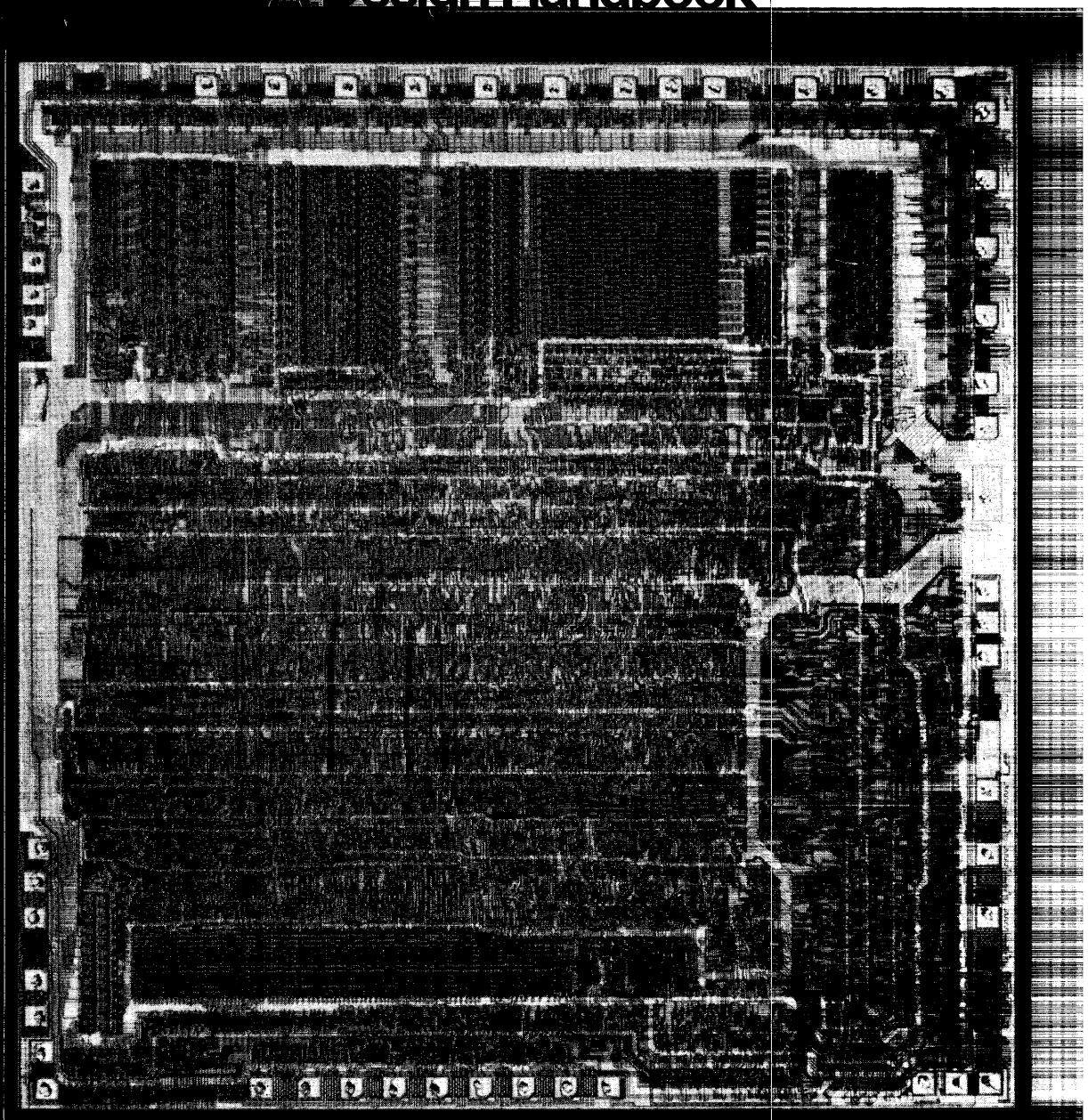
THE Z8000 MICROPROCESSOR

A Design Handbook



BRADLY K. FAWCETT

THE Z8000 MICROPROCESSOR
A Design Handbook

BRADLY K. FAWCETT

Senior Staff Engineer
Zilog, Inc.

PRENTICE-HALL, INC., Englewood Cliffs, N.J. 07632

Library of Congress Cataloging in Publication Data

Fawcett, Bradly K.
The Z8000 microprocessor.

Bibliography: p.
Includes index.
1. Electronic digital computers—Circuits.
2. Logic design. 3. Zilog Model Z8000 (Computer)
I. Title.
TK7888.F285 621.3819'58 82-392
ISBN 0-13-983742-6 AACR2
ISBN 0-13-983734-5 (pbk.)

Editorial production/supervision: *Barbara Bernstein*
Manufacturing buyer: *Gordon Osbourne*

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-13-983742-6
ISBN 0-13-983734-5 {pbk}

PRENTICE-HALL INTERNATIONAL, INC., *London*
PRENTICE-HALL OF AUSTRALIA PTY. LIMITED, *Sydney*
PRENTICE-HALL OF CANADA, LTD., *Toronto*
PRENTICE-HALL OF INDIA PRIVATE LIMITED, *New Delhi*
PRENTICE-HALL OF JAPAN, INC., *Tokyo*
PRENTICE-HALL OF SOUTHEAST ASIA PTE. LTD., *Singapore*
WHITEHALL BOOKS LIMITED, *Wellington, New Zealand*

Contents

Preface

With the advent of the 16-bit microprocessor, computing power and features formerly available only in minicomputers and large-scale computers are now provided on a single integrated-circuit chip. This text is a detailed study of one such microprocessor, the Z8000.

The emphasis of this book is on logic design with the Z8001 and Z8002 microprocessors. Other components in the Z8000 family of parts are also discussed. The components described in this book are available from

Zilog, Inc.
1315 Dell Avenue
Campbell, CA 95008

This book is intended for anyone interested in learning about the Z8000, but will be especially useful for those engineers involved in either hardware designs that use Z8000 family components or software/firmware designs for Z8000-based systems. The hardware engineer would use this text as a guide to interfacing the Z8000 microprocessors to memory and peripheral devices. Since effective assembly language programming depends on a thorough knowledge of the target processor's capabilities, the software engineer would use this text to gain insights into the Z8000's architecture and its relation to the microcomputer system.

Chapters 1 through 8 and Chapter 11 deal principally with the Z8001 and Z8002 microprocessors from both an architecture and system design viewpoint. Several system architecture concepts, such as operating systems

and memory management, are introduced and discussed in relation to Z8000 system design. Chapters 9, 10, 12, and 13 deal with other Z8000 family components that often are used in Z8000-based systems, including the Z8010 Memory Management Unit, peripherals, memories, and slave processors.

Since the emphasis of this book is on logic design, assembly language programming of the Z8000 is not covered in detail. An overview of the instruction set is included in Chapter 8. Several good texts that deal exclusively with Z8000 programming are currently available (see the Bibliography. Appendix C).

This book is not intended for the computer novice. The reader is assumed to have some experience with microprocessors and a familiarity with concepts such as registers, buffers, program counters, and interrupts.

As with any project of this magnitude, a great number of people were involved—too many to mention here—and the author wishes to thank them all. Special thanks to Steve Blank, who originally suggested that I write this book, and to the following engineers at Zilog, who constituted an informal technical review committee: John Banning, Ross Freeman, Dave Stevenson, Carl Johnson, Nai-Ting Hsu, Don Alpert, Dan Hillman, Gary Prosenko, and Pat Lin.

Every attempt has been made to assure the technical accuracy of the material in this book, but Mr. Murphy is not to be denied and, inevitably, errors will be found. Comments and criticisms are appreciated and will contribute to the accuracy of later editions.

Many of the figures in this book are from the following publications: *1981 Data Book*, Zilog, Inc., document 00-2034-01, 1981; *Z8000 CPU User's Reference Manual*, Zilog, Inc., Prentice-Hall, Inc., 1982; *A Small Z8000 System Application Note*, Zilog, Inc., document 03-8060-02, 1980; and *Z8010 MMU Technical Manual*, Zilog, Inc., document 00-20150A0, 1981.

BRADLY K. FAWCETT

ZILOG SALES OFFICES

West

Sales & Technical Center
Zilog, Incorporated
1315 Dell Avenue
Campbell, California 95008
Phone: (408) 446-4666
TWX: 910-338-7621

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

Sales & Technical Center
Zilog, Incorporated
15643 Sherman Way
Suite 430
Van Nuys, CA 91406
Phone: (213) 989-7485
TWX: 910-495-1765

Sales & Technical Center
Zilog, Incorporated
2918 N. 67th Place #2
Scottsdale, AZ 85251
Phone: (602) 990-1977

Sales & Technical Center
Zilog, Incorporated
1750 112th Ave. N.E.
Suite D161
Bellevue, WA 98004
Phone: (206) 454-5597

Midwest

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

Sales & Technical Center
Zilog, Incorporated
28349 Chagrin Blvd.
Suite 109
Woodmore, OH 44122
Phone: (216) 831-7040
FAX: 216-831-2957

South

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

Zilog, Incorporated
7113 Burnet Rd.
Suite 207
Austin, TX 78757
Phone: (512) 453-3216

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

East

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

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

Sales & Technical Center
Zilog, Incorporated
240 Cedar Knolls Rd.
Cedar Knolls, NJ 07927
Phone: (201) 540-1671

Zilog Sales
3300 Buckeye Rd.
Suite 401
Atlanta, GA 30341
Phone: (404) 451-8425

United Kingdom

Zilog (U.K.) Limited
Babbage House, King Street
Maidenhead SL6 IDU
Berkshire, United Kingdom
Phone: (628) 36131
Telex: 848609

West Germany

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

France

Zilog, Incorporated
Tour Europe
Cedex 7
92080 Paris La Defense
France
Phone: (1) 788-14-33
TWX: 611445F

Japan

Zilog, Japan KK
TBS Taikan Bldg.
3-3 Akasaka 5-Chome
Minato-Ku, Tokyo 107
Phone: (03) 587-0528
Telex: ESSOEAST J22846

1

The Z8000 CPU

Microprocessors were introduced in the early 1970s as 4-bit and 8-bit processors that were used largely as programmable logic devices. These devices could handle simple tasks in products such as calculators, games, and control systems. The advent of more sophisticated 8-bit microprocessors, such as the Z80, led to the design of microprocessor-based computer systems with sufficient memory resources to perform simple data processing tasks and more complex control functions. However, these 8-bit systems could rarely match the memory addressing and data processing abilities of the minicomputers and mainframe computers available at the time.

The Z8000 microprocessor represents a further step in the evolution of microelectronics—a 16-bit central processing unit (CPU) with features that, until now, could be found only in sophisticated mini- and large-scale computers. The Z8000 is not an extension of 8-bit architectures; its architecture is closer to that of popular minicomputers such as the PDP-11 than to the Z80. The high throughput, abundant CPU resources, and powerful instruction set of the Z8000 extend well beyond traditional microprocessor architectures. Special features in the Z8000 support operating-system-type environments, manipulation of complex data structures, and multiprocessing systems. Thus the Z8000 incorporates on one LSI chip an architecture designed to accommodate a wide range of applications, from relatively simple control systems to powerful data processing computers.

The Z8000 microprocessor is available in two versions: the Z8001 and Z8002. The Z8001 is a 48-pin CPU that can directly access 8 megabytes of memory per memory address space; the Z8002 is a 40-pin CPU that can di-

rectly access 65,536 bytes of memory per memory space. A memory management unit, the Z8010 MMU, can be used with the Z8001 in memory-intensive applications to organize and control memory use efficiently. An entire family of support devices is available, including intelligent peripheral controllers, serial and parallel input/output devices, and direct memory access (DMA) controllers.

The architectural features of the Z8000 CPU include sixteen 16-bit general-purpose registers, eight addressing modes, and an instruction set more powerful than that of most minicomputers. Data types ranging from single bits to 32-bit fields and byte strings are supported. The 110 instruction types combine with the addressing modes and data types to form a set of over 400 distinct instructions. These CPU resources exhibit a high degree of regularity. For example, more than 90% of the instructions use any of five main addressing modes, and operate on 8-bit, 16-bit, or 32-bit data types. This regularity in the CPU architecture greatly simplifies the programming of Z8000 processors. Operating system support is provided by features such as system and normal modes, multiple stacks, and a sophisticated interrupt structure. Hardware and software support of multiprocessing environments also is included in the Z8000 architecture.

External devices called Extended Processor Units (EPUs) can be used to extend the Z8000's basic instruction set. EPUs unburden the CPU by performing complex, time-consuming tasks such as floating-point arithmetic or data base management.

OPERATING SYSTEMS

The limited CPU resources and relatively small address spaces of previous generations of microprocessors made it difficult, if not impossible, to develop systems based on those microprocessors that could process more than one programming task at a time. (A programming task is the execution of one program, operating on its data.) Thus microcomputer systems have traditionally been single-user, single-task systems in which the user has direct control over the machine hardware. In contrast, the Z8000's regular architecture and large address space facilitate the design of Z8000-based systems that can handle multiple users and multiple programming tasks.

When multiple programming tasks are executed concurrently on a computer system, each task requires the use of some of the system's hardware resources. These resources include the CPU, memory, and input/output (I/O) devices. Invariably, there are not enough resources in the system to support the simultaneous execution of all the programming tasks; a method for sharing various resources between all the system's tasks must be defined. Multitasking computer systems usually employ a supervisory software program to coordinate and control the various programming tasks; such supervisory soft-

ware is called an operating system. An operating system's basic function is to allocate the system's resources in an efficient manner, while ensuring that the various tasks on the system execute without unintentionally affecting each other. Users of such systems would write their programs, sometimes called applications programs, to run under the control of the operating system.

In multitasking systems, the details of the hardware functions (for example, the timing and buffering of an I/O operation) are handled by the operating system. The user, whose program runs under the control of the operating system, no longer needs to be concerned with the intricacies of the machine hardware; he or she only needs to know how to access the operating system function that will produce the desired result (requesting that data be output to a printer, for example). Thus the user writes the applications program to run under the control of a particular operating system, not a particular hardware configuration (Fig. 1.1). In fact, only the operating system software should have direct access to hardware-related functions, in order to keep individual users' tasks from affecting each other. For example, if a given user were allowed to reconfigure a programmable I/O device, execution of other users' tasks that access that device could be affected in unpredictable and undesirable ways. Therefore, applications programs running under the control of the operating system should access hardware functions (such as I/O functions) only through the operating system.

The most valuable resource in any computer system is the CPU. If more than one applications program is to be executed by the CPU, the operating system must be able to switch the CPU easily from one programming task to another. A set of scheduling rules within the operating system will determine when the switching of tasks will occur. For example, batch operating systems execute each programming task sequentially from start to finish, whereas time-sharing systems execute each task for a given length of time, in a round-robin arrangement. If the operating system can switch tasks very quickly, the user will have the impression that the tasks are executing concurrently. Of course, each time the operating system switches to a new task, status information describing the previous task must be saved so that it can be resumed later in an orderly manner. The results of

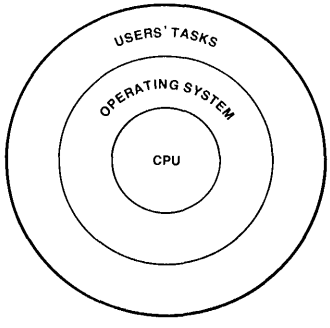


Figure 1.1 Users' tasks access the hardware through the operating system.

the execution of a given task should be the results that would appear if the programming task were run sequentially from start to finish without interruption.

The operating system must control the allocation of memory space as well as the allocation of CPU time. As the operating system switches the CPU from one programming task to another, it must ensure that the program is available in main memory for the CPU. Memory space for each task's program and data must be allocated in a manner that ensures that the execution of one task does not interfere with the execution of any other task. Often, memory allocation during task switching involves moving code or data between main memory and a secondary storage device such as a disk or magnetic tape. (The management of memory resources in Z8000 systems is discussed in Chapter 9.)

I/O handling and allocation is also an integral part of most operating systems. As mentioned above, having the operating system control all I/O accesses frees the individual users from having to know the details of the hardware and keeps tasks from interfering with each other. I/O devices often must be allocated to a specific task; for example, if a user wants to print a table of data, no other task should be allowed to use that printer until that user's print operation is complete. Interrupt processing must also be controlled by the operating system, not by the applications programs; since interrupts are caused by events external to the CPU operations, there is no way to predict which user's task will be running when an interrupt occurs.

Other functions that are controlled by operating systems include task synchronization (in applications where the ordering of tasks is critical), task communication (in applications where data are passed between tasks), debugging functions, and other resource allocation problems within a given computer system. Operating systems, like hardware designs, can be general purpose in nature or oriented to a particular application.

Many features of the Z8000 CPU architecture are designed to support the implementation of operating system software on Z8000-based computer systems. These features will be emphasized throughout this book as the various parts of the architecture are discussed. However, this does not mean to imply that Z8000-based systems must include an operating system. Many applications, such as most process control applications, would involve running only one programming task, making operating system software unnecessary.

THE Z-BUS

Every computer system can be thought of as having four major parts: a central processor unit (CPU), storage, input, and output (Fig. 1.2). The CPU is the primary functioning unit of the computer and usually includes

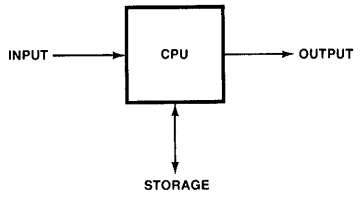


Figure 1.2 Major parts of a computer.

circuitry for performing arithmetic and logical functions (an arithmetic and logic unit, or ALU), some storage for ALU operands and results (registers), and control logic. The control logic decodes instructions and determines their execution and sequencing by sending the appropriate signals to the ALU and registers in the CPU, and the external storage and I/O devices outside the CPU. Of course, the instructions and data to be acted on must exist somewhere and be available to the CPU; hence storage devices such as semiconductor memories need to be interfaced to the CPU. The manipulation of programs and data by the CPU would be useless without some method of getting information from and sending results to the outside world; thus input and output devices such as card readers, cathode ray tube (CRT) terminals, and printers are needed. All four parts—CPU, storage, input, and output—must be present in some form in a computer system.

The Z8001 and Z8002 devices are microprocessors, not microcomputers. These chips comprise only the CPU portion of a computer system. Thus hardware design with the Z8000 microprocessors is largely a matter of interfacing the Z8000 CPUs with the appropriate memory and I/O devices. The specification of the signals used to interface the Z8001 and Z8002 to other components in the Z8000 family of parts is called the Z-Bus. A discussion of how to interface memory and peripheral devices to the Z8000 microprocessor is, then, actually a discussion of the Z-Bus signals.

The Z-Bus is not a hardware bus definition like the S-100 or similar buses, where certain signals are assigned to certain pins on a specific type of connector for a specific type of circuit board. The Z-Bus is merely the logical definition of the interface signals used in a Z8000-based system; the actual hardware implementation of this interface is left up to the designer.

Two types of operations can occur on the Z-Bus: transactions and requests. Transactions involve the transfer of data between two Z-Bus-compatible devices. Transactions are initiated by the bus master, that is, the one device in the system (usually a CPU) that has control of the Z-Bus. All data transfers occur over a 16-bit time-multiplexed address/data bus. The timing of Z-Bus data transfers is asynchronous; the sending and receiving device do not need to be synchronized by a common clock. Several Z-Bus signals are used to control the transfer, and only one transaction can proceed at a time.

Six kinds of transactions can occur on the Z-Bus: memory transactions, I/O transactions, interrupt and trap acknowledgments, EPU transfers,

memory refreshes, and internal operations. Memory transactions involve the transfer of a byte (8 bits) or word (16 bits) of data between the bus master and a memory location. I/O transactions are used to transfer a byte or word of data between the bus master and a peripheral device. Interrupt/trap acknowledgments are used to acknowledge an interrupt or trap and to transfer a word of data from the interrupting device to the CPU. EPU transfers are used to transfer a word of data between an Extended Processing Unit and the CPU. Refreshes and internal operations are the only bus transactions that do not involve a transfer of data. Refresh transactions are used to refresh dynamic memories. Internal operations occur when the CPU is performing an operation that does not require data to be transferred on the bus.

Z-Bus requests occur when a device other than the bus master needs to request access to a resource in the system. Only the bus master can initiate transactions, but a request can be initiated by any component that does not control the bus. Four types of requests can occur: an attempt to gain a bus master's attention (an interrupt request), an attempt to become a bus master (a bus request), a request for some shared resource in a multiprocessor system, such as a request to access a disk drive that is shared by two different processors' systems (a resource request), and a request to delay CPU instruction execution (a stop request). A daisy-chained priority mechanism is used to resolve conflicts resulting from simultaneous requests, eliminating the need for separate priority controllers. Thus the Z-Bus consists of a set of signals used to provide timing, control, address, and data information for Z-Bus transactions, requests, and request acknowledgments. These signals will be discussed in detail as the various Z8000 components interfaced by the Z-Bus are described throughout this book.

A Z-Bus component is a device that conforms to the Z-Bus interface protocols. Most of the components of the Z8000 family are Z-Bus components. There are four categories of Z-Bus components: CPUs, peripherals, requesters, and memories. A Z-Bus CPU is the default bus master and initiates most data transfers. Interrupt requests and bus control requests are serviced by the CPU. The Z8001 and Z8002 are Z-Bus CPUs. A Z-Bus peripheral is a device that responds to I/O transactions and generates interrupt requests. The Z8036 Counter/Timer Input/Output Circuit (CIO) and Z8038 FIFO Input/Output Interface Unit (FIO) are examples of Z-Bus peripherals. A Z-Bus requester is a device that can make bus requests, and when given control of the bus, can initiate data transactions. The Z8016 DMA Transfer Controller (DTC) is a Z-Bus requester. Z-Bus memories interface directly to the Z-Bus and respond to memory transactions. The Z6132 Quasi-Static RAM is a Z-Bus memory device.

The Z8000 family of parts are not the only components that can be interfaced with the Z-Bus. Z80 family components, for example, although not directly Z-Bus compatible, can be easily interfaced to Z-Bus CPUs using readily available transistor-transitor logic (TTL) components.

Z8000 CPU ARCHITECTURE

The Z8000 CPU is a single-voltage metal-oxide semiconductor (MOS) integrated circuit measuring about $\frac{1}{4}$ inch on a side and containing the equivalent of more than 17,500 transistors (Fig. 1.3). The circuit design uses dis-

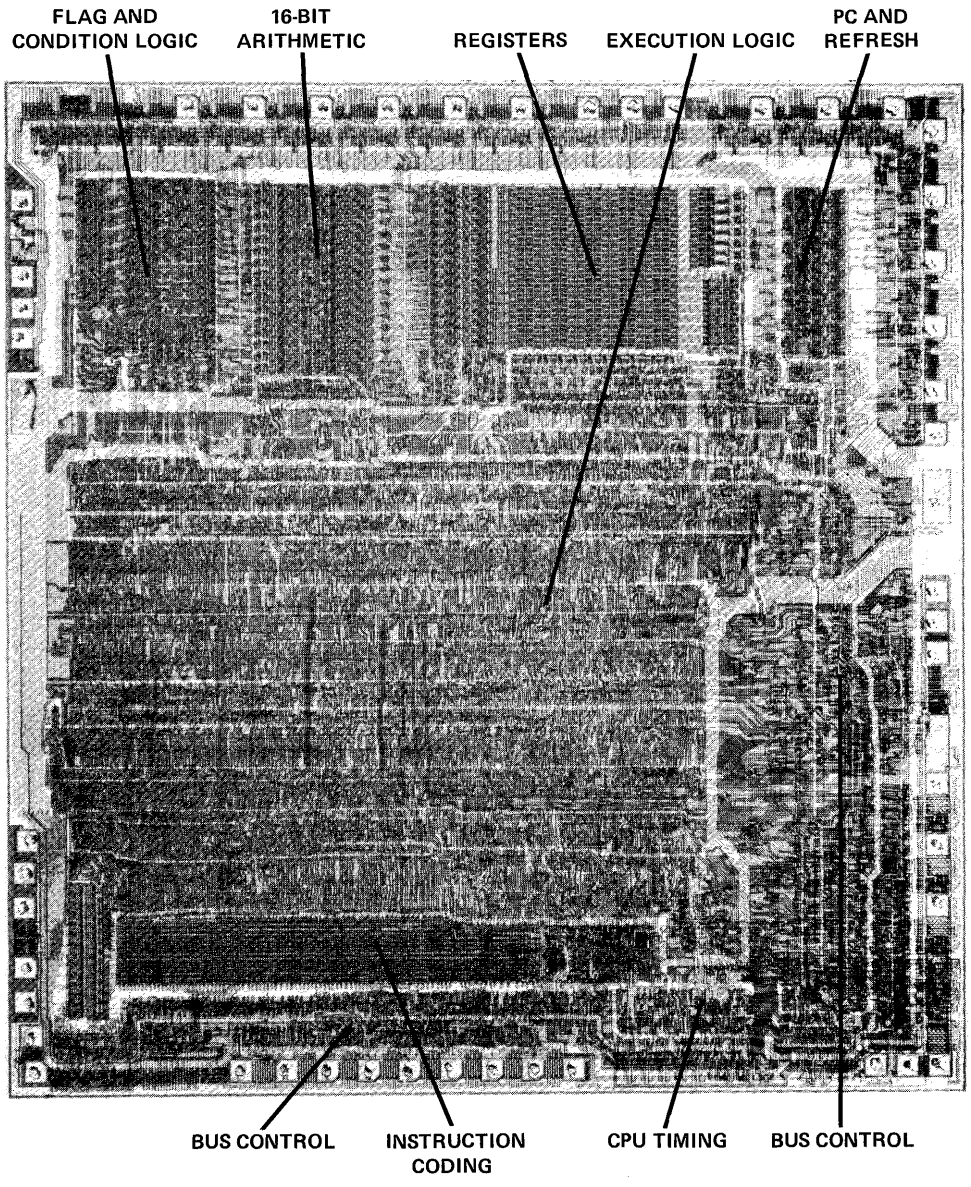


Figure 1.3 Z8000 CPU.

crete logic, not microcoding, to control instruction execution. This design approach results in a smaller circuit chip, thereby improving signal speed and making the device easier to manufacture. The Z8000 CPU has two versions, the 48-pin Z8001 and the 40-pin Z8002, which differ only in the manner and range of memory addressing. Physically, the two chips are identical; a forty-ninth bonding pad is used to configure the chip as a Z8001 or Z8002 before hermetic sealing into its dual-in-line package.

Figure 1.4 is a block diagram illustrating the major elements of the Z8000 CPU. A 16-bit bus is used for moving addresses and data within the CPU. Arithmetic and logical operations are performed on addresses and data by the ALU. The instruction execution control logic handles the fetching and execution of instructions. Exception handling control logic processes interrupts and traps. Automatic dynamic memory refresh mechanisms can be enabled with the refresh control logic. Sixteen 16-bit general-purpose registers are available to the programmer. Four additional special-purpose registers help control CPU operation: the program counter (PC), flag and control word (FCW), program status area pointer (PSAP), and refresh register. Communication with external memory and I/O devices is via a Z-Bus interface.

