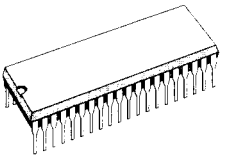
22 1993

SGS-THOMSON MICROELECTRONICS

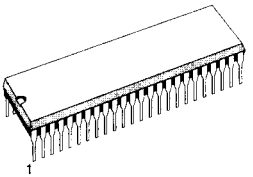
Z8001 SEGCPU
Z8002 CPU

CENTRAL PROCESSING UNIT

- Regular, easy-to-use architecture.
- Instruction set more powerful than many mini-computers.
- Directly addresses 8M bytes.
- Eight user-selectable addressing modes.
- Seven data types that range from bits to 32-bit long words and word strings.
- System and Normal operating modes.
- Separate code, data and stack spaces.
- Sophisticated interrupt structure.
- Resource-sharing capabilities for multiprocessing systems.
- Multi-programming support.
- Strong compiler support.
- Memory management and protection provided by Z8010 Memory Management Unit.
- 32-bit operations, including signed multiply and divide.
- Z-BUS compatible.
- 4,6 and 10MHz clock rate.



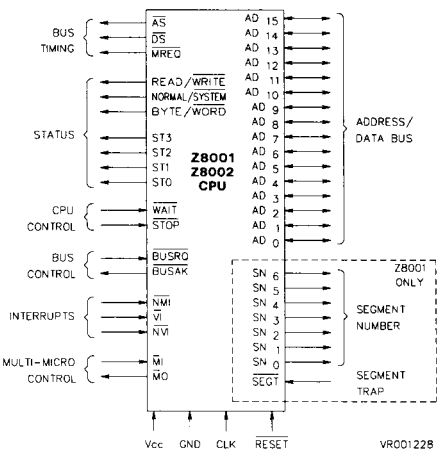
PDIP40



PDIP48

(Ordering information at the end of the datasheet)

Figure 1-1. Logic Function



GENERAL DESCRIPTION

The 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 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 the 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

GENERAL DESCRIPTION (Continued)

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 this chapter.

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 this chapter. 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 external devices (called Extended Processing Units - EPUs). In general, an EPU is dedicated to performing complex and time-consuming tasks so as to unburden the 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 this chapter.

Architecture

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 mini-computers. 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 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.

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 a 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.

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,

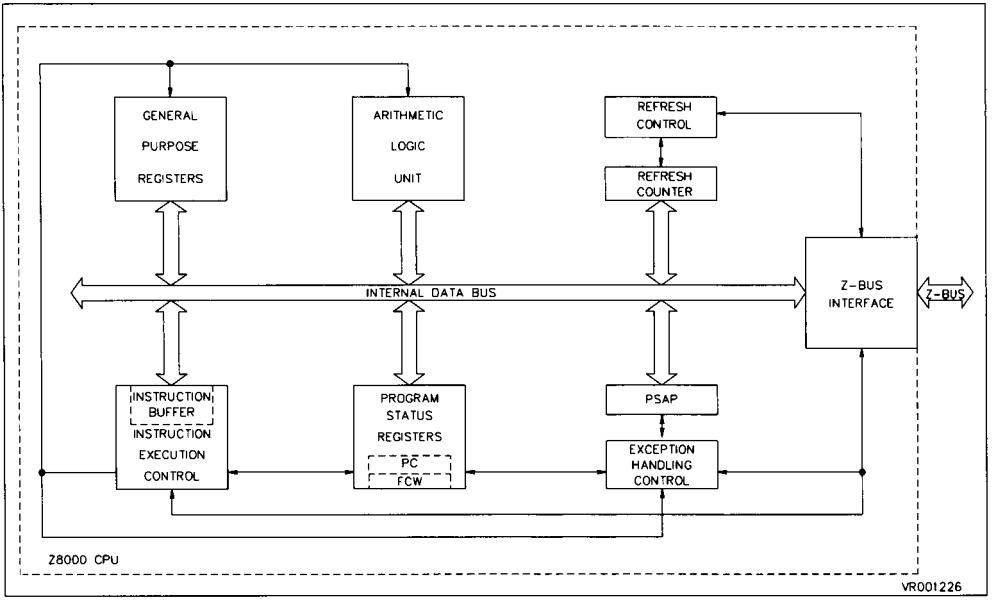
and typical high level language and operating system functions. The general philosophy of the instruction set is two-operand register-memory operations, which include as a special subset register-register operations. However, to improve code density, a few memory-memory operations are used for string manipulation. The two-address format reflects the most frequently occuring operations (such as A ← A + B). Also, having one of the operands in a rapidly accessible general-purpose register facilitates the use of intermediate results generated during a calculation.

The majority of operations deal with byte, word, or long-word operands, thereby providing a high degree of regularity. 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.

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 (in Chapter ADDRESSING MODES). Stacks are supported by the instruction set and by an external device (the Memory Management Unit Z8010 MMU) available with the Z8001.

Figure 1-2. Z8000 CPU Functional Block Diagram



Architecture (Continued)

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 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.)

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.

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 intented 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 occurence of interrupts or traps is always pushed on to the system stack before the new program status is loaded.

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 peripheral 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.

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.

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.

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 Z8038 FIO, have been designed to allow easy asynchronous communication between different CPUs.

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).

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.

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.

Architecture (Continued)

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.

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.

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 accumulator, index register, or base register. With a few exceptions, all basic addressing modes can be used with all instructions, as can the various data types.

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

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.

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 : non-maskable, 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 occurence 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.

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.

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 non-segmented 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 severals 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.

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 features. When the CPU encouters 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 APU 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.

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.

Register organization

The Z800 CPU is a register-oriented machine that offers sixteen 16-bit general-purpose registers and a set of special system registers. All general-purpose registers can be used as accumulators and all but one as index registers or memory pointers.

Register flexibility is created by grouping and overlapping multiple registers (Figure 1-3a and 1-3b). For byte operations, the first eight 16-bit registers (R0...R7) are treated as sixteen 8-bit registers (RL0, RH0,..., RL7, RH7). The sixteen 16-bit registers are grouped in pairs (RR0 ... RR14) to form 32-bit long-word registers. Similarly, the register set is grouped in quadruples (RQ0 ... RQ12) to form 64-bit registers.

Figure 1-3a. Z8001 General-Purpose Registers

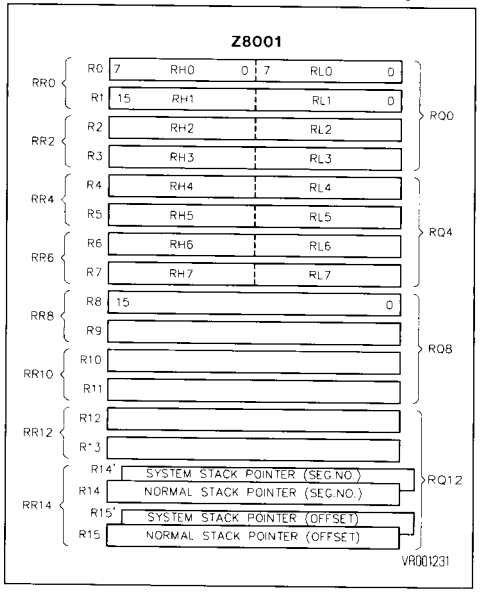
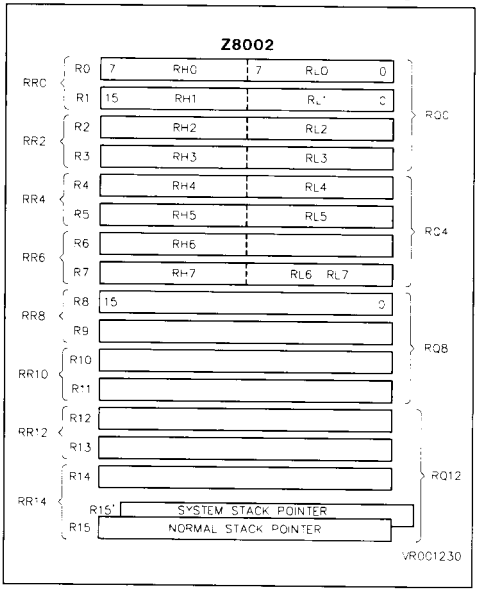


Figure 1-3b. Z8002 General-Purpose Registers



Instruction Set Summary

The Z8000 provides the following types of instructions :

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

Load and Exchange

Mnemonics	Operands	Addr. Modes	Clock Cycles (1) Word, Byte NS	SS	SL	Long Word NS	SS	SL	Operation
CLR **CLRB**	dst	R	7	-	-				**Clear** dst ← 0
		IR	8	-	-				
		DA	11	12	14				
		X	12	12	15				
EX **EXB**	R,src	R	6	-	-				**Exchange** R ↔ src
		IR	12	-	-				
		DA	15	16	18				
		X	16	16	19				
LD **LDB** **LDL**	R,src	R	3	-	-	5	-	-	**Load into Register** R ← src
		IM	7	-	-	11	-	-	
		IM	5	(byte	only)				
		IR	7	-	-	11	-	-	
		DA	9	10	12	12	13	15	
		X	10	10	13	13	13	16	
		BA	14	-	-	17	-	-	
		BX	14	-	-	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	-	-	17	-	-	
		BX	14	-	-	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	-	-				
		BX	15	-	-				
LDAR	R,src	RA	15	-	-				**Load Address Relative** R ← source address
LDK	R,src	IM	5	-	-				**Load Constant** R ← n (n = 0 ... 15)
LDM	R,src,n	IR	11	-	-	|			**Load Multipe** R ← 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	|			

Load and Exchange (Continued)

Mnemonics	Operands	Addr. Modes	Clock Cycles (1) Word, Byte NS	SS	SL	Long Word NS	SS	SL	Operation
LDR **LDRB** **LDRL**	R,src	RA	14	-	-	17	-	-	**Load Relative** R ← src (range - 32768 ... + 32767)
LDR **LDRB** **LDRL**	dst,R	RA	14	-	-	17	-	-	**Load Relative** (Store Relative) dst ← R (range - 32768 ... + 32767)
POP **POPL**	dst,IR	R IR DA X	8 12 16 16	- - 16 16	- - 18 19	12 19 23 23	- - 23 23	- - 25 26	**Pop** dst ← IR Autoincrement contents of R
PUSH **PUSHL**	IR,src	R IM IR DA X	9 12 13 14 14	- - - 14 14	- - - 16 17	12 - 20 21 21	- - - 21 21	- - - 23 24	**Push** IR ← src Autodecrement contents of R

Arithmetic

ADC **ADCB**	R,src	R	5	-	-				**Add with Carry** R ← R + src + carry
ADD **ADDB** **ADDL**	R,src	R IM IR DA X	4 7 7 9 10	- - - 10 10	- - - 12 13	8 14 14 15 16	- - - 16 16	- - - 18 19	**Add** R ← R + src
CP **CPB** **CPL**	R,src	R IM IR DA X	4 7 7 9 10	- - - 10 10	- - - 12 13	8 14 14 15 16	- - - 16 16	- - - 18 19	**Compare with Register** R - src
CP **CPB**	dst,IM	IR DA X	11 14 15	- 15 15	- 17 18				**Compare with Immediate** dst - IM
DAB	dst	R	5	-	-				**Decimal Adjust**
DEC **DECB**	dst,n	R IR DA X	4 11 13 14	- - 14 14	- - 16 17				**Decrement by n** dst ← dst - n (n = 1 ... 16)
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 ← remainder Long Word : $R_{n+2,n+3} \leftarrow R_{n...n+3}$ + src $R_{n,n+1}$ ← remainder
EXTS **EXTSB** **EXTSL**	dst	R	11	-	-	11	-	-	**Extend Sign** Extend sign of low order half of dst through high order half of dst

Arithmetic (Continued)

Mnemonics	Operands	Addr. Modes	Clock Cycles (1) Word, Byte NS	SS	SL	Long Word NS	SS	SL	Operation
INC **INCB**	dst,n	R IR DA X	4 11 13 14	- - 14 14	- - 16 17				**Increment by n** dst ← dst + n (n = 1 ... 16)
MULT **MULTL**	R,src	R IM IR DA X	70 70 70 71 72	- - - 72 72	- - - 74 75	282* 282* 282* 283* 284*	- - - 284* 284*	- - - 286* 287*	**Multiply** (signed) Word : $R_{n,n+1} \leftarrow R_{n+1}$ + src Long Word : $R_{n...n+3} \leftarrow R_{n+2,n+3}$ *Plus seven cycles for each 1 in the multiplicand
NEG **NEGB**	dst	R IR DA X	7 12 15 16	- - 16 16	- - 18 19				**Negate** dst ← 0 - dst
SBC **SBCB**	R,src	R	5	-	-				**Subtract with Carry** R ← R - src - carry
SUB **SUBB** **SUBL**	R,src	R IM IR DA X	4 7 7 9 10	- - - 10 10	- - - 12 13	8 14 14 15 16	- - - 16 16	- - - 18 19	**Subtract** R ← R - src

Logical

AND **ANDB**	R,src	R IM IR DA X	4 7 7 9 10	- - - 10 10	- - - 12 13				**AND** R ← R AND src
COM **COMB**	dst	R IR DA X	7 12 15 16	- - 16 16	- - 18 19				**Complement** dst ← NOT dst
OR **ORB**	R,src	R IM IR DA X	4 7 7 9 10	- - - 10 10	- - - 12 13				**OR** R ← R OR src
TCC **TCCB**	cc,dst	R						5	**Test Condition Code** Set LSB if cc is true
TEST **TESTB** **TESTL**	dst	R IR DA X	7 8 11 12	- - 12 12	- - 14 15	13 13 16 17	- - 17 17	- - 19 20	**Test** dst OR 0
XOR **XORB**	R,src	R IM IR DA X	4 7 7 9 10	- - - 10 10	- - - 12 13				**Exclusive OR** R ← R XOR src

Program Control

Mnemonics	Operands	Addr. Modes	Clock Cycles (1) Word, Byte NS	SS	SL	Long Word NS	SS	SL	Operation
CALL	dst	IR	10	-	15				**Call Subroutine** Autodecrement SP @ SP ← PC PC ← dst
		DA	12	18	20				
		X	13	18	21				
CALR	dst	RA	10	-	15				**Call Relative** Autodecrement SP @ SP ← PC PC ← PC + dst (range -4094 to +4096)
DJNZ **DBJNZ**	R,dst	RA	11	-	-				**Decrement and Jump if Non-Zero** R ← R - 1 If R ≠ 0 : PC ← PC + dst (range -254 to 0)
IRET (2)	-		13	-	16				**Interrupt Return** PS ← @ SP Autoincrement SP
JP	cc,dst	IR	10	-	15	(taken)			**Jump Conditional** If cc is true : PC ← dst
		IR	7	-	7	(not taken)			
		DA	7	8	10				
		X	8	8	11				
JR	cc,dst	RA	6	-	-				**Jump Condition Relative** If cc is true : PC ← PC + dst (range -256 to +254)
RET	cc	-	10	-	13	(taken)			**Return Conditional** If cc is true : PC ← @ SP Autoincrement SP
			7	-	7	(not taken)			
SC	src	IM	33	-	39				**System Call** Autodecrement SP @ SP ← old PS Push instruction PS ← System Call PS

Bit Manipulation

BIT **BITB**	dst,B	R	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	-	-				**Test Bit Dynamic** Z flag ← NOT dst bit specified by contents of R
RES **RESB**	dst,b	R	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	-	-				**Reset Bit Dynamic** Reset dst bit specified by contents R

Bit Manipulation (Continued)

Mnemonics	Operands	Addr. Modes	Clock Cycles (1) Word, Byte NS	Word, Byte SS	Word, Byte SL	Long Word NS	Long Word SS	Long Word SL	Operation
SET **SETB**	dst,b	R IR DA X	4 11 13 14	- - 14 14	- - 16 17				**Set Bit Dynamic** Set dst bit specified by b
SET **SETB**	dst,R	R	10	-	-				**Set Bit Dynamic** Set dst bit specified by contents of R
TSET **TSETB**	dst	R IR DA X	7 11 14 15	- - 15 15	- - 17 18				**Test and Set** S flag ← MSB of dst dst ← all 1s

Rotate and Shift

RL **RLB**	dst,n	R R	6 for n = 1 7 for n = 2						**Rotate Left** by n bits (n = 1, 2)
RLC **RLCB**	dst,n	R R	6 for n = 1 7 for n = 2						**Rotate Left through Carry** by n bits (n = 1, 2)
RLDB	R,src	R	9	-	-				**Rotate Digit Left**
RR **RRB**	dst,n	R R	6 for n = 1 7 for n = 2						**Rotate Right** by n bits (n = 1,2)
RRC **RRCB**	dst,n	R R	6 for n = 1 7 for n = 2						**Rotate Right through Carry** by n bits (n = 1,2)
RRDB	R,src	R	9	-	-				**Rotate Digit Right**
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
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
SRA **SRAB** **SRAL**	dst,n	R	(13 + 3n)			(13 + 3n)			**Shift Right Arithmetic** by n bits
SRL **SRLB** **SRLL**	dst,n	R	(13 + 3n)			(13 + 3n)			**Shift Right Logical** by n bits

Block Transfer and String Manipulation

Mnemonics	Operands	Addr. Modes	Clock Cycles (1)						Operation
			Word, Byte			Long Word			
			NS	SS	SL	NS	SS	SL	
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_X - src Autodecrement src address $R_Y \leftarrow R_Y - 1$ 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
LDD **LDDB**	dst,src,R	IR	20	-	-				**Load and Decrement** dst ← src Autodecrement dst and src addresses R ← R - 1
LDDR **LDDRB**	dst,rsc,R	IR	(11 + 9n)						**Load, Decrement and Repeat** dst ← src Autodecrement dst and src addresses R ← R - 1 Repeat until R = 0

Block Transfer and String Manipulation (Continued)

Mnemonics	Operands	Addr. Modes	Clock Cycles (1)						Operation
			Word, Byte			Long Word			
			NS	SS	SL	NS	SS	SL	
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 + 9n)						**Load, Increment and Repeat** dst ← src Autoincrement dst and src addresses R ← R - 1 Repeat until R = 0
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 src1 address R ← R -1
TRTDRB	src1,src2,R	IR	(11 + 14n)						**Translate and Test, Decr. and Repeat** RH1 ← src2 (src1) Autodecrement src1 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 src1 address R ← R - 1
TRTIRB	src1,src2,R	IR	(11 + 14n)						**Translate and Test, Incr. and Repeat** RH1 ← src2 (src1) Autoincrement src1 address R ← R - 1 Repeat until R = 0 or RH1 = 0

