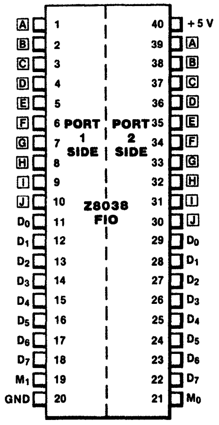


Figure 2. Pin Assignments

## Description (Continued)

The FIO supports the Z-BUS interrupt protocols, generating interrupts upon any of the following seven events: a write to a message register, change in data direction, pattern match, status match, over/underflow error, buffer full and buffer empty status. Each interrupt source can be enabled or disabled, and can also place an interrupt vector on the port address/data lines.

The data transfer logic of the FIO has been specially designed to work with DMA (Direct Memory Access) devices for high-speed transfers. It provides for data transfers to or from memory each machine cycle, while the DMA device generates memory address and control signals. The FIO also supports variably sized block length, improving system throughput when multiple variable length messages are transferred.

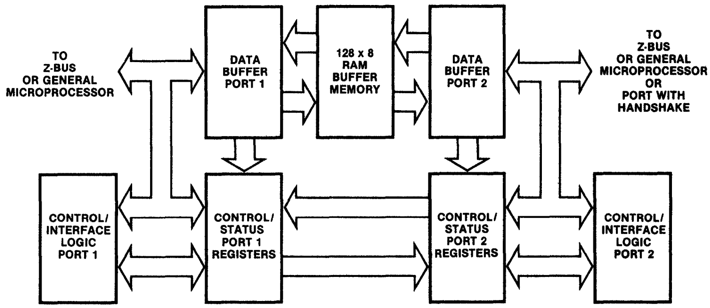

TO Z-BUS OR GENERAL MICROPROCESSOR
DATA BUFFER PORT 1
128 x 8 RAM BUFFER MEMORY
DATA BUFFER PORT 2
TO Z-BUS OR GENERAL MICROPROCESSOR OR PORT WITH HANDSHAKE
CONTROL/ INTERFACE LOGIC PORT 1
CONTROL/ STATUS PORT 1 REGISTERS
CONTROL/ STATUS PORT 2 REGISTERS
CONTROL/ INTERFACE LOGIC PORT 2

**Figure 3. Functional Block Diagram**

## Functional Description

**Operating Modes.** Ports 1 and 2 operate in any of the twelve combinations of operating modes listed in Table 2. Port 1 functions in either the Z-BUS or non-Z-BUS microprocessor modes, while Port 2 functions in Z-BUS, non-Z-BUS, Interlocked 2-Wire Handshake, and 3-Wire Handshake modes. Table 1 describes the signals and their corresponding pins in each of these modes.

The pin diagrams of the FIO ports are identical, except for two pins on the Port 1 side, which select that port's operating mode. Port 2's operating mode is programmed by two bits in Port 1's Control register 0. Table 2 describes the combinations of operating modes; Table 1 describes the control signals mapped to pins A–J in the five possible operating modes.

| Signal Pins | Z-BUS Low Byte | Z-BUS High Byte | Non-Z-BUS | Interlocked HS Port* | 3-Wire HS Port* |
|---|---|---|---|---|---|
| $\boxed{\text{A}}$ | $\overline{\text{REQ}}/\overline{\text{WT}}$ | $\overline{\text{REQ}}/\overline{\text{WT}}$ | $\overline{\text{REQ}}/\overline{\text{WT}}$ | RFD/$\overline{\text{DAV}}$ | RFD/$\overline{\text{DAV}}$ |
| $\boxed{\text{B}}$ | $\overline{\text{DMASTB}}$ | $\overline{\text{DMASTB}}$ | $\overline{\text{DACK}}$ | $\overline{\text{ACKIN}}$ | $\overline{\text{DAV}}$/DAC |
| $\boxed{\text{C}}$ | $\overline{\text{DS}}$ | $\overline{\text{DS}}$ | $\overline{\text{RD}}$ | FULL | DAC/RFD |
| $\boxed{\text{D}}$ | R/$\overline{\text{W}}$ | R/$\overline{\text{W}}$ | $\overline{\text{WR}}$ | EMPTY | EMPTY |
| $\boxed{\text{E}}$ | $\overline{\text{CS}}$ | $\overline{\text{CS}}$ | $\overline{\text{CE}}$ | $\overline{\text{CLEAR}}$ | $\overline{\text{CLEAR}}$ |
| $\boxed{\text{F}}$ | $\overline{\text{AS}}$ | $\overline{\text{AS}}$ | C/$\overline{\text{D}}$ | DATA DIR | DATA DIR |
| $\boxed{\text{G}}$ | $\overline{\text{INTACK}}$ | $A_0$ | $\overline{\text{INTACK}}$ | $\text{IN}_0$ | $\text{IN}_0$ |
| $\boxed{\text{H}}$ | IEO | $A_1$ | IEO | $\text{OUT}_1$ | $\text{OUT}_1$ |
| $\boxed{\text{I}}$ | IEI | $A_2$ | IEI | $\overline{\text{OE}}$ | $\overline{\text{OE}}$ |
| $\boxed{\text{J}}$ | $\overline{\text{INT}}$ | $A_3$ | $\overline{\text{INT}}$ | $\text{OUT}_3$ | $\text{OUT}_3$ |

*2 side only.

**Table 1. Control Signal Assignments**

| Mode Control Bits | | | | Operating Mode | |
|---|---|---|---|---|---|
| $M_1$ | $M_0$ | $B_1$ | $B_0$ | Port 1 | Port 2 |
| 0 | 0 | 0 | 0 | Z-BUS Low Byte | Z-BUS Low Byte |
| 0 | 0 | 0 | 1 | Z-BUS Low Byte | Non-Z-BUS |
| 0 | 0 | 1 | 0 | Z-BUS Low Byte | 3-Wire Handshake |
| 0 | 0 | 1 | 1 | Z-BUS Low Byte | 2-Wire Handshake |
| 0 | 1 | 0 | 0 | Z-BUS High Byte | Z-BUS High Byte |
| 0 | 1 | 0 | 1 | Z-BUS High Byte | Non-Z-BUS |
| 0 | 1 | 1 | 0 | Z-BUS High Byte | 3-Wire Handshake |
| 0 | 1 | 1 | 1 | Z-BUS High Byte | 2-Wire Handshake |
| 1 | 0 | 0 | 0 | Non-Z-BUS | Z-BUS Low Byte |
| 1 | 0 | 0 | 1 | Non-Z-BUS | Non-Z-BUS |
| 1 | 0 | 1 | 0 | Non-Z-BUS | 3-Wire Handshake |
| 1 | 0 | 1 | 1 | Non-Z-BUS | 2-Wire Handshake |

**Table 2. Operating Modes**

# Z8060
# Z8000™ FIFO Buffer Unit and Z-FIO Expander

Zilog

## Product Brief

March 1982

## Features

- Asynchronous, bidirectional first-in, first-out buffer.
- Extends depth of Z-FIO without limit.
- 128 × 8 organization.
- 3-state data outputs.
- Empty and Full status pins are wire-ORed among multiple stages.

## Description

The Z-FIFO first-in, first-out buffer unit is a 128 × 8-bit memory with bidirectional data transfer capability and handshake logic. Its structure is similar to that of other FIFOs that are commonly available, such as the AM2812 and the 3351. The handshake logic used is compatible with that of the Z8, the Z-CIO, and Z-FIO. Z-FIFO buffers can be cascaded, end to end, without limit, their RFD/$\overline{\text{DAV}}$ and $\overline{\text{ACKIN}}$ signals daisy-chained, to make a FIFO array any desired number of words deep. Two such channels in parallel, suitably controlled, make up a 16-bit-wide buffer array.

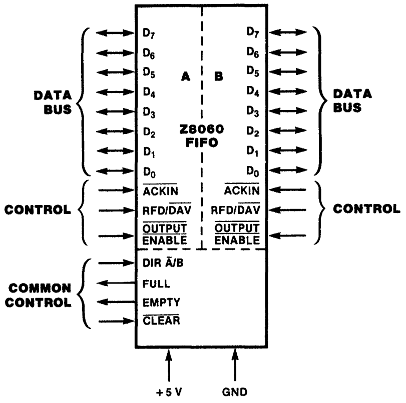


**Figure 1. Pin Functions**

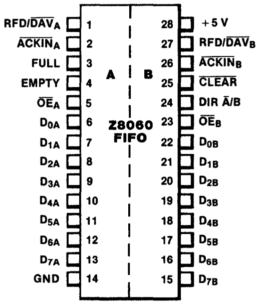


**Figure 2. Pin Assignments**

## Description
(Continued)

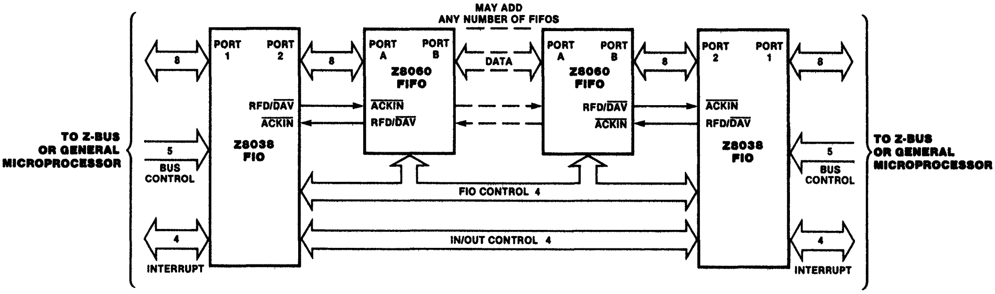


**Figure 3. Using FIFOs to Extend FIO Depth**

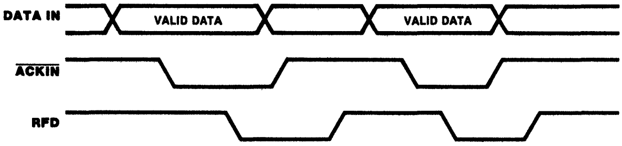


**Figure 4. Two-Wire Interlocked Handshake Timing (Input)**

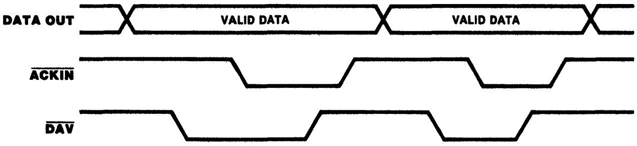


**Figure 5. Two-Wire Interlocked Handshake Timing (Output)**

# Z8090
# Z8000™ Z-UPC Universal Peripheral Controller

Zilog

## Product Brief

March 1982

### Features

- Complete slave microcomputer, for distributed processing Z-BUS use.
- Z8 architecture and instruction set.
- 2K bytes of on-chip ROM.
  - □ Available in standard or development configuration.
- Three programmable I/O ports, two with optional 2-Wire Handshake.
- Six levels of priority interrupts from eight sources: six external sources and two internal sources.
- Two programmable 8-bit counter/timers each with a 6-bit prescaler. Counter/Timer T0 is driven by an internal source, and Counter/Timer T1 can be driven by internal or external sources. Both counter/timers are independent of program execution.
- 256-byte register file, accessible by both the master CPU and Z-UPC, using a fail-safe message-passing protocol.

### Description

The Z8090 Universal Peripheral Controller (Z-UPC) is an intelligent peripheral controller for distributed processing applications (Figure 3). The Z-UPC unburdens the host processor by assuming tasks traditionally done by the host (or by added hardware), such as performing arithmetic, translating or formatting data, and controlling I/O devices. Based on the Z8 microcomputer architecture and instruction set, the Z-UPC contains 2K bytes of internal program ROM, a 256-byte register file, three 8-bit I/O ports, and two counter/timers.

The Z-UPC offers fast execution time; an effective use of memory; and sophisticated interrupt, I/O, and bit manipulation. Using a powerful and extensive instruction set combined with an efficient internal addressing scheme, the Z-UPC speeds program execution and efficiently packs program code into the on-chip ROM.

An important feature of the Z-UPC is an internal register file containing I/O port and control registers accessed both by the Z-UPC program and by its associated master CPU.