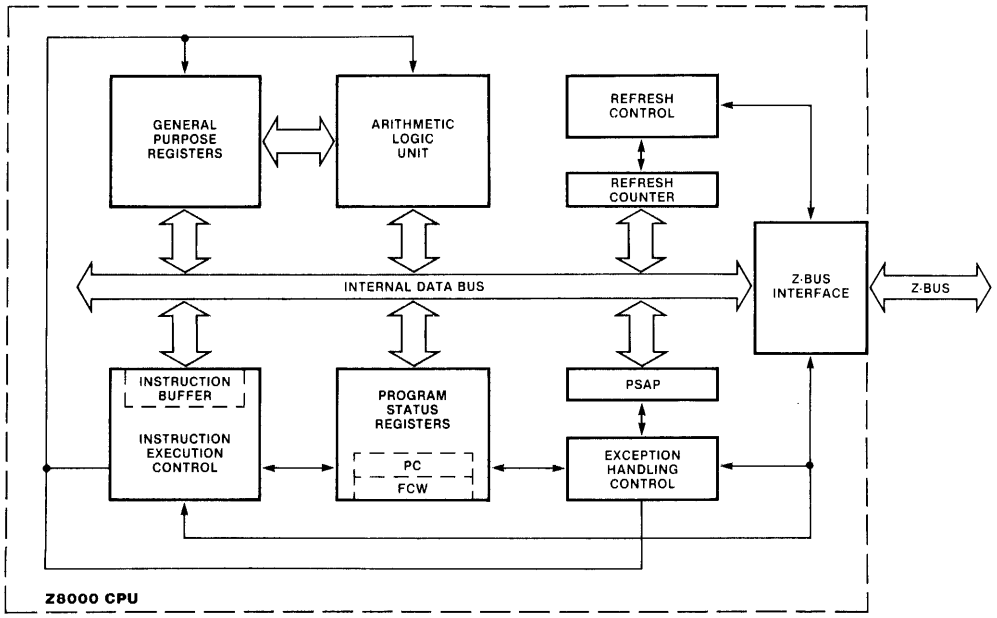


Figure 1.4 Z8000 CPU functional block diagram.

SYSTEM AND NORMAL MODES

The Z8000 CPUs execute in two different operating modes: system mode and normal mode. A control bit in the flag and control word (FCW) determines the current operating mode. The operating modes dictate which instructions can be executed and which stack pointers are used. In the system mode, all instructions can be executed; in the normal mode, instructions that directly affect the system hardware cannot be executed. The set of instructions that can be executed in system mode only are called privileged instructions and consist of all the input and output instructions, instructions that affect the FCW, PSAP, and refresh registers in the CPU, and the multi-micro instructions. (These instructions are described in Chapter 8. The stack pointers are discussed in Chapter 2.)

The CPU switches operating modes whenever the appropriate bit in the FCW is changed. This bit can be altered by a Load Control (LDCTL) instruction or by an exception condition (interrupt, trap, or reset). The Load Control instruction is a privileged instruction and provides a means for switching from system to normal mode. A special instruction, the System Call (SC), is used to generate a trap, providing a controlled means of switching from the normal to the system mode. An attempt to execute a privileged instruction while in the normal mode also generates a trap condition.

The distinction between system and normal modes allows the implementation of protected operating systems on Z8000-based computers. Operating system software would run in the system mode, controlling the system's resources and managing the execution of users' applications programs, which would run in the normal mode. Since normal-mode users cannot execute privileged instructions, those users cannot directly control those aspects of the CPU that affect the system's hardware configuration. If a normal-mode program needs to perform a hardware-related function such as an I/O operation, a request to the operating system can be made, via the trap mechanism (see Chapter 6). Thus only the operating system software performs hardware-related functions.

Z8002 PIN CONFIGURATION

Figure 1.5 shows the Z8002 CPU with the pins grouped according to function. Activity on these pins is governed by the Z-Bus protocols.

Address/Data Bus

The address/data lines (AD0-AD15) constitute a 16-bit time-multiplexed address and data bus; that is, sometimes these signals are addresses and some-

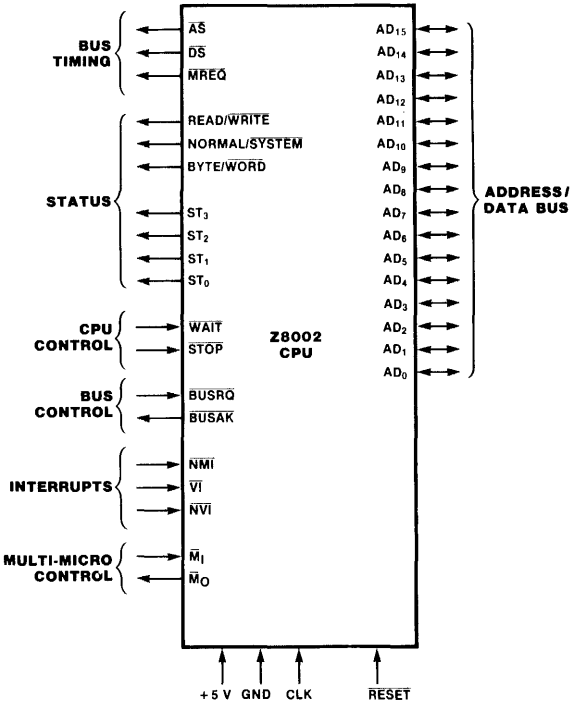


Figure 1.5 Z8002 pin functions.

times data. AD0 is the least significant bit position, and AD15 is the most significant. The addresses on this bus can be memory or I/O port addresses, depending on the type of transaction taking place. In the Z8002, I/O and memory addresses are always 16-bit words; the Z8002 can directly address 65,536 bytes of memory (64K bytes, where K = 1024) and 65,536 peripheral devices per address space. Addresses are always emitted by the CPU, but data can be an input or output, depending on whether the current transaction is a read or a write operation.

The address/data bus is a multiplexed bus in order to minimize the pin count on the CPU's package. Sixteen additional pins would be required to have separate, dedicated address and data buses, with very little, if any, gain in processor efficiency. Separate address and data lines could improve processor performance only during transactions where the address and data can be sent out simultaneously, that is, only during write operations. However, since instruction fetches are always memory read operation, read operations typically occur about eight times as often as write operations. Furthermore, most memory chips currently available cannot simultaneously accept both the address and data to be stored, so separate address and data buses would provide no timing advantage during memory writes. Therefore, the benefits of a package with fewer pins—higher reliability, smaller size, and decreased

power consumption—far outweigh any timing benefits that might accrue from having separate, nonmultiplexed address and data buses.

The multiplexing of the address and data buses also simplifies the direct addressing of internal control registers in programmable peripheral devices, without having separate address and data pins on those chips' packages.

Bus Timing Signals

The bus timing signals—address strobe ($\overline{AS}$), data strobe ($\overline{DS}$), and memory request ($\overline{MREQ}$)—are CPU outputs that control bus transactions by determining when the address/data bus holds addresses and when it holds data. In a typical data transfer, $\overline{AS}$ goes active (the bar above the signal name denotes an active low signal), indicating to external memory and I/O devices that valid address and status information is present on the bus. (The status lines are discussed below.) Thus the occurrence of an address strobe signals the start of a data transfer. Sometime later, $\overline{AS}$ becomes inactive and $\overline{DS}$ goes low, indicating that the data to be written to the previously addressed external device are now on the address/data bus, or that the data to be read from the device can be placed on the bus (Fig. 1.6). The timing of data transfers between the CPU and other devices is determined solely by the address and data strobes; the CPU and other devices do not have to share a

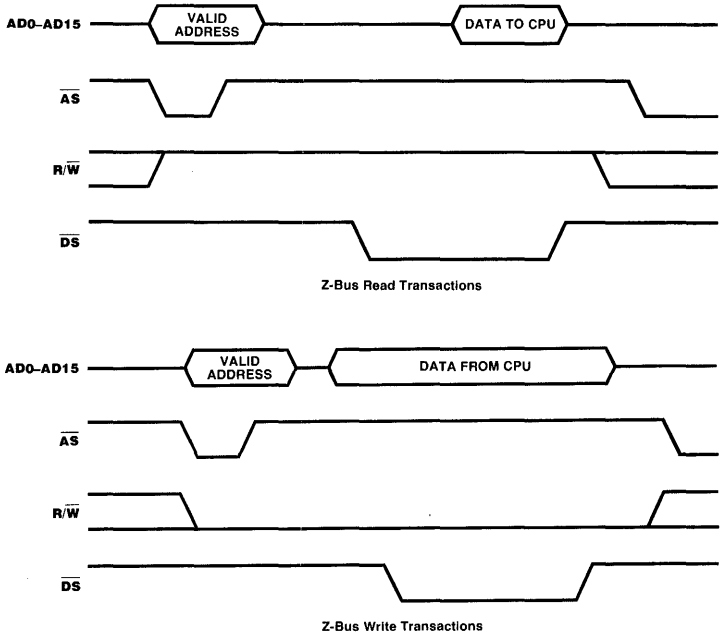


Figure 1.6 Z-Bus data transfers.

common clock signal. The address and data strobes are mutually exclusive, of course; only one or the other is active at any given time during the data transfer.

The $\overline{MREQ}$ signal indicates that the address/data bus is holding a memory address, and its falling edge can be used to time control signals to a memory system. However, as discussed below, certain of the status signals also can be used to differentiate between memory and I/O transactions; as a result, the $\overline{MREQ}$ signal is not used in many Z8000 system designs.

Status Signals

The status signals are CPU outputs that describe the type of transaction that is occurring on the address/data bus. This status information would be used to enable the appropriate buffers, drivers, and chip select logic necessary for proper completion of the data transfer.

The read/write signal (R/$\overline{W}$) describes the direction of the current data transfer; a low signal indicates that the CPU is writing data to an external device and a high signal indicates that the CPU is reading data from an external device (Fig. 1.6).

The byte/word line (B/$\overline{W}$) describes the size of the data field being transferred; a low indicates that a word (16 bits) is being transferred, whereas a high indicates a byte (8-bit) transfer. Bytes of data might be transferred on the lower half (AD0–AD7) or upper half (AD8–AD15) of the bus, depending on the address of the device involved in the transaction (see Chapter 3).

The normal/system signal (N/$\overline{S}$) indicates the current operating mode of the CPU; a low indicates system mode and a high indicates normal mode. This signal could be used by memory control logic to define two separate memory address spaces: normal-mode memory and system-mode memory. In other words, a Z8002-based system could include two separate areas of memory, with each area containing a maximum of 64K bytes. Memory accesses made when the N/$\overline{S}$ pin is high would access normal-mode memory, and accesses made when the N/$\overline{S}$ pin is low would access system-mode memory. The operating system software, which runs in system mode, would be in system-mode memory, inaccessible to the users' programs, which run in the normal mode and reside in normal-mode memory. Thus systems with memory control logic that uses the N/$\overline{S}$ signal to distinguish two memory address spaces would have built-in protection features that prevent individual users from accessing the operating system software.

Four additional status signals, ST0, ST1, ST2, and ST3, define the exact type of transaction occurring on the bus, as shown in Table 1.1.

The internal operation status code (0000) indicates that the CPU is involved in an ALU or other internal operation and that no data transfers are occurring on the bus. Internal CPU cycles will occur during the execution of

TABLE 1.1 STATUS CODE MEANINGS

ST3	ST2	ST1	ST0	Meaning
0	0	0	0	Internal operation
0	0	0	1	Memory refresh
0	0	1	0	Standard I/O reference
0	0	1	1	Special I/O reference
0	1	0	0	Segment trap acknowledge
0	1	0	1	Nonmaskable interrupt acknowledge
0	1	1	0	Nonvectored interrupt acknowledge
0	1	1	1	Vectored interrupt acknowledge
1	0	0	0	Data memory reference
1	0	0	1	Stack memory reference
1	0	1	0	EPU-data memory transfer
1	0	1	1	EPU-stack memory transfer
1	1	0	0	Instruction fetch, nth word (IFn)
1	1	0	1	Instruction fetch, first word (IF1)
1	1	1	0	CPU-EPU transfer
1	1	1	1	Reserved (not used in Z8001 and Z8002)

instructions where several arithmetic or logical operations are performed between data transfers, such as the Divide instruction.

The refresh status code (0001) indicates that a refresh cycle for dynamic memories is occurring on the bus. (The automatic memory refresh mechanism is described in Chapter 3.) Refresh and internal operations are the only bus transactions that do not involve a data transfer.

Two types of I/O transactions can occur on the bus, standard I/O (status of 0010) and special I/O (status of 0011). A given I/O operation generates standard or special I/O status depending solely on the I/O instruction being executed; there are separate I/O and special I/O instructions. During an I/O access, the state of the status lines can be used as part of the I/O devices' chip-select logic to define two separate I/O address spaces, a standard I/O address space and a special I/O address space (Fig. 1.7). Thus the Z8002 can address 65,536 standard I/O devices and 65,536 special I/O devices.

The only difference between standard I/O and special I/O bus transactions is the code that appears on the status lines. As a general convention, standard I/O operations will be used to access Z8000 peripherals, and special I/O operations will be used to access CPU support chips such as the Z8010 Memory Management Unit.

Four status codes indicate a transfer between the CPU and memory devices. A 1000 on the status lines means that the CPU is reading or writing an instruction's data operand; a 1001 code signals that the CPU is reading or writing to the stack; a 1101 code indicates that the CPU is fetching the first word of an instruction; a 1100 code signals the fetching of subsequent words in an instruction. Just as with I/O accesses, these status lines can be used to define separate memory address spaces: a data memory address space (corre-

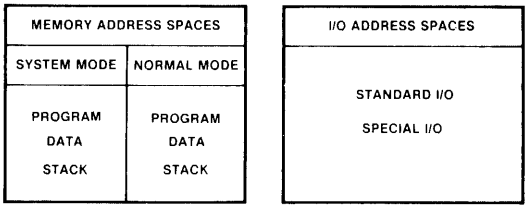


Figure 1.7 Z8000 memory and I/O address spaces.

sponding to the 1000 status code), a stack memory address space (the 1001 code), and a program memory address space (the 1100 and 1101 codes). Program code would be located in the program memory space and the data to be acted on in the data memory space. Stack memory is used as temporary storage and to hold program status information in the event of a subroutine call or exception condition. When combined with the N/$\overline{S}$ signal, six memory address spaces are possible: system-mode data, system-mode stack, system-mode program, normal-mode data, normal-mode stack, and normal-mode program (Fig. 1.7). For the Z8002, each of these spaces can hold up to 64K bytes of memory.

Extended Processor Units (EPUs) are devices that can be added to a Z8000 system to unburden the CPU from complex tasks, such as floating-point arithmetic. EPUs are designed to act on data resident in their internal registers; the CPU is responsible for transferring data between the EPUs registers and the rest of the system. Three status codes are reserved for transactions involving EPUs: the 1010 status indicates a transfer between an EPU register and data memory, a 1011 status indicates a transfer between an EPU register and stack memory, and a 1110 status indicates a transfer between an EPU register and a CPU register. (EPUs are discussed in Chapter 10.)

Status codes 0100 through 0111 indicate that the current bus activity is an interrupt or trap acknowledge sequence (see Chapter 6). These status lines would be decoded to generate the appropriate acknowledge signal for the interrupting device.

The 1111 status code is reserved for use in future, upward-compatible Z8000-family CPUs.

CPU Control

These control signals are CPU inputs that allow external devices to delay the operation of the CPU. The $\overline{WAIT}$ line can be used by memory or peripheral devices to increase the delay between the address strobe and data strobe during bus transactions. Data transfers on the Z-Bus are asynchronous; a slow memory or I/O device can stretch the timing of data transfers by an arbitrary length through control of the CPU's $\overline{WAIT}$ input. (Timing details are discussed in Chapter 5.) The $\overline{STOP}$ input is used to halt CPU operation

immediately before the fetch of the next instruction. EPUs use the $\overline{STOP}$ signal to synchronize their activities with the CPU.

Interrupts

Three different types of interrupt inputs are supported by the Z8000 architecture: nonmaskable interrupts, nonvectored interrupts, and vectored interrupts. The $\overline{NMI}$ signal is an interrupt input that cannot be disabled. Nonmaskable interrupts usually are reserved for catastrophic events that require the immediate attention of the CPU, such as an imminent power failure. The nonvectored interrupt ($\overline{NVI}$) and vectored interrupt ($\overline{VI}$) inputs can be enabled and disabled via manipulation of the CPU's flag and control word (FCW). When an interrupt is detected at one of these three inputs, information about the currently executing program is saved, and a routine to handle the interrupt is invoked (see Chapter 6). Nonmaskable interrupts and nonvectored interrupts each have one routine specified for servicing the interrupt. Vectored interrupts can result in the execution of one of a number of possible interrupt service routines. Which of those routines is executed will depend on a byte of data, called a vector, that is received from the interrupting device during the interrupt acknowledge cycle.

Bus Control

The bus control signals are used to implement a request/acknowledge daisy chain that other devices in the system can use to request control of the bus. $\overline{BUSREQ}$ is an input indicating that a Z-Bus requester (a DMA device, for example) is trying to gain control of the bus. The $\overline{BUSACK}$ output goes active when the CPU relinquishes control of the bus in response to a bus request. The CPU gives up control of the bus by tri-stating (electrically neutralizing) the address/data bus, bus timing signals, and bus status signals.

Multi-micro Control

The multi-micro in ($\overline{MI}$) and multi-micro out ($\overline{MO}$) signals are the CPUs interface to the Z-Bus resource-request daisy chain. These signals allow multiple processors to share common resources in a well-defined and controlled manner. (Z-Bus resource requests are discussed in Chapter 7.)

Z8001 PIN CONFIGURATION

The Z8001 differs from the Z8002 in the manner and range of memory addressing. All of the signal pins previously described for the Z8002 are also on the Z8001, and the Z8001 has eight additional pins (Fig. 1.8). These

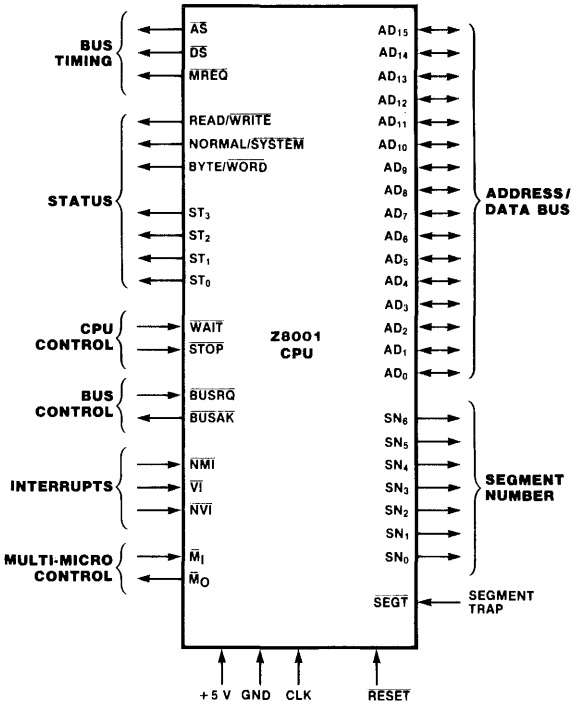


Figure 1.8 Z8001 pin functions.

eight additional signals are used to increase the Z8001's memory addressing capabilities.

Segment Number

Seven output pins (SN0–SN7) define a segment number. Memory addresses in the Z8001 consist of a 7-bit segment number and the 16-bit address that appears on the address/data bus when address strobe is active. The 16-bit portion of the address is called the offset. Since memory addresses are 23 bits long, the Z8001 can directly address 8,388,608 (8 megabytes or 8M bytes) of memory per memory address space. As with the Z8002, memory can be divided into the six distinct memory address spaces shown in Fig. 1.6, using the N/S and ST0–ST3 signals. Each of the six address spaces can, then, hold a maximum of 8M bytes. The segment number is not used as part of an I/O address; I/O addresses in the Z8001 and Z8002 are always 16 bits long.

Segment Trap

The segment trap signal ($\overline{\text{SEGT}}$) is an input to the CPU. Memory management logic can use this signal to cause a trap if an illegal memory access is attempted.

SEGMENTED ADDRESSING AND MEMORY MANAGEMENT

The Z8001 and Z8002 CPUs generate memory addresses of different lengths and types. The Z8002 uses a 16-bit address to specify one of 64K bytes of memory in a memory address space. Within each address space, memory is addressed in a linear manner. The Z8001 uses 23 bits to address memory, but this address is separated into a 7-bit segment number and a 16-bit segment offset. The segment number and offset portions of the address are distinct; the segment number specifies one of 128 possible segments, or blocks, of memory, and the offset specifies one of up to 64K bytes in that segment. Each memory segment is an independent block of memory; instructions and multiple-byte data elements cannot cross segment boundaries. The segment number cannot be altered by effective address calculations during instruction execution, such as indexing. The Z8001 can address 128 memory segments per memory address space, with each segment having a maximum of 64K bytes.

This division of memory into distinct blocks, called memory segmentation, provides a natural way of partitioning memory into different functional areas. Modern structured programming techniques dictate that a program's memory be divided into distinct areas dedicated to particular uses. For example, different areas of memory might hold the program's instruction code, data variables, and a buffer for an I/O device. Each of these memory areas may have particular attributes associated with that section; the program code might be in read-only memory, and an I/O buffer's memory might be accessible only during system-mode operation. Segmentation reflects this use of memory by allowing the programmer to specify different segments for each distinct memory area. Thus segmentation provides a convenient means for partitioning a large memory address space.

Further advantages of segmentation are realized when implementing a memory management scheme for providing memory protection and relocation. Memory is a limited resource within a computer system, a resource that often must be shared by many different and complex programming tasks. Memory management involves the efficient organization of those memory resources, while ensuring that each task has sufficient memory available when needed, without corrupting the execution of other tasks. Thus memory management is the process of allocating and protecting the system's memory resources; it is usually implemented with a combination of hardware logic and operating system software. A typical configuration would consist of memory-control logic that is programmed dynamically by the operating system.

The primary means of controlling the allocation of memory is by mapping logical addresses to physical addresses. As a processor executes a task, program code and data must be read from and written to memory. The addresses that appear in the program and are output by the CPU are called

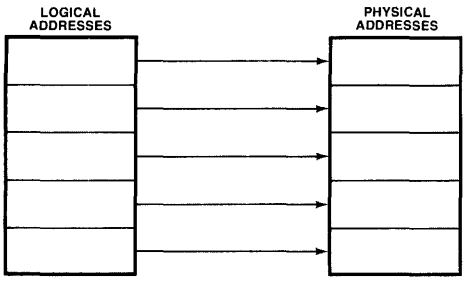


Figure 1.9 Example of identical logical and physical memory addresses.

logical addresses. On the other hand, the actual memory addresses in the system's hardware that access particular locations are called physical addresses. In simple systems with no memory management, logical and physical addresses are identical (Fig. 1.9). In more complex systems, the memory manager maps the logical addresses into the physical address space as programs execute (Fig. 1.10). Thus each independent user of the system need not be concerned that the logical addresses within a given applications program are the same as the logical addresses of another program on the system; the memory manager will route the logical addresses for each user into different physical memory addresses.

For example, Fig. 1.11 illustrates a system with two users, both of whom have specified logical addresses 4000 to 5000 in their programs. When user A's program is running, the memory manager will translate user A's logical addresses to physical addresses 3000 to 4000; when user B's program is running, the memory manager will translate user B's logical addresses to physical addresses 7000 to 8000. Of course, the operating system will have to inform the memory manager each time it switches between tasks. Thus, using a mapping algorithm, the memory manager can place each task's code and data anywhere within physical memory. Logical addresses emitted by each programming task are translated by the memory manager to the proper physical addresses for that task's code and data.

The logical address space might be larger, smaller, or equal in size to the

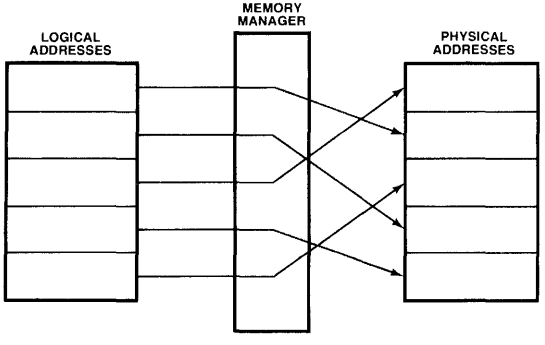


Figure 1.10 Mapping logical addresses to physical addresses using a memory manager.

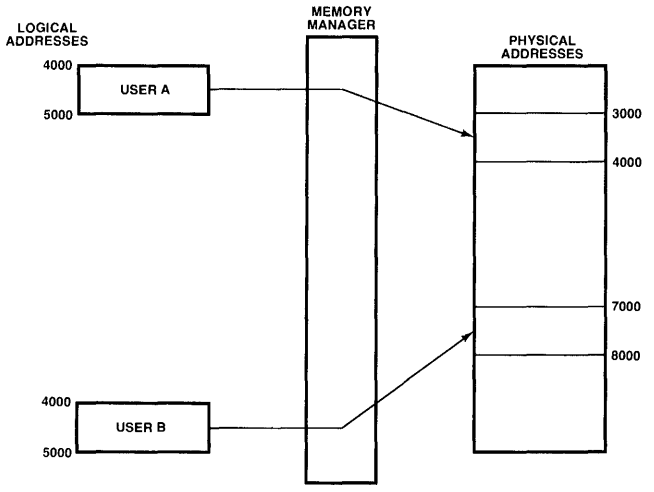


Figure 1.11 Mapping two users' logical addresses into physical addresses.

physical address space. Systems whose logical address space is larger than the physical address space are called virtual memory systems. In virtual memory systems, or in systems with a large number of different programming tasks, the memory requirements for all tasks currently running under operating system control might exceed the available physical memory. Therefore, for some users, only parts of their program's code and data may be in main memory at any given time. Suppose that the addresses that are "missing" in main memory for a given task are somehow marked in the memory manager. The memory manager can stop execution by causing a trap when a reference is made to a "missing" address. The trap routine could then retrieve the task's missing code or data from an intermediate storage device (a floppy disk system, for example), place it in physical memory, and allow the task to continue execution from where it left off. Space in physical memory would have to be found, which might involve bumping some other task's code or data onto the disk or other storage device. The operating system would have to keep careful track of which areas of physical memory are being used and in what way they are being used.

The use of segmented addressing in the Z8001 supports the implementation of memory management logic in Z8001-based systems. A memory segment is essentially a standard, variable-sized block of memory that can be assigned common attributes. Translation of logical to physical addresses can occur on a segment-by-segment basis. Virtual memory systems can swap entire segments between main memory and intermediate storage devices, as needed. Memory attributes such as read-only or system-mode-only can be assigned to segments. The memory manager would use the Z8001's segment trap ($\overline{\text{SEGT}}$) input to signal the CPU in the event of an illegal access, such as

an attempted write to a read-only segment. Thus, with memory segmentation, each task's code, data, and stack area can be assigned its own segment, thereby structuring memory in accordance with modern modular programming techniques. Furthermore, the protection attributes assigned to these segments help define the interface between the various tasks' program modules. For example, by placing the operating system software in segments with the "system-mode-only" attribute, the memory manager can automatically prevent users' programs, which run in normal mode, from altering the operating system. The Z8010 Memory Management Unit, a programmable memory manager for Z8001-based systems, is discussed in Chapter 9.

SEGMENTED AND NONSEGMENTED MODES

The type of addressing scheme used in a Z8000 system will affect how addresses are stored in that system. The Z8002's 16-bit addresses, sometimes called nonsegmented addresses, can be stored in a 16-bit register or in a word of memory. The Z8001's 23-bit segmented addresses, on the other hand, are embedded in a 32-bit-long word, and therefore require two 16-bit registers or two words of memory when stored. However, a method of using nonsegmented addresses in Z8001 programs is provided.

The Z8001 executes programs in one of two segmentation modes, segmented mode or nonsegmented mode, as determined by a control bit in the CPU's flag and control word (FCW). The segmentation mode determines the size and format of addresses that are directly manipulated by the program; in the segmented mode, programs act on 23-bit segmented addresses, and in the nonsegmented mode, programs act on 16-bit nonsegmented addresses. The segmented mode is available only on the Z8001; the Z8002 always executes in the nonsegmented mode. Therefore, programs written for the Z8002 can be run on the Z8001 in the nonsegmented mode without alteration.

