Zilog

Z8000 CPU Technical Manual

March 1981

Z8000 CPU
Technical Manual

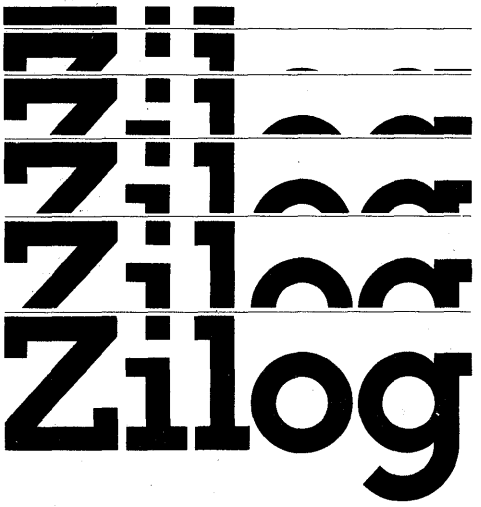
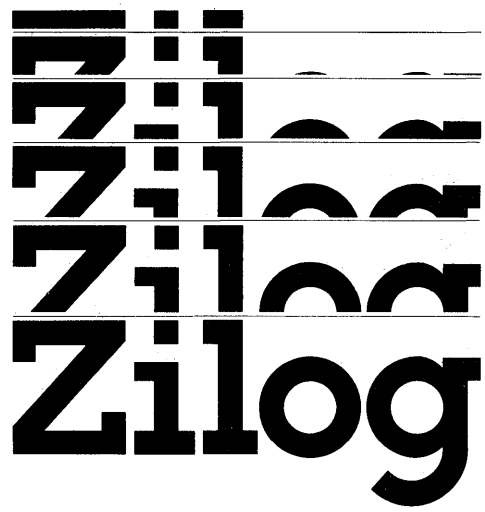


Table of Contents

Table of Contents (Continued)

Table of Contents (Continued)


Zilog

Chapter 1
Z8000 Processor Overview

1.1 Introduction

This chapter provides a summary description of the advanced architecture of the Z8000 Microprocessor, with special attention given to those architectural features that set the Z8000 CPU apart from its 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 microprocessor has been designed to accommodate a wide range of applications, from the relatively simple to the large and complex. The Z8000 CPU is offered in two versions: the Z8001 and the Z8002. Each CPU comes with an entire family of support components: a memory management unit, 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 provides in an LSI microprocessor design the flexibility and sophisticated features usually associated with mini- or mainframe computers.

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, sophisticated interrupt structure, a rich set of data types, separate I/O address spaces and, for the Z8001, a large address space and segmented memory addressing. Each of these features is treated in detail in the next section.

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

The CPU has been designed so that a powerful memory management system can be used to improve the utilization of the main memory and provide protection capabilities for the system. This is discussed in Section 1.3.12. Although memory management is an optional capability—the Z8000 CPU is an extremely sophisticated processor without memory management—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 very general mechanism for extending the basic instruction set through the use of extern al devices (called Extended Processing Units—EPUs). In general, an EPU is dedicated to performing complex and time-consuming tasks so as to unburden the CPU. Typical tasks for specialized EPUs include floating-point arithmetic, data base search and maintenance operations, network interfaces, and many others. This topic is treated in Section 1.5.

1.3 Architectural Features

The architectural resources of the Z8000 CPU include sixteen 16-bit general-purpose registers, seven data types ranging from bits to 32-bit long words 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 rich set of 414 instructions. Moreover, the 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 CPU generates status signals indicating the nature of the bus transaction that is being attempted; these can be used to implement sophisticated systems with multiple address spaces—memory areas dedicated to specific uses. The CPU also has two operating modes, system and normal, which can be used to separate operating system functions from normal application processes. I/O operations have have been separated from memory accesses, further enchancing the capability and integrity of Z8000-based systems, and a sophisticated interrupt structure facilitates the efficient operation of peripheral I/O devices. Moreover, the Extended Processing Unit (EPU) capability of the Z8000 allows the CPU to unload many time-consuming tasks onto external devices.

Special features of the Z8000 have been introduced to facilitate the implementation of multiple processor systems. In addition, the Z8001 CPU has a large, segmented addressing capability that greatly extends the applicability of microprocessors to large system applications.

1.3 Architectural Features (Continued)

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 instruction structure.

General-purpose registers can be used as accumulators, memory pointers or index registers. Their major advantage is that the particular use to which they are put can vary during the course of a program as the needs of the program change. Thus, the general-purpose register file avoids the critical bottlenecks of an implied or dedicated register architecture, which must save and restore the contents of dedicated registers 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 one 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, it is not necessary (for example) for a Z8000 user to dedicate a 32-bit register to hold a byte of data. Registers can be used efficiently in the Z8000.

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 operations available on a particular data type is frequently more important than additional, esoteric instructions, which 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 $A \leftarrow A + B$). 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. Also included in the instruction set are compact, one-word instructions for the most frequently used operations, such as branching short distances in a program.

The instruction set contains some 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 instructions 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 to specify the bit position within a byte or word as well as to specify the position statically in the instruction. The Rotate and Shift instructions are considerably more flexible than those in previous microprocessors. The String instructions are useful in 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 which 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 an external device (the Memory Management Unit, MMU) available with the Z8001.

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 are implied by specific instructions, including autoincrement. The first five modes listed

1.3 Architectural Features (Continued)

above are basic addressing modes that are used most frequently and apply to most instructions having more than one addressing mode. (In the Z8002, Base Address and Index modes are identical, and in the Z8001, Base Addressing capabilites can be simulated with all instructions, using Based Addressing or the Memory Management Unit and the Direct or Indexed Addressing mode.)

1.3.5 Multiple Memory Address Spaces. The Z8000 CPU facilitates the use of multiple address spaces. When the Z8000 CPU generates an address, it also outputs signals indicating the particular internal activity which 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 accesses (for example, by allowing only instruction fetches from an area designated to contain proprietary software). The Memory Management Unit (MMU) has been designed to provide precisely these kinds of protection features by using the CPU-generated status information.

1.3.6 System/Normal Mode of Operation. The Z8000 CPU can run in either system mode or normal mode. In system mode, all of the instructions can be executed and all of the CPU registers can be accessed. This mode is intended for use by programs performing operating system functions. In normal mode, some instructions may not 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 which deal with time dependent or system-interface events.

Programs executing in normal mode which have errors can always reproduce those errors for debugging purposes simply by re-executing the program with its original data. Programs using facilities available only in system mode may have errors due to timing considerations (e.g. based upon the frequency of disk requests and disk arm-position) that are harder to debug because these errors are not easily reproduced. Thus, the 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 further 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 insure that the normal stack is free of system information, the information saved on the occurrence of interrupts or traps is always pushed on to 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 and thus requires specific I/O instructions. This architectural separation allows better protection and has more potential for extension. The use of separate I/O spaces also conserves the limited Z8002 data memory space. There are in fact two separate I/O address spaces: standard I/O and special I/O. The main advantage of these two spaces is to provide for two types of peripheral support chips—standard I/O peripherals and special I/O peripherals—devices such as the Z8010 Memory Management Unit that do not respond to standard I/O commands, but do respond to special I/O commands. A second advantage of these two spaces is that they allow 8-bit peripherals to attach to the low-order eight bits (standard I/O) or to the high-order eight bits (special I/O) of the processor Address/Data bus.

The increased speed requirements of future microprocessors are likely to be achieved by tailoring memory and I/O references to their respective, characteristic reference patterns and by using simultaneous I/O and memory referencing. These future possibilities require an architectural separation today. Memory-mapped I/O is still possible, but loss of protection and lack of expandability are severe problems.

1.3.8 Interrupt Structure. The sophisticated 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 the throughput of the system. The CPU supports three types of interrupts. A non-maskable interrupt represents a catastrophic event which requires immediate handling to preserve system integrity. In addition, there are two types of maskable interrupts: non-vectored interrupts and vectored interrupts. The latter provides an automatic call to separate interrupt processing routines for each peripheral, depending on the vector presented by the peripheral to the Z8000.

1.3 Architectural Features (Continued)

The Z8000 has implemented a priority system for handling interrupts. Vectored interrupts have higher priority than non-vectored interrupts. 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 switch. This allows recursive task switches to occur while leaving the normal stack undisturbed by system information. The program state to handle the interrupt (new program status) is 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 architectural compatibility if new interrupts or traps are added to the architecture.

1.3.9 Multi-Processing. The increase in microprocessor computing power that the Z8000 represents makes simple the design of distributed processing systems having many low-cost microprocessors running dedicated processes.

The Z8000 provides some 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. These mechanisms, and new peripherals such as the Z-FIO, have been designed to allow easy asynchronous communication between different CPUs.

1.3.10 Large Address Space for the Z8001. For many applications, a basic address space of 64K bytes is insufficient. A large address space increases the range of applications of a system by permitting large, complex programs and data sets to reside in memory rather than be partitioned and swapped into a small memory as needed. A large address space greatly 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 version of the Z8000 generates 23-bit addresses, for a basic address space of 8 megabytes (8M or 8,388,608 bytes).

1.3.11 Segmented Addressing of the Z8001. The segmented version of the Z8000 CPU divides its 23-bit addresses into a 7-bit segment number and a 16-bit segment offset. The segment number serves as a 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 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 access to a large number of segments) is necessary if all the advantages of segmentation are to be realized. A system capable of directly accessing only, say, four segments would lack the needed flexibility and would be constrained by address space limitations.

1.3.12 Memory Management. Memory management consists primarily of dynamic relocation, protection, and sharing of memory. It offers the following advantages: providing a logical structure to the memory space that is independent of the actual physical location of data, protecting the user from inadvertent mistakes such as attempting to execute data, 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 physical addresses required for accessing the memory. This address transformation process is called relocation, which 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 CPU supports 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 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.

1.4 Benefits of the Architecture

The features of the Z8000 Architecture combine to provide several significant benefits: improvements in code density, compiler efficiency, operating system support, and support for high level data structures.

1.4.1 Code Density. Code density affects both processor speed and memory utilization. Code compaction 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. The most frequently used instructions are encoded in single-word formats. Fewer instructions are needed to accomplish a given task and a consistent and regular architecture further reduces the number of instructions required.

Code density is achieved in part by the use of special "short" formats for certain instructions which are shown by statistical analysis to be most frequently used by assemblers. A "short offset" mechanism has also been provided to allow a 2-word segmented address to be reduced to a single word; this format may be used by assemblers and compilers.

The largest reduction in program size and increase in speed results from the consistent and regular structure of the architecture and from the more powerful instruction set—factors that substantially reduce the number of instructions required for a task. The architecture is more regular relative to preceding microprocessors because its registers, address modes, and data types can be used in a more orderly fashion. Any general-purpose register except R0 can be specified as an acculumator, 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 registers dedicated to a specific purpose. This 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 adapt 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 unrelated 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 parameters and local variables on the procedure stack is supported by the "Index With Short Offset" addressing mode, as well as the Base Address and Base Index addressing modes. In addition, address arithmetic is aided by the Increment and Decrement instructions.

Testing of data, logical evaluation, initialization, and comparison 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 dramatic speed improvements over software simulations of these important tasks. In addition, any register except R0 can be used as a stack pointer by the Push and Pop instructions.

1.4.3 Operating System Support. Interrupt and task-switching features are included to improve operating system implementations. The memory-management and compiler-support features are also quite important.

The interrupt structure has three levels: nonmaskable, non-vectored, and vectored. When an interrupt occurs, the program status is saved on the stack with an indication of the reason for this state-switching before a new program status is loaded from a special area of memory. The program status consists of the flag register, the control bits, and the program counter. The reason for the occurrence is encoded in a vector that is read from the system bus and saved on the stack. In the case of a vectored interrupt, the vector also determines a jump table address that points to the interrupt processing routine.

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

Traps result in the same type of program status-saving as interrupts: in both cases, the information saved is pushed on to a system stack that keeps the normal stack undisturbed. The Load Multiple instruction allows the contents of registers to be saved efficiently in memory or on the stack. Running programs can cause program status changes under direct software control with the Load Program Status instruction.

Finally, exclusion and serialization can be achieved with the "atomic" Test And Set instruction that synchronizes asynchronous cooperating processes.

1.4 Benefit of the Architecture (Continued)

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 well-defined. Common data structures include arrays, lists, stacks, and strings. Since data structures are high-level constructs frequently used in programming, processor performance is significantly enchanced if the CPU provides mechanisms for efficiently manipulating them. The Z8000 offers such mechanisms.

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 mode 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, which 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 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 system and normal operating environments discussed earlier.

Byte and word strings are supported by the Translate and Translate And Test instructions. Decimal strings use the Decimal Adjust instruction to do decimal arithmetic on strings of BCD data, packed two characters per byte. The Rotate Digit instructions also manipulate 4-bit data.

1.4.5 Two CPU Versions: Z8001 and Z8002. The Z8000 CPU is offered in two versions: the Z8001 48-pin segmented CPU and the Z8002 40-pin nonsegmented CPU. The main difference between the two is addressing range. The Z8001 can directly address 8M bytes of memory; the Z8002 directly addresses 64K bytes. The Z8001 has a non-segmented mode of operation which permits it to execute programs written for the Z8002.

Not all applications require the large address space of the Z8001; for these applications the Z8002 is recommended. Moreover, many multiple-processor systems can be implemented with one Z8001 and several Z8002s, instead of exclusively using Z8001s. Since the same assembler generates code for both CPUs, users can buy only the power they require without having to worry about software incompatibility between processors.

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 have been set aside to implement this feature. When the CPU encounters an instruction with 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 which can 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, graphics support operations—a complete list would include most areas of computing.

1.6 Summary

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

Zilog

Chapter 2
Architecture

2.1 Introduction

This chapter provides an overview of the Z8000 CPU architecture. The basic hardware, operating modes and instruction set are all described. Differences between the two versions of the Z8000 (the nonsegmented Z8002 and the segmented Z8001) are noted where appropriate. Most of the subjects covered here are also treated with greater detail in later chapters of the manual.

2.2 General Organization

Figure 2.1 contains a block diagram that shows the major elements of the Z8000 CPU, namely:

- 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.
- A set of 16 general-purpose registers, which is used to contain addresses and data.
- Four special-purpose registers, which control the CPU operation.
- An Arithmetic and Logic Unit, which is used for manipulating data and generating addresses.
- An instruction execution control, which fetches and executes Z8000 instructions.
- An exception-handling control, which processes interrupts and traps.
- A refresh control, which generates memory refresh cycles.

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

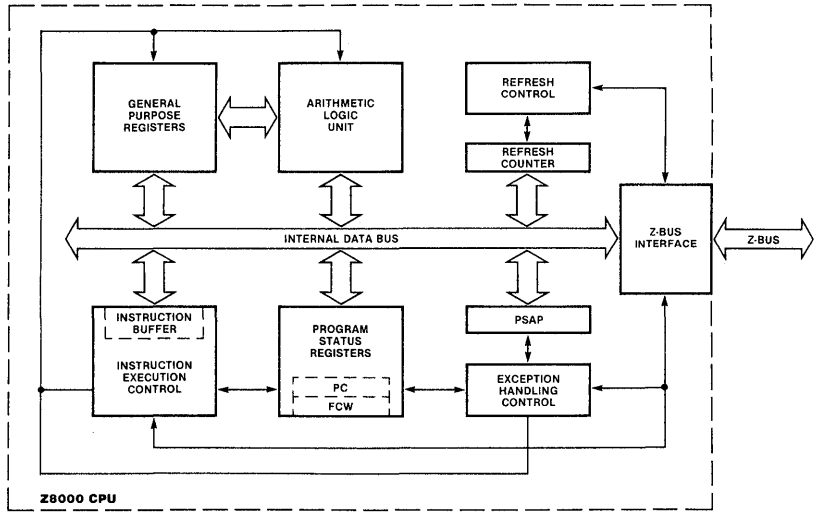


Figure 2-1. Z8000 CPU Functional Block Diagram

2.2 General Organization (Continued)

Figure 2.2 gives a system-level view of the Z8000. It is important to realize that the Z8000 CPU comes with a whole family of support components. The Z8000 Family has 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 shared bus that links the components of the system.
- A Z8001 or Z8002 CPU.
- One or more Extended Processing Units (EPUs), which are dedicated to performing specialized, time-consuming tasks.
- A memory sub-system, which in Z8001 systems can include one or more Memory Management Units (MMUs) that offer sophisticated memory allocation and protection features.
- One or more Direct Memory Access (DMA) controllers for high-speed data transfers.
- A large number of possible peripheral devices interfaced to the Z-Bus through Universal Peripheral Controllers (UPCs), Serial Communication Controllers (SCCs), Counter-Timer and Parallel I/O Controllers (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 in a distributed processing system.

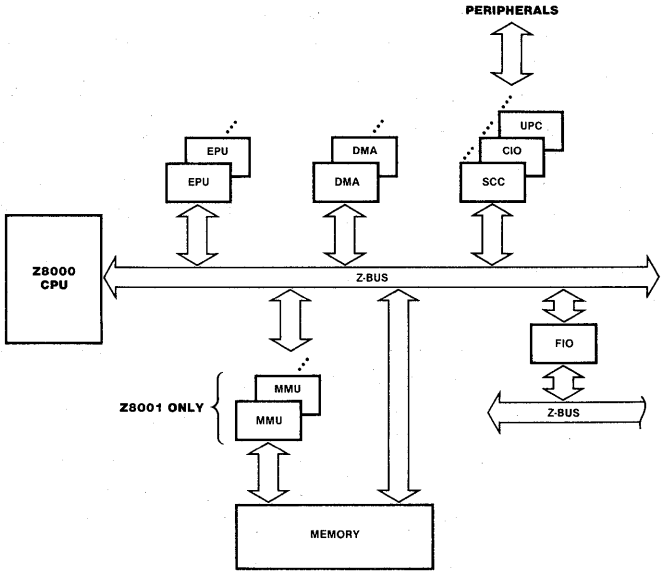


Figure 2-2. Z8000 System Configuration

2.3 Hardware Interface

Figure 2.3 shows the Z8000 pins grouped according to function. The Z8001 is packaged in a 48-pin DIP and the Z8002 is packaged in a 40-pin DIP. The eight additional pins on the Z8001 are the seven segment-number lines and the segment trap. Except for those eight, all pins on the two CPU versions are identical.

The Z8000 is a Z-Bus CPU; thus, activity on the pins is governed by the Z-Bus protocols (see The *Z-Bus Summary*). These protocols specify 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 and 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 only). These seven lines encode the addresses of up to 128 relocatable memory segments. The segment signals become valid before the address offset signals, thus supporting address relocation by the memory managment system.

2.3.3 Bus Timing. These three lines include Address Strobe ($\overline{AS}$), Data Strobe ($\overline{DS}$) and Memory Request ($\overline{MREQ}$). They 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 control signals to a memory system.

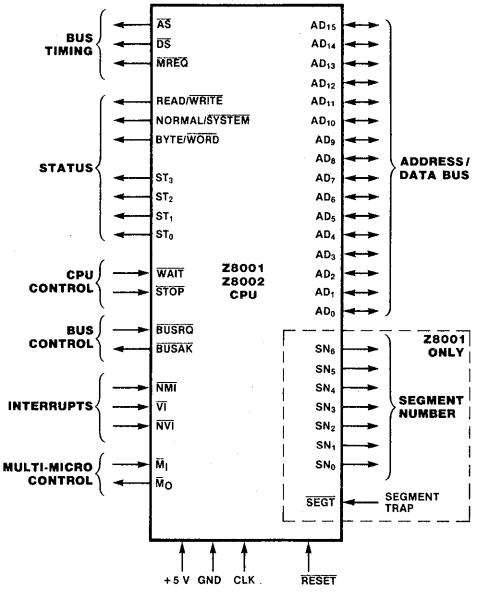


Figure 2-3. Z8000 Pin Functions

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	Non-maskable interrupt acknowledge
0 1 1 0	Non-vectored 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	EPA Transfer
1 1 1 1	Reserved

Table 2.1 Status Line Codes

2.3.4 Status. These lines function to indicate the kind of transaction on the bus (ST_0-ST_3), whether it is a read or write (R/$\overline{W}$, where High is Read and Low is Write), whether it is on byte or word data (B/$\overline{W}$, High = byte, Low = word), and whether the CPU is operating in normal mode or system mode (N/$\overline{S}$, High = normal, Low = system). The ST_0-ST_3 lines also encode additional characteristics of the bus transactions, as Table 2.1 shows. The availability of status information defining the type of bus transaction in advance of data transmission allows bidirectional drivers and other external hardware elements to be enabled before data is transferred.

2.3 Hardware Interface (Continued)

2.3.5 CPU Control. These inputs allow external devices to delay the operation of the CPU. The $\overline{WAIT}$ line, when active (Low), causes the CPU to idle in the middle of a bus transaction, taking extra clock cycles until the $\overline{WAIT}$ line goes inactive; it is typically used by memory or I/O peripherals which operate more slowly than the CPU. The Stop ($\overline{STOP}$) line halts internal CPU operation when the first word of an instruction (or the second word of an EPA instruction) has been fetched. This signal is useful for single-step instruction execution during debugging operations and for enabling Extended Processing Units to halt the CPU temporarily.

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 system bus, i.e., the signal lines that are common to several devices in a system. The external device requesting control of the bus inputs a bus request ($\overline{BUSREQ}$); the CPU responds with a bus acknowledge ($\overline{BUSACK}$) after three-starting, or electrically neutralizing, the Address/Data Bus, Bus Timing lines, Status lines, and Control lines. The Z-Bus allows a daisy chain to be used to enforce a priority among several external devices.

2.3.7 Interrupts. Three interrupt inputs are provided: non-maskable interrupts ($\overline{NMI}$), vectored interrupts ($\overline{VI}$) and non-vectored interrupts ($\overline{NVI}$). These permit external devices to suspend the CPU's execution of its current program and begin executing an interrupt service routine.

2.3.8 Segment Trap Request (Z8001 only). This input to the CPU is used by an external memory-management system to indicate that an illegal memory access has been attempted.

2.3.9 Multi-Micro Control. The Multi-Micro In ($\overline{MI}$) and Multi-Micro Out ($\overline{MO}$) lines are used in conjunction with instructions such as MSET and MREQ 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 3 include +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 the Z8000: 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 $\overline{AS}$ and ending with a rising edge of $\overline{DS}$. A *machine cycle* covers one basic CPU 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. Appendix A contains a complete description of Z8000 timing.

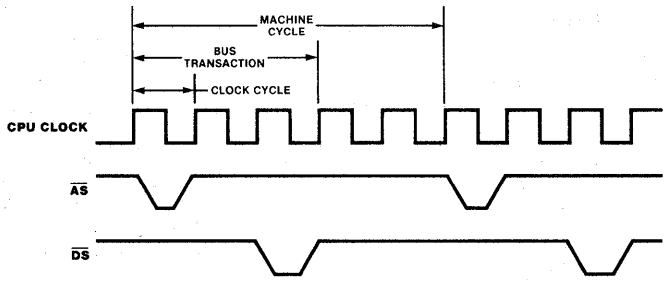


Figure 2-4. Basic Timing Periods

2.5 Address Spaces

The Z8000 supports two main address spaces corresponding to the two different kinds of locations 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.

For more information on address spaces, consult Chapter 3.

2.5.1 Memory Address Space. Memory address space can be further subdivided into Program Memory address space, Data Memory address space, and Stack Memory address space, each for both normal and system modes.

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, stack memory reference, and program memory

2.5 Address Spaces (Continued)

reference each correspond to a different status code at the ST_0–ST_3 outputs, allowing three address spaces to be distinguished for each of two operating modes, giving six address spaces in all. Each of the six address spaces has a range as great as the addressing ability of the processor. For the nonsegmented Z8002, each address space can have up to 64K bytes, giving a potential total system capacity of 384K bytes of directly addressable memory. The segmented Z8001, on the other hand, provides up to 48M bytes of directly addressable memory due to the 23-bit segmented addresses.

Segmentation is a means of partitioning memory into variable-size segments so that a variety of useful functions may be implemented, including:

- Protection mechanisms that prevent a user from referencing data belonging to others, attempting to modify read-only data or overflowing a stack.
- Virtual memory, which permits a user to write functioning programs under the assumption that the system contains more memory than is actually available.
- Dynamic relocating which allows the placement of blocks of data in physical memory independently of user addresses, allowing better management of the memory resources and sharing of data and programs.

The signals provided on the segmented Z8001 CPU assist in implementing these features, although additional software and external circuitry (such as the Z8010 MMU) are generally required to take full advantage of them. Chapter 3 contains an extensive discussion of segmentation and the Z8001.

2.5.2 I/O Address Space. I/O addresses are represented as 16-bit words for both the Z8001 and Z8002.

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.

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 MMU. 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 bytes to quadruple words. The word registers are specified in assembly-language statements as R0 through R15. Sixteen byte registers, RH0-RL7, which may be used as accumulators, overlap the first eight word registers. Register grouping for larger operands includes eight double-word (32-bit) registers, RR0-RR14, and four quad-word registers, RQ0-RQ12, which are used by a few instructions such as Multiply, Divide, and Extend Sign.

As Figure 2.5 illustrates, the CPU has two hardware stack pointers, one dedicated to each of the two basic operating modes, system and

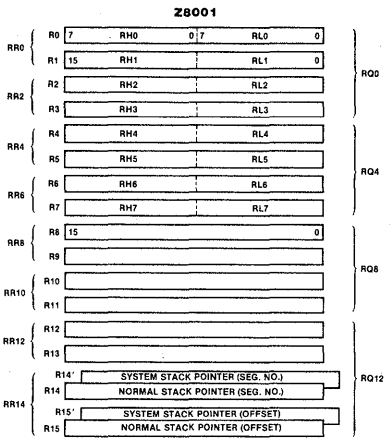


Figure 2-5a. Z8001 General-Purpose Registers (Register Address Space)

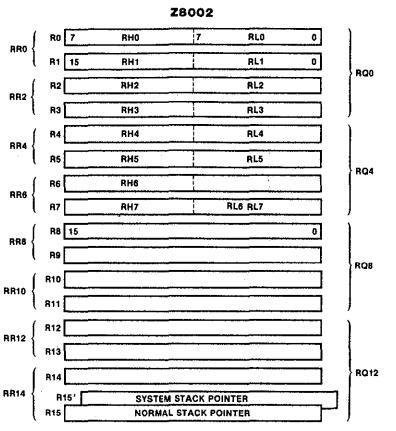


Figure 2-5b. Z8002 General-Purpose Registers (Registers Address Space)

2.6 General-Purpose Registers (Continued)

normal. The segmented Z8001 uses a two-word stack pointer for each mode (R14'/R15' or R14/R15), whereas the nonsegmented Z8002 uses 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 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, the normal stack pointer can be directly accessed as a special control register. The normal mode stack pointer can be assessed 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 versions in Figure 2.6. Each register can be manipulated by 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 Z8002, the Program Status registers consist of two words: one each for the FCW and the PC. In the segmented Z8001, there are 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 even 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 binary result of a previous decimal addition or subtraction into the correct decimal (BCD) result.

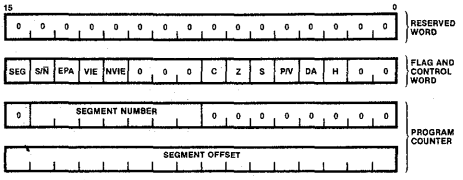
Section 6.3 provides more detail on these flags.

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

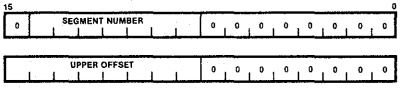
Non-Vectored Interrupt Enable (NVIE), Vectored Interrupt Enable (VIE). These bits determine whether or not the CPU will accept non-vectored or vectored interrupts (see Section 2.13).

System/Normal Mode (S/$\overline{N}$). 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/$\overline{S}$ pin) is the complement of this bit.

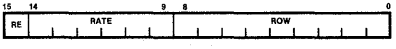
Extended Processor Architecture (EPA) Mode. When this bit is set to one, it indicates that the system contains Extended Processing Units, and hence extended instructions



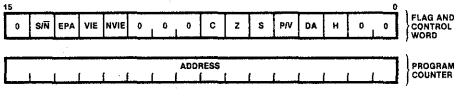
Z8001 Program Status Registers



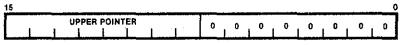
Z8001 Program Status Area Pointer



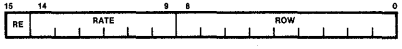
Z8001 Refresh Counter



Z8002 Program Status Registers



Z8002 Program Status Area Pointer



Z8002 Refresh Counter

Figure 2-6. CPU Special Registers

2.7 Special Purpose Registers (Continued)

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; it is always cleared in the nonsegmented Z8002. When set to one, the CPU is operating in segmented mode, and when 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 progam status values (FCWs and PCs) in main memory called the Program Status Area. New Program Status register values are fetched from this area when an interrupt or trap occurs. As shown in Figure 2.6, the PSAP comprises either one word (nonsegmented Z8002) or two words (segmented Z8001); for either configuration, the lower byte of the pointer must be zero. Refer to Chapter 7 for more details about the Program Status Area and its layout.

2.7.3 Refresh Counter. The CPU contains a programmable counter that can be used to refresh dynamic memory automatically. The refresh counter register consists of a 9-bit row counter, a 6-bit rate counter and an enable bit (Figure 2.6). Refer to Chapter 8 for details of the refresh mechanism.

2.8 Instruction Execution

In the normal course of events, the Z8000 CPU will spend 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 occasionally enters.

Stop/Refresh State. This is really one state, although it may be entered in two different ways: 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 Counter to generate refreshes. For more information, consult Chapter 8.

Bus-Disconnect State. This is the state the CPU enters when the DMA, or some other bus requester, takes over the bus. Program execution is suspended and the CPU disconnects itself from the bus.

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 which perform I/O) can be executed; in normal mode they cannot. This dichotomy allows the creation of operating system software, which controls CPU resources and is protected from application program action.

In addition, the CPU will be in either segmented or nonsegmented mode. In segmented mode, which is available only on the Z8001, the program uses 23-bit segmented addresses for memory accesses; in nonsegmented mode, which is available on both CPUs, the program uses 16-bit nonsegmented addresses for memory accesses.

While executing instructions, the mode of the CPU is controlled by bits in the FCW (Section 2.8). While handling interrupts, the CPU is always in system mode and, for the Z8001, in segmented mode.

2.9 Instructions

The Z8000 instruction 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 multiply 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 Instructions (Continued)

2.9.1 Instruction Formats. Formats of the instructions are shown in Figure 2.7. The two most significant bits in the instruction word determine whether the compact instruction format (A) or the general instruction format (B) is used. Compact formats encode the four most frequently used instructions into single words, thereby saving on instruction-memory usage and increasing execution speed. As long as the two most significant bits are not 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.

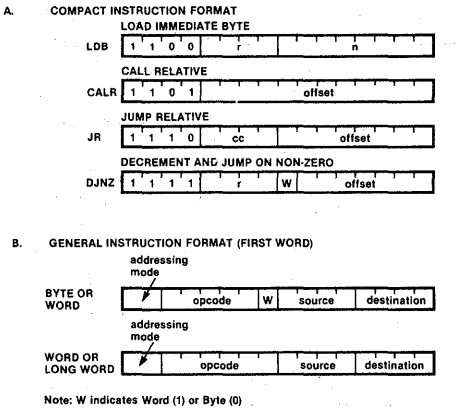


Figure 2-7. Instruction Formats

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 a number of instructions which operate upon it directly. These data types are:

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

Bits can be manipulated in registers or memory. Binary and decimal integers and logical values can be manipulated in registers only, although operands can be fetched directly from memory. Addresses are manipulated only in registers, and strings and stacks are 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 location of the data elements. Locations are designated using one of the following eight addressing modes:

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

Immediate Mode. The data element is located in the instruction.

Indirect Register Mode. The data element can be found in the location whose address is in a register.

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

Index Mode. The data element can be found in the location whose address is the sum of the contents of a 16-bit index value in a register and an address in the instruction.

Relative Address Mode. The data element can be found in the location whose address is the sum of the contents of the program counter and a 16-bit displacement in the instruction.

Base Address Mode. The data element can be found in the location whose address is the sum of a base address in a register and a 16-bit displacement in the instruction.

Base Index Mode. The data element can be found in the location whose address is the sum of a base address in one register and an index value in another register.

Chapter 5 defines and illustrates the eight addressing modes.

2.12 Extended Processing Architecture

An important feature of the Z8000 CPU architecture is the Extended Processing Architecture (EPA) facility. This facility provides a mechanism by which the basic instruction set of the CPU can be extended via external devices, called Extended Processing Units (EPUs). A special set of instructions, called extended instructions, is used to control this feature. When the CPU encounters one of these extended instructions in its instruction stream, it will either trap to a software trap handler to process the instruction or it will perform 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.

2.12 Extended Processing Architecture (Continued)

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

2.13 Exceptions

Three events can alter the normal execution of a Z8000 program: hardware interrupts that occur when a peripheral device needs service, synchronous software traps that occur when an error condition arises, and system reset. Chapter 7 contains a detailed description of exceptions and how they are handled. Interrupt requests and segmentation trap requests are accepted after the completion of the instruction execution during which they were made. At the end of the instruction execution, a spurious instruction fetch transaction is usually performed before the interrupt or acknowledge sequence begins, but the Program Counter is not affected by the spurious fetch.

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 Flag and Control Word (FCW) and the Program Counter (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 process or state. The four kinds of traps are:

Extended instruction attempted in non-EPA mode. The current instruction is an EPU instruction, but the system is not in EPA mode. This trap allows system software to either simulate instruction or abort the program.

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 violation (supplied by external circuit). A segmentation violation, such as using an offset larger than the defined length of the segment, can be made to cause an external memory management system to signal a segmentation trap. This can occur only with the segmented Z8001.

2.13.3 Interrupts. Interrupts are asynchronous events typically triggered by peripheral devices needing attention. The three kinds of interrupts associated with the three interrupt lines of the CPU are:

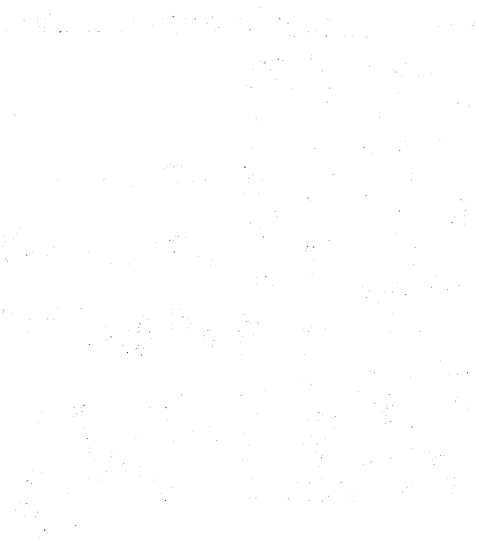
Non-maskable interrupts ($\overline{\text{NMI}}$). These interrupts cannot be disabled and are usually reserved for critical external events that require immediate attention.

Vectored interrupts ($\overline{\text{VI}}$). These interrupts cause eight bits of the vector output by the interrupting device to be used to select a particular interrupt service procedure to which the program automatically branches.

Non-vectored interrupts ($\overline{\text{NVI}}$). These interrupts are maskable interrupts which are all handled by the same interrupt procedure.

2.13.4 Trap and Interrupt Service Procedures. Interrupts and traps are handled similarly by the Z8000 CPU. The Z8000 CPU automatically acknowledges interrupts and processes traps in system mode. In the case of the segmented Z8001, the CPU uses the segmented mode regardless of its mode at the time of interrupt or trap. The program status information in effect just prior to the interrupt or trap is pushed onto the system stack. An additional word, which serves as an identifier for the interrupt or trap, also is 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.

Zilog



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. In either case, the location of the information must be specified by an *address* of some sort before that information can be accessed. A set of these addresses is called an *address space*.

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

- *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 operations:

- *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 Types of Address Spaces

Within the two general types of address spaces (memory and I/O), it is possible to distinguish several subcategories. Figure 3.1 shows the address spaces that are available on both the Z8001 and the Z8002.

The difference between the Z8001 and the Z8002 lies not in the number and type of address spaces, but rather in the organization and maximum size of each space. For the Z8001, each of the six memory address spaces contains 8M byte addresses grouped into 128 segments, for a total memory addressing capability of 48M bytes. For the Z8002, each memory space is a homogeneous collection of 64K byte addresses. In both the Z8001 and the Z8002, the I/O address spaces contain 64K port addresses. When an address is used to access data, the address spaces may be distinguished 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).

- *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 that user or system programs operate on.
- *Stack Spaces (status = 1001 or 1011), normal mode ($N/\overline{S}$ = 1) or system mode ($N/\overline{S}$ = 0).* These spaces can be used to address the system and normal program stacks.
- *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 Management Unit).

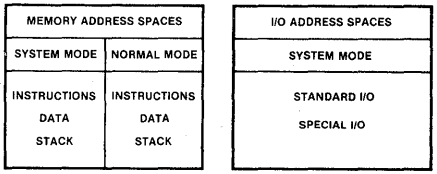


Figure 3-1. Address Spaces on the Z8001 and Z8002

3.3 I/O Address Spaces

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. In standard I/O space, byte ports must have an odd address; in special I/O space, byte ports must have an even address.

3.4 Memory Address Spaces

Each memory address space in the Z8002, or each segment in each memory address space on the Z8001, 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. As Chapter 6 explains in detail, different assembler mnemonics are used for addressing bytes, words, and long words. Moreover, 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 position 2^n in the conventional representation of 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 3.2). 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.

Certain memory locations are reserved for system-reset handling. These are described fully in Chapter 7. Except for these reserved locations, there are no memory addresses specifically designated for a particular purpose.

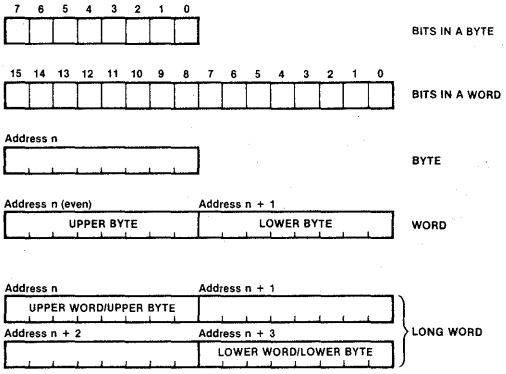


Figure 3-2. Addressable Data Elements

3.4.2 Segmented and Non-Segmented Addresses. The two versions of the Z8000 CPU generate two kinds of addresses with different lengths. The Z8002 generates a 16-bit address specifying one of 64K bytes. The Z8001 generates a 23-bit *segmented* address. A segmented address consists of a 7-bit *segment number*, which specifies one of 128 segments, and a 16-bit *offset*, which specifies one of up to 64K bytes in the segment. Each segment is an independent collection of bytes; thus, instructions and multiple byte data elements cannot cross segment boundaries. Some of the advantages of address segmentation are outlined in Section 3.4.3.

Figure 3.3 shows the format 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 a 32-bit long word and thus can be stored in a long word register (RRn) or a long word memory element. There is a short encoding of segmented addresses that appears in instructions and requires only 16 bits.

It is important to realize that even though the Z8001 can operate in nonsegmented mode (Chapter 4), it always generates segmented addresses. The segment number is supplied by the program counter segment number.

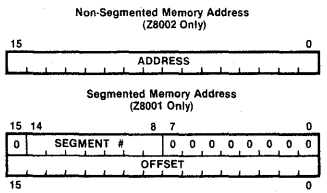


Figure 3-3. Segmented and Non-Segmented Address Formats

26-0001-0906 26-0001-0907

3.4 Memory Address Spaces (Continued)

3.4.3 Segmentation and Memory Management. Addresses manipulated by the programmer, used by instructions, and output by the Z8001 are called "logical addresses." An external memory-management circuit can translate logical addresses into physical (actual) memory addresses and perform certain checks to insure data and programs are properly accessed.

The Z8010 Memory Management Unit (MMU) performs this function for the segmented addresses produced by the Z8001 CPU. A single MMU keeps a descriptor for each of 64 segments. This descriptor tells where in physical memory the segment lies, 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). At the same time, the MMU checks for errors such as writing into a read-only segment or a system segment being accessed by a nonsystem program. MMUs are designed to be combined so that more than 64 segments can be supported at once. The CPU does not require MMUs; the segment number can be used directly as part of a physical address.

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

- Provision for flexible and efficient allocation of physical memory resources during the execution of programs.
- Hardware stack overflow protection.
- 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 users from system functions.

Segmentation in the Z8001 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 cycle. This keeps access to the segment descriptor in the MMU from 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 the 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 MMU and memory management can be found in *An Introduction to the Z8010 MMU Memory Management Unit* and in the *Z8010 MMU Technical Manual.*

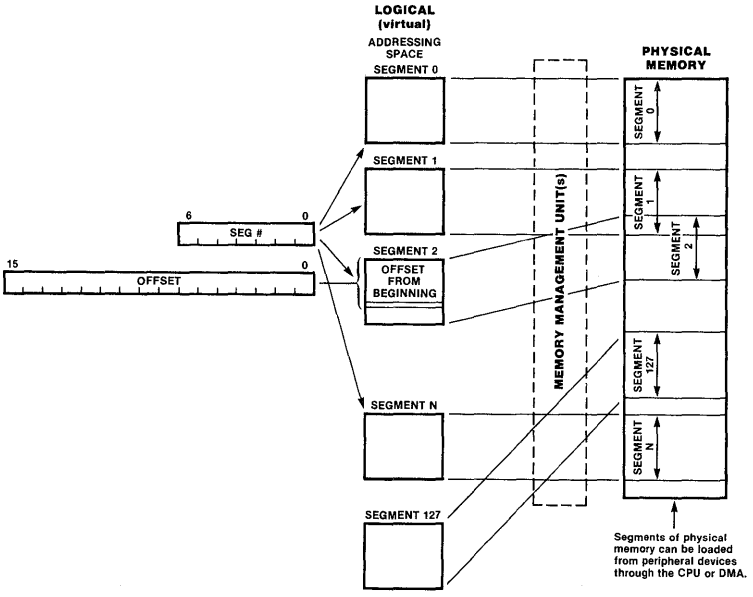


Figure 3-4. Segmented Address Translation

Zilog

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 manifest on 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 done. In this state, memory refresh transactions 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 refresh needs to be done, in which case the CPU temporarily enters Stop/Refresh state.
- An external stop request pushes the CPU into Stopped state.
- An external bus request pushes the CPU into Bus-Disconnect state.