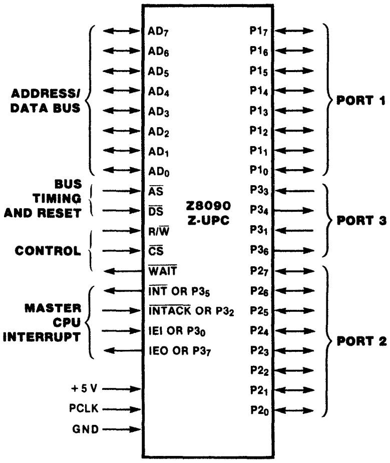


**Figure 1. Pin Functions**

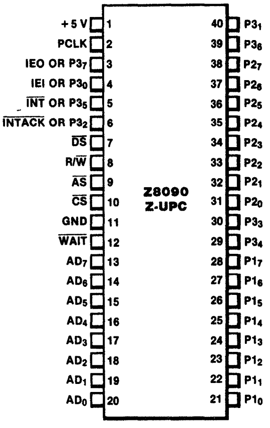


**Figure 2. Pin Assignments**

**Description** (Continued)

This architecture results in both byte and programming efficiency, because Z-UPC instructions can operate directly on I/O data without moving it to and from an accumulator. Such a structure allows the user to allocate as many general-purpose registers as the application requires for data buffers between the CPU and peripheral devices. All general-purpose registers can be used as address pointers, index registers, data buffers, or stack space.

The register file is logically divided into 16 groups, each consisting of 16 working registers. A Register Pointer is used in conjunction with short format instructions, resulting in tight, fast code and easy task switching.

Communication between the master CPU and the register file takes place via one group of 19 interface registers addressed directly by both the master CPU and the Z-UPC, or via a block transfer mechanism. Access by the master CPU is controlled by the Z-UPC to allow independence between the master CPU and Z-UPC software.

The Z-UPC has 24 pins that can be dedicated to I/O functions. Grouped logically into three 8-line ports, they can be programmed in many combinations of input or output lines, with or without handshake, and with push-pull or open-drain outputs. Ports 1 and 2 are bit-programmable; Port 3 has four fixed inputs and four outputs.

To relieve software from coping with real-time counting and timing problems, the Z-UPC has two 8-bit hardware counter/timers, each with a fixed divide-by-four, and a 6-bit programmable prescaler. Various counting modes may be selected.

In addition to the 40-pin standard configuration, the Z-UPC is available in four special configurations:

- A 64-pin RAM development version with external interface for up to 4K bytes of RAM and 36 bytes of internal ROM permitting down-loading from the master CPU.
- A Protopack RAM version with a socket for up to 2K bytes of RAM, with 36 bytes of internal ROM permitting down-loading from the master CPU.
- A 64-pin ROM development version with external interface for up to 4K bytes of ROM and no internal ROM.
- A Protopack ROM version with a socket for 2K bytes of ROM and no internal ROM.

This range of versions and configurations makes the Z-UPC compatible with most system peripheral device control methods.

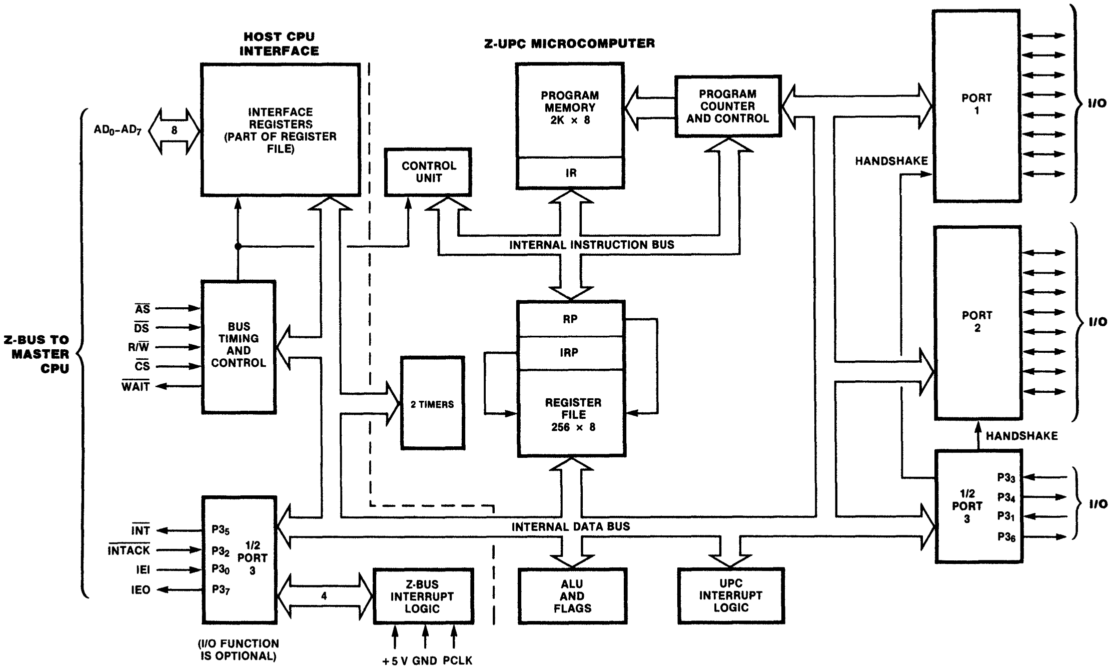


**Figure 3. Functional Block Diagram**

# Z-BUS®
# Component Interconnect

Zilog

## Descriptive Brief

March 1982

### Features

- Defines the interface protocols used by Z8000 family members for data transfer, interrupt signaling, and resource sharing.
- Provides multiplexed address/data bus shared by memory and I/O transfers, using separate protocols.
- Provides 16 or more memory address bits; 16-bit I/O addresses; 8 or 16 data bits.
- Allows direct addressing of registers within peripherals.
- Provides bus signals that allow separate CPU and peripheral clocks.
- Supports polling, vectored interrupts and non-vectored interrupts.
- Defines a simple priority interrupt scheme, without a separate controller, through a daisy-chain interrupt structure.
- Supports distributed control of bus and other shared resources through bus and resource request protocols.

### Description

The Z-BUS is the high-speed parallel shared bus that links components of the Z8000 Family and provides family members with a common communication interface that supports:

- *Data Transfer.* Data can be moved between bus masters (such as a CPU) and memories or peripherals.
- *Interrupts.* Interrupts can be generated by peripherals and serviced by CPUs over the bus.
- *Resource Control.* Distributed management of shared resources (including the bus itself) is supported by a daisy-chain priority mechanism.

The heart of the Z-BUS is a set of multiplexed address/data lines and the signals that control these lines. Multiplexing data and addresses onto the same lines makes more efficient use of pins, facilitates expansion of the number of data and address bits, and allows direct addressing of a peripheral's internal registers, which simplifies I/O programming.

A daisy-chained priority mechanism resolves interrupt and resource requests, thus allowing distributed control of the bus and eliminating the need for separate priority controllers. The resource-control daisy chain allows wide physical separation of components.

The Z-BUS is asynchronous in the sense that peripherals do not need to be synchronized with the CPU clock. All timing information is provided by Z-BUS signals.

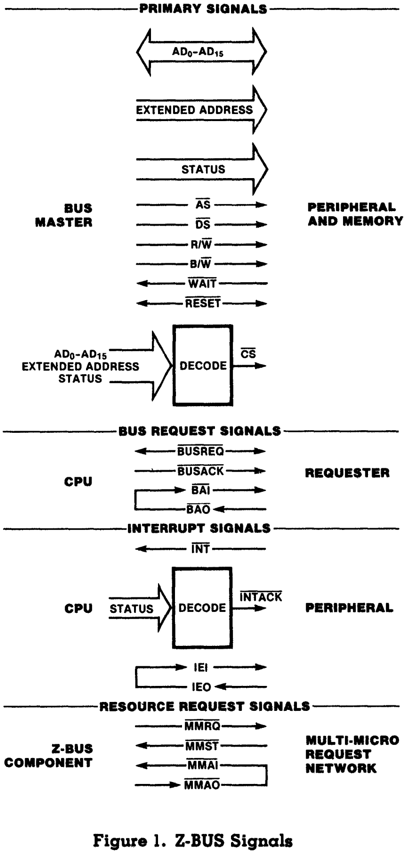


**Figure 1. Z-BUS Signals**

### Memory and I/O Data Transfers

When a processor accesses a memory location or I/O device via the Z-BUS, both the address and data are transferred over $AD_0$-$AD_{15}$. The address is transmitted while Address Strobe ($\overline{AS}$) is Low at the beginning of a transfer, and data is moved while Data Strobe ($\overline{DS}$) is Low at the end of a transfer (as shown in Figure 2). The status lines serve to distinguish between I/O and memory and among the various memory address spaces. The Read/Write (R/$\overline{W}$) line and Byte/Word (B/$\overline{W}$) line determine the type of transfer; $\overline{WAIT}$ allows slow memory or peripherals to delay data transfer.

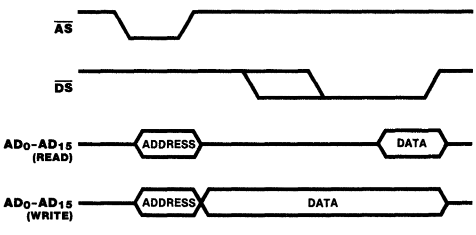
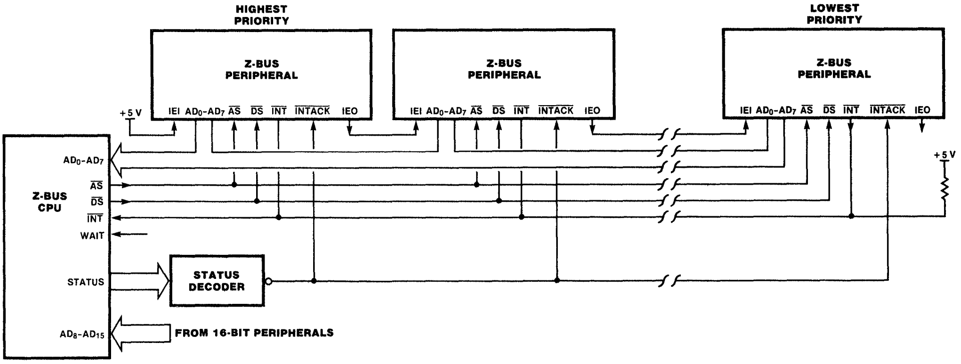


**Figure 2. Z-BUS Memory and I/O Transfers**

### Interrupt

The Z-BUS interrupt scheme is an interrupt-under-service priority daisy chain that requires no separate priority controller. Interrupt requests are all tied directly to the $\overline{INT}$ pin of the CPU. (See Figure 3.) Physical position along the IEI/IEO daisy chain determines the priority assigned to any given peripheral.

A complete interrupt cycle consists of an interrupt request followed by an interrupt-acknowledge transaction. The request, which consists of $\overline{INT}$ pulled Low by a peripheral, notifies the CPU that an interrupt is pending. The interrupt-acknowledge transaction, which is initiated by the CPU as a result of the request, performs two functions: 1) using the IEI/IEO daisy chain it selects the peripheral whose interrupt is to be acknowledged; 2) it obtains a vector that identifies the selected device and the cause of interrupt.



HIGHEST PRIORITY
LOWEST PRIORITY
Z-BUS PERIPHERAL
IEI $AD_0$-$AD_7$ $\overline{AS}$ $\overline{DS}$ $\overline{INT}$ $\overline{INTACK}$ IEO
Z-BUS PERIPHERAL
IEI $AD_0$-$AD_7$ $\overline{AS}$ $\overline{DS}$ $\overline{INT}$ $\overline{INTACK}$ IEO
Z-BUS PERIPHERAL
IEI $AD_0$-$AD_7$ $\overline{AS}$ $\overline{DS}$ $\overline{INT}$ $\overline{INTACK}$ IEO
+5 V
+5 V
Z-BUS CPU
$AD_0$-$AD_7$
$\overline{AS}$
$\overline{DS}$
$\overline{INT}$
WAIT
STATUS
$AD_8$-$AD_{15}$
STATUS DECODER
FROM 16-BIT PERIPHERALS

**Figure 3. Interrupt Connections**

### Bus and Resource Requests

For a device other than the CPU (which is default master) to gain control of the bus, it must make a bus request by forcing $\overline{BUSREQ}$ Low. After $\overline{BUSREQ}$ is pulled Low, the Z-BUS CPU relinquishes the bus and indicates this condition by pulling $\overline{BUSACK}$ Low. This Low signal is propagated through the $\overline{BAI}$/$\overline{BAO}$ daisy chain until it reaches a bus requester that is ready to use the bus.

This requester uses the bus and then releases $\overline{BUSREQ}$ and allows $\overline{BAO}$ to follow $\overline{BAI}$. When all simultaneously requesting devices have relinquished the bus, $\overline{BUSREQ}$ goes High, returning control of the bus to the CPU.

The resource request chain is used to share a resource among several Z-BUS CPUs, none of which is default master of that resource. The resource-request signals and protocol are similar to that of the bus request, except that there is no default master.