CPU OPERATING STATES

The Z8000 CPUs have three basic operating states: running state, stop/refresh state, and bus-disconnect state. Figure 1.12 illustrates these states and the conditions that can cause a change in state.

The running state is the normal state of the processor, wherein the CPU is executing instructions and handling exception conditions (interrupts and traps). While the CPU is in the running state, execution is controlled by the program counter (PC) and the flag and control word (FCW). The PC holds the memory address from which the next instruction is to be fetched. The

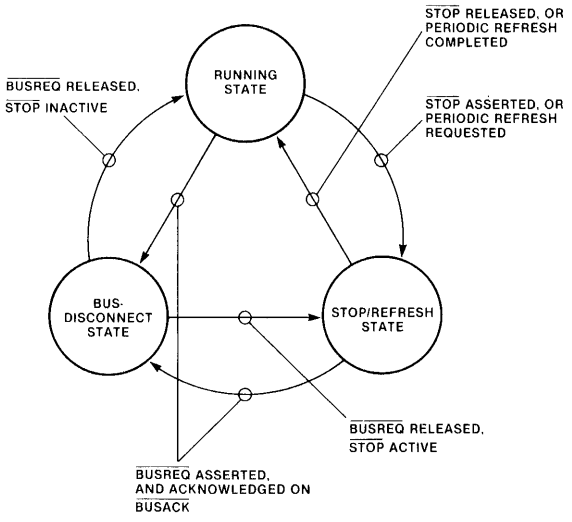


Figure 1.12 Z8000 operating states and transitions.

FCW contains control bits that determine the operating modes (system or normal, segmented or nonsegmented) and which interrupts are enabled. Instruction execution consists of two steps: (1) a single instruction of one or more words is fetched from program memory (IF1 or IFn status on the ST0-ST3 lines) at the address specified by the PC, and (2) the operation specified by the instruction is performed, with the PC and the flags in the FCW updated accordingly. After each instruction's execution, the CPU checks if any interrupts or traps are pending and enabled. If so, instruction execution is halted and an acknowledge sequence is performed (see Chapter 6). Three conditions can cause the CPU to leave the running state: a refresh request from the automatic memory refresh logic, the activation of the $\overline{\text{STOP}}$ input (a low at the input), or a bus request.

While in the stop/refresh state, the CPU generates memory refresh cycles (see Chapters 5 and 10), and does not perform any other functions. This feature is used by EPUs to suspend program execution. The CPU returns to the running state when the automatic refresh logic has completed its memory refresh operation, or when the $\overline{\text{STOP}}$ input is inactivated (returns high). A bus request while in the stop/refresh state will cause a transition to the bus-disconnect state.

The CPU enters the bus-disconnect state after receiving a bus request on the $\overline{\text{BUSREQ}}$ input and acknowledging it on the $\overline{\text{BUSACK}}$ output. While in this state, the CPU disconnects itself from the bus by tri-stating the address/data bus, bus timing, and bus status outputs. The CPU leaves this state when $\overline{\text{BUSREQ}}$ is inactivated. The bus-disconnect state is the highest-priority state, in that a bus request will force the CPU into this state regardless of what other state it is in and what other inputs it receives.

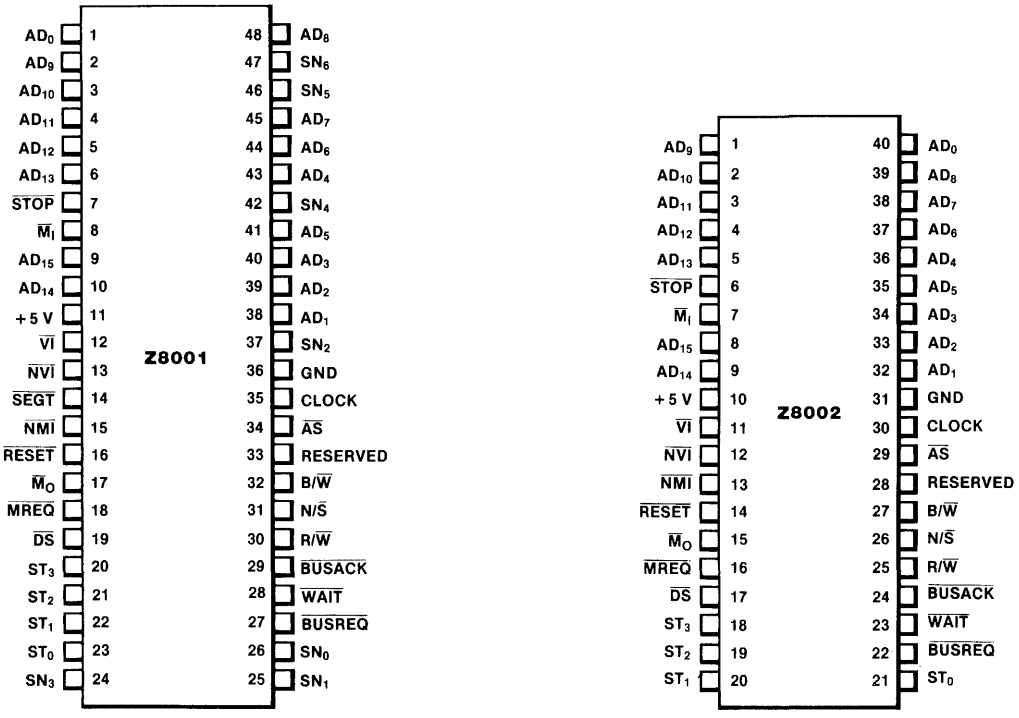


Figure 1.13 Z8000 CPU pin diagrams.

SYSTEM INPUTS

Figure 1.13 shows the full pin diagram for the Z8001 and Z8002. The Z8000 CPUs need a +5-V power source and draw a maximum of 300 mA. A single-phase clock is required; the clock specifications are discussed below. The $\overline{\text{RESET}}$ input provides a means of putting the CPU in a known starting condition (see Chapter 6). Note that one pin on each of the CPUs is not used; that pin is reserved for use on future upward-compatible Z8000 CPUs.

CPU CLOCK REQUIREMENTS

The Z8000 CPUs are dynamic MOS parts which require a single-phase clock input. The maximum clock rate is 4 MHz for the Z8001 and Z8002, 6 MHz for the Z8001A and Z8002A, and 10 MHz for the Z8001B and Z8002B. Since the parts are dynamic, the clock cannot be held high or low for more than 2 μs; therefore, the slowest allowable clock rate is 250 kHz. A TTL-generated clock signal is not adequate to drive the Z8000; active drivers are required to meet the stringent level, rise-time, and fall-time requirements.

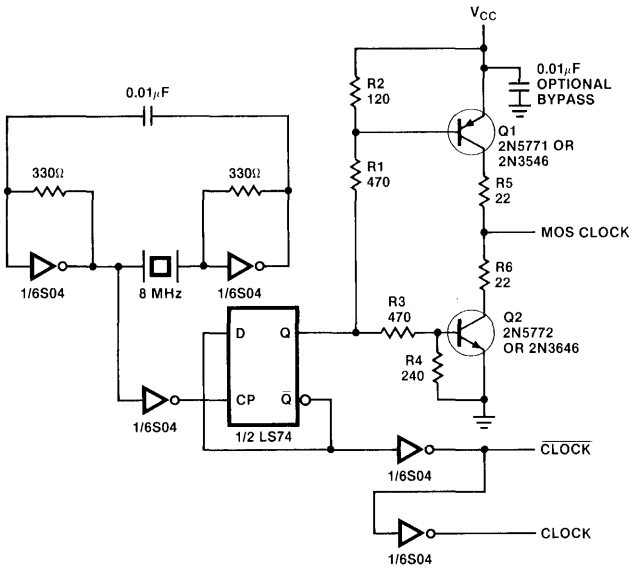


Figure 1.14 Z8000 CPU clock driver circuitry.

The clock input high voltage must be not more than 0.3 V above or less than 0.4 V below the +5-V power input. Rise and fall times cannot exceed 20 ns for the 4-MHz parts and 10 ns for the 6-MHz parts.

Figure 1.14 shows a clock driver circuit for the 4-MHz Z8000. A 4-MHz square wave is generated by dividing an 8-MHz crystal oscillator with a toggling flip-flop. A resistive pull-up could ensure the required clock-high level but cannot guarantee the required rise time while driving the chip's 50-pF input capacitance. The rise-time and fall-time requirements dictate the use of active pull-up and pull-down circuitry for the CPU clock. A TTL clock also is generated, for possible use by other circuits in the system.

2

CPU Registers

The Z8000 CPU is a register-oriented machine, with a set of sixteen 16-bit general-purpose registers and four special CPU control registers. Storing data in the registers allows shorter instructions and faster execution than with instructions that fetch data from memory. The CPU architecture provides a very regular register structure for manipulating byte (8-bit), word (16-bit), long-word (32-bit), and quad-word (64-bit) values. Certain special instructions also access specific bits in byte and word registers, and nibbles (4 bits) in bytes; individual bits can be set, reset, and tested, and nibbles are used to hold digits for binary-coded-decimal (BCD) arithmetic operations. Bits in a byte or word are numbered right to left starting from 0, from the least to the most significant bit (Fig. 2.1). Thus bit position n corresponds to the value 2^n in the representation of positive binary numbers.

Z8002 GENERAL-PURPOSE REGISTERS

The general-purpose register set of the Z8002 consists of 16 word registers, labeled R0 through R15, as illustrated in Fig. 2.2. Register data formats ranging from bytes to quad words are created by grouping and overlapping the 16 word registers. Sixteen byte registers (RL0, RH0, . . . , RL7, RH7) overlap the first eight word registers; RL0 is the least significant byte of word register R0, RH0 is the most significant byte of R0, and so on through RH7. (Any byte register can hold two digits for BCD arithmetic operations.) Eight long-word registers (RR0, . . . , RR14) are formed by grouping the

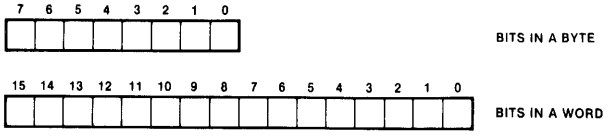


Figure 2.1 Numbering of bits in registers.

word registers into pairs (in fact, long-word registers are sometimes called register pairs); long-word register RR0 consists of word registers R0 and R1, RR2 consists of R2 and R3, and so on. The even-numbered register is the most significant word of the long word. The quad-word registers (RQ0, . . . , RQ12) are accessed only by the multiply and divide instructions. Each consists of four word registers; quad register RQ0 consists of word registers R0 through R3, with R0 being most significant. As a result of these groupings, half of the 16-bit registers and all of the 32-bit and 64-bit registers have addressable halves. For example, each half of word register R0 can be addressed separately as byte registers RH0 and RL0. Half of the 32-bit registers and all of the 64-bit registers have addressable quarters. This type of register hierarchy facilitates many programming tasks.

All of these registers are general purpose in nature and can be used to hold data or addresses. Each register can be used as an accumulator, that is, the source or destination of data involved in an ALU operation. Every word register, except R0, can hold a memory or I/O address (for indirect memory or I/O references), an index (for indexed memory references), or a stack pointer. Since all of these registers are general purpose, the particular use to which a register is put can vary during the course of a program, giving the programmer a great deal of flexibility. This architecture avoids the programming bottlenecks of an implied or dedicated register architecture, in which register contents must be saved and restored whenever the need for registers

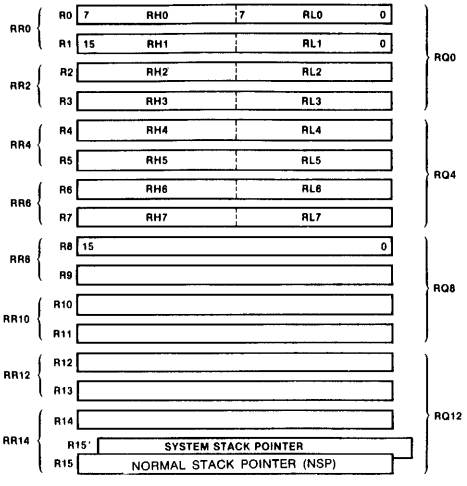


Figure 2.2 Z8002 general-purpose registers.

of a particular type exceeds the number of registers of that type in the processor. (For example, the Z80 architecture has only one accumulator, and most Z80 assembly language programs include numerous instructions to shift data in and out of the accumulator.) Furthermore, the grouping of the registers facilitates their efficient use; the Z8000 programmer does not need to dedicate a whole 16-bit register to hold a byte of data.

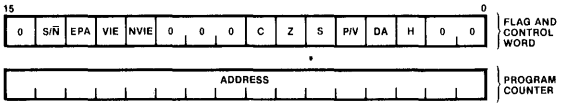
Included in the general-purpose register set is an implied stack pointer, R15. There are actually two implied stack pointers, one for system-mode operation and another for normal-mode operation. Although any register except R0 can be used as a stack pointer with the PUSH and POP instructions, R15 is the stack pointer used when program status information is to be saved on a stack. The system-mode stack pointer R15′ is used for saving program status information when an interrupt or trap occurs and for saving the PC during subroutine calls in system mode. The normal-mode stack pointer R15 is used for subroutine calls in normal mode. Since the implied stack pointer is a member of the general-purpose register set, it can be manipulated by any instruction that operates on the registers. In normal-mode operation, references to R15 will access the normal-mode stack pointer, and in system-mode operation, references to R15 will access the system-mode stack pointer. Thus the operating system and users' application programs can have separate stacks, and the normal-mode users cannot alter the system stack pointer. The normal-mode stack pointer can be accessed in system mode, however, as a special control register. This allows the operating system to initialize each normal-mode users' stack area pointer.

Z8002 CONTROL REGISTERS

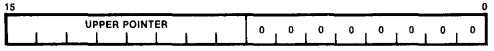
In addition to the general-purpose registers, the Z8002 has four 16-bit special-purpose registers that control the CPU's operation: the program counter (PC), flag and control word (FCW), program status area pointer (PSAP), and refresh control register (Fig. 2.3).

The PC and FCW are referred to as the program status registers. When an interrupt or trap occurs, these registers are saved on the system stack, and new program status describing the running environment of the service routine is loaded (see Chapter 6). The program counter holds the 16-bit address from which the next instruction is to be fetched; it is updated as each instruction is executed. The flag and control word contains control bits for determining the CPU operating modes and status flags for use in program branching instructions, such as jumps and returns.

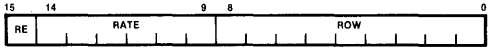
The low-order bits of the FCW (bits 0–7) hold the status flags that reflect the result of ALU operations. Bit 7, the carry flag (C), indicates a carry out of the most significant bit position of the register used as the destination for the result of an arithmetic operation. Bit 6, the zero flag (Z), is set when



Z8002 Program Status Registers



Z8002 Program Status Area Pointer



Z8002 Refresh Counter

Figure 2.3 Z8002 control registers.

the result of an operation is zero and cleared if the result is nonzero. Bit 5, the sign flag (S), is set when the result of an arithmetic operation is negative in two's-complement notation; that is, the most significant bit of the result is a 1. Bit 4, the parity/overflow flag (P/V), is an odd-parity bit for logical operations, and an overflow flag for arithmetic operations. When set, the overflow flag indicates that the result is greater than the largest number or less than the smallest number that can be represented in two's-complement notation in the destination register. Bit 3, the decimal adjust flag (D), is used for BCD arithmetic and indicates whether the last operation was an addition or subtraction. The Decimal Adjust Byte (DAB) instruction uses this flag as part of an algorithm for adjusting the binary results of an addition or subtraction of BCD digits into the correct BCD form. Bit 2, the half-carry flag (H), also is used by the DAB instruction; it indicates a carry out or a borrow into bit 3 as a result of an addition or subtraction of BCD digits. Neither the D nor the H flags are normally accessed by the programmer. Bits 0 and 1 of the FCW are not used.

The processor flags provide a means of controlling program branches and loops. The result of executing one instruction that alters the flags may determine the operation of a subsequent instruction that tests the flags' values, typically a conditional jump or return. The whole lower byte of the FCW can be read from or written to a byte register in system or normal modes using a special control instruction.

The upper byte of the FCW (bits 8–15) contains control bits that determine the operating modes of the CPU and control the interrupts. Bit 15 is always a 0 in the Z8002, signifying that the Z8002 always runs in the nonsegmented mode. Bit 14 is a the system/normal bit ($S/\overline{N}$); when this bit is a 1, the CPU is in the system mode, and when this bit is a 0, the CPU is in the

normal mode. The N/$\overline{S}$ output pin is the complement of the S/$\overline{N}$ bit in the FCW. Bit 13 is the extended processor architecture enable bit (EPA). When set to 1, this bit indicates that Extended Processor Units are present in the system, and, therefore, instructions reserved for EPUs will be executed. When the EPA bit is 0, the occurrence of an instruction code reserved for EPUs will cause a trap (see Chapters 6 and 10). Bit 12 is the vectored interrupt enable bit (VIE) and bit 11 is the nonvectored interrupt enable bit (NVIE). Setting the appropriate bit to a 1 enables the interrupt. Bits 8, 9, and 10 of the FCW are not used. These control bits can only be accessed in the system mode, via privileged CPU control instructions (see Chapter 9). Thus only the operating system software can alter the CPU's operating modes.

The program status area pointer (PSAP) holds a 16-bit address that points to an area in memory called the program status area. The program status area holds a list of the program status values (values for the PC and FCW) for every possible interrupt and trap service routine; the program status area resides in the system program memory address space. The low-order byte of the PSAP is always zero. (The program status area is described in Chapter 6.).

The refresh control register, consisting of a 9-bit row counter, a 6-bit rate counter, and an enable bit, is used to implement CPU-controlled, automatic refresh for dynamic memory. This refresh mechanism is discussed in Chapter 3.

Z8001 GENERAL-PURPOSE REGISTERS

The Z8001's general-purpose register set, shown in Fig. 2.4, is identical to the Z8002, with the exception of the implied stack pointer. Any general-

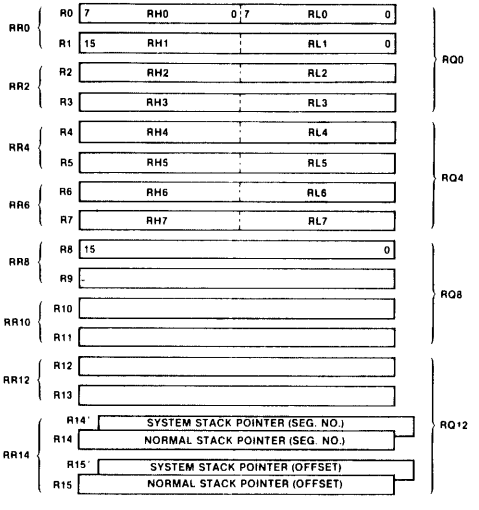


Figure 2.4 Z8001 general-purpose registers.

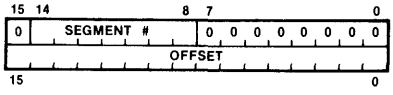


Figure 2.5 Format of segmented addresses stored in registers.

purpose register can be used as an accumulator. Any word register, except R0, can hold a nonsegmented memory address, an I/O address, or an index. When in segmented mode, two words are needed to store an address; segmented addresses can be held in any long-word register except RR0. The segment number is held in bits 8–14 of the even-numbered register, and the offset goes in the odd-numbered register (Fig. 2.5).

The implied stack pointer in the Z8001 is register pair RR14, where R14 holds the stack pointer's segment number and R15 holds the offset. As with the Z8002, there are actually two copies of the implied stack pointer, one for system-mode operation and one for normal-mode operation. RR14 is used as the implied stack pointer in the segmented mode, and R15 is the implied stack pointer in the nonsegmented mode.

Z8001 CONTROL REGISTERS

The Z8001's control registers include the program status registers, PSAP, and refresh control register (Fig. 2.6).

A reserved word, the FCW, and the PC define the Z8001's program status registers. The reserved word is not used in the Z8001 but is reserved for use in future, upward-compatible Z8000 family processors. The program counter is two words long, where one word holds the segment number and the other word holds the offset. The flag and control word is identical to the Z8002's FCW, with the exception of bit 15. This bit is the segmentation-mode bit; when set to 1, the CPU operates in the segmented mode, and when cleared to 0, the CPU operates in the nonsegmented mode. In the segmented mode, programs manipulate 23-bit segmented addresses; in the nonsegmented mode, programs manipulate 16-bit nonsegmented addresses. Thus Z8002 programs can be run on the Z8001 in the nonsegmented mode. However, the Z8001 always outputs segmented addresses during memory accesses, regardless of the operating mode. When a memory access is made in the nonsegmented mode, the offset is the 16-bit nonsegmented address generated by the program, and the segment number is the value of the segment number field of the program counter. In other words, the Z8001 in the nonsegmented mode is actually holding a fixed segment number on the SN0–SN6 outputs, and making all its accesses to that one segment. That segment will be the segment number that was in the program counter when the CPU was placed in the nonsegmented mode. The remaining bits of the Z8001's FCW function in the same manner as the corresponding bits in the Z8002's FCW, as described previously.

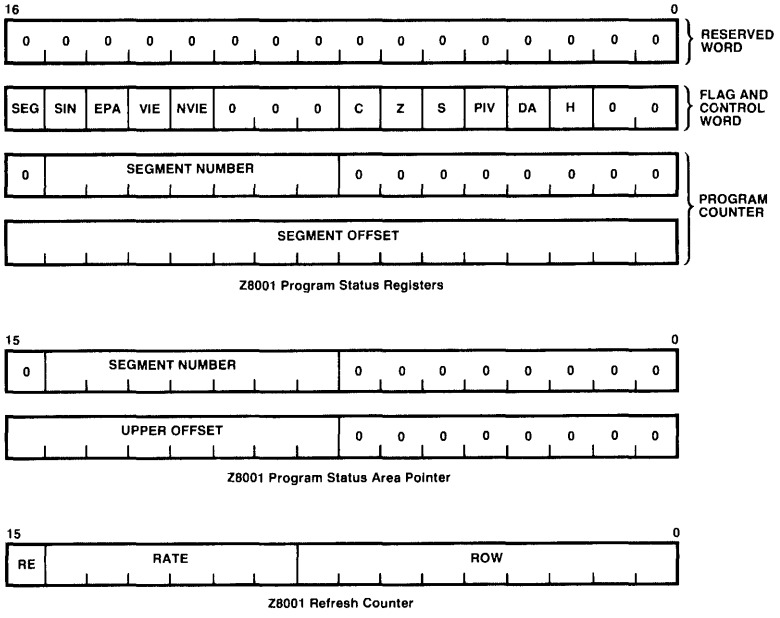


Figure 2.6 Z8001 control registers.

Besides determining which instructions can be executed, the contents of the S/$\overline{N}$ bit in the Z8001's FCW affect how the implied stack pointers are accessed. When in the normal mode (S/$\overline{N}$ = 0), a reference to R14, R15, or RR14 by an instruction will affect the normal-mode stack pointer. In the system mode (S/$\overline{N}$ = 1), a reference to R14, R15, or RR14 will access the system-mode stack pointer, unless the Z8001 is in the nonsegmented mode, in which case a reference to R14 will reference the normal-mode R14. In other words, when the Z8001 is in the segmented mode, RR14 is the implied stack pointer, and both system- and normal-mode copies of R14 and R15 can be accessed, depending on the current operating mode. In the nonseg-

TABLE 2.1 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 the nonsegmented mode.

mented mode, R15 is the implied stack pointer, and therefore only the normal-mode copy of R14 can be accessed. This operation is summarized in Table 2.1.

The PSAP in the Z8001 consists of two words that hold the segment number and offset address of the program status area (see Chapter 6). The low-order byte of the offset is always zero. The refresh control register has the same configuration in the Z8001 and Z8002, and is described in Chapter 3.

3

Interfacing to Memory

The Z8000 CPUs transfer data to and from memory asynchronously on the 16-bit multiplexed address/data bus. As discussed previously, the Z8001 and Z8002 have different memory addressing capabilities. The Z8002 generates 16-bit addresses and can directly address 64K bytes of memory per memory address space. The Z8001 generates 23-bit segmented addresses, consisting of a 7-bit segment number and a 16-bit offset, and can directly address 8M bytes of memory (128 segments of 64K bytes each) per memory address space.

MEMORY ADDRESS SPACES

Up to six different memory address spaces can be defined using the N/$\overline{S}$ and ST0–ST3 signals, as described in Chapter 1: system-mode program, system-mode data, system-mode stack, normal-mode program, normal-mode data, and normal-mode stack (Fig. 1.7). Thus, by using the status signals to define distinct memory address spaces, up to a sixfold increase in the addressing range can be realized.

In some applications, complete separation of these memory address spaces may not be desirable. Normal-mode programs often need to pass in-

formation to a system-mode routine. For example, a normal-mode program might ask the operating system to output a data file to a printer. The data to be output would have to accessible in both normal and system modes; a normal-mode program would initialize and manipulate the data, and a system-mode program would have to read the data in order to perform the requested output operation.

CPU registers could be used to pass data between normal-mode and system-mode routines. The normal-mode program would put the data in the registers before calling the operating system. The system-mode output routine would, then, only have to read the register contents. But since there are only 16 general-purpose registers, this scheme is feasible only for very small data files. Furthermore, other information in the registers, such as the value of the implied stack pointer, might have to be saved in memory and restored later.

A more general method would have the normal-mode program store the data file in memory and pass the starting address and length of the file to the operating system. These two parameters could be passed in registers or memory locations reserved for that purpose. However, the area in memory where the data file is stored would have to be accessible to both the system-mode and normal-mode programs. In other words, some block of memory addresses (perhaps a whole segment in a Z8001 system) would have to be in both the normal-mode data and system-mode data address spaces (Fig. 3.1).

Similarly, if programs are to be downloaded to the Z8000 system from some other computer, the program and data address spaces might need to be overlapped; the load commands used to download the code would write to the data memory address space, but execution of the downloaded code involves fetches from the program memory address space. Thus the memory area that holds the downloaded code must be in the data memory space during the download operation and in program memory space when the code is to be executed. Many systems may require data references to the informa-

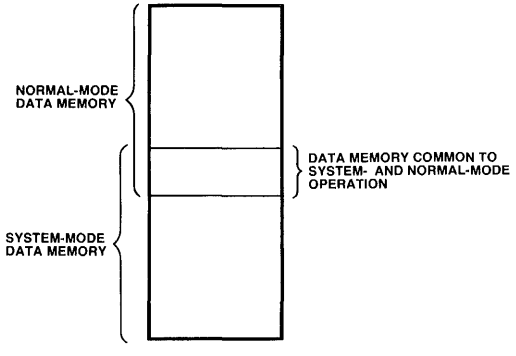


Figure 3.1 Memory shared by two address spaces.

tion placed on the stack during interrupt handling; therefore, the data and stack address spaces might share some common memory locations. Thus most Z8000 systems will have some memory locations that are shared among the six possible memory address spaces. (In Z8001 systems with a programmable memory manager, such as the Z8010 MMU, this problem is not as critical, since the attributes of a particular memory segment can be changed during program execution by reprogramming the memory manager.)

Few systems will, in practice, require the full addressing range of the CPU for all six possible memory address spaces. Few Z8001 applications, for example, need both 8M bytes of system-mode stack and 8M bytes of normal-mode stack.