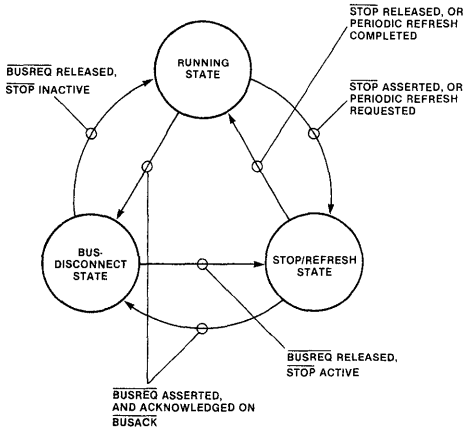


Figure 4-1. Operating States and Transistions

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. While the CPU is in Bus-Disconnect state, it does nothing. It enters Bus-Disconnect state from either Running state or Stop/Refresh state when a bus request has been received on $\overline{\text{BUSREQ}}$ and acknowledged on $\overline{\text{BUSACK}}$ as (described in Chapter 9). While in this state, it disconnects itself from the bus by 3-stating its output. It leaves Bus-Disconnect state when the external bus request has been released. Note that 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 conditions indicating that a different state should be entered.

4.2.4 Effect of Reset. Activation of the CPU's $\overline{\text{RESET}}$ line puts the CPU in 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 $\overline{\text{RESET}}$ is deactivated. When this occurs, the program enters one of the three operating states described above, depending on the state of $\overline{\text{BUSREQ}}$ and $\overline{\text{STOP}}$ inputs. 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, the flags control branching (as described in Chapter 6), and the control bits determine the mode in which the CPU operates and the interrupts that are masked (see Chapter 7).

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 and 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 Branch, Call, Interrupt Return and Load Program Status, 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 operation 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 the typical alternative of fetching each instruction only after the prior instruction has completed execution.

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

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 directly manipulated by programs. In segmented mode (SEG = 1), programs manipulate 23-bit segmented addresses; in nonsegmented mode (SEG = 0), programs generate 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:

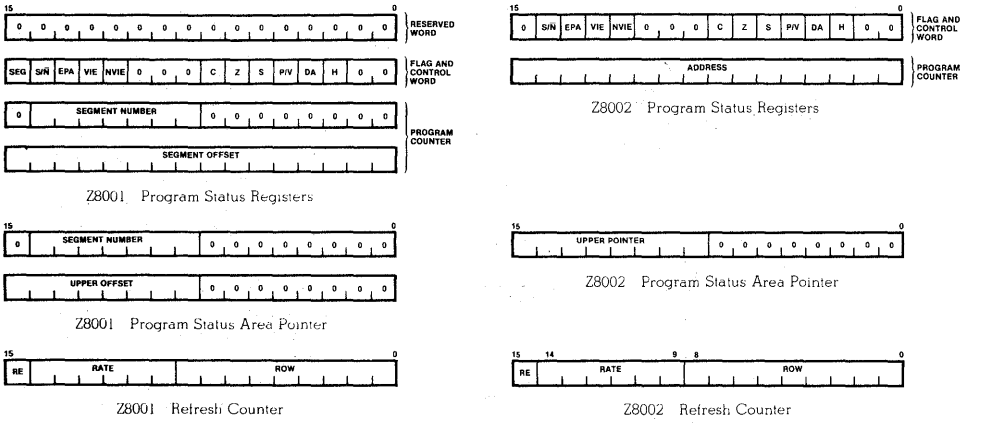


Figure 4-2. Program Status Registers

4.3 Instruction Execution (Continued)

- 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-bits in nonsegmented mode. They consist of a 7-bit segment number and either an 8-bit or 16-bit offset in segmented mode.

Segmented mode is available only on the Z8001 CPU; on the Z8002, the segment bit is always forced to zero, indicating nonsegmented mode. Because the Z8001 supports segmented and nonsegmented modes, it is possible to run programs written for the Z8002 on the Z8001 without alteration. The reverse is not possible. The Z8001 CPU always generates 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 operation 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, certain privileged instructions that alter sensitive parts of the machine state (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 copies of the Stack Pointer registers (Register 15 in the Z8002 and Registers 14 and 15 in the Z8001): one for normal mode and one for system mode. When in normal mode, a reference to the Stack Pointer register by an instruction will access the normal Stack Pointer. When in system mode, an access to the Stack Pointer register will reference the system Stack Pointer, unless the Z8001 is running in nonsegmented system mode, in which case a reference to R14 will access the normal mode R14. This is summarized in 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 instruction is executed or when an exception (interrupt, trap, or reset) occurs. There is a special instruction (system call) whose sole purpose is to generate a trap and thus provide a controlled transition from normal to system mode.

The distinction between normal/system mode allows the construction of a protected operating system. This is a program that runs in system mode and controls the system's resources, managing the execution of one or more application programs which run in normal mode. Normal and system modes, along with Memory Protection, provide the basis for protecting the operating system from malfunctions of application programs.

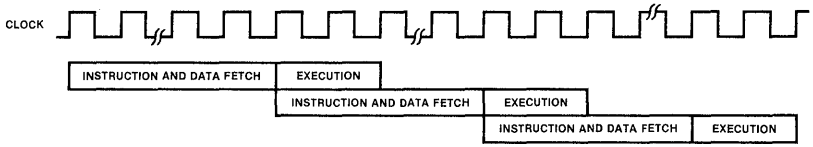


Figure 4-3. Instruction Look-Ahead

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	Normal R14	Normal R14	Normal R14
	System R15	System R15	Normal R15	Normal R15

Note: Z8002 always runs in nonsegmented mode.

Table 4.1 Registers Accessed by References to R14 and R15.

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. The execution of these instructions is controlled by the EPA control bit in the FCW.

When the EPA bit is zero, it indicates that there is no Extended Processing Unit connected to the CPU and causes the CPU to trap (as explained in Chapter 7) when it encounters an extended instruction. This allows the operation of the extended instruction to be simulated by software running on the CPU.

If the EPA bit is set, it indicates that an Extended Processing Unit is connected to the CPU in order to process the operation encoded in the extended instruction. 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. The CPU will fetch and begin executing the next instruction in its instruction stream. The Extended Processing Unit is expected to monitor the CPU's activity, participate in extended instruction data transfers initiated by the CPU, and execute the extended instruction. While the Extended Processing Unit is executing the instruction, the CPU can be fetching and executing further instructions. If the CPU fetches another extended instruction before the Extended Processing Unit is finished executing a previous instruction, the $\overline{\text{STOP}}$ line may be used to delay the CPU until the previous instruction is complete. This process is described more fully in Chapters 6 and 9.

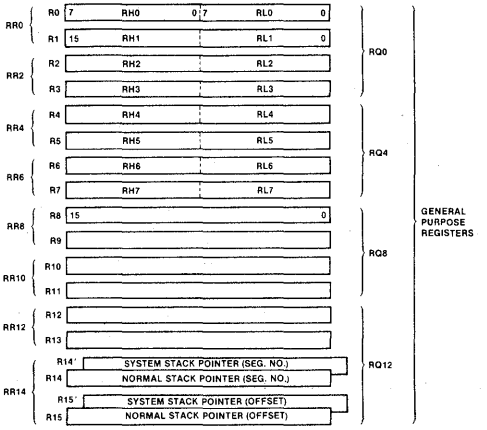
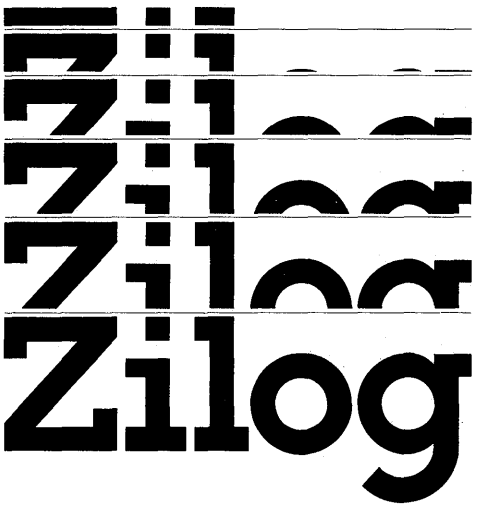


Figure 4-4. General-Purpose Registers

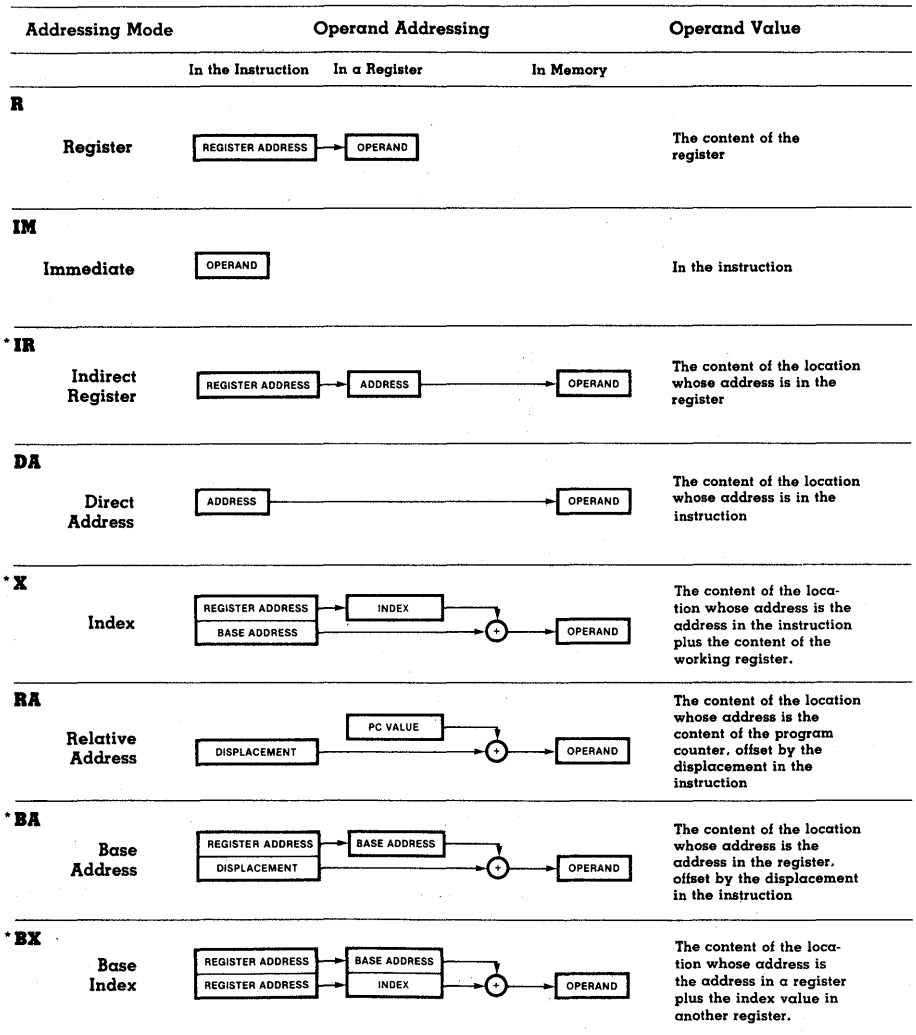

Zilog

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. Separate sets of 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 may 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.



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

Figure 5-1. Addressing Modes

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 within the register.
- 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 may be automatically stepped either forward or backward through memory locations.
- As index or base registers, where the contents of the register and the word(s) following the instruction are combined to produce the address of the operand. This allows efficient access to a variety of data structures.

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.
- Register R15' (or the double register RR14' in the Z8001) is used in acknowledging interrupts and therefore can never be used as an accumulator in system-mode operation. The system-mode registers, R14' and R15', are automatically accessed when R14, R15, or RR14 are referenced by instructions executed in system mode.

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

- Register R15 (or the double register RR14 in the Z8001) is used as a stack pointer for subroutine calls and returns.
- The byte register RH1 is used in the translate bulleted item instructions (TRDB, TRDRB, TRIB, TRIRB) and the translate and test instructions (TRTDB, TRTDRB, TRTIB, TRTIRB).
- Register R0 is used in extended instructions.

In 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 Program Counter normally is used only to keep track of the next instruction to be executed; whenever an instruction is fetched from memory, the PC is incremented to point to the next instruction. For addressing purposes, however, the updated PC serves as a base for referencing an operand relative to the location of an instruction. Operands specified by relative addressing reside in the program address space if the memory system distinguishes between program and data or stack address spaces.

Two of the addressing modes, Direct Address and Index, involve an I/O or memory address as part of the instruction. I/O addresses are always 16 bits long, as are nonsegmented memory addresses (Z8002), so these addresses occupy one word in the instruction. Segmented addresses generated by the Z8001 are 23 bits long. Within an instruction, a segmented address may occupy 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 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.

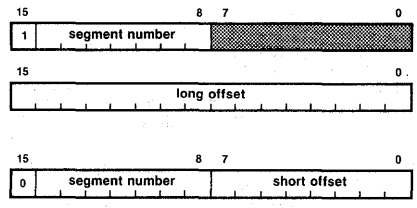


Figure 5-2. Segmented Memory Address Within Instruction.

NOTE: Shaded area is reserved.

5.3 Addressing Mode Descriptions

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

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

The descriptions are grouped into two sections—one for nonsegmented CPUs, the other for segmented CPUs. Users of the Z8002 need refer to the first section only; users of the Z8001 in nonsegmented mode should also refer to the first section, while users of Z8001 in segmented mode should refer to the second section. In the examples, hexadecimal notation is used for memory addresses and the contents of registers and memory locations. The % symbol precedes hexadecimal numbers in assembly language text.

5.4 Descriptions and Examples (Z8002 and Z8001 Nonsegmented Mode)

In this section, the addressing modes of both the Z8002 and the nonsegmented mode Z8001 are described.

5.4.1 Register (R). In the Register addressing mode the instruction processes data taken from a specified general-purpose register. Storing data in a register allows shorter instructions and faster execution than occur with instructions that access memory.

THE OPERAND VALUE IS THE CONTENTS OF THE REGISTER.

The operand is always in the register address space. The register length (byte, word, register pair, or register quadruple) is specified 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!
```

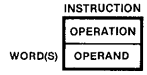
Before Execution

R2	A6B8
R3	9A20

After Execution

R2	9A20
R3	9A20

5.4.2 Immediate (IM). The Immediate addressing mode is the only mode that does not indicate a register or memory address as 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!
```

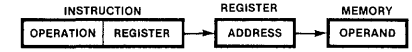
Before Execution

R2	6789

After Execution

R2	5589

5.4.3 Indirect Register (IR). In the Indirect Register addressing mode, the data processed is not the value in the specified register. Instead, the register holds the address of the data.



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 can be used except R0.

Depending on 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 operand will be in stack memory space if the stack pointer (R15) is used as the indirect register; otherwise, the operand will be in data memory space.

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

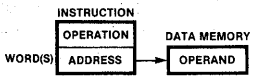
	⋮
170A	A023
170C	0B0E
170E	10D0
	⋮

After Execution

R2	0B0E
R3	0005
R4	2000
R5	170C

5.4 Descriptions and Examples (Z8002 and Z8001 Nonsegmented Mode) (Continued)

5.4.4 Direct Address (DA). In the Direct Addressing mode, the data processed is found at the address specified in the instruction.



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

Depending upon the instruction, the operand specified by DA mode will be either in I/O space (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. (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 address!
                   !5E23!
```

Before Execution

R2 | 6789 |

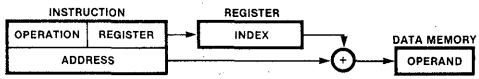
Memory

5E22	0106
5E24	0304
⋮	

After Execution

R2 | 0689 |

5.4.5 Index (X). In the Index Addressing mode, the instruction processes data located at an indexed address in memory. The indexed address is computed by adding the address specified in the instruction to an "index" contained in a word register, also specified by the instruction. 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.



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

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

Operands specified by X mode are always in the 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.

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!
```

Before Execution

R3 | 01FE |
R4 | 203A |

Memory

⋮	
2516	F3C2
2518	3D0E
251A	7ADA
⋮	

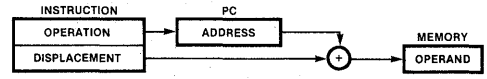
Address Calculation

```
  231A
+ 01FE
------
  2518
```

After Execution

R3 | 01FE |
R4 | 3D0E |

5.4.6 Relative Address (RA). In the Relative Addressing mode, the data processed is found at an address relative to the current instruction. The instruction specifies a two's complement displacement which is added to 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. (The assembler will take this into account in calculating the constant that is assembled into the instruction.)



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

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

As with the Direct Addressing mode, the Relative Addressing mode is used by certain program control instructions to specify the address of the next instruction to be executed (specifically, the result of the addition of the Program Counter value and the displacement is loaded into the Program Counter, except when executing the DJNZ or CALR instructions. The displacement is then subtracted from the PC, not added to it.) Relative addressing allows references forward or backward from the current Program Counter value and is used only for such instructions as Jumps or Calls and special loads (LDR) that can cross the normally strict boundary between program and data memory.

5.4 Descriptions and Examples (Z8002 and Z8001 Nonsegmented Mode) (Continued)

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 current!
                     !program counter!
                     ! + hex 6!
```

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

Before Execution

R2	A0F0
PC	0202

Program Memory

0202	3102 } Instruction
0204	0002 } Instruction
0206	E801
0208	FFFE

Address Calculation

```
  0206
+    2
------
  0208
```

After Execution

R2	FFFE
PC	0206

5.4.7 Base Address (BA). The Base Addressing mode is similar to Index mode in that a base and 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 resulting address points to the data to be processed. This addressing mode may be used only with the Load instructions. Base Addressing mode, as a complement to Index mode, allows random access to tables or other data structures where the displacement of an element within the structure is known, but the base of the particular structure must be computed by the program.

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

An operand specified by BA mode will be in stack memory space if the base register is the stack pointer (R15) and in data memory space 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

```
  20AA
+   18
------
  20C2
```

After Execution

RR2	R2	0A00
	R3	1500
	R4	3100
	R5	20AA

Memory

20C0	0ABE
20C2	0A00
20C4	1500
20C6	B0D1

5.4 Descriptions and Examples (Z8002 and Z8001 Nonsegmented Mode) (Continued)

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

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

An operand specified by BX mode will be in stack memory space if the base register is the stack pointer (R15) and in data memory otherwise.

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!
                    !R5 + the value in R3!
```

Before Execution

R2	1F3A
R3	FFFE
R4	0300
R5	1502

Data Memory

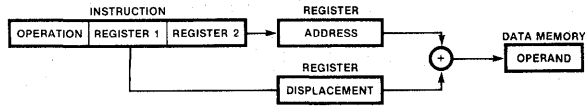
	⋮
14FE	0101
1500	B0DE
1502	F732
	⋮

Address Calculation

```
  1502
+ FFFE
  1500
```

After Execution

R2	B015
R3	FFFE
R4	0300
R5	1502

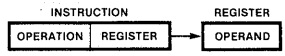


THE OPERAND VALUE IS THE CONTENTS OF THE LOCATION WHOSE ADDRESS IS THE CONTENTS OF THE ONE REGISTER OFFSET BY THE DISPLACEMENT IN THE SECOND REGISTER.

5.5 Descriptions and Examples (Segmented Z8001)

In this section, <<nn>> will often be used to refer to segment number nn.

5.5.1 Register (R). In the Register addressing mode, the instruction processes data taken from a specified general-purpose register. Storing data in a register allows shorter instructions and faster execution than occurs with instructions that access memory.



THE OPERAND VALUE IS THE CONTENTS OF THE REGISTER.

The operand is always in the register address space. 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!
```

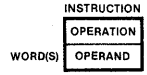
Before Execution

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

After Execution

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

5.5.2 Immediate (IM): The Immediate addressing mode is the only mode that 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.

26-0001-0921 26-0001-0922 26-0001-0923

5.5 Descriptions and Examples (Segmented Z8001) (Continued)

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!
```

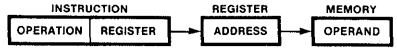
Before Execution

R2	6789

After Execution

R2	5589

5.5.3 Indirect Register (IR). In the Indirect Register addressing mode, the data processed is not the value in the specified register. Instead, the register holds the address of the data.



THE OPERAND VALUE IS 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 operand will be in stack memory space if the stack pointer (RR14) is used as the indirect register, otherwise the operand will be in data memory space.

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 may be used except R0 or RR0.

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 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:

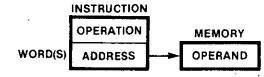
```
OUTB @R1,RL0
```

Before Execution

R0	0A23
R1	0011

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

5.5.4 Direct Address (DA). In the Direct Addressing mode, the data processed is found at the address specified as an operand in the instruction.



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

Depending upon the instruction, the operand specified by the Direct Address (DA) mode will be either in I/O space (standard 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 address serves as an immediate value that is loaded into the Program Counter.)

5.5 Descriptions and Examples (Segmented Z8001) (Continued)

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 lines "|" 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

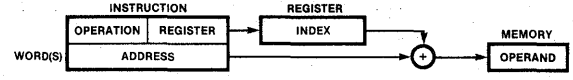
		⋮
<<15>>	0022	0206
	0024	0304
		⋮

After Execution

R2 | 0689 |

5.5.5 Index (X). In the Index addressing mode, the instruction processes data are located at an indexed address in memory. The indexed address is computed by adding the address specified in the instruction to an "index" contained in a word register, also specified by the instruction.

The *offset* of the operand address is computed by adding the 16-bit index value to the 8 or 16-bit offset portion of the address in the instruction. The segment *number* of the operand address comes directly from the instruction. (Any overflow is ignored—it neither sets the Overflow flag nor increments the segment number.) Indexed addressing allows random access to table 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.



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

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
		⋮

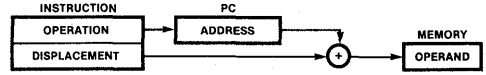
Address Calculation

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

After Execution

R3 | 01FE |
R4 | 3D0E |

5.5.6 Relative Address (RA). In the Relative Addressing mode, the data processed is found at an address relative to the current instruction. The instruction specifies a two's complement displacement which is added to the offset 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.)



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

5.5 Descriptions and Examples (Segmented Z8001) (Continued)

An operand specified by RA mode is always in the program memory address space. Either long or short format addresses may be used.

As with the Direct Addressing mode, the Relative Addressing mode is also used by certain program control instructions to specify the address of the next instruction to be executed (specifically, the result of the addition of the Program Counter value and the displacement is loaded into the Program Counter, except when executing the DJNZ or CALR instructions; the displacement is then subtracted from the PC, not added to it). Relative addressing allows short references forward or backward from the current Program Counter value and is used only for such instructions as Jumps and Calls and special loads (LDR). Note that because the segment number is unchanged relative addresses are located in the same segment as the instruction.

Assembler language format (see also Chapter 6):

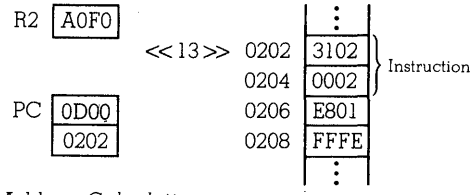
address

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 when the address calculation is performed, the constant that occurs in the instruction will actually be +2.

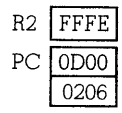
Before Execution



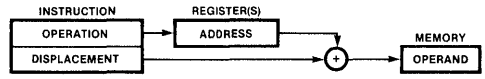
Address Calculation

```
<<13>>0206
+        2
<<13>>0208
```

After Execution



5.5.7 Base Address (BA). The Base Addressing mode is similar to 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, and the resulting address points to the data to be processed. (The segment number is not changed.) This addressing mode may be used only with the Load instructions. Base Addressing mode, as a complement to Index mode, 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 particular 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 can be used for the base address *except RR0*. The Base Address mode allows access to locations whose segment numbers are not known at assembly time.

An operand specified by BA mode will be in stack memory space if the base register is the stack pointer (RR14) and in data memory space otherwise.

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

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
```

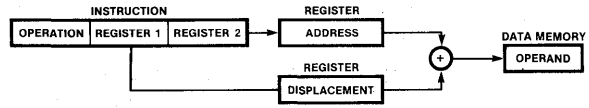
After Execution

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

Data Memory

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

5.5.8 Base Index (BX). The Base Index addressing mode is an extension of the Base Addressing mode and may 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 the base address. This mode allows access to memory locations whose physical addresses are computed at runtime and are not fully known at assembly time.



THE OPERAND VALUE IS THE CONTENTS OF THE LOCATION WHOSE ADDRESS IS THE CONTENTS OF THE ONE REGISTER OFFSET BY THE DISPLACEMENT IN THE SECOND REGISTER.

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 illegal in registers.

An operand specified by BX mode will be in stack memory space if the base register is the stack pointer (RR14) and in data memory otherwise.

Assembler language format (see also Chapter 6):

RRn (Rn)

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
```

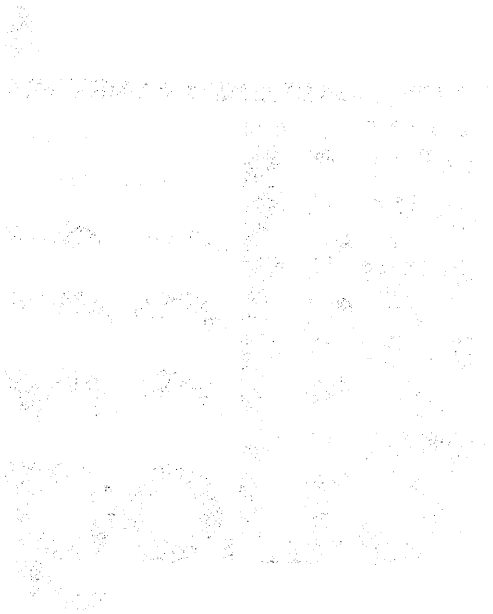
After Execution

RR2	R2	B0DE
	R3	FFFE
RR4	R4	0D00
	R5	1502

Data Memory

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

Zilog



Chapter 6
Instruction Set

6.1 Introduction

This chapter describes the instruction set of the Z8000. 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. Unusual instructions or features that are not typical of predecessor microprocessors are pointed out.

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. 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 may 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 CLRB	dst	Clear
EX EXB	dst, src	Exchange
LD LDB 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 LDRB LDRL	dst, src	Load Relative
POP POPL	dst, src	Pop
PUSH 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 (i.e., immediate data). These instructions are supported with the widest range of addressing modes, including the Base (BA) and the Base Index (BX) mode which are available here only. None of these instructions affect any of 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. A special one-word instruction, LDK, is also included to handle the frequent requirement for loading a small constant (0 to 15) into a register.

These instructions basically 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 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 and faster.

6.2 Functional Summary (Continued)

The Exchange instructions swap the contents of the source and destination operands.

The Load Multiple instruction provides for efficient saving and restoring of registers. This can significantly 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., one may save R9-R15 and R0-R3 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) may 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 may be a register or a memory location, specified by the DA, IR, or X addressing modes. Immediate data can also be pushed onto a stack one word at a time. Note that byte 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. They are useful for management of complex data structures.

6.2.2 Arithmetic Instructions

Instruction	Operand(s)	Name of Instruction
ADC ADCB	dst, src	Add with Carry
ADD ADDB ADDL	dst, src	Add
CP CPB CPL	dst, src	Compare
DAB	dst	Decimal Adjust
DEC DECB	dst, src	Decrement
DIV DIVL	dst, src	Divide
EXTS EXTSB EXTSL	dst	Extend Sign
INC INCB	dst, src	Increment
MULT MULTL	dst, src	Multiply
NEG NEGB	dst	Negate
SBC SBCB	dst, src	Subtract with Carry
SUB SUBB 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, 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 version of these instructions generally alters 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 (ADC, 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) mutliplies two 16-bit operands and produces a 32-bit result, which is loaded into the destination register pair. Similarly, 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 where the product has too many significant bits to be contained entirely 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 Functional Summary (Continued)

6.2.3 Logical Instructions.

Instruction	Operand(s)	Name of Instruction
AND ANDB	dst, src	And
COM COMB	dst	Complement
OR ORB	dst, src	Or
TEST TESTB TESTL	dst	Test
XOR 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 word 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 operand, which can be designated by any of four 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. The Complement and Test instructions can use four basic addressing modes to specify the destination.

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 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 statisfy 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 any instruction within the range -254 to +256 bytes from the current location. 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 and Call Relative 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, -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. These can be useful for implementing complex control structures such as dispatch tables.

The Conditional Return instruction 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 processor stack followed by the instruction itself, and a new program status is loaded from a dedicated part of

6.2 Functional Summary (Continued)

the Program Status Area. An 8-bit immediate source field in the instruction is ignored by the CPU hardware. It can be retrieved from the stack by the software which handles system calls and interpreted as desired, for example 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 BITB	dst, src	Bit Test
RES RESB	dst, src	Reset Bit
SET SETB	dst, src	Set Bit
TSET TSETB	dst	Test and Set
TCC TCCB	cc, dst	Test condition code

The instructions in this group are useful for manipulating individual bits in registers or memory. In most computers, this has to be done using the logical instructions with suitable masks, which is neither natural nor efficient.

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 statically by a value (0 to 7 for byte, 0 to 15 for word) in the instruction itself or it may be specified dynamically by the contents of a register, which could have been computed by previous instructions. In the latter case, the destination is restricted to a register. These instructions leave the flags unaffected. The companion Bit Test instruction (BIT, BITB) similarly tests a specified bit and sets the Z flag according to the state of the bit.

The Test and Set instruction (TSET, TSETB) is useful in multiprogramming and multiprocessing environments. It can be used for implementing synchronization mechanisms between processes on the same or different CPUs.

Another instruction in this group, Test Condition Code (TCC, TCCB) sets a bit in the destination register based on the state of the flags as specified by the condition code in the instruction. (See Section 5.6.1 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 RLB	dst, src	Rotate Left
RLC RLCB	dst, src	Rotate Left through Carry
RLDB	dst, src	Rotate Left Digit
RR RRB	dst, src	Rotate Right
RRC RRCB	dst, src	Rotate Right through Carry
RRDB	dst, src	Rotate Right Digit
SDA SDAB SDAL	dst, src	Shift Dynamic Arithmetic
SDL SDLB SDLL	dst, src	Shift Dynamic Logical
SLA SLAB SLAL	dst, src	Shift Left Arithmetic
SLL SLLB SLLL	dst, src	Shift Left Logical
SRA SRAB SRAL	dst, src	Shift Right Arithmetic
SRL SRLB SRLL	dst, src	Shift Right Logical

This group contains a rich repertoire of instructions for shifting and rotating data registers.

Instructions for shifting arithmetically or logically in either direction are available. 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 statically by a field in the instruction or dynamically by the contents of a register. The ability to determine the shift amount dynamically is a useful feature, which is not available in most minicomputers.

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

6.2 Functional Summary (Continued)

6.2.7 Block Transfer And String Manipulation Instructions.

Instruction	Operand(s)	Name of Instruction
CPD CPDB	dst, src, r, cc	Compare and Decrement
CPDRB	dst, src, r, cc	Compare, Decrement and Repeat
CPI CPIB	dst, src, r, cc	Compare and Increment
CPIR CPIRB	dst, src, r, cc	Compare, Increment and Repeat
CPSD CPSDB	dst, src, r, cc	Compare String and Decrement
CPSDR CPSDRB	dst, src, r, cc	Compare String, Decrement and Repeat
CPSI CPSIB	dst, src, r, cc	Compare String and Increment
CPSIR CPSIRB	dst, src, r, cc	Compare String, Increment and Repeat
LDD LDDB	dst, src, r	Load and Decrement
LDDR LDRB	dst, src, r	Load, Decrement and Repeat
LDI LDIB	dst, src, r	Load and Increment
LDIR 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

This is an exceptionally powerful group of instructions that provides a full complement of string comparison, string translation and block transfer functions. Using these instructions, a byte or word block 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. The more complex 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 may be repeated automatically while decrementing a length counter until it is zero, or they may 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 implementing more complex operations in software by 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. The autorepeat forms of these instructions always leave this flag set.

The D and H flags are not affected by any of these instructions. The C and 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). Note that in the segmented mode, only the low-order half of the register pair gets incremented or decremented as with all address arithmetic in the Z8000.

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. The address of the instruction itself, rather than the next one, is saved on the stack, and the contents of the operand pointer registers, as well as the repetition counter, are such that the instruction can simply be reissued after returning from the interrrupt without any visible difference in its effect.

6.2 Functional Summary (Continued)

6.2.8 Input/Output Instructions.

Instruction	Operand(s)	Name of Instruction
IN INB	dst, src	Input
IND INDB	dst, src, r	Input and Decrement
INDR INDRB	dst, src, r	Input, Decrement and Repeat
INI INIB	dst, src, r	Input and Increment
INIR INIRB	dst, src, r	Input, Increment and Repeat
OTDR OTDRB	dst, src, r	Output, Decrement and Repeat
OTIR OTIRB	dst, src, r	Output, Increment and Repeat
OUT OUTB	dst, src	Output
OUTD OUTDB	dst, src, r	Output and Decrement
OUTI OUTIB	dst, src, r	Output and Increment
SIN SINB	dst, src	Special Input
SIND SINDB	dst, src, r	Special Input and Decrement
SINDR SINDRB	dst, src, r	Special Input, Decrement and Repeat
SINI SINIB	dst, src, r	Special Input and Increment
SINIR SINIRB	dst, src, r	Special Input, Increment and Repeat
SOTDR SOTDRB	dst, src, r	Special Output, Decrement and Repeat
SOTIR SOTIRB	dst, src, r	Special Output, Increment and Repeat
SOUT SOUTB	dst, src	Special Output
SOUTD SOUTDB	dst, src, r	Special Output and Decrement
SOUTI 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 the Z-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. 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, 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 address space. For the Standard I/O instructions, the port number may be specified statically in the instruction or dynamically 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 form a powerful and complete complement of instructions for transferring blocks of data between I/O ports and memory. The operation of these instructions is very similar to that of the block move instructions described earlier, with the exception that one operand is always an I/O port which remains unchanged as 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 only be executed in system mode. The single byte/word I/O instructions don't 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.

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 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

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

6.2 Functional Summary (Continued)

6.2.10 Extended Instructions. The Z8000 architecture includes a powerful 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 Processor Architecture.) A group of six opcodes, 0E, 0F, 4E, 4F, 8E and 8F (in hexadecimal), is dedicated for the implementation of extended instructions using this facility. The five basic addressing modes (R, IR, DA, IM and X) can be used by extended instructions for accessing data for the EPUs.

There are four types of extended instructions 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 CPU flag and control word. The last type is useful when the program must branch based on conditions determined by the EPU. The action taken by the CPU upon encountering extended instructions is dependent upon the EPA control bit in the CPU's FCW. When this bit is set, it indicates that the system configuration includes EPUs; therefore, the instruction is executed. If this bit is clear, the CPU traps (extended instruction trap) so that a trap handler in software can emulate the desired operation.

6.3 Processor Flags

The processor flags are a 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 may be used to determine the operation of a subsequent instruction, typically a conditional jump instruction. An example is a Test 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 for the use of the programmer and the Z8000 processor:

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

The flags are modified by many instructions, including the arithmetic and logical instructions.

Appendix C lists the instructions and the flags they affect. In addition, there are Z8000 CPU control instructions which 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. They are not used explicitly by the programmer.

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. The contents of the flag register may 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. One of these instructions is 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 when 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.

6.3 Processor Traps (Continued)

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	0	1	0	0	1
+105	0	1	1	0	1	0	0	1
225	1	1	1	0	0	0	0	1
	↳1 = Overflow flag							

The result in this case (-95 in two's complement notation) is incorrect, thus the overflow flag would be 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 and the flag is set if the total is even (that is, P = 1). If the total is odd (P = 0), the flag is reset. This flag is often 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 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 normally accessed by the programmer.

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. It would take 16 bits to specify any of the 65,536 (2^{16}) boolean functions of the four flags. Since only a very small fraction of these are generally of interest, this procedure would be very wasteful. Sixteen functions of the flag settings found to be frequently useful 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, the assembler recognizes more than sixteen mnemonics 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.). Program clarity is enhanced by having separate mnemonics for the same binary value of the condition codes in these cases.

6.5 Instruction Interrupts and Traps

Interrupts are discussed in detail in Section 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. However, certain instructions which might take a long time to complete are designed to be interruptible so as to minimize the length of time it takes the CPU to respond to an interrupt. These are the iterative versions of the String and Block instructions and the Block I/O instruction. If an interrupt request is received during one of these interruptible instructions, the instruction is suspended after the current iteration. The address of the instruction itself, rather than the address of the following instruction, 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 which index into the block operands are such that after each iteration when the instruction is reissued upon returning from an interrupt, the effect is the same as if the instruction were not interrupted. This assumes, of course, the interrupt handler preserved the registers, which is a general requirement on interrupt handlers.

The longest noninterruptible instruction that can be used in normal mode is Divide Long (749 cycles in the worst case). Multi-Micro-Request, a privileged instruction, can take longer depending on the contents of the destination register.

Traps are synchronous events that result from the execution of an instruction. The action of the CPU in response to a trap condition is similar to the case of an interrupt (see Section 7). Traps are non-maskable.

6.5 Instruction Interrupts and Traps (Continued)

The Z8000 CPUs implement four kinds of traps:

- Extended Instruction
- Privileged Instruction in normal mode
- Segmentation violation
- System Call

The Extended Instruction trap occurs when an Extended Instruction is encountered, but the Extended Processor Architecture Facility is disabled, i.e., the EPA bit in the FCW is a 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.

The privileged instruction trap serves to protect the integrity of a system from erroneous or unauthorized actions of arbitrary 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 most of the instructions that operate on the FCW are privileged, as are instructions like HALT and IRET.

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.

6.6 Notation and Binary Encoding

The rest of this chapter consists of detailed descriptions of each instruction, listed in alphabetical order. 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 an instruction begins with the instruction mnemonic and instruction name in the top part of the page. Privileged instructions are also identified at the top.

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 operation of the instruction is presented next, followed by a detailed discussion of the instruction.

The next part specifies the effect of the instruction on the processor flags. This is followed by a table that presents all the variants of the instruction for each applicable addressing mode and operand size. For each of these variants, the following information is provided:

A. Assembler Language Syntax. The syntax is shown for each applicable operand width (byte, word or long). The invariant part of the syntax is given in UPPER CASE and must appear as shown. Lower case characters represent the variable part of the syntax, for which suitable values are to be substituted. The syntax shown is for the most basic form of the instruction recognized by the assembler. For example,

ADD Rd,#data

represents a statement of the form ADD R3,#35. The assembler will also accept variations like ADD TOTAL, #NEW-DELTA where TOTAL, NEW and DELTA have been suitably defined.

The following notation is used for register operands:

Rd, Rs, etc.:	a word register in the range R0-R15
Rbd Rbs:	a byte register RHn or RLn where n = 0 – 7
RRd RRs:	a register pair RR0, RR2, ... RR14
RQd:	a register quadruple RQ0, RQ4, RQ8 or RQ12

The "s" or "d" represents a source or destination operand. Address registers used in Indirect, Base and Base Index addressing modes represent word registers in nonsegmented mode and register pairs in segmented mode. A one-word register used in segmented mode is flagged and a footnote explains the situation.

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

6.6 Notation and Binary Encoding (Continued)

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 in Assembler Language Syntax. 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
RQ0	RR0	R0	RH0	0000
		R1	RH1	0001
	RR2	R2	RH2	0010
		R3	RH3	0011
RQ4	RR4	R4	RH4	0100
		R5	RH5	0101
	RR6	R6	RH6	0110
		R7	RH7	0111
RQ8	RR8	R8	RL0	1000
		R9	RL1	1001
	RR10	R10	RL2	1010
		R11	RL3	1011
RQ12	RR12	R12	RL4	1100
		R13	RL5	1101
	RR14	R14	RL6	1110
		R15	RL7	1111

For easy cross-references, the same symbols are used in the Assembler Language Syntax and the instruction format. In the case of addresses, the instruction format in segmented mode uses "segment" and "offset" to correspond to "address," while the instruction format contains "displacement," indicating that the assembler has computed the displacement and inserted it as indicated.

A condition code is indicated by "cc" in both the Assembler Language Syntax and the instruction formats. The condition codes, the flag settings they represent, and the binary encoding in the instruction 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

Note that 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. A short assembly language example is given showing the use of the instruction.

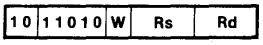
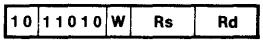
ADC
Add With Carry

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 ADCB Rbd, Rbs		5		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.

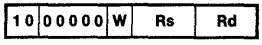
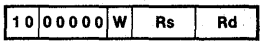
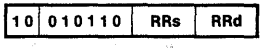
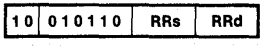
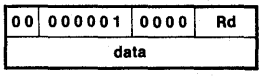
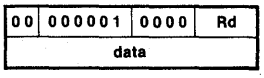
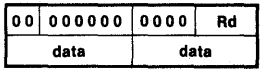
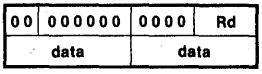
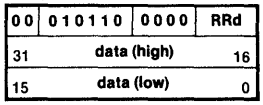
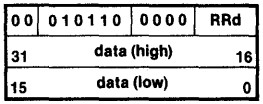
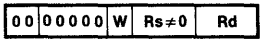
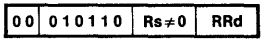
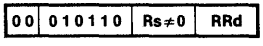
ADD
Add

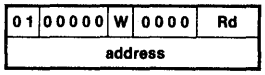
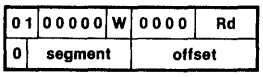
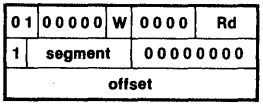
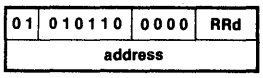
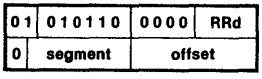
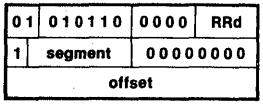
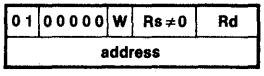
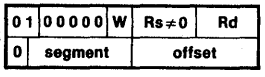
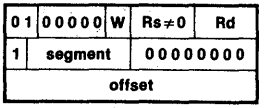
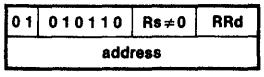
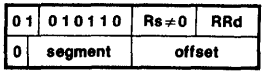
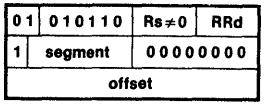
ADD dst, src — dst: R
ADDB — src: R, IM, IR, DA, X
ADDL

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.