Input/Output

Mnemonics	Operands	Addr. Modes	Clock Cycles (1)						Operation
			Word, Byte			Long Word			
			NS	SS	SL	NS	SS	SL	
IN (2) **INB** (2)	R,src	IR DA	10 12	- -	- -				**Input** R ← src
IND (2) **INDB** (2)	dst,src,R	IR	21	-	-				**Input and Decrement** dst ← src Autodecrement dst address R ← R - 1
INDR (2) **INDRB** (2)	dst,src,R	IR	(11 + 10n)						**Input, Decrement and Repeat** dst ← src Autodecrement dst address R ← R - 1 Repeat until R = 0
INI (2) **INIB** (2)	dst,src,R	IR	21	-	-				**Input and Increment** dst ← src Autoincrement dst address R ← R - 1
INIR (2) **INIRB** (2)	dst,src,R	IR	(11 + 10n)						**Input, Increment and Repeat** dst ← src Autoincrement dst address R ← R - 1 Repeat until R = 0
OUT (2) **OUTB** (2)	dst,R	IR DA	10 12	- -	- -				**Output** dst ← R
OUTD (2) **OUTDB** (2)	dst,src,R	IR	21	-	-				**Output and Decrement** dst ← src Autodecrement src address R ← R - 1
OTDR (2) **OTDRB** (2)	dst,src,R	IR	(11 + 10n)						**Output, Decrement and Repeat** dst ← src Autodecrement src address R ← R - 1 Repeat until R = 0
OUTI (2) **OUTIB** (2)	dst,src,R	IR	21	-	-				**Output and Increment** dst ← src Autoincrement src address R ← R - 1
OTIR (2) **OTIRB** (2)	dst,src,R	IR	(11 + 10n)						**Output, Increment and Repeat** dst ← src Autoincrement src address R ← R - 1 Repeat until R = 0

Input/Output (Continued)

Mnemonics	Operands	Addr. Modes	Clock Cycles (1)						Operation
			Word, Byte			Long Word			
			NS	SS	SL	NS	SS	SL	
SIN (2) **SINB** (2)	R,src	DA	12	-	-				**Special Input** R ← src
SIND (2) **SINDB** (2)	dst,src,R	IR	21	-	-				**Special Input and Decrement** dst ← src Autodecrement dst address R ← R - 1
SINDR (2) **SINDRB** (2)	dst,src,R	IR	(11 + 10n)						**Special Input, Decr. and Repeat** dst ← src Autodecrement dst address R ← R - 1 Repeat until R = 0
SINI (2) **SINIB** (2)	dst,src,R	IR	21	-	-				**Special Input and Increment** dst ← src Autoincrement dst address R ← R - 1
SINIR (2) **SINIRB** (2)	dst,src,R	IR	(11 + 10n)						**Special Input, Incr. and Repeat** dst ← src Autoincrement dst address R ← R - 1 Repeat until R = 0
SOUT (2) **SOUTB** (2)	dst,src	DA	12	-	-				**Special Output** dst ← src
SOUTD (2) **SOUTDB** (2)	dst,src,R	IR	21	-	-				**Special Output and Decrement** dst ← src Autodecrement src address R ← R - 1
SOTDR (2) **SOTDRB** (2)	dst,src,R	IR	(11 + 10n)						**Special Output, Decr. and Repeat** dst ← src Autodecrement src address R ← R - 1 Repeat until R = 0
SOUTI (2) **SOUTIB** (2)	dst,src,R	IR	21	-	-				**Special Output and Increment** dst ← src Autoincrement src address R ← R - 1
SOTIR (2) **SOTIRB** (2)	dst,src,R	IR	(11 + 10n)						**Special Output, Incr. and Repeat** dst ← src Autoincrement src address R ← R - 1 Repeat until R = 0

CPU Control

Mnemonics	Operands	Addr. Modes	Clock Cycles (1)						Operation
			Word, Byte			Long Word			
			NS	SS	SL	NS	SS	SL	
COMFLG	flags	-	7	-	-				**Complement Flag** (Any combination of C, Z, S, P/V)
DI (2)	int	-	7	-	-				**Disable Interrupt** (Any combination of NVI, VI)
EI (2)	int	-	7	-	-				**Enable Interrupt** (Any combination of NVI, VI)
HALT (2)	-	-	(8 + 3n)						**HALT**
LDCTL (2)	CTLR,src	R	7	-	-				**Load into Control Register** CTLR ← src
LDCTL (2)	dst,CTLR	R	7	-	-				**Load from Control Register** dst ← CTLR
LDCTLB	FLGR,src	R	7	-	-				**Load into Flag Byte Register** FLGR ← src
LDCTLB	dst,FLGR	R	7	-	-				**Load from Flag Byte Register** dst ← FLGR
LDPS (2)	src	IR DA X	12 16 17	- 20 20	16 22 23				**Load Program Status** PS ← src
MBIT (2)	-	-	7	-	-				**Test Multi-Micro Bit** Set S if $\overline{MI}$ is Low ; reset S if $\overline{MI}$ is High.
MREQ (2)	dst	R	(12 + 7n)						**Multi-Micro Request**
MRES (2)	-	-	5	-	-				**Multi-Micro Reset**
MSET (2)	-	-	5	-	-				**Multi-Micro Set**
NOP	-	-	7	-	-				**No Operation**
RESFLG	flag	-	7	-	-				**Reset Flag** (Any combination of C, Z, S, P/V)
SETFLG	flag	-	7	-	-				**Set Flag** (Any combination of C, Z, S, P/V)

Notes :

1. NS = Non-Segmented
   SS = Segmented Short Offset
   SL = Segmented Long Offset.
2. Privileged instruction. Executed in system mode only.

Condition Codes

Code	Meaning	Flag Settings	CC Field
	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	P/V = 1	0100
NOV	No overflow	P/V = 0	1100
PE	Parity is even	P/V = 1	0100
PO	Parity is odd	P/V = 0	1100
GE	Greater than or equal (signed)	(S XOR P/V) = 0	1001
LT	Less than (signed)	(S XOR P/V) = 1	0001
GT	Greater than (signed)	[Z OR (S XOR P/V)] = 0	1010
LE	Less than or equal (signed)	[Z OR (S XOR P/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 condition codes have identical flag settings and binary fields in the instruction :
Z = EQ, NZ = NE, C = ULT, NC = UGE, OV = PE, NOV = PO

PIN CONFIGURATION

Introduction

This chapter covers the external manifestations (e.g., the activity on the CPU pins) that result from the operations described in prerious chapters. Since the pins are connected to the system bus much of the discussion will center on the bus and bus operations. The Z8000 CPU is designed to be compatible with the 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.

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
- *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
- *EPU transfer.* This type is used to transfer 16 bits of data between the CPU and an EPU
- *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
- *Refresh.* These transactions do not transfer data. They refresh dynamic memory
- *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

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
- *Bus request.* This type is used to request control of the bus to initiate transactions
- *Resource request.* This type is used to request control of a particular system resource
- *Stop request.* This type is used to delay CPU instruction execution

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, relinguishes 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.

Cpu Pins

The CPU pins can be grouped into five categories according to their functions.

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 occuring 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 memroy, 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.

Figure 2-1. Z8001 Pin Configuration

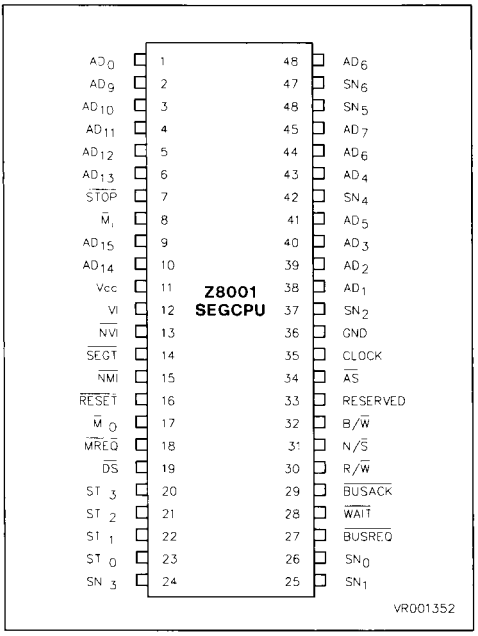
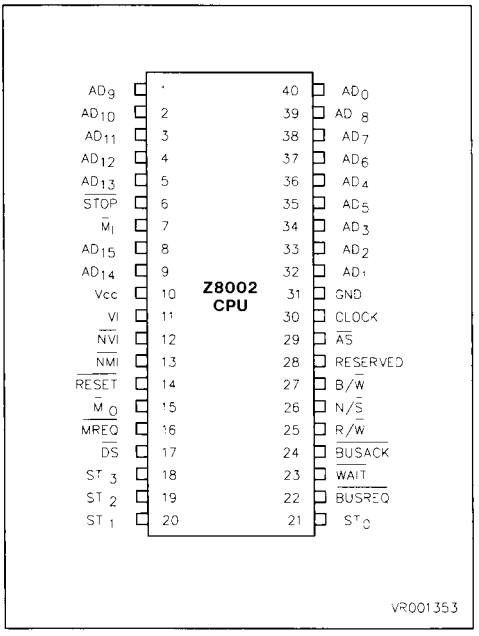


Figure 2-2. Z8002 Pin Configuration



CPU Pins (Continued)

- **$\overline{\text{WAIT}}$.** *(Input, active Low).* A Low on this line indicates that the responding device needs more time to complete a transaction.
- **$\overline{\text{MREQ}}$.** *Memory Request (Output, active Low, 3-state).* A falling edge on this line indicates that the address/data bus is holding a memory address.

Bus Control Pins. These pins carry signals for requesting and obtaining control of the bus from the CPU.

- **$\overline{\text{BUSREQ}}$.** *Bus Request (Input, active Low).* A Low indicates that a bus requester has obtained or is trying to obtain control of the bus.
- **$\overline{\text{BUSACK}}$.** *Bus Acknowledge (Output, active Low).* A Low on this line indicates that the CPU has relinquished control of the bus in response to a bus request.

Interrupt/Trap Pins. These pins convey interrupt and external trap requests to the CPU.

- **$\overline{\text{NMI}}$.** *Non-Maskable Interrupt (Input, Edge activated).* A High-to-Low transition on $\overline{\text{NMI}}$ requests a non-maskable interrupt.
- **$\overline{\text{NVI}}$.** *Non-Vectored Interrupt (Input, active Low).* A Low on this line requests a non-vectored interrupt.
- **$\overline{\text{VI}}$.** *Vectored Interrupt (Input, active Low).* A Low on this line requests a vectored interrupt.
- **$\overline{\text{SEGT}}$.** *Segment Trap (Z8001 only, Input, active Low).* A Low on this line requests a segment trap.

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.

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.

Figure 2-4. Pin Functions

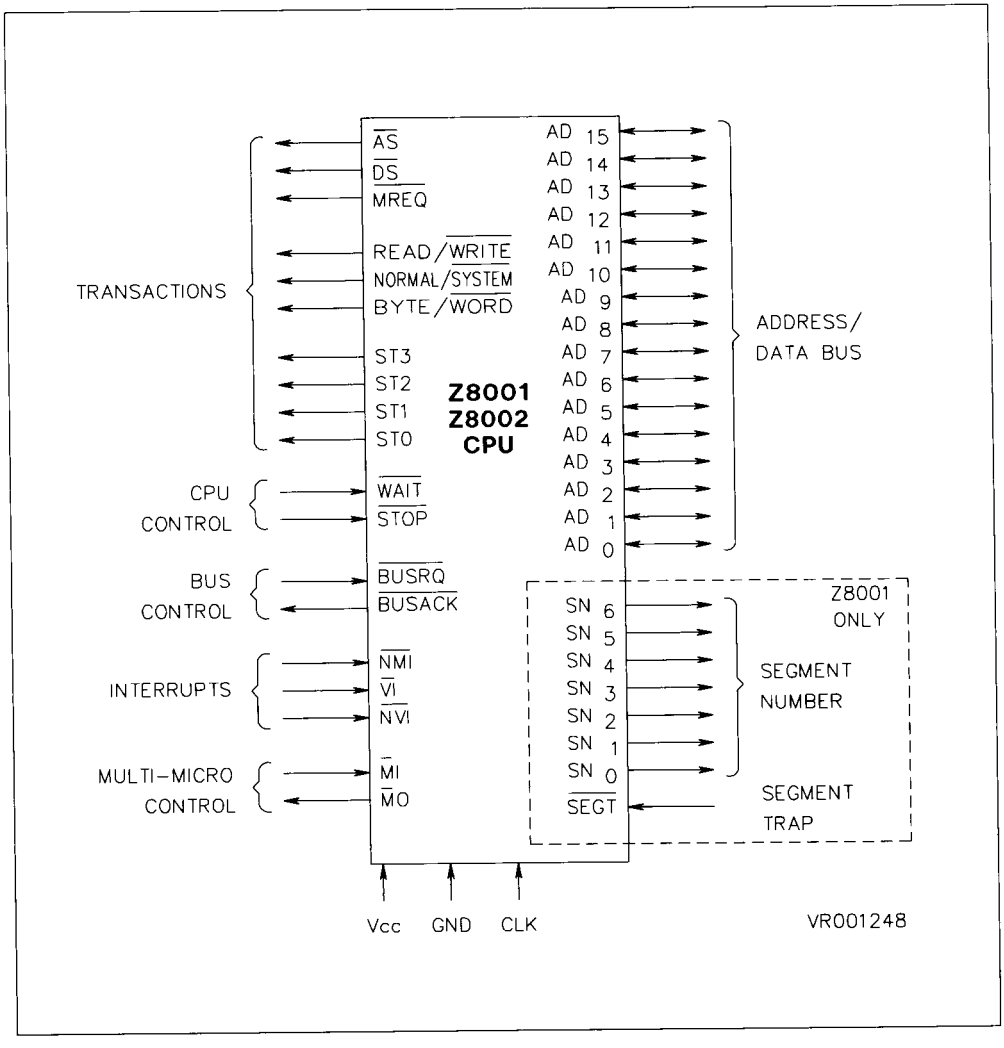
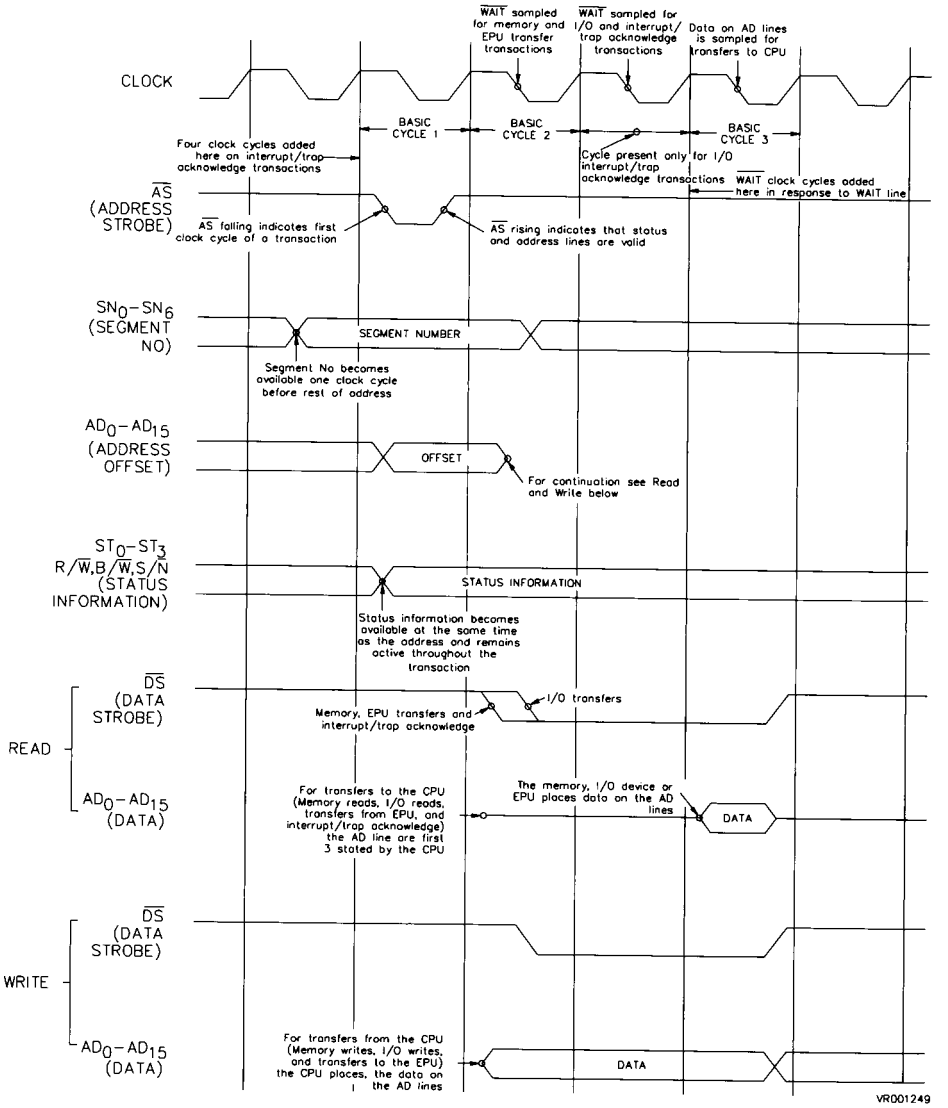


Figure 2-5. Transaction Timing



Transactions

Data transfers to and from the CPU are accomplished through the use of transactions. Figure 2-5 shows the general timing for a transaction.

All transactions start with Address Strobe ($\overline{AS}$) being driven Low and then raised High by the CPU. On the rising edge of $\overline{AS}$, the status lines ST_0 - ST_3 are valid ; these lines indicate the type of transaction being initiated (see Table 2-1 ; 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.

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.