MEMORY ORGANIZATION

Each memory address space in the Z8002 and each memory segment in the Z8001 is a string of up to 64K bytes numbered consecutively in ascending order. The byte is the basic addressable memory element in Z8000 systems, and, therefore, each address emitted by a Z8000 CPU designates a particular byte of memory. However, three other types of data elements in memory are addressable: bits, words, and long words. The type of data element addressed depends on the instruction being executed. Certain instructions act on specific bits in memory by specifying a byte or word address and the number of the bit within the byte or word. Most of the instructions for performing arithmetic and logical operations on data have byte, word, and long-word formats. A few special instructions act on strings of bytes or words, where the length of the string is given in a register; however, these instructions access the bytes or words one at a time.

Although memory is addressed as bytes, the Z8000 architecture defines a 16-bit-wide data path (the 16-bit address/data bus), and memory is organized in the same way, as 16-bit words. Each 16-bit word of memory is made up of two 8-bit bytes; thus each Z8002 memory address space and Z8001 memory segment actually contains up to 32K words of memory, where both bytes in a word are accessible. A memory segment consists of two banks of

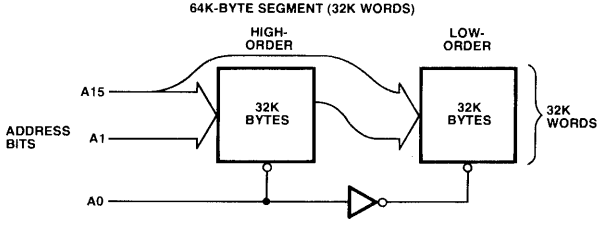


Figure 3.2 Organization of a memory segment.

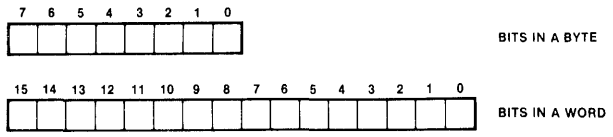


Figure 3.3. Numbering of bits in bytes and words of memory.

up to 32K bytes each, as shown in Fig. 3.2; one bank holds the bytes with even-numbered addresses, and the other bank holds the bytes with odd-numbered addresses. A word always consists of an even-addressed byte and the next consecutive odd-addressed byte. Therefore, only the upper 15 bits of the address (the offset in segmented addresses), A1–A15, are needed to specify a word in memory. However, each byte in a word can be addressed separately; the least significant bit of the address, A0, is used to specify a byte within a word during byte accesses. All Z8000 instruction fetches and stack references are word accesses, but data references can be byte or word accesses.

Bits in bytes and words are numbered right to left in the traditional fashion; that is, the rightmost bit is the least significant bit and is labeled bit 0 (Fig. 3.3). Thus bit position n corresponds to the value 2^n in the representation of positive binary numbers.

The two bytes in a word, however, are numbered left to right; the most significant byte in the word is the even-addressed byte (A0 = 0), and the least significant byte is the odd-addressed byte (A0 = 1), as shown in Fig. 3.4. The address of a word is the address of its most significant byte; therefore, words always have even addresses (A0 = 0) (Fig. 3.5). For example, the word at memory address 6000 consists of the byte with address 6000 (the most significant half of the word) and the byte at address 6001 (the least significant half of the word). This convention is the opposite of many machines, such as the PDP-11 minicomputers, and may seem confusing at first, since the most significant byte of a word has an address that is numer-

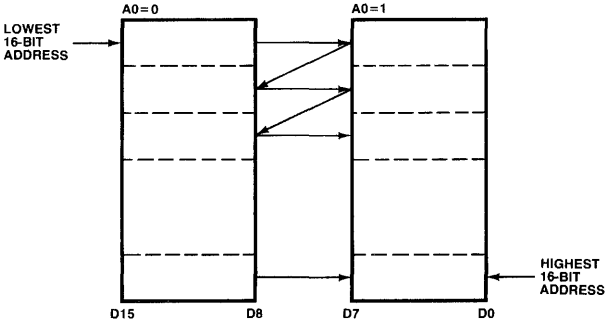


Figure 3.4 Sequencing of bytes in memory.

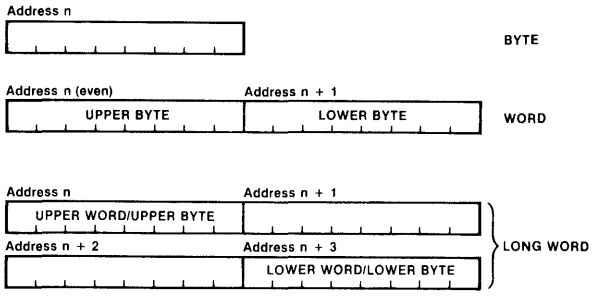


Figure 3.5 Addressing of data elements in memory.

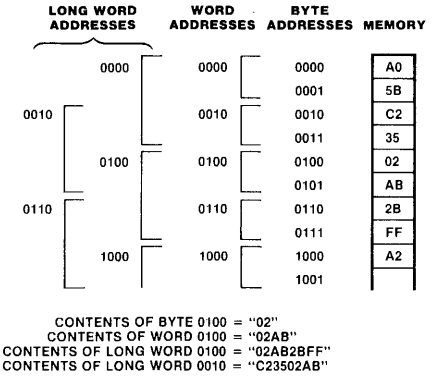


Figure 3.6 Memory addressing example.

ically one smaller than the least significant byte of that word. However, this type of memory organization greatly simplifies the manipulation of strings of byte data. Strings of bytes can be stored in consecutive memory locations and addressed in consecutive numeric order (Fig. 3.4).

Some Z8000 instructions allow manipulation of long-word (32-bit) data elements. Just as with words, the address of a long-word memory operand is the address of its most significant byte (Fig. 3.5), which is also the byte in the long word with the smallest numerical address. Words and long words in memory always start on even addresses. An example of byte, word, and long-word addressing is illustrated in Fig. 3.6.

MEMORY CONTROL LOGIC

Memory control logic is the logic needed to interface particular semiconductor memory devices to the Z-Bus signals output by Z8000 CPUs. This logic typically includes multiplexers and demultiplexers for interfacing to the address/data bus, and chip-select and timing logic for controlling the memory chips. A block diagram of the memory control logic for one memory segment is illustrated in Fig. 3.7. Timing of the Z-Bus transactions between the

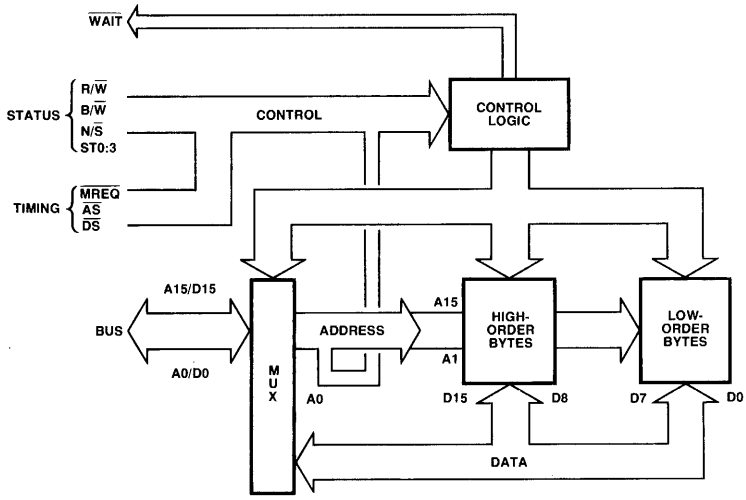


Figure 3.7 Typical Z8000 memory interface block diagram.

memory and the CPU is determined by the address and data strobes ($\overline{AS}$ and $\overline{DS}$). The memory control logic might generate $\overline{WAIT}$ signals if memory access times are longer than the default CPU memory cycle timing (see Chapter 5). The status lines (N/$\overline{S}$, B/$\overline{W}$, R/$\overline{W}$, ST0–ST3) and, in Z8001 systems, the segment number, are used to generate chip selects to the memory, since they define the address space, segment, and type of memory access. (The $\overline{MREQ}$ signal might be used in place of the ST0–ST3 lines to signal a memory access in systems where separate program, data, and stack memory address spaces are not distinguished.) Thus this group of signals, together with the address/data bus, defines the Z-Bus signals used for CPU–memory transfers (Fig. 3.8). Of course, not every system will use every one of these signals; for example, a system designed for word accesses only might not use the B/$\overline{W}$ CPU output.

The start of a Z-Bus transaction is signaled by $\overline{AS}$ going low (Fig. 3.9). The status signals and the address on the address/data bus are guaranteed to be valid at the rising edge of $\overline{AS}$; this edge can be used to latch the address and status information. (The address can also be latched by the falling edge of $\overline{MREQ}$, which happens slightly sooner.)

During memory reads, $\overline{DS}$ going low indicates that the CPU has reversed the direction of the address/data bus, and the data to be read from memory can be placed on the bus. Thus $\overline{DS}$ would be used to enable the buffers that drive the data from memory onto the bus. For memory writes, $\overline{DS}$ goes low after the address is replaced on the bus by the data to be written; $\overline{DS}$ would then be used to enable the write to memory. The memory control logic can lengthen the data transfer timing by pulling the $\overline{WAIT}$ line low, thereby

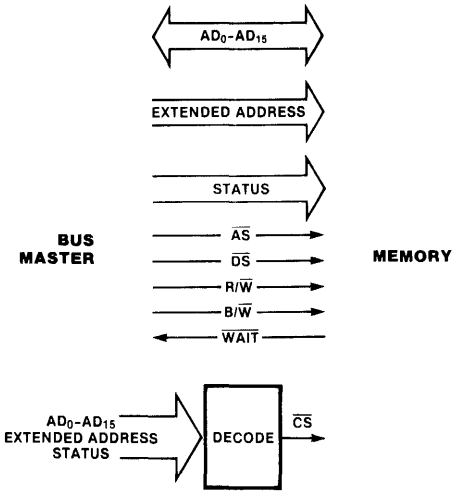


Figure 3.8 Z-Bus signals for CPU-memory transactions.

allowing the Z8000 CPUs to interface with memories with arbitrarily long access times.

The status lines are used to generate memory chip enables and help control the bus buffering. The ST0–ST3 lines, decoded to provide instruction fetch, data access, or stack access status signals, can be used in conjunction with the N/$\overline{S}$ signal to select a particular memory address space. The R/$\overline{W}$ and B/$\overline{W}$ lines are used with the $\overline{DS}$ signal to drive the data buffers and enable writes to memory.

Address bits A1–A15 are propagated to address-decode circuitry to access a particular word in the specified memory address space and/or segment. Address bit A0 is used by the control logic to select a particular byte in the word addressed by A1–A15 during byte accesses, as described below.

During memory reads, the R/$\overline{W}$ line will be high. Regardless of the state of the B/$\overline{W}$ line, the memory control logic should place a word of data on the bus during $\overline{DS}$ active. The data should be the contents of the word of memory specified by address bits A1–A15 latched during $\overline{AS}$. If the B/$\overline{W}$ line is low, indicating a word transfer, the CPU will, of course, read the entire word of data. If the B/$\overline{W}$ line is high, indicating a byte transfer, logic internal to the CPU will select one-half of the word of data returned on the bus as the byte to be read; if A0 was 0 during the address strobe, the CPU will read the upper half (D8–D15) of the bus, and if A0 was 1, the CPU will read the lower half (D0–D7) of the bus. Thus, for memory reads, the memory control logic can ignore the state of the B/$\overline{W}$ line and A0, and just return the contents of the word addressed by A1–A15 (Fig. 3.10); during byte reads, the CPU will internally select the appropriate half of the word.

For memory writes, the R/$\overline{W}$ line is low. If the B/$\overline{W}$ is also low, sig-

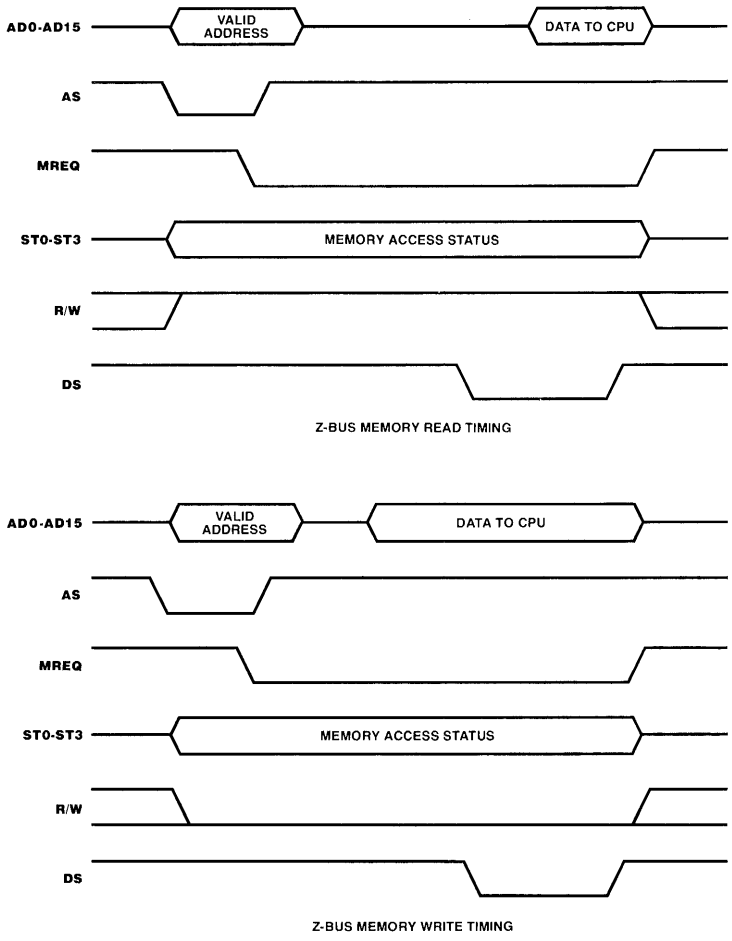


Figure 3.9 Z-Bus memory transaction timing.

nifying a word transfer, the memory control logic can ignore A0. The word of data supplied by the CPU during $\overline{DS}$ active is written into the memory word addressed by A1–A15 (Fig. 3.11). If the B/$\overline{W}$ line is high, signifying a byte transfer, the CPU will duplicate the byte of data on both halves of the address/data bus during $\overline{DS}$ active. The memory control logic should examine the value of A0 to decide which bank of bytes is to be written; A1–A15 address a word, and the high-order byte of that word receives the written data if A0 = 0, and the low-order byte of that word receives the data if A0 = 1 (Fig. 3.12). Therefore, the only time the memory control logic needs to examine A0 and select just one byte of the word addressed by A1–A15 is

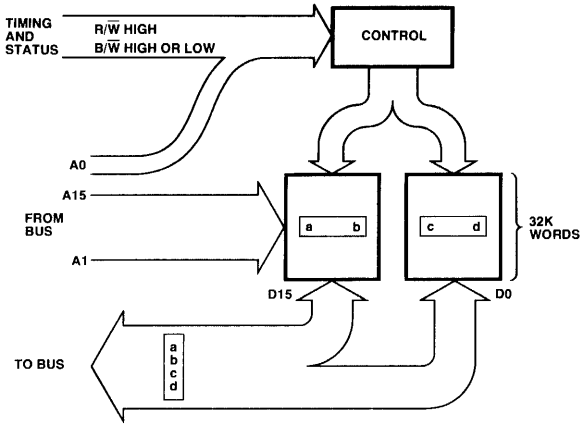


Figure 3.10 Reading from memory.

when a byte write operation is performed. (The replication of byte data on both halves of the bus during byte write operations may not be implemented on future Z8000 family CPUs; therefore, system designs should not depend on this feature.)

This type of memory organization has several important ramifications. The ability to write to specific bytes in a word avoids time-consuming read-modify-write cycles, wherein a CPU would have to read a whole word of data from memory, internally modify one byte of the word, and then write the word back into memory. Words of data must be aligned on word boundaries (that is, on even addresses). Word accesses always ignore the value of address bit 0; therefore, it is impossible to access a "word" consisting of a byte with an odd-numbered address and the next consecutive even-addressed byte. Words and long words always have even addresses, and the even-numbered byte of a word is always the most significant byte. Instructions are always words and should be aligned at word boundaries. As a result, the

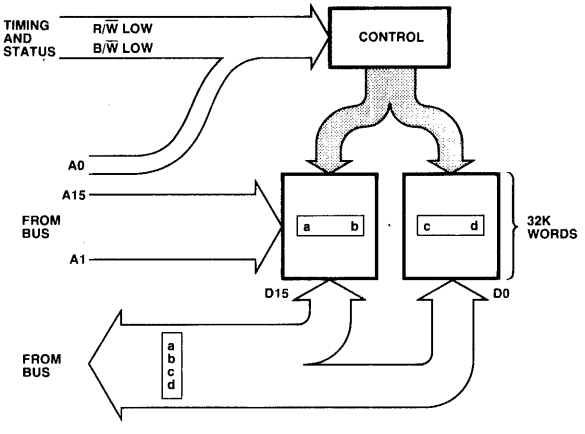


Figure 3.11 Writing words to memory.

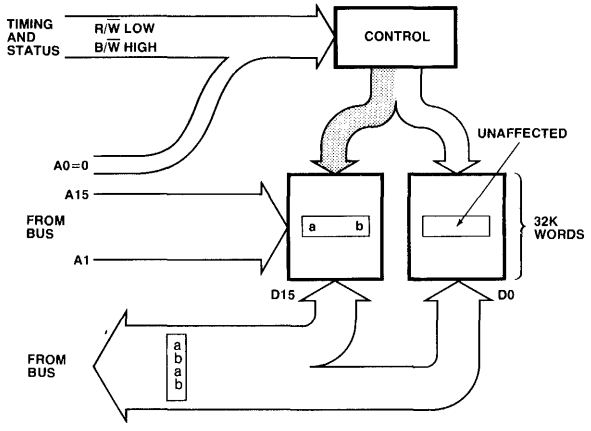


Figure 3.12 Writing bytes to memory.

program counter should always hold an even value. The implied system- and normal-mode stack pointers should also have even-valued contents.

The timing of the address strobe ($\overline{AS}$) in relation to the address and status signals at the memory control logic should be examined carefully. The status and address lines are guaranteed to be stable CPU outputs before the rising edge of $\overline{AS}$. However, these signals may pass through buffers and decoders before reaching the memory control logic. If $\overline{AS}$ is used to latch these signals at the memory controller, the maximum delay time for the status and address signals to pass through any intermediate decoding or buffering logic must be taken into account, and the $\overline{AS}$ to the memory control logic may need to be delayed accordingly. (Appendix A contains timing information for the Z8000 CPUs.)

EXAMPLE 1: INTERFACING TO Z6132'S

Rather simple memory control logic is needed in Z8000 systems when using Z-Bus-compatible memories such as the Zilog Z6132 Quasi-Static RAM. The Z6132 is a 4K × 8 dynamic RAM with on-board memory refresh capability. A functional pin diagram is shown in Fig. 3.13. Inputs include 12 bits of address, for addressing the 4096 bytes of memory on the chip, an address clock (AC), a data strobe ($\overline{DS}$), a write enable ($\overline{WE}$), and a chip select ($\overline{CS}$). Data lines D0–D7 serve as data inputs during memory writes and data outputs during memory reads. The $\overline{BUSY}$ signal is an input or output, depending on the refresh mode chosen.

A read or write operation to a Z6132 memory chip starts on the rising edge of the address clock (AC) input, at which point the chip select ($\overline{CS}$), write enable ($\overline{WE}$), and address inputs A0–A11 are examined (Fig. 3.14). If

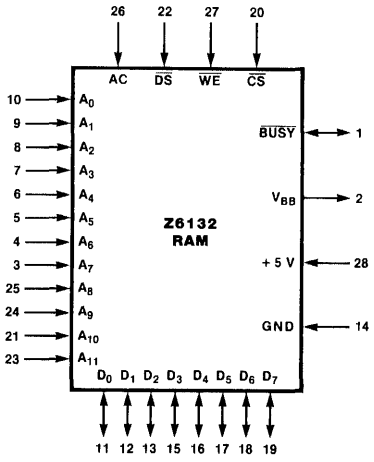


Figure 3.13 Z6132 RAM functional pinout.

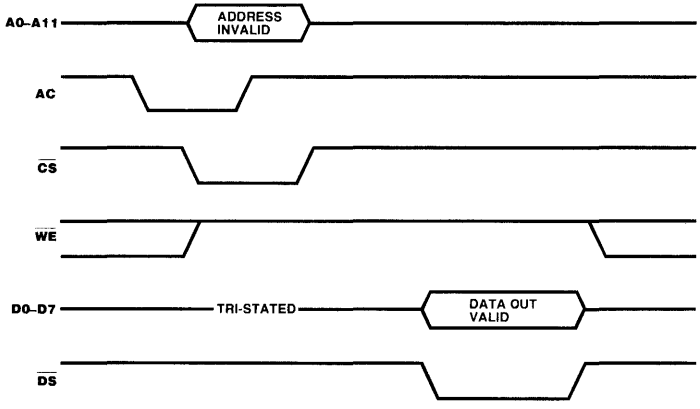


Figure 3.14 Z6132 timing.

the chip is not selected ($\overline{CS}$ is high), all other inputs are ignored for the rest of that cycle, that is, until the next rising edge of AC. For a memory read ($\overline{WE}$ high), a low on the $\overline{DS}$ will activate the data outputs D0-D7; in other words, the $\overline{DS}$ is a data output enable signal. For memory writes ($\overline{WE}$ low), the falling edge of $\overline{DS}$ latches the data on the D0-D7 inputs into the addressed memory location. Note that the Z6132 timing is very similar to the Z-bus memory timing shown in Fig. 3.9. (Further details of Z6132 operation are discussed in Chapter 12.)

Consider a Z8002-based system with 4K words of memory, where the six memory address spaces are not differentiated; in other words, the same 4K words of memory are accessed by the CPU for system-mode and normal-mode program, data, and stack references. Thus the $\overline{MREQ}$ signal can be used in lieu of the ST0-ST3 status lines to indicate a memory access. The 4K words of memory can be organized as two banks of 4K bytes each, where one Z6132 chip holds the even-addressed bytes, and a second Z6132 holds the odd-addressed bytes (Fig. 3.15). During word and byte read transactions, both banks are accessed simultaneously, and the word on the address/data bus consists of one byte from each bank. For byte writes, the banks are accessed separately.

A connection diagram for this memory interface is shown in Fig. 3.16. Ignoring detailed timing considerations for now, the $\overline{AS}$ CPU output would connect directly to the Z6132's AC input; the $\overline{DS}$ CPU output connects to the memory's $\overline{DS}$ input; the R/$\overline{W}$ CPU output connects to the memory's $\overline{WE}$ input. The $\overline{BUSY}$ outputs from memory are tied to the CPU's $\overline{WAIT}$ input.

Addresses are latched internally in the Z6132 by the rising edge of AC, and the Z6132's data lines are tri-stated except when $\overline{DS}$ is active. As a result, the address/data bus does not need to be demultiplexed or latched external to the memory chips. The Z6132's address inputs A0-A11 are used to select one of 4096 bytes in the memory chip. Since the address bit on

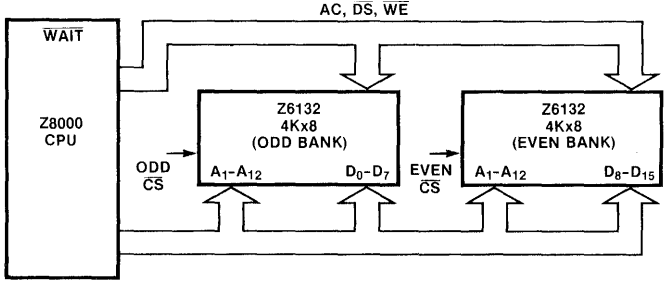


Figure 3.15 Block diagram of a Z8000-Z6132 interface for a 4K word system.

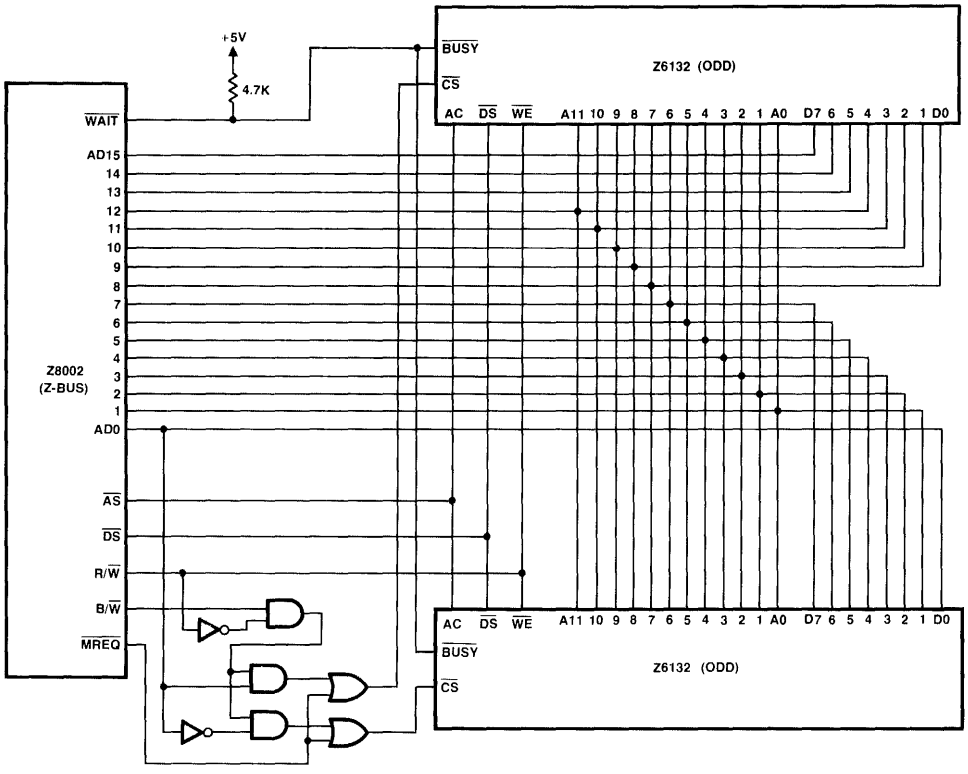


Figure 3.16 Z8002-Z6132 interface logic for a 4K word system.

bus line AD0 is used to address bytes in a word, not to address the word itself, this line is not used as an address input to the Z6132's. Thus AD1 supplies the A0 memory address input, AD2 is connected to the memory's A1, and so on. (Since only 4K words are being addressed, the address bits that appear on bus signals AD13, AD14, and AD15 are not used.) The Z6132 that holds the even-addressed bytes has its data lines connected to the upper half of the CPU's address/data bus, since the even-addressed byte is the most significant byte in a word. The Z6132 with the odd-addressed bytes has its data lines connected to the lower half of the address/data bus.

The chip-select signals to memory are generated from the R/$\overline{W}$, B/$\overline{W}$, and $\overline{MREQ}$ status lines and the address bit on AD0. Chip selects are latched internally in the Z6132 by the rising edge of AC. Assuming that $\overline{MREQ}$ is active, both Z6132's are to be chip selected if the B/$\overline{W}$ line is low, indicating a word access, or if the R/$\overline{W}$ line is high, indicating a memory read. For byte writes, the Z6132 holding the even-addressed bytes is chip selected if AD0 is a 0 during $\overline{AS}$ active, and the Z6132 holding the odd-addressed bank of bytes is chip selected if AD0 is a 1 during $\overline{AS}$ active.