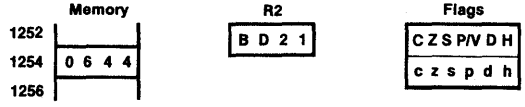
Flags:
C: Set if there is a carry from the most significant bit of the result; cleared otherwise
Z: Set if the result is zero; cleared otherwise
S: Set if the result is negative; cleared otherwise
V: Set if arithmetic overflow occurs, that is, if both operands were of the same sign and the result is 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 ADDB Rbd, Rbs		4		4
	ADDL RRd, RRs		8		8
IM:	ADD Rd, #data		7		7
	ADDB Rbd, #data		7		7
	ADDL RRd, #data		14		14
IR:	ADD Rd, @Rs[1] ADDB Rbd, @Rs[1]		7		7
	ADDL RRd, @Rs[1]		14		14

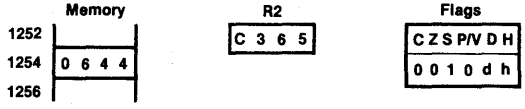
Source Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
DA:	ADD Rd, address ADDB Rbd, address		9	SS 	10
				SL 	12
	ADDL RRd, address		15	SS 	16
				SL 	18
X:	ADD Rd, addr(Rs) ADDB Rbd, addr(Rs)		10	SS 	10
				SL 	13
	ADDL RRd, addr(Rs)		16	SS 	16
				SL 	19

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

Before instruction execution:



After instruction execution:



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

AND
And

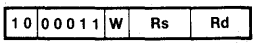
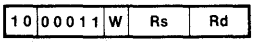
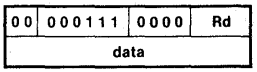
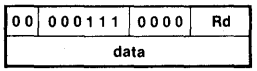
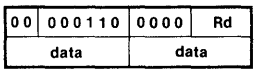
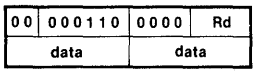
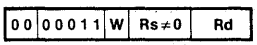
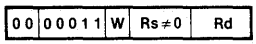
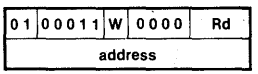
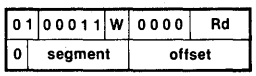
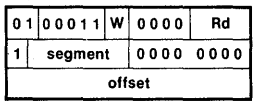
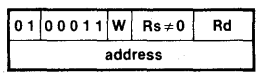
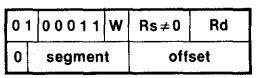
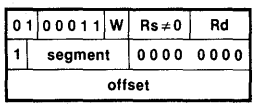
AND dst, src
ANDB

dst: R
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 ANDB Rbd, Rs		4		4
IM:	AND Rd, #data		7		7
	ANDB Rbd, #data		7		
IR:	AND Rd, @Rs[1] ANDB Rbd, @Rs[1]		7		7
DA:	AND Rd, address ANDB Rbd, address		9	SS 	10
				SL 	12
X:	AND Rd, addr(Rs) ANDB Rbd, addr(Rs)		10	SS 	10
				SL 	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
BITB

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

or

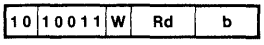
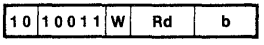
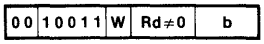
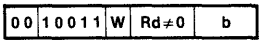
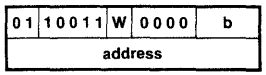
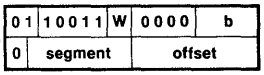
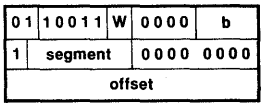
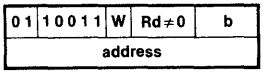
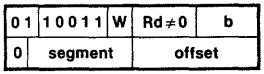
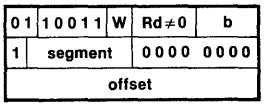
dst: R
src: R

Operation: Z ← NOT dst (src)

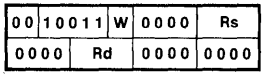
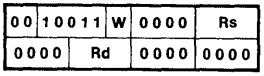
The specified bit within the destination operand is tested, and the Z flag is set to one if the specified bit is zero; otherwise the Z flag is cleared to zero. The contents of the destination are not affected. The bit number (the source) can be specified statically as an immediate value, or dynamically 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 BITB Rbd, b		4		4
IR:	BIT @Rd[1], b BITB @Rd[1], b		8		8
DA:	BIT address, b BITB address, b		10	SS 	11
				SL 	13
X:	BIT addr(Rd), b BITB addr(Rd), b		11	SS 	11
				SL 	14

Bit Test Static (Continued)

Source Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
R:	BIT Rd, Rs BITB Rbd, Rs		10		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.

CALL
Call

CALL dst dst: IR, DA, X

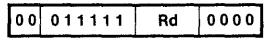
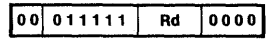
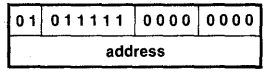
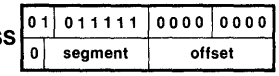
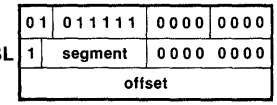
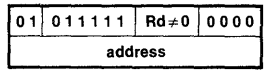
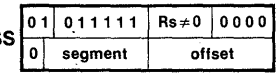
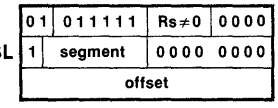
Operation:

Nonsegmented
SP ← SP − 2
C → SP ← PC
PC ← dst

Segmented
SP ← SP − 4
@SP ← PC
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 byte 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]		10		15
DA:	CALL address		12	SS 	18
				SL 	20
X:	CALL addr(Rd)		13	SS 	18
				SL 	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 dst dst: RA

Operation:

Nonsegmented
SP ← SP − 2
@SP ← PC
PC ← PC − (2 × displacement)

Segmented
SP ← SP − 4
@SP ← PC
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 if nonsegmented, or RR14 if segmented. (The program counter value used is the address of the first instruction byte 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 calculated by doubling the displacement in the instruction, then subtracting this value from the current value of the PC to derive the destination address. The displacement is a 12-bit signed value in the range −2048 to +2047. Thus, the destination address must be in the range −4092 to +4098 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.

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

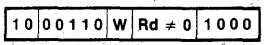
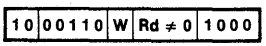
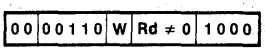
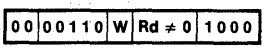
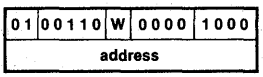
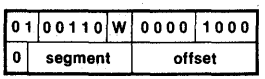
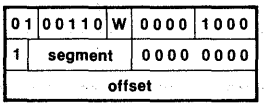
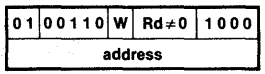
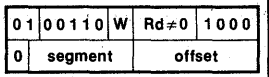
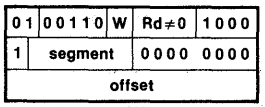
CLR dst
CLRB

dst: R, IR, DA, X

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 CLRB Rbd		7		7
IR:	CLR @Rd[1] CLRB @Rd[1]		8		8
DA:	CLR address CLRB address		11	SS 	12
				SL 	14
X:	CLR addr(Rd) CLRB addr(Rd)		12	SS 	12
				SL 	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.

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

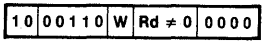
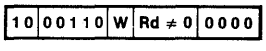
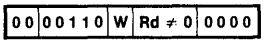
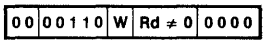
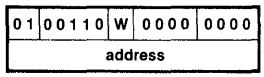
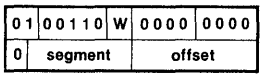
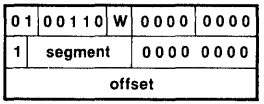
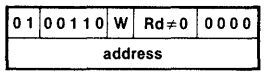
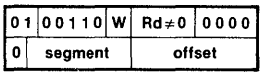
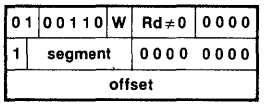
COM dst
COMB

dst: R, IR, DA, X

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 COMB Rbd		7		7
IR:	COM @Rd[1] COMB @Rd[1]		12		12
DA:	COM address COMB address		15	SS 	16
				SL 	18
X:	COM addr(Rd) COMB addr(Rd)		16	SS 	16
				SL 	19

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

```
COM  R1
```

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

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 dst, src
CPB
CPL

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

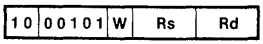
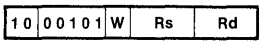
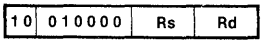
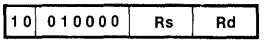
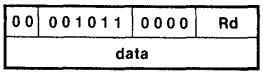
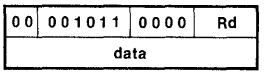
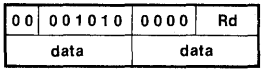
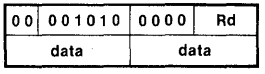
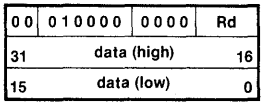
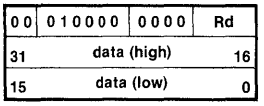
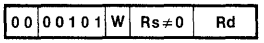
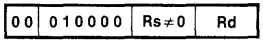
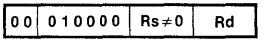
Operation: dst – src

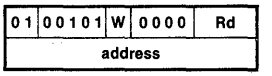
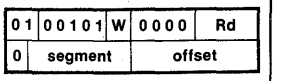
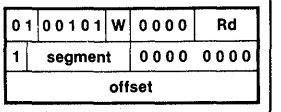
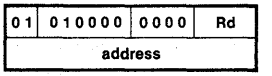
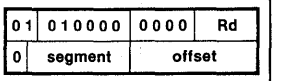
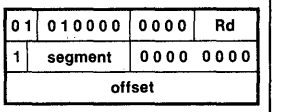
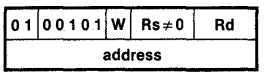
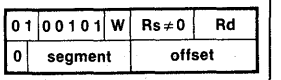
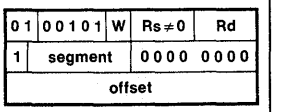
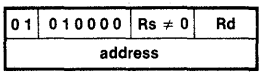
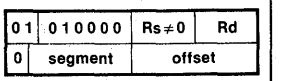
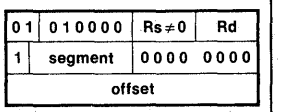
The source operand is compared to (subtracted from) the destination operand, and the appropriate flags set accordingly, which may then be used for arithmetic and logical conditional jumps. Both operands are unaffected, with the only action being the setting of the flags. Subtraction is performed by adding 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:

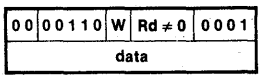
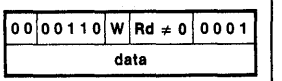
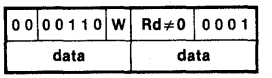
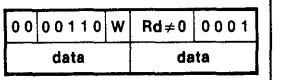
C: Cleared if there is a carry from the most significant bit of the result; set otherwise, indicating a "borrow"
Z: Set if the result is zero; cleared otherwise
S: Set if the result is negative; cleared otherwise
V: Set if arithmetic overflow occurs, that is, if both operands were of opposite signs and the sign of the result is the same as the sign of the source; cleared otherwise
D: Unaffected
H: Unaffected

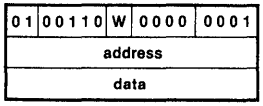
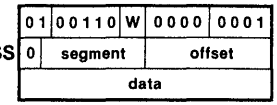
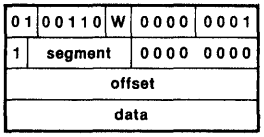
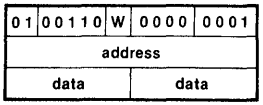
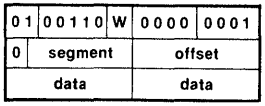
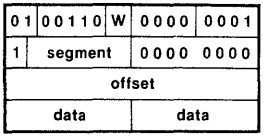
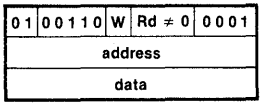
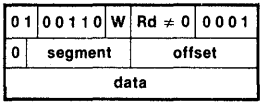
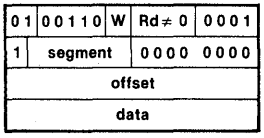
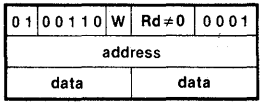
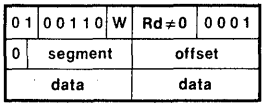
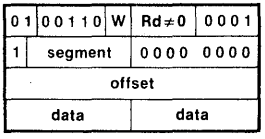
Compare Register

Source Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
R:	CP Rd, Rs CPB Rbd, Rbs		4		4
	CPL RRd, RRs		8		8
IM:	CP Rd, #data		7		7
	CPB Rbd, #data		7		7
	CPL RRd, #data		14		14
IR:	CP Rd, @Rs[1] CPB Rbd, @Rs[1]		7		7
	CPL RRd, @Rs[1]		14		14

Source Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
DA:	CP Rd, address CPB Rbd, address		9	SS 	10
				SL 	12
	CPL RRd, address		15	SS 	16
				SL 	18
X:	CP Rd, addr(Rs) CPB Rbd, addr(Rbs)		10	SS 	10
				SL 	13
	CPL RRd, addr(Rs)		16	SS 	16
				SL 	19

Compare Immediate

Destination Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
IR:	CP @Rd[1], #data		11		11
	CPB @Rd[1], #data		11		11

Destination Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
DA:	CP address, #data		14		15
					17
	CPB address, #data		14		15
					17
X:	CP addr(Rd), #data		15		15
					18
	CPB addr(Rd), #data		15		15
					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 — dst: IR
CPDB — src: IR

Operation:

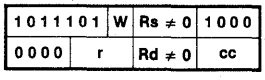
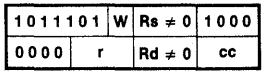
dst – src
AUTODECREMENT src (by 1 if byte, by 2 if word)
r ← r – 1

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 6.6.1 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.

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 CPDB Rbd, @Rs[1], r, cc		20		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 dst, src, r, cc — dst: IR
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

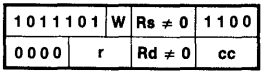
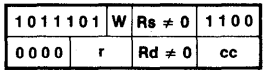
This instruction is used to search a string of data for an element meeting the specified condition. The contents of the location addressed by the source register are compared to (subtracted from) the destination operand, and the Z flag is set if the condition code specified by "cc" would be set by the 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 "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).

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 CPDRB Rbd, @Rs[1], r, cc		11+9n		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 3, 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 5. 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:

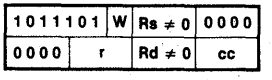
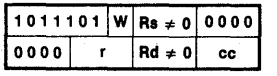
dst – src
AUTOINCREMENT src (by 1 if byte; by 2 if word)
r ← r - 1

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 is specified by "cc" would be set by the comparison; otherwise the Z flag is cleared. See section 6.6.1 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 source, destination, and counter registers must be separate and non-overlapping registers. The word register specified by "r" (used as a counter) is then decremented by one.

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 CPIB Rbd, @Rs[1], r, cc		20		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!
        LD      RL0,#'9'             !largest numeric char!
LOOP:
        CPB     @R1,#'0'             !test char < '0'!
        JR      ULT,NONNUMERIC
        CPIB    RL0, @R1, R3, ULT    !test char > '0'!
        JR      Z, 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

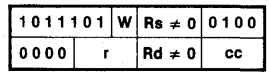
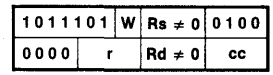
This instruction is used to search a string of data for an element meeting the specified condition. The contents of the location addressed by the source register are compared to (subtracted from) the destination operand, and the Z flag is set if the condition code specified by "cc" would be set by the 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 CPIRB Rbd,@Rs[1], r, cc		11+9n		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:

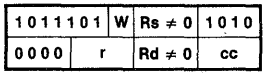
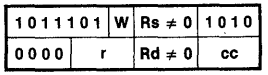
dst – src
AUTODECREMENT dst and src (by 1 if byte; by 2 if word)
r ← r – 1

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.

Flags:

C: Cleard 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 CPSDB @Rd[1],@Rs[1],r,cc		25		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
CPSDRB

dst: IR
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

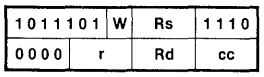
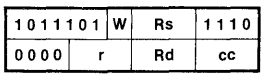
This instruction is used to compare two strings of data until the specified condition is true. The contents of the location addressed by the source register are compared to (subtracted from) the contents of the location addressed by the destination register. The Z flag is set if the condition code specified 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).

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 CPSDRB@Rd[1],@Rs[1],r,cc		11 + 14n		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 r was decremented to 0. R13 will contain %0FFE, R14 will contain %1FFE, 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 dst, src, r, cc — dst: IR
CPSIB — src: IR

Operation:

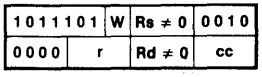
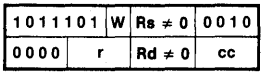
dst − src
AUTOINCREMENT dst and src (by 1 if byte, by 2 if word)
r ← r − 1

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 6.6.1 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.

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

Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
IR:	CPSI @Rd[1],@Rs[1],r,cc CPSIB @Rd[1],@Rs[1],r,cc		25		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 6.6.1 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).

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. The source, destination, and counter 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 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 CPSIRB @Rd[1],@Rs[1],r,cc	1011101 | W | Rs ≠ 0 | 0110 0000 | r | Rd ≠ 0 | cc	11 + 14n	1011101 | W | Rs ≠ 0 | 0110 0000 | r | Rd ≠ 0 | cc	11 + 14n

Example: The CPSIR instruction can be used to compare test strings for lexicographic order. (For most common character encoding — for example, ASCII and EBCDIC — lexicographic order is the same as alphabetic order for alphabetic test 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 ascending lexicographic order:

A
A ␣ A
A B C
A B CD
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 dst dst: R

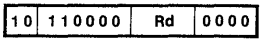
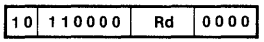
Operation: dst ← DA dst

The destination byte is adjusted to form two 4-bit BCD digits following an addition or subtraction operation. 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
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 valid 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		5		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 dst, src
DECB

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

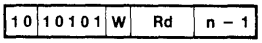
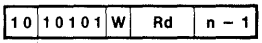
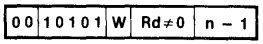
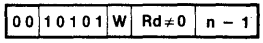
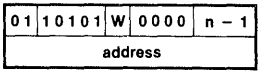
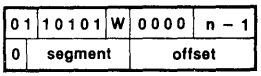
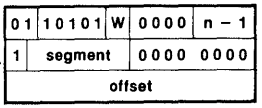
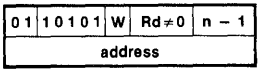
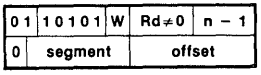
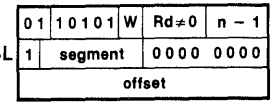
Operation: dst ← dst − src (where src = 1 to 16)

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 DECB Rbd, #n		4		4
IR:	DEC @Rd[1], #n DECB @Rd[1], #n		11		11
DA:	DEC address, #n DECB address, #n		13	SS 	14
				SL 	16
X:	DEC addr(Rd), #n DECB addr(Rd), #n		14	SS 	14
				SL 	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

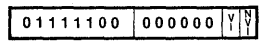
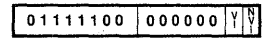
DI Int Int: VI, NVI

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 int		7		7

Example: If the NVI and VI control bits are set (1) in 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 dst, src
DIVL

dst: R
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 (16:31) ← quotient
dst (0:15) ← 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 (32:63) ← quotient
dst (0:31) ← 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.

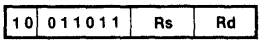
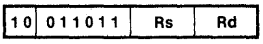
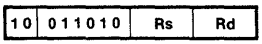
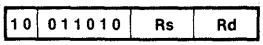
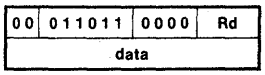
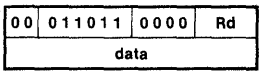
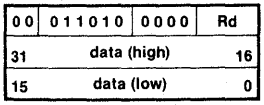
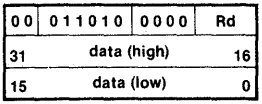
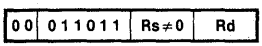
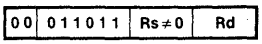
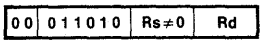
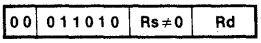
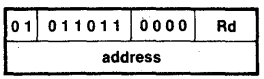
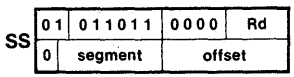
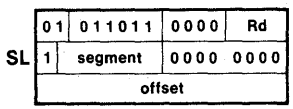
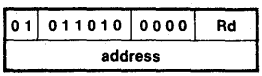
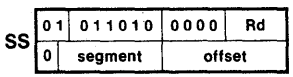
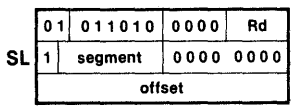
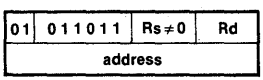
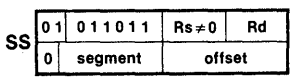
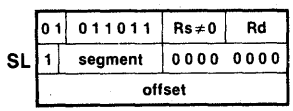
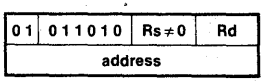
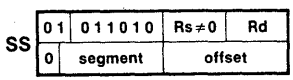
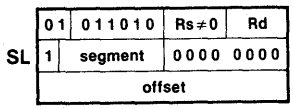
CASE 2. If the divisor is zero, the destination register remains unchanged, the overflow and zero flags are set to one and the carry and sign flags are cleared to zero.

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				
	DIVL RQd, RRs				
IM:	DIV RRd, #data				
	DIVL RQd, #data				
IR:	DIV RRd, @Rs[1]				
	DIVL RQd, @Rs[1]				
DA:	DIV RRd, address				
					
	DIVL RQD, address				
					
X:	DIV RRd, addr(Rs)				
					
	DIVL RQd, addr(Rs)				
					

Note 1: Word register in nonsegmented mode, register pair in segmented mode.
Note 2: Execution times for each instruction are given in the table under Example.

Example: The following table gives the DIV instruction execution times for word and long word operands in all possible addressing modes.

src	Word			Long Word		
	NS	SS	SL	NS	SS	SL
R	107	—	—	744	—	—
IM	107	—	—	744	—	—
IR	107	107	107	744	744	744
DA	108	108	111	745	746	748
X	109	109	112	746	746	749
(Divisor is zero)						
R	13	13	13	30	30	30
IM	13	13	13	30	30	30
IR	13	13	13	30	30	30
DA	14	15	17	31	32	34
X	15	15	18	32	32	35
(Absolute value of the high-order half of the dividend is larger than the absolute value of the divisor)						
R	25	25	25	51	51	51
IM	25	25	25	51	51	51
IR	25	25	25	51	51	51
DA	26	27	29	52	53	55
X	27	27	30	53	53	56

Note that for proper execution, the "dst field" in the instruction format encoding must be even for DIV, and must be a multiple of 4 (0, 4, 8, 12) for DIVL. If the source operand in DIVL is a register, the "src field" must be even.

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 than indicated for divide by zero and certain overflow conditions.

DJNZ
Decrement and Jump if Not Zero

DJNZ R, dst
DBJNZ dst: RA

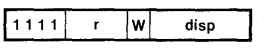
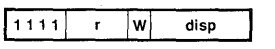
Operation:

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 byte 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. 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, displacement DBJNZ Rb, displacement		11		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 for register pairs.

EI
Enable Interrupts

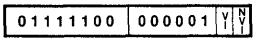
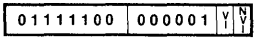
EI int Int: VI, NVI

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		7		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
EXB

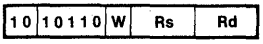
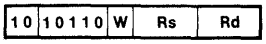
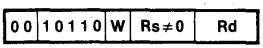
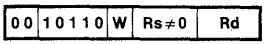
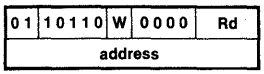
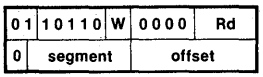
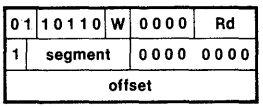
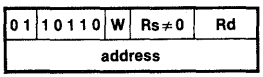
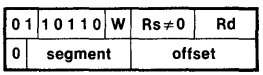
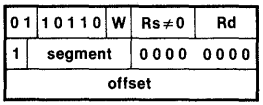
dst: R
src: R, IR, DA, X

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 EXB Rbd, Rbs		6		6
IR:	EX Rd, @Rs[1] EXB Rbd, @Rs[1]		12		12
DA:	EX Rd, address EXB Rbd, address		15	SS 	16
				SL 	18
X:	EX Rd, addr(Rs) EXB Rbd, addr(Rs)		16	SS 	16
				SL 	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.

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

EXTSB dst
EXTS
EXTSL

dst: R

Operation:

Byte
if dst (7) = 0 then dst (8:15) ← 000...000
else dst (8:15) ← 111...111

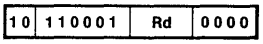
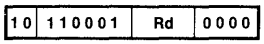
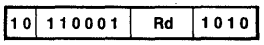
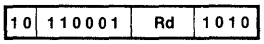
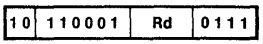
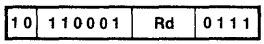
Word
if dst (15) = 0 then dst (16:31) ← 000...000
else dst (16:31) ← 111...111

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 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		11		11
	EXTS RRD		11		11
	EXTSL RQd		11		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

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

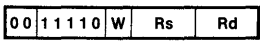
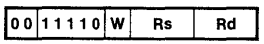
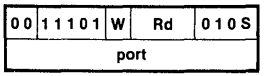
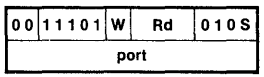
Note 1: Interrupts are recognized at the end of each 3-cycle period; thus n = number of periods without interruption.

IN dst, src **INB**	dst: R src: IR, DA
SIN dst, src **SINB**	dst: R 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 normal 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	Cycles	Instruction Format	Cycles
IR:	IN Rd[1], @Rs INB Rbd[1], @Rs		10		10
DA:	IN Rd, port INB Rbd, port SIN Rd, port SINB Rbd, port		12		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. Word register in nonsegmented mode; register pair in segmented mode.

INC
Increment

INC dst, src
INCB

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

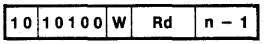
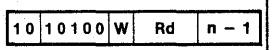
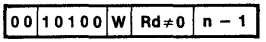
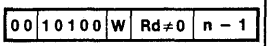
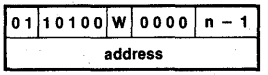
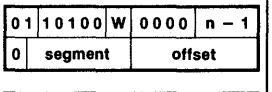
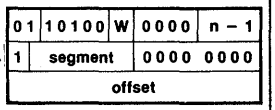
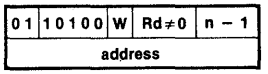
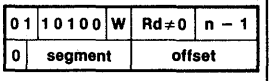
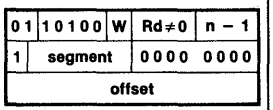
Operation: dst ← dst + src (src = 1 to 16)

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 INCB Rbd, #n		4		4
IR:	INC @Rd[1], #n INCB @Rd[1], #n		11		11
DA:	INC address, #n INCB address, #n		13	SS 	14
				SL 	16
X:	INC addr(Rd), #n INCB addr(Rd), #n		14	SS 	14
				SL 	17

Example: If register RH2 contains %21, the statement

```
INCB  RH2,#6
```

will leave the value %27 in RH2.

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

IND (SIND)

(Special) Input and Decrement

IND dst, src, r — dst: IR
INDB — src: IR
SIND
SINDB

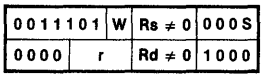
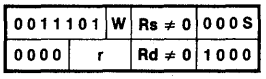
Operation:

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 INDB @Rd[1], @Rs, r SIND @Rd[1], @Rs, r SINDB @Rd[1], @Rs, r		21		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.

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

INDR (SINDR)

Privileged Instruction

(Special) Input, Decrement and Repeat

INDR dst, src, r
INDRB
SINDR
SINDRB

dst: IR
src: IR

Operation:

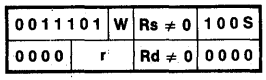
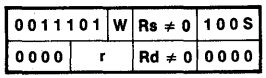
dst ← src
AUTODECREMENT dst (by 1 if byte, by 2 if word)
r ← r − 1
repeat until r = 0

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 INDRB @Rd[1], @Rs, r SINDR @Rd[1], @Rs, r SINDRB @Rd[1], @Rs, r		11 + 10n		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.

INI (SINI)

Privileged Instruction

(Special) Input and Increment

INI dst, src, r
INIB
SINI
SINIB

dst: IR
src: IR

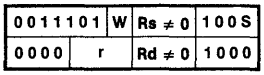
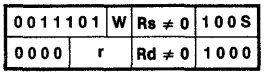
Operation:

dst ← src
AUTOINCREMENT dst (by 1 if byte, by 2 if word)
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
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:	INI @Rd[1], @Rs, r INIB @Rd[1], @Rs, r SINI @Rd[1], @Rs, r SINIB @Rd[1], @Rs, r		21		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.

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

(Special) Input, Increment and Repeat

INIR dst, src, r
INIRB
SINIR
SINIRB

dst: IR
src: IR

Operation:

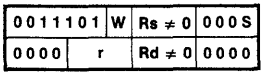
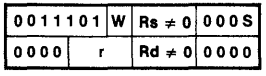
dst ← src
AUTOINCREMENT dst (by 1 if byte, by 2 if word)
r ← r − 1
repeat until r = 0

This instruction is used for block input of strings of data. INIR and INIRB are used for normal 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.

Flags:

C: Unaffected
Z: Unaffected
S: Unaffected
V: Set
D: Unaffected
H: Unaffected

Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
IR:	INIR @Rd[1], @Rs, r INIRB @Rd[1], @Rs, r SINIR @Rd[1], @Rs, r SINIRB @Rd[1], @Rs, r		11 + 10n		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 %0051 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.

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 processor stack and discarded. Then contents of the location addressed by the system processor 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, the operation of IRET in nonsegmented mode is undefined. A Z8001 must be in segmented mode when an IRET instruction is performed.

Flags:

C: Loaded from processor stack
Z: Loaded from processor stack
S: Loaded from processor stack
P/V: Loaded from processor stack
D: Loaded from processor stack
H: Loaded from processor 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 | Rd≠0 | cc	15/7
DA:	JP cc, address	01 | 011110 | 0000 | cc address	7/7	SS: 01 | 011110 | 0000 | cc 0 | segment | offset	8/8
				SL: 01 | 011110 | 0000 | cc 1 | segment | 0000 0000 offset	10/10
X:	JP cc, addr(Rd)	01 | 011110 | Rd≠0 | cc address	8/8	SS: 01 | 011110 | Rd≠0 | cc 0 | segment | offset	11/11
				SL: 01 | 011110 | Rd≠0 | cc 1 | segment | 0000 0000 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.

Note 1: Word register in nonsegmented mode, register pair in segmented mode.
Note 2: The two values correspond to jump taken and jump not taken.

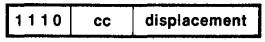
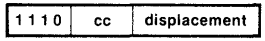
JR cc, dst dst: RA

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 6.6.1 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 to derive the destination address. The updated PC value is taken to be the address of the instruction byte 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		6		6

Example: If the result of the last arithmetic operation executed is negative, the following 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
LDB
LDL

dst: R
src: R, IR, DA, X, BA, BX

or
dst: IR, DA, X, BA, BX
src: R
or
dst: R, IR, DA, X
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.

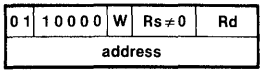
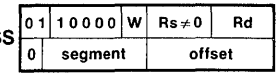
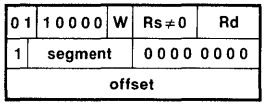
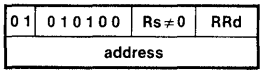
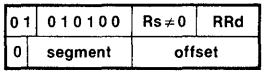
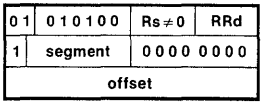
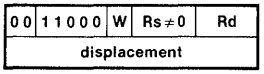
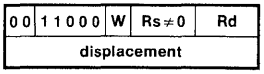
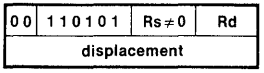
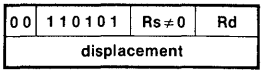
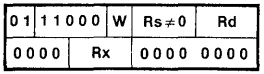
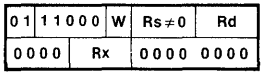
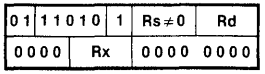
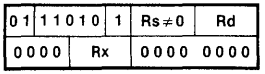
Flags: No flags affected

Load Register

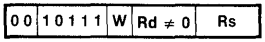
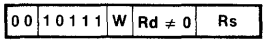
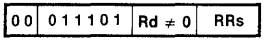
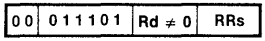
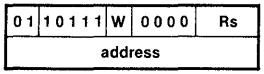
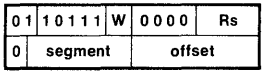
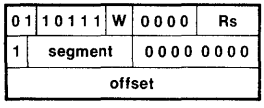
Source Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
R:	LD Rd, Rs LDB Rbd, Rbs	10 | 10000 | W | Rs | Rd	3	10 | 10000 | W | Rs | Rd	3
	LDL RRd, RRs	10 | 010100 | RRs | RRd	5	10 | 010100 | RRs | RRd	5
IR:	LD Rd, @Rs[1] LDB Rbd, @Rs[1]	00 | 10000 | W | Rs≠0 | Rd	7	00 | 10000 | W | Rs≠0 | Rd	7
	LDL RRd, @Rs[1]	00 | 010100 | Rs≠0 | RRd	11	00 | 010100 | Rs≠0 | RRd	11
DA:	LD Rd, address LDB Rbd, address	01 | 10000 | W | 0000 | Rd address	9	SS: 01 | 10000 | W | 0000 | Rd 0 | segment | offset	10
				SL: 01 | 10000 | W | 0000 | Rd 1 | segment | 0000 0000 offset	12
	LDL RRd, address	01 | 010100 | 0000 | RRd address	12	SS: 01 | 010100 | 0000 | RRd 0 | segment | offset	13
				SL: 01 | 010100 | 0000 | RRd 1 | segment | 0000 0000 offset	15

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

Load Register (Continued)

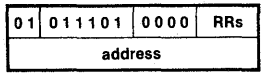
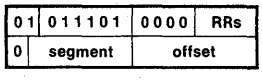
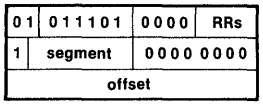
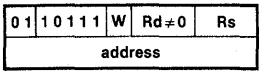
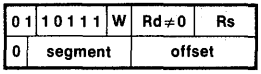
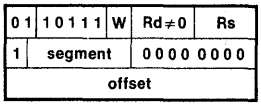
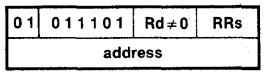
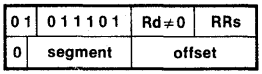
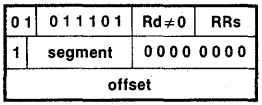
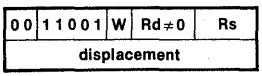
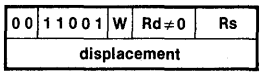
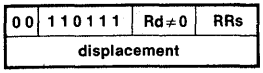
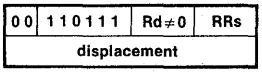
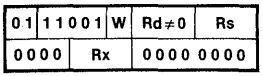
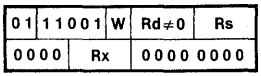
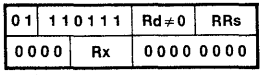
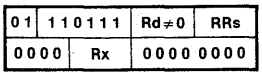
Source Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
X:	LD Rd, addr(Rs) LDB Rbd, addr(Rs)		10	SS 	10
				SL 	13
	LDL RRd, addr(Rs)		13	SS 	13
				SL 	16
BA:	LD Rd, Rs[1](#disp) LDB Rbd, Rs[1](#disp)		14		14
	LDL RRd, Rs[1](#disp)		17		17
BX:	LD Rd, Rs[1](Rx) LDB Rd, Rs[1](Rx)		14		14
	LDL RRd, Rs[1](Rx)		17		17

Load Memory

Destination Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
IR:	LD @Rd[1], Rs LDB @Rd[1], Rbs		8		8
	LDL @Rd[1], RRs		11		11
DA:	LD address, Rs LDB address, Rbs		11	SS 	12
				SL 	14

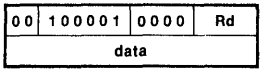
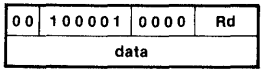
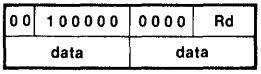
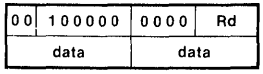
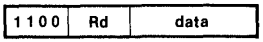
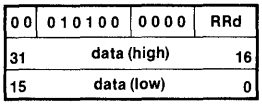
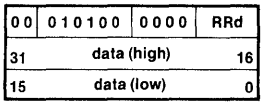
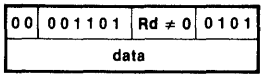
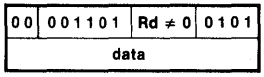
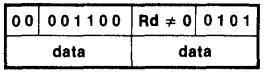
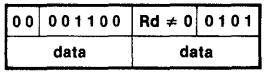
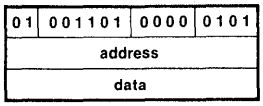
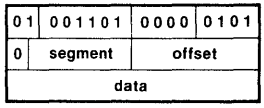
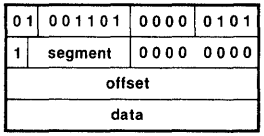
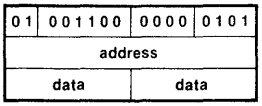
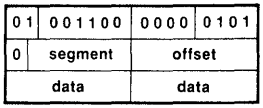
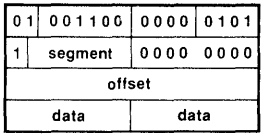
Note 1: Word register in nonsegmented mode, register pair in segmented mode.

Load Memory (Continued)

Destination Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
DA:	LDL address, RRs		14	SS 	15
				SL 	17
X:	LD addr(Rd), Rs LDB addr(Rd), Rbs		12	SS 	12
				SL 	15
	LDR addr(Rd), RRs		15	SS 	15
				SL 	18
BA:	LD Rd[1](#disp), Rs LDB Rd[1](#disp), Rbs		14		14
	LDL Rd[1](#disp), RRs		17		17
BX:	LD Rd[1](Rx), Rs LDB Rd[1](Rx), Rbs		14		14
	LDL Rd[1](Rx), RRs		17		17

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

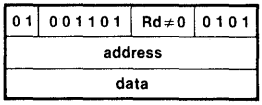
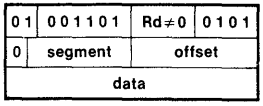
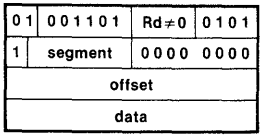
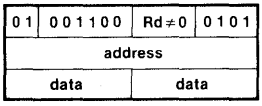
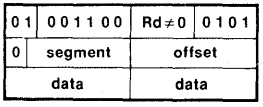
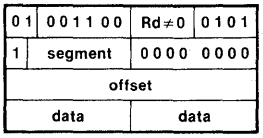
Load Immediate Value

Destination Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
R:	LD Rd, #data		7		7
	LDB Rbd, #data[2]		7		7
			5		5
	LDL RRd, #data		11		11
IR:	LD @Rd[1], #data		11		11
	LDB @Rd[1], #data		11		11
DA:	LD address, #data		14		15
					17
	LDB address, #data		14		15
					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		15	**SS** 	15
				SL 	18
	LDB addr(Rd), #data		15	**SS** 	15
				SL 	18

Example: Several examples of the use of the Load instruction are treated in detail in Chapter 4 under addressing modes.

LDA dst, src

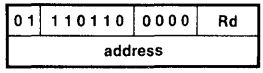
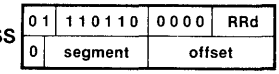
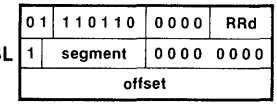
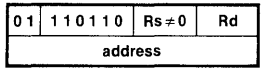
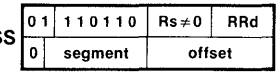
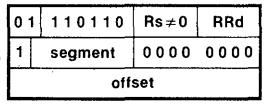
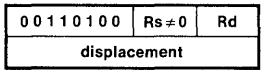
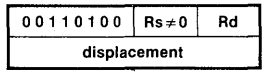
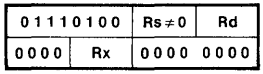
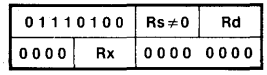
dst: R
src: DA, X, BA, BX

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 based 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		12	SS 	13
				SL 	15
X:	LDA Rd[1], addr(Rs)		13	SS 	13
				SL 	16
BA:	LDA Rd[1], Rs[1] (#disp)		15		15
BX:	LDA Rd[1], Rs[1] (Rx)		15		15

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 (<<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 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 byte 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 displacement	15	00110100 | 0000 | Rd 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

LDCTL dst, src

dst: CTLR
src: R
or
dst: R
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:

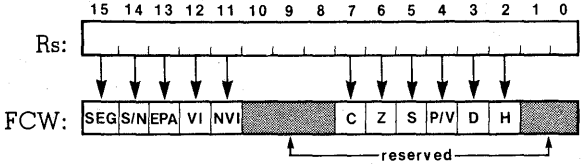
FCW	Flag and Control Word
REFRESH	Refresh Control
PSAPSEG	Program Status Area Pointer - segment number
PSAPOFF	Program Status Area Pointer - offset
NSPSEG	Normal Stack Pointer - segment number
NSPOFF	Normal Stack Pointer - offset

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)

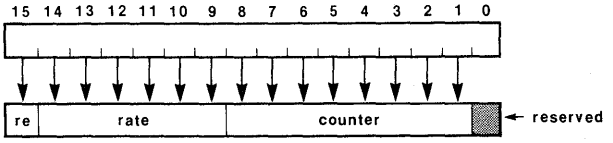


LDCTL REFRESH, Rs

Operation: REFRESH (1:15) ← Rs (1:15)

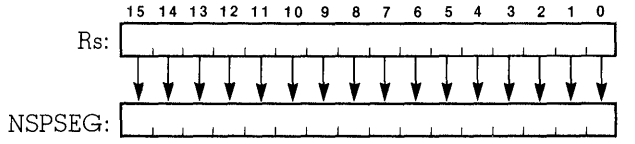
Rs:

REFRESH:



LDCTL NSPSEG, Rs

Operation: NSPSEG (0:15) ← Rs (0:15)

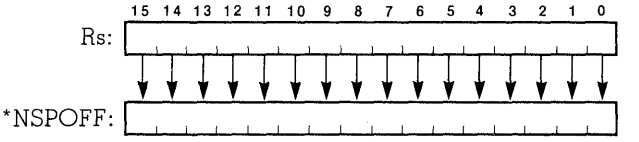


In segmented mode, the NSPSEG register is the normal mode R14 and contains the segment number of the normal mode processor stack pointer which is otherwise inaccessible for system mode.