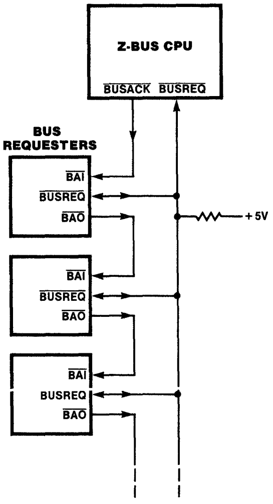


**Figure 4. Bus Request Connections**

# C

Zilog

LOWER NIBBLE (HEX), UPPER INSTRUCTION BYTE

UPPER NIBBLE (HEX), UPPER INSTRUCTION BYTE

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | **ADDB**<br>R ← IR<br>R ← IM | **ADD**<br>R – IR<br>R ← IM | **SUBB**<br>R ← IR<br>R ← IM | **SUB**<br>R ← IR<br>R ← IM | **ORB**<br>R ← IR<br>R ← IM | **OR**<br>R ← IR<br>R ← IM | **ANDB**<br>R ← IR<br>R ← IM | **AND**<br>R ← IR<br>R ← IM | **XORB**<br>R ← IR<br>R ← IM | **XOR**<br>R ← IR<br>R ← IM | **CPB**<br>R ← IR<br>R ← IM | **CP**<br>R ← IR<br>R ← IM | **See**<br>**Table**<br>**1** | **See**<br>**Table**<br>**1** | **EXTEND**<br>**INST** | **EXTEND**<br>**INST** |
| 1 | **CPL**<br>R ← IR<br>R ← IM | **PUSHL**<br>IR ← IR | **SUBL**<br>R ← IR<br>R ← IM | **PUSH**<br>IR ← IR | **LDL**<br>R ← IR<br>R ← IM | **POPL**<br>IR ← IR | **ADDL**<br>R ← IR<br>R ← IM | **POP**<br>IR ← IR | **MULTL**<br>R ← IR<br>R ← IM | **MULT**<br>R ← IR<br>R ← IM | **DIVL**<br>R ← IR<br>R ← IM | **DIV**<br>R ← IR<br>R ← IM | **See**<br>**Table**<br>**2** | **LDL**<br>IR ← R | **JP**<br>PC ← IR | **CALL**<br>PC ← IR |
| 2 | **LDB**<br>R ← IR<br>R ← IM | **LD**<br>R ← IR<br>R ← IM | **RESB**<br>IR ← IM<br>R ← R | **RES**<br>IR ← IM<br>R ← R | **SETB**<br>IR ← IM<br>R ← R | **SET**<br>IR ← IM<br>R ← R | **BITB**<br>IR ← IM<br>R ← R | **BIT**<br>IR ← IM<br>R ← R | **INCB**<br>IR ← IM | **INC**<br>IR ← IM | **DECB**<br>IR ← IM | **DEC**<br>IR ← IM | **EXB**<br>R ← IR | **EX**<br>R ← IR | **LDB**<br>IR ← R | **LD**<br>IR ← R |
| 3 | **LDB**<br>R ← BA<br>**LDRB**<br>R ← RA | **LD**<br>R ← BA<br>**LDR**<br>R ← RA | **LDB**<br>BA ← R<br>**LDRB**<br>RA ← R | **LD**<br>BA ← R<br>**LDR**<br>RA ← R | **LDA**<br>R ← BA<br>**LDAR**<br>R ← RA | **LDL**<br>R ← BA<br>**LDRL**<br>R ← RA | **RSVD** | **LDL**<br>BA ← R<br>**LDRL**<br>RA ← R | **RSVD** | **LDPS**<br>IR | **See**<br>**Table**<br>**3A** | **See**<br>**Table**<br>**3B** | **INB**<br>R ← IR | **IN**<br>R ← IR | **OUTB**<br>IR ← R | **OUT**<br>IR ← R |
| 4 | **ADDB**<br>R ← X<br>R ← DA | **ADD**<br>R ← X<br>R ← DA | **SUBB**<br>R ← X<br>R ← DA | **SUB**<br>R ← X<br>R ← DA | **ORB**<br>R ← X<br>R ← DA | **OR**<br>R ← X<br>R ← DA | **ANDB**<br>R ← X<br>R ← DA | **AND**<br>R ← X<br>R ← DA | **XORB**<br>R ← X<br>R ← DA | **XOR**<br>R ← X<br>R ← DA | **CPB**<br>R ← X<br>R ← DA | **CP**<br>R ← X<br>R ← DA | **See**<br>**Table**<br>**1** | **See**<br>**Table**<br>**1** | **EXTEND**<br>**INST** | **EXTEND**<br>**INST** |
| 5 | **CPL**<br>R ← X<br>R ← DA | **PUSHL**<br>IR ← X<br>IR ← DA | **SUBL**<br>R ← X<br>R ← DA | **PUSH**<br>IR ← X<br>IR ← DA | **LDL**<br>R ← X<br>R ← DA | **POPL**<br>IR → X<br>IR → DA | **ADDL**<br>R ← X<br>R ← DA | **POP**<br>IR → X<br>IR → DA | **MULTL**<br>R ← X<br>R ← DA | **MULT**<br>R ← X'<br>R ← DA | **DIVL**<br>R ← X<br>R ← DA | **DIV**<br>R ← X | **See**<br>**Table**<br>2 | **LDL**<br>X ← R<br>DA ← R | **JP**<br>PC ← X<br>PC ← DA | **CALL**<br>PC ← X<br>PC ← DA |
| 6 | **LDB**<br>R ← X<br>R ← DA | **LD**<br>R ← X<br>R ← DA | **RESB**<br>X ← IM<br>DA ← IM | **RES**<br>X ← IM<br>DA ← IM | **SETB**<br>X ← IM<br>DA ← IM | **SET**<br>X ← IM<br>DA ← IM | **BITB**<br>X ← IM<br>DA ← IM | **BIT**<br>X ← IM<br>DA ← IM | **INCB**<br>X ← IM<br>DA ← IM | **INC**<br>X ← IM<br>DA ← IM | **DECB**<br>X ← IM<br>DA ← IM | **DEC**<br>X ← IM<br>DA ← IM | **EXB**<br>R ↔ X<br>R ↔ DA | **EX**<br>R ↔ X<br>R ↔ DA | **LDB**<br>X ← R<br>DA ← R | **LD**<br>X ← R<br>DA ← R |
| 7 | **LDB**<br>R ← BX | **See**<br>**Table**<br>**7** | **LDB**<br>BX ← R | **LD**<br>BX ← R | **LDA**<br>R ← BX | **LDL**<br>R ← BX | **LDA**<br>R ← X<br>R ← DA | **LDL**<br>BX ← R | **RSVD** | **LDPS**<br>PS ← X<br>PS ← DA | **HALT** | **See**<br>**Table**<br>**7** | **EI**<br>**DI** | **See**<br>**Table**<br>**7** | **RSVD** | **SC** |
| 8 | **ADDB**<br>R ← R | **ADD**<br>R ← R | **SUBB**<br>R ← R | **SUB**<br>R ← R | **ORB**<br>R ← R | **OR**<br>R ← R | **ANDB**<br>R ← R | **AND**<br>R ← R | **XORB**<br>R ← R | **XOR**<br>R ← R | **CPB**<br>R ← R | **CP**<br>R ← R | **See**<br>**Table**<br>**1** | **See**<br>**Table**<br>**1** | **EXTEND**<br>**INST.** | **EXTEND**<br>**INST.** |
| 9 | **CPL**<br>R ← R | **PUSHL**<br>IR ← R | **SUBL**<br>R ← R | **PUSH**<br>IR ← R | **LDL**<br>R ← R | **POPL**<br>R ← IR | **ADDL**<br>R ← R | **POP**<br>R ← IR | **MULTL**<br>R ← R | **MULT**<br>R ← R | **DIVL**<br>R ← R | **DIV**<br>R ← R | **See**<br>**Table**<br>**2** | **RSVD** | **RET**<br>PC ← (SP) | **RSVD** |
| A | **LDB**<br>R ← R | **LD**<br>R ← R | **RESB**<br>R ← IM | **RES**<br>R ← IM | **SETB**<br>R ← IM | **SET**<br>R ← IM | **BITB**<br>R ← IM | **BIT**<br>R ← IM | **INCB**<br>R ← IM | **INC**<br>R ← IM | **DECB**<br>R ← IM | **DEC**<br>R ← IM | **EXB**<br>R ↔ R | **EX**<br>R ↔ R | **TCCB**<br>R | **TCC**<br>R |
| B | **DAB**<br>R | **EXTS**<br>**EXTSB**<br>**EXTSL**<br>**R** | **See**<br>**Table**<br>**4** | **See**<br>**Table**<br>**4** | **ADCB**<br>R ← R | **ADC**<br>R ← R | **SBCB**<br>R ← R | **SBC**<br>R ← R | **See**<br>**Table**<br>**5** | **RSVD** | **See**<br>**Table**<br>**6** | **See**<br>**Table**<br>**6** | **RRDB**<br>R | **LDK**<br>R ← IM | **RLDB**<br>R | **RSVD** |
| C | **LDB**<br>R ← IM | ⟶ | | | | | | | | | | | | | | |
| D | **CALR**<br>PC ← RA | ⟶ | | | | | | | | | | | | | | |
| E | **JR**<br>PC ← RA | ⟶ | | | | | | | | | | | | | | |
| F | **DJNZ**<br>**DBJNZ**<br>PC ← RA | ⟶ | | | | | | | | | | | | | | |

**Op Code Map**

Notes:

1) Reserved Instructions (RSVD) should not be used. The result of their execution is not defined.

2) The execution of an extended instruction will result in an Extended Instruction Trap if the EPA bit in the FCW is a zero. If the flag is a one the Extended Instruction will be executed by the EPU function.

| LOWER NIBBLE (HEX), LOWER INSTRUCTION BYTE | 0C | 0D | 4C | 4D | 8C | 8D |
|---|---|---|---|---|---|---|
| 0 | **COMB**<br>IR | **COM**<br>IR | **COMB**<br>X<br>DA | **COM**<br>X<br>DA | **COMB**<br>R | **COM**<br>R |
| 1 | **CPB**<br>IR,IM | **CP**<br>IR,IM | **CPB**<br>X,IM<br>DA,IM | **CP**<br>X,IM<br>DA,IM | **LDCTLB**<br>R←FLGS | **SETFLG** |
| 2 | **NEGB**<br>IR | **NEG**<br>IR | **NEGB**<br>X<br>DA | **NEG**<br>X<br>DA | **NEGB**<br>R | **NEG**<br>R |
| 3 | RSVD | RSVD | RSVD | RSVD | RSVD | **RESFLG** |
| 4 | **TESTB**<br>IR | **TEST**<br>IR | **TESTB**<br>X<br>DA | **TEST**<br>X<br>DA | **TESTB**<br>R | **TEST**<br>R |
| 5 | **LDB**<br>IR←IM | **LD**<br>IR←IM | **LDB**<br>X←IM<br>DA←IM | **LD**<br>X←IM<br>DA←IM | RSVD | **COMFLG** |
| 6 | **TSETB**<br>IR | **TSET**<br>IR | **TSETB**<br>X<br>DA | **TSET**<br>X<br>DA | **TSETB**<br>R | **TSET**<br>R |
| 7 | RSVD | RSVD | RSVD | RSVD | RSVD | **NOP** |
| 8 | **CLRB**<br>IR | **CLR**<br>IR | **CLRB**<br>X<br>DA | **CLR**<br>X<br>DA | **CLRB**<br>R | **CLR**<br>R |
| 9 |  | **PUSH**<br>IM |  |  | **LDCTLB**<br>FLGS←R |  |

**Table 1. Upper Instruction Byte**

| LOWER NIBBLE (HEX), LOWER INSTRUCTION BYTE | 1C | 5C | 9C |
|---|---|---|---|
| 0 | RSVD | RSVD | RSVD |
| 1 | **LDM**<br>R←IR | **LDM**<br>R←X<br>R←DA |  |
| 8 | **TESTL**<br>IR | **TESTL**<br>X<br>DA | **TESTL**<br>R |
| 9 | **LDM**<br>IR←R | **LDM**<br>X←R<br>DA←R |  |