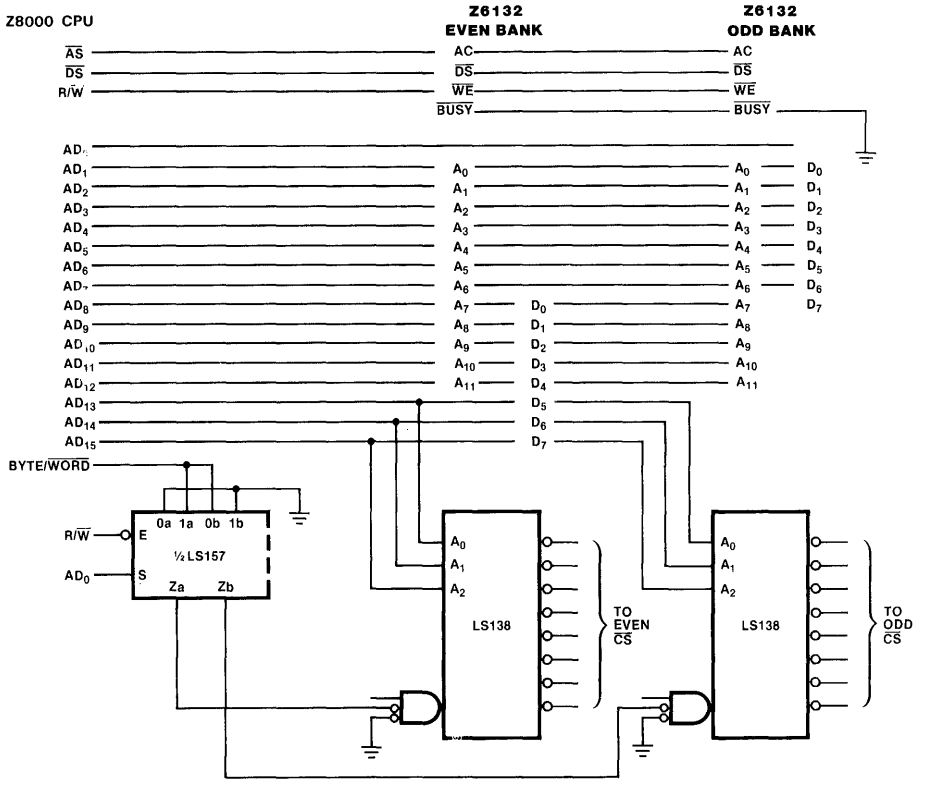


Figure 3.17 Block diagram of a Z8000-Z6132 interface for a 32K word system.

This design can be easily expanded to include the full 64K bytes addressable by the Z8002. The address bits output by the CPU on AD13, AD14, and AD15 are decoded to generate chip selects to eight pairs of Z6132 memories, where each pair of memory devices holds 4K words. In Fig. 3.17, two 74LS138 3-to-8 decoders are used to decode the upper three address bits to generate the chip-select signals to the memories. A 74LS157 2-to-1 multiplexer replaces the combinational logic of the previous example, and it enables the 74LS138's to activate the chip selects for the odd bank of bytes, even bank of bytes, or both, as determined by the B/$\overline{W}$, R/$\overline{W}$, and AD0 signals. Not shown in Fig. 3.16, but still necessary, is the $\overline{MREQ}$ signal, which would also be used as an enable signal to the 74LS138 decoders.

If separate memory address spaces are required, the N/$\overline{S}$ and STO-ST3 status signals must also be decoded and used as part of the logic that generates the chip selects. Similarly, the segment number in a Z8001 system would be decoded as part of the chip-select logic to select only the appropriate memory devices within the particular memory segment being accessed.

If the access time of the memory being used is longer than the normal interval from the rising edge of $\overline{AS}$ to the rising edge of $\overline{DS}$, $\overline{WAIT}$ signals to the CPU may need to be generated. (Wait-state generation is discussed in Chapter 5.)

In both of the examples above, signals from the Z8000 CPU are directly connected to the appropriate memory and logic devices. In actual systems, buffering is usually required for CPU outputs, since the CPU can drive only one standard TTL load. (CPU bus and signal buffering is discussed in Chapter 11.)

EXAMPLE 2: INTERFACING TO Z6132's AND 2716'S

Of course, memory devices other than Z-Bus-compatible Z6132's can be interfaced to the Z-Bus. For example, Fig. 3.18 is a block diagram for a Z8002-based system with 2K words of read-only program memory and 4K

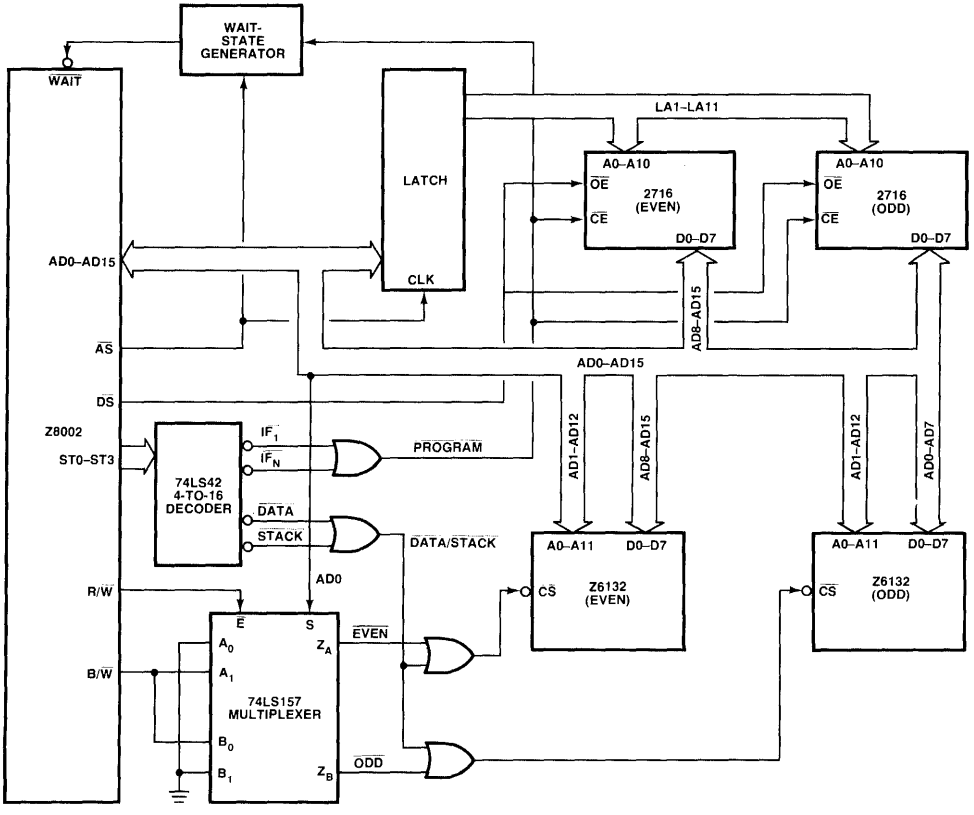


Figure 3.18 Z8002 interface to 2716's and Z6132's.

words of random access data/stack memory. The program memory address space consists of two 2K × 8 2716 erasable programmable read-only memories (EPROMs). The data and stack memory space is shared by two Z6132 4K × 8 RAMs.

Since program memory accesses are differentiated from data and stack accesses in this example, the ST0–ST3 lines are used as part of the chip-select logic. A 74LS42 4-to-16 decoder is used to decode the status lines. IF1 or IFn status (that is, any instruction fetch) will activate the chip enable ($\overline{CE}$) inputs of the 2716 EPROMs; since accesses to EPROMs are always reads, both byte banks of EPROMs are activated each time program memory is accessed. (Remember, the memory control logic has to select a byte within a word only for byte writes.) Since 2716's require that the address stay valid at the memory's address inputs throughout the memory access, the addresses output by the CPU are latched by the rising edge of $\overline{AS}$. (74LS373 octal latches or a similar part could be used.) The latched address lines LA1–LA11 are propagated to the address inputs of the 2716's. LA0 is not used, since memory read operations always read the whole word onto the bus. The output enable ($\overline{OE}$) for the EPROMs is connected to $\overline{DS}$, allowing the addressed data to be placed on the bus when $\overline{DS}$ is active. Since the 2716's tri-state their data outputs when $\overline{OE}$ is inactive, the data outputs can be tied directly to the address/data bus. The 2716 that holds the high-order bytes for each word has its data lines connected to the upper half of the bus (AD8–AD15), and the 2716 that holds the low-order bytes of each word has its data lines connected to AD0–AD7. The relatively slow access times of the 2716 will probably require the generation of a $\overline{WAIT}$ signal to the CPU to slow down memory accesses. This $\overline{WAIT}$ signal would be generated for each program memory access.

The interface to the Z6132's is similar to that of the previous examples, except that the chip-select logic is enabled by data and stack memory accesses instead of $\overline{MREQ}$. All reads and word writes will cause both byte banks to be selected, whereas byte writes cause either the even-addressed bank of bytes or the odd-addressed bank to be selected, depending on AD0. For clarity's sake, the $\overline{AS}$, $\overline{DS}$, R/$\overline{W}$, and $\overline{WAIT}$ connections between the Z8002 and the Z6132's are not shown in Fig. 3.18. The $\overline{BUSY}$ output of the Z6132's could be ORed with the output of the wait-state generator to produce the $\overline{WAIT}$ signal back to the CPU. Again, buffering of the signals from the CPU is necessary but is not shown here.

A memory interface example for a system using dynamic RAMs is described in Chapter 11.

MEMORY REFRESH

The Z80 CPU was the first microprocessor that included logic for automatically refreshing dynamic memories. In Z80 systems, a refresh is performed

after each instruction fetch cycle. The Z8000 CPUs also have automatic memory refresh capability. The Z8000 refresh function is more flexible than the Z80's; the refresh operation can be enabled or disabled and the refresh rate can be set under program control.

The refresh function in the Z8000 CPUs is controlled by one of the CPU control registers, the refresh register. The refresh register is identical in format in the Z8001 and the Z8002; it contains an enable bit, a 6-bit rate counter, and a 9-bit row address counter (Fig. 3.19). The enable bit, bit 15, is set to enable automatic refresh and reset to disable refresh. The rate counter, bits 9 through 14, determines the time between successive refreshes. When a refresh cycle occurs, the row address in the row counter, bits 0–8, is output on the address/data bus.

The rate counter is a programmable 6-bit prescaler which can be loaded with a value from 0 to 63. This counter is decremented every four CPU clock cycles (every 1 μs at 4 MHz). A refresh cycle is performed as soon as possible after the transition from a count of 1 to a count of 0. The original starting count is automatically reloaded after the counter reaches 0. Loading a starting count of zero provides the longest possible period between refresh cycles: 64 $\times$ 4 $\times$ the clock period.

During a refresh cycle, the row counter in the refresh register is output on AD0–AD8, and the ST3–ST0 status lines are set to 0001. Since memory in Z8000 systems is word organized, and AD0 is used only to distinguish bytes within a word, AD0 is not considered part of the memory's row address, and is always 0 in the refresh register. The row counter is incremented by two after each refresh; thus the row counter cycles through 256 distinct addresses on the AD1–AD8 lines, providing refresh for up to 256 rows of dynamic memory. (Most 16K dynamic RAMs are organized with 128 rows to be refreshed. Some 64K RAMs have 256 rows to be refreshed.)

A memory refresh cycle occurs as soon as possible after the rate count goes to zero. The refresh cycle takes three CPU clock periods; refresh timing will be analyzed in Chapter 5. Of course, the CPU cannot issue memory refreshes when it does not have control of the bus due to a bus request from another device, such as a DMA controller. However, internal circuitry in the CPU will record up to two missed refresh cycles and issue those refreshes immediately after regaining control of the bus.

The refresh register can be read from or written to using the Load Control (LDCTL) instruction when the CPU is in the system mode. Resetting the CPU causes bit 15 of the refresh register to be reset, disabling the automatic refresh function. Automatic memory refresh is begun by loading a rate that guarantees the proper interval between refreshes into the rate

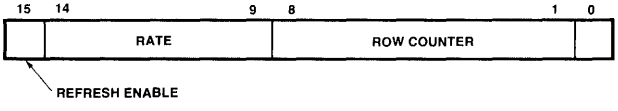


Figure 3.19 Refresh register.

LOWER NIBBLE OF UPPER BYTE

UPPER NIBBLE OF UPPER BYTE	0 1	2 3	4 5	6 7	8 9	A B	C D	E F
0								
1								
2								
3								
4			NO REFRESH					
5								
6								
7								
8	256	4	8	12	16	20	24	28
9	32	36	40	44	48	52	56	60
A	64	68	72	76	80	84	88	92
B	96	100	104	108	112	116	120	124
C	128	132	136	140	144	148	152	156
D	160	164	168	172	176	180	184	188
E	192	196	200	204	208	212	216	220
F	224	228	232	236	240	244	248	252

Figure 3.20 Refresh period in clock cycles.

counter and setting bit 15. Figure 3.20 shows the relationship between the value loaded into the upper byte of the refresh register and the number of clock cycles between each refresh. For example, loading a 9E00 into the refresh register means that a refresh will be generated every 60 clock cycles (every 15 μs in a 4-MHz system, which is adequate to satisfy the worst-case refresh requirements of typical 16K RAMs).

When reading the refresh register using the LDCTL instruction, only the row counter portion (bits 0–8) of the register can be read. Thus the next row address to be refreshed can be read anytime while in the system mode.

4

Interfacing to Peripherals

The Z8000 CPUs interface to I/O devices in the same manner as memory: asynchronous data transfers on the 16-bit multiplexed address/data bus. I/O addresses emitted by the CPU on the bus are always 16 bits long. (The segment number output by the Z8001 applies only to memory addressing, and is not used for I/O addressing.) Thus the Z8000 can address up to 65,536 I/O devices per I/O address space. Each peripheral device can be byte or word organized; both byte and word I/O instructions are included in the instruction set.

I/O ADDRESS SPACES

There are two I/O address spaces in Z8000 systems: standard I/O and special I/O (Fig. 1.6). These I/O address spaces are distinct from the memory address spaces; they can be accessed only by the execution of specific I/O instructions. All I/O instructions are privileged instructions and therefore can be executed in the system mode only. Thus, in an operating system environment, only the operating system software can access peripheral devices. If an applications program running in the normal mode attempts an I/O instruction, a trap will occur (see Chapter 6). Therefore, the separation of the I/O and memory address spaces and the inclusion of specific I/O instructions lead to a built-in protection mechanism for operating system applications. Memory-mapped I/O schemes are possible in Z8000 systems, but since instructions that act on memory can then cause I/O data transfers, it would be

possible for normal-mode users to access I/O devices directly. Implementation of a protected operating system would be difficult, if not impossible. The separation of the memory and I/O address spaces also conserves memory space. With memory-mapped I/O, memory addressing capability is sacrificed, since the I/O device addresses are mapped into the memory address space.

The standard I/O and special I/O address spaces are distinguished by decoding the ST0–ST3 status signals, as described in Chapter 1. Up to 65,536 peripheral devices can be addressed in each of the address spaces. Each of the address spaces is accessed through a separate set of I/O instructions; that is, there is a set of assembly language instructions for standard I/O operations and another distinct set of instructions for special I/O operations. The only difference between the execution of a standard I/O instruction and the corresponding special I/O instruction is the code on the ST0–ST3 status lines during the data transfer to the I/O device. As a convention, standard I/O is used for peripheral devices, and special I/O is reserved for use with programmable CPU support devices, such as the Z8010 Memory Management Unit (MMU). This is just a convention, however, not a requirement of the CPU architecture. (Systems that include Z8010 MMUs would use special I/O instructions to program the MMUs. Systems without MMUs can use both the standard I/O and special I/O address spaces for peripheral devices.)

I/O INTERFACE SIGNALS

The set of Z-Bus signals used to interface a Z8000 CPU to a peripheral device is similar to the signals used for interfacing to memory. Data transfers occur on the 16-bit multiplexed address/data bus. The chip-select signal to a peripheral is decoded from the 16-bit address that appears on the bus at the start of an I/O transaction and from the ST0–ST3 status lines, which signal when an I/O transfer is occurring. Timing of data transfers between the CPU and peripherals is controlled by the address and data strobes ($\overline{AS}$ and $\overline{DS}$). The peripheral might return an active $\overline{WAIT}$ signal during data transfers to extend the data transfer timing. The R/$\overline{W}$ and B/$\overline{W}$ signals define the direction and size of the data transfer. Thus these signals, together with the $\overline{RESET}$ line, define the Z-Bus interface to peripheral devices (Fig. 4.1).

The Z8000 CPU accesses peripherals in a manner similar to memory accesses. The start of a CPU-peripheral data transfer is signaled by $\overline{AS}$ going low (Fig. 4.2). The status signals (which should indicate standard or special I/O status) and the peripheral address on the address/data bus are guaranteed to be valid at the rising edge of $\overline{AS}$; this edge can be used to latch these signals, if necessary. The status and address signals would be decoded to generate the chip-select signals to the system's peripheral devices.

During I/O read operations, $\overline{DS}$ going low indicates that the CPU has

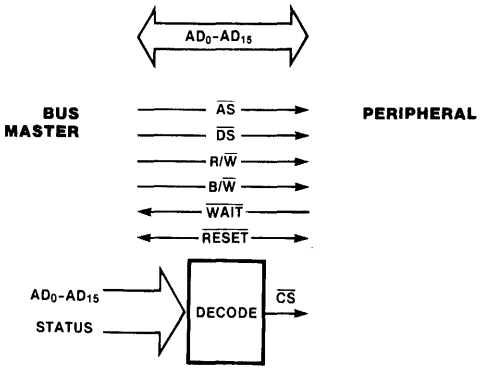


Figure 4.1 Z-Bus signals for CPU-peripheral data transfers.

reversed the direction of the address/data bus, and the data to be read from the addressed peripheral can be placed on the bus. Thus $\overline{DS}$ can be used to enable the buffers that drive data from the peripheral onto the bus. For I/O writes, $\overline{DS}$ goes low after the CPU replaces the address on the bus with the

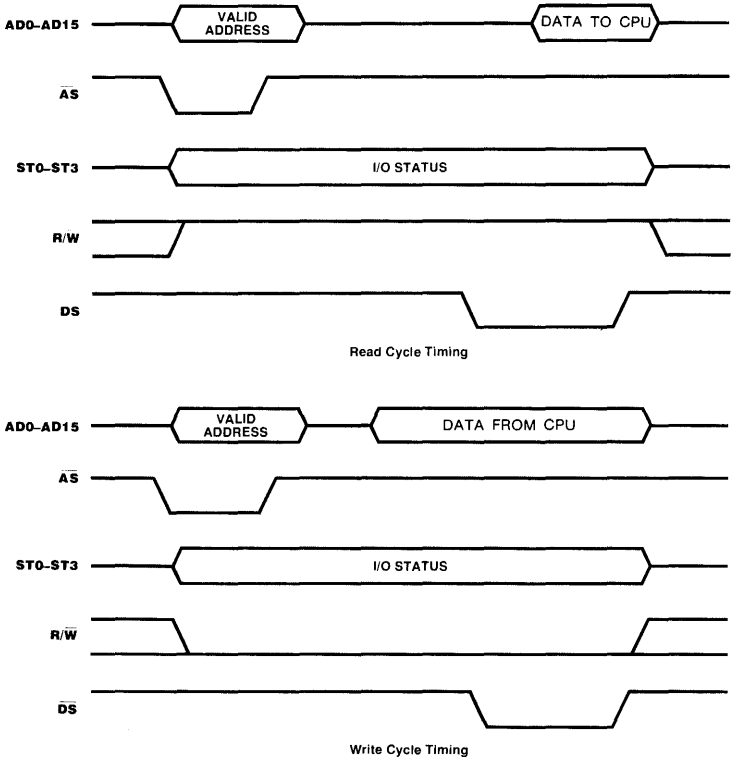


Figure 4.2 CPU-peripheral data transfer timing.

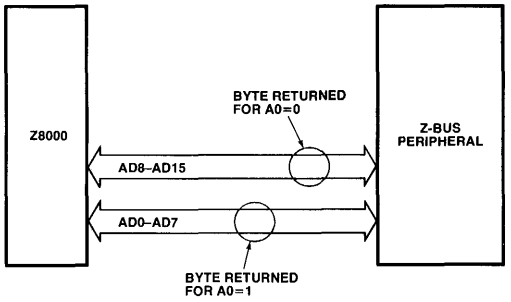


Figure 4.3 CPU-peripheral byte data transfers.

data to be written; $\overline{DS}$ is used to enable the write to the peripheral. The I/O device can lengthen the data transfer timing by asserting an active $\overline{WAIT}$ signal, thereby allowing the Z8000 CPUs to interface to I/O devices with arbitrarily long access times. The timing of data transfers is determined entirely by the $\overline{AS}$, $\overline{DS}$, and $\overline{WAIT}$ signals; the CPU clock is not part of the Z-Bus interface.

Data transactions in a Z8000 system can be 8-bit or 16-bit transfers, allowing the use of both byte-oriented and word-oriented peripherals. The Z8000 handles byte I/O transactions in much the same manner as byte memory operations. For byte write I/O operations, the CPU outputs the byte of data on both halves of the address/data bus during $\overline{DS}$, and the byte peripheral can read the data from either half of the bus. For byte read I/O operations, the CPU will read the whole word returned on the bus, and internally select one byte of that word. If A0 (port address bit 0) was a 0 during $\overline{AS}$ active, the CPU will select the upper half of the bus as the byte to be read; if A0 was a 1, the CPU will read the byte from the lower half of the bus (Fig. 4.3). Thus byte peripherals must use the appropriate half of the bus for data transfers—the upper half of the bus if the peripheral has an even port address, and the lower half of the bus if the peripheral has an odd port address. For word I/O operations, the CPU will read or write a whole 16-bit word regardless of the port address, and, therefore, word peripherals can have an even or an odd address.

I/O CONTROL LOGIC

I/O control logic is the logic needed to interface a particular I/O device to the Z-Bus signals from the CPU. Z8000 family peripherals, such as the Z8036 Counter/Timer and Parallel I/O Unit, are Z-Bus compatible and need very little interface logic. These devices connect directly to the Z-Bus, and the only external logic needed is the logic for decoding the address and status lines to generate chip selects (see Chapter 12).

Other peripherals besides Z8000 family devices can be used with a Z8000 CPU. In most cases, interface logic is needed to multiplex, demulti-

plex, and drive the address/data bus, to generate chip selects from the address and status signals, and to generate the appropriate timing signals for that peripheral from the address and data strobes.

For example, suppose that a Z8000 system has an byte I/O device consisting of eight lamps and eight switches. This peripheral is assigned I/O port address 0003. When reading port 0003, the CPU will read a byte of data from the switches. When writing to port 0003, the Z8000 will write a byte of data to the lamps. Figure 4.4 is a block diagram of the logic used to implement the interface between the CPU and the lamps and switches. The address and status lines are latched by the rising edge of $\overline{AS}$. If the address lines are 0003 and the ST3–ST0 status lines are 0010 (standard I/O status), the decode logic pulls the $\overline{SELECT}$ signal low. If the operation is a write (R/$\overline{W}$ low), the data on the lower half of the bus are latched by the data strobe and propagated to the lamps. If the operation is a read, the tri-state buffer is enabled by the data strobe, feeding the output of the eight switches onto the address/data bus, where it can be read by the CPU. This byte I/O device must have an odd address since all data connections are made to the lower half of the address/data bus.

An example of a Z-Bus interface to Z80 peripherals is illustrated in Chapter 11.

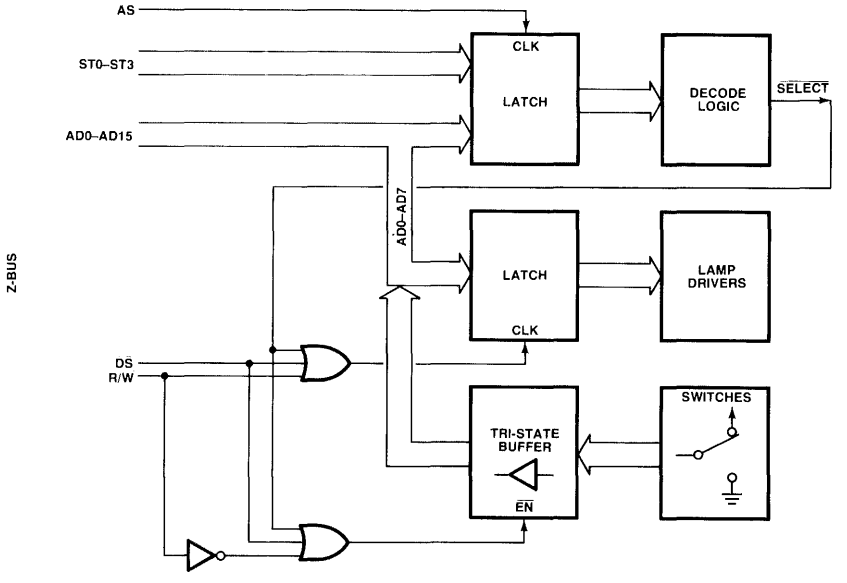


Figure 4.4 Z-Bus interface to a byte peripheral consisting of switches and lamps.

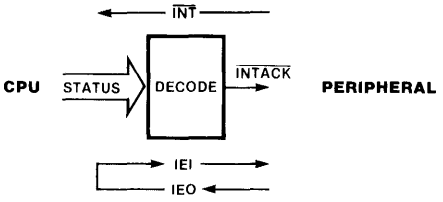


Figure 4.5 Z-Bus interrupt signals.

Z-BUS INTERRUPT SIGNALS

Besides responding to data transactions on the Z-Bus (which are always initiated by a CPU or some other bus master), peripherals in Z8000 systems can also request the CPU's attention via the CPU's interrupt inputs. The CPU responds to an interrupt by entering an interrupt acknowledge cycle; the interrupt acknowledge signal to the peripheral is generated by decoding the ST0–ST3 status signals (see Table 1.1). Peripherals can use any of the CPU's interrupt inputs ($\overline{NMI}$, $\overline{NVI}$, or $\overline{VI}$), and more than one peripheral can be connected to the same CPU interrupt input. Interrupt priority among peripherals that share a CPU interrupt line is determined by a hardware daisy chain. Each I/O device sharing the interrupt has an input, Interrupt Enable In (IEI), and an output, Interrupt Enable Out (IEO), that are used to establish an interrupt-under-service daisy chain. Thus the Z-Bus signals necessary for interrupt requests from a peripheral include an interrupt ($\overline{INT}$) signal from the peripheral, an interrupt acknowledge ($\overline{INTACK}$) signal from the CPU (decoded from the ST0–ST3 signals), and the IEI and IEO daisy-chain signals (Fig. 4.5). Interrupts and the interrupt daisy chain are described in detail in Chapter 6.

The CPU's vectored interrupt ($\overline{VI}$) input is the most commonly used interrupt for peripheral control. During the interrupt acknowledge cycle for vectored interrupts, an 8-bit vector is output by the interrupting device. The CPU reads this vector and uses it to determine the location of the interrupt service routine. During the vectored interrupt acknowledge sequence, the CPU reads the vector from the bottom half of the address/data bus (AD0–AD7). The interface logic for an I/O device is usually much simpler if all transfers between the device and the CPU can occur on the same bus lines. Thus byte peripherals that use vectored interrupts are usually connected to the bottom half of the address/data bus, so both the data and the interrupt vector can be transferred on AD0–AD7. These byte peripherals would, then, have odd addresses.

5

Instruction and Interface Timing

The Z8000 CPUs execute instructions by stepping through basic timing sequences such as memory reads and writes, I/O reads and writes, and interrupt acknowledgments. These timing sequences, called machine cycles, are designed to use the Z8000's multiplexed addresss/data bus in an efficient manner. Typical Z8000 systems will use the bus for transactions well over 80% of execution time. (In comparison, the Z80 is involved in transactions on its address and data buses about 65% of its execution time.) Thus Z8000 systems often have better throughput than competitive devices that run at faster clock speeds.