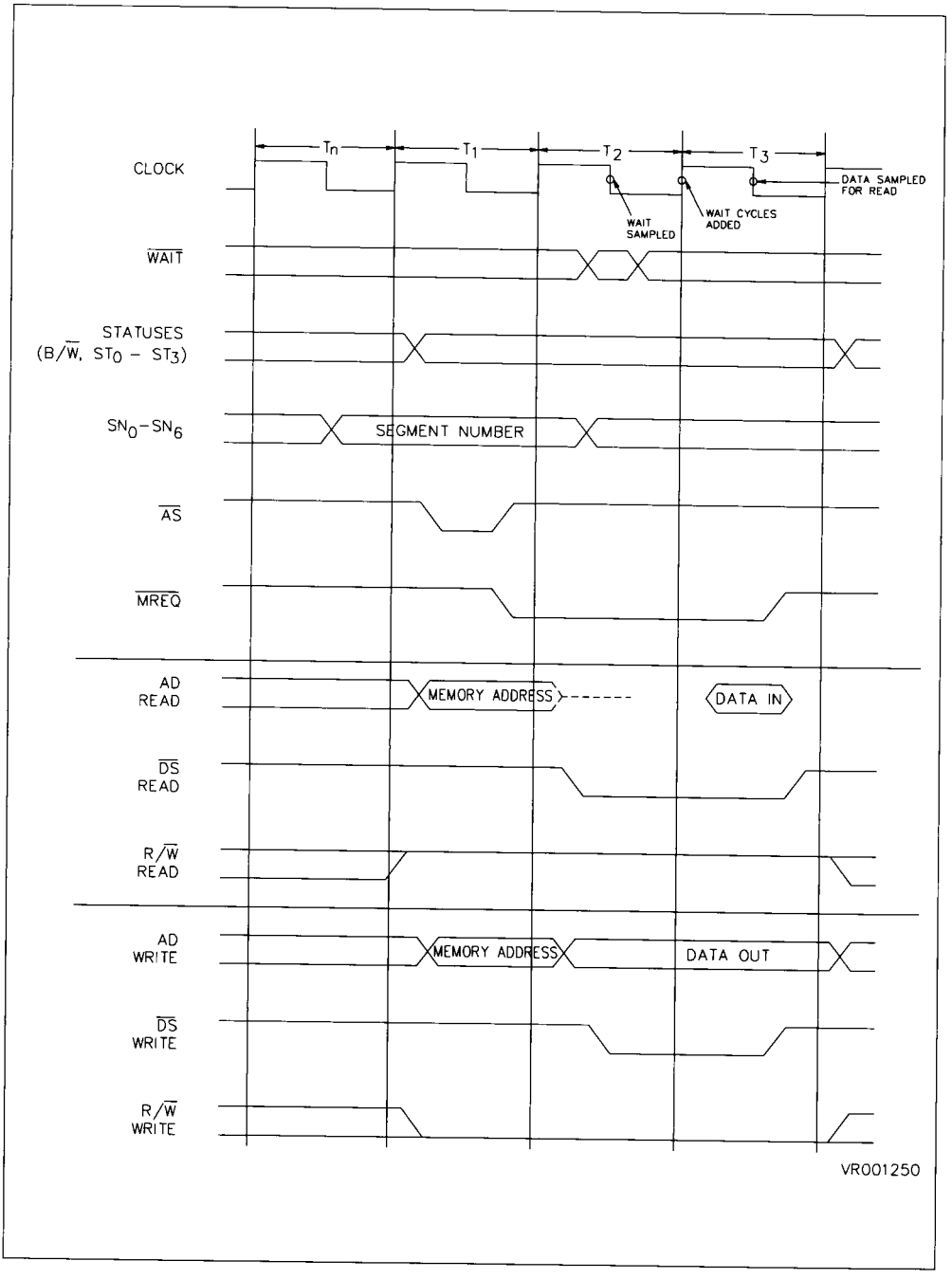
$\overline{WAIT}$. As shown in Figure 2-5, $\overline{WAIT}$ is sampled on a falling clock edge one cycle before data is sampled by the CPU (Read) or $\overline{DS}$ rises (Read or Write). If $\overline{WAIT}$ is Low, another cycle is added to the transaction before data is sampled or $\overline{DS}$ 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 in order for the CPU to function correctly. This requires asynchronously generated $\overline{WAIT}$ signals to be synchronized before they are input the CPU.

Table 2-1. Status Code

Kind of Transaction	ST3 - ST0	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	

Figure 2-6. Memory Read and Write Transaction



Transactions (Continued)

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 and to fetch and store memory data. They are also generated to store old program status and fetch new program status during interrupt and trap handling and after reset. As shown in Figure 2-6, a memory transaction is three clock cycles long unless 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.
- 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.

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 2-7. 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). 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.

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.

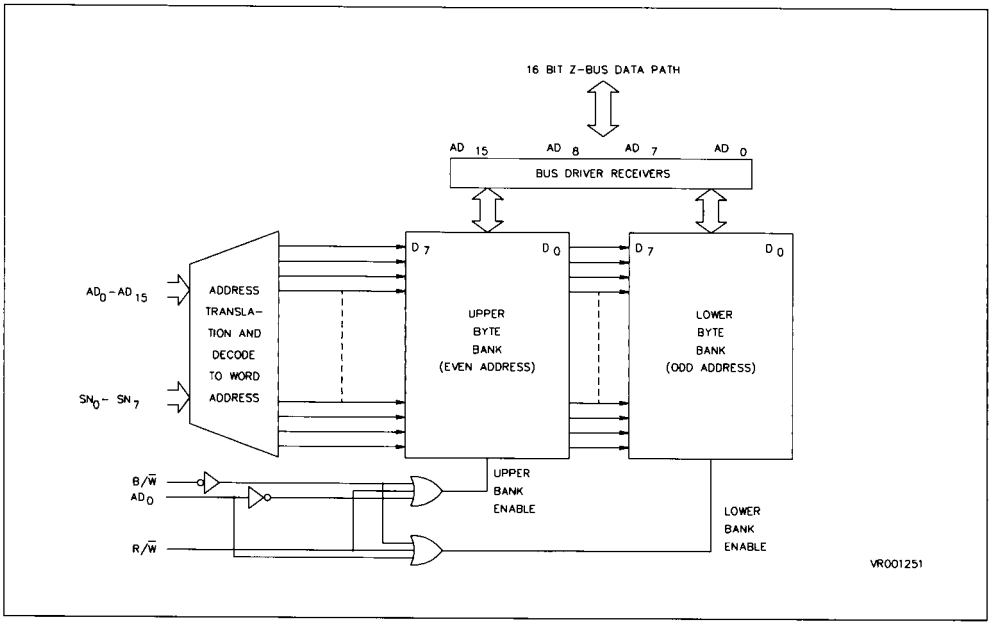
As shown in Figure 2-8, I/O transactions are four clock cycles long at minimum, and they may be lengthened by the addition of $\overline{WAIT}$ cycles. The extra clock cycles allow for slower peripheral operation.

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).

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.

Figure 2-7. Memory Organization

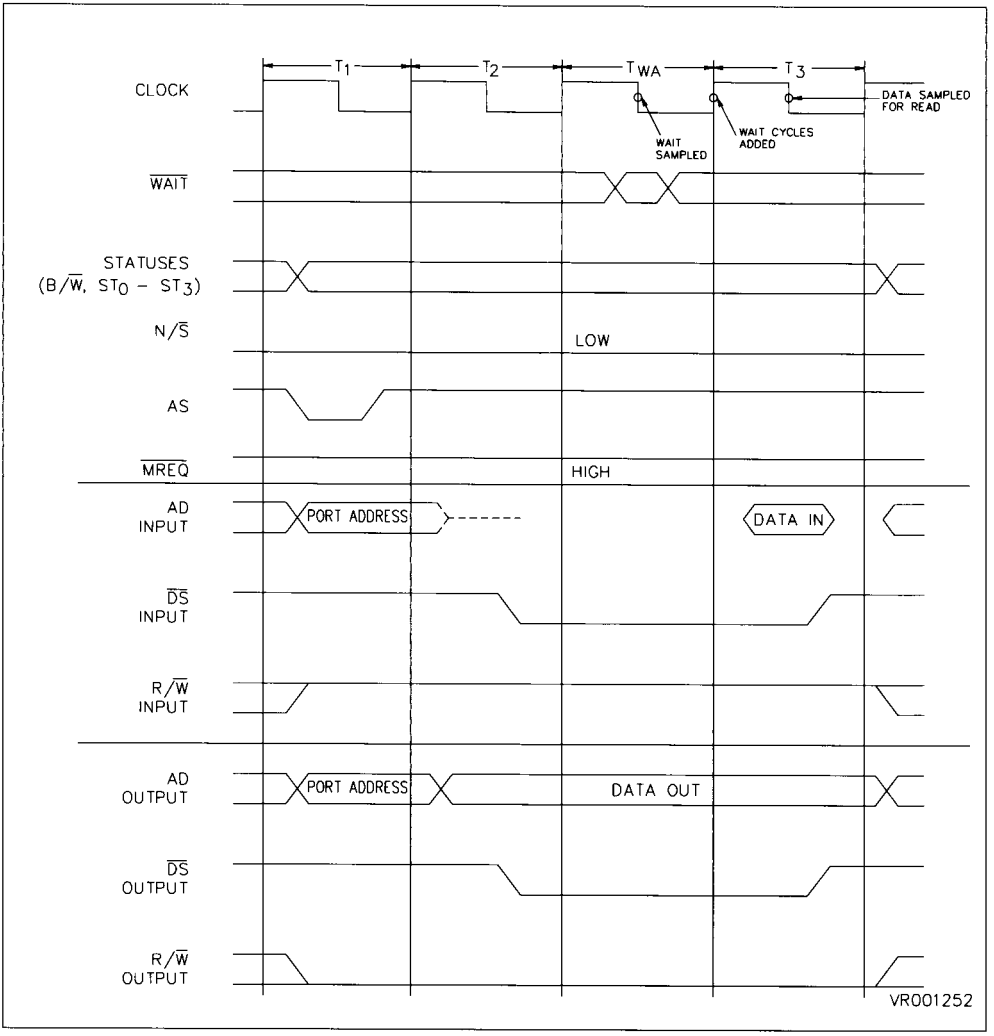


Transactions (Continued)

EPU transfer transactions have the same form as memory transactions (Figure 2-6) 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, 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 2-8. Input/Output Transaction



Transactions (Continued)

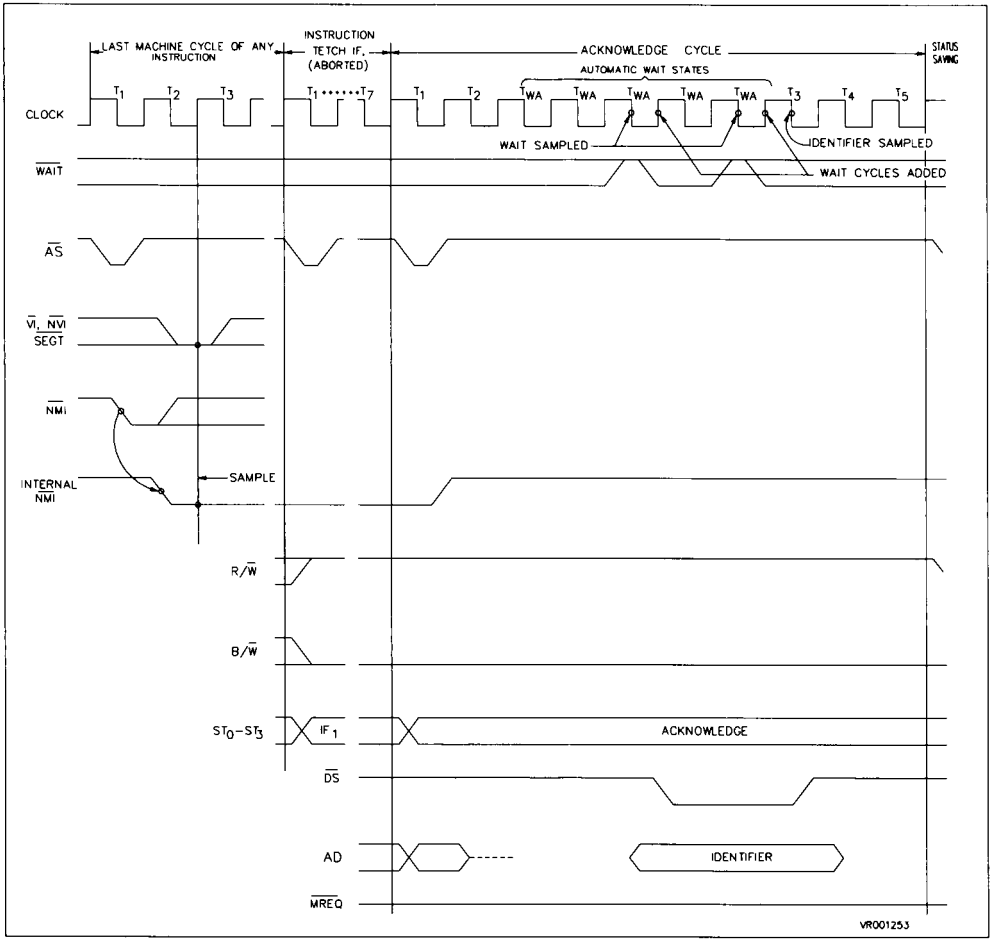
Interrupt/Trap Acknowledge Transactions. These transactions acknowledge an interrupt or trap and read a 16-bit identifier word from the device that generated the interrupt or trap. The transactions are generated automatically by the hardware when an interrupt or segment trap is detected.

These transactions are eight clock cycles long at a minimum (as shown in Figure 2-9), 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 tread. (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 identifier 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.

Figure 2-9. Interrupt and Segment Trap Request an Acknowledge Transition



As shown in Figure 2-9, 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.

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.

For internal operation transaction (shown in Figure 2-10), 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 2-11) is generated by the Z8000 CPU's refresh mechanism 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 AD0 - AD8 during the normal time for addresses. This transaction can be used to generate refreshes for dynamic RAMs. The value of N/S, R/W, and B/W is the same as for the immediately preceeding transaction.

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

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.

Figure 2-10. Internal Operation Timing

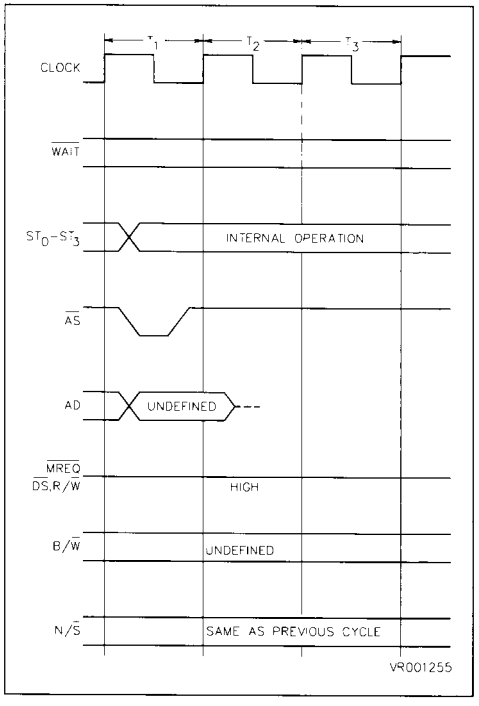
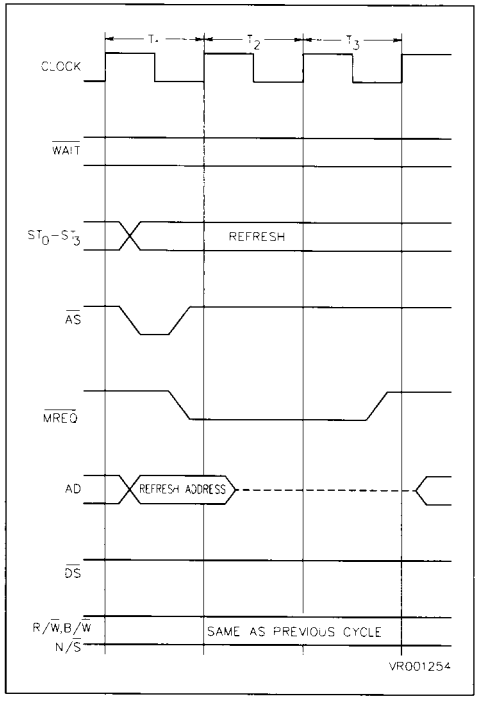


Figure 2-11. Memory Refresh Timing



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 tranfers 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 until the instruction completes execution.

Besides determining whether or not to participate in the exection 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.

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 for bus request, an acknowledge signal is sent 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.

Interrupt/Trap Request. The Z8000 CPU supports three interrupts and one external trap (segment trap) as shown in Figure 2-9. 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.

Requests (Continued)

- 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 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.

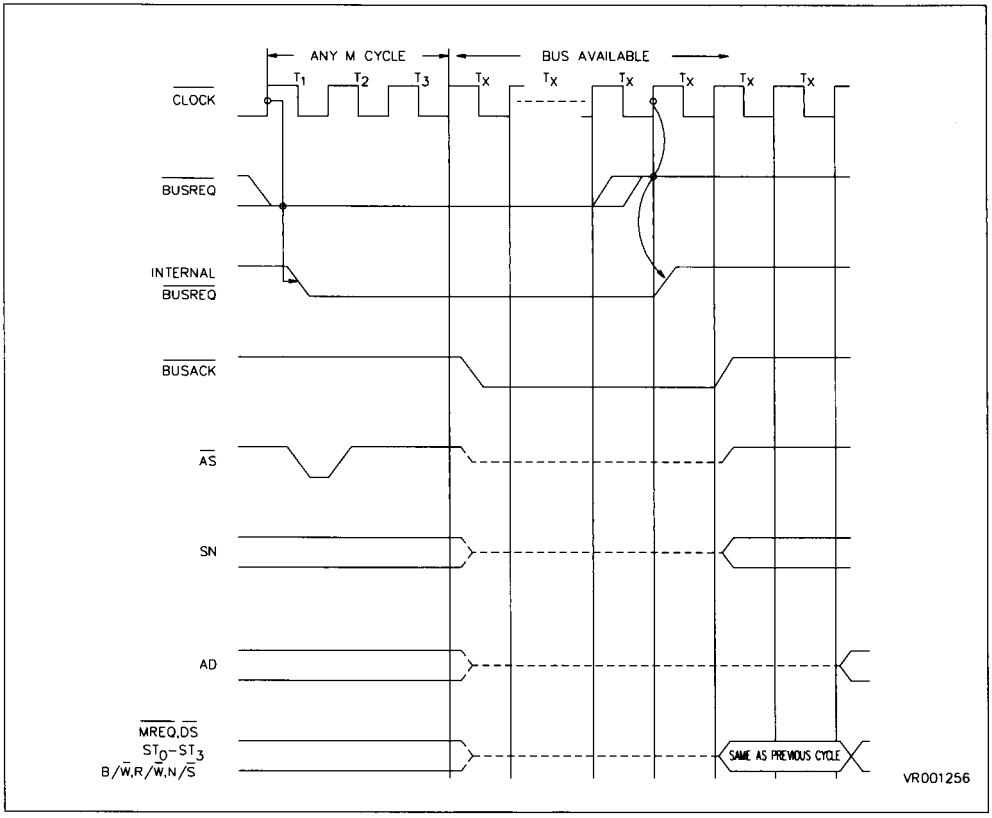
For more information about the system-level aspects of the interrupt structure, consult the Z-BUS Summary.