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)



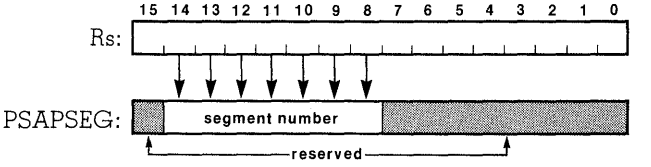
***NSP in nonsegmented mode**

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 Z8002, the mnemonic "NSP" should be used in the assembly language statement, and indicates the same control register as the mnemonic "NSPOFF".

LDCTL PSAPSEG, Rs

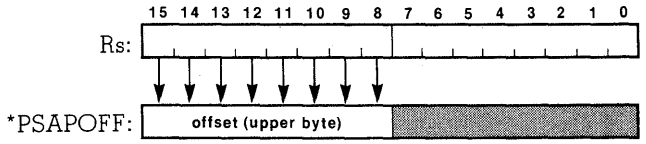
Operation: PSAPSEG (8:14) ← Rs (8:14)



The PSAPSEG register may not be used in the nonsegmented Z8002. In the segmented Z8001, 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)



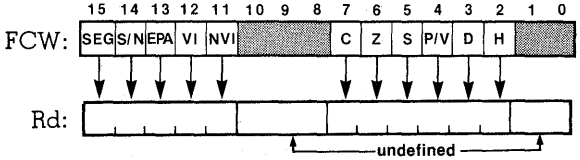
***PSAP in nonsegmented mode**

In the nonsegmented Z8002, the mnemonic "PSAP" should be used in the assembly language statement and indicates the same control register as the mnemonic "PSAPOFF". In the segmented Z8001, 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

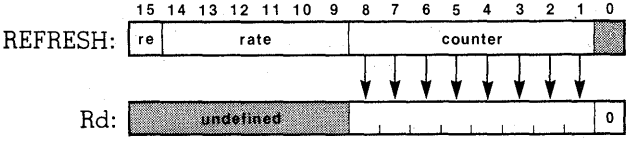
LDCTL Rd, FCW

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

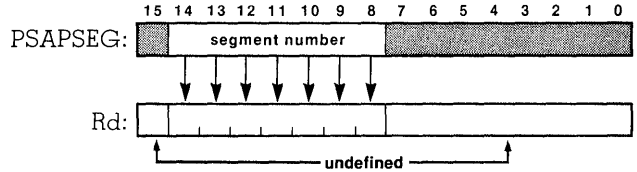
Operation:
Rd (1:8) ← REFRESH (1:8)
Rd (0) ← UNDEFINED
Rd (9:15) ← UNDEFINED



LDCTL Rd, PSAPSEG

Operation:

Rd (8:14) ← PSAPSEG (8:14)
Rd (0:7) ← UNDEFINED
Rd (15) ← UNDEFINED

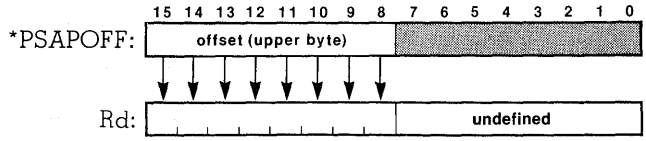


This instruction may not be used in the nonsegmented version.

LDCTL Rd, PSAPOFF
Rd, PSAP

Operation:

Rd (8:15) ← PSAPOFF (8:15)
Rd (0:7) ← UNDEFINED



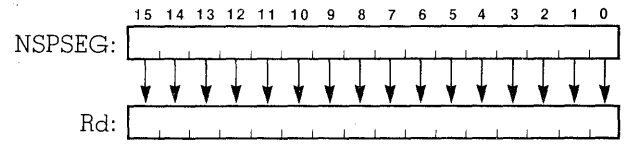
***PSAP in nonsegmented mode**

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:

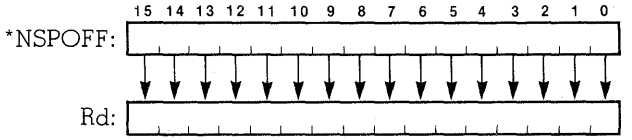
Rd (0:15) ← NSPSEG (0:15)



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 PSAP, Rs	01111101 | Rs | 1101	7	01111101 | Rs | 1101	7
	LDCTL NSPSEG, Rs			01111101 | Rs | 1110	7
	LDCTL NSPOFF, Rs 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 LDCTL Rd, PSAP	01111101 | Rd | 0101	7	01111101 | Rd | 0101	7
	LDCTL Rd, NSPSEG			01111101 | Rd | 0110	7
	LDCTL Rd, NSPOFF Rd, NSP	01111101 | Rd | 0111	7	01111101 | Rd | 0111	7

LDCTLB dst, src

dst: FLAGS
src: R
or
dst: R
src: FLAGS

Operation: dst ← src

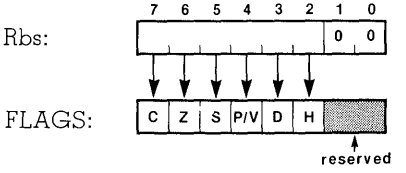
This instruction is used to load the FLAGS register or to transfer its contents into a general-purpose register. Note that this is not a privileged instruction.

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.

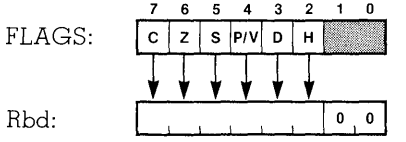


Load From FLAGS Register

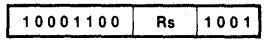
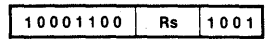
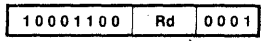
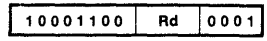
LDCTLB Rbd, FLAGS

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		7		7
	LDCTLB Rbd, FLAGS		7		7

LDD dst, src, r
LDDB

dst: IR
src: IR

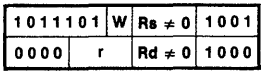
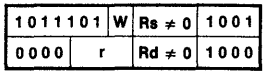
Operation:

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 source destination, and counter registers must be separate and non-overlapping registers. The word register specified by "r" (used as a counter) is then decremented by one.

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 LDDB @Rs[1], @Rd[1], r		20		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.

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

LDDR
Load, Decrement and Repeat

LDDR dst, src, r
LDDRB

dst: IR
src: IR

Operation:

dst ← src
AUTODECREMENT dst and src (by 1 if byte, by 2 if word)
r ← r − 1
repeat until r = 0

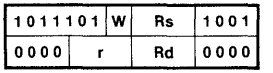
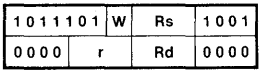
This instruction is used for block transfers of strings of data. The contents of the location addressed by the source register are loaded into the location addressed by the destination register. The source and destination registers are then decremented by one if LDDRB, or by two if LDDR, thus moving 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
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:	LDDR @Rd[1], @Rs[1], r LDDRB @Rd[1], @Rs[1], r		11 + 9n		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
LDIB

dst: IR
src: IR

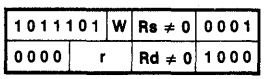
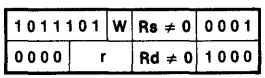
Operation:

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

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:	LDI @Rd[1], @Rs[1], r LDIB @Rd[1], @Rs[1], r		20		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:
```

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

LDIR
Load, Increment and Repeat

LDIR dst, src, r — dst: IR
LDIRB — 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 LDIRB @Rd[1], @Rs[1], r	1011101 | W | Rs ≠ 0 | 0001 0000 | r | Rd ≠ 0 | 0000	11 + 9n	1011101 | W | Rs ≠ 0 | 0001 0000 | r | Rd ≠ 0 | 0000	11 + 9n

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.

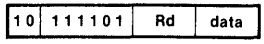
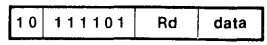
LDK dst, src

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		5		5

Example: To load register R3 with the constant 9:

```
LDK  R3,#9
```

LDM
Load Multiple

LDM dst, src, n

dst: R
src: IR, DA, X
or
dst: IR, DA, X
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.

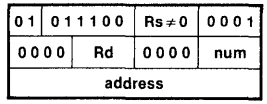
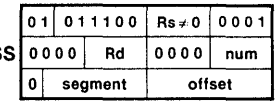
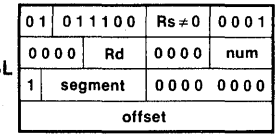
The starting address is computed once at the start of execution, and incremented by two for each register loaded. If the original address computation involved a register, the register's value will not be affected by the address incrementation during execution. Similarly, modifying that register during a load from memory will not affect the address used by this instruction.

Flags: No flags affected

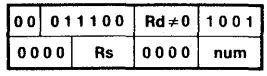
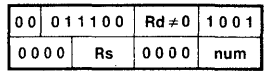
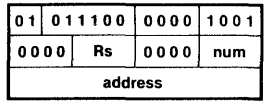
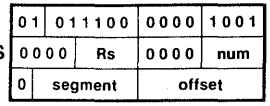
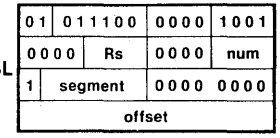
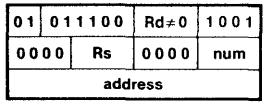
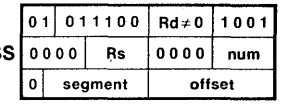
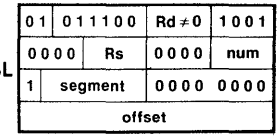
Load Multiple - Registers From Memory

Source Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode		
		Instruction Format	Cycles		Instruction Format	Cycles
IR:	LDM Rd, @Rs[1], #n	00 | 011100 | Rs≠0 | 0001 0000 | Rd | 0000 | num	11+3n		00 | 011100 | Rs≠0 | 0001 0000 | Rd | 0000 | num	11+3n
DA:	LDM Rd, address, #n	01 | 011100 | 0000 | 0001 0000 | Rd | 0000 | num address	14+3n	**SS**	01 | 011100 | 0000 | 0001 0000 | Rd | 0000 | num 0 | segment | offset	15+3n
				SL	01 | 011100 | 0000 | 0001 0000 | Rd | 0000 | num 1 | segment | 0000 0000 offset	17+3n

Load Multiple - Registers From Memory (Continued)

Source Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode		
		Instruction Format	Cycles	Instruction Format		Cycles
X:	LDM Rd, addr(Rs), #n		15 + 3n		SS	15 + 3n
					SL	18 + 3n

Load Multiple - Memory From Registers

Destination Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode		
		Instruction Format	Cycles	Instruction Format		Cycles
IR:	LDM@Rd[1], Rs, #n		11 + 3n			11 + 3n
DA:	LDM address, Rs, #n		14 + 3n		SS	15 + 3n
					SL	17 + 3n
X:	LDM addr(Rd), Rs, #n		15 + 3n		SS	15 + 3n
					SL	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.

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

LDPS
Load Program Status

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 the segmented Z8001 version. 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 | Rs≠0 | 0000	16
DA:	LDPS address	01 | 111001 | 0000 | 0000 address	16	SS: 01 | 111001 | 0000 | 0000 0 | segment | offset	20
				SL: 01 | 111001 | 0000 | 0000 1 | segment | 0000 0000 offset	22
X:	LDPS addr(Rs)	01 | 111001 | Rs≠0 | 0000 address	17	SS: 01 | 111001 | Rs≠0 | 0000 0 | segment | offset	20
				SL: 01 | 111001 | Rs≠0 | 0000 1 | segment | 0000 0000 offset	23

Example: In the nonsegmented Z8002 version, 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. Note: Word register is used in nonsegmented mode, register pair in segmented mode.

LDR
Load Relative

LDR dst, src **LDRB** **LDRL**	dst: R src: RA or dst: RA 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 byte 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.

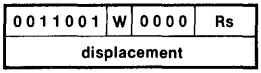
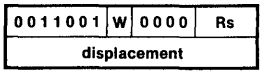
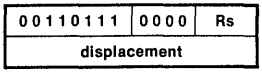
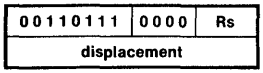
This instruction must be used to modify memory locations containing program information, such as the Program Status Area, if program and data space are allocated to different segments.

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 LDRB Rbd, address	0011000 | W | 0000 | Rd displacement	14	0011000 | W | 0000 | Rd displacement	14
	LDRL RRd, address	00110101 | 0000 | Rd displacement	17	00110101 | 0000 | Rd 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 LDRB address, Rbs		14		14
	LDRL address, RRs		17		17

Example:

```
LDR R2, DATA        !register R2 is loaded with the value in the!
                    !location named DATA!
```

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:
```

MREQ
Multi-Micro Request

MREQ dst dst: R

Operation:

```
Z ← 0
if MI low (active)  then S ← 0
                         MO forced high (inactive)
                    else MO forced low (active)
                         repeat dst ← dst - 1 until dst = 0
                         if MI low (active)  then  S ← 1
                                             else  S ← 0
                                                   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 multi-micro input and output pins (MI and MO), with the S and Z flags indicating the availability of the resource after the MREQ instruction has been executed.

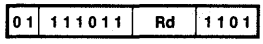
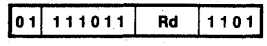
First, the Z flag is cleared. Then the MI pin is tested. If the MI pin is low (active), the S flag is cleared and the MO pin is forced high (inactive),thus indicating that the resource is not available and removing any previous request by the CPU from the MO line.

If the MI pin is high (inactive), indicating that the resource may be available, a sequence of machine operations occurs. First, the 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. Then the MI pin is tested to determine whether the request for the resource was acknowledged. If the MI pin is low (active), the S flag is set to one, indicating that the resource is available and access is granted. If the MI pin is still high (inactive), the S flag is cleared to zero, and the MO pin is forced high (inactive), indicating that the request was not granted and removing the request signal for the MO. Finally, in either case, the Z flag is set to one, indicating that the original test of the MI pin caused a request to be made.

S flag	Z flag	MO	Indicates
0	0	high	Request not signalled (resource not available)
0	1	high	Request not granted (resource not available)
1	1	low	Request granted (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		12+7n		12+7n

Example:

```
TRY:
            LD      R0,#50      !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

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 dst, src
MULTL

dst: R
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

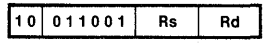
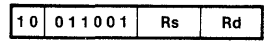
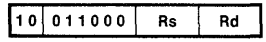
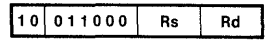
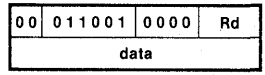
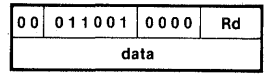
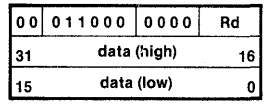
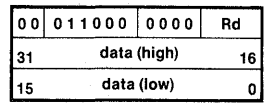
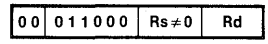
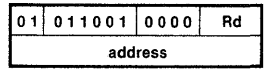
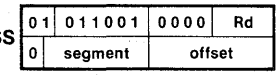
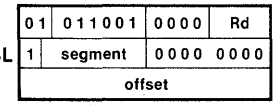
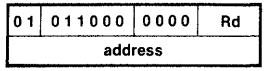
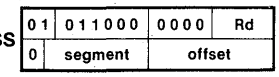
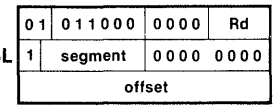
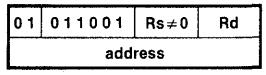
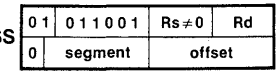
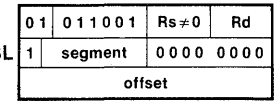
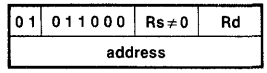
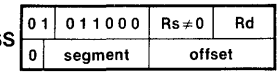
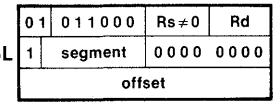
(n = number of bits equal to one in the absolute value of the low-order table 32 bits of the destination operand)

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^{31} 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				
	MULTL RQd, RRs				
IM:	MULT RRd, #data				
	MULTL RQd, #data				
IR:	MULT RRd, @Rs[1]				
	MULTL RQd, @Rs[1]				
DA:	MULT RRd, address			SS  SL 	
	MULTL RQd, address			SS  SL 	
X:	MULT RRd, addr(Rs)			SS  SL 	
	MULTL RQd, addr(Rs)			SS  SL 	

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
NEGB

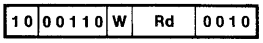
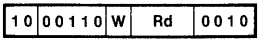
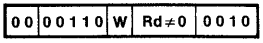
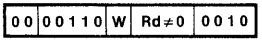
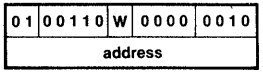
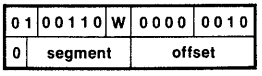
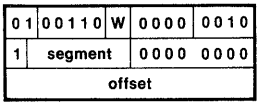
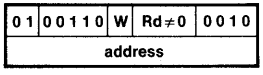
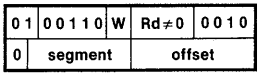
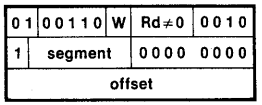
dst: R, IR, DA, X

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 NEGB Rbd		7		7
IR:	NEG @Rd[1] NEGB @Rd[1]		12		12
DA:	NEG address NEGB address		15	SS 	16
				SL 	18
X:	NEG addr(Rd) NEGB addr(Rd)		16	SS 	16
				SL 	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

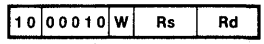
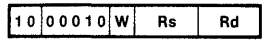
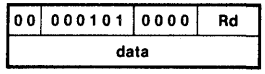
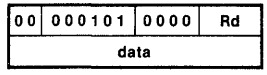
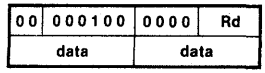
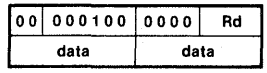
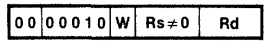
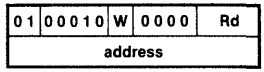
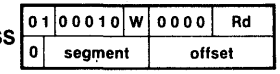
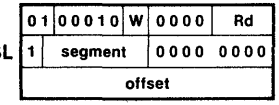
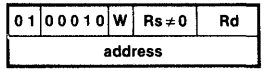
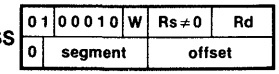
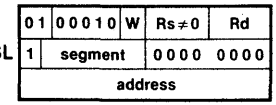
OR dst, src
ORB

dst: R
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 ORB Rbd, Rbs		4		4
IM:	OR Rd, #data		7		7
	ORB Rbd, #data		7		7
IR:	OR Rd, @Rs[1] ORB Rbd, @Rs[1]		7		7
DA:	OR Rd, address ORB Rbd, address		9	SS 	10
				SL 	12
X:	OR Rd, addr(Rs) ORB Rbd, addr(Rs)		10	SS 	10
				SL 	13

Example: If register RL3 contains %C3 (11000011) and the source operand is the immediate value %7B (01111011), the statement

```
ORB   RL3,#%7B
```

will leave the value %FB (11111011) in RL3.

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

OTDR (SOTDR)

Privileged Instruction

(Special) Output, Decrement and Repeat

OTDR dst, src, r
OTDRB
SOTDR
SOTDRB

dst: IR
src: IR

Operation:

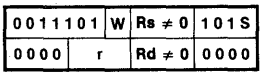
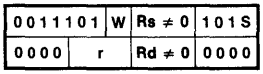
dst ← src
AUTODECREMENT src (by 1 if byte, by 2 if word)
r ← r - 1
repeat until r = 0

This instruction is used for block output of strings of data. OTDR and OTDRB are used for normal 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	Cycles[2]	Instruction Format	Cycles[2]
IR:	OTDR @Rd,@Rs[1], r OTDRB @Rd,@Rs[1], r SOTDR @Rd,@Rs[1], r SOTDRB @Rd,@Rs[1], r		11 + 10n		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.

OTIR (SOTIR)

Privileged Instruction

(Special) Output, Increment and Repeat

OTIR dst, src, r
OTIRB
SOTIR
SOTIRB

dst: IR
src: IR

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 normal 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
Z: Undefined
S: Unaffected
V: Set
D: Unaffected
H: Unaffected

Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
IR:	OTIR @Rd, @Rs[1], r OTIRB @Rd, @Rs[1], r SOTIR @Rd, @Rs[1], r SOTIRB @Rd, @Rs[1], r	0011101 | W | Rs ≠ 0 | 001S 0000 | r | Rd ≠ 0 | 0000	11 + 10n	0011101 | W | Rs ≠ 0 | 001S 0000 | r | Rd ≠ 0 | 0000	11 + 10n

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.

OUT (SOUT)

(Special) Output

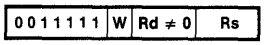
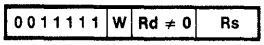
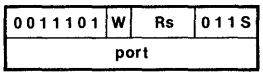
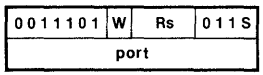
Privileged Instruction

OUT dst, src	dst: IR, DA
OUTB	src: R
SOUT dst, src	dst: DA
SOUTB	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 normal 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	Cycles	Instruction Format	Cycles
IR:	OUT @Rd, Rs OUTB @Rd, Rbs		10		10
DA:	OUT port, Rs OUTB port, Rbs SOUT port, Rs SOUTB port, Rbs		12		12

Example: If register R6 contains %5252, the instruction

```
OUT   %1120, R6
```

will output the value %5252 to the port %1120.

OUTD (SOUTD)

(Special) Output and Decrement

OUTD dst, src, r
OUTDB
SOUTD
SOUTDB

dst: IR
src: IR

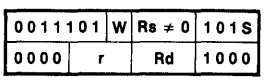
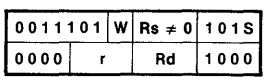
Operation:

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 normal 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	Cycles	Instruction Format	Cycles
IR:	OUTD @Rd, @Rs[1], r OUTDB @Rd, @Rs[1], r SOUTD @Rd, @Rs[1], r SOUTDB @Rd, @Rs[1], r		21		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.

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

OUTI (SOUTI)

Privileged Instruction

(Special) Output and Increment

OUTI dst, src, r
OUTIB
SOUTI
SOUTIB

dst: IR
src: IR

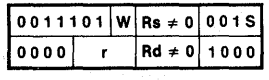
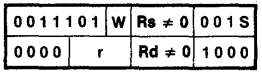
Operation:

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 normal 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	Cycles	Instruction Format	Cycles
IR:	OUTI @Rd, @Rs[1], r OUTIB @Rd, @Rs[1], r SOUTI @Rd, @Rs[1], r SOUTIB @Rd, @Rs[1], r		21		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.

POP
Pop

POP dst, src
POPL

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

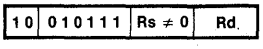
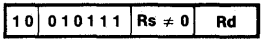
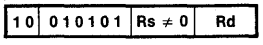
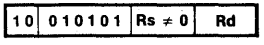
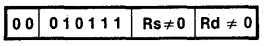
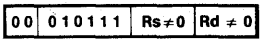
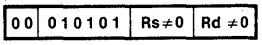
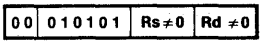
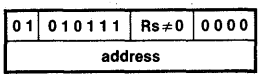
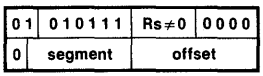
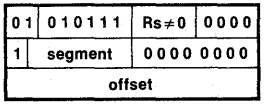
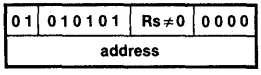
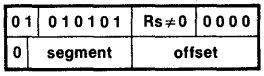
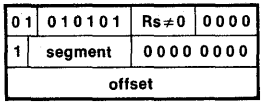
Operation:

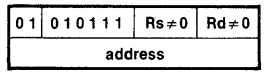
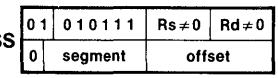
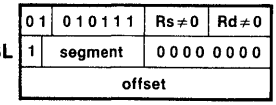
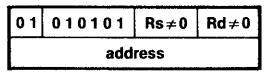
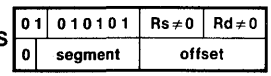
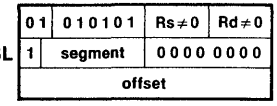
dst ← src
AUTOINCREMENT src (by 2 if word, by 4 if long)

The contents of the location addressed by the source register (a stack pointer) are loaded into the destination. The source register is then incremented by a value which equals the size in bytes of the destination operand, thus removing the top element 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 the POPL instruction, 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]		8		8
	POPL RRd, @Rs[1]		12		12
IR:	POP @Rd[1], @Rs[1]		12		12
	POPL @Rd[1], @Rs[1]		19		19
DA:	POP address, @Rs[1]		16	SS 	16
				SL 	19
	POPL address, @Rs[1]		23	SS 	23
				SL 	26

Destination Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
X:	POP addr(Rd), @Rs[1]		16	SS 	16
				SL 	19
	POPL addr(Rd), @Rs[1]		23	SS 	23
				SL 	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.

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

PUSH
Push

PUSH dst, src
PUSHL

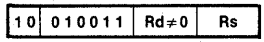
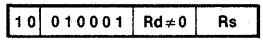
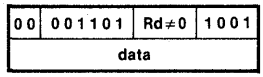
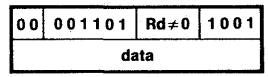
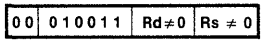
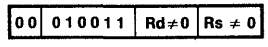
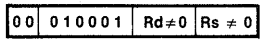
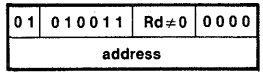
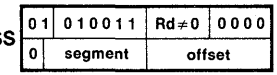
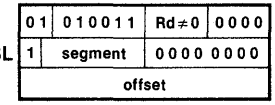
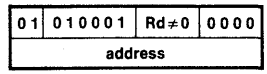
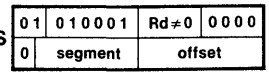
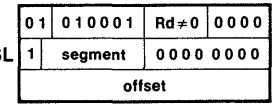
dst: IR
src: R, IM, IR, DA, X

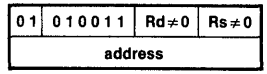
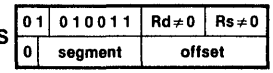
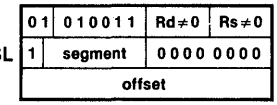
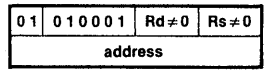
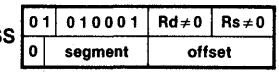
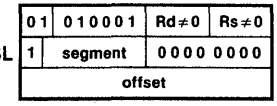
Operation: AUTODECREMENT dst (by 2 if word, by 4 if long)
dst ← src

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
R:	PUSH @Rd[1], Rs		9		9
	PUSHL @Rd[1], RRs		12		12
IM:	PUSH @Rd[1], #data		12		12
IR:	PUSH @Rd[1], @Rs[1]		13		13
	PUSHL @Rd[1], @Rs[1]		20		20
DA:	PUSH @Rd[1], address		14	SS 	14
				SL 	17
	PUSHL@Rd[1], address		21	SS 	13
				SL 	24

Source Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
X:	PUSH @Rd[1], addr(Rs)		14	SS 	14
				SL 	17
	PUSHL @Rd[1], addr(Rs)		21	SS 	21
				SL 	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
RESB

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

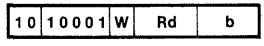
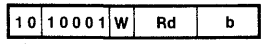
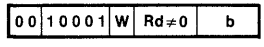
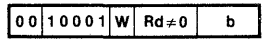
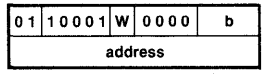
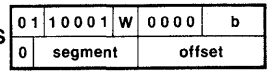
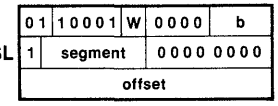
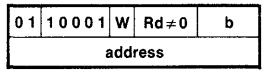
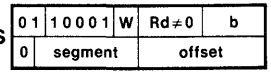
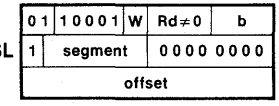
Operation: dst(src) ← 0

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 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.

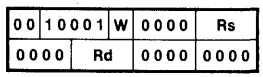
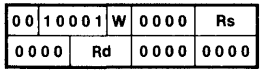
Only the lower four bits of the source operand are used to specify the bit number for RES, while only the lower three bits of the source operand are used with RESB. When the source operand is an immediate value, the "src field" in the instruction format encoding contains the bit number in the lowest 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 RESB Rbd, #b		4		4
IR:	RES @Rd^{1}, #b RESB @Rd^{1}, #b		11		11
DA:	RES address, #b RESB address, #b		13	SS 	14
				SL 	16
X:	RES addr(Rd), #b RESB addr(Rd), #b		14	SS 	14
				SL 	17

Reset Bit Dynamic

Source Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
R:	RES Rd, Rs RESB Rbd, Rs		10		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.

RESFLG
Reset Flag

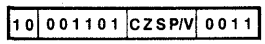
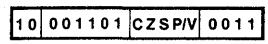
RESFLG flag flag: C, Z, S, P, V

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		7		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 cc

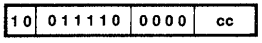
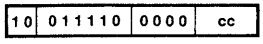
Operation:

Nonsegmented
if cc is true then
PC ← @SP
SP ← SP + 2

Segmented
if cc is true then
PC ← @SP
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		10/7		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).

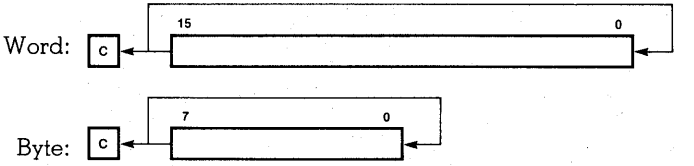
Note 1: The two values correspond to return taken and return not taken.

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)

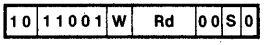
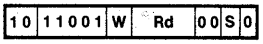


The contents of the destination operand are rotated left one bit position if the source operand is 1, or two bit positions if the source operand is 2. The most significant bit (msb) of the destination operand is moved to the bit 0 position and also replaces the C flag.

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 RLB Rbd, #n		6/7		6/7

Example:

If register RH5 contains %88 (10001000), the statement

```
RLB   RH5
```

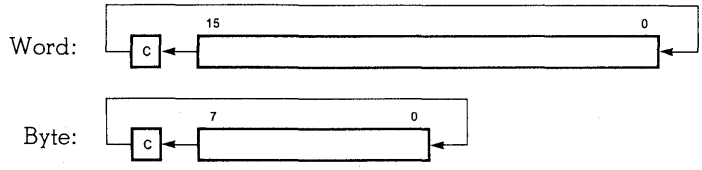
will leave the value %11 (00010001) in RH5 and the Carry flag will be set to one.

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 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



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 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.

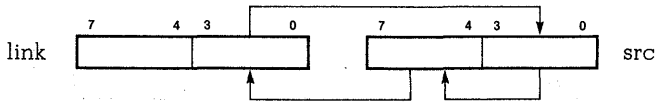
RLDB
Rotate Left Digit

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.

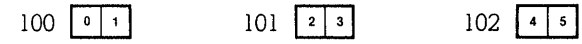
The same byte register must not be used as both the source and the link.

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

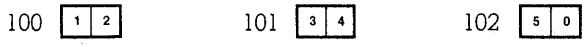
Example: If location 100 contains the BCD digits 0,1 (00000001), location 101 contains 2,3 (00100011), and location 102 contains 4,5 (01000101)



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.



In segmented mode, R2 would be replaced by a register pair.

RR
Rotate Right

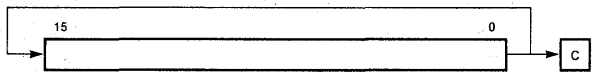
RR dst, src
RRB

dst: R
src: IM

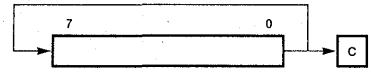
Operation:

Do src times: (src = 1 or 2)
tmp ← dst
c ← tmp (0)
dot (msb) ← tmp (0)
dst (n − 1) ← tmp (n) (for n = 1 to msb)

Word:



Byte:

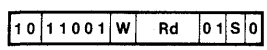
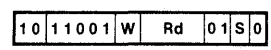


The contents of the destination operand are rotated right one bit position if the source operand is 1, or two bit positions if the source operand is 2. The least significant bit of the destination operand is moved to the most significant bit (msb) and also replaces the C flag.

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 RRB Rbd, #n		6/7		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.

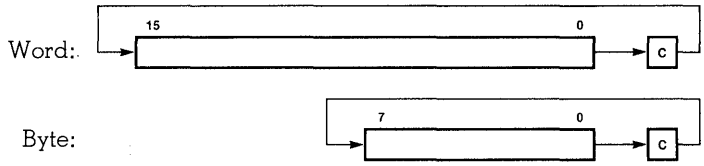
Note 1: s = 0 for rotation by 1 bit; s = 1 for rotation by 2 bits.
Note 2: The given execution times are for rotation by 1 and 2 bits respectively.

RRC dst, src
RRCB

dst: R
src: IM

Operation:

Do src times: (src = 1 or 2)
tmp ← c
c ← dst (0)
dst (n) ← dst (n + 1) (for n = 0 to msb − 1)
dst (msb) ← tmp



The contents of the destination operand with the C flag are rotated 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 replaces the C flag and the previous value of the C flag is moved to the most significant bit (msb) 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 least 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	Nonsegmented Mode		Segmented Mode	
		Instruction Format[1]	Cycles[2]	Instruction Format[1]	Cycles[2]
	RRC Rd, #n RRCB Rbd, #n	10 11001 W Rd 11 S 0	6/7	10 11001 W Rd 11 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 (1000000010111) in R0 and clear the Carry flag.

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.

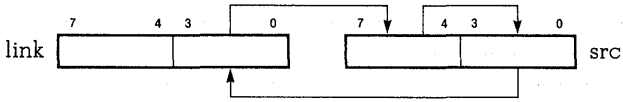
RRDB
Rotate Right Digit

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.

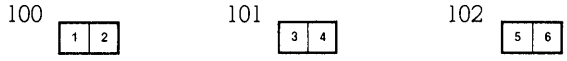
The same byte register must not be used as both the source and the link.

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

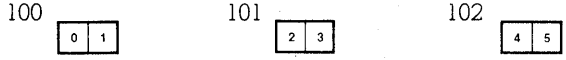
Example: If location 100 contains the BCD digits 1,2 (00010010), location 101 contains 3,4 (00110100), and location 102 contains 5,6 (01010110)



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
SBCB

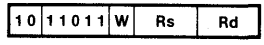
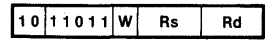
dst: R
src: R

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 SBCB Rbd, Rbs		5		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 src src: IM

Operation:

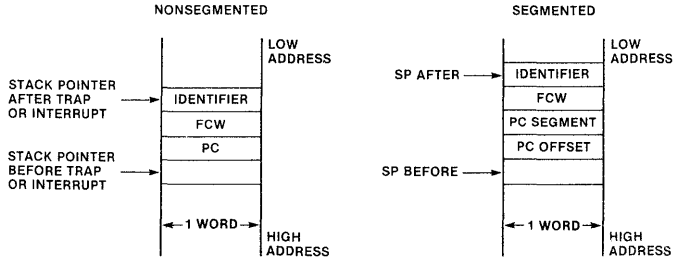
Nonsegmented
SP ← SP − 4
@SP ← PS
SP ← SP − 2
@SP ← instruction
PS ← System Call PS

Segmented
SP ← SP − 6
@SP ← PS
SP ← SP − 2
@SP ← instruction
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 byte following the SC instruction.)

The system stack pointer is always used (R15 in nonsegmented mode, or RR14 in segmented mode), 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 Z8001 version always executes the segmented mode of the System Call instruction, regardless of the current mode, and sets the Segmentation Mode bit (SEG) to segmented mode (= 1) at the start of the SC instruction execution. Both the Z8001 and Z8002 versions 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:

No flags affected
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 %1000, respectively, the instruction

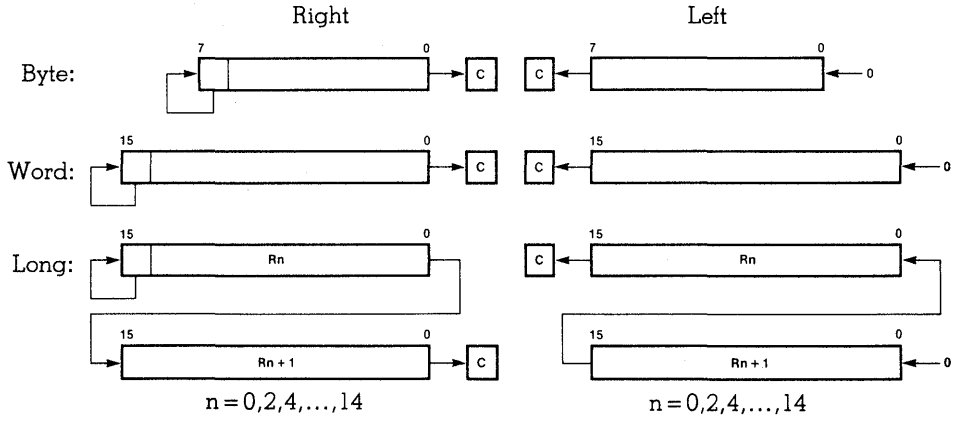
```
SC  #3                    !system call, request code = 3!
```

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 %1000.

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

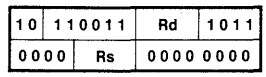
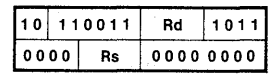
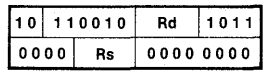
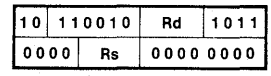
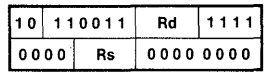
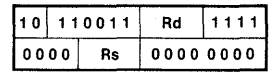


The destination operand is shifted arithmetically left or right by the number of bit positions specified by the contents of the source operand, a word register.

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		15 + 3n		15 + 3n
	SDAB Rbd, Rs		15 + 3n		15 + 3n
	SDAL RRd, Rs		15 + 3n		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.

Note 1: n = number of bit positions; the execution time for n = 0 is the same as for n = 1.

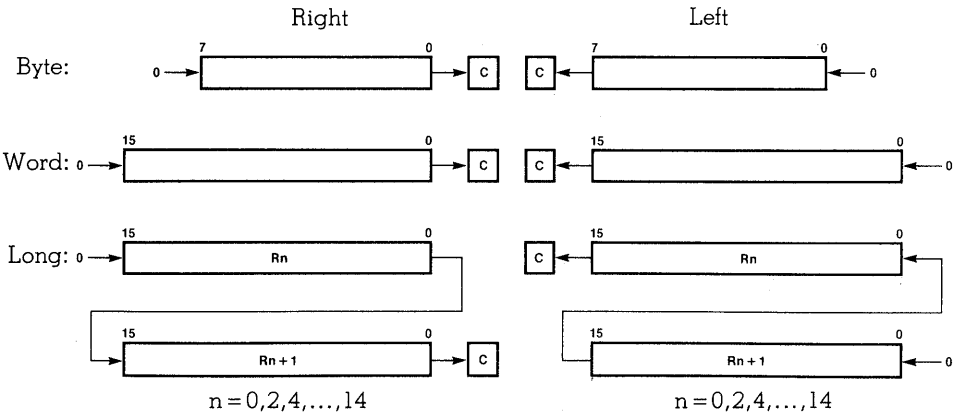
SDL dst, src
SDLB
SDLL

dst: R
src: R

Operation:

Right
Do src times
  c ← dst (0)
  dst (n) ← dst (n + 1) (for n = 0 to msb − 1)
  dst (msb) ← 0

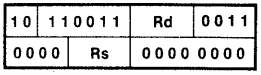
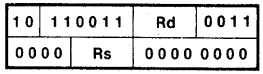
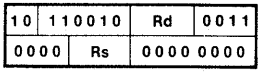
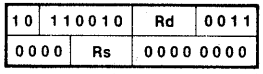
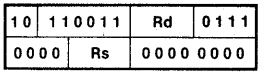
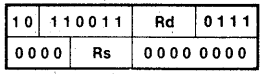
Left
Do src times
  c ← dst (msb)
  dst (n + 1) ← dst (n) (for n = msb − 1 to 0)
  dst (0) ←



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 SDL, from −16 to +16 for SDLB 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		15 + 3n		15 + 3n
	SDLB Rbd, Rs		15 + 3n		15 + 3n
	SDLL RRd, Rs		15 + 3n		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 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.

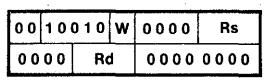
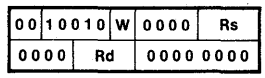
Only the lower four bits of the source operand are used to specify the bit number for SET, while only the lower three bits of the source operand are used with SETB. When the source operand is an immediate value, the "src field" in the instruction format encoding contains the bit number in the lowest 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 SETB Rbd, #b	10 | 10010 | W | Rd | b	4	10 | 10010 | W | Rd | b	4
IR:	SET @Rd[1], #b SETB @Rd[1], #b	00 | 10010 | W | Rd≠0 | b	11	00 | 10010 | W | Rd≠0 | b	11
DA:	SET address, #b SETB address, #b	01 | 10010 | W | 0000 | b address	13	SS: 01 | 10010 | W | 0000 | b 0 | segment | offset	14
				SL: 01 | 10010 | W | 0000 | b 1 | segment | 0000 0000 offset	16
X:	SET addr(Rd), #b SETB addr(Rd), #b	01 | 10010 | W | Rd≠0 | b address	14	SS: 01 | 10010 | W | Rd ≠ 0 | b 0 | segment | offset	14
				SL: 01 | 10010 | W | Rd ≠ 0 | b 1 | segment | 0000 0000 offset	17

Set Bit Dynamic

Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
R:	SET Rd, Rs SETB Rbd, Rs		10		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 1: Word register in nonsegmented mode, register pair in segmented mode.