Throughput in Z8000 CPUs is optimized in two ways. First, the encoding of instruction operation codes (opcodes) is designed so that the most frequently used instructions, such as jumps and loads, have short opcodes and fast execution times. For example, a register-to-register load has a single word opcode and executes in three CPU clock cycles. Second, the execution of an instruction is overlapped with the fetch of the next instruction whenever possible, as explained below.

INSTRUCTION PREFETCH

Most Z8000 instructions conclude with several clock periods dedicated to internal CPU operations, such as arithmetic operations where a CPU register is the destination. While these internal operations are being executed, the CPU will start to fetch the next instruction's opcode (Fig. 5.1). This overlapping

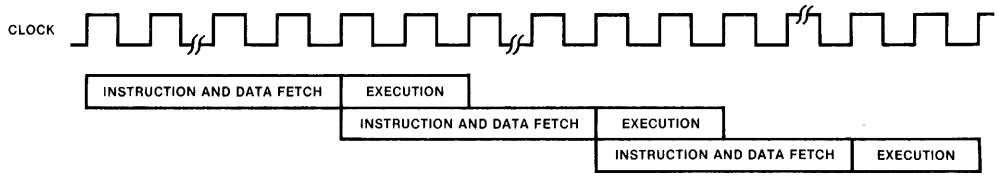


Figure 5.1 Instruction prefetch.

of instruction execution and fetching is called instruction prefetch, and improves throughput by two to three clock cycles per instruction.

Prefetch does not apply to every Z8000 instruction, however. Instructions that alter the program counter during their execution, such as jump instructions, cannot be overlapped with the following instruction's fetch, since the location of the next instruction is not determined until the jump instruction is completed. Also, instructions that involve a load to a memory or I/O location cannot be overlapped with the following instruction; these instructions end with a memory or I/O write, and the address/data bus is busy with this access. Consequently, the bus is not available for the next instruction fetch until the write operation is completed.

BASIC TIMING PERIODS

Figure 5.2 illustrates the three basic timing periods of the Z8000: clock cycles, bus transactions, and machine cycles. A clock cycle, also called a T-state, is one cycle of the CPU clock, starting with a rising edge. A bus transaction is the time required for a single data transaction on the address/data bus (that is, the time needed to move one 16-bit word of data between the CPU and a memory or I/O device). The start of a bus transaction is signaled by $\overline{AS}$ going low; bus transactions end with the rising edge of $\overline{DS}$. A bus transaction always takes three or more clock cycles depending on the type

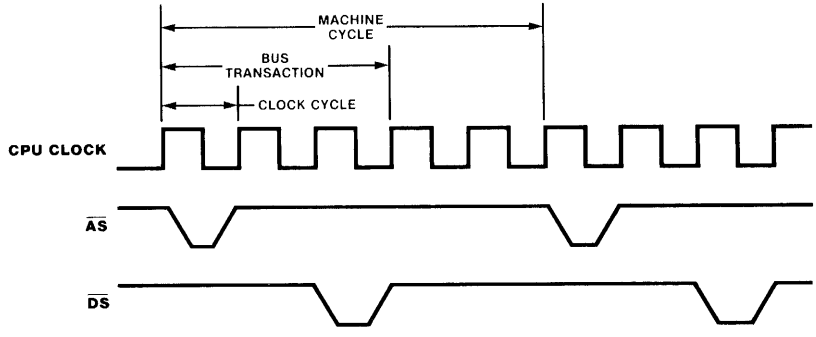


Figure 5.2 Basic timing periods for the Z8000.

of transaction and the state of the CPU's $\overline{WAIT}$ input during the transaction. A machine cycle is the time required for one basic CPU operation; that is, a machine cycle is the time from the start of one bus transaction to the start of the next bus transaction. A machine cycle can extend beyond the end of a bus transaction, as shown in Fig. 5.2, to allow time for the routing of data internal to the CPU after the bus transaction is completed.

There are four basic types of machine cycles that can occur during the execution of a Z8000 instruction: memory cycles, I/O cycles, internal operation cycles, and memory refresh cycles. Instruction execution is modular; all instructions are executed with combinations of these four basic machine cycles.

MEMORY CYCLES

Memory transactions move data between the CPU and memory. Memory transactions include instruction fetches, data reads and writes during instruction execution, and the storing of old program status and the fetching of new program status during interrupt and trap processing. (Interrupts and traps are discussed in Chapter 6.) For a given bus transaction, the code on the ST0–ST3 status lines indicates if the transaction involves a memory access and, if so, what memory address space is being accessed (Table 5.1). The $\overline{MREQ}$ line also can be used to signal that a memory access is taking place.

Instruction fetches and data reads are identical memory read cycles, with the exception of the code that appears on the ST0–ST3 status signals. Thus memory read cycles are generated to fetch instructions from memory, to read data from memory during the execution of instructions, and to fetch new program status during the interrupt acknowledge sequence or after a reset.

Figure 5.3 shows the timing of a standard memory read bus transaction. During the first clock period (T1), a 16-bit memory address is output on the address/data bus and the appropriate status information is generated on the R/$\overline{W}$, B/$\overline{W}$, N/$\overline{S}$, and ST0–ST3 lines. A high signal on the R/$\overline{W}$ line and a memory access status code on ST0–ST3 indicate a memory read transaction.

TABLE 5.1 STATUS CODES FOR MEMORY ACCESSES

ST0–ST3	Type of memory transaction
1000	Access to data memory address space
1001	Access to stack memory address space
1101	Instruction fetch of first word of instruction
1100	Instruction fetch of subsequent words of instruction
1010	Access to data memory address space involving an EPU
1011	Access to stack memory address space involving an EPU

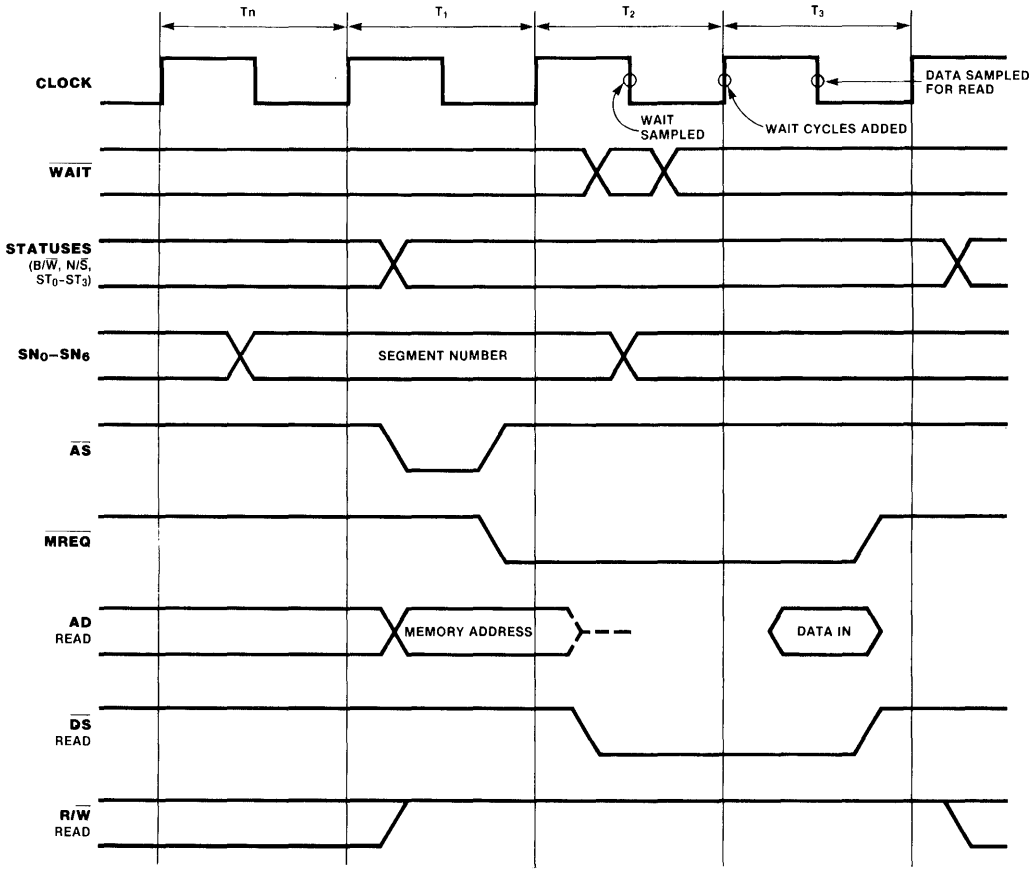


Figure 5.3 Memory read bus transaction timing.

The address strobe ($\overline{AS}$) goes low, signaling that the address is present on the bus. All address and status information is guaranteed to be valid on the rising edge of $\overline{AS}$. The status information stays valid for the remainder of the bus transaction.

For the Z8001, the segment-number portion of the address is output during the clock period preceding T1, that is, during the last clock period of the previous machine cycle (Tn in Fig. 5.3). The segment number is output earlier than the offset address during a memory access cycle to compensate for delays in memory management hardware. For example, the Z8010 MMU uses the segment-number output by the CPU to access an internal MMU register. The contents of that register control the address translation process that determines what physical address will be output by the MMU for a given logical address from the CPU. (The segment number is a part of the logical address.) This internal MMU access does take some time, of course. But

since the CPU outputs the segment number earlier than the offset portion of the logical address, the MMU has extra time to make its internal register access, and is ready to complete the address translation when the offset finally does appear. Thus the delay in the MMU between receiving a valid logical offset address from the Z8001 CPU and outputting a valid physical address to memory is minimized. (Timing considerations for the MMU are discussed further in Chapter 9.)

During T2 of the memory read cycle, the address is removed from the bus. $\overline{DS}$ is lowered, indicating that the bus is available for data. The contents of the addressed memory location can be placed on the bus at any time while $\overline{DS}$ is active.

The address/data bus is sampled by the CPU on the falling edge of the clock during T3. If the B/$\overline{W}$ line is high, indicating a byte read, only one half of the data bus will be sampled—the upper half if A0 was a 0 when the address was emitted during T1, or the lower half if A0 was a 1 (see Chapter 3). The contents of the addressed memory location must be placed on the bus before the middle of T3. The memory access time in a Z8000 system is, therefore, the time from the rising edge of $\overline{AS}$ (address guaranteed valid) until the falling edge of the clock in T3 (minus a small setup time for the data). For a three-clock-period memory read (no active $\overline{WAIT}$ signal) on a Z8000 system running at 4 MHz, this access time is a minimum of 360 ns. With a 6-MHz clock, the minimum memory access time is 220 ns; with a 10-MHz clock, the minimum memory access time is 140 ns.

If a long memory access time is necessary, the time interval between $\overline{DS}$ going low and the start of T3 can be extended by pulling the CPU's $\overline{WAIT}$ input low. The $\overline{WAIT}$ input is sampled during the middle of T2 at the falling edge of the clock. If the $\overline{WAIT}$ input is low at this time, an additional clock period is inserted into the bus transaction between T2 and T3. The $\overline{WAIT}$ signal is sampled again at the falling ege of the clock during this additional clock period; if the $\overline{WAIT}$ line is still low, yet another clock period is inserted before T3, and so on. These additional clock periods are called wait states, and designated Tw. Figure 5.4 shows memory read timing with one wait state inserted. The status of the $\overline{WAIT}$ input is sampled during each wait state to determine if additional wait states are to be added. Thus the delay between T2 and T3 can be arbitrarily long, in increments of one clock period, as controlled by the $\overline{WAIT}$ input. This allows the Z8000 to be easily interfaced to memories with arbitrarily long access times. None of the status or control outputs from the CPU change during wait states; in other words, all the CPU outputs maintain the values they had at the end of T2 during all wait states.

The memory write cycle is very similar to the memory read, as shown in Fig. 5.5. Memory write machine cycles are used to store data in memory during instruction execution and to store program status during the interrupt acknowledge sequence. During T1, address and status information is output;

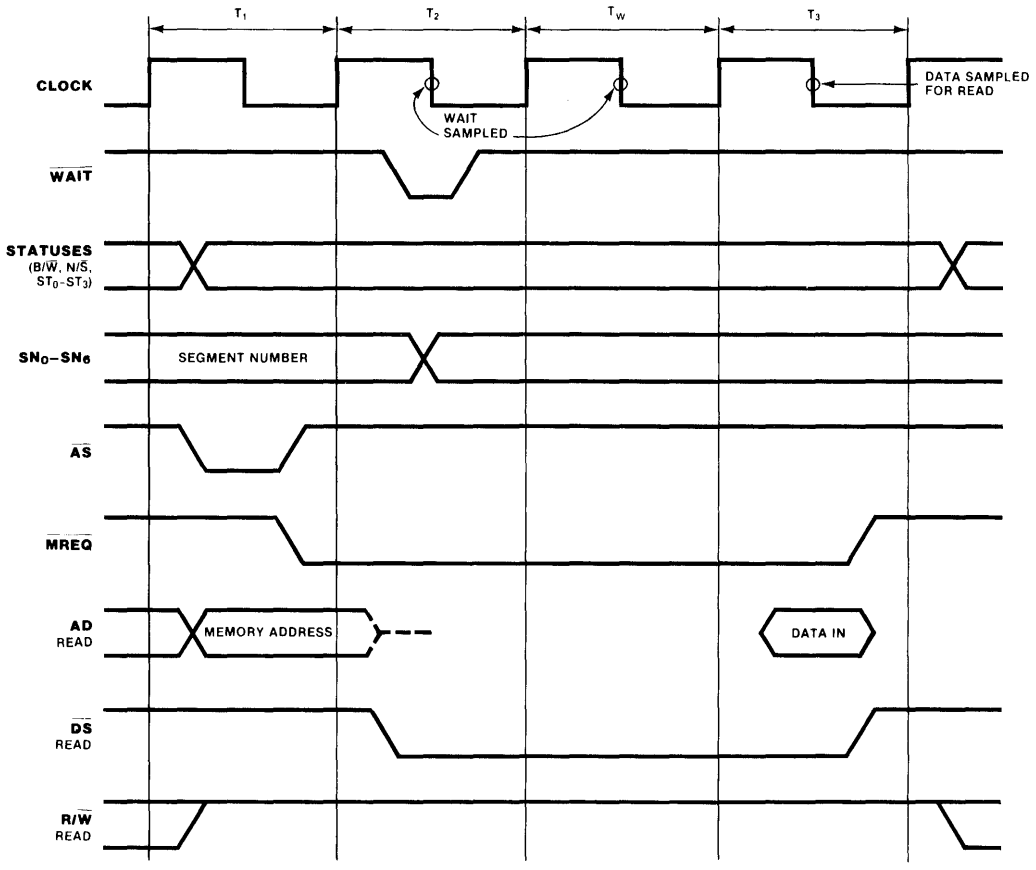


Figure 5.4 Memory read bus transaction timing with one wait state.

the address and status signals are valid at the rising edge of $\overline{AS}$. Memory transaction status on the ST0–ST3 lines and a low on the R/$\overline{W}$ line indicate that this transaction is a memory write. During T2, the address is removed from the bus, the data to be written to memory are placed on the bus by the CPU, and $\overline{DS}$ goes active. For byte transactions (B/$\overline{W}$ high), the byte of data to be written will appear on both halves of the address/data bus while $\overline{DS}$ is active. The bus transaction is completed during T3 when $\overline{DS}$ goes high. The data are guaranteed to be valid on the bus and can be written into memory anytime while $\overline{DS}$ is low. Thus the access time for a memory write is the time from the rising edge of $\overline{AS}$ until the rising edge of $\overline{DS}$. As with memory reads, the $\overline{WAIT}$ input is sampled in the middle of T2, and any resulting wait states are added between T2 and T3.

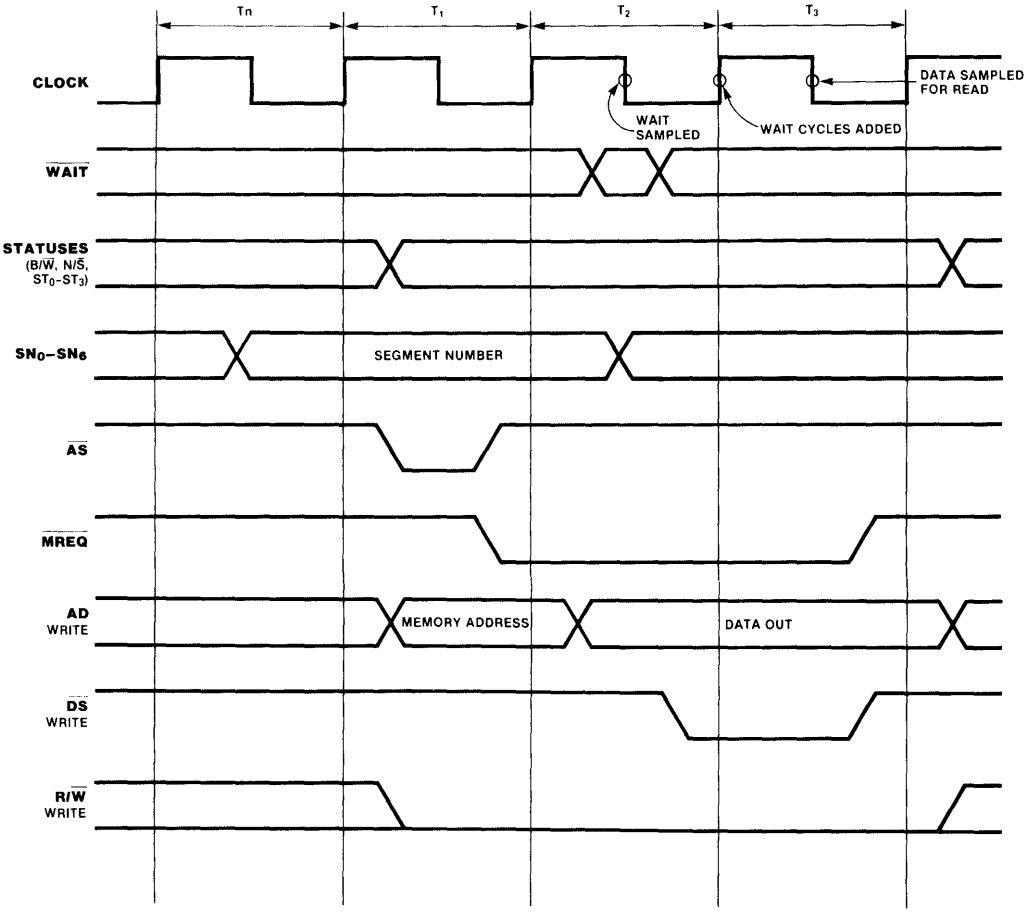


Figure 5.5 Memory write bus transaction timing.

The bus transaction time for a memory read or write is always three clock periods, excluding possible wait states. However, for some instructions, the machine cycle for a memory access can be one to four clock cycles longer than the bus transaction time. These extra cycles are added after T3, but before the start of the next bus transaction; they are used to allow extra time for the routing of data internal to the CPU.

I/O CYCLES

An I/O bus transaction moves data to or from a peripheral device and is generated as the result of the execution of an I/O instruction. The ST0-ST3

status lines indicate a standard I/O (0010 status) or a special I/O (0011 status) reference during I/O cycles.

I/O read timing is illustrated in Fig. 5.6. I/O bus transactions are four clock cycles long as a minimum but can be lengthened by the addition of wait states. During T1, the 16-bit I/O address is placed on the address/data bus, and the appropriate status signals are output. The address and status information are guaranteed to be valid at the rising edge of $\overline{AS}$. An I/O read is indicated by I/O status on the ST0–ST3 lines and a high on the R/$\overline{W}$ line. The N/$\overline{S}$ output is always low during I/O transactions, since I/O instructions can be executed only in the system mode. The status information remains valid for the remainder of the bus transaction.

During T2 of the I/O read cycle, the I/O address is removed from the bus and then $\overline{DS}$ is lowered. $\overline{DS}$ low indicates that the bus is available to receive data from the peripheral device. Data must be placed on the bus before the falling edge of the clock in the middle of T3, at which time the

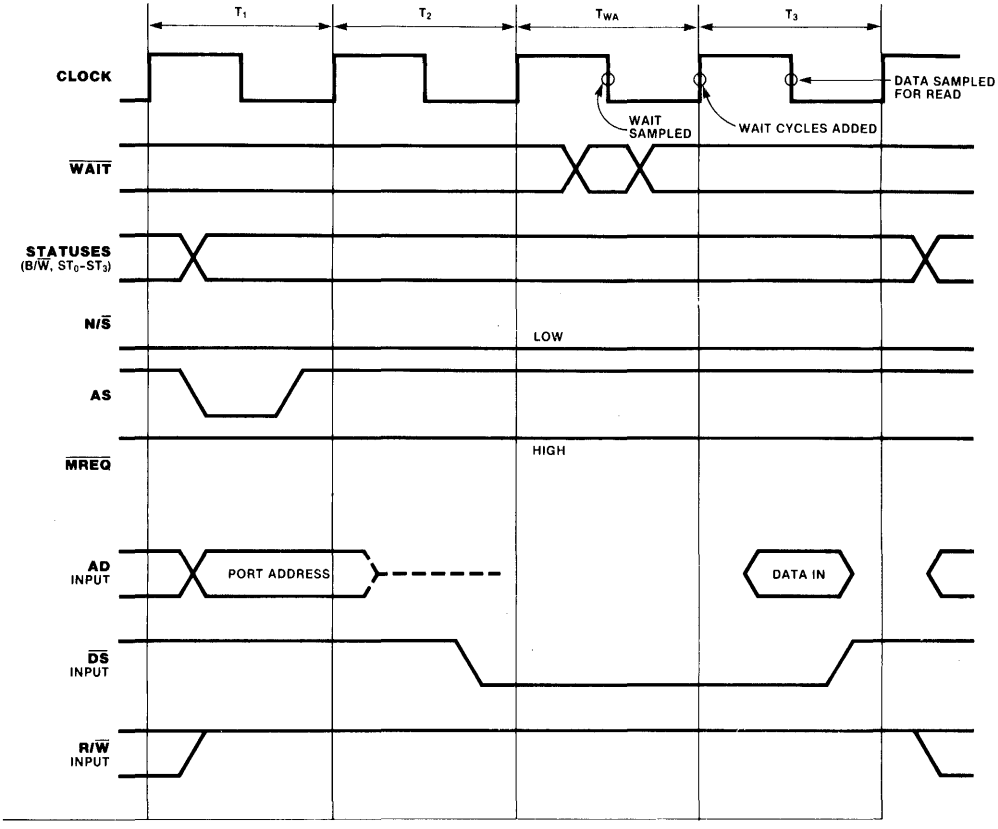


Figure 5.6 I/O read bus transaction timing.

CPU samples the data bus. For a byte read (B/$\overline{W}$ high), the CPU will accept data from one-half of the 16-bit bus, depending on the value of A0 of the I/O address output during T1.

One wait state, T_{WA}, is always added between T2 and T3 of the I/O machine cycle, regardless of the state of the CPU's $\overline{WAIT}$ input. On the falling edge of the clock in the middle of T_{WA}, the $\overline{WAIT}$ input is sampled. If the $\overline{WAIT}$ line is low, another wait state is inserted before T3. The $\overline{WAIT}$ input is sampled during each wait state, and, as with memory cycles, an arbitrary number of wait states can be added to the I/O cycle between T2 and T3. None of the status or control signals change during the wait states. Thus the I/O bus transaction timing can be lengthened through the use of wait states to accommodate slow peripheral devices.

The I/O write transaction timing is shown in Fig. 5.7. The I/O address and status information is output during T1 and is valid at the rising edge of $\overline{AS}$. An I/O write is signaled by I/O status on the ST0–ST3 lines and a low

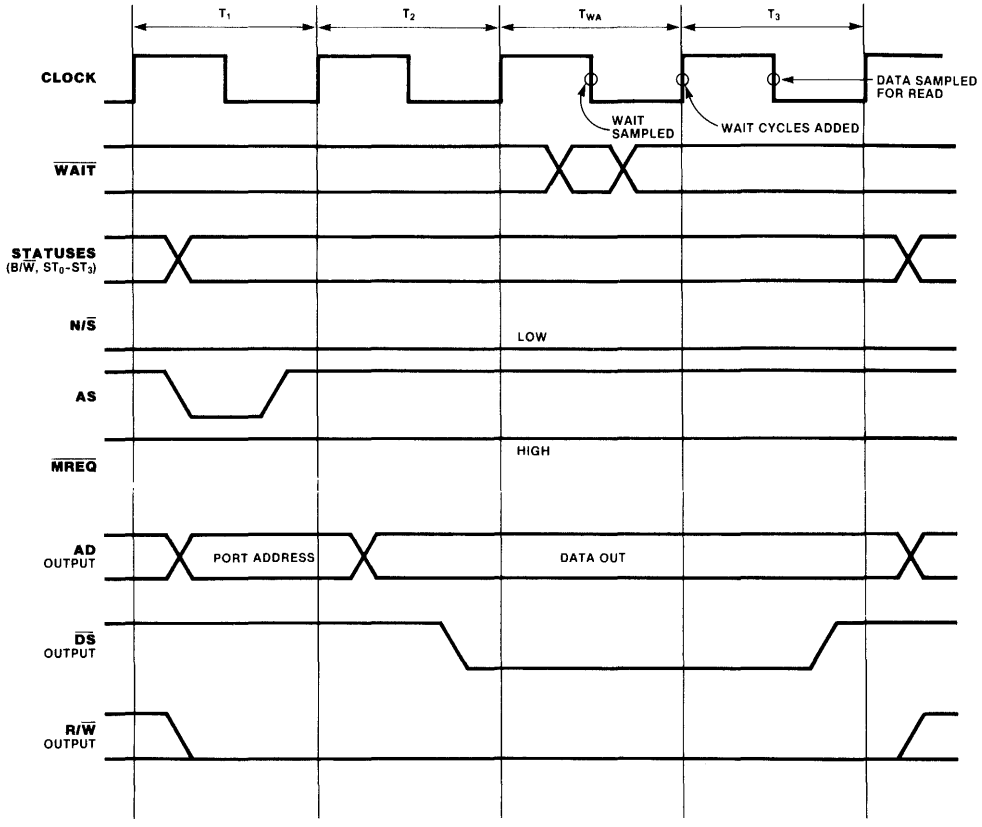


Figure 5.7 I/O write bus transaction timing.

on the R/$\overline{W}$ line. During T2, the address is removed from the address/data bus, the data to be written are placed on the bus, and $\overline{DS}$ goes low. For a byte write (B/$\overline{W}$ high), the byte of data to be written appears on both halves of the bus. The data to be written remain on the bus and can be read by the peripheral device throughout the duration of $\overline{DS}$ active. One automatic wait state, T_{WA}, is always inserted between T2 and T3. The rising edge of $\overline{DS}$ marks the end of the write operation. Of course, more wait states can be added between T2 and T3 by pulling the $\overline{WAIT}$ input low. The timing of I/O transactions is similar to the timing of memory transactions, except for the addition of the one automatic wait state.