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 2-12). A bus request is initiated by pulling $\overline{\text{BUSREQ}}$ Low. Several bus requesters may be wired to the $\overline{\text{BUSREQ}}$ pin ; priorities are resolved externally to the CPU, usually by a priority daisy chain.

The asynchronous $\overline{\text{BUSREQ}}$ signal generates an internal $\overline{\text{BUSREQ}}$, which is synchronous. If the external $\overline{\text{BUSREQ}}$ is Low at the beginning of any machine cycle, the internal $\overline{\text{BUSREQ}}$ will cause the bus acknowledge line ($\overline{\text{BUSACK}}$) to be asserted after the current machine cycle is completed. The CPU then enters Bus-Disconnect state and gives up control of the bus. All CPU Output pins, except $\overline{\text{BUSREQ}}$ and $\overline{\text{MO}}$, are 3-stated.

The CPU regains control of the bus two clock cycles after $\overline{\text{BUSREQ}}$ rises. Any device desiring control of the bus must wait at least two cycles after $\overline{\text{BUSREQ}}$ has risen before pulling it down again.

Figure 2-12. Bus Request/Acknowledge Timing



Resource Request. The CPU generates resource requests by executing the Multi-Micro Request (MREQ) instruction. The CPU tests the availability of the shared resource by examining $\overline{MI}$. If $\overline{MI}$ is High, the resource is available, otherwise the CPU must try again later. The $\overline{MO}$ pin is used to make the resource request. $\overline{MO}$ is pulled Low, then, after a delay for arbitration of priority, $\overline{MI}$ 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, $\overline{MO}$ must be deactivated. But if successful, $\overline{MO}$ must be kept active until the CPU is ready to release the resource whereupon $\overline{MO}$ is deactivated by an MRES instruction.

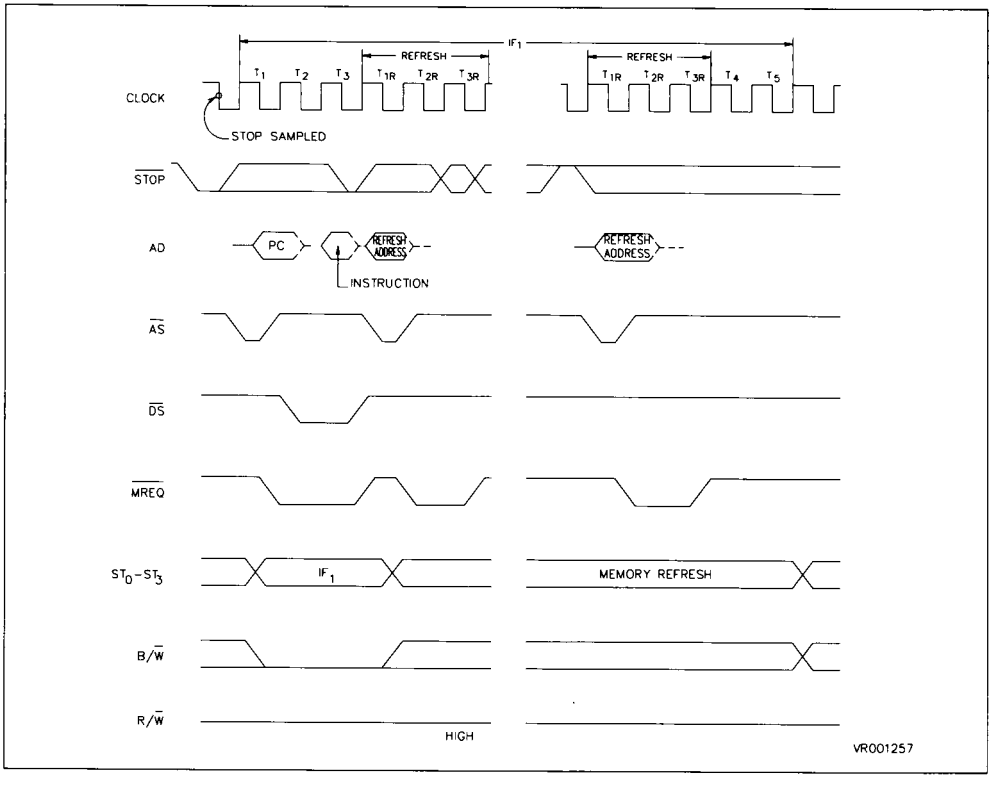
The Z-BUS Summary describes an arbitration scheme that is implemented with a resource request daisy chain.

Stop Request. As shown in Figure 2-13, the $\overline{STOP}$ pin is normally sampled on the falling clock edge immediately preceding an initial instruction fetch cycle. If $\overline{STOP}$ 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.

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 falling clock edge immediately preceding the second word of an instruction fetch -if the first word indicates an extended instuction. 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.

Figure 2-13. Stop Timing



Reset

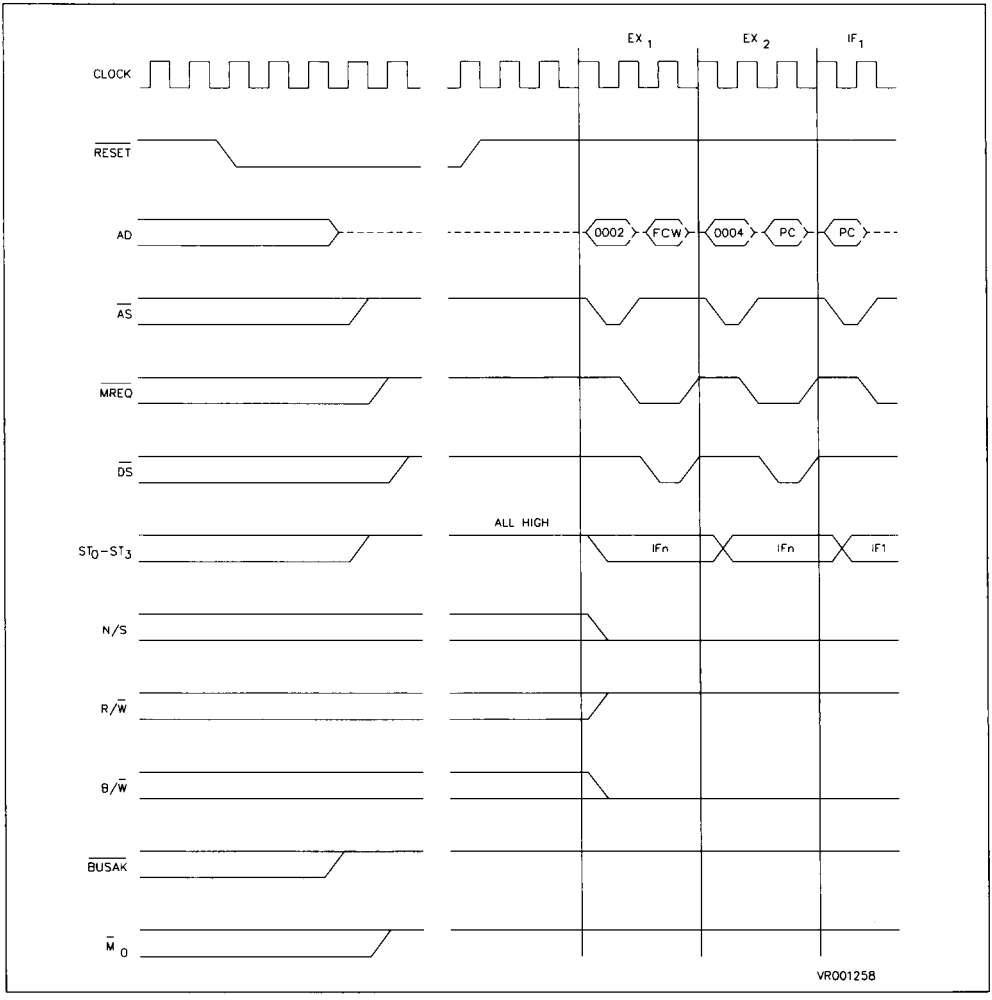
A hardware reset puts the Z8000 in a known state and initializes selected control registers of the CPU to system specifiable values (as A reset will begin at the end of any clock cycle, if the $\overline{RESET}$ line is Low.

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 2-14), 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.

Figure 2-14. Reset Timing



ARCHITECTURE

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 segment Z8001) are noted where appropriate.

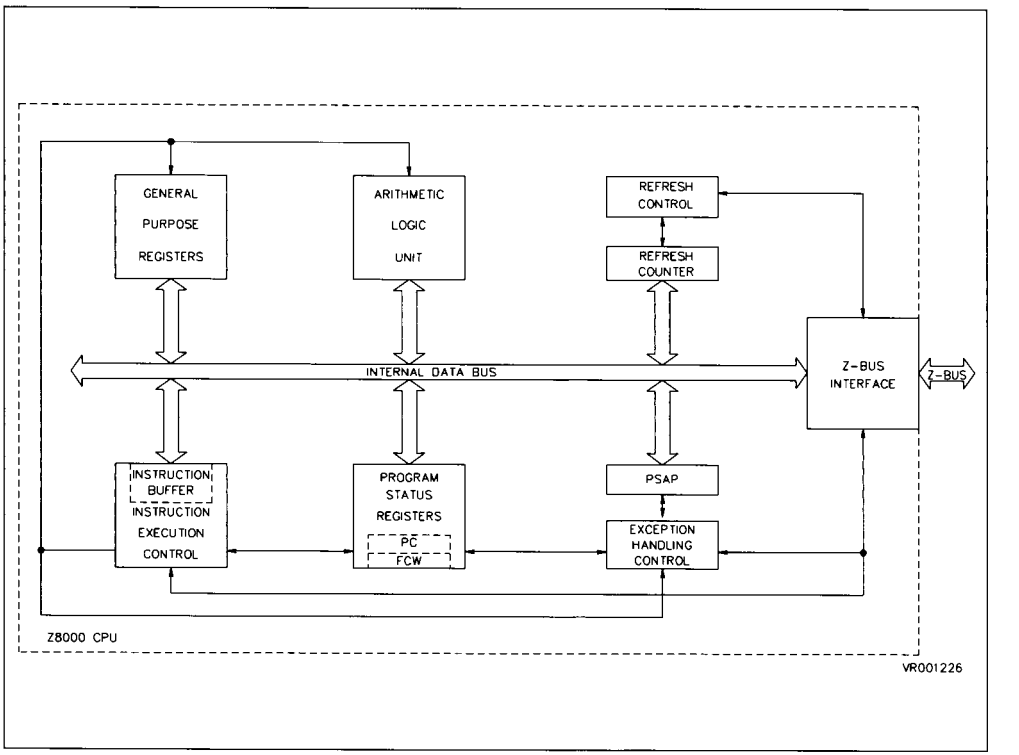
General Organization

Figure 3-1 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 retrieves 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 3-1. Z8000 CPU Functional Block Diagram

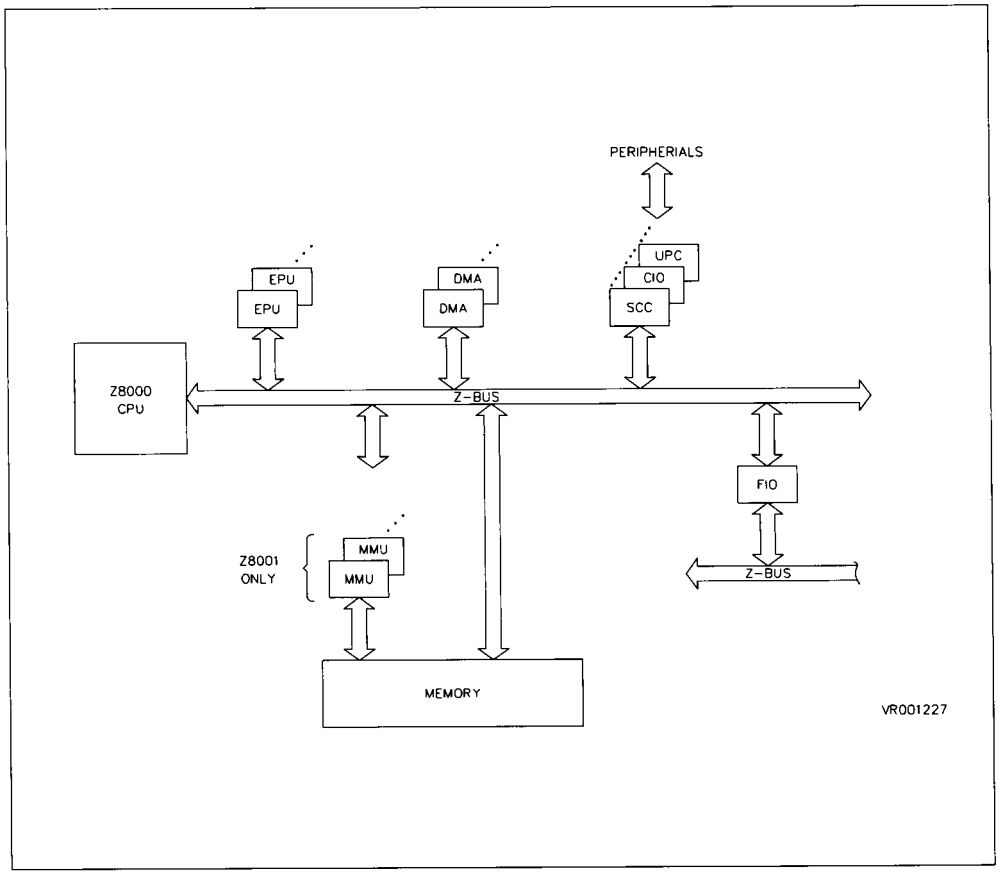


General Organization (Continued)

Figure 3-2 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 3-2. Z8000 System Configuration



Hardware Interface

Figure 3-3 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 PIN CONFIGURATION.

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.

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 management system.

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.

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 3-1 shows. The availabililty 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.

Figure 3-3. Z8000 Pin Functions

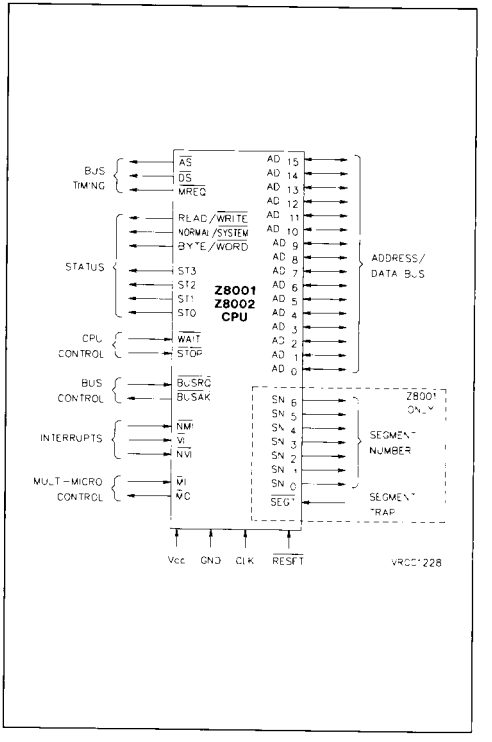


Table 3-1. Status Line Codes

ST_3 - ST_0	Definition
0000	Internal operation
0001	Memory refresh
0010	I/O reference
0011	Special I/O reference
0100	Segment trap acknowledge
0101	Non-maskable interrupt acknowledge
0110	Non-vectored interrupt acknowledge
0111	Vectored interrupt acknowledge
1000	Data memory request
1001	Stack memory request
1010	Data memory request (EPU)
1011	Stack memory request (EPU)
1100	Instruction space access
1101	Instruction fetch, first word
1110	EPA transfer
1111	Reserved

Hardware Interface (Continued)

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 retrieved. This signal is useful for single-step instruction execution during debugging operations and for enabling Extended Processing Units to halt the CPU temporarily.

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.

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 execution an interrupt service routine.

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.

Multi-Micro Control

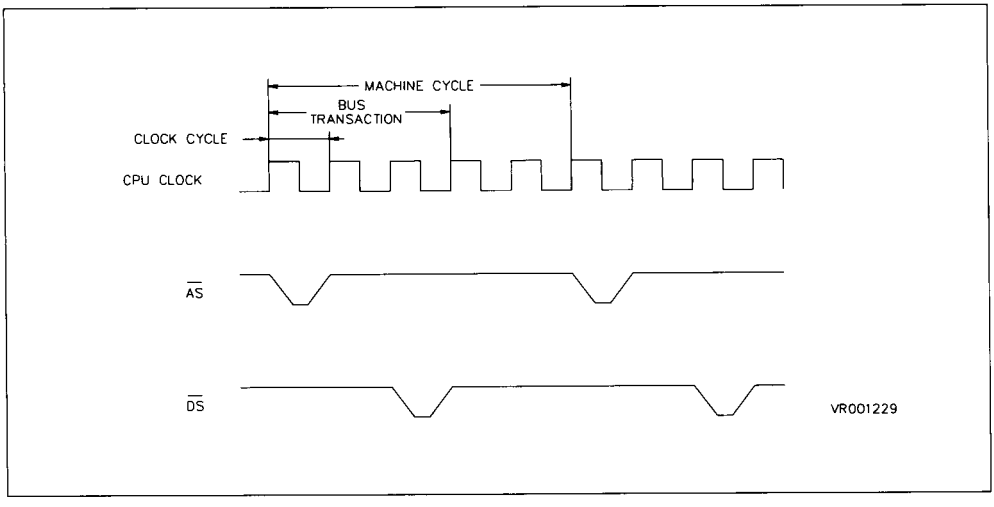
The Multi-Micro In ($\overline{MI}$) and Multi-Micro Out ($\overline{MO}$) lines are used in conjunction with instructions such as MSET and MREQ to coordinate multiple-CPU systems. They allow exclusive use by one CPU of a shared resource in a multiple-CPU system.

System Inputs. The four inputs shown at the bottom of Figure 3-3 include +5V power, ground, a single-phase clock signal and a CPU reset. The reset function is described in Chapter EXCEPTIONS.

Timing

Figure 3-4 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.

Figure 3-4. Basic Timing Periods



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 ADDRESS SPACES.

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 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 space 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 over-flowing 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 ADDRESS SPACES contains an extensive discussion of segmentation and the Z8001.

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).

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 ADDRESSING MODES).

Figure 3-5 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 3-5 illustrates, the CPU has two hardware stack pointers, one dedicated to each of the two basic operating modes, system and 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 call in system mode. The normal stack pointer is used for subroutine call 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 accessed as a special control register.