SETFLG
Set Flag

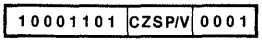
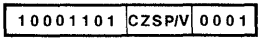
SETFLG flag Flag: C, Z, S, P, V

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		7		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

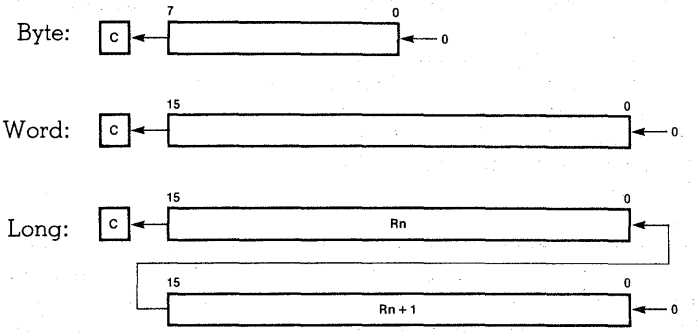
SLA dst, src
SLAB
SLAL

dst: R
src: IM

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

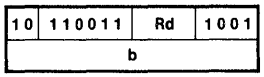
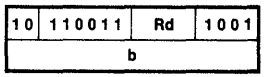
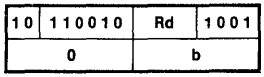
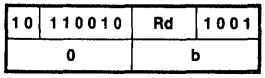
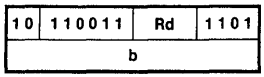
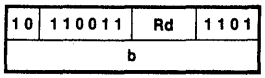
The destination operand is shifted arithmetically left the number of bit positions specified by the source operand. For SLAB, the source is in the range 0 to 8; for SLA, the source is in the range 0 to 16; for SLAL, the source is in the range 0 to 32. A shift of zero positions does not affect the destination; however, the flags are set according to the destination value. 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.

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		13 + 3b		13 + 3b
	SLAB Rbd, #b		13 + 3b		13 + 3b
	SLAL RRd, #b		13 + 3b		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.

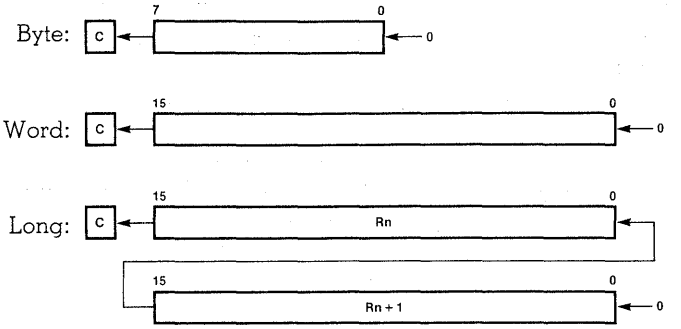
SLL
Shift Left Logical

SLL dst, src
SLB
SLLL

dst: R
src: IM

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

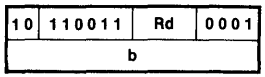
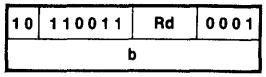
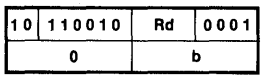
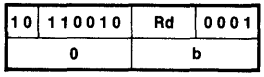
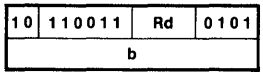
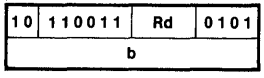
The destination operand is shifted logically left by the number of bit positions specified by the source operand. For SLLB, the source is in the range 0 to 8; for SLL, the source is in the range 0 to 16; for SLLL, the source is in the range 0 to 32. A shift of zero positions does not affect the destination; however, the flags are set according to the destination value. The setting of the carry bit is undefined for zero shift. The 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.

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		13 + 3b		13 + 3b
	SLLB Rbd, #b		13 + 3b		13 + 3b
	SLLL RRd, #b		13 + 3b		13 + 3b

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.

Note 1: b = number of bit positions; the execution time for b = 0 is the same as for b = 1.

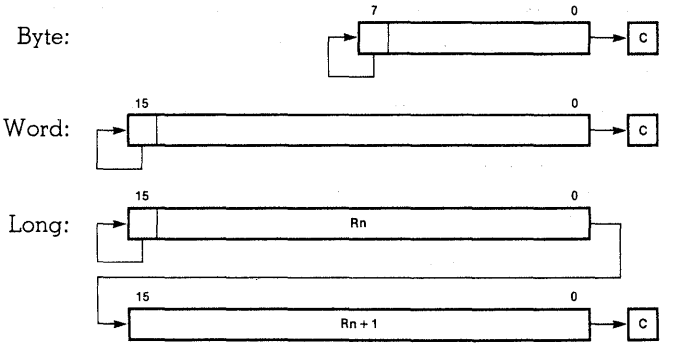
SRA
Shift Right Arithmetic

SRA dst, src
SRAB
SRAL

dst: R
src: IM

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

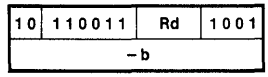
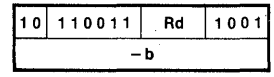
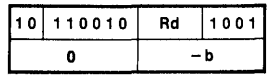
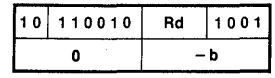
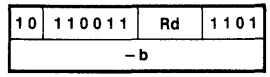
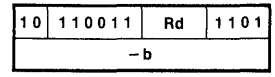
The destination operand is shifted arithmetically right by the number of bit positions specified by the source operands. For SRAB, the source is in the range 0 to 8; for SRA, the source is in the range 0 to 16; for SRAL, the source is in the range 0 to 32. A right shift of zero for SRA is not possible. 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		13 + 3b		13 + 3b
	SRAB Rbd, #b		13 + 3b		13 + 3b
	SRAL RRd, #b		13 + 3b		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.

Note 1: b = number of bit positions; the execution time for b = 0 is the same as for b = 1.

SRL
Shift Right Logical

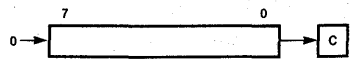
SRL dst, src
SRLB
SRLL

dst: R
src: IM

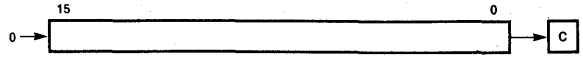
Operation:

Do src times:
c ← dst (0)
dst (n) ← dst (n + 1)(for n = 0 to msb − 1)
dst (msb) ← 0

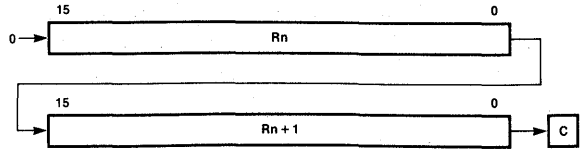
Byte:



Word:



Long:



n = 0, 2, 4, ..., 14

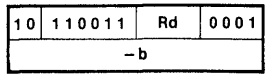
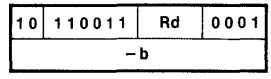
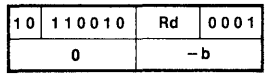
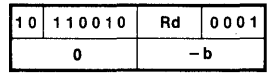
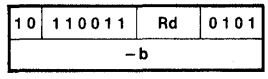
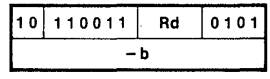
The destination operand is shifted logically right by the number of bit positions specified by the source operand. For SRLB, the source operand is in the range 0 to 8; for SRL, the source is in the range 0 to 16; for SRLL, the source is in the range 0 to 32. A right shift of zero for SRL is not possible. 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 rotation. 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		13 + 3b		13 + 3b
	SRLB Rbd, #b		13 + 3b		13 + 3b
	SRLL RRd, #b		13 + 3b		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.

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
SUBB
SUBL

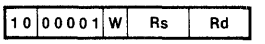
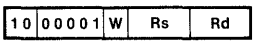
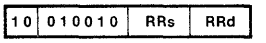
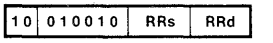
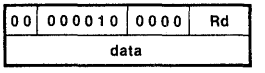
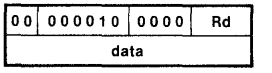
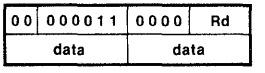
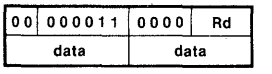
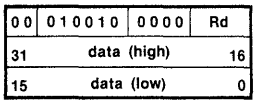
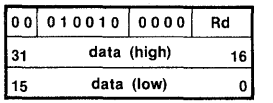
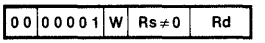
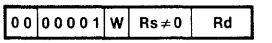
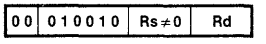
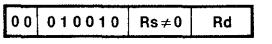
dst: R
src: R, IM, IR, DA, X

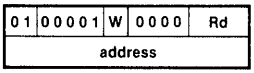
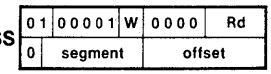
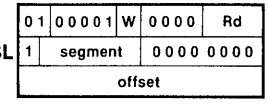
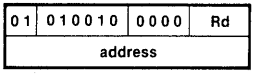
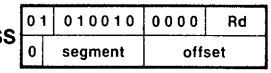
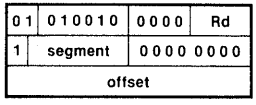
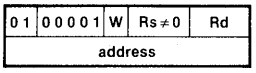
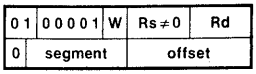
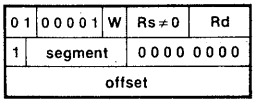
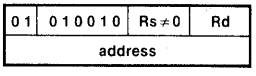
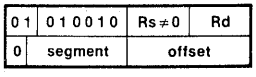
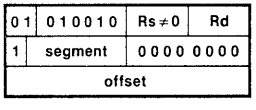
Operation: dst ← dst − src

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 SUBB Rbd, Rbs		4		4
	SUBL RRd, RRs		8		8
IM:	SUB Rd, #data		7		7
	SUBB Rbd, #data		7		7
	SUBL RRd, #data		14		14
IR:	SUB Rd, @Rs[1] SUBB Rbd, @Rs[1]		7		7
	SUBL RRd, @Rs[1]		14		14

Source Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
DA:	SUB Rd, address SUBB Rbd, address		9	SS 	10
				SL 	12
	SUBL RRd, address		15	SS 	16
				SL 	18
X:	SUB Rd, addr(Rs) SUBB Rbd, addr(Rs)		10	SS 	10
				SL 	13
	SUBL RRD, addr(Rs)		16	SS 	16
				SL 	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.

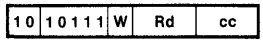
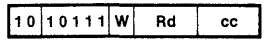
TCC
Test Condition Code

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 TCCB cc, Rbd		5		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.

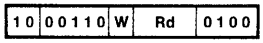
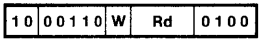
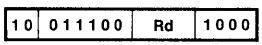
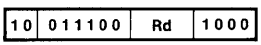
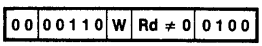
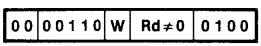
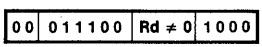
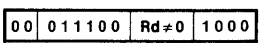
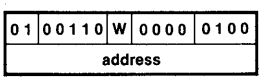
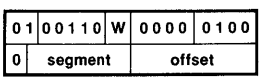
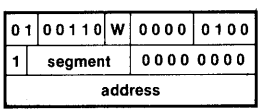
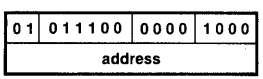
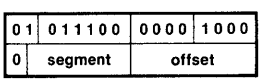
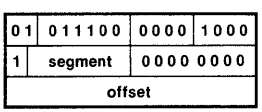
TEST dst
TESTB
TESTL

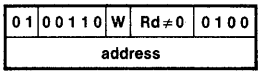
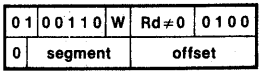
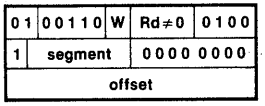
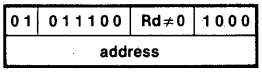
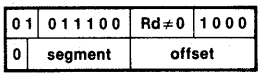
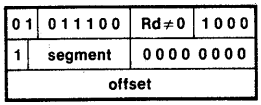
dst: R, IR, DA, X

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 TESTB Rbd		7		7
	TESTL RRd		13		13
IR:	TEST @Rd[1] TESTB @Rd[1]		8		8
	TESTL @Rd[1]		13		13
DA:	TEST address TESTB address		11	SS 	12
				SL 	14
	TESTL address		16	SS 	17
				SL 	19

Destination Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles	Instruction Format	Cycles
X:	TEST addr(Rd) TESTB addr(Rd)		12	SS 	12
				SL 	15
			17	SS 	17
				SL 	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 dst, src, r

dst: IR
src: IR

Operation:

dst ← src[dst]
AUTODECREMENT dst by 1
r ← r − 1

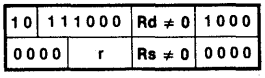
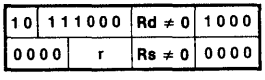
This instruction is used to translate a string of bytes from one code to another code. The contents of the location addressed by the destination register (the "target byte") are used as an index into a table of translation values whose lowest address is contained in the source register. The index is 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. 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		25		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.

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

TRDRB
Translate, Decrement and Repeat

TRDRB dst, src, R

dst: IR
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.

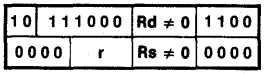
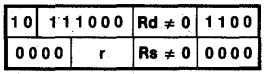
The destination register is then decremented by one, thus moving the pointer to the previous element in the string. The word register specified by "r" (used as a counter) is then decremented by one. The entire operation is repeated until the result of decrementing r is zero. This instruction can translate 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:

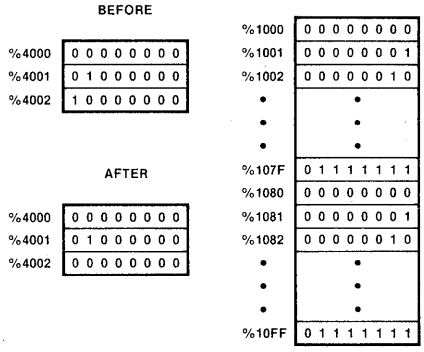
C: Unaffected
Z: Undefined
S: Unaffected
V: Set
D: Unaffected
H: Unaffected

Addressing Mode	Assembler Language Syntax	Nonsegmented Mode		Segmented Mode	
		Instruction Format	Cycles[2]	Instruction Format	Cycles[2]
IR:	TRDRB @Rbd[1], @Rbs[1], r		11 + 14n		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 0000 | r | Rs ≠ 0 | 0000	25	10 | 111000 | Rd ≠ 0 | 0000 0000 | r | Rs ≠ 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
•	•
•	•
•	•
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
•	•
•	•
•	•
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

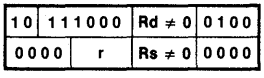
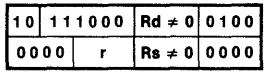
This instruction is used to translate a string of bytes from one code to another code. The contents of the location addressed by the destination register (the "target byte") are used as an index into a table of translation values whose lowest address is contained in the source register. The index is 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		11 + 14n		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 src1, src2, R

src 1: IR
src 2: IR

Operation:

RH1 ← src2[src1]
AUTODECREMENT 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 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, destination, 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 @Rs1[1], @Rs2[1], r	10 | 111000 | Rs1 ≠ 0 | 1010 0000 | r | Rs2 ≠ 0 | 0000	25	10 | 111000 | Rs1 ≠ 0 | 1010 0000 | r | Rs2 ≠ 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.

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

TRTDRB src 1, src 2, R src 1: IR
src 2: 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, destination, 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:

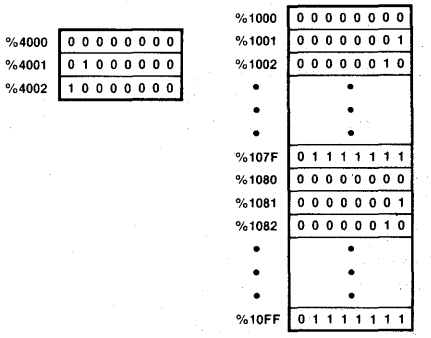
C: Unaffected
Z: Set if the translation value loaded into RH1 is zero; cleared otherwise
S: Unaffected
V: Set if the result of decrementing r is zero; cleared otherwise
D: Unaffected
H: Unaffected

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 0000 | r | Rs2 ≠ 0 | 1110	11 + 14n	10 | 111000 | Rs1 ≠ 0 | 1110 0000 | r | Rs2 ≠ 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.



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

Note 2: n = number of data elements translated.

TRTIB src 1, src 2, R

src 1: IR
src 2: IR

Operation:

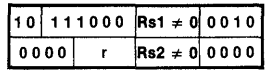
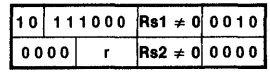
RH1 ← src2[src1]
AUTOINCREMENT src1 by 1
r ← r − 1

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, destination, 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		25		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 src 1, src 2, R

src 1: IR
src 2: 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.

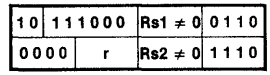
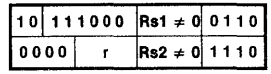
The first source register is then incremented by one, thus moving the pointer to the next element in the string. The word register specified by "r" (used as a counter) is then decremented by one. The entire operation is repeated until either the Z flag is clear, indicating that a non-zero translation 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, destination, 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		11 + 14n		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
TSETB

dst: R, IR, DA, X

Operation:

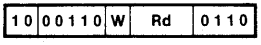
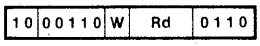
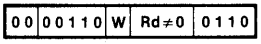
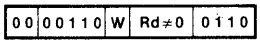
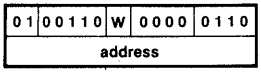
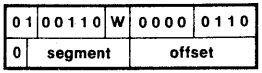
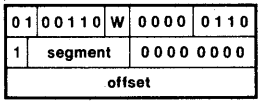
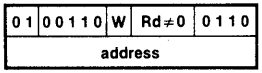
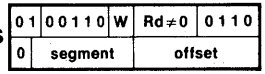
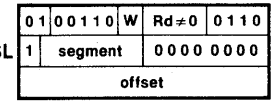
S ← dst(msb)
dst(0:msb) ← 111...111

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. This instruction 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, $\overline{\text{BUSRQ}}$ 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. This instruction should not be used to synchronize software processes residing on separate processors where the destination is a shared memory location, unless this locking mechanism can be guaranteed to function correctly with multi-processor accesses.

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 TSETB Rbd		7		7
IR:	TSET @Rd[1] TSETB @Rd[1]		11		11
DA:	TSET address TSETB address		14	**SS** 	15
				SL 	17
X:	TSET addr(Rd) TSETB addr(Rd)		15	**SS** 	15
				SL 	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

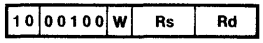
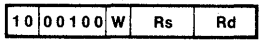
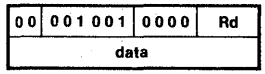
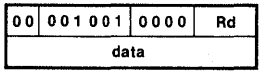
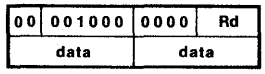
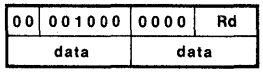
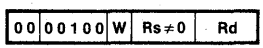
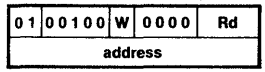
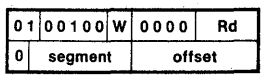
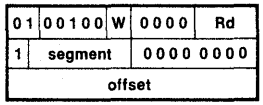
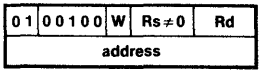
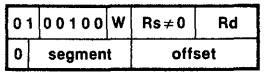
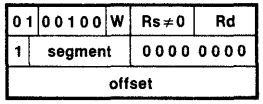
XOR dst, src
XORB

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 XORB Rbd, Rbs		4		4
IM:	XOR Rd, #data		7		7
	XORB Rbd, #data		7		7
IR:	XOR Rd, @Rs[1] XORB Rbd, @Rs[1]		7		7
DA:	XOR Rd, address XORB Rbd, address		9	SS 	10
				SL 	12
X:	XOR Rd, addr(Rs) XORB Rbd, addr(Rs)		10	SS 	10
				SL 	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.

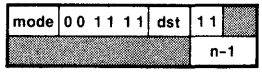
Fields ignored by the CPU are shaded in the diagrams of the templates. The 2-bit field in bit positions 0 and 1 of the first word of each template would normally be used as an identification field for selecting one of up to four EPUs in a multiple EPU system configuration. Other shaded fields would typically 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.

Flags/Registers: No flags or CPU registers are affected by this instruction.



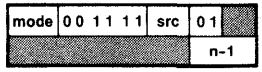
mode	dst	NS	Clock Cycles SS	SL
0 0	IR (dst ≠ 0)	11 + 3n		
0 1	X (dst ≠ 0)	15 + 3n	15 + 3n	18 + 3n
0 1	DA (dst = 0)	14 + 3n	15 + 3n	17 + 3n

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.



mode	src	Clock Cycles NS	SS	SL
0 0	IR (src ≠ 0)	11+3n		
0 1	X (src ≠ 0)	15+3n	15+3n	18+3n
0 1	DA (src = 0)	14+3n	15+3n	17+3n

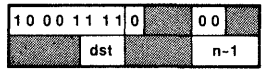
Extended Instruction
Load CPU from EPU

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 transferred consecutively, with register 0 following register 15.

Flags/Registers: No flags are affected by this instruction.

Execution Time: 11 + 3n cycles.



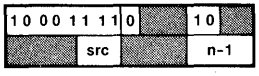
Extended Instruction
Load EPU from CPU

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.

Execution Time: 11 + 3n cycles.



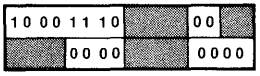
Extended Instruction
Load FCW from EPU

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.

Execution Time: 14 cycles.



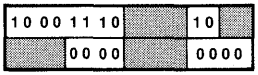
Extended Instruction
Load EPU from FCW

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.

Execution Time: 14 cycles.



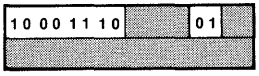
Extended Instruction
Internal EPU Operation

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: The flags in the FCW are unaffected by this instruction.

Execution Time: 14 cycles.



Zilog

Chapter 7
Exceptions

7.1 Introduction

The Z8000 CPU supports three types of exceptions (conditions that can alter the normal flow of program execution):

- interrupts
- traps
- reset

Interrupts are asynchronous events typically triggered by peripheral devices needing attention. They cause the processor to temporarily suspend its present program execution in order to service the requesting device. Traps are synchronous events that are responses by the CPU to certain events detected during the attempted execution of an instruction. Thus, the major distinction 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. 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. The action that the Z8000 CPU takes in response to an interrupt, trap, or reset is similar; hence, they are treated together in this chapter.

7.2 Interrupts

Three kinds of interrupts are activated by three different pins on the Z8000 CPU. (Interrupt handling for all interrupts is discussed in Section 7.6.)

7.2.1 Non-Maskable Interrupt ($\overline{\text{NMI}}$). This type of interrupt cannot be disabled (masked) by software. It is typically reserved for highest-priority 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 may be used to identify the source of the interrupt or trap. In vectored interrupts, this identifier is also used by the CPU hardware as a pointer to select a particular interrupt service routine. The processing of vectored interrupts is thus considerably faster than would be the case if a general trap handler had to first examine the identifier, then branch off to the appropriate service routine. These interrupts can be disabled by software.

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 non-vectored interrupts are serviced by the same routine. They can be disabled by software.

7.3 Traps

The Z8001 and Z8002 CPUs support three traps generated internally. The Z8001 supports a fourth trap, which is generated externally (but synchronously) by the Memory Management Unit. Since a trap always occurs when all its defining conditions are present, traps cannot be disabled. (Trap handling operations are discussed in Section 7.6.)

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. This trap allows the program to simulate the operations of the EPU when none is present in the system or to abort the program.

7.3.2 Privileged Instruction Trap. This trap occurs whenever an attempt is made to execute a privileged instruction while the CPU is in normal mode (S/$\overline{\text{N}}$ bit in the FCW is cleared). This trap allows the CPU to detect and prevent operation (such as I/O) that could disable the system.

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 between normal mode and system mode.

7.3.4 Segment Trap. This trap occurs when ever the $\overline{\text{SEGT}}$ line is asserted on a Z8001, regardless of the state of the SEG bit in the FCW. This trap is generated by external memory management hardware, such as the Z8010 Memory Management Unit (MMU), and is the result of detecting a memory access violation (such as an offset larger than the assigned segment length) or a write warning (a write into the lowest 256 bytes of a stack). See the *MMU Technical Manual* for more information on memory management hardware.

7.4 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 $\overline{\text{RESET}}$ line is Low.

A system reset overrides all other considerations, 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 $\overline{\text{RESET}}$ becoming Low, AD_0–AD_{15} are 3-stated; $\overline{AS}$, $\overline{DS}$, $\overline{MREQ}$, $\overline{BUSACK}$, and $\overline{MO}$ are forced High; ST_0–ST_3 are forced High and SN_0 – SN_6 are forced Low. The R/$\overline{W}$, B/$\overline{W}$, and N/$\overline{S}$ lines are undefined. $\overline{\text{RESET}}$ must be held Low five clock cycles to properly 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, the first cycle reads the FCW from location 0002, the next reads the PC from location 0004, and the following initial instruction fetch cycle starts the program. Each of these fetches is made from system program address space. In the Z8002, the first cycle reads the PC from location 0004 and the following initial instruction fetch cycle starts the program. Each of these fetches is made from the program address space.

7.5 Interrupt Disabling

Vectored and nonvectored interrupts can be enabled or disabled independently via software 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 may be changed by automatically loading a new FCW during an interrupt or trap acknowledge sequence and may be restored to its previous setting by an Interrupt Return (IRET) instruction. When VIE is 1, vectored interrupts are enabled; when NVIE is 1, non-vectored interrupts are enabled. These two flags may be set or cleared together or separately. In addition, these control bits are set when the FCW is loaded. Any control bit may be changed by the occurrence of an interrupt or trap and then be restored to its previous setting by an Interrupt Return (IRET) instruction.

When a type of interrupt has been disabled, the CPU ignores any interrupt request on the corresponding input pin. Because maskable interrupt requests are not retained by the CPU, the request signal must be asserted until the CPU acknowledges the request.

7.6 Interrupt and Trap Handling

The CPU response to a trap or interrupt request consists of five steps: acknowledging the external request (for interrupts and segment traps), saving the old program status information, loading a new program status, executing the service routine, and returning to the interrupted task. Interrupt timing is shown on page A-2.

7.6.1 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 receive from the external device a 16-bit identifier word, which will be saved with the old program status. Before the acknowledge cycle, the CPU enters segmented (Z8001 only) system mode. (The N/$\overline{S}$ line indicates that a transition has been made to system mode.) The old FCW is not affected by this change in mode. The CPU remains in this mode until it begins to execute the exception service routine, at which time its mode is dictated by the FCW.

7.6.2 Status Saving. The old program status information is saved by being pushed on the system stack in the following order: the Program Counter (PC: 16 bits for Z8002; 16-bit offset followed by a word containing the 7-bit segment number for Z8001); the Flag and Control Word (FCW); 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 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.

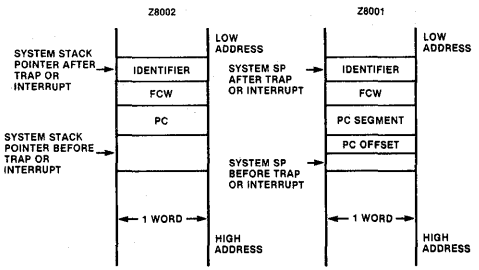


Figure 7-1. Format of Saved Program Status in the System Stack

7.6 Interrupt and Trap Handling (Continued)

The following table shows the PC value that is pushed on the stack for each type of interrupt and trap.

Exception:	PC Value Is Address of:
Extended Instruction Trap	Next Instruction (Single Word Privileged Instruction)
Privileged Instruction Trap	Second Word of Instruction (Multiple Word Privileged Instruction)
System Call Trap	Next Instruction
Segment Trap	Next Instruction*†
All Interrupts	Next Instruction†

* *Assumes successful completion of instruction fetch*

† *If executing an interruptable instruction (e.g. LDIR) and the instruction has not completed, then the next instruction is the current instruction.*

7.6.3 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. 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 Program Status registers (which includes the Flag and Control Word and the Program Counter).

Note that the size of each program status block depends on the version of the Z8000 (two words for the nonsegmented Z8002 and four words for the segmented Z8001).

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) or 128 (Z8001) different values in the 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, 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, the identifier value 0 selects the first PC value, the value 2 selects the second PC, and so on up to the identifier value 254, which selects the 128th PC value. All vectors on Z8001 systems must be even.

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 Z8002 and two words for the segmented Z8001. 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).

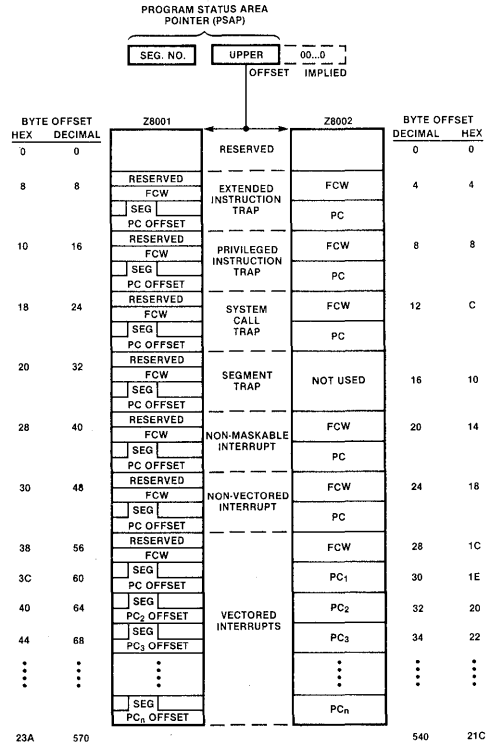


Figure 7-2. Program Status Area

7.6 Interrupt and Trap Handling (Continued)

7.6.4 Executing the Service Routine. Loading the new program status automatically initializes the Program Counter to the starting address of the service routine to process the interrupt or trap. This program is now executed. Because a new FCW was 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 before exiting permit interrupts to be handled in a nested fashion.

7.6.5 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 CPUs, IRET can be executed only in segmented mode; in nonsegmented mode the operation is undefined.

7.7 Priority

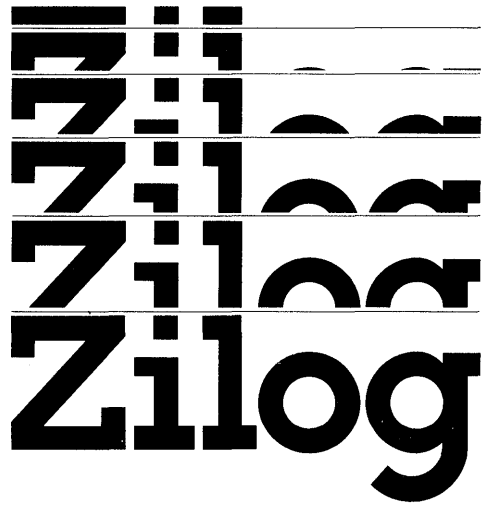
Because it is possible for several exceptions to occur simultaneously, the CPU enforces a priority scheme for deciding which event will be honored first. The following gives the descending priority order:

- *Reset*
- *Internal Trap* (i.e., privileged instruction, system call, extended instruction)
- *Non-Maskable Interrupt*
- *Segment Trap* (Z8001 only)
- *Vectored Interrupt*
- *Nonvectored Interrupt*

This is how the priority system works:

- Whenever a reset is requested, it is immediately performed.
- 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, old 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 and nonvectored interrupts.
- If any enabled exceptions remain, the highest-priority one is acknowledged, the old status is saved, and the new status is loaded. Note that in this case, the old status is the PC and FCW of the previous exception's service routine.
- This process 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.
- The execution of the service routines now proceeds in reverse priority order. That is, the lowest priority exception is serviced first.
- After all the exceptions have been serviced, the original status is restored and execution resumes.

Within each of the classes above, there can be multiple-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 multiple devices share the same request line, the possibility arises that more than one device may request service from the CPU simultaneously. Either all the interrupt sources must be serviced simultaneously (as with the MMU) or competing requests must be resolved externally to the CPU, for example, by means of a daisy-chain or priority interrupt controller. This resolution is done during the interrupt acknowledge cycle.


Zilog

Chapter 8
Refresh

8.1 Introduction

The Z8000 CPU has 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 Counter is set to one (Figure 8.1), memory refresh is performed periodically at a rate specified by the RATE field in the counter. (See Section 8.3.)
- When the $\overline{STOP}$ line is activated, the CPU generates memory refreshes continuously. (See Section 8.4.)

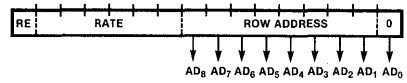


Figure 8-1. Refresh Control Register

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 may be inserted immediately after the last clock cycle of any transaction.

During a refresh cycle, the status lines are set to 0001 and the address lines AD_1–AD_8 are set to the value of the row address counter. Address lines AD_9–AD_{15} are undefined, and AD_0 is always 0. 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.

8.3 Periodic Refresh

The Refresh Enable (RE) bit controls only Periodic Refresh; refresh cycles may be generated using the $\overline{STOP}$ 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. (Thus, if there is a 4 MHz clock, the refresh period can be from 1 μs to 64 μs.)

The LDCTL instruction is used to set the refresh rate, to set or clear RE, or to initialize or read the ROW field. (See Section 6.7 for a detailed discussion of this instruction.)

The refresh cycle is generated as soon as possible after the refresh period has elapsed. This usually means 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, when $\overline{BUSACK}$ is asserted and the CPU enters Bus-Disconnect state) or when the $\overline{WAIT}$ line is deactivated, the CPU issues the skipped refresh cycles. To deal with this situation, both Z8000 CPUs have internal circuitry that records when the refresh period has elapsed and refresh cycles cannot be generated. When the CPU regains control of the bus, or when the $\overline{WAIT}$ line is reactivated, it immediately issues the skipped refresh cycles. The internal circuitry can record up to two such 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: Running, Stop, and Bus-Disconnect states (see Section 2.8). Stop state is entered during the first word fetch of an instruction if $\overline{STOP}$ is activated before the machine cycle begins, or during the second word fetch of an EPA instruction if the $\overline{STOP}$ line is activated before the start of the machine cycle. When $\overline{STOP}$ is found High again, one more refresh cycle is performed, then the remaining clock cycles of the instruction fetch are executed. (See Appendix A for more timing information.)

Zilog

Chapter 9
External Interface

9.1 Introduction

This chapter covers the external manifestations (e.g., 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 *Z-Bus Summary*. 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 operations 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 transactions can occur:

- *Memory transaction.* This type is used to transfer eight or 16 bits of data to or from a memory location (Section 9.4.2).
- *I/O transaction.* This type is used to transfer 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.* This type is used to transfer 16 bits of data between the CPU and an EPU (Section 9.4.4).
- *Interrupt/Trap Acknowledge.* This type is used to acknowledge an interrupt or trap and to transfer an identification/status word from the interrupting or trapping device (Section 9.4.5).
- *Refresh.* These transactions do not transfer data. They refresh dynamic memory (Section 9.4.6).
- *Internal operation.* These transactions do not transfer data. They indicate that the CPU is performing an operation that does not require data to be transferred on the bus (Section 9.4.6).

Only the bus master may initiate transactions. A request, however, may be initiated by a component that does not have control of the bus. Four types of requests can occur:

- *Interrupt request.* This type is used to request the attention of the CPU (Section 9.6.1).
- *Bus request.* This type is used to request control of the bus to initiate transactions (Section 9.6.2).
- *Resource request.* This type is used to request control of a particular system resource (Section 9.6.3).
- *Stop request.* This type is used to delay CPU instruction execution (Section 9.6.4).

When an interrupt or bus request is made, it is answered by the CPU according to its type: for interrupt request, an interrupt acknowledge transaction is initiated; for bus requests, the CPU enters Bus Disconnect state, relinquishes the bus, and activates an acknowledge signal; for stop requests, the CPU stops execution and enters Stop/Refresh state. A resource request is generated by the CPU when it executes a multi-micro request instruction.

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 (Output, 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, only the offset portion of memory addresses is carried on these lines.

SN_0-SN_7. *Segment Number (Z8001 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 Address, 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.

R/$\overline{W}$. *Read/Write (Output, Low = Write, 3-state).* This signal determines the direction of data transfer for memory, I/O, or EPU 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.

$\overline{WAIT}$. *(Input, active Low).* A Low on this line indicates that the responding device needs more time to complete a transaction.

$\overline{MREQ}$. *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.

$\overline{BUSREQ}$. *Bus Request (Input, active Low).* A Low indicates that a bus requester has obtained or is trying to obtain control of the bus.

$\overline{BUSACK}$. *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.

$\overline{NMI}$. *Non-Maskable Interrupt (Input, Edge activated).* A High-to-Low transition on $\overline{NMI}$ requests a non-maskable interrupt.

$\overline{NVI}$. *Non-Vectored Interrupt (Input, active Low).* A Low on this line requests a non-vectored interrupt.

$\overline{VI}$. *Vectored Interrupt (Input, active Low).* A Low on this line requests a vectored interrupt.

$\overline{SEGT}$. *Segment Trap (Z8001 only, Input, active Low).* A Low on this line requests a segment trap.

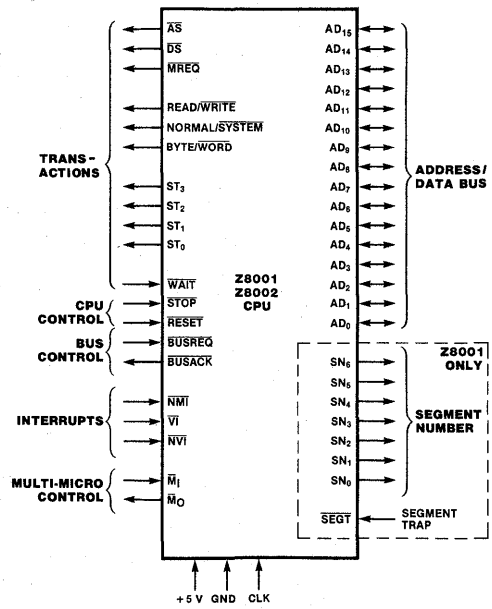


Figure 9-1. Pin Functions

9-3. CPU Pins (Continued)

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 which control the overall operation of the CPU.

$\overline{\text{STOP}}$. *(Input, active Low).* This line is used to suspend CPU operation during the fetch of the first word of an instruction.

$\overline{\text{RESET}}$. *(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. Figure 9.2 shows the general timing for a transaction.

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 indi-

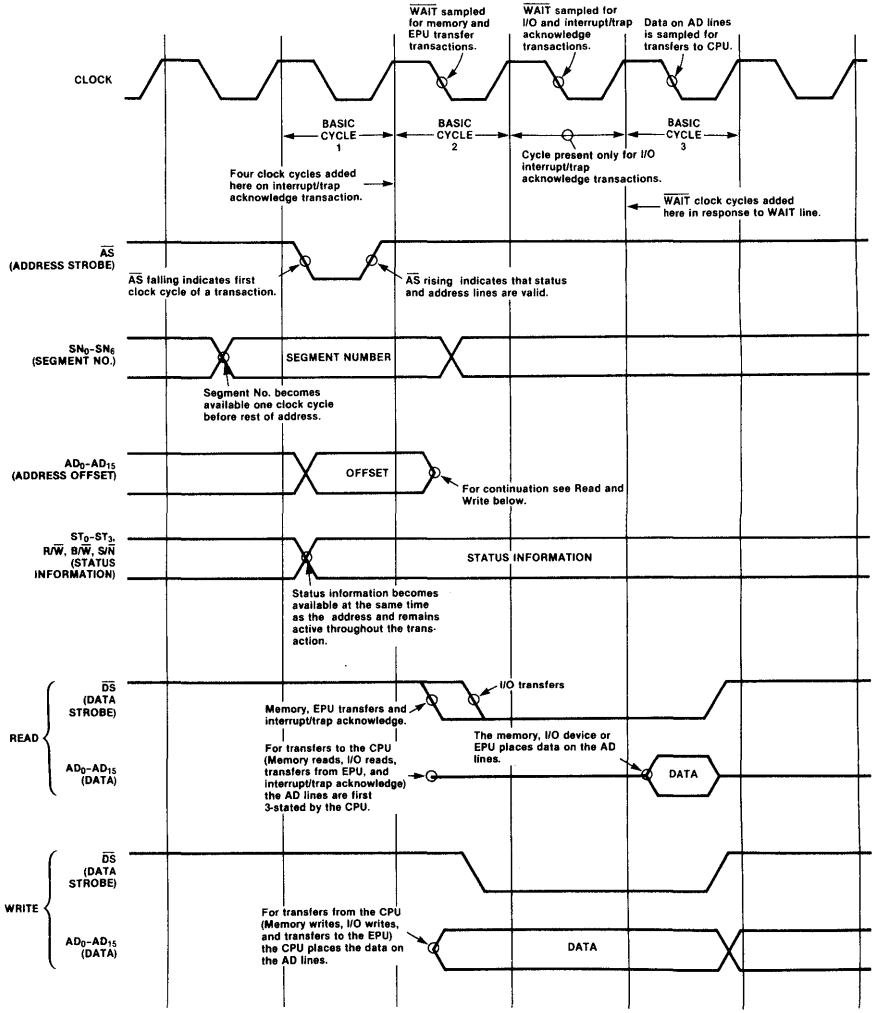


Figure 9-2. Transaction Timing

9-4. Transactions (Continued)

cate the type of transaction being initiated (see Table 9.1; the six types of transactions are discussed in the sections that follow). 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, 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 length 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, the segment number lines SN_0-SN_6 are valid one clock cycle earlier to allow for external memory management hardware. See Chapter 2 for more information.)

The CPU uses Data Strobe ($\overline{DS}$) to time the actual data transfer. (Note that refresh and internal operation transactions do not transfer any data and thus do 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 makes AD_0-AD_{15} 3-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 just before raising $\overline{DS}$ High.

9.4.1 $\overline{WAIT}$. As shown in Figure 9.2, $\overline{WAIT}$ is sampled on a falling clock edge one cycle before data is sampled by the CPU (Read) or $\overline{DS}$ rises (Read or Write). If $\overline{WAIT}$ is Low, another cycle is added to the transaction before data is sampled or $\overline{DS}$ rises. In this added cycle and all subsequent cycles added due to $\overline{WAIT}$ being Low, $\overline{WAIT}$ 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 (for example) slow memories or I/O devices that are not yet ready for data transfer.

It must be emphasized that the $\overline{WAIT}$ input is synchronous. Thus, it must meet the setup and hold times given in Appendix A in order for the CPU to function correctly. This requires asynchronously generated $\overline{WAIT}$ signals to be synchronized before they are input into the CPU.