INTERNAL OPERATION CYCLES

The execution of some Z8000 instructions, such as multiplies and divides, need extra machine cycles during which the CPU is performing internal operations (data routing and ALU operations) and no data transactions are occurring on the address/data bus. During these times, an internal operation bus transaction takes place (Fig. 5.8). During T1, arbitrary values are output on the address/data bus and the $\overline{AS}$ is pulsed. The ST0-ST3 status lines indicate internal operation status (0000), the R/$\overline{W}$ line is high, the N/$\overline{S}$ line remains the same as in the previous machine cycle, and the B/$\overline{W}$

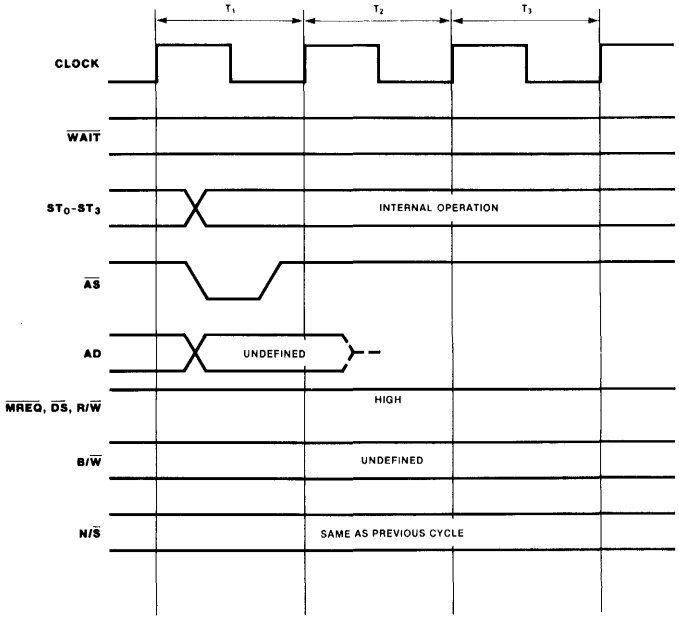


Figure 5.8 Internal operation bus transaction timing.

line is undefined. These status conditions hold throughout the remainder of the machine cycle. The $\overline{DS}$ and $\overline{MREQ}$ control lines also are held high throughout the cycle. Thus an internal cycle looks like a memory cycle without the data strobe, and no data are transferred.

Figure 5.8 shows an internal operation cycle that is three clock periods long; that is the minimum number of clock periods for an internal operation machine cycle. An internal operation can extend up to a maximum of eight clock periods, depending on the instruction being executed. Eight clock periods is the longest possible machine cycle in the Z8000, excluding the interrrupt acknowledge cycle (10 clocks), unless a memory or I/O access has multiple wait states added via the $\overline{WAIT}$ input. Address strobe ($\overline{AS}$) is pulsed during T1 of every machine cycle, regardless of whether or not a valid address is emitted or a data transfer is to occur in that cycle. Therefore, $\overline{AS}$ can be used to trigger memory refresh logic external to the CPU, such as the refresh in the Z6132 Quasi-Static RAM.

The upper limit on the number of clock periods in a machine cycle ensures a fast response time for bus requests. When a CPU receives a bus request via the $\overline{BUSREQ}$ input, that request will be serviced at the end of the current bus transaction. (Bus requests are discussed in Chapter 7.)

The $\overline{WAIT}$ input is not sampled during internal operation cycles.

MEMORY REFRESH CYCLES

A memory refresh cycle is generated by the Z8000 CPU's automatic memory refresh logic. If automatic refresh is enabled (that is, bit 15 in the refresh register is set), the Z8000 will enter a memory refresh cycle as soon as possible after the rate counter in the refresh register goes to 0. Memory refresh machine cycles are always three clock periods long, as shown in Fig. 5.9. During T1, refresh status (0001) is emitted on the ST0–ST3 status lines, the 9-bit row address from the refresh register is emitted on AD0–AD8 of the address/data bus, and $\overline{AS}$ is pulsed. The $\overline{MREQ}$ line goes low, and the R/$\overline{W}$, B/$\overline{W}$ and N/$\overline{S}$ lines remain the same as in the preceding machine cycle. These status conditions remain throughout the refresh cycle. The $\overline{DS}$ stays high throughout the refresh operation and no data transfer takes place. The system's memory control logic would use the refresh status indication on the ST0–ST3 lines and the row address emitted on the bus to activate the refresh of dynamic memories.

After the rate counter in the refresh register counts down to 0, the CPU will insert the refresh cycle immediately after the next IF1 machine cycle (the fetch of the first word of the next instruction) or the next internal operational cycle, whichever comes first. Since the rate counter is programmable, the user can ensure that refreshes will occur only as often as neces-

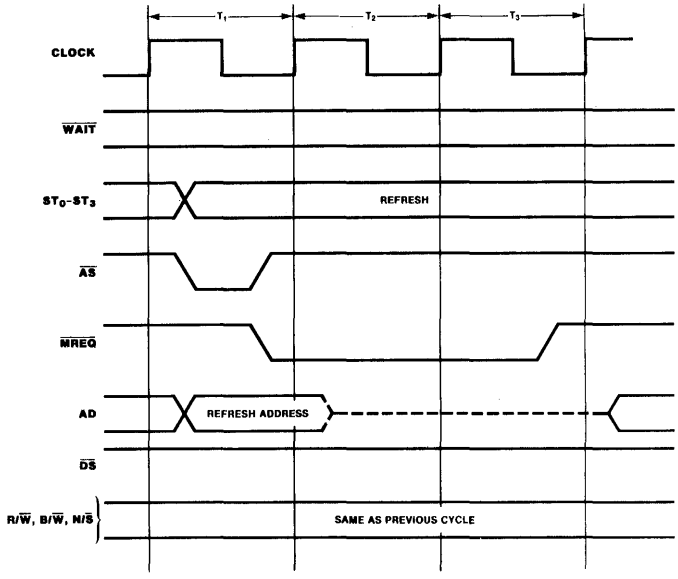


Figure 5.9 Memory refresh bus transaction timing.

sary for the system's memory devices. Therefore, the effect of the memory refresh on CPU throughput can be minimized. Of course, if the automatic memory refresh mechanism is disabled (bit 15 in the refresh register is a 0), memory refresh cycles will never occur.

As with internal operations, the $\overline{\text{WAIT}}$ input is not sampled during memory refresh cycles. Internal operations and memory refreshes are the only types of bus transactions that do not involve a transfer of data.

All Z8000 instructions are executed using a combination of memory, I/O, and internal operation cycles, with refresh cycles added as dictated by the refresh control register. For example, adding two word registers takes one memory read cycle (an IF1 cycle) that is four clock periods long. Dividing a word register by another word register takes a five-clock-period memory read (IF1 cycle), followed by two six-clock-period and one seven-clock-period internal cycles. Writing a byte from a register to an I/O device (direct-address addressing mode) requires a five-clock-period and a three-clock-period instruction fetch, and a four-clock-period I/O cycle.

Special exception conditions can cause timing sequences other than the four basic machine cycles described above. The CPU responds to interrupts, traps, and bus requests with special acknowledgment cycles. An active $\overline{\text{RESET}}$ or $\overline{\text{STOP}}$ input forces the CPU into special sequences. Execution of a HALT or MREQ (multi-micro request) instruction also alters CPU timing sequences. These special timing cycles will be discussed in later chapters.

AC TIMING CHARACTERISTICS

Table 5.2 lists some of the timing characteristics of the Z8000 CPUs that apply to memory and I/O interfacing. These figures are compiled for Z8001/Z8002 CPUs running at 4 MHz and Z8001A/Z8002A CPUs running at 6 MHz. Most of these timing parameters are dependent on the cycle time of the CPU clock; see Appendix A for the complete Z8000 AC timing characteristics and the corresponding timing diagram.

During T1 of a memory or I/O access, the rising edge of $\overline{AS}$ signals that the address and status information is valid from the CPU. TdA(As) and TdS (AS) (first two entries of Table 5.2) define the setup time for the address and status outputs before the rising edge of $\overline{AS}$; TdAS(A) is the hold time for the address after $\overline{AS}$ goes inactive. Thus if $\overline{AS}$ is used to latch the address as part of the memory control logic, the setup and hold requirements of that latch are dictated by these parameters. Similar setup and hold times for the address in relation to the falling edge of $\overline{MREQ}$ are also given.

For a memory read, the delay from the rising edge of $\overline{AS}$ (address valid) until the CPU samples the data bus defines the memory access time, TdAS(DR). If $\overline{MREQ}$ is used in place of $\overline{AS}$ to detect a valid address, a slight gain in access time can be realized [TdMR(DR)].

TABLE 5.2 Z8000 CPU AC TIMING CHARACTERISTICS

Symbol	Parameter	Minimum time (ns)	
		Z8001/Z8002	Z8001A/Z8002A
TdA(AS)	Address Valid to $\overline{AS}\uparrow$ delay	55	35
TdS(AS)	Status Valid to $\overline{AS}\uparrow$ delay	50	30
TdAS(A)	$\overline{AS}\uparrow$ to Address not Valid delay	70	45
TdA(MR)	Address Valid to $\overline{MREQ}\downarrow$ delay	55	35
TdMR(A)	$\overline{MREQ}\downarrow$ to Address not Valid delay	70	35
TdAS(DR)	$\overline{AS}\uparrow$ to Data In Required Valid (Memory Read)	360	220
TdMR(DR)	$\overline{MREQ}\downarrow$ to Data In Required Valid (Memory Read)	375	230
TsDR(C)	Data In to Clock$\downarrow$ Setup Time (Memory or I/O Read)	30	20
ThDR(DS)	Data In to $\overline{DS}\uparrow$ Hold Time (Memory or I/O Read)	0	0
TdDW(DSW)	Data Out Valid to $\overline{DS}\downarrow$ delay (Write)	55	35
TdDS(DW)	$\overline{DS}\uparrow$ to Data Out and Status not Valid (Write)	75	45
TsWT(C)	$\overline{WAIT}$ to Clock$\downarrow$ Setup Time	50	30
ThWT(C)	$\overline{WAIT}$ to Clock$\downarrow$ Hold Time	10	10

During memory and I/O reads, valid data must be present on the bus slightly before the falling edge of the clock in the middle of T3, as dictated by the minimum requirement for TsDI(C). However, there is no hold time requirement on the data after $\overline{DS}$ goes inactive. For memory and I/O writes, valid data is output from the CPU before $\overline{DS}$ is pulled low [TdDW(DSW)], and stays valid after $\overline{DS}$ returns high [TdDS(DW)]. Thus setup and hold-time requirements for the data inputs of memory and I/O devices are easily satisfied.

TsWT(C) and ThWT(C) are the setup and hold-time requirements of the $\overline{WAIT}$ input in relation to the falling edge of the clock in T2. These requirements must be satisfied in order for the $\overline{WAIT}$ input to be properly sampled and wait states inserted in a memory or I/O cycle.

MEMORY INTERFACE TIMING: AN EXAMPLE

When designing memory and I/O interface logic for Z8000-based systems, the AC timing characteristics of the CPU as well as the basic timing of the memory and I/O bus transactions must be taken into account. For example, suppose that the Z8002-Z6132 system with 4K words of memory described in Chapter 3 (repeated here as Fig. 5.10) is running with a 4-MHz CPU clock. Since the memory access time for the Z8002 is a minimum of 360 ns, the Z6132's must have an access time less than 360 ns. The Z6132-5 memory chip, with a maximum access time of 300 ns, would be a logical choice. Some AC timing characteristics of the Z6132-5 are shown in Table 5.3. Assume that the combinational logic in Fig. 5.10 consists of low-power Schottky TTL logic, such as 74LS00's, with a maximum gate delay of 10 ns.

The Z8002's address/data bus, $\overline{AS}$, $\overline{DS}$, and R/$\overline{W}$ signals are connected directly to the Z6132's. Examining Table 5.2 and 5.3, the setup and hold-time requirements of the Z6132-5 for the address and $\overline{WE}$ signals with respect to $\overline{AS}$ rising and the data with respect to $\overline{DS}$ falling are easily met by the Z8002. For example, the Z6132-5 requires that the address stay valid

TABLE 5.3 Z6132-5 AC TIMING CHARACTERISTICS

		Nanoseconds	
Symbol	Parameter	Min.	Max.
TsCS(AC)	$\overline{CS}$ to AC↑ Setup Time	0	
ThCS(AC)	$\overline{CS}$ to AC↑ Hold Time	50	
TsA(AC)	Address to AC↑ Setup Time	0	
ThA(AC)	Address to AC↑ Hold Time	50	
TsW(AC)	$\overline{WE}$ to AC↑ Setup Time	-20	
ThW(AC)	$\overline{WE}$ to AC↑ Hold Time	80	
TdAC(DO)	AC↑ to Data Out Delay (Read)		300
TdDS(DOz)	$\overline{DS}$↑ to Data Out Float (Read)	40	
TsDI(DS)	Data In to $\overline{DS}$↓ Setup Time (Write)	0	
ThDI(DS)	Data In to $\overline{DS}$↓ Hold Time (Write)	60	

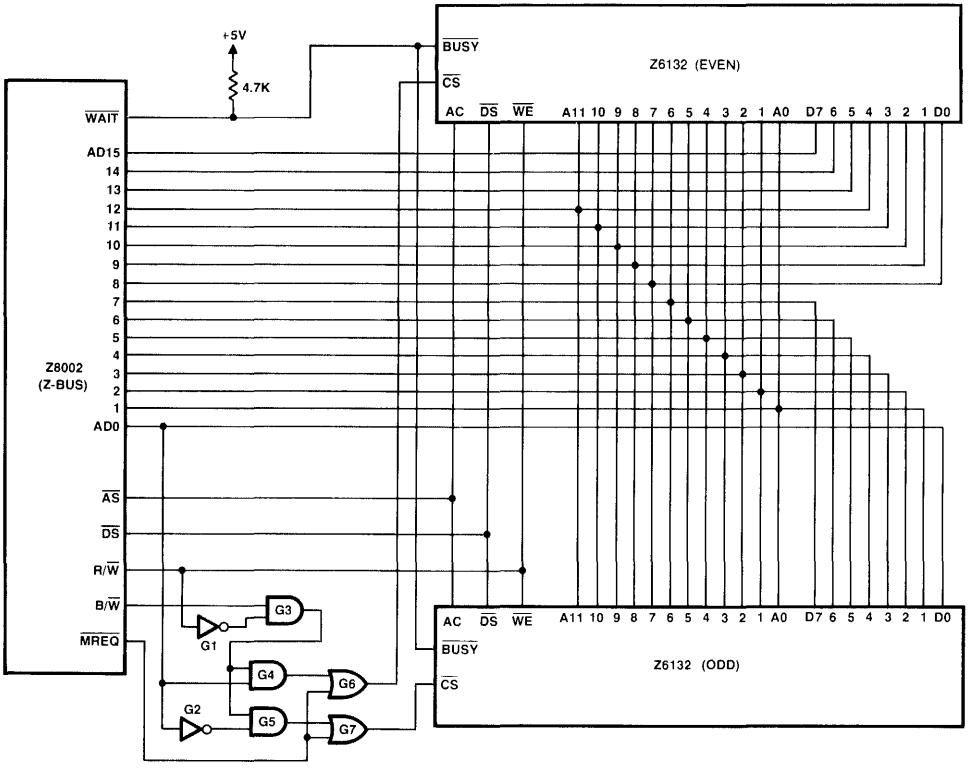


Figure 5.10 Z8002-Z6132 interface logic for a 4K word system.

for 50 ns after AC rises [ThA(AC) in Table 5.3], whereas the Z8002 guarantees that the address will remain valid 70 ns after $\overline{AS}$ goes high [TdAS(A) in Table 5.2].

The only timing parameter that needs to be examined in this system is the chip select ($\overline{CS}$) setup time for the memory. The Z8002 guarantees that status information is valid at least 50 ns before $\overline{AS}$ rises. The worst-case timing condition for the chip select occurs during byte writes, when the R/$\overline{W}$ signal must propagate through four gates (G1, G3, G4, and G6) before the $\overline{CS}$ signal is stable. Assuming the worst-case conditions, the delay through each gate is 10 ns, or 40 ns for all four gates. The $\overline{CS}$ signal is not valid until 40 ns after the status is valid, or 10 ns before the rising edge of $\overline{AS}$. Since the setup time for $\overline{CS}$ is 0, this still satisfies the timing requirements for the Z6132-5.

As memory systems get larger and more complex, the logic needed to generate chip selects also grows more complex (see Figures 3.17 and 3.18, for example). In larger systems, the $\overline{AS}$ signal to memory may need to be delayed before reaching the memory control logic to ensure that all address and status information meets the setup and hold-time requirements for the

memory used. For example, if the delay from status valid until chip select valid in the system analyzed above were 55 ns instead of 40 ns, the $\overline{AS}$ to memory would have to be delayed 5 ns to ensure that the rising edge of $\overline{AS}$ does not arrive at the memory before $\overline{CS}$ is stable. If buffers are added between the CPU and memory control logic, the minimum and maximum delays through the buffers also must be considered.

WAIT-STATE GENERATION

If a memory or peripheral device is too slow to respond in the time allowed within a standard memory or I/O bus transaction, the access time of the data transfer can be lengthened by adding wait states. Wait states are inserted into a transaction by pulling the $\overline{WAIT}$ input to the CPU low; this input is sampled internally by the CPU on the falling edge of the clock during T2 and all wait states (Tw's) in all memory and I/O machine cycles. The $\overline{WAIT}$ input must meet the setup and hold-time requirements shown in Table 5.2 in order for the CPU to recognize it and add the desired wait states to the transaction.

The $\overline{WAIT}$ signal to the CPU can be generated in a number of ways. The most common is to synchronize the $\overline{WAIT}$ input to the CPU clock. For example, suppose that all the program memory in a Z8002-based system is implemented in 2716-5 EPROMs (such as the system in Fig. 3.18). With a 4-MHz CPU clock, the minimum memory access time without any wait states would be 360 ns. However, 2716-5's have an access time of 490 ns, thereby requiring that one wait state be added during each program memory access. (With one wait state, the minimum memory access time would be 360 + 250 = 610 ns at 4 MHz.) An SN7474 dual D-type flip-flop could be used to generate the $\overline{WAIT}$ signal, as illustrated in Fig. 5.11. When $\overline{AS}$ goes low at the start of each cycle, the Q output of both flip-flops is forced low. On the next rising clock edge (the start of T2), Q1 goes high but Q2 remains low. The CPU will sample the $\overline{WAIT}$ input in the middle of T2 and, if the cycle is a program memory access, the $\overline{WAIT}$ input will be low, causing a wait state to be inserted after T2. On the next positive clock edge (the beginning of Tw), Q2 goes high, driving the $\overline{WAIT}$ line high. The CPU samples the $\overline{WAIT}$ input again in Tw and, since it is inactive, no further wait states are added. Since the $\overline{WAIT}$ signal is gated by $\overline{PROGRAM\ MEMORY\ STATUS}$, the wait state will be added only during accesses to program memory. ($\overline{PROGRAM\ MEMORY\ STATUS}$ is formed by decoding the ST0–ST3 status signals.)

Insertion of multiple wait states for a particular memory or I/O access is easily implemented with a parallel-load shift register. A 4-bit shift register, such as the SN74178 or SN74195, can be used for inserting up to three wait states. For example, a circuit for adding three wait states to each I/O access is diagrammed in Fig. 5.12. I/O status is decoded from the ST0–ST3 status

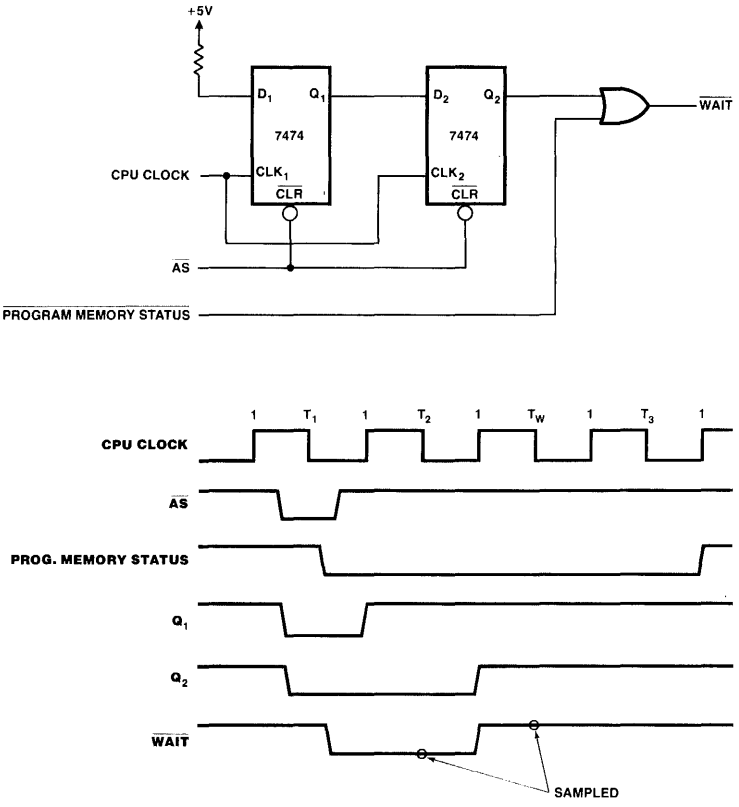


Figure 5.11 Generation of a single wait state with D-type flip-flops.

lines; as long as I/O status is not present, the shift register performs a parallel load on each positive CPU clock edge (the SN74178 is negative edge triggered, so the CPU clock is inverted). The D input is propagated to the Q_D output, and $\overline{\text{WAIT}}$ is held high. When I/O status does appear during T1 of an I/O access cycle, the shift register enters its shift mode. The next positive CPU clock edge will shift the register one place to the right. The Q_C output was low prior to the shift, since the C input was low during the last parallel load, so a low value is shifted into Q_D. The next two shifts will also shift low values into Q_D, since Q_B and Q_A start out low; subsequent shifts will send Q_D high again, since the serial input to the register is a 1. Thus three wait states will be inserted in the cycle. If only two wait states are required, the A input to the shift register can be tied high instead of low.

Both of the examples above used the status outputs to trigger the generation of the $\overline{\text{WAIT}}$ signal. A chip select to a given device or group of devices could also have been used, if wait states are to be inserted only when those

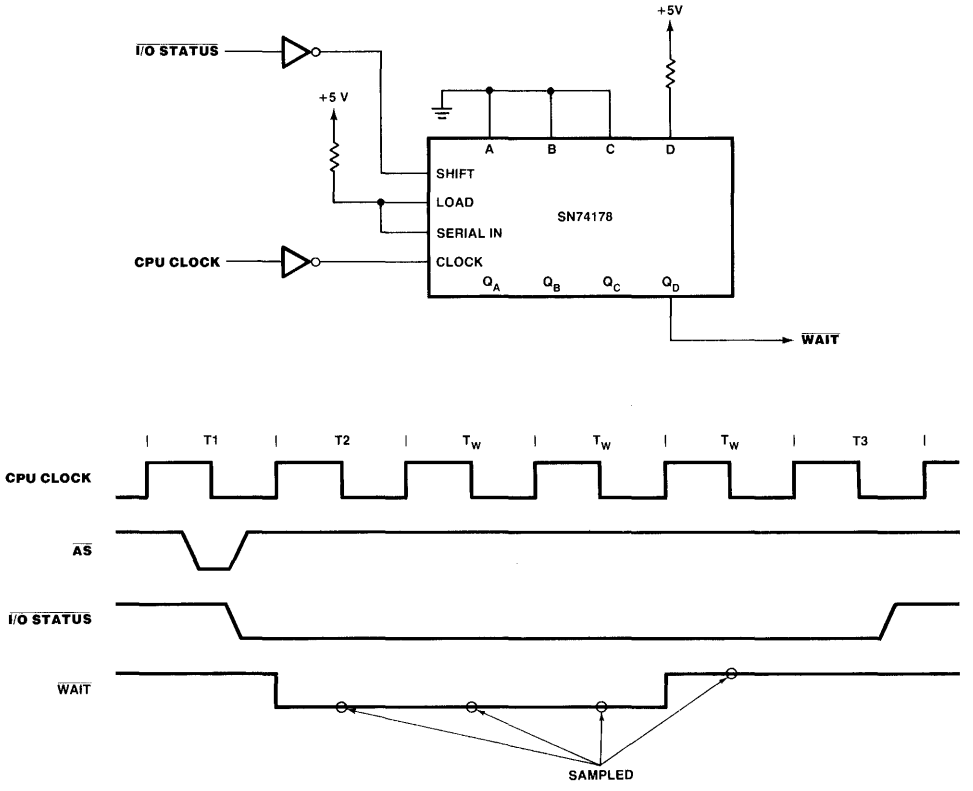


Figure 5.12 Generation of three wait states with a shift register.

particular devices are accessed. If more than three wait states are needed, 8-bit shift registers such as the SN74165 can be used, or shift registers can be cascaded together to any length. If $\overline{\text{WAIT}}$ signals are generated by more than one circuit in the system, they would be ORed together before being input to the CPU.

The ability to insert wait states into any memory, I/O, or interrupt acknowledge cycle allows the CPU to be interfaced easily to slow memory and peripheral devices. All Z-Bus data transactions are asynchronous; the access time for a memory or I/O transaction can be arbitrarily long, in increments of one CPU clock cycle, as determined by $\overline{\text{WAIT}}$ generation logic external to the CPU.

6

Interrupts, Traps, and Resets

The execution of a single instruction in a computer consists of several steps: the instruction's opcode is fetched from memory, execution of the instruction is performed, the appropriate flags are set, and the program counter is updated to point at the next instruction. Typically, instructions within a programming task are executed sequentially in the order in which they appear in memory or in an order determined by instructions that change the program counter, such as jumps and subroutine calls. Three events can alter the normal execution of a Z8000 program: interrupts, traps, and resets. These events are called exception conditions. Interrupts are caused by an active signal on one of the CPU's three interrupt inputs: $\overline{\text{NMI}}$, $\overline{\text{NVI}}$, and $\overline{\text{VI}}$. Traps occur if certain error conditions occur during instruction execution, such as an attempt to execute a privileged instruction while in normal-mode operation. Interrupts and traps are recognized at the end of the execution of the current instruction and cause the CPU to temporarily suspend the execution of the current programming task. Execution is transferred to a procedure that performs whatever actions are necessary as a result of the interrupt or trap; such a procedure is called a service routine. Resets are caused by an active low level on the $\overline{\text{RESET}}$ input. A reset overrides all current operatting conditions and puts the CPU in a known state for starting program execution.

INTERRUPTS

Interrupts are the means by which a peripheral device can request the CPU's attention. Interrupts are asynchronous events that can occur at any time during program execution. An interrupt causes the currently executing task to be suspended while the CPU responds to the device that asserted the inter-

rupt. Allowing peripherals to interrupt normal CPU operation eliminates the need for periodic polling of the status of peripheral devices, thereby increasing system throughput while improving response time.

The Z8000 CPUs support three kinds of interrupts: nonmaskable, nonvectored, and vectored. A CPU's interrupt input can be shared by several different interrupt sources; several peripheral devices might be connected to the $\overline{VI}$ input, for example. Priority among devices using a common interrupt is determined by a hardware daisy chain.

Nonmaskable interrupts, as the name implies, are interrupts that cannot be disabled via program execution. Usually, nonmaskable interrupts are used to inform the CPU of a situation that requires immediate handling to preserve system integrity, such as an imminent power failure. Nonmaskable interrupts are edge triggered; a high-to-low transition on the CPU's $\overline{NMI}$ input will cause recognition of this interrupt.

Vectored and nonvectored interrupts are maskable; that is, the CPU can be programmed to ignore or respond to active low levels on the $\overline{VI}$ and $\overline{NVI}$ inputs. Bit 12 of the flag and control word (FCW) is the vectored interrupt enable, and bit 11 is the nonvectored interrupt enable. A 1 in the appropriate bit enables the interrupt, and a 0 disables it. (Of course, these control bits in the FCW can be altered only in system mode.) Typically, vectored and nonvectored interrupts are used by peripheral devices to request servicing from the CPU. These interrupts are level sensitive; in order for the CPU to recognize a vectored or nonvectored interrupt, the $\overline{VI}$ or $\overline{NVI}$ input must be held low until the resulting interrupt acknowledge sequence is executed.