**Table 2. Upper Instruction Byte**

| LOWER NIBBLE (HEX), LOWER INSTRUCTION BYTE | 3A | 3B |
|---|---|---|
| 0 | **INIB**<br>IR←IR<br>**INIRB**<br>IR←IR | **INI**<br>IR←IR<br>**INIR**<br>IR←IR |
| 1 | **SINIB**<br>IR←IR<br>**SINIRB**<br>IR←IR | **SINI**<br>IR←IR<br>**SINIR**<br>IR←IR |
| 2 | **OUTIB**<br>IR←IR<br>**OTIRB**<br>IR←IR | **OUTI**<br>IR←IF<br>**OUTIR**<br>IR←IR |
| 3 | **SOUTIB**<br>IR←IR<br>**SOTIRB**<br>IR←IR | **SOUTI**<br>IR←IR<br>**SOTIR**<br>IR←IR |
| 4 | **INB**<br>R←DA | **IN**<br>R←DA |
| 5 | **SINB**<br>R←DA | **SIN**<br>R←DA |
| 6 | **OUTB**<br>DA←R | **OUT**<br>DA←R |
| 7 | **SOUTB**<br>DA←R | **SOUT**<br>DA←R |
| 8 | **INDB**<br>IR←IR<br>**INDRB**<br>IR←IR | **IND**<br>IR←IR<br>**INDR**<br>IR←IR |
| 9 | **SINDB**<br>IR←IR<br>**SINDRB**<br>IR←IR | **SIND**<br>IR←IR<br>**SINDR**<br>IR←IR |
| A | **OUTDB**<br>IR←IR<br>**OTDRB**<br>IR←IR | **OUTD**<br>IR←IR<br>**OTDR**<br>IR←IR |
| B | **SOUTDB**<br>IR←IR<br>**SOTDRB**<br>IR←IR | **SOUTD**<br>IR←IR<br>**SOTDR**<br>IR←IR |

**Table 3. Upper Instruction Byte**

| LOWER NIBBLE (HEX), LOWER INSTRUCTION BYTE | B2 | B3 |
|---|---|---|
| 0 | **RLB**<br>(1 bit)<br>R | **RL**<br>(1 bit)<br>R |
| 1 | **SLLB**<br>R<br>**SRLB**<br>R | **SLL**<br>R<br>**SRL**<br>R |
| 2 | **RLB**<br>(2 bits)<br>R | **RL**<br>(2 bits)<br>R |
| 3 | **SDLB**<br>R | **SDL**<br>R |
| 4 | **RRB**<br>(1 bit)<br>R | **RR**<br>(1 bit)<br>R |
| 5 | RSVD | **SLLL**<br>**R**<br>**SRLL** |
| 6 | **RRB**<br>(2 bits)<br>R | **RR**<br>(2 bits)<br>R |
| 7 | RSVD | **SDLL**<br>R |
| 8 | **RLCB**<br>(1 bit)<br>R | **RLC**<br>(1 bit)<br>R |
| 9 | **SLAB**<br>R<br>**SRAB**<br>**R** | **SLA**<br>R<br>**SRA**<br>**R** |
| A | **RLCB**<br>(2 bits)<br>R | **RLC**<br>(2 bits)<br>R |
| B | **SDAB**<br>R | **SDA**<br>R |
| C | **RRCB**<br>(1 bit)<br>R | **RRC**<br>(1 bit)<br>R |
| D | RSVD | **SLAL**<br>R<br>**SRAL** |
| E | **RRCB**<br>(2 bits)<br>R | **RRC**<br>(2 bits)<br>R |
| F | RSVD | **SDAL**<br>R |

**Table 4.**
**Upper Instruction Byte**

| LOWER NIBBLE (HEX), LOWER INSTRUCTION BYTE | B8 |
|---|---|
| 0 | **TRIB**<br>IR |
| 1 | RSVD |
| 2 | **TRTIB**<br>IR |
| 3 | RSVD |
| 4 | **TRIRB**<br>IR |
| 5 | RSVD |
| 6 | **TRTIRB**<br>IR |
| 7 | RSVD |
| 8 | **TRDB**<br>IR |
| 9 | RSVD |
| A | **TRTDB**<br>IR |
| B | RSVD |
| C | **TRDRB**<br>IR |
| D | RSVD |
| E | **TRTDRB**<br>IR |
| F | RSVD |

**Table 5.**
**Upper Instruction Byte**

| | BA | BB |
|---|---|---|
| 0 | **CPIB**<br>IR | **CPI**<br>IR |
| 1 | **LDIB**<br>IR←IR<br>**LDIRB**<br>IR←IR | **LDI**<br>IR←IR<br>**LDIR**<br>IR←IR |
| 2 | **CPSIB**<br>IR | **CPSI**<br>IR |
| 3 | RSVD | RSVD |
| 4 | **CPRIB**<br>IR | **CPIR**<br>IR |
| 5 | RSVD | RSVD |
| 6 | **CPSIRB**<br>IR | **CPSIR**<br>IR |
| 7 | RSVD | RSVD |
| 8 | **CPDB**<br>IR | **CPD**<br>IR |
| 9 | **LDDB**<br>IR←IR<br>**LDDRB**<br>IR←IR | **LDD**<br>IR←IR<br>**LDDR**<br>IR←IR |
| A | **CPSDB**<br>IR | **CPSD**<br>IR |
| B | RSVD | RSVD |
| C | **CPDRB**<br>IR | **CPDR**<br>IR |
| D | RSVD | RSVD |
| E | **CPSDRB**<br>IR | **CPSDR**<br>IR |
| F | RSVD | RSVD |

**Table 6.**
**Upper Instruction Byte**

| | 7B | 7D |
|---|---|---|
| 0 | **IRET**<br>PC←(SSP) | RSVD |
| 1 | RSVD | RSVD |
| 2 | RSVD | **LDCTL**<br>R←FCW |
| 3 | RSVD | **LDCTL**<br>R←RFRSH |
| 4 | RSVD | **LDCTL**<br>R←<br>PSAPSEG |
| 5 | RSVD | **LDCTL**<br>R←<br>PSAPOFF |
| 6 | RSVD | **LDCTL**<br>R←NSPSEG |
| 7 | RSVD | **LDCTL**<br>R←NSPOFF |
| 8 | **MSET** | RSVD |
| 9 | **MRES** | RSVD |
| A | **MBIT** | **LDCTL**<br>FCW←R |
| B | RSVD | **LDCTL**<br>RFRSH←R |
| C | ↓ | **LDCTL**<br>PSAPSEG<br>←R |
| D | **MREQ**<br>R | **LDCTL**<br>PSAPOFF<br>←R |
| E | RSVD | **LDCTL**<br>NSPSEG←R |
| F | RSVD | **LDCTL**<br>NSPOFF←R |

**Table 7.**
**Upper Instruction Byte**

# Topical Index

| Instruction Description | Mnemonic | Data Types | Addressing Modes | Flags Affected |
|---|---|---|---|---|
| **Arithmetic** | | | | |
| Add with Carry | ADC | B, W | R | C, Z, S, V, D[1], H[1] |
| Add | ADD | B, W, L | R, IM, IR, DA, X | C, Z, S, V, D[1], H[1] |
| Compare (Immediate) | CP | B, W | IR, DA, X | C, Z, S, V |
| Compare (Register) | CP | B, W, L | R, IM, IR, DA, X | C, Z, S, V |
| Decimal Adjust Bit | DAB | B | IR | C, Z, S |
| Decrement | DEC | B, W | R, IR, DA, X | Z, S, V |
| Divide | DIV | W, L | R, IM, IR, DA, X | C, Z, S, V |
| Extend Sign | EXTS | B, W, L | R | C, Z, S, V |
| Increment | INC | B, W | R, IR, DA, X | Z, S, V |
| Multiply | MULT | W, L | R, IM, IR, DA, X | C, Z, S, V |
| Negate | NEG | B, W | R, IR, DA, X | C, Z, S, V |
| Subtract with Carry | SBC | B, W | R | C, Z, S, V, D[1], H[1] |
| Subtract | SUB | B, W, L | R, IM, IR, DA, X | C, Z, S, V, D[1], H[1] |
| **Bit Manipulation** | | | | |
| Bit Test | BIT | B, W | R | Z |
| Bit Reset (Static) | RES | B, W | R, IR, DA, X | — |
| Bit Reset (Dynamic) | RES | B, W | R | — |
| Bit Set (Static) | SET | B, W | R, IR, DA, X | — |
| Bit Set (Dynamic) | SET | B, W | R | — |
| Bit Test and Set | TSET | B, W | R, IR, DA, X | S |
| **Block Transfer and String Manipulation** | | | | |
| Compare and Decrement | CPD | B, W | IR | C, Z, S, V |
| Compare, Decrement, and Repeat | CPDR | B, W | IR | C, Z, S, V |
| Compare and Increment | CPI | B, W | IR | C, Z, S, V |
| Compare, Increment, and Repeat | CPIR | B, W | IR | C, Z, S, V |
| Compare String and Decrement | CPSD | B, W | IR | C, Z, S, V |
| Compare String, Decrement, and Repeat | CPSDR | B, W | IR | C, Z, S, V |
| Compare String and Increment | CPSI | B, W | IR | C, Z, S, V |
| Compare String, Increment, and Repeat | CPSIR | B, W | IR | C, Z, S, V |
| Load and Decrement | LDD | B, W | IR | V |
| Load, Decrement, and Repeat | LDDR | B, W | IR | V |
| Load and Increment | LDI | B, W | IR | V |
| Load, Increment, and Repeat | LDIR | B, W | IR | V |
| Translate and Decrement | TRDB | B | IR | Z, V |
| Translate, Decrement, and Repeat | TRDRB | B | IR | Z, V |
| Translate and Increment | TRIB | B | IR | Z, V |
| Translate, Increment, and Repeat | TRIRB | B | IR | Z, V |
| Translate, Test, and Decrement | TRTDB | B | IR | Z, V |
| Translate, Test, Decrement, Repeat | TRTDRB | B | IR | Z, V |
| Translate, Test, and Increment | TRTIB | B | IR | Z, V |
| Translate, Test, Increment, and Repeat | TRTIRB | B | IR | Z, V |
| **CPU Control Instructions** | | | | |
| Complement Flag | COMFLG | — | — | C[2], Z[2], S[2], P[2], V[2] |
| Disable Interrupt | DI | — | — | — |
| Enable Interrupt | EI | — | — | — |
| Halt | HALT | — | — | — |
| Load Control Register (from register) | LDCTL | — | R | C[2], Z[2], S[2], P[2], D[2], H[2] |
| Load Control Register (to register) | LDCTL | — | — | — |
| Load Program Status | LDPS | — | IR, DA, X | C, Z, S, P, D, H |
| Multi-Bit Test | MBIT | — | — | S |
| Multi-Micro Request | MREQ | — | — | Z, S |
| Multi-Micro Reset | MRES | — | — | — |
| Multi-Micro Set | MSET | — | — | — |
| No Operation | NOP | — | — | — |
| Reset Flag | RESFLG | — | — | C[2], Z[2], S[2], P[2], V[2] |
| Set Flag | SETFLG | — | — | C[2], Z[2], S[2], P[2], V[2] |

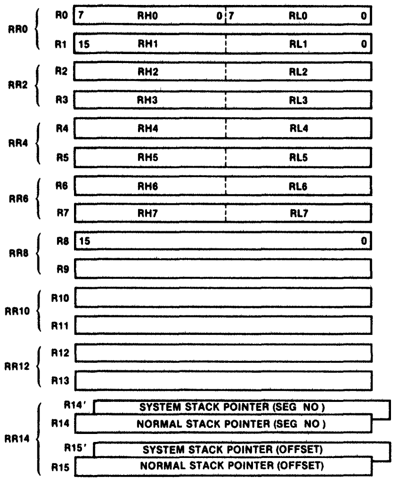
[1] Flag affected only for byte operation.

[2] Flag modified only if specified by the instruction.