Figure 3-5a. Z8001 General-Purpose Registers (Register Address Space)

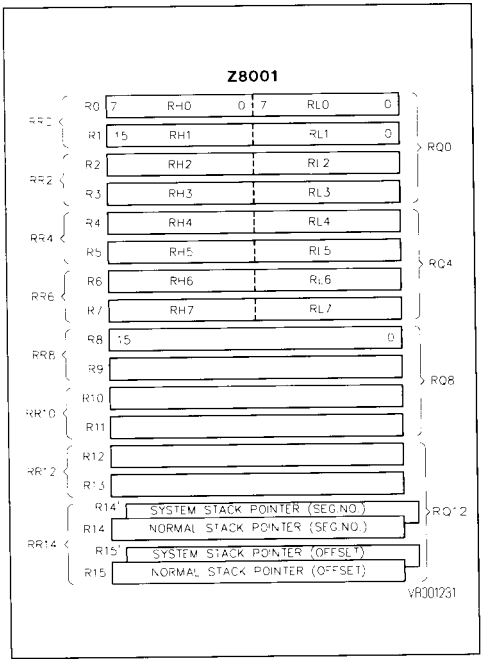
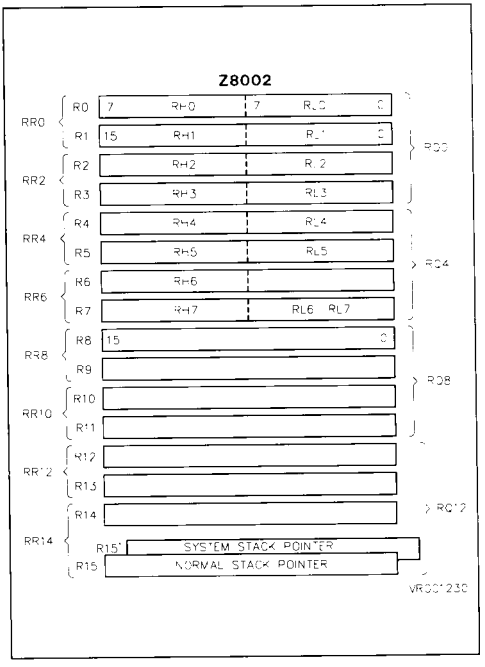


Figure 3-5b. Z8002 General-Purpose Registers (Register Address Space)



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 3-6. Each register can be manipulated by software executing in system mode, and some are modified automatically by certain operations.

Programs 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.

The Z8001,2 CPU Programming Manual gives more information about these flages.

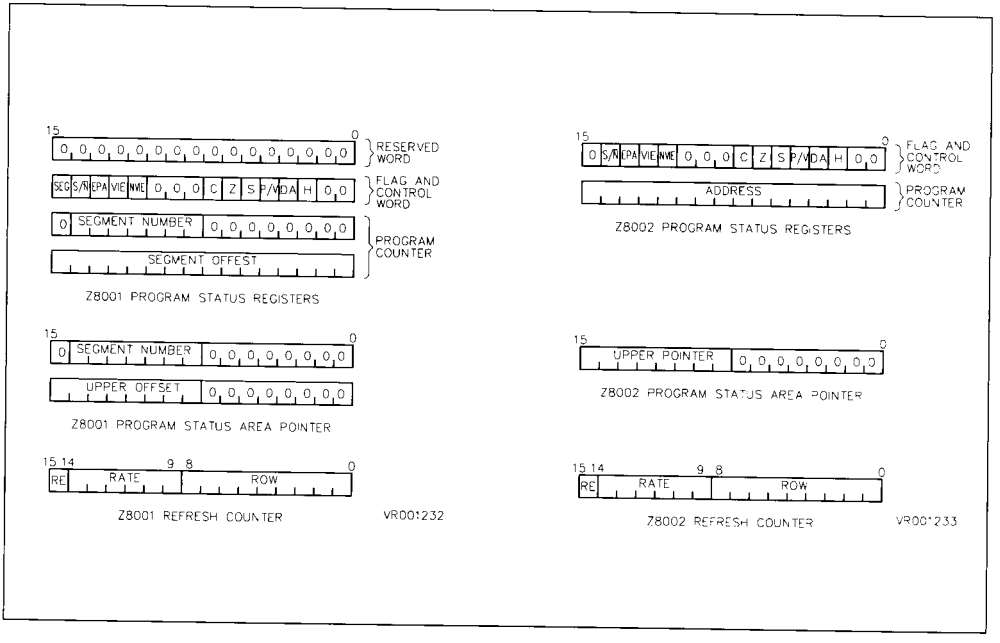
The control bits, which occupy the high-order byte of the FCW, are used to enable various interrupts or to control CPU operating modes. The control bits are :

- **Non-Vectored Interrupt Enable (NVIE), Vectored Interrupt Enable (VIE).** These bits determined whether or not the CPU will accept non-vectored or vectored interrupts (see page 48).
- **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 page 43). The CPU output status line (N/$\overline{S}$ pin) is the complement of this bit.
- **Extended Processor Architecture (EPA) Mode.** When this bit it set to one, it indicates that the system contains Extended Processing Units, and hence extended instructions encountered in the CPU instruction stream are executed (see page 45). 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 page 43).

Program Status Area Pointer (PSAP). The Program Status Area Pointer points to an array of program 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 3-6, 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 EXCPTIONS for more details about the Program Status Area and its layout.

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 3-6). Refer to Chapter REFRESH for details of the refresh mechanism.

Figure 3-6. CPU Special Registers



Instruction Execution

In the normal course of events, the Z8000 CPU will spend most of its time retrieving 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 REFRESH.
- **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 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 instruction (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 (see Figure 5-2). While handling interrupts, the CPU is always in system mode and, for the Z8001, in segmented mode.

Instructions

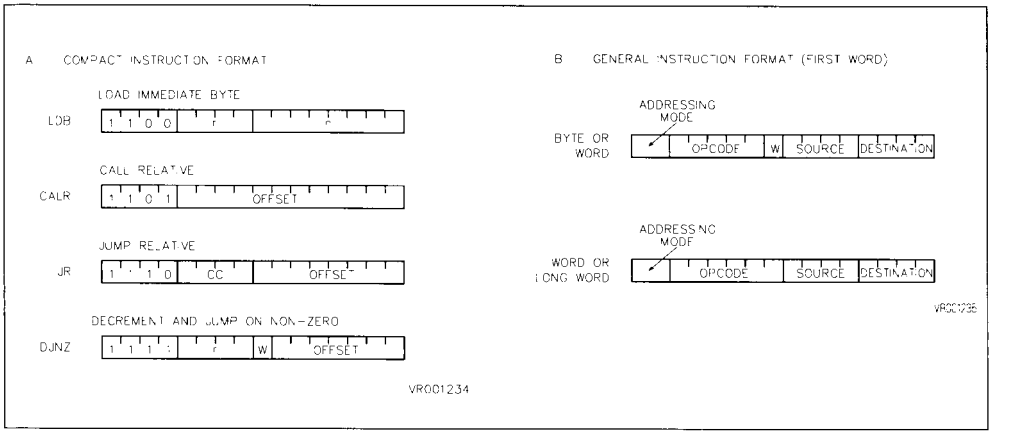
The Z8000 instruction set contains over 400 different instructions which are formed by combining the 110 distinct instruction types (opcodes) with the various data types and addressing modes. The complete set is divided into the following groups :

- **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.

The Z8001,2 CPU Programming Manual contains full details of the instruction set.

Instruction Formats. Formats of the instructions are shown in Figure 3-7. 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 3-7. Instruction Formats



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.

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 ADDRESSING MODES defines and illustrates the eight addressing modes.

Extended Processing Architecture

The extended Processing Architecture (EPA) provides an extremely flexible and modular approach to expanding both the hardware and software capabilities of the Z8000 CPU. Features of the EPA include :

- Specialized instructions for external processors or software traps may be added to CPU instruction set.
- Increases throughput of the system by using up to four specialized external processors in parallel with the CPU.
- Permits modular design of Z8000-based systems.
- Provides easy management of multiple microprocessor configurations via "single instruction stream" communication.
- Simple interconnection between extended processing units and Z8000 CPU requires no additional external supporting logic.
- Supports debugging of suspect hardware against proven software.

Specific benefits include :

- EPUs can be added as the system grows and as EPUs with specialized functions are developed.
- Control of EPUs is accomplished via a "single instruction stream" in the Z8000 CPU, eliminating many significant system software and bus contention management obstacles that occur in other multiprocessor (e.g., master-slave) organization schemes.

The processing power of the Z8000 can be boosted beyond its intrinsic capability by Extended Processing Architecture. Simply stated, EPA allows the Z8000 CPU to accomodate up to four Extended Processing Units (EPUs), which perform specialized functions in parallel with the CPU's main instruction execution stream.

The use of extended processors to boost the main CPU's performance capability has been proven with large mainframe computers and minicomputers. In these systems, specialized functions such as array processing, special input/output processing, and data communications processing are typically assigned to extended processor hardware. These extended processors are complex computers in their own right.

The Extended Processing Architecture combines the best concepts of these proven performance boosters with the latest in high-density MOS integrated-circuit design. The result is an elegant expansion of design capability - a powerful microprocessor architecture capable of connecting single-chip EPUs that permits very effective parallel processing and makes for a smoothly integrated instruction stream from the Z8000 programmer's point of view. A typical addition to the current Z8000 instruction set is Floating Points Instructions.

The Extended Processing Units connect directly to the Z-BUS and continuously monitor the CPU instruction stream.

When an extended instruction is detected, the appropriate EPU responds, obtaining or placing data or status information on the Z-BUS using the Z8000-generated control signals and performing its function as directed.

The Z8000 CPU is responsible for instructing the EPU and delivering operands and data to it. The EPU recognizes instructions intended for it and executes them, using data supplied with the instruction and/or data within its internal registers. There are four classes or EPU instructions :

- Data transfers between main memory and EPU registers.
- Data transfers between CPU registers and EPU registers.
- EPU internal operations.
- Status transfers between the EPUs and the Z8000 CPU Flag and Control Word register (FCW).

Four Z8000 addressing modes may be utilized with transfers between EPU registers and the CPU and main memory :

- Register,
- Indirect Register,
- Direct Address,
- Indexed.

In addition to the hardware-implemented capabilities of the Extended Processing Architecture, there is an extended instruction trap mechanism to permit software simulation of EPU functions. A control bit in the Z8000 FCW register indicates whether actual EPUs are present or not. If not, when an extended instruction is detected, the Z8000 traps on the instruction, so that a software "trap handler" can emulate the desired EPU function - a very useful development tool. The EPA software trap routine supports the debugging of suspect hardware against proven software. This feature will increase in significance as designers become familiar with the EPA capability of the Z8000 CPU.

Figure 3-8. Typical Extended Processor Configuration

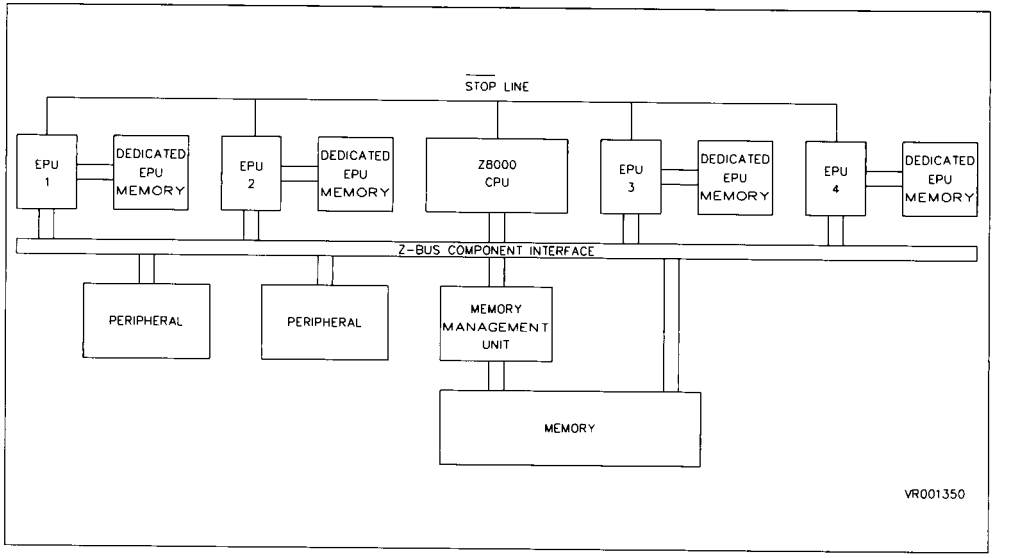
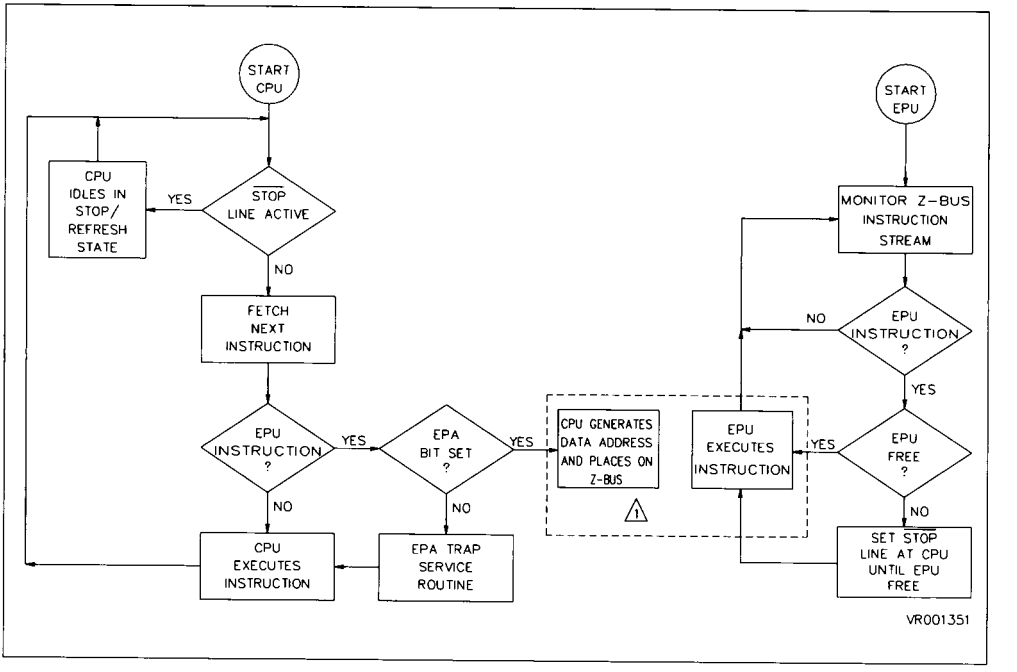


Figure 3-9. EPA and Z8000 CPU Instruction Execution



Extended Processing Architecture (Continued)

This software trap mechanism facilitates the design of systems for later addition of EPUs : initially, the extended function is executed as a trap subroutine ; when the EPU is finally attached, the trap subroutine is eliminated and the EPA control bit is set. Application software is unaware of the change.

Extended Processing Architecture also offers protection against extended instruction overlapping. Each EPU connects to the Z8000 CPU via the STOP line so that if an EPU is requested to perform a second extended instruction function before it has completed the previous one, it can put the CPU into the Stop/Refresh state until execution of the previous extended instruction is complete.

EPA and CPU instruction execution are shown in Figure 3-9. The CPU begins operation by fetching an instruction and determining whether it is a CPU or an EPU command. The EPU meanwhile monitors the Z-BUS for its own instructions. If the CPU encounters an EPU command, it checks to see whether an EPU is present ; if not, the EPU may be simulated by an EPU instruction trap software routine ; if an EPU is present, the necessary data and/or address is placed on the Z-BUS. If the EPU is free when the instruction and data for it appear, the extended instruction is executed. If the EPU is still processing a previous instruction, it activates the CPU's $\overline{\text{STOP}}$ line to lock the CPU off the Z-BUS until execution is complete. After the instruction is finished, the EPU deactivates the $\overline{\text{STOP}}$ line and CPU transactions continue.

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 retrieve transaction is usually performed before the interrupt or acknowledge sequence begins, but the Program Counter is not affected by the spurious retrieval.

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 retrieved from a reserved area of memory to reinitialize the Flag and Control Word (FCW) and the Program Counter (PC).

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.

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.

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 EXCEPTIONS.

ADDRESS SPACES

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 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 retrievals,* decribed in Chapter CPU OPERATIONS.
- *Operand retrievals and stores,* described in Chapter ADDRESSING MODES.
- *Exception processing,* described in Chapter EXCEPTIONS.
- *Refreshes,* described in Chapter REFRESH.

Timing information concerning addresses is described in Chapter PIN CONFIGURATION.

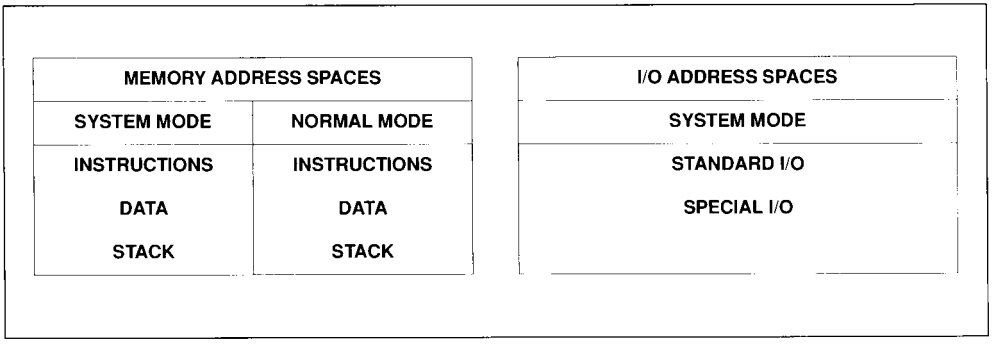
Types of Address Spaces

Within the two general types of address spaces (memory and I/O), it is possible to distinguish several subcategories. Figure 4-1 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 sytem 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 4-1. Address Spaces on the Z8001 and Z8002



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.

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 words.*

Addressable Data Elements. The nature of the data element being addressed depends on the instruction being executed. As Chapter EXCEPTIONS explains in detail, different assembler mnemonics are used for addressing bytes, words, and long words. Moreover, only certain instructions can address bits.