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 (Chapter 4) and to fetch and store memory data (Chapter 5). They are also generated to store old program status and fetch new program status during interrupt and trap handling and after reset (Chapter 7).

As shown in Figure 9.3, a memory transaction is three clock cycles long unless

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 Trap
	0101	Non-Maskable Interrupt
	0110	Non-Vectored 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 Transfer	1110	
Reserved	1111	

Table 9-1. Status Codes

9-4. Transactions (Continued)

extended as explained above in $\overline{WAIT}$. 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 (1100, 1101) address space (Chapter 3).
- Whether the first word of an instruction is being fetched (1101).
- Whether the data for the access is to be supplied (write) or captured (read) by an Extended Processing Unit (1010, 1011).

Status codes 1000 and 1001 may also indicate that the EPU is to capture or supply the data.

For the Z8002, the full memory address will be on AD_0-AD_{15} when $\overline{AS}$ 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 $\overline{AS}$ rises. The segment portion will also be on SN_0-SN_6 approximately one cycle before AD_0-AD_{15} is valid.

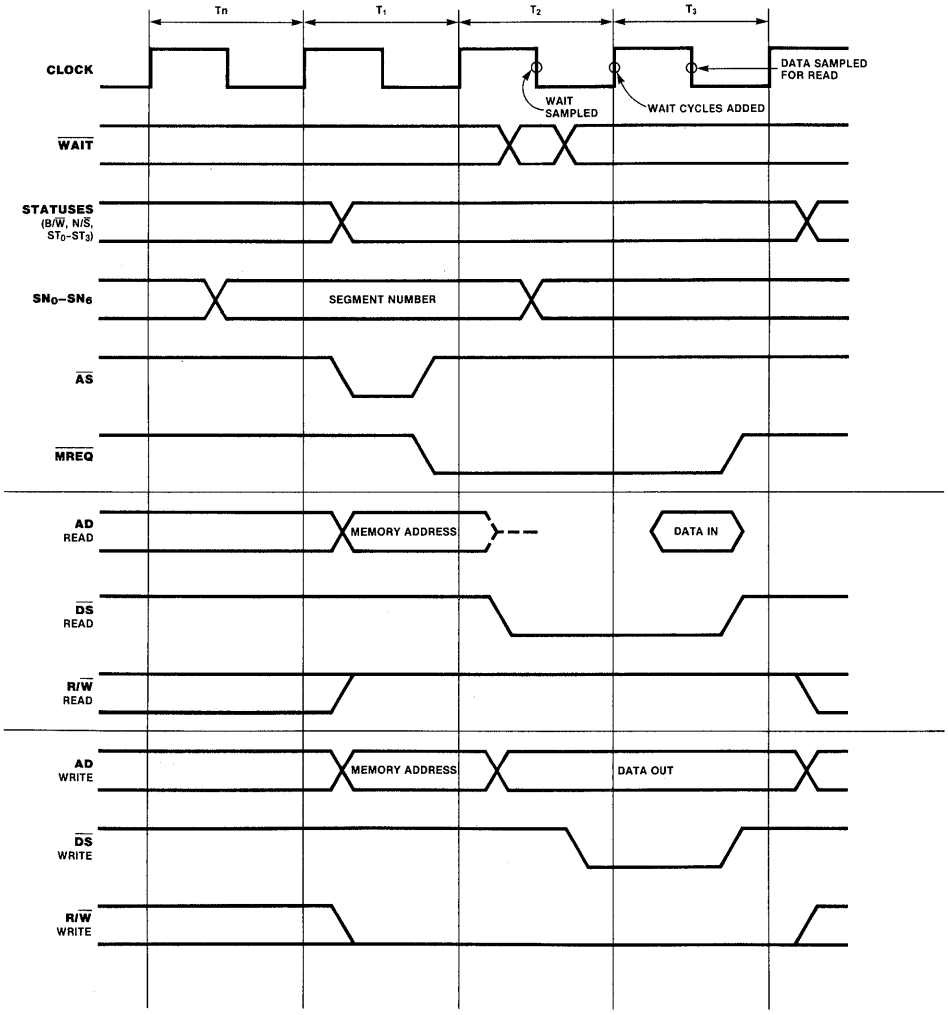


Figure 9-3. Memory Read and Write Transaction

9-4. Transactions (Continued)

Bytes transferred to or from odd memory (address bit 0 is 1) locations are always transmitted on lines AD_0–AD_7 (bit 0 on AD_0). Bytes transferred to or from even memory locations (address bit 0 is 0) are always transmitted on lines AD_8–AD_{15} (bit 0 on AD_8). Thus, the memory attached to a Z8000 will look like that shown in Figure 9.4. For byte reads (B/$\overline{W}$ High, R/$\overline{W}$ High) the CPU uses only the byte whose address it output. For byte writes (B/$\overline{W}$ High, R/$\overline{W}$ Low), the memory should store only the byte whose address was output. During byte memory writes, the CPU places the same byte on both halves of the bus, and the proper byte must be selected by testing A_0. For word transfers, (B/$\overline{W}$ = Low), all 16 bits are captured by the CPU (Read: R/$\overline{W}$ = High) or stored by the memory (Write: R/$\overline{W}$ = Low).

As explained more fully in Section 9.5, a Z8001 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 minimum, and they may be lengthened by the addition of $\overline{WAIT}$ cycles. The extra clock cycles allow for slower peripheral operation.

The status lines indicate whether the access is to the Standard I/O (0010) or Special I/O (0011) Address Spaces. The N/$\overline{S}$ line is always Low, indicating system mode. The I/O address is found on AD_0–AD_{15} when $\overline{AS}$ rises. Since the I/O address is always 16 bits long, the segment number lines are undefined on Z8001 CPUs. For byte transfers (B/$\overline{W}$ = High) in Standard I/O space, addresses must be odd; for byte transfers in Special I/O space, addresses must be even.

Word data (B/$\overline{W}$ = Low) to or from the CPU is transmitted on AD_0–AD_{15}. Byte data (B/$\overline{W}$ = High) is transmitted on AD_0–AD_7 for Standard I/O and on AD_8–AD_{15} for Special I/O. This allows peripheral devices or CPU support devices to attach to only eight of the 16 AD_0–AD_{16} 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).

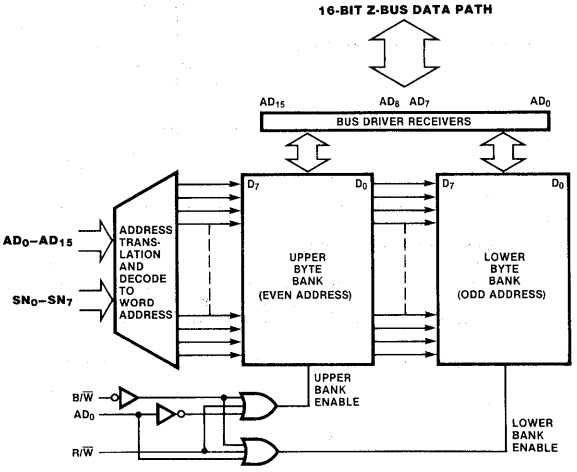


Figure 9-4. Memory Organization

9-4. Transactions (Continued)

9.4.4 EPU Transfer Transactions. These transactions move data between the CPU and an Extended Processing Unit (EPU), thus allowing the CPU to transfer data to or from an EPU or to read or write an EPU's Status Registers. They are generated during the execution of the EPA instruction.

EPU transfer transactions have the same form as memory transactions (Figure 9.3) and thus are three clock cycles long, unless extended by $\overline{WAIT}$. No address is generated, and there is only one status code that can be used on the ST_0-ST_3 lines (1110). In a multiple EPU system, the EPU which is to participate in a transaction is selected implicitly, as described in Section 9.5, rather than by an address.

The data transferred is 16-bit words (B/$\overline{W}$ = Low), except for transfers 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).

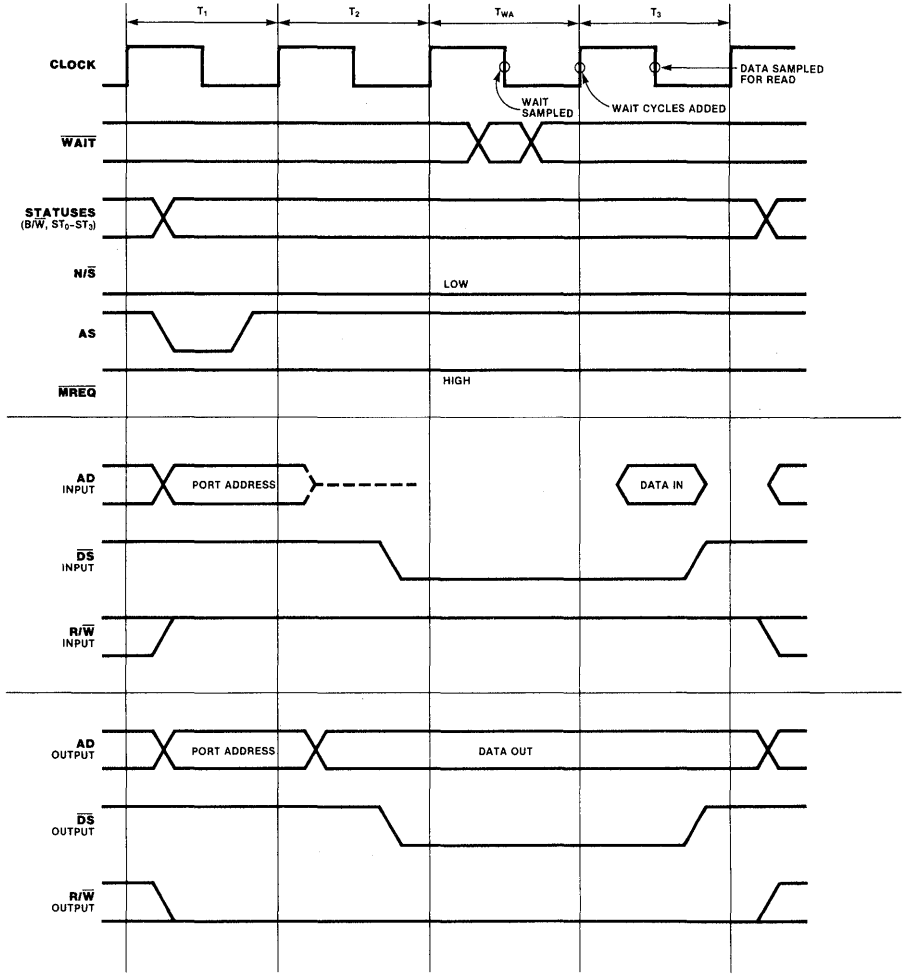


Figure 9-5. Input/Output Transaction

9.4.5 Interrupt/Trap Acknowledge Transactions. These 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 automatically by the hardware when an interrupt or segment trap is detected.

These transactions are eight clock cycles long at a minimum (as shown in Figure 9.6), having five automatic $\overline{WAIT}$ cycles. The $\overline{WAIT}$ 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 Summary* 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 (0100), Non-Maskable Interrupt (0101), Non-Vectored 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 indentifier word, which is always 16 bits long and is captured from the AD_0-AD_{15} lines on the falling clock edge just before $\overline{DS}$ is raised High.

As shown in Figure 9.6, there are two places where $\overline{WAIT}$ is sampled and thus a $\overline{WAIT}$ cycle may be inserted. 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 Internal Operations and Refresh Transactions. There are two kinds of bus transactions made by the CPU that do not transfer data: internal operations and memory refresh. Both transactions look like a memory transaction, except that Data Strobe remains High and no data is transferred.

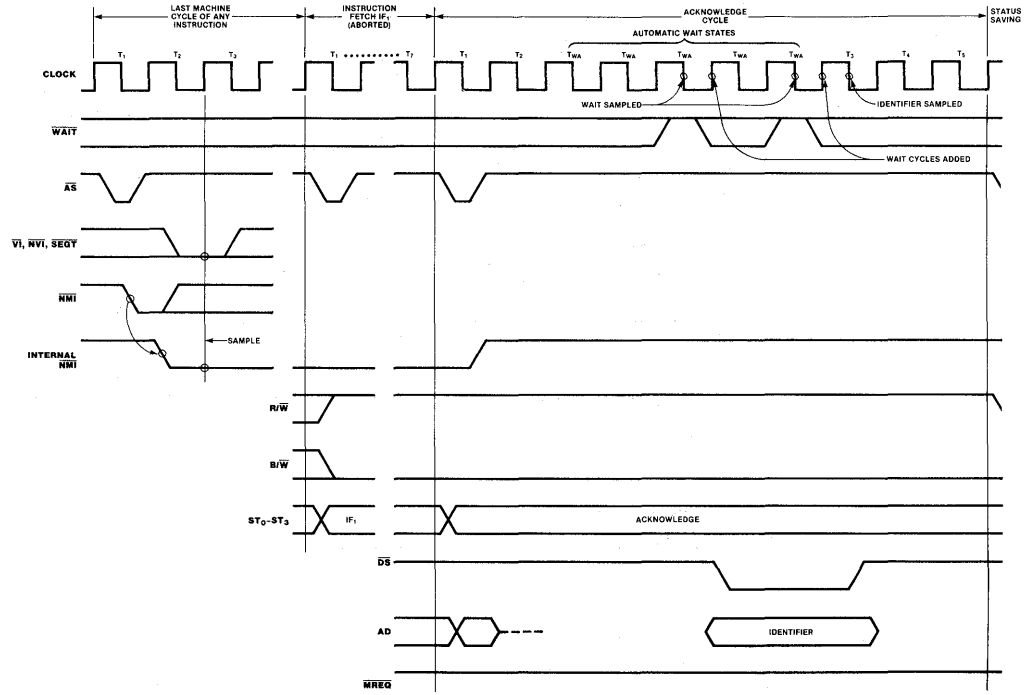


Figure 9-6. Interrupt and Segment Trap Request and Acknowledge Transition.

9-4. Transactions (Continued)

For internal operation transaction (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 cycle of any other transaction. The memory refresh counter's 9-bit ROW field is output on AD_0–AD_8 during the normal time for addresses. This transaction can be used to generate refreshes for dynamic RAMs. The value of N/$\overline{S}$, R/$\overline{W}$, and B/$\overline{W}$ is the same as for the immediately preceeding transaction.

$\overline{WAIT}$ is not sampled during internal operation or refresh cycles.

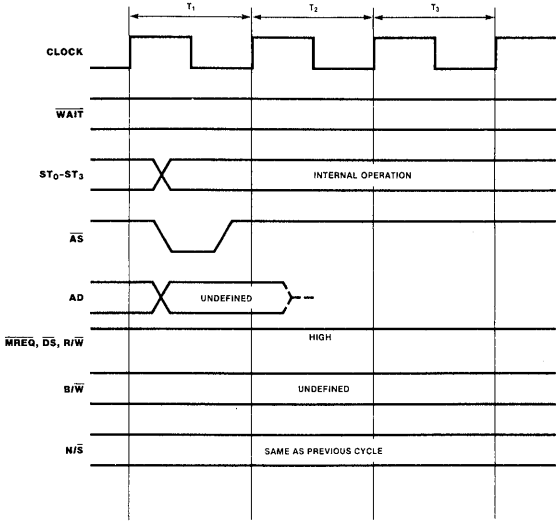


Figure 9-7. Internal Operation Timing

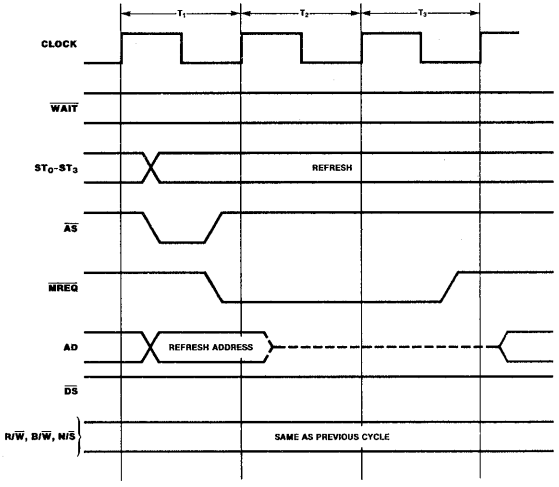


Figure 9-8. Memory Refresh Timing

9.5 CPU and Extended Processing Unit Interaction

A Z8000 CPU and one or more Extended Processing Units (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 monitors 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 puts its AD lines in 3-state while $\overline{DS}$ is Low, so that the EPU may use them.

In order to know which transaction it is to participate in, the EPU must track the following sequence of events:

- When the CPU fetches the first word of an instruction (ST_3-ST_0 = 1101), the EPU must also capture the instruction returned by memory. If the instruction is an extended instruction, it will have an ID field which indicates (along with the second instruction) whether or not the EPU is to execute the instruction.
- 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.
- 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 (ST_3-ST_0 = 1000, 1001, 1010, or 1011). 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 ($\overline{DS}$ Low). EPU memory transfers are always word-oriented (B/$\overline{W}$ Low).
- 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).

Note that in order to follow this sequence, an EPU will have to monitor the $\overline{BUSACK}$ 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 by the EPUs from the extended instructions they capture.

A final aspect of CPU-EPU interaction is the use of the CPU's $\overline{STOP}$ 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 $\overline{STOP}$ pin to stop the CPU (as described in Section 9.7) until the instruction completes execution.

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 and how many words of instruction will be fetched before the data is transferred.
- The number of words of data to be transferred for memory or EPU-CPU transfers.
- The operation to be performed on its data.

9.6 Requests

There are three kinds of request signals that the Z-Bus supports and the Z8000 CPU participates in. 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 of the resource by default.

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 interrupt/trap requests, an interrupt/trap acknowledge transaction is initiated (Section 9.4.4); 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. Requests (Continued)

9.6.1 Interrupt/Trap Request. The Z8000 CPU supports three interrupts and one external trap (segment trap) as shown in Figure 9.6. The Interrupt Request line ($\overline{INT}$) of a device that is capable of generating an interrupt may be tied to any of the three Z8000 interrupt pins ($\overline{NMI}$, $\overline{NVI}$, $\overline{VI}$). Several devices can be connected to one pin, the devices arranged in a priority daisy chain (see the *Z-Bus Summary*). The segment trap pin ($\overline{SEGT}$) 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 $\overline{NMI}$ input is asynchronously edge-detected, and the internal $\overline{NMI}$ latch is set. At the beginning of the last clock cycle in the last machine cycle of any instruction, the $\overline{VI}$, $\overline{NVI}$, and $\overline{SEGT}$ inputs are sampled along with the state of the internal $\overline{NMI}$ latch.
- If an interrupt or trap is detected, the subsequent initial instruction fetch cycle is exercised, but aborted.
- The next machine cycle is the interrupt acknowledge transaction (see Section 9.4.4) that results in an identifier word from the highest-priority interrupting device being read off the AD lines.
- This word, along with the program status information, is stored on the system stack, and new status information is loaded (see Chapter 7).

For more information about the system-level aspects of the interrupt structure, consult the *Z-Bus Summary*.

9.6.2 Bus Request. To generate transactions on the bus, a potential bus master (such as the 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 $\overline{BUSREQ}$ Low. Several bus requesters may be wired to the $\overline{BUSREQ}$ pin; priorities are resolved externally to the CPU, usually by a priority daisy chain (see the *Z-Bus Summary*).

The asynchronous $\overline{BUSREQ}$ signal generates an internal $\overline{BUSREQ}$, which is synchronous. If the external $\overline{BUSREQ}$ is Low at the beginning

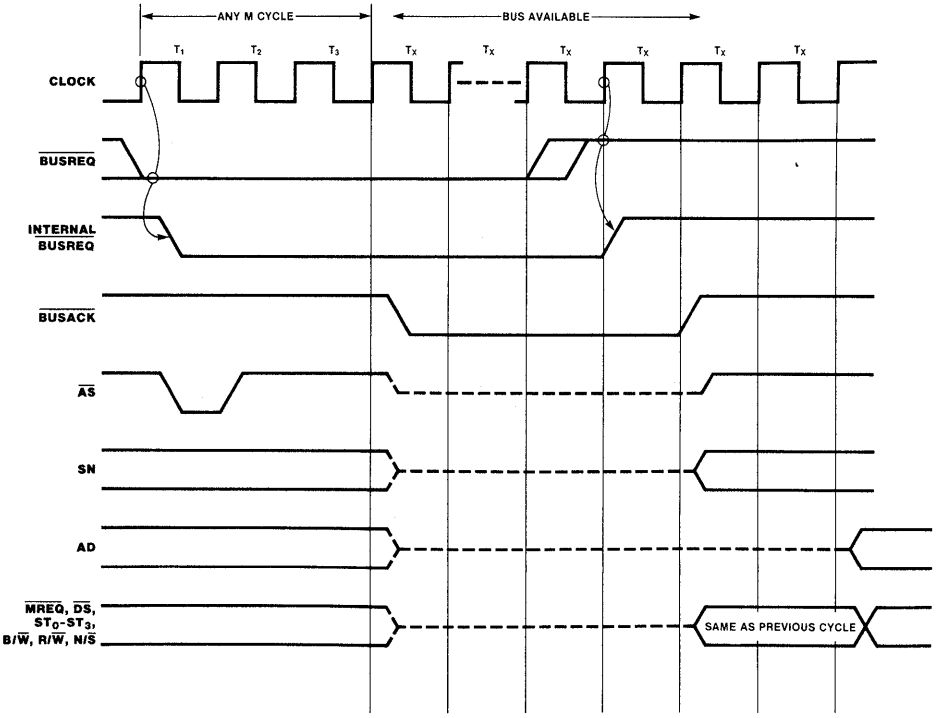


Figure 9-9. Bus Request/Acknowledge Timing

9-6. Requests (Continued)

of any machine cycle, the internal BUSREQ will cause the bus acknowledge line (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 BUSREQ and MO, are 3-stated.

The CPU regains control of the bus two clock cycles after BUSREQ rises. Any device desiring control of the bus must wait at least two cycles after 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 MI. If MI is High, the resource is available, otherwise the CPU must try again later. The MO pin is used to make the resource request. MO is pulled Low, then, after a delay for arbitration of priority, 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. In the case of failure, MO must be deactivated. But if successful, MO must be kept active until the CPU is ready to release the resource whereupon MO is deactivated by an MRES instruction.

The *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 STOP pin is normally sampled on the falling clock edge immediately preceding an initial instruction fetch cycle. If 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 Counter is incremented by two after every refresh cycle.

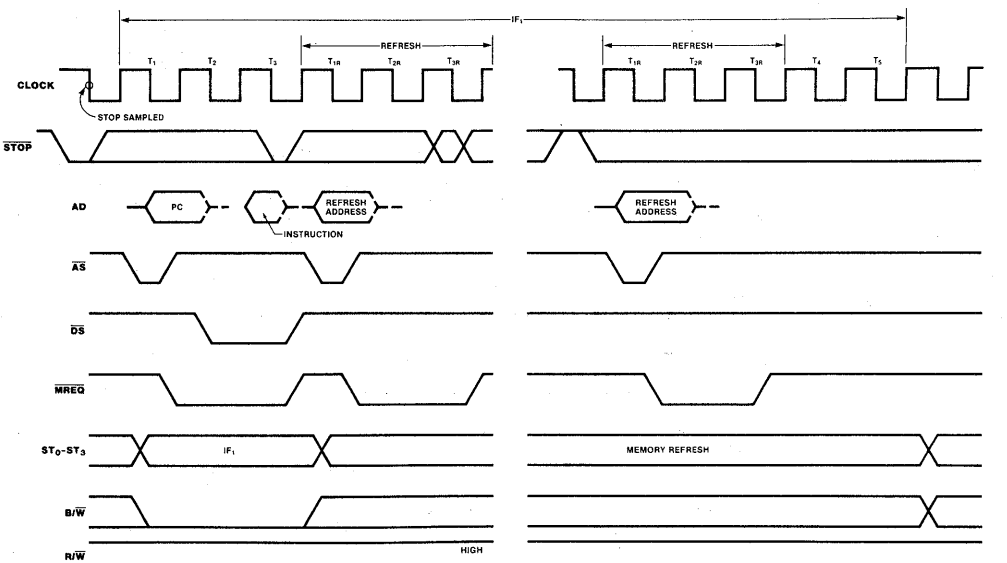


Figure 9-10. Stop Timing

9-6. Requests (Continued)

When $\overline{STOP}$ is found High again, the next refresh cycle is completed, then the original instruction continues.

If the EPA bit in the FCW is set (indicating an EPU is in the system), the $\overline{STOP}$ line is also sampled on the on the falling clock edge immediately preceding the second word of an instruction fetch—if the first word indicates an extended instruction. Thus, the $\overline{STOP}$ line may 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 $\overline{STOP}$ line may also be used to externally single-step the CPU.

9.7 Reset

A hardware reset puts the Z8000 in a known state and initializes selected control registers of the CPU to system specifiable values (as described in Section 7.4). A reset will begin at the end of any clock cycle, if the $\overline{RESET}$ line is Low.

A system reset overrides all other operations of the chip, 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 $\overline{RESET}$ line becoming low (Figure 9.11), AD_0–AD_{15} are 3-stated; $\overline{AS}$, $\overline{DS}$, $\overline{MREQ}$, $\overline{BUSACK}$, M_0, and ST_0–ST_3 are forced High; SN_0–SN_6 are forced Low. The R/$\overline{W}$, B/$\overline{W}$ and N/$\overline{S}$ lines are undefined. Reset must be held Low at least five clock cycles.

After $\overline{RESET}$ has returned High for three clock cycles, consecutive memory-read transactions are executed in the system mode to initialize the Program Status Registers. These correspond to the memory accesses described in Section 7.4

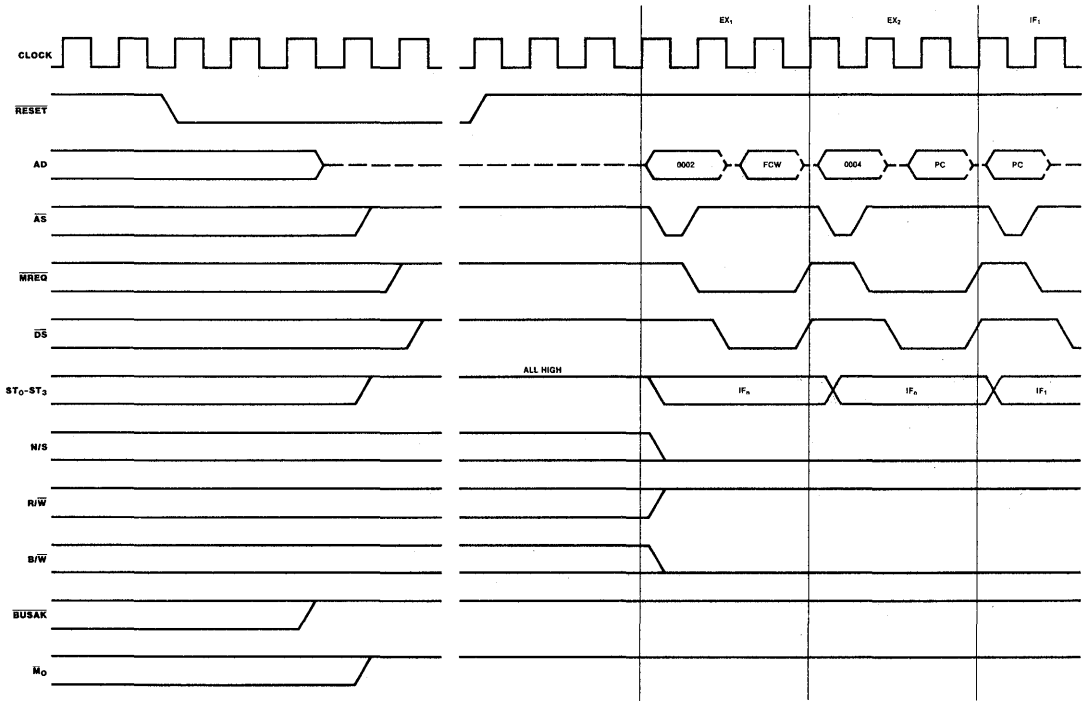
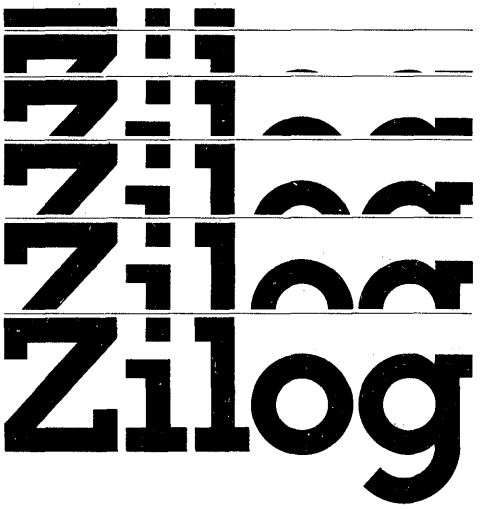
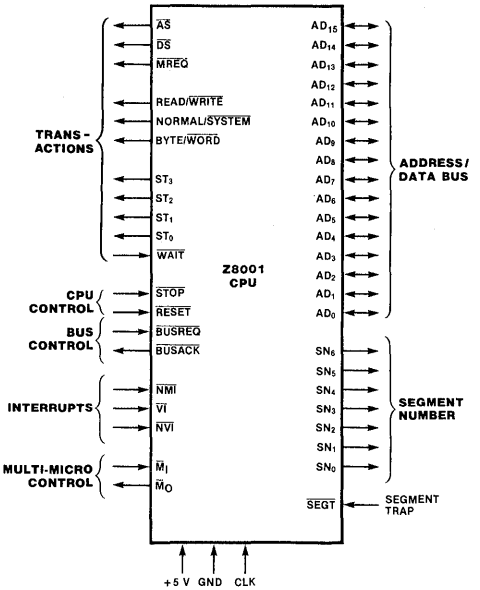


Figure 9-11. Reset Timing

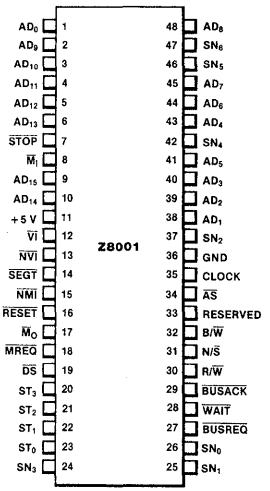
A



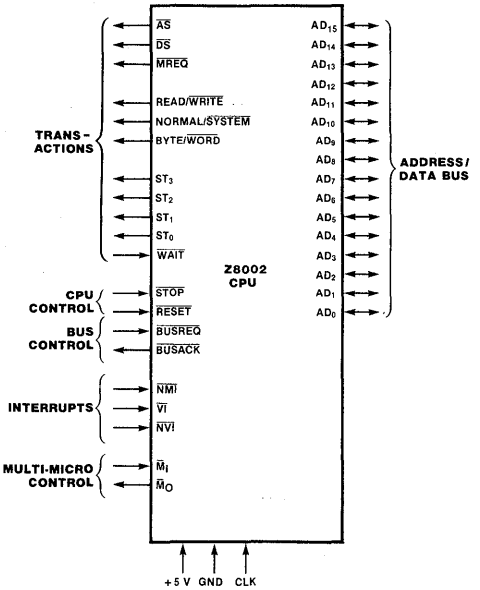
Appendix A



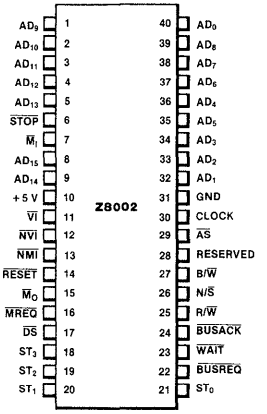
Z8001 CPU Pin Functions



Z8001 Pin Assignments

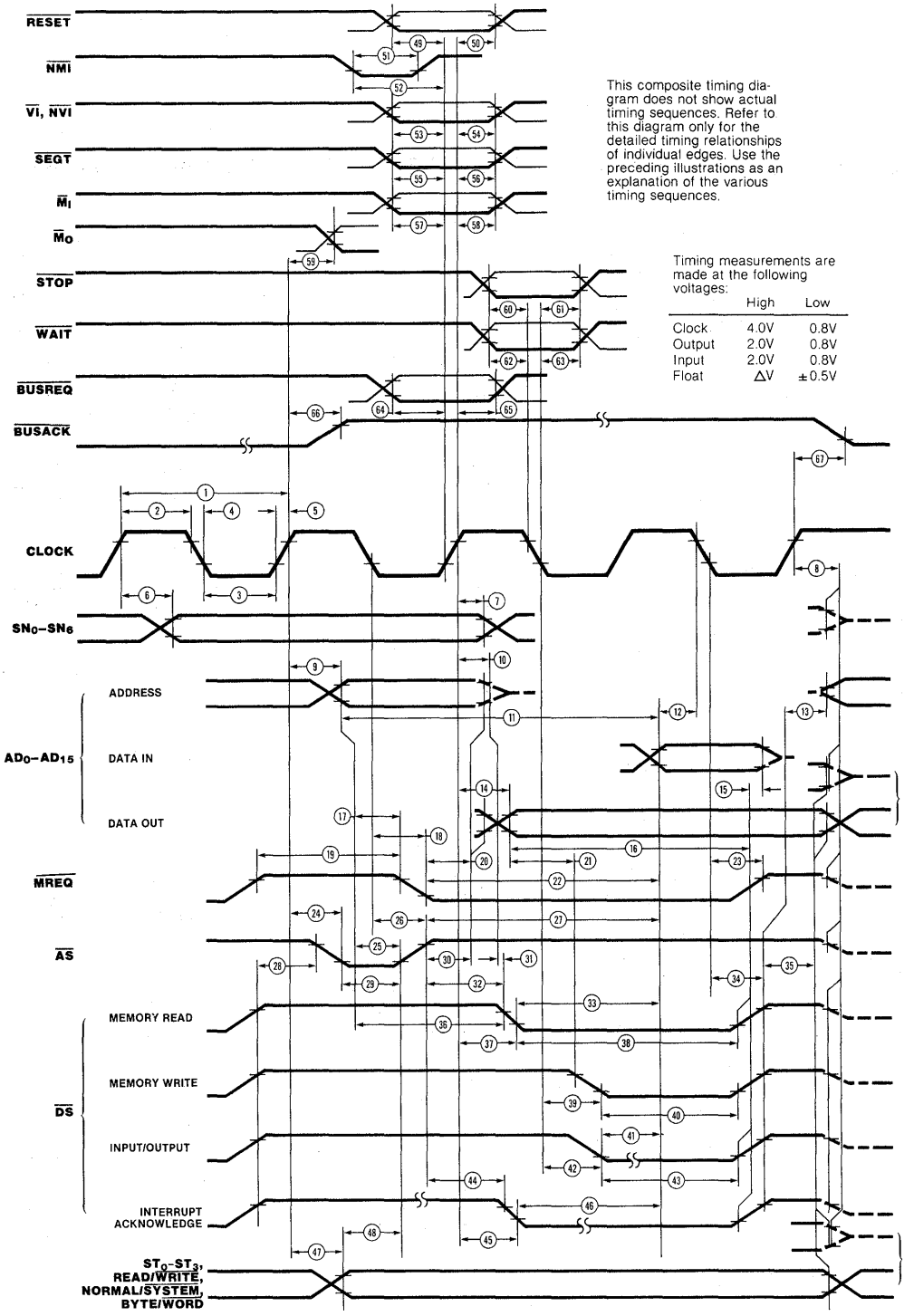


Z8002 CPU Pin Functions



Z8002 Pin Assignments

Composite AC Timing Diagram



This composite timing diagram does not show actual timing sequences. Refer to this diagram only for the detailed timing relationships of individual edges. Use the preceding illustrations as an explanation of the various timing sequences.

Timing measurements are made at the following voltages:

	High	Low
Clock	4.0V	0.8V
Output	2.0V	0.8V
Input	2.0V	0.8V
Float	ΔV	±0.5V

AC Characteristics

Number	Symbol	Parameter	Z8001/Z8002 Min (ns)	Z8001/Z8002 Max (ns)	Z8001A/Z8002A Min (ns)	Z8001A/Z8002A Max (ns)
1	TcC	Clock Cycle Time	250	2000	165	2000
2	TwCh	Clock Width (High)	105	2000	70	2000
3	TwCl	Clock Width (Low)	105	2000	70	2000
4	TfC	Clock Fall Time		20		10
5	TrC	Clock Rise Time		20		10
6	TdC(SNv)	Clock ↑ to Segment Number Valid (50 pF load)		130		110
7	TdC(SNn)	Clock ↑ to Segment Number Not Valid	20		10	
8	TdC(Bz)	Clock ↑ to Bus Float		65		55
9	TdC(A)	Clock ↑ to Address Valid		100		75
10	TdC(Az)	Clock ↑ to Address Float		65		55
11	TdA(DI)	Address Valid to Data In Required Valid		455*		305*
12	TsDI(C)	Data In to Clock ↓ Setup Time	50		20	
13	TdDS(A)	$\overline{DS}$ ↑ to Address Active	80*		40*	
14	TdC(DO)	Clock ↑ to Data Out Valid		100		75
15	ThDI(DS)	Data In to $\overline{DS}$ ↑ Hold Time	0		0	
16	TdDO(DS)	Data Out Valid to $\overline{DS}$ ↑ Delay	295*		195*	
17	TdAMR	Address Valid to MREQ ↓ Delay	55*		35*	
18	TdC(MR)	Clock ↓ to MREQ ↓ Delay		80		70
19	TwMRh	MREQ Width (High)	210*		135*	
20	TdMR(A)	MREQ ↓ to Address Not Active	70*		35*	
21	TdDO(DSW)	Data Out Valid to $\overline{DS}$ ↓ (Write) Delay	55*		35*	
22	TdMR(DI)	MREQ ↓ to Data In Required Valid		350*		225*
23	TdC(MR)	Clock ↓ MREQ ↑ Delay		80		60
24	TdC(ASf)	Clock ↑ to $\overline{AS}$ ↓ Delay		80		60
25	TdA(AS)	Address Valid to $\overline{AS}$ ↑ Delay	55*		35*	
26	TdC(ASr)	Clock ↓ to $\overline{AS}$ ↑ Delay		90		80
27	TdAS(DI)	$\overline{AS}$ ↑ to Data In Required Valid		340*		215*
28	TdDS(AS)	$\overline{DS}$ ↑ to $\overline{AS}$ ↓ Delay	70*		35*	
29	TwAS	$\overline{AS}$ Width (Low)	85*		55*	
30	TdAS(A)	$\overline{AS}$ ↑ to Address Not Active Delay	60*		30*	
31	TdAz(DSR)	Address Float to $\overline{DS}$ (Read) ↓ Delay	0		0	
32	TdAS(DSR)	$\overline{AS}$ ↑ to $\overline{DS}$ (Read) ↓ Delay	70*		35*	
33	TdDSR(DI)	$\overline{DS}$ (Read) ↓ to Data In Required Valid		185*		130*
34	TdC(DSr)	Clock ↓ to $\overline{DS}$ ↑ Delay		70		65
35	TdDS(DO)	$\overline{DS}$ ↑ to Data Out and STATUS Not Valid	75*		45*	
36	TdA(DSR)	Address Valid to $\overline{DS}$ (Read) ↓ Delay	180*		110*	
37	TdC(DSR)	Clock ↑ to $\overline{DS}$ (Read) ↓ Delay		120		85
38	TwDSR	$\overline{DS}$ (Read) Width (Low)	275*		185*	
39	TdC(DSW)	Clock ↓ to $\overline{DS}$ (Write) ↓ Delay		95		80
40	TwDSW	$\overline{DS}$ (Write) Width (Low)	185*		110*	
41	TdDSI(DI)	$\overline{DS}$ (Input) ↓ to Data In Required Valid		320*		200*
42	TdC(DSf)	Clock ↓ to $\overline{DS}$ (I/O) ↓ Delay		120		100
43	TwDS	$\overline{DS}$ (I/O) Width (Low)	410*		255*	
44	TdAS(DSA)	$\overline{AS}$ ↑ to $\overline{DS}$ (Acknowledge) ↓ Delay	1065*		690*	
45	TdC(DSA)	Clock ↑ to $\overline{DS}$ (Acknowledge) ↓ Delay		120		85
46	TdDSA(DI)	$\overline{DS}$ (Ack.) ↓ to Data In Required Delay		435*		295*
47	TdC(S)	Clock ↑ to Status Valid Delay		110		85
48	TdS(AS)	Status Valid to $\overline{AS}$ ↑ Delay	50*		30*	
49	TsR(C)	$\overline{RESET}$ to Clock ↑ Setup Time	180		70	
50	ThR(C)	$\overline{RESET}$ to Clock ↑ Hold Time	0		0	
51	TwNMI	$\overline{NMI}$ Width (Low)	100		70	
52	TsNMI(C)	$\overline{NMI}$ to Clock ↑ Setup Time	140		70	
53	TsVI(C)	$\overline{VI}$, NVI to Clock ↑ Setup Time	110		50	
54	ThVI(C)	$\overline{VI}$, NVI to Clock ↑ Hold Time	0		0	
55	TsSGT(C)	$\overline{SEGT}$ to Clock ↑ Setup Time	70		55	
56	ThSGT(C)	$\overline{SEGT}$ to Clock ↑ Hold Time	0		0	
57	$TsM_I(C)$	$\overline{M_I}$ to Clock ↑ Setup Time	180		110	
58	$ThM_I(C)$	$\overline{M_I}$ to Clock ↑ Hold Time	0		0	
59	$TdC(M_O)$	Clock ↑ to M_O Delay		120		85
60	TsSTP(C)	$\overline{STOP}$ to Clock ↓ Setup Time	140		70	
61	ThSTP(C)	$\overline{STOP}$ to Clock ↓ Hold Time	0		0	
62	TsWT(C)	$\overline{WAIT}$ to Clock ↓ Setup Time	50		30	
63	ThWT(C)	$\overline{WAIT}$ to Clock ↓ Hold Time	10		10	
64	TsBRQ(C)	$\overline{BUSREQ}$ to Clock ↑ Setup Time	90		80	
65	ThBRQ(C)	$\overline{BUSREQ}$ to Clock ↑ Hold Time	10		10	
66	TdC(BAKr)	Clock ↑ to $\overline{BUSACK}$ ↑ Delay		100		75
67	TdC(BAKf)	Clock ↑ to $\overline{BUSACK}$ ↓ Delay		100		75

*Clock-cycle-time-dependent characteristics.