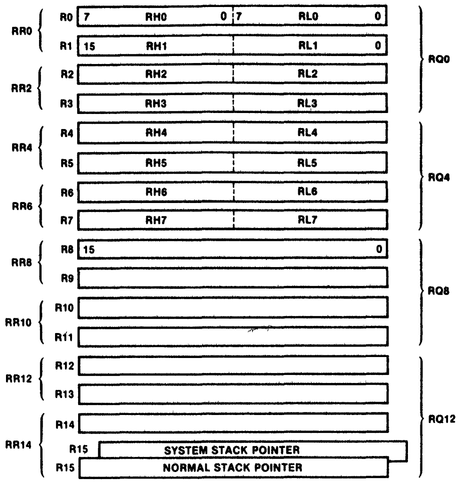
## Topical Index (Continued)

| Instruction Description | Mnemonic | Data Types | Addressing Modes | | Flags Affected |
|---|---|---|---|---|---|
| **Input/Output Instructions**[3] | | | **Regular** | **Special** | |
| Input | (S)IN[3] | B, W | IR, DA | (DA) | — |
| Input and Decrement | (S)IND[3] | B, W | IR | (IR) | V |
| Input, Decrement and Repeat | (S)INDR[3] | B, W | IR | (IR) | V |
| Input and Increment | (S)INI[3] | B, W | IR | (IR) | V |
| Input, Increment, and Repeat | (S)INIR[3] | B, W | IR | (IR) | V |
| Output | (S)OUT[3] | B, W | IR, DA | (DA) | — |
| Output and Decrement | (S)OUTD[3] | B, W | IR | (IR) | V |
| Output, Decrement, and Repeat | (S)OUTDR[3] | B, W | IR | (IR) | V |
| Output and Increment | (S)OUTI[3] | B, W | IR | (IR) | V |
| Output, Increment, and Repeat | (S)OUTIR[3] | B, W | IR | (IR) | V |
| **Logical Instructions** | | | | | |
| And | AND | B, W | R, IM, IR, DA, X | | Z, S, P |
| Complement | COM | B, W | R, IR, DA, X | | Z, S, P |
| Or | OR | B, W | R, IM, IR, DA, X | | Z, S, P |
| Test | TEST | B, W, L | R, IR, DA, X | | Z, S, P |
| Test Condition Code | TCC | B, W | R | | — |
| Exclusive Or | XOR | B, W | R, IM, IR, DA, X | | Z, S, P |
| **Program Control Instructions** | | | | | |
| Call Procedure | CALL | — | IR, DA, X | | — |
| Call Procedure Relative | CALR | — | RA | | — |
| Decrement, Jump if Not Zero | DJNZ | B, W | RA | | — |
| Interrupt Return | IRET | — | — | | C, Z, S, P, D, H |
| Jump | JP | — | IR, DA, X | | — |
| Jump Relative | JR | — | RA | | — |
| Return From Procedure | RET | — | — | | — |
| System Call | SC | — | — | | — |
| **Rotate and Shift Instructions** | | | | | |
| Rotate Left | RL | B, W | R | | — |
| Rotate Left Through Carry | RLC | B, W | R | | C, Z, S, V |
| Rotate Left Digit | RLDB | B | R | | Z, S |
| Rotate Right | RR | B, W | R | | C, Z, S, V |
| Rotate Right Through Carry | RRC | B, W | R | | C, Z, S, V |
| Rotate Right Digit | RRDB | B | R | | Z, S |
| Shift Dynamic Arithmetic | SDA | B, W, L | R | | C, Z, S, V |
| Shift Dynamic Logical | SDL | B, W, L | R | | C, Z, S, V |
| Shift Left Arithmetic | SLA | B, W, L | R | | C, Z, S, V |
| Shift Left Logical | SLL | B, W, L | R | | C, Z, S, V |
| Shift Right Arithmetic | SRA | B, W, L | R | | C, Z, S, V |
| Shift Right Logical | SRL | B, W, L | R | | C, Z, S, V |

[3] Each I/O instruction has a Special counterpart used to alert other devices that a Special I/O transaction is occurring. The Special I/O mnemonic is S + Regular mnemonic. Refer to section 6.2.8 for further details.



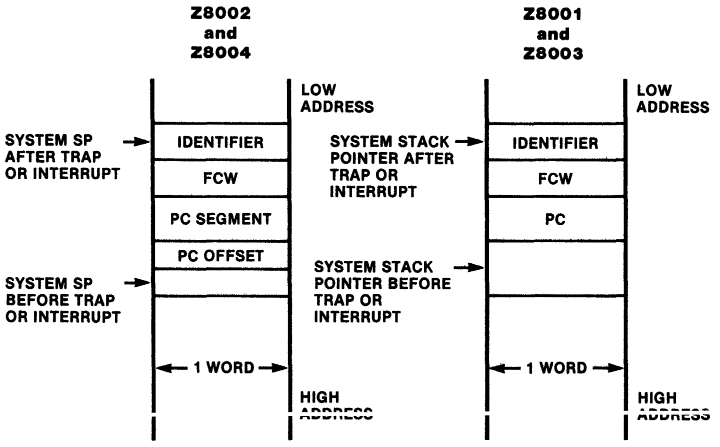
**Z8001 General Purpose Registers**



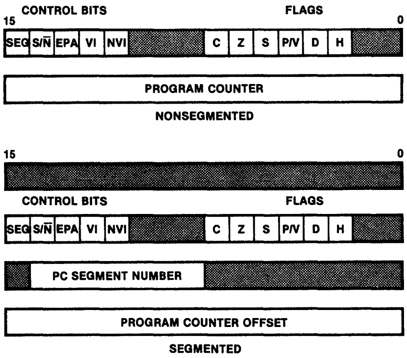
**Z8002 General Purpose Registers**

| Register | | | | Binary | Hex |
|---|---|---|---|---|---|
| RQ0 | RR0 | R0 | RH0 | 0000 | 0 |
| | | R1 | RH1 | 0001 | 1 |
| | RR2 | R2 | RH2 | 0010 | 2 |
| | | R3 | RH3 | 0011 | 3 |
| RQ4 | RR4 | R4 | RH4 | 0100 | 4 |
| | | R5 | RH5 | 0101 | 5 |
| | RR6 | R6 | RH6 | 0110 | 6 |
| | | R7 | RH7 | 0111 | 7 |
| RQ8 | RR8 | R8 | RL0 | 1000 | 8 |
| | | R9 | RL1 | 1001 | 9 |
| | RR10 | R10 | RL2 | 1010 | A |
| | | R11 | RL3 | 1011 | B |
| RQ12 | RR12 | R12 | RL4 | 1100 | C |
| | | R13 | RL5 | 1101 | D |
| | RR14 | R14 | RL6 | 1110 | E |
| | | R15 | RL7 | 1111 | F |

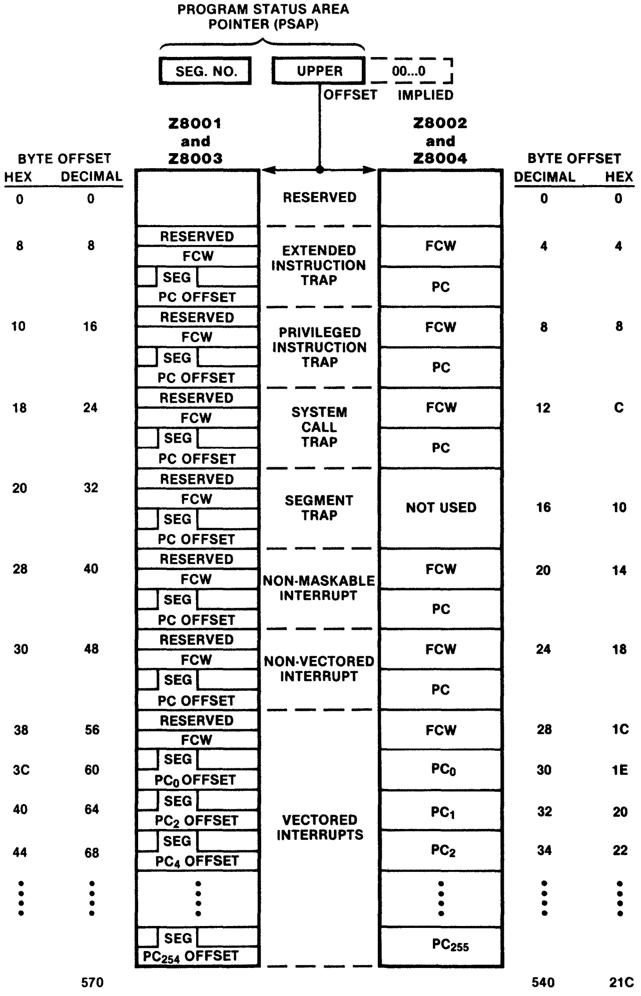
**Binary Encoding for Register Fields**



**Format of Saved Program Status in the System Stack**



**Program Status Blocks**



**Program Status Area**

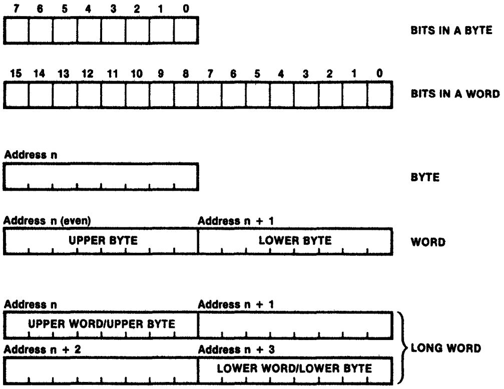
## Condition Codes

| Code | Meaning | Flag Setting | Binary |
|---|---|---|---|
| F | Always false* | - | 0000 |
| | 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 |
| PE | Parity even | P = 1 | 0100 |
| PO | Parity odd | P = 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 |
| UGE | Unsigned greater than or equal | C = 0 | 1111 |
| ULT | Unsigned less than | C = 1 | 0111 |
| UGT | Unsigned greater than | ((C =0) AND (Z = 0)) = 1 | 1011 |
| ULE | Unsigned less than or equal | (C OR Z) = 1 | 0011 |

This table provides the condition codes and the flag settings they represent.

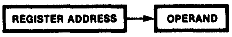
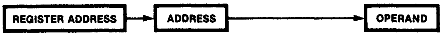
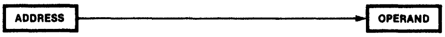
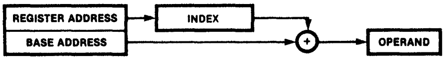
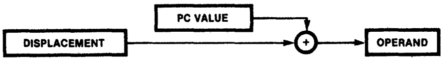
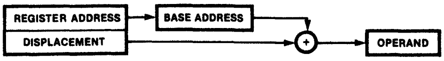
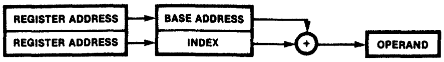
Note that some of the condition codes correspond to identical flag settings: i.e., Z-EQ, NZ-NE, NC-UGE, PE-OV, PO-NOV.

*Presently not implemented in PLZ/ASM Z8000 compiler.



**Addressable Data Elements**

## Z8000 Addressing Modes

| Addressing Mode | Operand Addressing: In the Instruction / In a Register / In Memory | Operand Value |
|---|---|---|
| **R**<br>Register |  | The content of the register |
| **IM**<br>Immediate | | In the instruction |
| ***IR**<br>Indirect Register |  | The content of the location whose address is in the register |
| **DA**<br>Direct Address |  | The content of the location whose address is in the instruction |
| ***X**<br>Index |  | The content of the location whose address is the address in the instruction plus the content of the working register. |
| **RA**<br>Relative Address |  | The content of the location whose address is the content of the program counter, offset by the displacement in the instruction |
| ***BA**<br>Base Address |  REGISTER ADDRESS<br>BASE ADDRESS<br>DISPLACEMENT<br>+<br>OPERAND | The content of the location whose address is the address in the register, offset by the displacement in the instruction |
| ***BX**<br>Base Index |  REGISTER ADDRESS<br>BASE ADDRESS<br>REGISTER ADDRESS<br>INDEX<br>+<br>OPERAND | The content of the location whose address is the address in a register plus the index value in another register. |

*Do not use R0 or RR0 as indirect, index, or base registers.