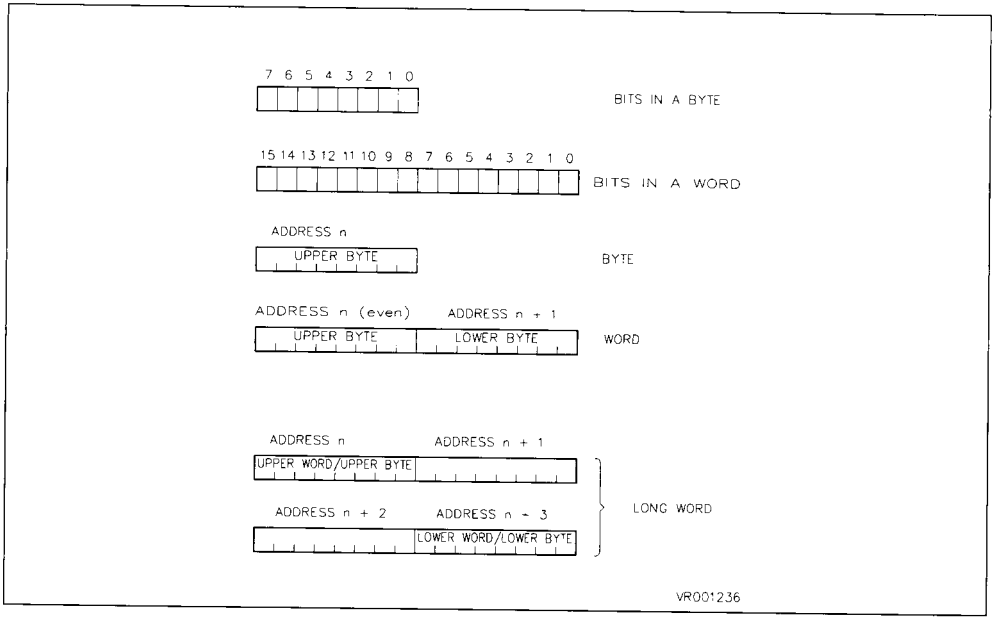
A bit can be addressed by specifying a byte or word address and the number of the bit within the byte (0-7) or word (0-15). Bits are numbered right-to-left, from the least to the most significant. This is consistent with the convention that bit n corresponds to position 2^n in the conventional representation of binary numbers (see Figure 4-2).

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 4-2). 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 EXCEPTIONS. Except for these reserved locations, there are not memory addresses specifically designated for a particular purpose.

Figure 4-2. Addressable Data Elements

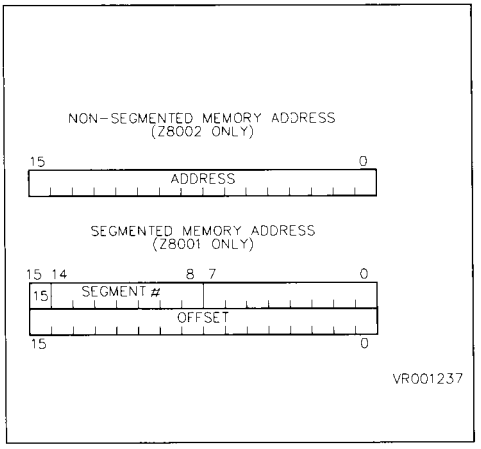


Segmented and Nonsegmented Addresses. The two versions of the Z8000 CPU generate two kinds of addresses with different lengths. The Z8002 generates a 16-bit address specifying one of 64K bytes. The Z8001 generates a 23-bit *segmented* address. A segmented address consists of a 7-bit *segment number*, 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 this Chapter.

Figure 4-3 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-lenght 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 CPU OPERATIONS), it always generates segmented addresses. The segment number is supplied by the program counter segment number.

Figure 4-3. Segmented and Nonsegmented Address Formats



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 4-4). 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 :

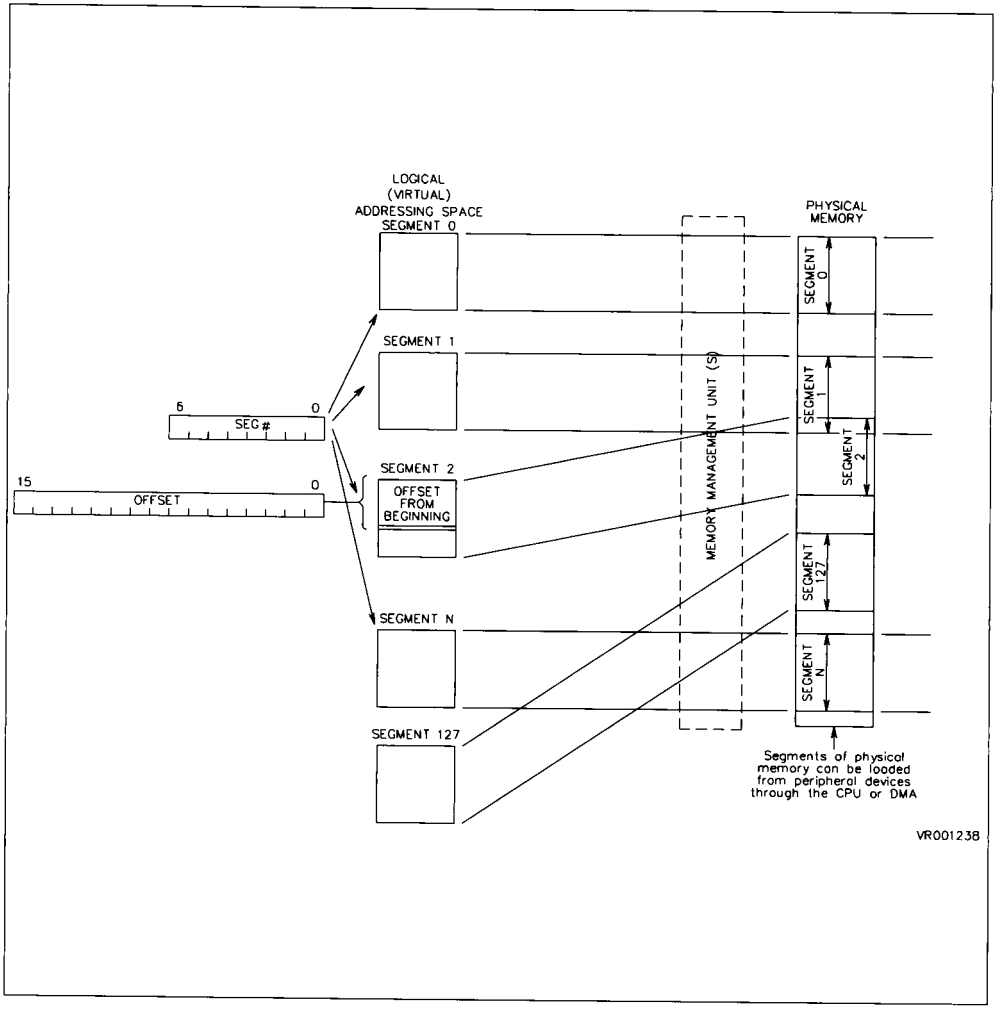
- Provision for flexible and efficient allocation of physical memory resources during the execution of programs.
- Hardware stack overflow protection.
- Support for multiple, independently executing programs that can share access to common code and data.
- Protection from unauthorized or unintentional 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.

Figure 4-4. Segmented Address Translation



CPU OPERATION

Introduction

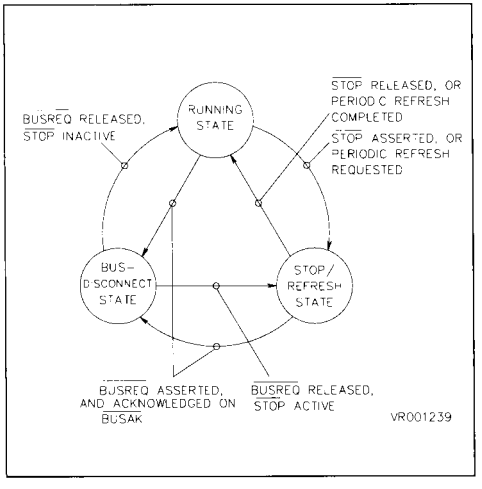
This chapter gives a fundamental description of the operating states of the Z8000 CPU and the process of instruction execution.

Operating States

The Z8000 CPU has three operationg 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 $\overline{STOP}$ line is asserted or the refresh counter indicates that a periodic refresh should be done. In this state, memory refresh transactions are generated continually. Bus-Disconnect state is entered when the CPU acknowledges a bus request and gives up control of the system bus. Figure 5-1 shows the three states and the conditions that cause state transitions.

Running State. While the CPU is in Running state, it is either executing instructions or handling exceptions. The CPU is normally in Running state, but will leave this state in response to one of three conditions :

Figure 5-1. Operating States and Transitions



- 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.

Stop/Refresh State. While the CPU is in Stop/Refresh state, it generates a continuous stream of refresh cycles 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.

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.

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 $\overline{BUSREQ}$ and acknowledged on $\overline{BUSACK}$. 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 Bust-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.

Effect of Reset. Activation of the CPU's $\overline{RESET}$ 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 $\overline{RESET}$ is deactivated. When this occurs, the program enters one of the three operating states described above, depending on the state of $\overline{BUSREQ}$ and $\overline{STOP}$ inputs.

Instruction Execution

While the CPU is in Running state and executing instructions, it is controlled by the Program Status registers (Figure 5-2). The Program Counter gives the address from which instructions are fetched, the flags control branching, and the control bits determine the mode in which the CPU operates and the interrupts that are masked.

Instruction execution consists of the repeated application of two steps :

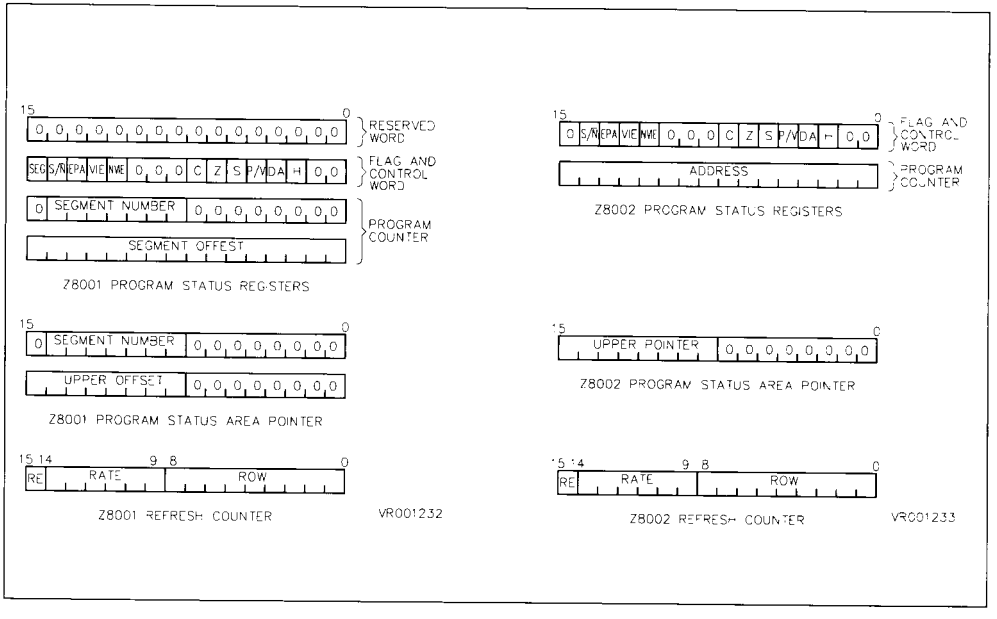
- Fetch one or more words comprising a single instruction from the program memory address space at the address specified by the Program Counter (PC).
- Perform the operation specified by the instruction and update the Program Counter and flags in the Program Status registers.

The operation performed by an instruction and the way the flags are updated depends on the particular instruction being executed 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 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 5-3. 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 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, 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.

Figure 5-2. Program Status Registers



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.

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 :

- 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 and 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.

Normal and System Modes. The operation mode of the CPU (system mode or normal mode) determines which isntructions 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 5-4, 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

Figure 5-3. Instruction Look-Ahead

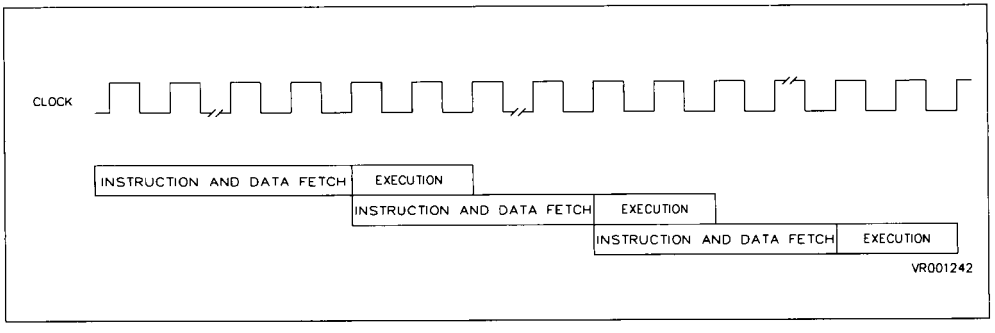


Table 5-1. Registers Accessed by References to R14 and R15

Register Referenced by Instruction	System Mode		Normal Mode	
	Segmented	Nonsegmented	Segmented	Nonsegmented
R14	System R14	Normal R14	Normal R14	Normal R14
R15	System R15	System R15	Normal R15	Normal R15
RR14	System R14	Normal R14	Normal R14	Normal R14
	System R15	System R15	Normal R15	Normal R15

Note : Z8002 always runs in nonsegmented mode.

Instruction Execution (Continued)

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 5-1.

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.

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.

Extended Instructions

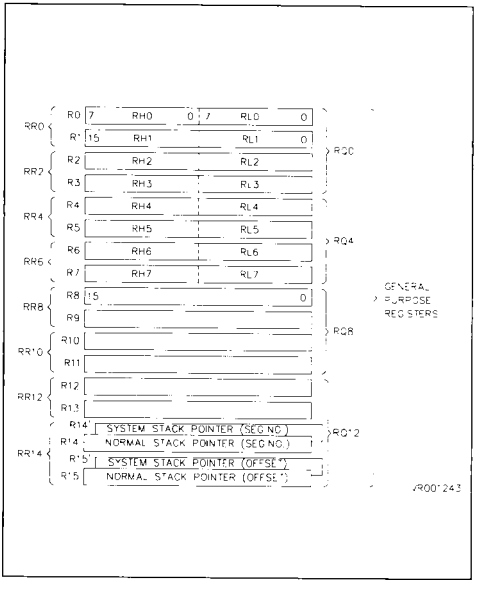
The Z8000 CPU supports seven types of extended instructions, which can be executed cooperatively by the CPU and an external Extended Processing Unit. The execution of these instructions is controlled by the EPA control bit in the FCW.

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 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.

Figure 5-4. General-Purpose Registers



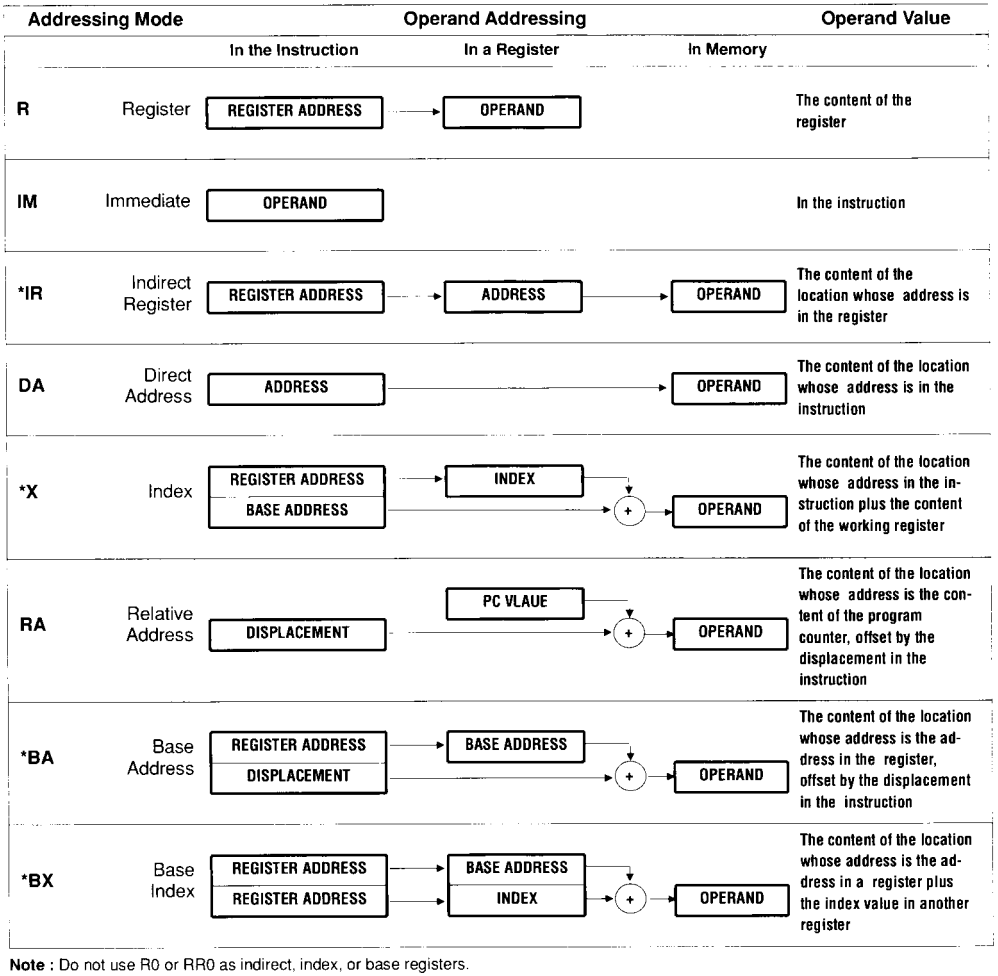
ADDRESSING MODES

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 6-1 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 6-1, the term "operand" refers to the data to be operated upon.

Figure 6-1. Addressing Modes



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

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 :

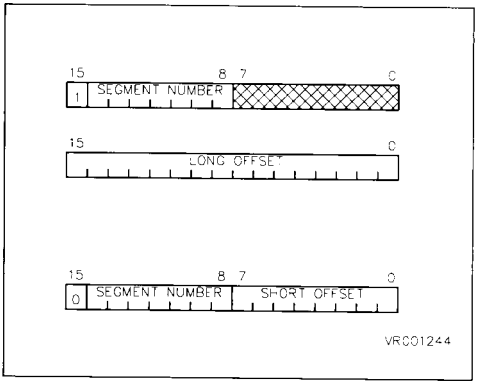
- Register R15 (or the double register RR14 in the Z8001) is used as a stack pointer for subroutine calls and returns.
- The byte register RH1 is used in the translate bulleted item instructions (TRDB, TRDRB, TRIB, TRIRB) and the translate and test instructions (TRTDB, TRTDRB, TRTIB, TRTIRB).
- Register 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 6-2 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 6-2. Segmented Memory Address Within Instruction



Note : Shaded area is reserved.