Clock-Cycle-Time-Dependent Characteristics

Number	Symbol	Z8001/Z8002 Equation	Z8001A/Z8002A Equation
11	TdA(DI)	2TcC + TwCh - 150 ns	2TcC + TwCh - 95 ns
13	TdDS(C)	TwCl - 25 ns	TWCl - 30 ns
16	TdDO(DS)	TcC + TwCh - 60 ns	TcC + TwCh - 40 ns
17	TdA(MR)	TwCh - 50 ns	TwCh - 35 ns
19	TwMRh	TcC - 40 ns	TcC - 30 ns
20	TdMR(A)	TwCl - 35 ns	TwCl - 35 ns
21	TdDO(DSW)	TwCh - 50 ns	TwCh - 35 ns
22	TdMR(DI)	2TcC - 150 ns	2TcC - 105 ns
25	TdA(AS)	TwCh - 50 ns	TwCh - 35 ns
27	TdAS(DI)	2TcC - 160 ns	2TcC - 115 ns
28	TdDS(AS)	TwCl - 35 ns	TwCl - 35 ns
29	TwAS	TwCh - 20 ns	TwCh - 15 ns
30	TdAS(A)	TwCl - 45 ns	TwCl - 40 ns
32	TdAS(DSR)	TwCl - 35 ns	TwCl - 35 ns
33	TdDSR(DI)	TcC + TwCh - 170 ns	TcC + TwCh - 105 ns
35	TdDS(DO)	TwCl - 30 ns	TwCl - 25 ns
36	TdA(DSR)	TcC - 70 ns	TcC - 55 ns
38	TwDSR	TcC + TwCh - 80 ns	TcC + TwCh - 50 ns
40	TwDSW	TcC - 65 ns	TcC - 55 ns
41	TdDSI(DI)	2TcC - 180 ns	2TcC - 130 ns
43	TwDS	2TcC - 90 ns	2TcC - 75 ns
44	TdAS(DSA)	4TcC + TwCl - 40 ns	4TcC + TwCl - 40 ns
46	TdDSA(DI)	2TcC + TwCh - 170 ns	2TcC + TwCh - 105 ns
48	TdS(AS)	TwCh - 55 ns	TwCh - 40 ns

Absolute Maximum Ratings

Voltages on all inputs and outputs with respect to GND . . . . . -0.3 V to +7.0 V

Operating Ambient Temperature . . . . . 0°C to +70°C

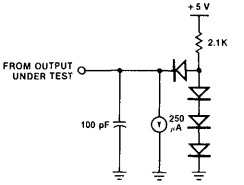
Storage Temperature . . . . . -65°C to +150°C

Stresses greater than those listed under Absolute Maximum Ratings may cause permanent damage to the device. This is a stress rating only; operation of the device at any condition above those indicated in the operational sections of these specifications is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.

Standard Test Conditions

The characteristics below apply for the following standard test conditions, unless otherwise noted. All voltages are referenced to GND. Positive current flows into the referenced pin. Standard conditions are as follows:

- +4.75 V ≤ V_{CC} ≤ +5.25 V
- GND = 0 V
- 0°C ≤ T_A ≤ +70°C



All ac parameters assume a load capacitance of 100 pF max, except for parameter 6 (50 pF max). Timing references between two output signals assume a load difference of 50 pF max.

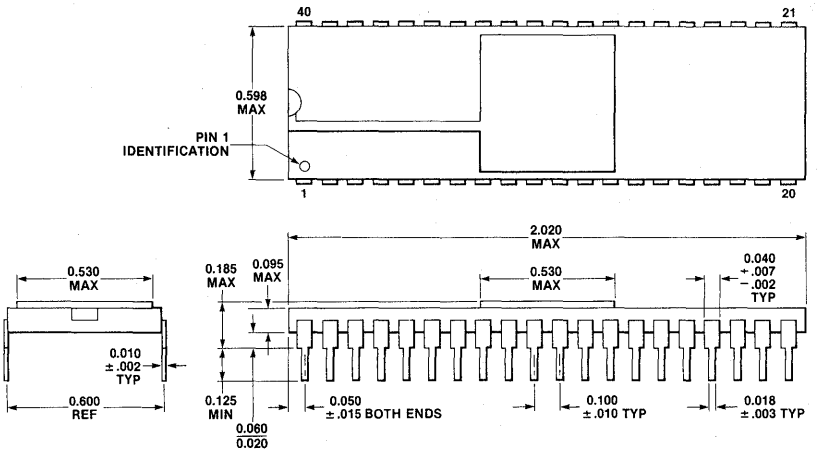
DC Characteristics

Symbol	Parameter	Min	Max	Unit	Condition
V_{CH}	Clock Input High Voltage	V_{CC}-0.4	V_{CC}+0.3	V	Driven by External Clock Generator
V_{CL}	Clock Input Low Voltage	-0.3	0.45	V	Driven by External Clock Generator
V_{IH}	Input High Voltage	2.0	V_{CC}+0.3	V	
$V_{IHRESET}$	High Voltage on Reset Pin	2.4	V_{CC} to 0.3	V	
V_{IL}	Input Low Voltage	-0.3	0.8	V	
V_{OH}	Output High Voltage	2.4		V	I_{OH} = -250 µA
V_{OL}	Output Low Voltage		0.4	V	I_{OL} = +2.0 µA
I_{IL}	Input Leakage		±10	µA	0.4 ≤ V_{IN} ≤ +2.4 V
I_{ILSEGT}	Input Leakage on Segt Pin	-100	100	µA	
I_{OL}	Output Leakage		±10	µA	0.4 ≤ V_{IN} ≤ +2.4 V
I_{CC}	V_{CC} Supply Current		300	mA	

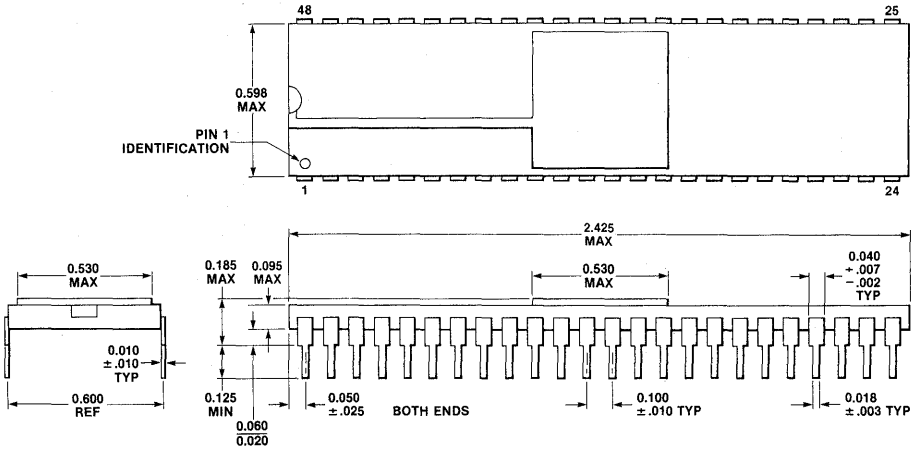
Ordering Information

Part Number	Temperature Range	Number of Pins	Package	Description
Z8001 CPU	0°C to +70°C	48	Ceramic	Segmented 16-Bit Microprocessor
Z8002 CPU	0°C to +70°C	40	Ceramic	Non-Segmented 16-Bit Microprocessor

Package Dimensions



40-Pin Ceramic Package (Z8002)



48-Pin Ceramic Package (Z8001)

B

Zilog

Z8010 MMU Memory Management Unit

Zilog

Product Brief

Preliminary

August 1979

Features

- Dynamic segment relocation makes software addresses independent of physical memory addresses.
- Sophisticated access validation protects memory areas from unauthorized or unintentional access.
- MMU architecture supports multiprogramming systems.
- Sixty-four variable-sized segments from 256 to 64K bytes can be managed within a total physical address space of 16M bytes; all 64 segments are randomly accessible.
- Multiple MMUs can support several translation tables for each of the six Z8001 address spaces.

Description

Declining memory costs coupled with the increasing power of microprocessors has accelerated the use of high-level languages, sophisticated operating systems, complex programs and large data bases in micromputer systems. The Z8001 microprocessor CPU supports these trends with an eight megabyte direct address space as well as a rich and powerful instruction set. The Z8010 Memory Management Unit (MMU) provides flexible and efficient support for this large address space by offering dynamic segment relocation as well as numerous memory-protection features.

The primary memory of a computer is one of its major resources. As such, the management of this resource becomes a major concern as demands on it increase. These demands arise from multiple users (or multiple tasks within a dedicated application), the need to increase system integrity by limiting access

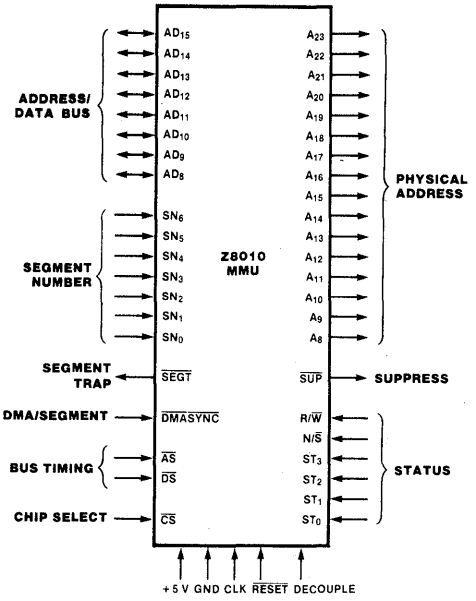


Figure 1. Pin Functions

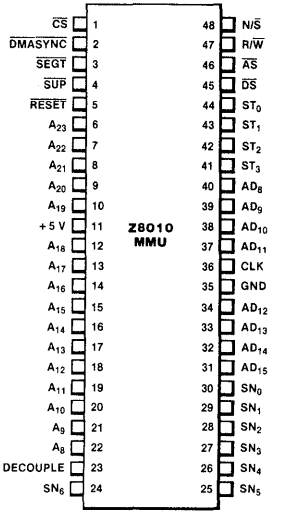


Figure 2. Pin Assignments

Description (Continued)

to various portions of the memory, and from the need to structure large, complex programs and systems.

Multiple tasks (or users) of a system that can reside anywhere in memory are called *relocatable*. Generally, systems in which all tasks are relocatable offer far greater flexibility in responding to changing system environments. Another aspect of multiple-task environments is sharing: separate tasks can execute the same program on different data, or several tasks may execute different programs using the same data.

Unfortunately, a problem that arises in multiple-task systems is that of system integrity. Tasks must be protected from unwanted interactions with other tasks; user tasks must be prohibited from performing operating system functions; and user tasks must also be protected from themselves so they cannot overflow the areas allotted to them.

In addition to these considerations, support for the design and implementation of large, complex programs and systems is itself an important consideration. Modern trends are toward the partitioning of a complex task into small, simple, self-contained subtasks that have well-defined interfaces. Because these subtasks interact with each other, communication between them must be carefully controlled. Memory-management systems can offer effective solutions for implementing large systems modularly designed.

The Z8010 Memory Management Unit supports multiple-process and large modular software systems with dynamic segment relocation. Futhermore, it enhances system integrity with a powerful set of memory protection features.

Relocation. 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 and providing a flexible, efficient method for supporting multi-programming systems.

The Z-MMU uses a translation table to transform the 23-bit logical addresses from the Z8001 CPU into 24-bit addresses for the physical memory. Memory segments are variable in size from 256 bytes to 64K, in increments of 256 bytes. Pairs of Z-MMUs support the 128 segment numbers available for the various Z8001 CPU address spaces. Within an address space, any number of Z-MMUs can be used to accommodate multiple translation tables for system and normal operating modes, or to support more sophisticated memory-management systems.

System Integrity. Z-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 "personality" consisting of several attributes when it is initially entered into the Z-MMU. When a memory reference is made, these attributes are checked against the status information supplied by the Z8001 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 and take appropriate action to correct the problem.

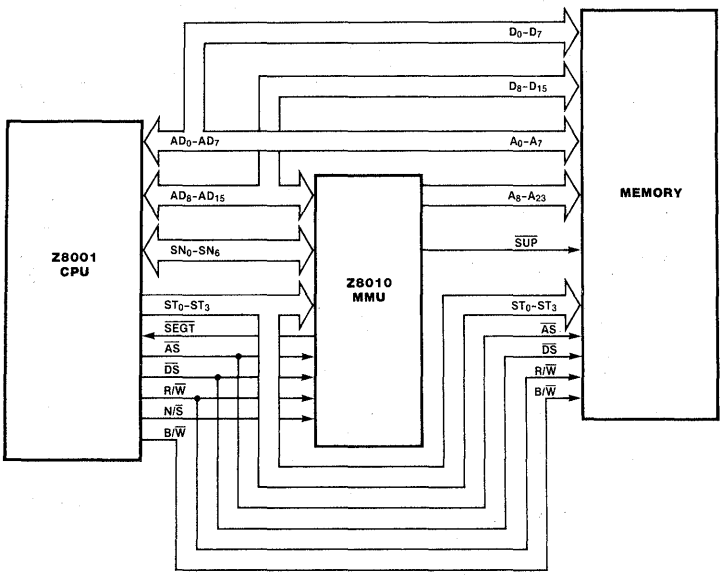


Figure 3. The MMU in a Z8000 System

Z-Bus System Structure

Zilog

Descriptive Brief

Preliminary

August 1979

Features

- Multiplexed address/data bus, shared by I/O and memory.
- Peripherals may be asynchronous.
- Up to 24-bit memory address, 16-bit I/O.
- 8 or 16 data bits.
- Daisy-chained bus request.
- Daisy-chained resource request.
- Vectored or nonvectored interrupts.
- Separate memory and I/O address space.

Description

The Z-bus is a shared bus that links the components of the Z8000 family. A bus user can be any device that can generate bus transactions. Five different types of transactions can be passed on the Z-bus to serve the basic needs of I/O and memory structures in a distributed-processing environment. The five types are:

- Memory access
- I/O transfer
- Interrupt
- Bus request
- Resource request

Direct addressing of the internal registers of peripherals is facilitated by the use of multiplexed address and data lines. (See Figure 1.) The Z-bus is asynchronous, so peripherals' clocks need not be synchronized with the CPU clock, which is therefore not transmitted on the bus directly. The signals (strobes, acknowledges, etc.) generated in the course of any transaction provide all necessary timing information.

Memory Access. Status signals issued by the CPU distinguish memory transactions from others and select the address space to be accessed. Slow memory devices may assert the $\overline{WAIT}$ signal to prolong the transaction. Extended addresses may be used with the segmented Z8001 CPU and the Z8010 MMU. Other status signals define direction (R/$\overline{W}$), Normal/System (N/$\overline{S}$), Byte/Word (B/$\overline{W}$), and the various address spaces.

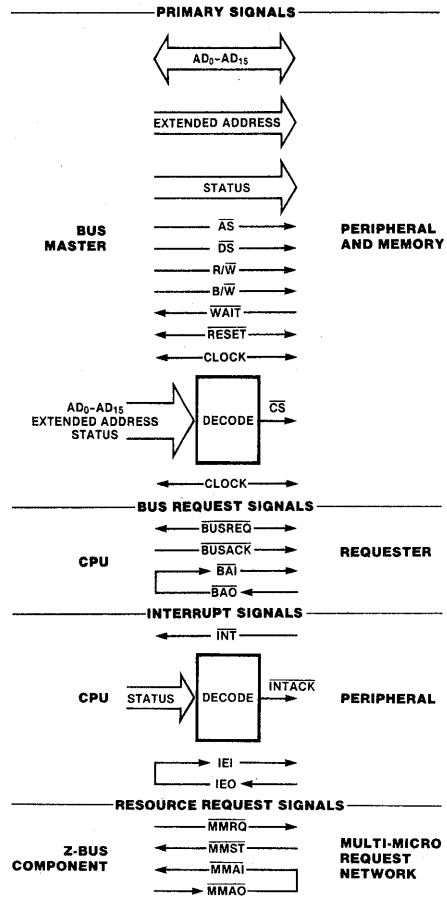


Figure 1. Z-Bus Signals

Description
(Continued)

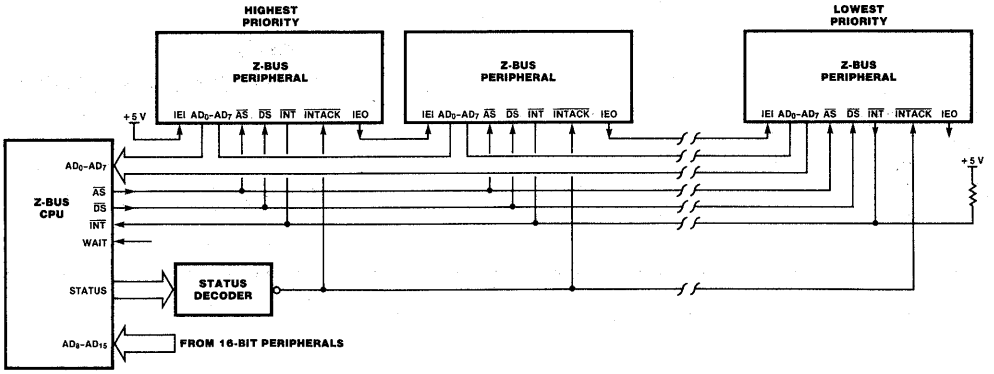


Figure 2. Interrupt Connections

I/O Transfer. The status line I/O reference distinguishes I/O transactions from others. The 16-bit multiplexed bus is used for address and data (without extension), and $\overline{AS}$, $\overline{DS}$, R/$\overline{W}$, B/$\overline{W}$, and $\overline{WAIT}$ are used in a similar way.

Direct addressing of the internal registers of peripherals is facilitated by the use of multiplexed address and data lines. (See Figure 1.) The Z-bus is asynchronous, so peripherals' clocks need not be synchronized with the CPU clock, which is therefore not transmitted on the bus directly. The signals (strobes, acknowledges, etc.) generated in the course of any transaction provide all necessary timing information.

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 2.) Physical position along the IEI/IEO daisy chain determines the priority assigned to any given peripheral. Upon receipt of an $\overline{INT}$ signal, the CPU issues an $\overline{INTACK}$. (See Figure 3.) This temporarily inhibits further interrupt requests, while all devices that have initiated interrupt requests prior to that $\overline{INTACK}$ drop their IEO outputs. (Multiple $\overline{INT}$s might occur simultaneously.) The highest-priority IEO has the effect of removing IEI inputs from all devices beyond it on the same daisy chain, thereby preventing them from requesting interrupts further until their IEI inputs are restored. Three Wait cycles after the leading edge of $\overline{INTACK}$ (or more, if $\overline{WAIT}$ has been asserted by the highest-priority device requesting service), to allow the chain to settle, a $\overline{DS}$ from the CPU stimulates the one highest-priority requesting peripheral to place its vector on the bus. Two (or more) additional Wait cycles later, the service routine is invoked, and $\overline{INTACK}$ is returned high. At this time, all requesters of higher priority than the one being serviced (those whose IEI lines are still high) are enabled, and may generate new interrupt requests. Once a peripheral has been serviced it unmasks the daisy chain so lower-priority interrupts can be generated.

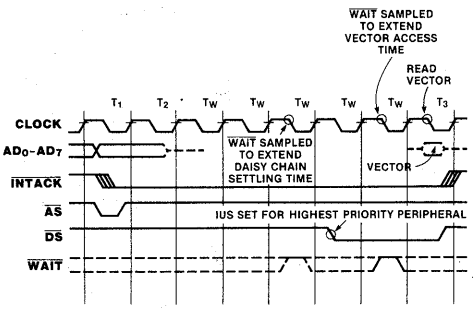


Figure 3. Interrupt Acknowledge Timing

Bus Request. The bus request is used to transfer control of the Z-bus for memory or I/O transactions. The $\overline{BUSRQ}$ input line to the Z-bus CPU, the wired-OR of $\overline{BRQ}$ outputs from all requesters, initiates a bus request. The $\overline{BUSAK}$ output line from the CPU is daisy-chained through $\overline{BAI}$ inputs and $\overline{BAO}$ outputs of all requesters in order of priority, to grant use of the bus to the first requester whose $\overline{BAO}$ is held high at that time.

Resource Request. 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 protocol is similar to that of the bus request, except for an added status line that inhibits all requesters from issuing requests any time the resource is busy. The acknowledge daisy-chain resolves contention in the event of simultaneous requests.

Z8034 UPC Universal Peripheral Controller

Zilog

Product Brief

Preliminary

February 1980

Features

- Complete slave microcomputer, for distributed-processing Z-Bus use.
- Unmatched power of Z8 architecture, instruction set.
- Three programmable I/O ports, two with 2-wire handshake, or any combination of data and control lines.
- Six levels of priority interrupts to Z-UPC.
- Two programmable 8-bit counter/timers with 6-bit prescalers.
- 256 byte register file, accessible by both master CPU and Z-UPC, as allocated by Z-UPC program.
- 2K bytes of on-chip program ROM for efficiency, versatility.

Description

The Z-UPC Universal Peripheral Controller is a distributed microcomputer that performs the three basic interfacing functions needed to interface a CPU with peripherals: device control by ROM-resident internal software, data manipulation, such as reformatting or arithmetic, and data buffering in internal registers.

The Z-UPC is similar to the Z8 microcomputer and uses the Z8 instruction set. Under program control, its three 8-line I/O ports can be tailored to the needs of its user. Permanently configured as a single-chip controller with 2K bytes of internal ROM, the Z-UPC executes instructions in 2.2 μs average using a 4-MHz clock source. Its register file contains 256 bytes, of which 234 are general-purpose registers, 19 are status and control registers, and three are port registers.

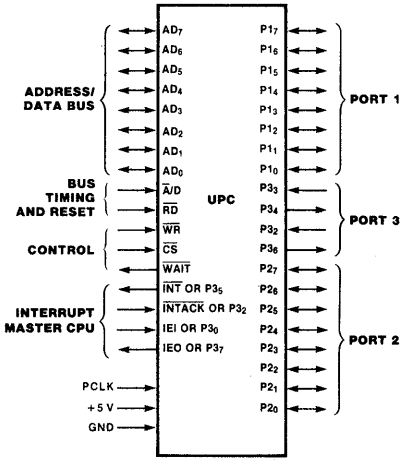


Figure 1. Pin Functions

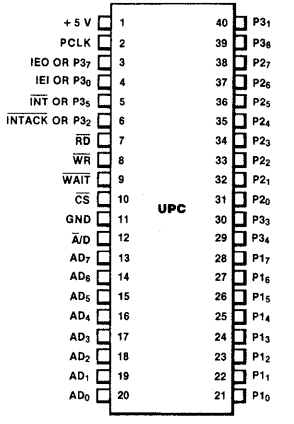


Figure 2. Pin Assignments

Description (Continued)

The Z-UPC Universal Peripheral Controller is an intelligent device that generates all the control signals peripheral devices need. Because it does off-line arithmetic, translates data before transmitting, and buffers data, the Z-UPC unburdens the master CPU, thereby increasing the overall speed and efficiency of the system in which it resides.

Based upon the Z8 microcomputer architecture, the Z-UPC offers fast execution time, efficient use of memory, and sophisticated interrupt, I/O, and bit manipulation. Its powerful and extensive instruction types, combined with its efficient internal register addressing scheme, not only speeds program execution, but also efficiently packs program into the on-chip ROM.

A unique characteristic of the Z-UPC is its register file, which contains I/O port and control registers that can be accessed both by the Z-UPC program and by its associated master CPU. This results in byte efficiency, programming efficiency, and address space efficiency because Z-UPC instructions can operate directly on I/O data without moving it to and from an accumulator. It also allows the Z-UPC user to allocate as data buffer between the CPU and the peripheral all register space not in use as accumulators, address pointers, index registers, or stack. Registers not used as buffer are protected against CPU access.

The register file is divided into 16 groups of 16 working registers each. A register pointer allows fast, short-format instructions to access any one of these groups quickly, resulting in fast and easy task switching. Two-way communication between the master CPU and the register file is facilitated by another pointer that positions 16 interface registers anywhere within the register file. These registers are accessed directly by both the master CPU and the slave Z-UPC. Four more registers, similarly accessed, convey control and status information.

All of Z-Bus's daisy-chained priority interrupt system can be implemented in the Z-UPC under software control, or the Z-UPC can be programmed to function in a polled environment. In all, 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 inputs, outputs, and bidirectional lines, with or without handshake and with push-pull or open-drain outputs.

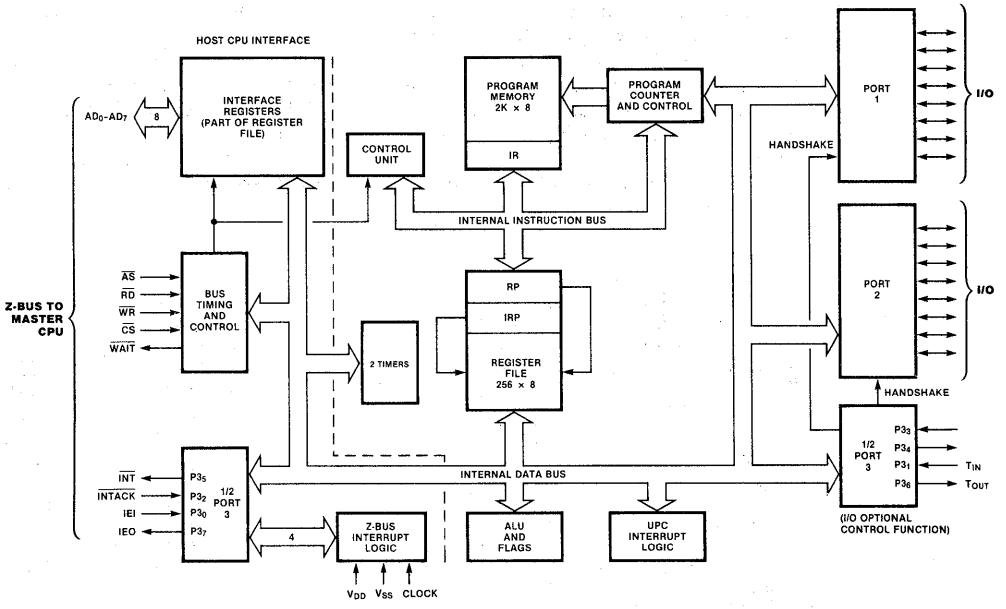


Figure 3. Z-UPC Functional Block Diagram

Z8036 CIO Counter/Timer and Parallel I/O Unit

Zilog

Product Brief

Preliminary

February 1980

Features

- Two independent 8-bit double-buffered bidirectional I/O ports plus a special-purpose 4-bit I/O port.
- Four handshake modes including 3-wire.
- Request/Wait line for high speed data transfer.
- Three independent 16-bit counters.
- All registers read/write and directly addressable.
- Flexible pattern recognition logic, programmable as 16-input interrupt controller.

Description

The Z8036 CIO Counter/timer and Parallel I/O element is a general-purpose peripheral circuit that satisfies 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. Also, each register is given its own unique address so it can be accessed directly—no special sequential operations are required. The Z-CIO is directly Z-Bus compatible.

Either 8-bit I/O port can be a handshake byte port or a bit port. In the bit mode, data direction is programmable bit by bit. In the handshake mode, the ports can be input, output, or bidirectional, and they may be linked to form a 16-bit port. The four handshake modes include 3-wire (like IEEE-488), interlocked (for interfacing to a Z-UPC, Z-FIO or another Z-CIO), strobed, and pulsed. The pulsed mode connects one counter/timer with the handshake logic for interfacing a mechanical device such as a printer. The 4-bit port provides handshake controls, special controls (Wait/Request) or general-purpose I/O.

The counter/timer section contains three 16-bit counters, two of which can be software-configured as a 32-bit counter/timer. Up to

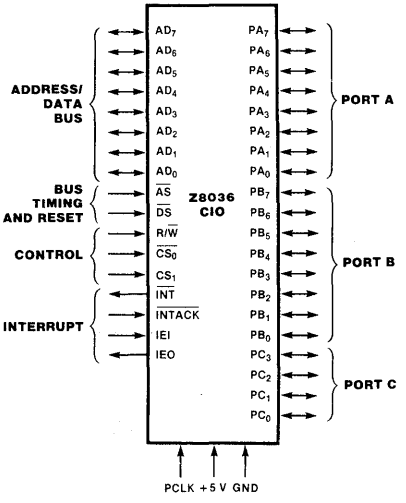


Figure 1. Pin Functions

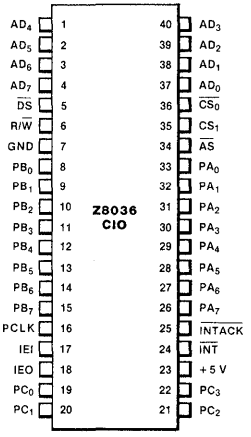


Figure 2. Pin Assignments

Description (Continued)

four I/O lines for each counter are available for direct external control and status information. All counters have a programmable output duty cycle, continuous or single-cycle operation, and the counting process can be programmed to be either retriggered or nonretriggered.

Figure 3 shows how the Z-CIO is used. The two general purpose 8-bit ports are similar. They can be programmed as handshake driven, double-buffered ports (input, output, or bidirectional) or as control ports in which the direction of each bit is individually programmable. Port B can also be specified to provide external access for two of the counter/timers. Each port includes pattern recognition logic allowing interrupt generation when a specified pattern is detected. The pattern recognition logic can be programmed so that the port functions like a priority interrupt controller.

To control these capabilities, each port contains 13 registers. Three of these, the input, output, and buffer registers, are data path registers. Two others, the mode specification and handshake specification registers, define the mode of the port and specify what handshake to use, if any. The reference pattern for the pattern recognition logic is defined in three registers, the pattern polarity, pattern transition, and pattern mask registers. The detailed characteristics of each bit path (for example, the direction of data flow, or whether a path is inverting or noninverting) are programmed using the data path polarity, data direction, and special I/O control registers. The primary control and status bits are grouped in a single register so that after the ports are configured initially, only this register need be accessed often. One register contains the interrupt vector associated with each port. To facilitate initialization, the port logic is designed so that if a capability of the port is not required the registers associated with that capability are ignored and need not be programmed.

The function of port C depends upon the roles of ports A and B. Port C provides handshake lines for the other two when required. Any bits of port C not so used can be used as I/O lines or as external access to the third counter/timer.

Besides the data input and output registers, three registers are needed. These specify the details of each bit path: data path polarity, data direction, and special I/O control.

The three counter/timers are all identical. Each is composed of a 16-bit down-counter, a 16-bit time constant register (which holds the value loaded into the down-counter), a 16-bit current count register (used to read the contents of the down-counter), and two 8-bit registers for control and status (the mode select and control registers). All three share a common vector register.

Each counter/timer can be programmed as either counter or timer. Up to four port I/O lines can be designated as external access lines for it. The lines are: Counter Input, Gate Input, Trigger Input, and Counter/Timer Output. Three different counter/timer output duty cycles are available: pulse, one-shot, or square wave. The operation of the counter/timer can be specified to be either single cycle or continuous. The counting sequence may be retriggered or nonretriggered, under program control.

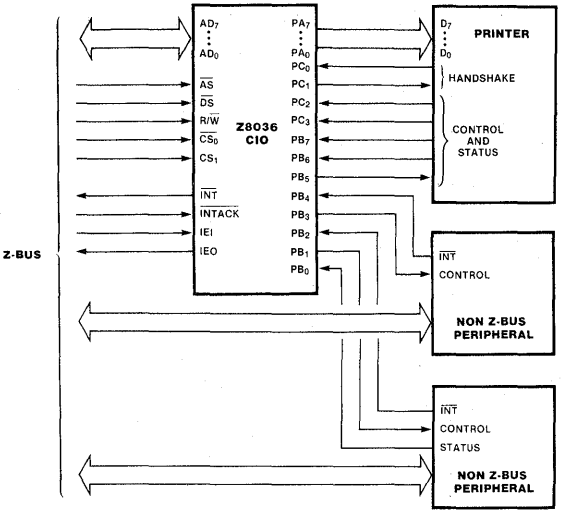


Figure 3. Functional Block Diagram

Z8030 SCC Serial Communications Controller

Zilog

Product Brief

Preliminary

February 1980

Features

- Two independent, 0 to 1M bit per second, full-duplex channels, each with its own quartz oscillator, baud-rate generator, and digital phase-locked loop for clock recovery.
- Multi-protocol operation under program control.
- Programmable for NRZ, NRZI, or FM coding.
- Asynchronous mode with 5 to 8 bits and 1, 1½, or 2 stop bits per character, programmable clock factor, break detection and generation, and parity, overrun, and framing error detection.
- Bisynchronous 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 SDLC 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. However, the Z-SCC also contains a repertoire of new, sophisticated 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. This versatile device also supports virtually any other serial data transfer application (cassette or diskette interface, for example).

The device can generate and check CRC

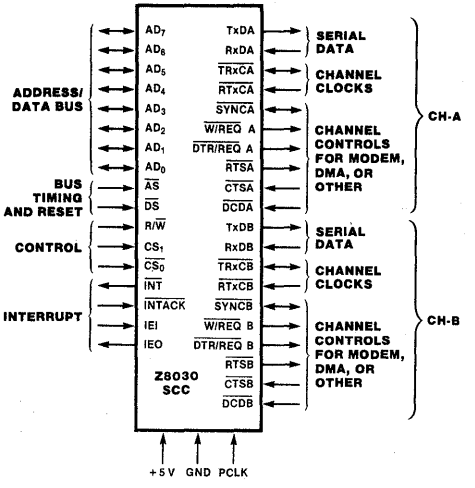


Figure 1. Pin Functions

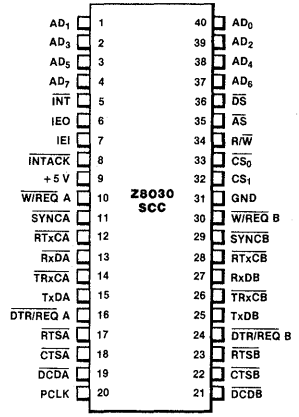


Figure 2. Pin Assignments

Description (Continued)

codes in any synchronous mode and can be 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 ten 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 both channels 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.

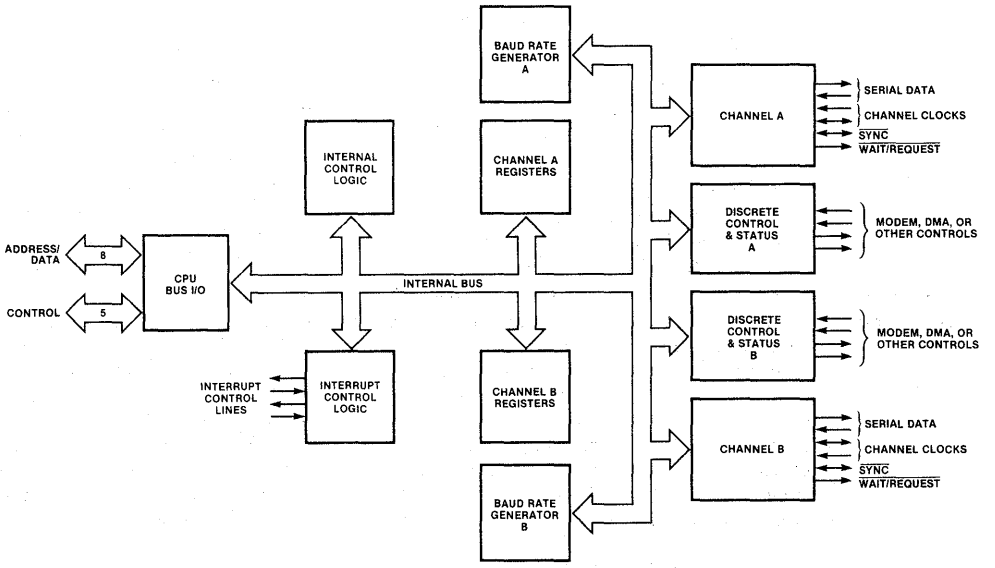


Figure 3. Functional Block Diagram

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.

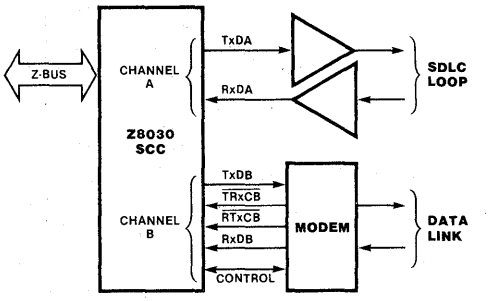


Figure 4. Loop Secondary Station and Serial Port

Z8038 FIO
FIFO Input/Output Interface Unit

Zilog

Product Brief

Preliminary

February 1980

Features

- Asynchronous bidirectional FIFO buffer, used with most major microprocessors as CPU/CPU or CPU/peripheral interface.
- Interlocked 2-wire or 3-wire handshake port mode; Empty, Full, and Request/Wait lines for high-speed data transfer.
- 128 x 8 organization, expandable to any width; cascadable to any depth.
- Preset byte count in FIO buffer can interrupt CPU.
- All registers directly addressable.
- Vectored/non-vectored interrupts on pattern/status match, over/underflow error, buffer status.

Description

The Z-FIO is a general-purpose microprocessor interface that provides elastic buffering between asynchronous CPUs in a parallel-processor network or between CPU and peripheral circuits. The Z-FIO can interface a Z-Bus microprocessor or any other major processor to another microprocessor or to a peripheral circuit or port.

In Z8000 systems, the FIO furthers distributed-processor operation because it can interconnect components or subsystems operating at different speeds. Also, it can increase system throughput by transferring words as well as bytes. This bidirectional device accepts data and holds it until it can be used by another device in the system. In most I/O transactions, introducing a 128-deep buffer cuts interrupt servicing overhead by two orders of magnitude.

The Z-FIO greatly facilitates system throughput by moving variable-size blocks under either direct memory access or interrupt control—an especially important consideration when fast peripheral circuits need interfacing. Complete status information is also provided for operation in polled environments.

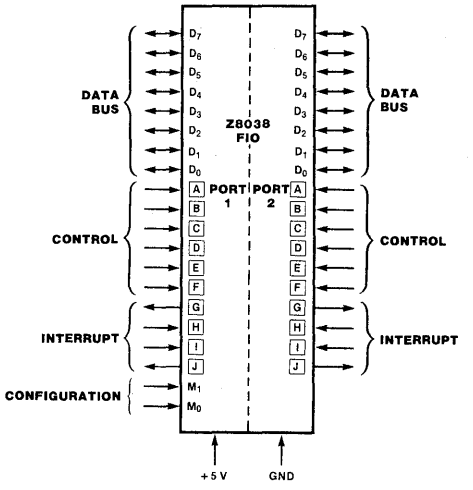


Figure 1. Pin Functions

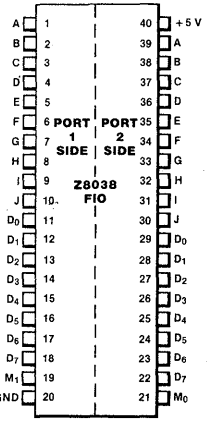


Figure 2. Pin Assignments

Pin Assignments

	Z-Bus Low Byte	Z-Bus High Byte	Non-Z-Bus	2-Wire HS Port*	3-Wire HS Port*
A	$\overline{REQ/WT}$	$\overline{REQ/WT}$	$\overline{REQ/WT}$	RFD/$\overline{DAV}$	RFD/$\overline{DAV}$
B	$\overline{DMASTB}$	$\overline{DMASTB}$	$\overline{DACK}$	$\overline{ACKIN}$	$\overline{DAV}$/DAC
C	$\overline{DS}$	$\overline{DS}$	$\overline{RD}$	FULL	DAC/RFD
D	R/$\overline{W}$	R/$\overline{W}$	$\overline{WR}$	EMPTY	EMPTY
E	$\overline{CS}$	$\overline{CS}$	$\overline{CS}$	$\overline{CLEAR}$	$\overline{CLEAR}$
F	$\overline{AS}$	$\overline{AS}$	C/$\overline{D}$	DATA DIR	DATA DIR
G	$\overline{INTACK}$	A_0	$\overline{INTACK}$	IN_0	IN_0
H	IEO	A_1	IEO	OUT_1	OUT_1
I	IEI	A_2	IEI	$\overline{OE}$	$\overline{OE}$
J	$\overline{INT}$	A_3	$\overline{INT}$	OUT_3	OUT_3

*Port 2 side only. See table below.

Description (Continued)

The internal functions of the Z-FIO are shown in the block diagram (Figure 3). It is made up of two ports that are identical except for programming. The port programmed by pins M_0 and M_1 is called Port 1; the port programmed by bits B_0 and B_1 is called Port 2.

Each port of the FIO has sixteen programmable registers that define operating protocols and pin signals. Common to both ports, and situated between them, is the 128 × 8 RAM used for data storage. The RAM is capable of simultaneous, independent read and write operations. This means, for example, that the Port 1 CPU can write a byte of data into the FIO without disturbing a simultaneous read operation by the Port 2 CPU. The outputs of the read and write counters are used to address the buffer RAM, and also are fed into a subtractor to determine the current number of bytes in the memory. This number can be read by either CPU from a status register dedicated to each port. Another programmable register is compared against the status register to generate interrupts and/or start and stop DMA transfers. A pair of port registers allows for communication between CPUs, bypassing the main buffer memory.

Operating Modes. The Z-FIO has twelve different programmable modes (Table below). The states of two package pins determine the mode of operation of Port 1, and Port 2 is programmed by two bits (B_0 and B_1) in one of the Port 1 control registers.

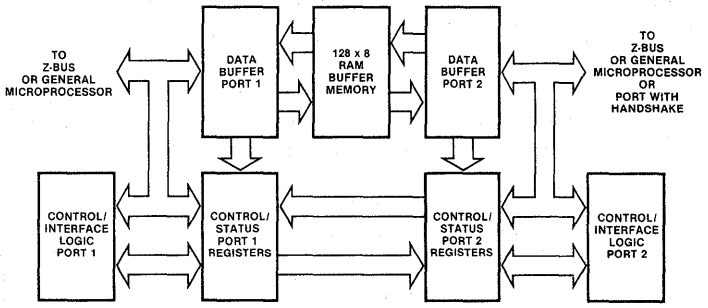


Figure 3. Functional Block Diagram

Operating Modes

Mode	M_1	M_0	B_1	B_0	1	2
0	0	0	0	0	Z-Bus Low Byte	Z-Bus Low Byte
1	0	0	0	1	Z-Bus Low Byte	Non-Z-Bus
2	0	0	1	0	Z-Bus Low Byte	3-Wire HS
3	0	0	1	1	Z-Bus Low Byte	2-Wire HS
4	0	1	0	0	Z-Bus High Byte	Z-Bus High Byte
5	0	1	0	1	Z-Bus High Byte	Non-Z-Bus
6	0	1	1	0	Z-Bus High Byte	3-Wire HS
7	0	1	1	1	Z-Bus High Byte	2-Wire HS
8	1	0	0	0	Non-Z-Bus	Z-Bus Low Byte
9	1	0	0	1	Non-Z-Bus	Non-Z-Bus Low Byte
10	1	0	1	0	Non-Z-Bus	3-Wire HS
11	1	0	1	1	Non-Z-Bus	2-Wire HS

Z8060 FIFO
FIFO Buffer Unit and Z-FIO Expander

Zilog

Product Brief

Preliminary

February 1980

Features

- Asynchronous, bidirectional first-in, first-out buffer.
- Extends depth of Z-FIO without limit.
- 128 x 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 x 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/$\overline{DAV}$ and $\overline{ACKIN}$ 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.