## Powers of 2 and 16

| $2^n$ | n |
|---|---|
| 256 | 8 |
| 512 | 9 |
| 1 024 | 10 |
| 2 048 | 11 |
| 4 096 | 12 |
| 8 192 | 13 |
| 16 384 | 14 |
| 32 768 | 15 |
| 65 536 | 16 |
| 131 072 | 17 |
| 262 144 | 18 |
| 524 288 | 19 |
| 1 048 576 | 20 |
| 2 097 152 | 21 |
| 4 194 304 | 22 |
| 8 388 608 | 23 |
| 16 777 216 | 24 |

**Powers of 2**

$2^0 = 16^0$

$2^4 = 16^1$

$2^8 = 16^2$

$2^{12} = 16^3$

$2^{16} = 16^4$

$2^{20} = 16^5$

$2^{24} = 16^6$

$2^{28} = 16^7$

$2^{32} = 16^8$

$2^{36} = 16^9$

$2^{40} = 16^{10}$

$2^{44} = 16^{11}$

$2^{48} = 16^{12}$

$2^{52} = 16^{13}$

$2^{56} = 16^{14}$

$2^{60} = 16^{15}$

| $16^n$ | n |
|---|---|
| 1 | 0 |
| 16 | 1 |
| 256 | 2 |
| 4 096 | 3 |
| 65 536 | 4 |
| 1 048 576 | 5 |
| 16 777 216 | 6 |
| 268 435 456 | 7 |
| 4 294 967 296 | 8 |
| 68 719 476 736 | 9 |
| 1 099 511 627 776 | 10 |
| 17 592 186 044 416 | 11 |
| 281 474 976 710 656 | 12 |
| 4 503 599 627 370 496 | 13 |
| 72 057 594 037 927 936 | 14 |
| 1 152 921 504 606 846 976 | 15 |

**Powers of 16**

| 8 | | 7 | | 6 | | 5 | | 4 | | 3 | | 2 | | 1 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Hex** | **Decimal** | **Hex** | **Decimal** | **Hex** | **Decimal** | **Hex** | **Decimal** | **Hex** | **Decimal** | **Hex** | **Decimal** | **Hex** | **Decimal** | **Hex** | **Decimal** |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 268,435,456 | 1 | 16,777,216 | 1 | 1,048,576 | 1 | 65,536 | 1 | 4,096 | 1 | 256 | 1 | 16 | 1 | 1 |
| 2 | 536,870,912 | 2 | 33,554,432 | 2 | 2,097,152 | 2 | 131,072 | 2 | 8,192 | 2 | 512 | 2 | 32 | 2 | 2 |
| 3 | 805,306,368 | 3 | 50,331,648 | 3 | 3,145,728 | 3 | 196,608 | 3 | 12,288 | 3 | 768 | 3 | 48 | 3 | 3 |
| 4 | 1,073,741,824 | 4 | 67,108,864 | 4 | 4,194,304 | 4 | 262,144 | 4 | 16,384 | 4 | 1,024 | 4 | 64 | 4 | 4 |
| 5 | 1,342,177,280 | 5 | 83,886,080 | 5 | 5,242,880 | 5 | 327,680 | 5 | 20,480 | 5 | 1,280 | 5 | 80 | 5 | 5 |
| 6 | 1,610,612,736 | 6 | 100,663,296 | 6 | 6,291,456 | 6 | 393,216 | 6 | 24,576 | 6 | 1,536 | 6 | 96 | 6 | 6 |
| 7 | 1,879,048,192 | 7 | 117,440,512 | 7 | 7,340,032 | 7 | 458,752 | 7 | 28,672 | 7 | 1,792 | 7 | 112 | 7 | 7 |
| 8 | 2,147,483,648 | 8 | 134,217,728 | 8 | 8,388,608 | 8 | 524,288 | 8 | 32,768 | 8 | 2,048 | 8 | 128 | 8 | 8 |
| 9 | 2,415,919,104 | 9 | 150,994,944 | 9 | 9,437,184 | 9 | 589,824 | 9 | 36,864 | 9 | 2,304 | 9 | 144 | 9 | 9 |
| A | 2,684,354,560 | A | 167,772,160 | A | 10,485,760 | A | 655,360 | A | 40,960 | A | 2,560 | A | 160 | A | 10 |
| B | 2,952,790,016 | B | 184,549,376 | B | 11,534,336 | B | 720,896 | B | 45,056 | B | 2,816 | B | 176 | B | 11 |
| C | 3,221,225,472 | C | 201,326,592 | C | 12,582,912 | C | 786,432 | C | 49,152 | C | 3,072 | C | 192 | C | 12 |
| D | 3,489,660,928 | D | 218,103,808 | D | 13,631,488 | D | 851,968 | D | 53,248 | D | 3,328 | D | 208 | D | 13 |
| E | 3,758,096,384 | E | 234,881,024 | E | 14,680,064 | E | 917,504 | E | 57,344 | E | 3,584 | E | 224 | E | 14 |
| F | 4,026,531,840 | F | 251,658,240 | F | 15,728,640 | F | 983,040 | F | 61,440 | F | 3,840 | F | 240 | F | 15 |
| **8** | | **7** | | **6** | | **5** | | **4** | | **3** | | **2** | | **1** | |

**Hexadecimal and Decimal Interger Conversion Table**

### To Convert Hexadecimal to Decimal

1. Locate the column of decimal numbers corresponding to the left-most digit or letter of the hexadecimal: select from this column and record the number that corresponds to the position of the hexadecimal digit or letter.
2. Repeat step 1 for the units (second from the left) position.
3. Repeat step 1 for the units (third from the left) position.
4. Add the numbers selected from the table to form the decimal number.

To convert integer numbers greater than the capacity of the table, use the techniques below:

### Hexadecimal to Decimal

Succesive cumulative multiplication from left to right, adding units position.

***Example:*** $D34_{16} = 3380_{10}$

```
D =   13
     ×16
     208
3 =  +13
     211
     ×16
    3376
4 =   +4
    3380
```

**Example:**

| **Conversion of Hexadecimal Value** | **D34** |
|---|---|
| 1. D | 3328 |
| 2. 3 | 48 |
| 3. 4 | 6 |
| 4. Decimal | 3380 |

### To Convert Decimal to Hexadecimal

1. (a) Select from the tabel the highest decimal number that is equal to or less than the number to be converted.
   (b) Record the hexadecimal of the column containing the selected number.
   (c) Subtract the selected decimal from the number to be converted.
2. Using the remainder from step 1(c) repeat all of step 1 to develop the second position of the hexadecimal (and a remainder).
3. Using the remainder from step 2 repeat all of step 1 to develop the units position of the hexadecimal.
4. Combine terms to form the hexadecimal number.

### Decimal to Hexadecimal

Divide and collect the remainder in reverse order.

***Example:*** $3380_{10} = D34_{16}$

```
16|3380  remainder
16| 211 ↘4   ↑
16|  13 ↘3   |
        ↘D   |
```

**Example:**

| **Conversion of Decimal Value** | **3380** |
|---|---|
| 1. D | −3328 |
| | 52 |
| 2. 3 | −48 |
| | 4 |
| 3. 4 | −4 |
| 4. Hexadecimal | D34 |

## ASCII Characters

| Hexadecimal | Character | Meaning | Hexadecimal | Character | |
|---|---|---|---|---|---|
| 00 | NUL | NULL Character | 40 | @ | |
| 01 | SOH | Start of Heading | 41 | A | |
| 02 | STX | Start of Text | 42 | B | |
| 03 | ETX | End of Text | 43 | C | |
| 04 | EOT | End of Transmission | 44 | D | |
| 05 | ENQ | Enquiry | 45 | E | |
| 06 | ACK | Acknowledge | 46 | F | |
| 07 | BEL | Bell | 47 | G | |
| 08 | BS | Backspace | 48 | H | |
| 09 | HT | Horizontal Tabulation | 49 | I | |
| 0A | LF | Line Feed | 4A | J | |
| 0B | VT | Vertical Tabulation | 4B | K | |
| 0C | FF | Form Feed | 4C | L | |
| 0D | CR | Carriage Return | 4D | M | |
| 0E | SO | Shift Out | 4E | N | |
| 0F | SI | Shift In | 4F | O | |
| 10 | DLE | Data Link Escape | 50 | P | |
| 11 | DC1 | Device Control 1 | 51 | Q | |
| 12 | DC2 | Device Control 2 | 52 | R | |
| 13 | DC3 | Device Control 3 | 53 | S | |
| 14 | DC4 | Device Control 4 | 54 | T | |
| 15 | NAK | Negative Acknowledge | 55 | U | |
| 16 | SYN | Synchronous Idle | 56 | V | |
| 17 | ETB | End of Transmission Block | 57 | W | |
| 18 | CAN | Cancel | 58 | X | |
| 19 | EM | End of Medium | 59 | Y | |
| 1A | SUB | Substitute | 5A | Z | |
| 1B | ESC | Escape | 5B | [ | |
| 1C | FS | File Separator | 5C | \ | |
| 1D | GS | Group Separator | 5D | ] | |
| 1E | RS | Record Separator | 5E | ^ | |
| 1F | US | Unit Separator | 5F | _ | |
| 20 | SP | Space | 60 | ` | |
| 21 | ! | | 61 | a | |
| 22 | " | | 62 | b | |
| 23 | # | | 63 | c | |
| 24 | $ | | 64 | d | |
| 25 | % | | 65 | e | |
| 26 | & | | 66 | f | |
| 27 | ' | | 67 | g | |
| 28 | ( | | 68 | h | |
| 29 | ) | | 69 | i | |
| 2A | * | | 6A | j | |
| 2B | + | | 6B | k | |
| 2C | , | | 6C | l | |
| 2D | – | | 6D | m | |
| 2E | . | | 6E | n | |
| 2F | / | | 6F | o | |
| 30 | 0 | | 70 | p | |
| 31 | 1 | | 71 | q | |
| 32 | 2 | | 72 | r | |
| 33 | 3 | | 73 | s | |
| 34 | 4 | | 74 | t | |
| 35 | 5 | | 75 | u | |
| 36 | 6 | | 76 | v | |
| 37 | 7 | | 77 | w | |
| 38 | 8 | | 78 | x | |
| 39 | 9 | | 79 | y | |
| 3A | : | | 7A | z | |
| 3B | ; | | 7B | { | |
| 3C | < | | 7C | ¦ | |
| 3D | = | | 7D | } | |
| 3E | > | | 7E | ~ | |
| 3F | ? | | 7F | DEL | Delete |

D

Zilog

# Appendix D

## INTRODUCTION

This Appendix presents the software algorithms required to restart an aborted Z8003 or Z8004 instruction. It was assumed in the preparation of this document that the reader would be familar with the following: 1) the Z8000 assembly language, 2) operating systems, particularly memory management, 3) the Z8010 Memory Management Unit (MMU), and 4) the Z8015 Paged Memory Management Unit (PMMU).

Restarting most instructions only requires setting the program counter to point to the first word of the aborted instruction. Some instructions, however, are aborted after they have modified some CPU registers but before their execution is completed. For example if a "Compare and Increment" instruction is aborted during execution because the memory number or word to be used for the comparison is not in main memory, then the register used as a counter will have been decremented. Thus, before restarting the instruction, the counter register must be incremented.

When an instruction is aborted, the CPU saves the values contained by the Flag and Control Word (FCW) and the Program Counter (PC) of the aborted instruction on the system stack together with information read from the MMUs during the address/trap acknowledge sequence. The fault handler routine is automatically invoked to process the address translation trap. This routine saves a copy of the aborted program's registers so that another program can be executed while the aborted program waits for data or instructions to be loaded into the main memory.

In the following discussion, the terms "PC" and "register" refer to copies of the aborted program's PC and registers, which typically reside in main memory during the suspension of the aborted program's execution.

The steps for restarting an instruction are as follows:

1. Determine which MMU or PMMU caused the CPU instruction abort.

2. If the MMU or PMMU that caused the abort was managing stack memory and the abort was the result of a Page Write Warning (PWW) then, exit routine; otherwise, continue with the next step.

3. Determine whether or not the hardware was in the segmented mode.

4. Read the address of the aborted instruction from the appropriate MMU (or PMMU) and external hardware and update the PC with it. The address's segment number and high byte are obtained from the MMU (or PMMU); the low byte is obtained from external hardware.

5. Read the violation address from the appropriate MMU (or PMMU) and perform any action required to establish the validity of the requested address (e.g., bring in the page(s) from secondary memory, or mark an already resident page as valid).

6. Read the Bus Status register from the appropriate MMU (or PMMU). If the status indicates that a trap had ocurred during an instruction fetch cycle then exit the routine; otherwise, continue with the next step.

7. Using the updated PC, examine the instruction to see if any CPU registers have been modified. (Suggestion: use the upper byte of the instruction and a 256-bit table which identifies potential cases.) If the instruction did not modify any registers, then exit the routine. Some instructions aborted in a paged virtual memory system require that the number of successful data reads and writes

performed during the executed portion of the instruction cycle be saved for restart. This information is normally read from a hardware data transfer counter set up to count the number of successful data transfers performed since the completion of the last instruction fetch cycle. In the paged version of the MMU, however, this information is automatically collected and stored.

8. Call the register fix-up routine, and exit the current routine.

## FIX-UP ROUTINE

The fix-up routine examines the aborted instruction and modifies the register file if necessary. The number of instructions that can generate memory traps depends upon whether a segmented or paged virtual memory system is implemented. In a paged system, data can cross page boundaries; this operation, however adds complexity to the register fix-up routine as well as increasing the number of instructions that might have modified registers before being aborted.

When a program is run in System mode, several assumptions regarding the operating system are made (no assumptions are made about programs run in Normal mode):

- The fault handler will not generate a fault until all critical data has been saved.
- The system stack always resides in main memory, thus, accessing the system stack never causes a fault.
- I/O buffers always reside in primary memory; thus, an I/O instruction never causes a fault.
- The Program Status Area always resides in main memory.