Addressing Mode Descriptions

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

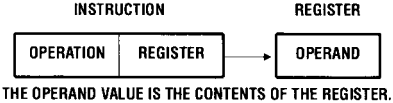
- Explains how the operand address is calculated,
- Indicates which address space (Register, I/O, Special I/O, Data Memory, Stack Memory, or Program Memory) the operand is located in,
- Shows the assembly language format used to specify the addressing mode, and
- Works through an example.

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.

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.

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!
```

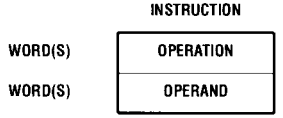
Before Execution

R2	A6B8
R3	9A20

After Execution

R2	9A20
R3	9A20

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 :

data

Example of IM mode :

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

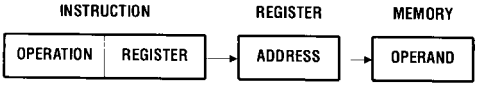
Before Execution

R2	6789

After Execution

R2	5589

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 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 way 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 :

@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

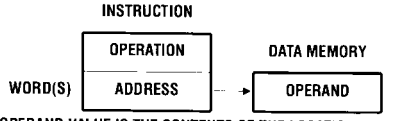
Memory

•	
170A	A023
170C	0B0E
170E	10D0
•	

After Execution

R2	0B0E
R3	0005
R4	2000
R5	170C

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 :

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

After Execution

R2 0689

Memory

•	
5E22	0106
5E24	0304
•	

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 :

address (Rn)

Example of X mode :

```
LD R4,%231A(R3)     !load into R4 the con-!
                    !tnents 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
	•

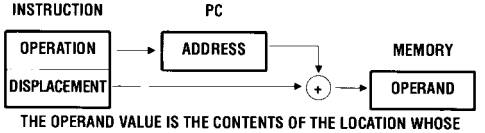
Address Calculation

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

After Execution

R3	01FE
R4	3D0E

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 instructions 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.

Assembler language format :

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	
0206	E801	
0208	FFFE	
	•	

Address Calculation

```
  0206
+    2
  0208
```

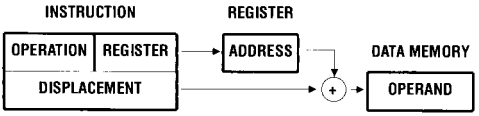
After Execution

R2	FFFE
PC	0206

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 :

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

```
  20A1A
+   18
  20C2
```

After Execution

RR2	R2	0A00
	R3	1500
	R4	3100
	R5	20AA

Memory

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

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 :

Rn (Rm)

Example of B mode :

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

Before Execution

R2	1F3A
R3	FFFE
R4	0300
R5	1502

Data Memory

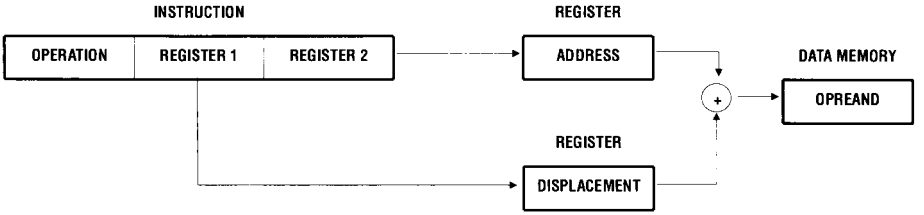
	•
14FE	0101
1500	B0DE
1502	F732
	•

Address Calculation

```
  1502
+ FFFE
  1500
```

After Execution

R2	B015
R3	FFFE
R4	0300
R5	1502

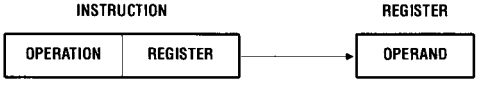


THE OPERAND VALUE IS THE CONTENTS OF THE LOCATION WHOSE ADDRESS IS THE CONTENTS OF THE ONE REGISTER OFFSET BY THE DISPLACEMENT IN THE SECOND REGISTER.

Descriptions and Examples (Segmented Z8001)

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

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 format :

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!

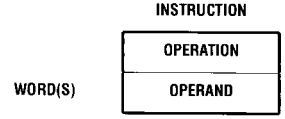
Before Execution

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

After Execution

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

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.

Because an immediate operands 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 :

#data

Example of IM mode :

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

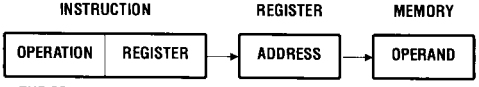
Before Execution

R2	6789

After Execution

R2	5589

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 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 format :

@Rn	Contains I/O or Special I/O address.
@RRn	Contains memory address.

Example of memory access using IR mode :

LD R2, @R4 !load into R2 with 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
•	

After Execution

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

Note : Segment Number 20.

Example of I/O using IR mode :

OUTB @R1,RL0

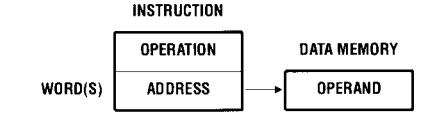
Before Execution

R0	0A23
R1	0011

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

SGS-THOMSON MICROELECTRONICS

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.)

Assembler language format :

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

After Execution

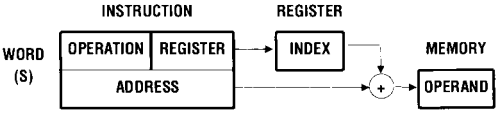
R2 0689

Memory

		•
<<15>>	0022	0206
	0024	0304
		•

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,                !load into R4 the con-!
<<15>>%231A(R3)       !tents of the memory!
                      ! location whose!
                      !address is segment 5,!
                      !displacement 231A + !
                      !the value in R3!
```

Before Execution

R3 01FE

R4 203A

Address Calculation

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

After Execution

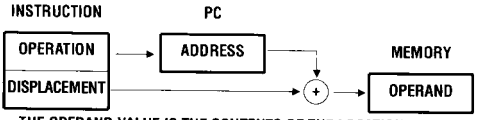
R3 01FE

R4 3D0E

Memory

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

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

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 :

address

Example of RA mode :

```
LDR R2,$+6        !load into R2 the!
                  !contents 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

R2	A0F0

After Execution

PC	0D00
R4	0202

Memory

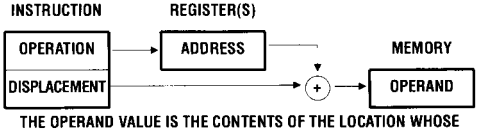
		•
<<13>>	0202	3102
	0204	0002
	0206	E801
	0208	FFFE
		•

Instruction

Address Calculation

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

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 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 :