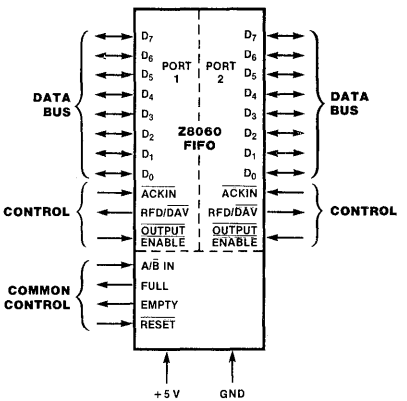


Figure 1. Pin Functions

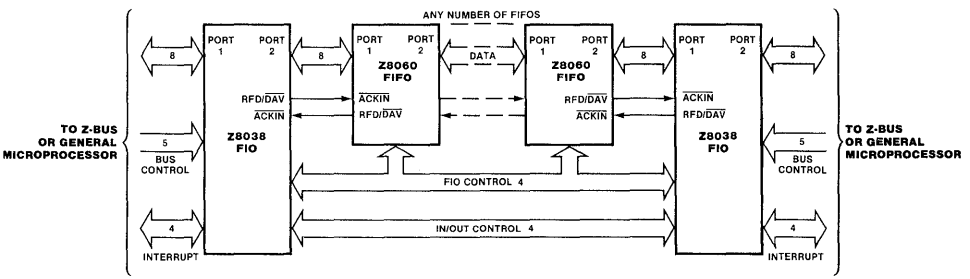


Figure 2. Using FIFOs to Extend FIOs

c

Zilog

Appendix C

Mnemonics	Operands	Addr. Modes	Clock Cycles* Word, Byte NS	SS	SL	Long Word NS	SS	SL	Operation
ADC **ADCB**	R,src	R	5						**Add with Carry** R ← R + src + carry
ADD **ADDB** **ADDL**	R,src	R	4	4	4	8	8	8	**Add** R ← R + src
		IM	7	7	7	14	14	14	
		IR	7			14			
		DA	9	10	12	15	16	18	
		X	10	10	13	16	16	19	
AND **ANDB**	R,src	R	4	4	4				**AND** R ← R AND src
		IM	7	7	7				
		IR	7						
		DA	9	10	12				
		X	10	10	13				
BIT **BITB**	dst,b	R	4	4	4				**Test Bit Static** Z flag ← NOT dst bit specified by b
		IR	8						
		DA	10	11	13				
		X	11	11	14				
BIT **BITB**	dst,R	R	10	10	10				**Test Bit Dynamic** Z flag ← NOT dst bit specified by contents of R
CALL	dst	IR	10	10	15				**Call Subroutine** Autodecrement SP @ SP ← PC PC ← dst
		DA	12	18	20				
		X	13	18	21				
CALR	dst	RA	10	10	15				**Call Relative** Autodecrement SP @ SP ← PC PC ← PC + dst(range -4094 to +4096)
CLR **CLRB**	dst	R	7	7	7				**Clear** dst ← 0
		IR	8						
		DA	11	12	14				
		X	12	12	15				
COM **COMB**	dst	R	7	7	7				**Complement** dst ← NOT dst
		IR	12						
		DA	15	16	18				
		X	16	16	19				
COMFLG	flags		7	7	7				**Complement Flag** (Any combination of C, Z, S, P/V)
CP **CPB** **CPL**	R,src	R	4	4	4	8	8	8	**Compare with Register** R - src
		IM	7	7	7	14	14	14	
		IR	7			14			
		DA	9	10	12	15	16	18	
		X	10	10	13	16	16	19	
CP **CPB**	dst,IM	IR	11						**Compare with Immediate** dst - IM
		DA	14	15	17				
		X	15	15	18				

*** NS = Non-Segmented, SS = Short Segmented Offset, SL =Segmented Long Offset, Blank = Not Implemented.**

Mnemonics	Operands	Addr. Modes	Clock Cycles: Word, Byte NS	SS	SL	Long Word NS	SS	SL	Operation
CPD **CPDB**	R_X,src,R_Y,cc	IR	20						**Compare and Decrement** R_X – src Autodecrement src address $R_Y \leftarrow R_Y - 1$
CPDR **CPDRB**	R_X,src,R_Y,cc	IR	(11 + 9n)						**Compare, Decrement and Repeat** R_Y – src Autodecrement src address $R_X \leftarrow R_Y - 1$ Repeat until cc is true or $R_Y = 0$
CPI **CPIB**	R_X,src,R_Y,cc	IR	20						**Compare and Increment** R_X – src Autoincrement src address $R_Y \leftarrow R_Y - 1$
CPIR **CPIRB**	R_X,src,R_Y,cc	IR	(11 + 9n)						**Compare, Increment and Repeat** R_X – src Autoincrement src address $R_Y \leftarrow R_Y - 1$ Repeat until cc is true or $R_Y = 0$
CPSD **CPSDB**	dst,src,R,cc	IR	25						**Compare String and Decrement** dst – src Autodecrement dst and src addresses R ← R – 1
CPSDR **CPSDRB**	dst,src,R,cc	IR	(11 + 14n)						**Compare String, Decr. and Repeat** dst – src Autodecrement dst and src addresses R ← R – 1 Repeat until cc is true or R = 0
CPSI **CPSIB**	dst,src,R,cc	IR	25						**Compare String and Increment** dst – src Autoincrement dst and src addresses R ← R – 1
CPSIR **CPSIRB**	dst,src,R,cc	IR	(11 + 14n)						**Compare String, Incr. and Repeat** dst – src Autoincrement dst and src addresses R ← R – 1 Repeat until cc is true or R = 0
DAB	dst	R	5	5	5				**Decimal Adjust**
DEC **DECB**	dst,n	R IR DA X	4 11 13 14	4 14 14	4 16 17				**Decrement by n** dst ← dst – n (n = 1...16)
DI*	int		7	7	7				**Disable Interrupt** (Any combination of NVI, VI)
DIV **DIVL**	R,src	R IM IR DA X	107 107 107 108 109	 107 109 109	 107 111 112	744 744 744 745 746	 744 746 746	 744 748 749	**Divide** (signed) Word: $R_{n+1} \leftarrow R_{n,n+1}$ + src $R_n \leftarrow$ remainder Long Word: $R_{n+2,n+3} \leftarrow R_{n...n+3}$ + src $R_{n,n+1} \leftarrow$ remainder

***Privileged instruction. Executed in system mode only.**

Mnemonics	Operands	Addr. Modes	Clock Cycles: Word, Byte NS	SS	SL	Long Word NS	SS	SL	Operation
DJNZ **DBJNZ**	R,dst	RA	11	11	11				**Decrement and Jump if Non-Zero** R ← R − 1 If R ≠ 0: PC ← PC + dst(range -254 to 0)
EI*	int		7	7	7				**Enable Interrupt** (Any combination of NVI, VI)
EX **EXB**	R,src	R IR DA X	6 12 15 16	6 16 16	6 18 19				**Exchange** R ← src
EXTS **EXTSB** **EXTSL**	dst	R	11	11	11	11	11	11	**Extend Sign** Extend sign of low order half of dst through high order half of dst
HALT*			(8 + 3n)						**HALT**
IN* **INB***	R,src	IR DA	10 12	 12	 12				**Input** R ← src
INC **INCB**	dst,n	R IR DA X	4 11 13 14	4 14 14	4 16 17				**Increment by n** dst ← dst + n (n = 1...16)
IND* **INDB***	dst,src,R	IR	21						**Input and Decrement** dst ← src Autodecrement dst addresed R ← R − 1
INDR* **INDRB***	dst,src,R	IR	(11 + 10n)						**Input, Decrement and Repeat** dst ← src Autodecrement dst address R ← R − 1 Repeat until R = 0
INI* **INIB***	dst,src,R	IR	21						**Input and Increment** dst ← src Autoincrement dst address R ← R − 1
INIR* **INIRB***	dst,src,R	IR	(11 + 10n)						**Input, Increment and Repeat** dst ← src Autoincrement dst address R ← R − 1 Repeat until R = 0
IRET*			13	13	16				**Interrupt Return** PS ← @ SP Autoincrement SP
JP	cc,dst	IR IR DA X	10 7 7 8	 8 8	15 7 10 11	(taken) (not taken)			**Jump Conditional** If cc is true: PC ← dst
JR	cc,dst	RA	6	6	6				**Jump Conditional Relative** If cc is true: PC ← PC + dst (range -256 to +254)

***Privileged instruction. Executed in system mode only.**

Mnemonics	Operands	Addr. Modes	Clock Cycles: Word, Byte NS	SS	SL	Long Word NS	SS	SL	Operation
LD **LDB** **LDL**	R,src	R	3	3	3	5	5	5	**Load into Register** R ← src
		IM	7	7	7	11	11	11	
		IM	5	(byte only)					
		IR	7			11			
		DA	9	10	12	12	13	15	
		X	10	10	13	13	13	16	
		BA	14		14	17		17	
		BX	14		14	17		17	
LD **LDB** **LDL**	dst,R	IR	8			11			**Load into Memory** (Store) dst ← R
		DA	11	12	14	14	15	17	
		X	12	12	15	15	15	18	
		BA	14	14	14	17	17	17	
		BX	14	14	14	17	17	17	
LD **LDB**	dst,IM	IR	11						**Load Immediate into Memory** dst ← IM
		DA	14	15	17				
		X	15	15	18				
LDA	R,src	DA	12	13	15				**Load Address** R ← source address
		X	13	13	16				
		BA	15	15	15				
		BX	15	15	15				
LDAR	R,src	RA	15	15	15				**Load Address Relative**
LDCTL*	CTLR,src	R	7	7	7				**Load into Control Register** CTLR ← src
LDCTL*	dst,CLTR	R	7	7	7				**Load from Control Register** dst ← CTLR
LDCTLB	FLGR,src	R	7	7	7				**Load into Flag Byte Register** FLGR ← src
LDCTLB	dst,FLGR	R	7	7	7				**Load from Flag Byte Register** dst ← FLGR
LDD **LDDB**	dst,src,R	IR	20						**Load and Decrement** dst ← src Autodecrement dst and src addresses R ← R + 1
LDDR **LDDRB**	dst,src,R	IR	(11 + 9 n)						**Load, Decrement and Repeat** dst ← src Autodecrement dst and src addresses R ← R − 1 Repeat until R = 0
LDI **LDIB**	dst,src,R	IR	20						**Load and Increment** dst ← src Autoincrement dst and src addresses R ← R − 1
LDIR **LDIRB**	dst,src,R	IR	(11 + 9 n)						**Load, Increment and Repeat** dst ← src Autoincrement dst and src addresses R ← R − 1 Repeat until R = 0

Mnemonics	Operands	Addr. Modes	Clock Cycles: Word, Byte NS	SS	SL	Long Word NS	SS	SL	Operation
LDK	R,src	IM	5	5	5				**Load Constant** R ← n (n = 0...15)
LDM	R,src,n	IR	11						**Load Multiple** dst ← src (n consecutive words) (n = 1...16)
		DA	14	15	17	+ 3n			
		X	15	15	18				
LDM	dst,R,n	IR	11						**Load Multiple** (Store Multiple) dst ← R (n consecutive words) (n = 1...16)
		DA	14	15	17	+ 3n			
		X	15	15	18				
LDPS*	src	IR	12						**Load Program Status** PS ← src
		DA	16	20	22				
		X	17	20	23				
LDR **LDRB**	R,src	RA	14	14	14	17	17	17	**Load Relative** R ← src (range -32768... + 32767)
LDR **LDRB** **LDRL**	dst,R	RA	14	14	14	17	17	17	**Load Relative** (Store Relative) dst ← R (range -32768... + 32767)
MBIT*			7	7	7				**Test Multi-Micro Bit** Set if M_I is Low; reset S if M_I is High.
MREQ*	dst	R	(12 + 7n)						**Multi-Mircre Request**
MRES*			5	5	5				**Multi-Micro Reset**
MSET*			5	5	5				**Multi-Micro Set**
MULT **MULTL**	R,src	R	70	70	70	282 +	282 +	282 +	**Multiply** (signed) Word: $R_{n,n+1} \leftarrow R_{n+1} \cdot$ src Long Word: $R_{n...n+3} \leftarrow R_{n+2,\,n+3} \cdot$ src + Plus seven cycles for each 1 in the absolute value of the low order 16 bits of the multiplicand.
		IM	70	70	70	282 +	282 +	282 +	
		IR	70			282 +			
		DA	71	72	74	283 +	283 +	286 +	
		X	72	72	75	284 +	284 +	287 +	
NEG **NEGB**	dst	R	7	7	7				**Negate** dst ← 0 − dst
		IR	12						
		DA	15	16	18				
		X	16	16	19				
NOP			7	7	7				**No Operation**
OR **ORB**	R,src	R	4	4	4				**OR** R ← R OR src
		IM	7	7	7				
		IR	7						
		DA	9	10	12				
		X	10	10	13				
OTDR* **OTDRB***	dst,src,r	IR	(11 + 10 n)						**Output, Decrement and Repeat** dst ← src Autodecrement src address R ← R − 1 Repeat until R = 0

***Privileged instructions. Executed in system mode only.**

Mnemonics	Operands	Addr. Modes	Clock Cycles: Word, Byte NS	SS	SL	Long Word NS	SS	SL	Operation
OTIR* **OTIRB***	dst,src,R	IR	(11 + 10 n)						**Output, Increment and Repeat** dst ← src Autoincrement scr address R ← R − 1 Repeat until R = 0
OUT* **OUTB***	dst,R	IR	10						**Output** dst ← R
		DA	12	12	12				
OUTD* **OUTDB***	dst,src,R	IR	21						**Output and Decrement** dst ← src Autodecrement src address R ← R − 1
OUTI* **OUTIB***	dst,src,R	IR	21						**Output and Increment** dst ← src Autoincrement src address R ← R − 1
POP **POPL**	dst,IR	R	8	8	8	12	12	12	**Pop** dst ← IR Autoincrement contents of R
		IR	12			19			
		DA	16	16	18	23	23	25	
		X	16	16	19	23	23	26	
PUSH **PUSHL**	IR,src	R	9	9	9	12	12	12	**Push** Autodecrement contents of R IR ← src
		IM	12	12	12				
		IR	13			20			
		DA	14	14	16	16	21	23	
		X	14	14	17	21	21	24	
RES **RESB**	dst,b	R	4	4	4				**Reset Bit Static** Reset dst bit specified by b
		IR	11						
		DA	13	14	16				
		X	14	14	17				
RES **RESB**	dst,R	R	10	10	10				**Reset Bit Dynamic** Reset dst bit specified by contents R
RESFLG	flag		7	7	7				**Reset Flag** (Any combination of C, Z, S, P/V)
RET	cc		10	10	13	(taken)			**Return Conditional** If cc is true: PC ← @ SP Autoincrement SP
			7	7	7	(not taken)			
RL **RLB**	dst,n	R	6 for n = 1						**Rotate Left** by n bits (n = 1, 2)
		R	7 for n = 2						
RLC **RLCB**	dst,n	R	6 for n = 1						**Rotate Left through Carry** by n bits (n = 1, 2)
		R	7 for n = 2						
RLDB	R,src	R	9	9	9				**Rotate Digit Left**
RR **RRb**	dst,n	R	6 for n = 1						**Rotate Right** by n bits (n = 1, 2)
		R	7 for n = 2						
RRC **RRCB**	dst,n	R	6 for n = 1						**Rotate Right through Carry** by n bits (n = 1, 2)
		R	7 for n = 2						

***Privileged instruction. Executed in system mode only.**

Mnemonics	Operands	Addr. Modes	Clock Cycles: Word, Byte NS	Word, Byte SS	Word, Byte SL	Long Word NS	Long Word SS	Long Word SL	Operation
RRDB	R,src	R	9	9	9				**Rotate Digit Right**
SBC **SBCB**	R,src	R	5	5	5				**Subtract with Carry** R ← R − src − carry
SC	src	IM	33		39				**System Call** Autodecrement SP @ SP ← old PS Push instruction PS ← System Call PS
SDA **SDAB** **SDAL**	dst,R	R	(15 + 3n)			(15 + 3n)			**Shift Dynamic Arithmetic** Shift dst left or right by contents of R
SDL **SDLB** **SDLL**	dst,R	R	(15 + 3n)			(15 + 3n)			**Shift Dynamic Logical** Shift dst left or right by contents of R
SET **SETB**	dst,b	R IR DA X	4 11 13 14	4 14 14	4 16 17				**Set Bit Static** Set dst bit specified by b
SET **SETB**	dst,R	R	10	10	10				**Set Bit Dynamic** Set dst bit specified by contents of R
SETFLG	flag	X	7	7	7				**Set Flag** (Any combination of C, Z, S, P/V)
SIN* **SINB***	R,src	DA	12	12	12				**Special Input** R ← src
SIND* **SINDB***	dst,src,R	IR	21						**Special Input and Decrement** dst ← src Autodecrement dst address R ← R − 1
SINDR* **SINDRB***	dst,src,R	IR	(11 + 10n)						**Special Input, Decrement and Repeat** dst ← src Autodecrement dst address R ← R − 1 Repeat until R = 0
SINI* **SINIB***	dst,src,R	IR	21						**Special Input and Increment** dst ← src Autoincrement dst address R ← R − 1
SINIR* **SINIRB***	dst,src,R	IR	(11 + 10n)						**Special Input, Increment and Repeat** dst ← src Autoincrement dst address R ← R − 1 Repeat until R = 0
SLA **SLAB** **SLAL**	dst,n	R	(13 + 3n)			(13 + 3n)			**Shift Left Arithmetic** by n bits
SLL **SLLB** **SLLL**	dst,n	R	(13 + 3n)			(13 + 3n)			**Shift Left Logical** by n bits

***Privileged instruction. Executed in system mode only.**

Mnemonics	Operands	Addr. Modes	Clock Cycles: Word, Byte NS	SS	SL	Long Word NS	SS	SL	Operation
SOTDR* **SOTDRB***	dst,src,R	IR	(11 + 10 n)						**Special Output, Decr. and Repeat** dst ← src Autodecrement src address R ← R − 1 Repeat until R = 0
SOTIR* **SOTIRB***	dst,src,R	R	(11 + 10 n)						**Special Output, Incr. and Repeat** dst ← src Autoincrement src address R ← R − 1 Repeat until R = 0
SOUT* **SOUTB***	dst,src	DA	12	12	12				**Special Output** dst ← src
SOUTD* **SOUTDB***	dst,src,R	IR	21						**Special Output and Decrement** dst ← src Autodecrement src address R ← R − 1
SOUTI* **SOUTIB***	dst,src,R	IR	21						**Special Output and Increment** dst ← src Autoincrement src address R ← R − 1
SRA **SRAB** **SRAL**	dst,n	R	(13 + 3 n)			(13 + 3 n)			**Shift Right Arithmetic** by n bits
SRL **SRLB** **SRLL**	dst,n	R	(13 + 3 n)			(13 + 3 n)			**Shift Right Logical** by n bits
SUB **SUBB** **SUBL**	R,src	R	4	4	4	8	8	8	**Subtract** R ← R − src
		IM	7	7	7	14	14	14	
		IR	7			14			
		DA	9	10	12	15	16	18	
		X	10	10	13	16	16	19	
TCC **TCCB**	cc,dst	R	5	5	5				**Test Condition Code** Set LSB if cc is true
TEST **TESTB**	dst	R	7	7	7	13	13	13	**Test** dst OR 0
		IR	8			13			
		DA	11	12	14	16	17	19	
		X	12	12	15	17	17	20	

***Privileged instructions. Executed in system mode only.**

Mnemonics	Operands	Addr. Modes	Clock Cycles: Word, Byte NS	SS	SL	Long Word NS	SS	SL	Operation
TRDB	dst,src,R	IR	25						**Translate and Decrement** dst ← src(dst) Autodecrement dst address R ← R − 1
TRDRB	dst,src,R	IR	(11 + 14n)						**Translate, Decrement and Repeat** dst ← src(dst) Autodecrement dst address R ← R − 1 Repeat until R = 0
TRIB	dst,src,R	IR	25						**Translate and Increment** dst ← src(dst) Autoincrement dst address R ← R − 1
TRIRB	dst,src,R	IR	(11 + 14n)						**Translate, Increment and Repeat** dst ← src(dst) Autoincrement dst address R ← R − 1 Repeat until R = 0
TRTDB	src1,src2,R	IR	25						**Translate and Test, Decrement** RH1 → src2 (src1) Autodecrement src 1 address R ← R − 1
TRTDRB	src1,src2,R	IR	(11 + 14n)						**Translate and Test, Decr. and Repeat** RH1 ← src2 (src1) Autodecrement src 1 address R ← R − 1 Repeat until R = 0 or RH1 ≠ 0
TRTIB	src1,src2,R	IR	25						**Translate and Test, Increment** RH1 ← src2 (src1) Autoincrement src address R ← R − 1
TRTIRB	src1,src2,R	IR	(11 + 14n)						**Translate and Test, Incr. and Repeat** RH1 ← src2 (src1) Autoincrement src 1 address R ← R1 Repeat until R = 0 or RH1 ≠ 0
TSET **TSETB**	dst	R IR DA X	7 11 14 15	7 15 15	7 17 18				**Test and Set** S flag ← MSB of dst dst ← all 1s
XOR **XORB**	R,src	R IM IR DA X	4 7 7 9 10	4 7 10 10	4 7 12 13				**Exclusive OR** R ← R XOR src

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** R ← IR R ← IM	**ADD** R ← IR R ← IM	**SUBB** R ← IR R ← IM	**SUB** R ← IR R ← IM	**ORB** R ← IR R ← IM	**OR** R ← IR R ← IM	**ANDB** R ← IR R ← IM	**AND** R ← IR R ← IM	**XORB** R ← IR R ← IM	**XOR** R ← IR R ← IM	**CPB** R ← IR R ← IM	**CP** R ← IR R ← IM	**See Table 1**	**See Table 1**	**EXTEND INST**	**EXTEND INST**
1	**CPL** R ← IR R ← IM	**PUSHL** IR ← IR	**SUBL** R ← IR R ← IM	**PUSH** IR ← IR	**LDL** R ← IR R ← IM	**POPL** IR ← IR	**ADDL** R ← IR R ← IM	**POP** IR ← IR	**MULTL** R ← IR R ← IM	**MULT** R ← IR R ← IM	**DIVL** R ← IR R ← IM	**DIV** R ← IR R ← IM	**See Table 2**	**LDL** IR ← R	**JP** PC ← IR	**CALL** PC ← IR
2	**LDB** R ← IR R ← IM	**LD** R ← IR R ← IM	**RESB** IR ← IM R ← R	**RES** IR ← IM R ← R	**SETB** IR ← IM R ← R	**SET** IR ← IM R ← R	**BITB** IR ← IM R ← R	**BIT** IR ← IM R ← R	**INCB** IR ← IM	**INC** IR ← IM	**DECB** IR ← IM	**DEC** IR ← IM	**EXB** R ↔ IR	**EX** R ↔ IR	**LDB** IR ← R	**LD** IR ← R
3	**LDB** R ← BA **LDRB** R ← RA	**LD** R ← BA **LDR** R ← RA	**LDB** BA ← R **LDRB** RA ← R	**LD** BA ← R **LDR** RA ← R	**LDA** R ← BA **LDAR** R ← RA	**LDL** R ← BA **LDRL** R ← RA	**RSVD**	**LDL** BA ← R **LDRL** RA ← R	**RSVD**	**LDPS** IR	**See Table 3A**	**See Table 3B**	**INB** R ← IR	**IN** R ← IR	**OUTB** IR ← R	**OUT** IR ← R
4	**ADDB** R ← X R ← DA	**ADD** R ← X R ← DA	**SUBB** R ← X R ← DA	**SUB** R ← X R ← DA	**ORB** R ← X R ← DA	**OR** R ← X R ← DA	**ANDB** R ← X R ← DA	**AND** R ← X R ← DA	**XORB** R ← X R ← DA	**XOR** R ← X R ← DA	**CPB** R ← X R ← DA	**CP** R ← X R ← DA	**See Table 1**	**See Table 1**	**EXTEND INST**	**EXTEND INST**
5	**CPL** R ← X R ← DA	**PUSHL** IR ← X IR ← DA	**SUBL** R ← X R ← DA	**PUSH** IR ← X IR ← DA	**LDL** R ← X R ← DA	**POPL** IR → X IR → DA	**ADDL** R ← X R ← DA	**POP** IR → X IR → DA	**MULTL** R ← X R ← DA	**MULT** R ← X R ← DA	**DIVL** R ← X R ← DA	**DIV** R ← X	**See Table** 2	**LDL** X ← R DA ← R	**JP** PC ← X PC ← DA	**CALL** PC ← X PC ← DA
6	**LDB** R ← X R ← DA	**LD** R ← X R ← DA	**RESB** X ← IM DA ← IM	**RES** X ← IM DA ← IM	**SETB** X ← IM DA ← IM	**SET** X ← IM DA ← IM	**BITB** X ← IM DA ← IM	**BIT** X ← IM DA ← IM	**INCB** X ← IM DA ← IM	**INC** X ← IM DA ← IM	**DECB** X ← IM DA ← IM	**DEC** X ← IM DA ← IM	**EXB** R ↔ X R ↔ DA	**EX** R ↔ X R ↔ DA	**LDB** X ← R DA ← R	**LD** X ← R DA ← R
7	**LDB** R ← BX	**See Table 7**	**LDB** BX ← R	**LD** BX ← R	**LDA** R ← BX	**LDL** R ← BX	**LDA** R ← X R ← DA	**LDL** BX ← R	**RSVD**	**LDPS** PS ← X PS ← DA	**HALT**	**See Table 7**	**EI** **DI**	**See Table 7**	**RSVD**	**SC**
8	**ADDB** R ← R	**ADD** R ← R	**SUBB** R ← R	**SUB** R ← R	**ORB** R ← R	**OR** R ← R	**ANDB** R ← R	**AND** R ← R	**XORB** R ← R	**XOR** R ← R	**CPB** R ← R	**CP** R ← R	**See Table 1**	**See Table 1**	**EXTEND INST.**	**EXTEND INST.**
9	**CPL** R ← R	**PUSHL** IR ← R	**SUBL** R ← R	**PUSH** IR ← R	**LDL** R ← R	**POPL** R ← IR	**ADDL** R ← R	**POP** R ← IR	**MULTL** R ← R	**MULT** R ← R	**DIVL** R ← R	**DIV** R ← R	**See Table 2**	**RSVD**	**RET** PC ← (SP)	**RSVD**
A	**LDB** R ← R	**LD** R ← R	**RESB** R ← IM	**RES** R ← IM	**SETB** R ← IM	**SET** R ← IM	**BITB** R ← IM	**BIT** R ← IM	**INCB** R ← IM	**INC** R ← IM	**DECB** R ← IM	**DEC** R ← IM	**EXB** R ↔ R	**EX** R ↔ R	**TCCB** R	**TCC** R
B	**DAB** R	**EXTS** **EXTSB** **EXTSL** R	**See Table 4**	**See Table 4**	**ADCB** R ← R	**ADC** R ← R	**SBCB** R ← R	**SBC** R ← R	**See Table 5**	**RSVD**	**See Table 6**	**See Table 6**	**RRDB** R	**LDK** R ← IM	**RLDB** R	**RSVD**
C	**LDB** R ← IM	→														→
D	**CALR** PC ← RA	→														→
E	**JR** PC ← RA	→														→
F	**DJNZ** **DBJNZ** 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	OC	OD	4C	4D	8C	8D
0	**COMB** IR	**COM** IR	**COMB** X DA	**COM** X DA	**COMB** R	**COM** R
1	**CPB** IR,IM	**CP** IR,IM	**CPB** X,IM DA,IM	**CP** X,IM DA,IM	**LDCTLB** R←FLGS	**SETFLG**
2	**NEGB** IR	**NEG** IR	**NEGB** X DA	**NEG** X DA	**NEGB** R	**NEG** R
3	RSVD	RSVD	RSVD	RSVD	RSVD	**RESFLG**
4	**TESTB** IR	**TEST** IR	**TESTB** X DA	**TEST** X DA	**TESTB** R	**TEST** R
5	**LDB** IR←IM	**LD** IR←IM	**LDB** X←IM DA←IM	**LD** X←IM DA←IM	RSVD	**COMFLG**
6	**TSETB** IR	**TSET** IR	**TSETB** X DA	**TSET** X DA	**TSETB** R	**TSET** R
7	RSVD	RSVD	RSVD	RSVD	RSVD	**NOP**
8	**CLRB** IR	**CLR** IR	**CLRB** X DA	**CLR** X DA	**CLRB** R	**CLR** R
9		**PUSH** IM			**LDCTLB** FLGS←R	

Table 1. Upper Instruction Byte

LOWER NIBBLE (HEX), LOWER INSTRUCTION BYTE	1C	5C	9C
0	RSVD	RSVD	RSVD
1	**LDM** R←IR	**LDM** R←X R←DA	
8	**TESTL** IR	**TESTL** X DA	**TESTL** R
9	**LDM** IR←R	**LDM** X←R DA←R	

Table 2. Upper Instruction Byte

LOWER NIBBLE (HEX), LOWER INSTRUCTION BYTE	3A	3B
0	**INIB** IR←IR **INIRB** IR←IR	**INI** IR←IR **INIR** IR←IR
1	**SINIB** IR←IR **SINIRB** IR←IR	**SINI** IR←IR **SINIR** IR←IR
2	**OUTIB** IR←IR **OTIRB** IR←IR	**OUTI** IR←IR **OUTIR** IR←IR
3	**SOUTIB** IR←IR **SOTIRB** IR←IR	**SOUTI** IR←IR **SOTIR** IR←IR
4	**INB** R←DA	**IN** R←DA
5	**SINB** R←DA	**SIN** R←DA
6	**OUTB** DA←R	**OUT** DA←R
7	**SOUTB** DA←R	**SOUT** DA←R
8	**INDB** IR←IR **INDRB** IR←IR	**IND** IR←IR **INDR** IR←IR
9	**SINDB** IR←IR **SINDRB** IR←IR	**SIND** IR←IR **SINDR** IR←IR
A	**OUTDB** IR←IR **OTDRB** IR←IR	**OUTD** IR←IR **OTDR** IR←IR
B	**SOUTDB** IR←IR **SOTDRB** IR←IR	**SOUTD** IR←IR **SOTDR** IR←IR

Table 3. Upper Instruction Byte

LOWER NIBBLE (HEX), LOWER INSTRUCTION BYTE	B2	B3
0	**RLB** (1 bit) R	**RL** (1 bit) R
1	**SLLB** R **SRLB** R	**SLL** R **SRL** R
2	**RLB** (2 bits) R	**RL** (2 bits) R
3	**SDLB** R	**SDL** R
4	**RRB** (1 bit) R	**RR** (1 bit) R
5	RSVD	**SLLL** R **SRLL**
6	**RRB** (2 bits) R	**RR** (2 bits) R
7	RSVD	**SDLL** R
8	**RLCB** (1 bit) R	**RLC** (1 bit) R
9	**SLAB** R **SRAB** R	**SLA** R **SRA** R
A	**RLCB** (2 bits) R	**RLC** (2 bits) R
B	**SDAB** R	**SDA** R
C	**RRCB** (1 bit) R	**RRC** (1 bit) R
D	RSVD	**SLAL** R **SRAL**
E	**RRCB** (2 bits) R	**RRC** (2 bits) R
F	RSVD	**SDAL** R

Table 4.
Upper Instruction Byte

LOWER NIBBLE (HEX), LOWER INSTRUCTION BYTE	B8
0	**TRIB** IR
1	RSVD
2	**TRTIB** IR
3	RSVD
4	**TRIRB** IR
5	RSVD
6	**TRTIRB** IR
7	RSVD
8	**TRDB** IR
9	RSVD
A	**TRTDB** IR
B	RSVD
C	**TRDRB** IR
D	RSVD
E	**TRTDRB** IR
F	RSVD

Table 5.
Upper Instruction Byte

BA	BB
CPIB IR	**CPI** IR
LDIB IR←IR **LDIRB** IR←IR	**LDI** IR←IR **LDIR** IR←IR
CPSIB IR	**CPSI** IR
RSVD	RSVD
CPRIB IR	**CPIR** IR
RSVD	RSVD
CPSIRB IR	**CPSIR** IR
RSVD	RSVD
CPDB IR	**CPD** IR
LDDB IR←IR **LDDRB** IR←IR	**LDD** IR←IR **LDDR** IR←IR
CPSDB IR	**CPSD** IR
RSVD	RSVD
CPDRB IR	**CPDR** IR
RSVD	RSVD
CPSDRB IR	**CPSDR** IR
RSVD	RSVD

Table 6.
Upper Instruction Byte

7B	7D
IRET PC←(SSP)	RSVD
RSVD	RSVD
RSVD	**LDCTL** R←FCW
RSVD	**LDCTL** R←RFRSH
RSVD	**LDCTL** R← PSAPSEG
RSVD	**LDCTL** R← PSAPOFF
RSVD	**LDCTL** R←NSPSEG
RSVD	**LDCTL** R←NSPOFF
MSET	RSVD
MRES	RSVD
MBIT	**LDCTL** FCW←R
RSVD	**LDCTL** RFRSH←R
	LDCTL PSAPSEG ←R
MREQ R	**LDCTL** PSAPOFF ←R
RSVD	**LDCTL** NSPSEG←R
RSVD	**LDCTL** NSPOFF←R

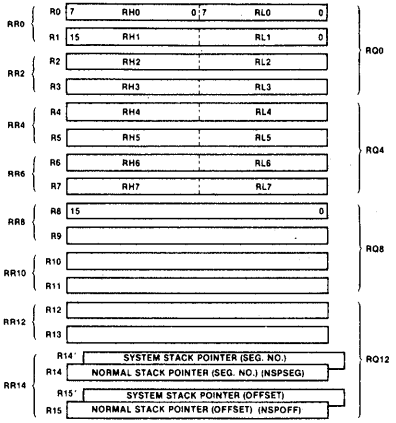
Table 7.
Upper Instruction Byte

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]

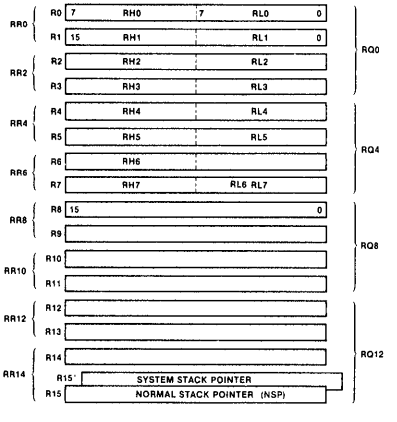
1. Flag affected only for byte operation.
2. Flag modified only if specified by the instruction.

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.



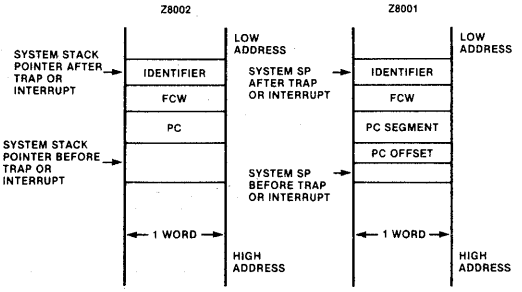
Z8001 General Purpose Registers



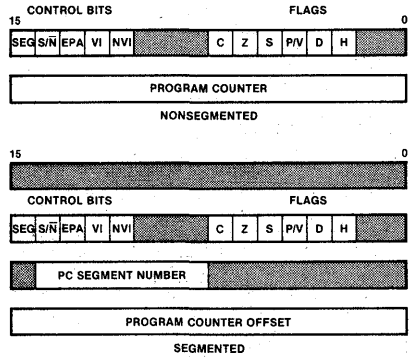
Z8002 General Purpose Registers

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

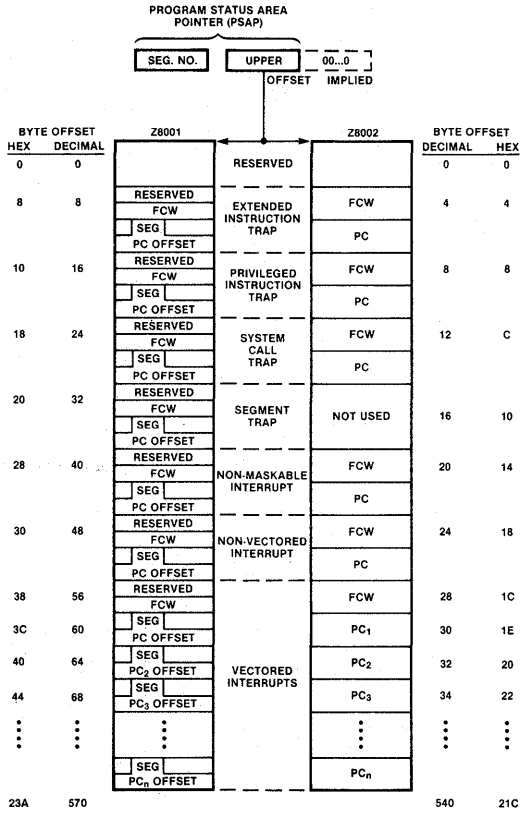
Binary Encoding for Register Fields



Format of Saved Program Status in the System Stack



Program Status Blocks



Program Status Area

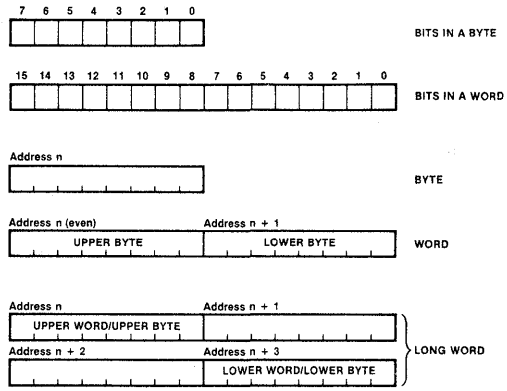
Condition Codes

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			Operand Value
	In the Instruction	In a Register	In Memory	
R				
Register	REGISTER ADDRESS →	OPERAND		The content of the register
IM				
Immediate	OPERAND			In the instruction
***IR**				
Indirect Register	REGISTER ADDRESS →	ADDRESS →	OPERAND	The content of the location whose address is in the register
DA				
Direct Address	ADDRESS →		OPERAND	The content of the location whose address is in the instruction
***X**				
Index	REGISTER ADDRESS → BASE ADDRESS	INDEX → (+)	OPERAND	The content of the location whose address is the address in the instruction plus the content of the working register.
RA				
Relative Address	DISPLACEMENT →	PC VALUE → (+)	OPERAND	The content of the location whose address is the content of the program counter, offset by the displacement in the instruction
***BA**				
Base Address	REGISTER ADDRESS → DISPLACEMENT →	BASE ADDRESS → (+)	OPERAND	The content of the location whose address is the address in the register, offset by the displacement in the instruction
***BX**				
Base Index	REGISTER ADDRESS → REGISTER ADDRESS →	BASE ADDRESS → INDEX → (+)	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 mulitplication 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

Glossary of Terms

address: A number 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 based 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: Based Indexed addressing is similar to Based 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 non-segmented 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: One of the two Program Status registers; it contains *flags* (q.v.) and bits that control the operation of the CPU.

Immediate (I) 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 Z8001.

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 Z8001 CPU. 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 Z8001 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 word.

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: 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 *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 Z8001 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 Z8001 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 1011. 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, unimplemented instruction and privileged instructions executed in normal mode, and an external trap, the segmentation 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.

$\overline{\text{WAIT}}$ 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.

Zilog Sales Offices

West

Sales & Technical Center
Zilog, Incorporated
1333 Lawrence Expressway
Suite 400
Santa Clara, CA 95051
Tele: (408) 446-9848
TWX: 910-338-7621

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

Sales & Technical Center
Zilog, Incorporated
15643 Sherman Way
Suite 430
Van Nuys, CA 91406
Tele: (213) 989-7484
TWX: 910-495-1765

Midwest

Sales & Technical Center
Zilog, Incorporated
890 East Higgins Road
Suite 147
Schaumburg, IL 60195
Tele: (312) 885-8080
TWX: 910-291-1064

South

Sales & Technical Center
Zilog, Incorporated
2711 Valley View, Suite 103
Dallas, TX 75234
Tele: (214) 243-6550
TWX: 910-860-5850

Technical Center
Zilog, Incorporated
1442 U.S. Hwy 19 South
Suite 135
Clearwater, FL 33516
Tele: (813) 535-5571

East

Sales & Technical Center
Zilog, Incorporated
Corporate Place
99 South Bedford St.
Burlington, MA 01803
Tele: (617) 273-4222
TWX: 710-332-1726

Sales & Technical Center
Zilog, Incorporated
110 Gibraltar Road
Horsham, PA 19044
Tele: (215) 441-8282
TWX: 510-665-7077

United Kingdom

Zilog (U.K.) Limited
Babbage House, King Street
Maidenhead SL6 1DU
Berkshire, United Kingdom
Tele: (628) 36131
TELEX: 848609

West Germany

Zilog GmbH
Zugspitzstrasse 2a
D-8011 Vaterstetten
Munich, West Germany
Tele: 08106 4035
TELEX: 529110 Zilog d.

Japan

Zilog, Japan KK
Linden Sky Heights
Bldg. 1F
13-2 Sakuragaoka-Machi
Shibuya-Ku Tokyo 105
Japan
Tele: (813) 496-4428
TWX: 781-23723 Lawright

Zilog, Inc. 10340 Bubb Road, Cupertino, California 95014 Telephone (408)446-4666 TWX 910-338-7621

00-2010-C Printed in USA