The reasons behind these assumptions are as follows: If the system stack is not located in main memory, the "saved" PC and FCW data pushed in response to an interrupt or trap acknowledge is lost unless captured by external hardware.

If the Program Status Area (PSA) is not located in main memory, the occurrence of any trap or interrupt causes an address trap to be generated when the new program status is fetched. The new address trap forces the CPU to jump to whatever memory address was present on the bus when the MMU stopped generating trap requests (that is, the address of the "fetched" program status).

The location of input/output buffers outside of main memory would result (except for extremely low speed devices) in transfer overruns or underruns. Such operations would cause data read from devices to be lost upon the detection of a memory fault.

Tables 1 and 2 list all the instructions that may require modification to the registers before they can be restarted. Instructions not listed do not require additional action other than the correction of the Program Counter. The lists presented in Tables 1 and 2 are based on the Z8003/4 implementation of the Z8000 instruction set. Only those actions given in these tables are to be performed before restarting the corresponding instruction. All actions listed must be performed even if the specifications of the instruction involved indicate that registers will be modified during its execution.

Only those registers indicated in Tables 1 and 2 should be "corrected" in the case of an abort.

**Table 1. Instructions That May Have Modified CPU Registers When Aborted in a Segmented Virtual Memory.**

| INSTRUCTION | DESCRIPTION |
|---|---|
| LDI(R): | If bus cycle status indicates that a read was attempted to an absent segment ($R/\overline{W}$ bit=1), increment the Counter register by one.<br>If a write was attempted to an absent segment ($R/\overline{W}$ bit=0), increment the Counter register by one and decrement the <u>Destination</u> Pointer register by one if byte move or by two if word move. |
| CPI(R), TR(T)I(R):<br>CPD(R), TR(T)D(R): | Increment the Counter register by one. |
| CPSI(R): | Increment the Counter register by one. Compare the Source Pointer address with the violation address. If they are equal, then no further action is required; otherwise decrement the <u>Destination</u> Pointer register by one if comparing bytes compare or by two if comparing words. |
| LDD(R), CPSD(R): | Same as the increment versions, but the Destination Pointer must be incremented. |
| CALL(R):<br>CALL:<br>CALR: | Increment R15 (the offset field of the Stack Pointer) by two if in a nonsegmented mode or by four if in a segmented mode. |
| POP: | If $R/\overline{W}$ bit of bus cycle status=0 (write attempted to an absent segment), decrement Stack Pointer by 2 and restart instruction. |
| POPL: | Same as POP but decrement by 4. |

**Table 2. Instructions That May Have Modified CPU Registers When Aborted in a Paged Virtual Memory System.**

| INSTRUCTION | DESCRIPTION |
|---|---|
| LDL from memory: | If the Destination register pair was used in the address calculation <u>and</u> the Data Read/Write counter in the PMMU indicates that one read was successfully completed (i.e., the second half of the long word being loaded caused the page fault), then the even register of the pair was modified and the register may require correction before restarting the instruction; otherwise no action is required.<br>In segmented mode:<br>If the addressing mode was Indirect Register or Base, store the violation address segment number in the even register of the destination pair.<br>If the addressing mode was Index and the even register of the destination pair was used as the index register, subtract the base address offset in the instruction from the violation address offset, store the result in the index register, and decrement that register by two. |

Table 2. (Continued)

| INSTRUCTION | DESCRIPTION |
|---|---|
| | If the addressing mode was Base Index then:<br>If the Destination pair was used as the Base Address pair, store the violation address segment number in the even register of the destination pair.<br>If the even register of the destinaton pair was used as the index register, subtract the Base Address offset from the violation address offset, store the result in the Index register, and decrement that register by two.<br>In nonsegmented mode:<br>If the addressing mode was Indirect register and the even register of the destination pair was used as the Indirect register, then decrement the violation address by two and store the result in the even register of the destination pair.<br>If the addressing mode was Base or Index and the even register of the destination pair was used as the base or index register, subtract the address component in the instruction from the violation address, store the result in the even register of the destination pair, and decrement that register by two.<br>If the addressing mode was Base Index with one of the address registers used as the even register of the destination pair, subtract the other address register from the violation address, store the result in the even register of the destination pair and decrement that register by two.<br>If the addressing mode was Base Index and the even register was used as both the Base and Index register, decrement the violation address by two, store the result in the even register of the register pair, and shift that register one position to the right (divide by two). |
| PUSHL from memory: | If bus status indicates that a write was aborted (i.e., the bus status is not $C_{16}$ or $D_{16}$) and the data Read/Write counter indicates completion of three data transactions, increment the Stack Pointer by four; otherwise, no action is required. |
| LDI(R), CPI(R),<br>CPSI(R), TR(T)I(R):<br>LDD(R), CPD(R),<br>CPSD(R), TR(T)D(R): | Same as given in Table 1 for segmented virtual memory instruction CALL (R), CALL and CALR. |
| LDPS: | If the Data Read/Write counter indicates that no read was successfully completed, no action is required.<br>If one read was successfully completed, then if the saved FCW equals the first word of the PS (i.e., the CPU was in nonsegmented mode), clear the Segmentation mode bit and set the System mode bit in the saved FCW.<br>If two reads were successfully completed (i.e., the CPU was in Segmented mode), set both the Segmentation and System mode bits in the saved FCW. |

Table 2. (Continued)

| INSTRUCTION | DESCRIPTION |
|---|---|
| RET: | If the Data Read/Write counter indicates that no reads were successfully completed, and if in Nonsegmented mode, decrement R15 (the offset field of the Stack Pointer) by two; otherwise, no action is required.<br>If one read was successful, decrement R15 by four. |
| POP: | If one read was successful, decrement stack pointer by two. |
| POPL: | If two reads were successful, decrement stack pointer by four. |
| LDM from memory: | If bus status and the Data Read/Write counter indicate that n reads were successfully completed, then a register used in the address calculation may have been modified. If this is the case, the register needs to be corrected in the manner described below; otherwise, no action is required.<br>If the Indirect Register addressing mode was used in Segmented mode and the indirect register pair has been modified, subtract 2(n + 1) from the violation address offset and store the segment number and computed offset in the register pair. In Nonsegmented mode, subtract 2(n + 1) from the violation address and store the result in the Indirect register.<br>If the Index addressing mode was used and the index register has been modified, subtract the offset in the instruction and 2(n + 1) from the violation offset, and store the result in the Index register. |

## ALGORITHM FOR SEGMENTED VIRTUAL MEMORY REGISTER FIX-UP

**Definitions:**

| | | |
|---|---|---|
| RW | = | Read/Write counter in PMMU or R/W bit in MMU bus cycle status register |
| SEG | = | Segmented/Nonsegmented mode, SEG = 1 --> Segmented mode of operation |
| VADDR | = | violation address (two words if segmented, one word if nonsegmented) |
| Rs, RRs | = | source register |
| Rs1 | = | source register Rs + 1 (for example, if RR8 = S RRs then Rs1 is R9) |
| Rd, RRd | = | destination register |
| Rd1 | = | destination register Rd + 1 |
| Rc | = | count register |
| LN | = | lowest nibble of first word of an instruction |

```
Upper Byte of Op Code    Fix-Up
                  B8:    (Translate)
                         Rc <-- Rc + 1;
       D0-DF, 1F, 5F:    (Call, Call Relative)
                         IF SEG = 1 THEN R15 <-- R15 + 4
                                    ELSE R15 <-- R15 + 2
                  BA:    (Load or Compare Byte String)
                         Rc <-- Rc + 1;
                         CASE LN
                 LDI:      1:  IF RW = 0   THEN IF SEG = 1 THEN Rd1 <-- Rd1 - 1
                           (i.e. IF WRITE)                ELSE Rd <-- Rd - 1
         CPSI, CPSIR:    2,6:  IF SEG = 1  THEN IF RRs ≠ VADDR THEN Rd1 <-- Rd1 - 1
                                           ELSE IF Rs ≠ VADDR THEN Rd <-- Rd - 1
                 LDD:      9:  IF RW = 0   THEN IF SEG = 1 THEN Rd1 <-- Rd1 + 1
                           (i.e. IF WRITE)                ELSE Rd <-- Rd + 1
         CPSD, CPSDR:    A,E:  IF SEG = 1  THEN IF RRs ≠ VADDR THEN Rd1 <-- Rd1 + 1
                                           ELSE IF Rs ≠ VADDR THEN Rd <-- Rd + 1
                  BB:    (Load or Compare Word String)
                         Rc <-- Rc + 1;
                         CASE LN
                           1:  IF RW = 0   THEN IF SEG = 1 THEN Rd1 <-- Rd1 - 2
                           (i.e. IF WRITE)                ELSE Rd <-- Rd - 2
                         2,6:  IF SEG = 1  THEN IF RRs ≠ VADDR THEN Rd1 <-- Rd1 - 2
                                           ELSE IF Rs  ≠ VADDR THEN Rd <-- Rd - 2
                           9:  IF RW = 0   THEN IS SEG = 1 THEN Rd1 <-- Rd1 + 2
                           (i.e. IF WRITE)                ELSE Rd <-- Rd + 2
                         A,E:  IF SEG = 1  THEN IF RRs ≠ VADDR THEN Rd1 <-- Rd1 + 2
                                           ELSE IF Rs  ≠ VADDR THEN Rd <-- Rd + 2
```

## ADDITIONAL CASES FOR PAGED VIRTUAL MEMORIES

**Additional Definitions:**

RW = Read/Write Counter in PMMU

FCW = saved FCW of aborted program

PSW = first word of data fetched during LDPS instruction (i.e., if RW = 1 then PSW = contents of memory location VADDR - 2)

VSEG = violation segment number

VOFF = violation offset

IOFF = offset in address in the aborted instruction

Rx = index register

```
Upper
Byte                Fix-Up

11,51:  (Push Long from memory)
        IF RW = 3 THEN IF SEG = 1 THEN Rd1 <-- Rd1 + 4
                                  ELSE Rd <-- Rd + 4

14,35:  (Load Long from memory--Indirect, Base, using RRd or Rd as the address register)
        IF RW = 1 THEN IF SEG = 1 THEN Rd <-- VSEG
                                  ELSE Rd <-- VOFF - 2

   54:  (Load Long--Index using Rd as the index register)
        IF RW = 1 THEN IF Rs  = 0 THEN Rd <-- VOFF - IOFF - 2

   75:  (Load Long--Base Index)
        IF RW = 1 THEN IF SEG = 1 THEN IF RRd is the address register
                                  THEN Rd <-- VSEG
                                  ELSE IF Rd = Rx THEN Rd <-- VSEG
                                                  ELSE Rd <-- VOFF - Rs - 2
                                  ELSE IF Rs ≠ Rx THEN Rd <-- (VOFF - 2) / 2
                                  ELSE IF Rd = Rs THEN Rd <-- VOFF - Rx - 2
                                                  ELSE Rd <-- VOFF - Rs - 2
   39:  (Load Program Status)
        IF RW = 1 THEN IF FCW = PSW THEN FCW <-- 4000
                  ELSE IF RW = 2 THEN FCW <-- C000

   9E:  (Return)
        IF RW = 0 AND SEG = 0 THEN R15 <-- R15 - 2
                              ELSE IF RW = 1 THEN R15 <-- R15 - 4

   1C:  (Load Multiple--Indirect)
        IF SEG = 1 THEN Rs <-- VSEG; Rs1 <-- VOFF - 2(n + 1)
                   ELSE Rs <-- VOFF - 2(n + 1)

   5C:  (Load Multiple--Index)
        IF Rx = 0 THEN Rs <-- VOFF - IOFF - 2(n + 1)
```

Zilog

# Glossary

**abort:** The interruption of an instruction execution cycle before its completion. Abort interrupts occur in Z8000 virtual memory systems when the executing instruction references information not in main memory.

**address:** An entity that specifies one particular element in a set of similar elements. May be either a *memory address* or an *I/O address* (q.q.v). (See also *segmented address, logical address, physical address.*)

**address space:** A set of addresses. The Z8000 can access eight separate address spaces: normal-mode program memory space, system-mode program memory space, normal-mode data memory space, system-mode data memory space, normal-mode stack memory space, system-mode stack memory space, standard I/O space, and special I/O space. (See *normal mode, system mode, program memory address space, data memory address space, stack memory address space, standard I/O address space, and special I/O address space.*)

**addressing mode:** The way in which the address of an *operand* (q.v.) is specified. There are eight addressing modes: *Register, Immediate, Indirect Register, Direct Address, Index, Base Address, Relative Address, Base Index* (q.q.v).