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	A0F0
	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
```

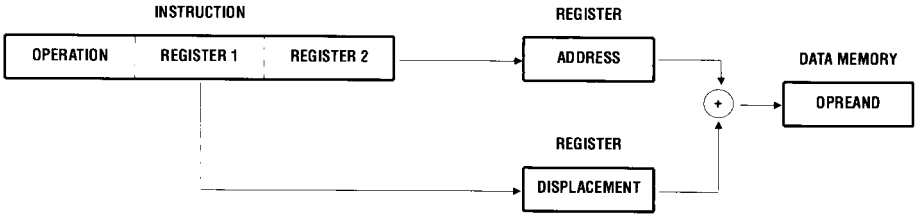
After Execution

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

Data Memory

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

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 register. 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

<<31>> 14FE	0101
1500	B0DE
1502	F732

EXCEPTIONS

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.

Interrupts

Three kinds of interrupts are activated by three different pins on the Z8000 CPU.

Non-Maskable Interrupt (NMI). This type of interrupt cannot be disabled (masked) by software. It is typically reserved for highest-priority external events that require immediate attention.

Vectored Interrupt (VI). One result of any interrupt or trap is that a 16-bit identifier word is pushed onto the system stack.

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.

Nonvectored Interrupts (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.

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.

Extended Instruction Trap. This trap occurs when the CPU encounters an extended instruction 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.

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.

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.

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.

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.

Reset (Continued)

Within five clock cycles of the $\overline{RESET}$ 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. $\overline{RESET}$ must be held Low five clock cycles to properly reset the CPU.

Three clock cycles after $\overline{RESET}$ 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.

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 occurence 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 request are not retained by the CPU, the request signal must be asserted until the CPU acknowledges the request.

Interrupt And Trap Handling

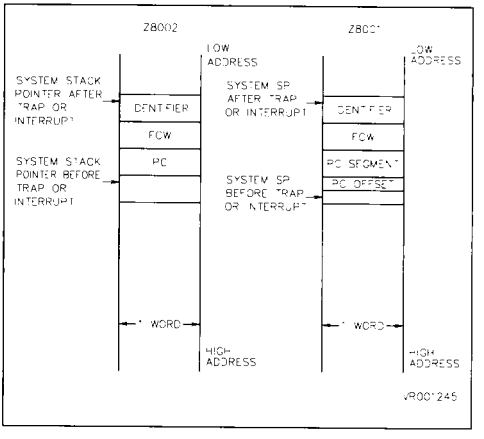
The CPU response to a trap or interrupt request consists of five steps : acknowledging the external request (for interrupts and segment traps), saving the old program status information, loading a new program status, executing the service routine, and returning to the interrupted taks.

Acknowledge Cycle. An external acknowledge cycle is required only for externally generated request.

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 unti it begins to execute the exception service routine, at which time its mode is dictated by the FCW.

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



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 (1, 2)
All Interrupts	Next Instruction (2)

Notes :

1. Assumes successful completion of instruction fetch.
2. If executing an interruptable instruction (e.g. LDIR) and the instruction is the current instruction.

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.

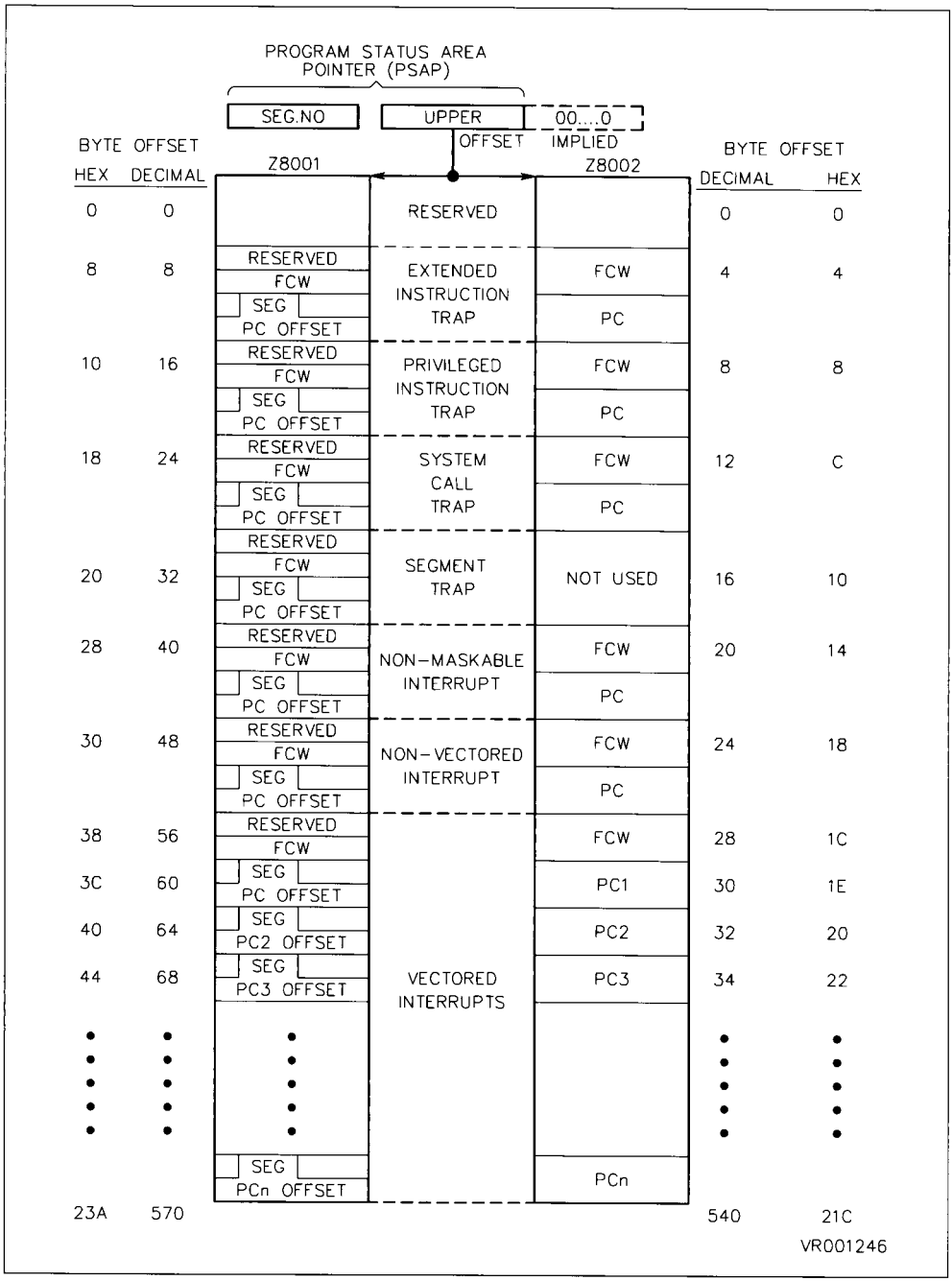
The program Status Area is addressed by a special control register, the Program Status Area Pointer, or PSAP. This pointer is one word for the nonsegmented Z8002 and two words for segmented Z8001. As shown in Figure 7-2, the pointer contains a segment number (if applicable) and the high-order byte of 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).

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.

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.

Figure 7-2. Program Status Area



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 non-maskable 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.

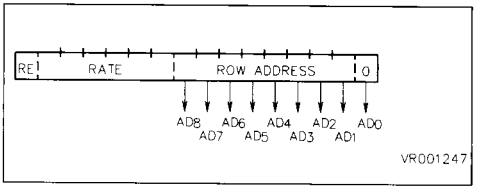
REFRESH

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.
- When the $\overline{STOP}$ line is activated, the CPU generates memory refreshes continuously.

Figure 8-1. Refresh Control Register



Refresh Cycles

The refresh mechanism is a way of generating a special kind of bus transaction called a *refresh cycle.*

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.

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 cyles. (Thus, if there is a 4MHz clock, the refresh period can be from 1μs to 64μs.)

The LDCTL instruction is used to set the refresh rate, to set or clear RE, or to initialize or read the ROW field.

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 $\overline{WAIT}$ lines 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.

Stop-state Refresh

The CPU has three internal operating states : Running, Stop, and Bus-Disconnect states Stop state is entered duing 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.

CHARACTERISTICS AND TIMING

AC Characteristics

No.	Symbol	Parameter	Z8001/Z8002		Z8001A/Z8002A		Z8001B/Z8002B	
			Min. (ns)	Max. (ns)	Min. (ns)	Max. (ns)	Min. (ns)	Max. (ns)
1	T_{CC}	Clock Cycle Time	250	2000	165	2000	100	2000
2	T_{WCH}	Clock Width (High)	105	2000	70	2000	40	
3	T_{WCL}	Clock Width (Low)	105	2000	70	2000	40	
4	T_{FC}	Clock Fall Time		20		10		10
5	T_{RC}	Clock Rise Time		20		15		10
(1) 6	$T_{DC(SNV)}$	Clock ↑ to Segment Number Valid (50pF load)		130		110		70
(1) 7	$T_{DC(SNN)}$	Clock ↑ to Segment Number Not Valid	20		10		5	
8	$T_{DC(BZ)}$	Clock ↑ to Bus Float		65		55		40
9	$T_{DC(A)}$	Clock ↑ to Address Valid		100		75		50
10	$T_{DC(AZ)}$	Clock ↑ to Address Float		65		55		40
11	$T_{DA(DR)}$	Address Valid to Read Data Required Valid		475*		305*		180*
12	$T_{SDR(C)}$	Read Data to Clock ↓ Setup Time	30		20		10	
13	$T_{DDS(A)}$	$\overline{DS}$ ↑ to Address Active	80*		45*		20*	
14	$T_{DC(DW)}$	Clock ↑ to Write Data Valid		100		75		50
15	$T_{HDR(DS)}$	Read Data to $\overline{DS}$ ↑ Hold Time	0		0		0	
16	$T_{DDW(DS)}$	Write Data Valid to $\overline{DS}$ ↑ Delay	295*		195*		110*	
17	$T_{DA(WR)}$	Address Valid to $\overline{MREQ}$ ↓ Delay	(55)*		(35)*		20*	
18	$T_{DC(MR)}$	Clock ↓ to $\overline{MREQ}$ ↓ Delay		80		70		40
19	T_{WMRH}	$\overline{MREQ}$ Width (High)	210*		135*		80*	
20	$T_{DMR(A)}$	$\overline{MREQ}$ ↓ to Address Not Active	70*		35*		20*	
21	$T_{DDW(DSW)}$	Write Data Valid to $\overline{DS}$ ↓ (Write) Delay	55*		35*		15*	
22	$T_{DMR(DR)}$	$\overline{MREQ}$ ↓ to Read Data Required Valid	375*		230*		140*	
23	$T_{DC(MR)}$	Clock ↓ $\overline{MREQ}$ ↑ Delay		80		60		45
24	$T_{DC(ASF)}$	Clock ↑ to $\overline{AS}$ ↓ Delay		80		60		40
25	$T_{DA(AS)}$	Address Valid to $\overline{AS}$ ↑ Delay	55*		35*		20*	
26	$T_{DC(ASR)}$	Clock ↓ to $\overline{AS}$ ↑ Delay		90		80		40
27	$T_{DA(DR)}$	$\overline{AS}$ ↑ to Read Data Required Valid	360*		220*		140*	
28	$T_{DDS(AS)}$	$\overline{DS}$ ↑ to $\overline{AS}$ ↓ Delay	70*		35*		15*	
29	T_{WAS}	$\overline{AS}$ Width (Low)	85*		55*		30*	
30	$T_{DAS(A)}$	$\overline{AS}$ ↑ to Address Not Active Delay	70		45		20*	
31	$T_{DAZ(DSR)}$	Address Float to $\overline{DS}$ (Read) ↓ Delay	0		0		0	
32	$T_{DAS(DSR)}$	$\overline{AS}$ ↑ to $\overline{DS}$ (Read) ↓ Delay	80*		55*		30*	
33	$T_{DDSR\ (DR)}$	$\overline{DS}$ (Read) ↓ to Read Data Required Valid	205*		130*		70*	
34	$T_{DC(DSR)}$	Clock ↓ to $\overline{DS}$ ↑ Delay		70		65		45
35	$T_{DDS(DW)}$	$\overline{DS}$ ↑ to Write Data Not Valid	75*		45*		25*	

Notes :

1. Only for Z8001

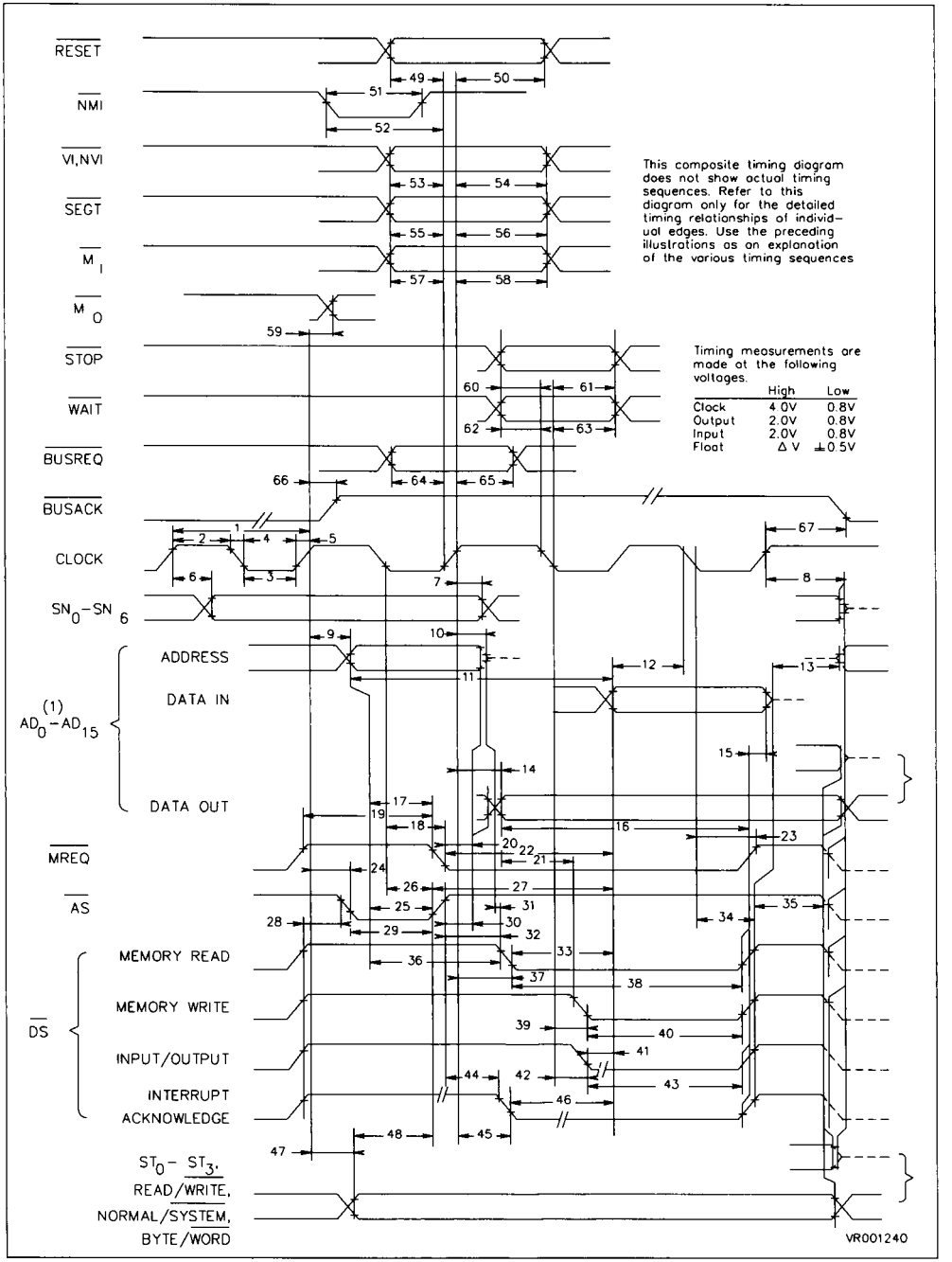
* Clock cycle-table dependent. See table on next page.

AC Characteristics (Continued)

No.	Symbol	Parameter	Z8001/Z8002		Z8001A/Z8002A		Z8001B/Z8002B	
			Min. (ns)	Max. (ns)	Min. (ns)	Max. (ns)	Min. (ns)	Max. (ns)
36	$T_{DA(DSR)}$	Address Valid to $\overline{DS}$ (Read) ↓ Delay	180*		110*		65*	
37	$T_{DC(DSR)}$	Clock ↑ to $\overline{DS}$ (Read) ↓ Delay		120		85		60
38	T_{WDSR}	$\overline{DS}$ (Read) Width (Low)	275*		185*		110*	
39	$T_{DC(DSW)}$	Clock ↓ to $\overline{DS}$ (Write) ↓ Delay		95		80		60
40	T_{WDSW}	$\overline{DS}$ (Write) Width (Low)	185*		110*		75*	
41	$T_{DSI(DR)}$	$\overline{DS}$ (I/O) ↓ to Read Data Required Valid	330*		210*		120*	
42	$T_{DC(DSF)}$	Clock ↓ to $\overline{DS}$ (I/O) ↓ Delay		120		90		60
43	T_{WDS}	$\overline{DS}$ (I/O) Width (Low)	410*		255*		160*	
44	$T_{DAS(DSA)}$	$\overline{AS}$ ↑ to $\overline{DS}$ (Acknowledge) ↓ Delay	1065*		690*		410*	
45	$T_{DC(DSA)}$	Clock ↑ to $\overline{DS}$ (Acknowledge) ↓ Delay		120		85		65
46	$T_{DDSA(DR)}$	$\overline{DS}$ (Acknowledge) ↓ to Read Data Required Delay	455*		295*		165*	
47	$T_{DC(S)}$	Clock ↑ to Status Valid Delay		110		85		60
48	$T_{DS(AS)}$	Status Valid to $\overline{AS}$ ↑ Delay	50*		30*		10*	
49	$T_{SR(C)}$	$\overline{RESET}$ to Clock ↑ Setup Time	180*		70		50	
50	$T_{HR(C)}$	$\overline{RESET}$ to Clock ↑ Hold Time	0		0		0	
51	T_{WNMI}	$\overline{NMI}$ Width (Low)	100		70		50	
52	$T_{SNMI(C)}$	$\overline{NMI}$ to Clock ↑ Setup Time	140		70		50	
53	$T_{SVI(C)}$	$\overline{VI}$, $\overline{NVI}$ to Clock ↑ Setup Time	110		50		40	
54	$T_{HVI(C)}$	$\overline{VI}$, $\overline{NVI}$ to Clock ↑ Hold Time	20		20		10	
55	$T_{SSGT(C)}$	$\overline{SEGT}$ to Clock ↑ Setup Time	70		55		40	
56	$T_{HSGT(C)}$	$\overline{SEGT}$ to Clock ↑ Hold Time	0		0		0	
57	$T_{SMI(C)}$	$\overline{MI}$ to Clock ↑ Setup Time	180		140		80	
58	$T_{HMI(C)}$	$\overline{MI}$ to Clock ↑ Hold Time	0		0		0	
59	$T_{DC(MD)}$	Clock ↑ to $\overline{MO}$ Delay		120		85		70
60	$T_{SSTP(C)}$	$\overline{STOP}$ to Clock ↓ Setup Time	140		100		50	
61	$T_{HSTP(C)}$	$\overline{STOP}$ to Clock ↓ Hold Time	0		0		0	
62	$T_{SW(C)}$	$\overline{WAIT}$ to Clock ↓ Setup Time	50		30		20	
63	$T_{HW(C)}$	$\overline{WAIT}$ to Clock ↓ Hold Time	10		10		5	
64	$T_{SBRQ(C)}$	$\overline{BUSREQ}$ to Clock ↑ Setup Time	90		80		60	
65	$T_{HBRQ(C)}$	$\overline{BUSREQ}$ to Clock ↑ Hold Time	10		10		5	
66	$T_{DC(BAKR)}$	Clock ↑ to $\overline{BUSACK}$ ↑ Delay		100		75		60
67	$T_{DC(BAKF)}$	Clock ↑ to $\overline{BUSACK}$ ↓ Delay		100		75		60
68	T_{WA}	Address Valid Width	150*		95*		50*	
69	$T_{DDS(S)}$	$\overline{DS}$ ↑ to $\overline{STATUS}$ Not Valid	80*		55*		30*	

Note : Clock cycle-table dependent. See table on next page.

Composite AC Timing Diagram



Clock-Cycle-Time-Dependent Characteristics

Number	Symbol	Z8001/Z8002 Equation	Z8001A/Z8002A Equation	Z8001B/Z8002B Equation
11	$T_{DA(DR)}$	$2T_{CC} + T_{WCH}$ - 130ns	$2T_{CC} + T_{WCH}$ - 95ns	$2T_{CC} + T_{WCH}$ - 60ns
13	$T_{DDS(A)}$	T_{WCL} - 25ns	T_{WCL} - 25ns	T_{WCL} - 20ns
16	$T_{DDW(DS)}$	$T_{CC} + T_{WCH}$ - 60ns	$T_{CC} + T_{WCH}$ - 40ns	$T_{CC} + T_{WCH}$ - 30ns
17	$T_{DA(WR)}$	T_{WCH} - 50ns	T_{WCH} - 35ns	T_{WCH} - 20ns
19	T_{WMRH}	T_{CC} - 40ns	T_{CC} - 30ns	T_{CC} - 20ns
20	$T_{DMR(A)}$	T_{WCL} - 35ns	T_{WCL} - 35ns	T_{WCL} - 20ns
21	$T_{DDW(DSW)}$	T_{WCH} - 50ns	T_{WCH} - 35ns	T_{WCH} - 25ns
22	$T_{DMR(DR)}$	$2T_{CC}$ - 130ns	$2T_{CC}$ - 100ns	$2T_{CC}$ - 60ns
25	$T_{DA(AS)}$	T_{WCH} - 50ns	T_{WCH} - 35ns	T_{WCH} - 20ns
27	$T_{DA(DR)}$	$2T_{CC}$ - 140ns	$2T_{CC}$ - 110ns	$2T_{CC}$ - 60ns
28	$T_{DDS(AS)}$	T_{WCL} - 35ns	T_{WCL} - 35ns	T_{WCL} - 25ns
29	T_{WAS}	T_{WCH} - 20ns	T_{WCH} - 15ns	T_{WCH} - 10ns
30	$T_{DAS(A)}$	T_{WCL} - 35ns	T_{WCL} - 25ns	T_{WCL} - 20ns
32	$T_{DAS(DR)}$	T_{WCL} - 25ns	T_{WCL} - 15ns	T_{WCL} - 10ns
33	$T_{DDSR\ (DR)}$	$T_{CC} + T_{WCH}$ - 150ns	$T_{CC} + T_{WCH}$ - 105ns	$T_{CC} + T_{WCH}$ - 70ns
35	$T_{DDS(DW)}$	T_{WCL} - 30ns	T_{WCL} - 25ns	T_{WCL} - 15ns
36	$T_{DA(DSR)}$	T_{CC} - 70ns	T_{CC} - 55 ns	T_{CC} - 35ns
38	T_{WDSR}	$T_{CC} + T_{WCH}$ - 80ns	$T_{CC} + T_{WCH}$ - 50ns	$T_{CC} + T_{WCH}$ - 30ns
40	T_{WDSW}	T_{CC} - 65ns	T_{CC} - 55ns	T_{CC} - 25ns
41	$T_{DDSI(DR)}$	$2T_{CC}$ - 170ns	$2T_{CC}$ - 120ns	$2T_{CC}$ - 80ns
43	T_{WDS}	$2T_{CC}$ - 90ns	$2T_{CC}$ - 75ns	$2T_{CC}$ - 40ns
44	$T_{DAS(DSA)}$	$4T_{CC} + T_{WCL}$ - 40ns	$4T_{CC} + T_{WCL}$ - 40ns	$4T_{CC} + T_{WCL}$ - 30ns
46	$T_{DDSA(DR)}$	$2T_{CC} + T_{WCH}$ - 150ns	$2T_{CC} + T_{WCH}$ - 105ns	$2T_{CC} + T_{WCH}$ - 75ns
48	$T_{DS(AS)}$	T_{WCH} - 55ns	T_{WCH} - 40ns	T_{WCH} - 30ns
68	T_{WA}	T_{CC} - 90ns	T_{CC} - 70ns	T_{CC} - 50ns
69	$T_{DDS(S)}$	T_{WCL} - 25ns	T_{WCL} - 15ns	T_{WCL} - 10ns

Absolute Maximum Ratings

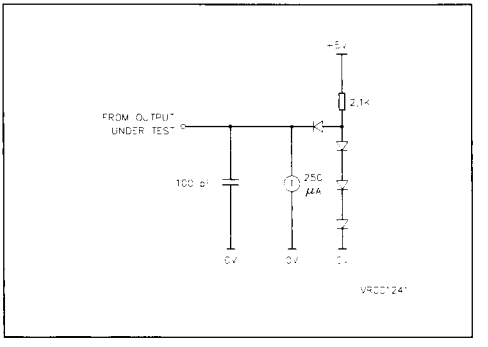
Symbol	Parameter	Value	Unit
V_{DD}	Voltages on all inputs and outputs with respect to GND	-0.3 to +7.0	V
T_A	Operating Ambient Temp.	0 to +70	°C

Note : 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.

Test Conditions

The characteristics below apply for the following test conditions, unless otherwise noted. All voltages are referenced to GND (0V). Positive current flows into the referenced pin. Available operating temperature range is :

0°C to +70°C, V_{CC} = 5V ± 5%



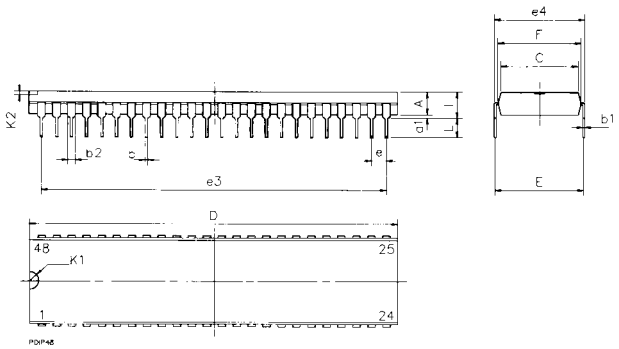
All AC parameters assume a load capacitance of 100pF max., except for parameter 6 (50pF max). Timing references between two output signals assume a load difference of 50pF max.

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	
$V_{IH\ RESET}$	Input High Voltage on $\overline{RESET}$ pin	2.4	V_{CC} to .3	V	
V_{IL}	Input Low Voltage	-0.3	0.8	V	
V_{OH}	Ouput HIgh Voltage	2.4		V	I_{OH} = -250µA
V_{OL}	Output Low Voltage		0.4	V	I_{OL} = +2.0mA
I_{IL}	Input Leakage		±10	µA	0.4 ≤ V_{IN} ≤ +2.4V
$I_{IL\ SEGT}$	Input Leakage on SEGT pin	-100	100	µA	
I_{OL}	Ouput Leakage		±10	µA	0.4 ≤ V_{IN} ≤ +2.4V
I_{CC}	V_{CC} Supply Current		300	mA	4MHz - 6MHz
I_{CC}	V_{CC} Supply Current		350	mA	10MHz

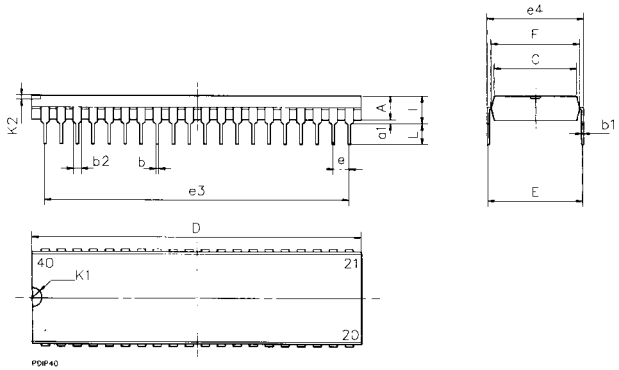
MECHANICAL PACKAGE

Figure 9-1. PDIP48



Dim.	mm			inches		
	Min	Typ	Max	Min	Typ	Max
A	–	–	–	–	–	–
a1		0.63			0.024	
b		0.45			0.017	
b1	0.23		0.31	0.009		0.012
b2		1.27			0.050	
C	–	–	–	–	–	–
D			62.74			2.470
E	15.20		16.68	0.598		0.656
e		2.54			0.100	
e3		58.42			2.300	
e4	–	–	–	–	–	–
F			14.10			0.555
I		4.45			0.175	
L		3.30			0.129	

Figure 9-2. PDIP40



Dim.	mm			inches		
	Min	Typ	Max	Min	Typ	Max
A	–	–	–	–	–	–
a1		0.63			0.025	
b		0.45			0.018	
b1	0.23		0.31	0.009		0.012
b2		1.27			0.050	
C	–	–	–	–	–	–
D			52.58			2.070
E	15.20		16.68	0.598		0.656
e		2.54			0.100	
e3		48.26			1.900	
e4	–	–	–	–	–	–
F			14.10			0.555
I		4.45			0.175	
L		3.30			0.129	

ORDERING INFORMATION

Type	Package		Temperature	Clock
Z8001 B1V	Plastic	DIL48	0/+70°C	4MHz
Z8001 AB1V				6MHz
Z8001 BB1V				10MHz
Z8002 B1V	Plastic	DIL 40	0/+70°C	4MHz
Z8002 AB1V				6MHz
Z8002 BB1V				10MHz

Information furnished is believed to be accurate and reliable. However, SGS-THOMSON Microelectronics assumes no responsibility for the consequences of use of such information nor for any infringement of patents or other rights of third parties which may result from its use. No license is granted by implication or otherwise under any patent or patent rights of SGS-THOMSON Microelectronics. Specification mentioned in this publication are subject to change without notice. This publication supersedes and replaces all information previously supplied. SGS-THOMSON Microelectronics products are not authorized for use as critical components in life support devices or systems without express written approval of SGS-THOMSON Microelectronics.

© 1991 SGS-THOMSON Microelectronics – Printed in Italy – All Rights Reserved

SGS-THOMSON Microelectronics GROUP OF COMPANIES
Australia - Brazil - France - Germany - Hong Kong - Italy - Japan - Korea - Malaysia - Malta - Morocco - The Netherlands - Singapore - Spain - Sweden - Switzerland - Taiwan - United Kingdom - U.S.A.

82

APR 27 1993 036563 ✓