**autodecrement:** The contents of a register are decremented and then used as specified by the instruction.

**autoincrement:** The contents of a register are used as specified by the instruction and then incremented.

**Base Address (BA) addressing mode:** A base address consists of a register that contains the base and a 16-bit *displacement* (q.v.). The displacement is added to the base and the resulting address indicates the *effective address* (q.v.). In nonsegmented mode, the base address is held in a *word register* (q.v.) and the displacement is in the instruction. In segmented mode, the segmented base address is held in a register pair and the displacement is in the instruction.

**Base Index (BX) addressing mode:** Base Index addressing is similar to Base addressing except that the displacement ("index"), as well as the base, is held in a register. In nonsegmented mode, the base address is held in a word register and the index is held in a word register. In segmented mode, the segmented base address is held in a *register pair* (q.v.) and the index is held in a word register.

**BCD digit:** A Binary Coded Decimal digit is an encoding of the ten decimal digits into a 4-bit code that is simply the first ten binary numbers in the binary number system (starting with 0). This code is used to represent and process numbers in the base-10 (decimal) format.

**bus:** A group of signal lines, which connects the devices in a system.

**Bus-Disconnect state:** The CPU state during which the CPU is not the bus master and may not initiate *transactions* (q.v.) on the bus.

**bus master:** The device in control of the bus. Must be a device that is able to initiate transactions.

**bus request:** A request for control of the bus.

**byte:** A byte is eight contiguous bits; a byte in memory starts on an addressable byte boundary.

**byte register:** An 8-bit register. The Z8000 CPU contains 16 general-purpose byte registers, designated RLn and RHn (n = 0-7).

**clock cycle:** One cycle of the CPU clock, beginning with a rising edge.

**condition:** An event detected by the hardware and indicated by setting the appropriate flag. A condition is caused by the execution of an instruction and is always reproducible. The Z8000 has six flags to record these events, called *status flags* (q.v.).

**context switching:** Interrupting the activity in progress and switching to another activity. A context switch involves saving for later restoration the contents of the general-purpose registers, the *Program Counter* and the *Flag and Status Word* (q.v.).

**CPU state:** Either *Running state, Stop/Refresh state,* or *Bus-Disconnect state* (q.q.v.).

**data memory address space:** A *memory address space* (q.v.) that is identified by the status codes 1000 or 1010.

**data structure:** A logical organization of primitive elements (e.g. byte or word) whose format and access conventions are well-defined. Examples of data structures are tables, lists and arrays.

**data type:** The way in which bits are grouped and interpreted. For an instruction, the data type of an operand determines its size and the significance of its bits. Operand data types include byte, word, long word, byte string, word string, and BCD digit.

**Direct Address (DA) addressing mode:** In this mode, the operand address is contained within the instruction.

**displacement:** A number contained in the instruction for use in calculating the *effective address* (q.v.) of an operand. The displacement is added to the contents of a register during the calculation.

**DMA:** Direct Memory Access is a method for transferring data to or from main memory at high speed by avoiding the CPU registers.

**effective address:** The address obtained after indirect or indexing modification. In nonsegmented mode, the effective address is a 16-bit number. In segmented mode, the effective address consists of a 7-bit segment number and 16-bit offset. In systems with memory management, the effective address is the logical address which must be translated to obtain the physical memory address.

**flags:** Bits in the *Flag and Control Word* (q.v.) that indicate *conditions* (q.v.).

**Flag and Control Word (FCW):** One of the two Program Status registers; it contains *flags* (q.v.) and bits that control the operation of the CPU.

**Immediate (IM) addressing mode:** In this mode, the operand is contained within the instuction.

**Index (X) addressing mode:** In this mode, the operand address is obtained by adding the contents of an index register (q.v.) to a base address contained in the instruction.

**index register:** A word register used to contain a displacement for use in effective address calculation.

**Indirect Register (IR) addressing mode:** In this mode, the operand address is contained within a register.

**instruction fetch:** An access to *program memory address space* (q.v.).

**interrupt request:** An event other than a trap or jump or call instruction that changes the normal flow of instruction execution. (See *non-maskable, non-vectored,* and *vectored* interrupts.)

**interrupt service routine:** The routine executed in response to an interrupt.

**interrupt/trap acknowledge transaction:** The transaction initiated by the CPU in response to an interrupt or trap. Obtains an identifier word from the interrupting device or memory management hardware.

**I/O address:** The address of an I/O port, always 16 bits long. Word ports may have even or odd addresses, Special I/O byte ports are even, Standard I/O byte ports are odd.

**I/O transaction:** A transaction that transfers data to or from a peripheral device or memory management hardware.

**logical address:** The address manipulated by the programmer, used by instructions and output by the Z8000.

**long word:** A long word is 32 contiguous bits; a long word in memory starts on an even addressable byte boundary.

**machine cycle:** One basic CPU operation, starting with a bus *transaction* (q.v.).

**memory address:** An address specifying a location in memory. Word and long-word addresses must be even, byte addresses may be even or odd.

**memory management:** The process of translating *logical addresses* into *physical addresses* (q.q.v.), plus certain protection functions.

**memory transactions:** A transaction that transfers data to or from main memory.

**Normal mode:** A *Running-state* (q.v.) mode in which the $S/\overline{N}$ flag in the FCW is 0 and the $N/\overline{S}$ line is High. In this mode, the CPU may not execute *privileged instructions* (q.v.).

**non-maskable interrupts:** *Interrupts* (q.v.) which cannot be disabled.

**nonsegmented mode:** A Running-state mode of the Z8000 CPUs. For segmented CPUs in this mode, all addresses are generated with the same *segment number* (q.v.).

**non-vectored interrupts:** *Interrupts* (q.v.) which do not use the identifier word as a vector to an *interrupt service routine* (q.v.).

**offset:** In a Z8001 CPU, the 16-bit value that appears on the AD lines when an address is generated.

**operand:** An item of data operated on by an instruction.

**physical address:** The address required for accessing the memory, obtained from the logical address generated by the Z8000 by memory management hardware, for example, the Z8010 Memory Management Unit.

**privileged instruction:** An instruction intended for use primarily by an operating system, which can be executed only in System mode. In general, instructions that change the processor state or perform I/O are privileged.

**Program Counter (PC):** One of the two *Program Status registers* (q.v.). Contains the address of the current instruction.

**program memory address space:** The *memory address space* (q.v.) indicated by the status codes (1100 or 1101).

**Program Status Area:** The area in memory reserved for the starting program status of the interrupt and trap service routines.

**Program Status Area Pointer (PSAP):** The register that contains the starting address of the Program Status Area.

**Program Status registers:** The two registers (PC and FCW) that contain the program status.

**Refresh counter:** A register that controls the Z8000 dynamic memory, periodic-refresh mechanism. Used to set the refresh rate and to enable the mechanism.

**Refresh cycle:** A type of transaction used to refresh dynamic memory. It is three clock cycles long.

**Refresh/Stop state:** A CPU state entered whenever the $\overline{\text{STOP}}$ line is asserted. A continuous stream of *refresh cycles* (q.v.) is generated.

**register:** A storage location in hardware logic other than the memory. Bits within a register are numbered from 0, with the least significant being the rightmost. See also byte register, word register, register pair, and register quad.

**Register (R) addressing mode:** In this mode, the operand is in a general-purpose register.

**register pair:** One of eight pairs of general-purpose word registers, designated RRn (n = 0, 2, 4, ...12, 14).

**register quad:** One of four groups of four word registers, designated RQn (n = 0, 4, 8, 12).

**Relative Address (RA) addressing mode:** In this mode, the operand address is calculated by adding a displacement found in the instruction to the current PC value.

**request:** Either an *interrupt request, bus request, resource request*, or *$\overline{\text{STOP}}$ request* (qq.v). An external device requests that the CPU perform some action.

**reset:** An internal CPU operation that initializes the Program Status registers. It is activated by the $\overline{\text{RESET}}$ line.

**Running state:** One of the three CPU states. In this state, the CPU is fetching and executing instructions or handling interrupts.

**segment:** In a Z8001, a set of adjacent memory addresses (up to 64K) with the same *segment number* (q.v.) on lines $SN_0$-$SN_6$.

**segment number:** A number specifying a memory *segment* (q.v.). Placed on the $SN_0$-$SN_6$ lines during memory transactions in Z8001 system. Part of a *segmented address* (q.v.).

**segmented address:** In segmented Z8000 CPU's, a 23-bit value consisting of a 7-bit *segment number* (q.v.) and a 16-bit *offset* (q.v.).

**segmented mode:** One of the Running-state modes of the segmented Z8000 CPU. In this mode, CPU generates addresses that can have different segment numbers.

**Special I/O address space:** An *I/O address space* (q.v.) that is identified by the status code 0011. Used to access memory management hardware.

**stack:** A data structure used for temporary storage or for procedure and interrupt service routine linkages. A stack uses the last-in, first-out concept. As items are added to, or pushed onto, the stack, the stack pointer decrements; as items are removed from, or popped off, the stack, the stack pointer increments.

**stack memory address space:** A *memory address space* (q.v.) that is identified by the status codes 1001 and 1011.

**Stack Pointer:** A general-purpose register indicating the top (lowest address) of a stack.

**Standard I/O address space:** *An I/O address space* (q.v.) that is identified by the status code 0010. Used for accessing peripherals.

**status code:** A 4-bit encoding of the CPU's current transaction, for example, internal operation, segment trap acknowledge, or stack memory request.

**status flags:** Status flags are set according to the outcome of certain instructions to direct the subsequent flow of the program as necessary. There are six status flags: Carry, Zero, Sign, Parity/Overflow, Decimal Adjust and Half Carry. The first four are grouped together to determine the condition code, the last two are used in programs manipulating BCD digits.

**status lines:** The lines $ST_0$-$ST_3$, which contain the status code during transactions.

**stop request:** A request that is made by activating the $\overline{\text{STOP}}$ line.

**Stop/Refresh state:** See Refresh/Stop state.

**System mode:** A Running-state mode (q.v.) in which the S/$\overline{\text{N}}$ flag in the FCW is 1 and the N/$\overline{\text{S}}$ line is Low. In this mode, the CPU may exercise *privileged instructions* (q.v.).

**transaction:** One of the basic bus operations. A transaction lasts three or more clock cycles and covers a single data movement on the bus.

**trap:** A condition that occurs at the end of an instruction that caused an illegal operation. The Z8000 traps are internal traps arising from system call, EPA instruction and privileged instructions executed in normal mode, and an external trap, the segmentation/address trap, arising from memory access violations in systems with memory management. A trap is similar to an interrupt in that it causes the executing program to be interrupted and the Program Status registers to be saved on the system stack. Traps cannot be disabled.

**vectored interrupts:** *Interrupts* (q.v.) which use the identifier word as a vector to the *interrupt service routine* (q.v.). May be disabled.

**virtual memory:** A memory management technique in which the system's logical memory address space is not necessarily the same as, and can be much larger than, the available physical memory address space. Virtual memory is supported by use of memory mapping hardware and secondary storage devices.

**$\overline{\text{WAIT}}$ cycle:** A clock cycle during which the $\overline{\text{WAIT}}$ line is active. Used to prolong transactions, since no signal line is sampled while $\overline{\text{WAIT}}$ is active.

**word:** Two contiguous bytes (16 bits) starting on an even addressable byte boundary. Bits are numbered from the right, 0 through 15. A word is identified by the address of the byte containing the most significant bit, bit 15.

**word register:** A 16-bit register.

I

Zilog

# Index

# Reader's Comments

**Your feedback about this document helps us ascertain your needs and fulfill them in the future. Please take the time to fill out this questionnaire and return it to us. This information will be helpful to us and, in time, to future users of Zilog products.**

Title of this document:

Your Name:

Company Name:

Address:

Briefly describe application:

**Does this publication meet your needs?** □ Yes □ No  If no, why?

**How are you using this publication?**

- □ As an introduction to the subject?
- □ As a reference?
- □ As an instructor or student?

**How do you find the material?**

| | Excellent | Good | Poor |
|---|---|---|---|
| Technicality | □ | □ | □ |
| Organization | □ | □ | □ |
| Completeness | □ | □ | □ |

**Rated on a scale of 1 to 10, this document is a** ________.

**What would have improved the material?**

**Other comments and suggestions:**

**If you found any mistakes in this document, please let us know what and where they are:**

Please clip, fold, and return to Zilog, Inc.

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

**BUSINESS REPLY MAIL**

FIRST CLASS PERMIT NO. 35, CAMPBELL, CA.

POSTAGE WILL BE PAID BY ADDRESSEE

**Zilog**

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

## Zilog Sales Offices

### 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

00-2010-06

Printed in USA