One result of any interrupt is that a 16-bit word is read from the address/data bus as part of the interrupt acknowledge cycle. This word can be used to identify the source of the interrupt, if so desired, and is called an identifier word. For vectored interrupts, the lower byte of this word is a pointer that selects a particular service routine. In other words, the CPU will use the byte returned on AD0–AD7 during the interrupt acknowledge cycle to determine the starting location of the service routine for that interrupt. This byte of data is called an interrupt vector. Thus a vectored interrupt will result in the execution of one of a number of possible service routines, depending on the value of the vector read from the peripheral. Nonvectored and nonmaskable interrupts, on the other hand, each have just one service routine corresponding to that interrupt type. That same routine is called each time the interrupt occurs. However, that routine could examine the identifier word returned during the acknowledge sequence, and then branch to the appropriate procedure corresponding to the identifier's value.

TRAPS

Traps are synchronous events that are triggered by attempting to execute certain instructions. Unlike interrupts, traps are predictable and repeatable;

if a particular instruction in a program causes a trap, that trap should recur each time that instruction is executed with the same set of conditions (that is, the same processor state). The Z8001 and Z8002 CPUs recognize three kinds of traps: extended instruction traps, privileged instruction traps, and system calls. Each of these trap conditions is generated internally by the CPU during program execution. The Z8001 CPU recognizes a fourth, externally generated trap, the segmentation trap.

Extended instruction traps are caused by an attempt to execute an instruction intended for an Extended Processor Unit (EPU) when no EPUs are present in the system. Bit 13 of the FCW is the extended processor architecture (EPA) bit; a 1 in this bit signals that EPUs are present in the system, and a 0 means no EPUs are present. The instructions that are meant to be executed by the Extended Processor Units, and not by the CPU, include all instructions with an opcode whose first word has a 0E, 0F, 4E, 4F, 8E, or 8F (hexadecimal) in the upper byte. If one of these instructions is encountered and the EPA bit of the FCW is 0, an extended instruction trap will occur. The service routine for the trap could simulate the action that the EPU would take if it were present, thereby allowing the system to function without an EPU that is to be acquired and added to the system at a later time.

Privileged instruction traps occur when execution of a privileged instruction is attempted while in normal-mode operation. The current operating mode (system or normal) is determined by the S/$\overline{N}$ bit (bit 14) in the FCW. I/O instructions, instructions that alter the control bits in the FCW, the multi-micro instructions, and the HALT instruction are all privileged instructions. This trap prevents normal-mode users from corrupting the operating system environment; that is, normal-mode programs cannot act directly on any of the system's hardware functions. The service routine for the privileged instruction trap could simulate the operation attempted by the normal-mode user; this service routine would be part of the operating system, and the trap would be considered as a request to the operating system to perform some hardware function.

The system call trap occurs when a System Call (SC) instruction is executed. The System Call instruction is, in essence, a software trap. By executing a System Call instruction, a normal-mode program would initiate execution of the system call trap's service routine. This service routine could be written as part of the operating system to provide some system-mode functions, such as I/O routines, that can be accessed by normal-mode users. Thus the system call trap, like the privileged instruction trap, can be used to provide normal-mode programs with a controlled means of accessing operating system functions.

The segmentation trap occurs whenever the $\overline{SEGT}$ input to the Z8001 is pulled low, regardless of whether the processor is in the segmented or the nonsegmented mode. This trap is generated by memory management hardware, such as the Z8010 MMU, when that hardware detects an illegal mem-

ory access. As part of memory management, memory segments in Z8001 systems can be assigned specific sizes and attributes that determine what types of accesses can be made to that segment. If an illegal access is attempted, the MMU notifies the CPU via the segmentation trap. For example, a segmentation trap could signal an attempted write to a read-only segment, or an attempt to access a memory address outside the limits of a segment. The Z8010 MMU includes status registers that can be read as part of the segmentation trap service routine to determine the exact cause of the trap. (The Z8010 MMU is described in Chapter 9.)

The action that Z8000 CPUs take in response to an interrupt or trap is very similar. The major distinction between interrupts and traps is their origin. Interrupts are asynchronous events caused by a device outside the CPU and are usually independent of the currently executing instruction. Traps are synchronous events caused by instruction execution, and are always reproducible by reexecuting the program that caused the trap.

INTERRUPT AND TRAP HANDLING

At the beginning of T3 of the last machine cycle of an instruction, the Z8000 CPUs sample the interrupt and $\overline{\text{SEGT}}$ inputs to see if an interrupt or segmentation trap is pending. If an interrupt input is active and enabled, or if the execution of the last instruction caused a trap condition, the CPU will respond to the interrupt or trap instead of executing the next instruction of the current task. Response to an interrupt or trap consists of six steps: determining priority of competing events, executing an acknowledge cycle (for interrupts and segmentation traps only), saving the current program status, loading the service routine's program status, executing the service routine, and returning to the interrupted program.

PRIORITIES OF EXCEPTIONS

If more than one interrupt or trap condition is present at one time (that is, when sampled during the last machine cycle of an instruction), the CPU responds to the exception condition with the highest priority. Internal traps (extended instruction, privileged instruction, or system call) have the highest priority, followed by nonmaskable interrupts, segmentation traps, vectored interrupts, and nonvectored interrupts, in that order. Of course, resets have a higher priority than any other condition; whenever the $\overline{\text{RESET}}$ pin is active, a reset is performed immediately.

For some of the classes of events listed above there can be multiple sources for a given event. The internal traps are mutually exclusive and, therefore, no priority resolution is needed within that class of events. The other

exception conditions arise from external sources. If more than one device shares a common request line (the $\overline{\text{SEGT}}$ or an interrupt line), a request for service from more than one device using that line can occur simultaneously at the CPU. For segmentation traps where more than one MMU signals the trap, the Z8001 services the traps simultaneously, in that the acknowledge sent during the interrupt acknowledge sequence is accepted by all the MMUs (see Chapter 9). For interrupt requests, priority between multiple devices sharing an interrupt input to the CPU is resolved external to the CPU via a daisy-chain priority scheme. (This daisy chain is described later in the chapter.) Priority resolution among the daisy-chained peripherals is done during the interrupt acknowledge cycle.

INTERRUPT ACKNOWLEDGE

The interrupt acknowledge cycle is initiated by the CPU when responding to an interrupt request or segmentation trap. The acknowledge cycle serves two purposes—it selects the peripheral whose interrupt is to be acknowledged, and it reads an identifier word from the interrupting device. For vectored interrupts, the identifier includes the 8-bit interrupt vector that is used to determine the location of the service routine.

Interrupt acknowledge cycles are necessary only for exception conditions resulting from events external to the CPU, that is, interrupts and segmentation traps. Acknowledge cycles are not part of the response to an internally generated trap; since the cause of the trap is internal to the CPU, there is no need to acknowledge an external device or read an identifier word. (The timing of the interrupt acknowledge cycle is described later in the chapter.)

SAVING PROGRAM STATUS

Once the acknowledgment is complete, if one is necessary, the CPU must save enough status information about the program being executed at the time of the exception to be able to return successfully to that program after the service routine for the exception is completed. The results of executing a given programming task must be the same regardless of whether or not an interrupt or trap occurred during the task's execution. Therefore, the CPU must save all the information about the current task's running environment: the location of the next instruction to be executed, the operating modes of the CPU, and the flag conditions resulting from the last instruction's execution. This information is contained in the CPU's program status registers—the PC and FCW (Fig. 6.1). When responding to an interrupt or trap, the

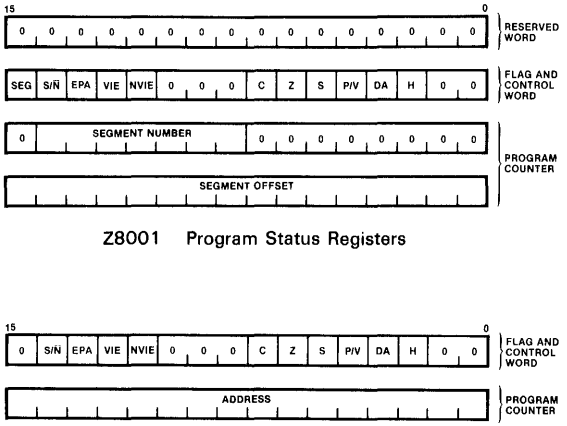


Figure 6.1 Program status registers.

CPU pushes the PC, FCW, and an identifier word onto the stack, using the system-mode implied stack pointer (system-mode RR14 in the Z8001, system-mode R15 in the Z8002). Status is saved on the system stack regardless of the CPU's operating mode at the time of the exception. For the Z8001, both the segment number and offset portions of the PC are saved on the stack, even if the CPU was in the nonsegmented mode when the exception condition occurred. In other words, the CPU always enters the system mode and, for the Z8001, the segmented mode during status saving in response to an interrupt or trap. The PC is pushed first (the offset portion first for the Z8001), followed by the FCW, and then the identifier word. For interrupts and segmentation traps, the identifier is the value read from the address/data bus during the interrupt acknowledge cycle. For internal traps, the identifier is the first word of the instruction that caused the trap. The format for the saved program status on the system stack is illustrated in Fig. 6.2. Table 6.1 lists the PC value that is pushed on the stack for each type of interrupt and trap.

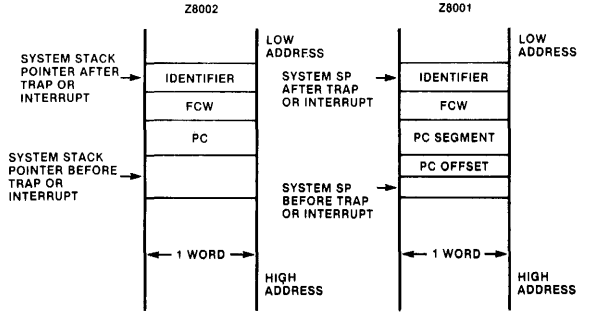


Figure 6.2 Saved program status on the system stack.

TABLE 6.1 PC VALUE SAVED ON STACK FOR EACH TYPE OF INTERRUPT AND TRAP

Exception	PC value is address of:
Extended instruction trap	Second word of instruction
Privileged instruction trap	Next instruction (single-word privileged instruction) Second word of instruction (multiple-word privileged instruction)
System call trap	Next instruction
Segment trap	Next instruction[a,b]
All interrupts	Next instruction[b]

[a]Assumes successful completion of instruction fetch.
[b]If executing an interruptable instruction (e.g., LDIR), the next instruction is the current instruction.

PROGRAM STATUS AREA

After saving program status for the task that was interrupted, the CPU loads in new program status values—a new PC and FCW, in other words—that define the CPU operating modes and starting address of the service routine. This new program status is loaded from a block of memory called the Program Status Area. Figure 6.3 is a diagram of the Program Status Area for the Z8001 and Z8002.

The starting address of the Program Status Area is determined by the contents of the Program Status Area Pointer (PSAP), which is a CPU control register (Fig. 6.4). The PSAP is loaded using the Load Control (LDCTL) instruction. The PSAP is a single word register in the Z8002 (to hold a 16-bit address) and two word registers in the Z8001 (to hold a segmented address). The low-order byte of the PSAP is always all 0's; therefore, the starting address of a Program Status Area is always on a 256-byte address boundary in memory. The Program Status Area contains a list of the program status values (that is, values for the PC and FCW) that are loaded into the CPU's PC and FCW registers during exception processing to determine the operating modes and starting location of each interrupt and trap service routine. In other words, after saving the program status of the task that was interrupted, the CPU fetches new program status values for the service routine by reading the appropriate memory locations in the Program Status Area pointed to by the PSAP.

The Program Status Area holds an FCW and PC value for the service routine for every possible type of interrupt or trap. The particular program status values fetched from the Program Status Area are a function of the type of exception that occurred and, for vectored interrupts, the vector returned by the peripheral during the interrupt acknowledge cycle. For each interrupt or trap there is, then, a block of memory in the Program Status

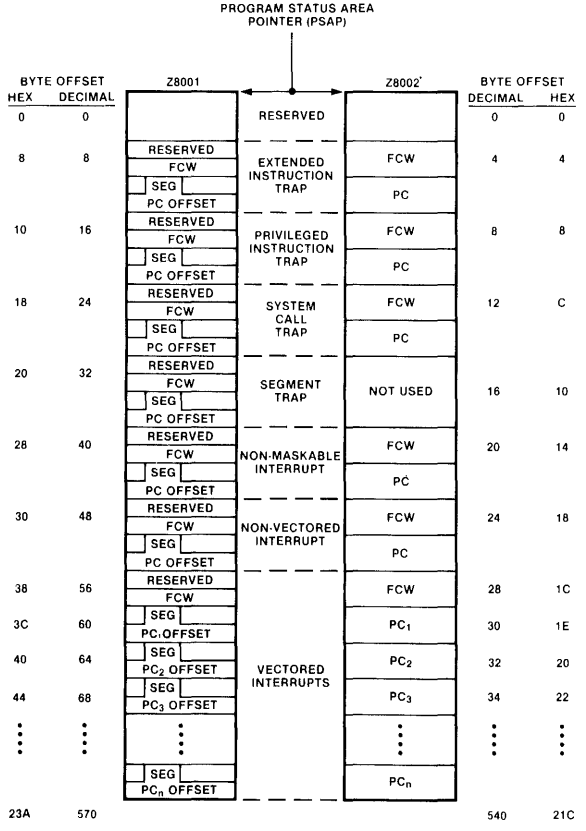
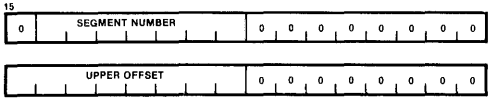


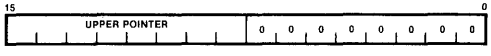
Figure 6.3 Program status areas for the Z8001 and Z8002.

Area that contains the values to be loaded into the FCW and PC to initiate execution of the appropriate service routine.

For the Z8002, the FCW and PC are each always one word (16 bits) long. Thus 4 bytes of memory are needed in the Program Status Area for the program status values for each type of exception. Starting at the location pointed to by the PSAP, the first two words of the Program Status Area are reserved. (Reserved areas are locations that are not used by the Z8002, but might be used by future upward-compatible Z8000 family processors.



Z8001 Program Status Area Pointer



Z8002 Program Status Area Pointer

Figure 6.4 PSAP register.

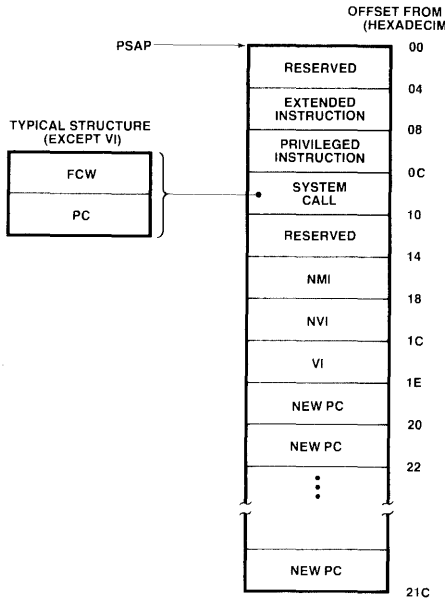


Figure 6.5 Z8002 Program Status Area.

These areas should be filled with all 0's.) The next two words, starting at address 4 in the Program Status Area, are the FCW and PC value for the extended instruction trap service routine. This is followed by the FCW and PC values for the privileged instruction trap and system call service routines, a reserved area of 4 bytes, and then FCW and PC values for the nonmaskable interrupt and nonvectored interrupt service routines, respectively (Fig. 6.5). For vectored interrupts, an FCW value that will be loaded for all vectored interrupts is at location 1C (hexadecimal) in the Program Status Area. This is followed by a list of 256 possible PC values, corresponding to the 256 possible 8-bit vectors that can be returned by the interrupting peripheral during the interrupt acknowledge cycle. The vector returned by the peripheral is used as an index into this list to select the starting PC value for the interrupt service routine; a vector of 0 corresponds to the PC value at location 1E (hex) in the Program Status Area, a vector of 1 corresponds to the PC value at location 20 (hex), and so on up to a vector value of 255 (FF hex). Thus the value of the vector will determine which PC value is fetched when loading the program status registers for the service routine from the Program Status Area.

The program status registers for the Z8001 consist of a reserved word, the FCW, and the PC (Fig. 6.1). The program counter is two words long, where one word holds the segment number and the other holds the offset address. Thus four words are needed in each block of the Z8001's Program Status Area to hold the program status values for each interrupt and trap service routine. The first four words of the Program Status Area for the Z8001

are reserved (Fig. 6.6). The next block of four words holds the program status values for the extended instruction trap service routine. The first word of this block is a reserved word, followed by an FCW value, the segment-number portion of the PC value, and finally, the offset portion of the PC value. (When segment numbers are stored in a 16-bit word, the 7-bit segment number is placed in bits 8–14. Bit 15 and bits 0–7 are set to 0.) The following blocks in the Program Status Area contain the program status values for the privileged instruction trap, system call, segmentation trap, nonmaskable interrupt, and nonvectored interrupt service routines, in that order. Each of these blocks has the same format: a reserved word, the FCW value, the PC's segment number, and the PC's offset for that service routine. Location 3A (hex) of the Z8001's Program Status Area holds the FCW value for all of the vectored interrupt service routines. It is followed by a list of 128 possible PC values, with each PC having a segment number and an offset part. Thus each of the 128 possible PC values fills two words in the Program Status Area. The 8-bit vector returned by the peripheral during the interrupt acknowledge cycle is used as an index into this list of possible PC values. A vector of 0 will select the first PC value, whose segment number is at location 3C (hex) and whose offset is at location 3E (hex) of the Program Status Area. If the vector is 2, the next PC value will be selected, and so on. Only even-valued vectors are used in the Z8001 (that is, the least significant bit of the vector must be a 0); therefore, there are 128 unique vectors that can be returned by the peripheral, corresponding to the 128 PC values listed in the Program Status Area.

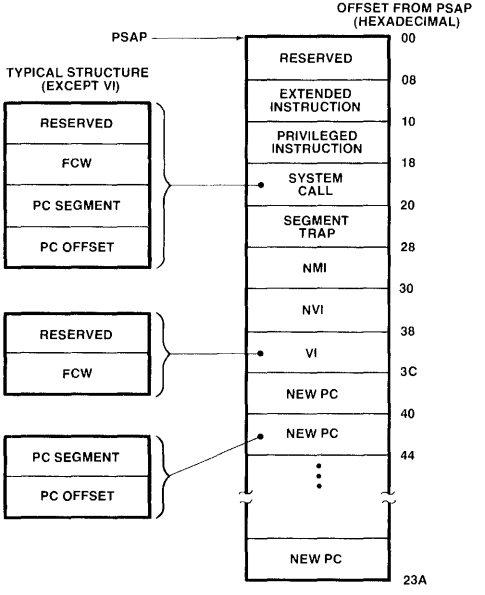


Figure 6.6 Z8001 Program Status Area.

The Program Status Area can begin at any 256-byte boundary in memory (any memory location with an offset address whose low-order byte is 0) except for address 0 in a Z8002 system and segment 0, address 0 in a Z8001 system. (This restriction is due to the method in which the CPUs respond to a reset, as described later in the chapter.) When new program status is fetched during the response to an exception condition, the fetch is made in system-mode operation with a status code of 1100 (program reference) on the ST3-ST0 lines. Thus the Program Status Area always resides in the system-mode program memory address space.

In review, when an exception condition occurs, the CPU will save the program status (FCW and PC) of the task that was interrupted, and use the PSAP to determine the starting location of the Program Status Area. The CPU will then fetch new values for the program status from the Program Status Area; these values are loaded into the FCW and PC registers in the CPU, and execution of the service routine is begun. The new value of the FCW is not effective until the start of the fetch of the first instruction of the service routine; thus the new FCW value will not affect the status pins while the fetches from the Program Status Area are being completed. The first instruction of the service routine will be fetched from the address specified in the PC that was loaded from the Program Status Area. The service routine will be running in the operating modes (segmented or nonsegmented, system or normal, EPA and interrupts enabled or disabled) specified by the FCW value loaded from the Program Status Area.

If desired, maskable interrupts can be disabled at the start of a service routine by a suitable choice of the FCW's value. This would allow critical information to be stored or processed before subsequent interrupts are handled. If interrupts are enabled before the end of the service routine, nested interrupts are possible, that is, the service routine could itself be interrupted while another interrupt's service routine is run.

Several different Program Status Areas could be set up in memory, with different ones being used at various times during program execution. Specifying a new Program Status Area would involve only changing the PSAP register contents with a Load Control (LDCTL) instruction. For the Z8001, changing the PSAP takes two LDCTL instructions—one for writing the segment number and the other for writing the offset. Care must be taken to ensure that no exceptions occur between the two instructions, since an unintended PSAP value could be in effect at that time.

INTERRUPT RETURNS

A service routine is program code that is executed in response to an interrupt or trap. In the Z8000 CPUs, execution of a service routine starts after new program status has been loaded into the PC and FCW from the Pro-

gram Status Area. Upon completion of the service routine, execution of the task that was interrupted by the exception is resumed. In order to terminate the service routine and return to the interrupted task, that task's program status must be retrieved from the system stack, where it was stored before the service routine was run. The Z8000's Interrupt Return (IRET) instruction is used to end routines for both interrupts and traps.

Execution of the IRET in a Z8002 automatically pops three words from the system stack: the identifier word is popped and discarded (that is, the stack pointer is incremented without saving the identifier word), and the interrupted task's FCW and PC values are popped and loaded into the CPU's program status registers. After the IRET, the next instruction fetch will be from the location addressed by the PC, which is the point at which the task was interrupted. IRET is a privileged instruction that can be executed only in system mode; IRET always pops the old program status from the system-mode stack memory address space. The "popped" value of the FCW is not effective until the next instruction fetch, so the status pins will not be affected by the new FCW value until execution of the IRET is completed.

In Z8001 CPUs, execution of the IRET automatically pops four words from the system-mode stack—the identifier, FCW, PC segment number, and PC offset—since four words are always pushed on the stack when servicing an exception. Therefore, the Z8001 must be in segmented mode when the IRET is executed. It is the programmer's responsibility to ensure that the Z8001 is in the segmented mode before any IRET instruction is encountered.

PERIPHERAL INTERRUPT DAISY CHAIN

The Z8000 CPUs support three types of interrupts: nonmaskable, nonvectored, and vectored. Each of these interrupts can have multiple sources in a Z8000 system, with several peripherals sharing a common interrupt input to request servicing from the CPU. When peripherals share a common interrupt line, a method of prioritizing interrupt requests from these peripherals is needed. This prioritization is implemented by means of a daisy chain external to the CPU. The interrupt daisy chain is formed using two signals at each peripheral device: Interrupt Enable In (IEI), an input to each peripheral, and Interrupt Enable Out (IEO), an output from each peripheral. The interrupt daisy chain has two functions: during interrupt acknowledge transactions, it determines which interrupt source on the daisy chain is being acknowledged, and at all other times, it determines which interrupt sources can initiate an interrupt request.

Figure 6.7 shows an example of the interrupt daisy-chain structure. Four peripheral devices share a common interrupt request line to the CPU, where any device can pull the line low to make a request. This signal might be the CPU's nonmaskable, nonvectored, or vectored interrupt input. The

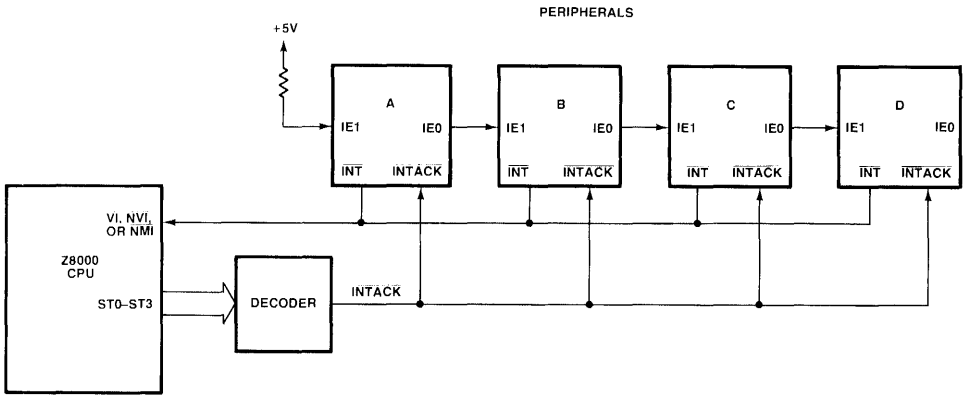


Figure 6.7 Interrupt daisy chain with four devices sharing a common interrupt line.

appropriate acknowledge signal, decoded from the ST0–ST3 status lines, is connected to each peripheral's interrupt acknowledge input. The highest-priority device has its IEI line tied high; its IEO output is connected to the next device's IEI input, and so on down the daisy chain. The order in which the peripherals are connected on the daisy chain determines their relative priority; if two devices on the daisy chain simultaneously assert an interrupt request, the higher-priority device will be acknowledged first. Furthermore, a given peripheral device is not allowed to interrupt the service routine of a higher-priority device on the daisy chain.

A high level (logical 1) on a peripheral's IEI input means that the peripheral is free to request an interrupt by pulling the interrupt line to the CPU low. The IEO output is used by each peripheral to allow or stop interrupt requests from devices with lower priority on the daisy chain.

The interrupt protocol on the daisy chain is illustrated in the state diagram of Fig. 6.8. In the quiescent state (no interrupts being asserted or serviced), each device on the chain passes its IEI input to its IEO output. Thus each device sees a high level on its IEI input and is free to request servicing by asserting an interrupt. When a peripheral does need servicing, it can interrupt the CPU by pulling the interrupt line low only if that device's IEI input is high and no interrupt acknowledge cycle is taking place (as signaled by the appropriate status at the last rising edge of $\overline{AS}$). When a device asserts an interrupt, its IEO line is not pulled low; IEO continues to follow IEI until the interrupt is acknowledged.

Sometime after the peripheral pulls the interrupt request line low, the CPU will respond with an interrupt acknowledge cycle. The delay between the interrupt request and the acknowledge will depend on the length of the current instruction being executed; interrupt inputs are sampled during the last machine cycle of every instruction. When the interrupting peripheral

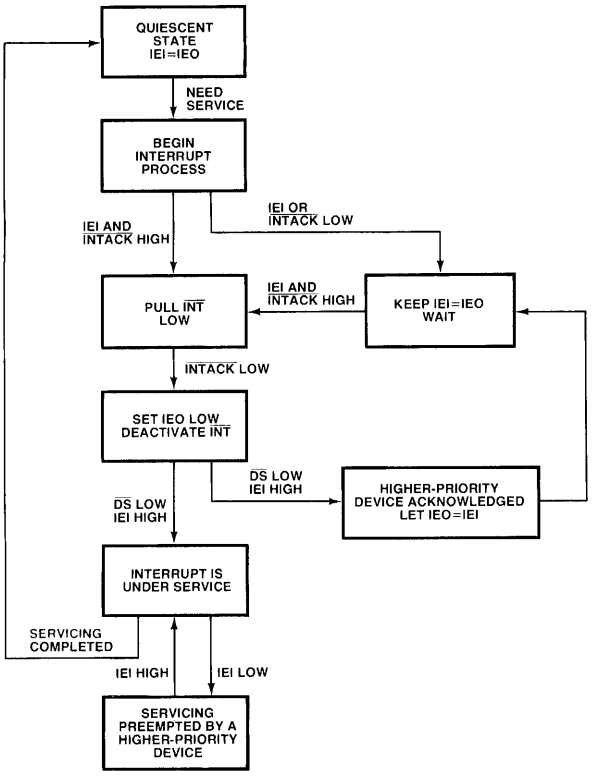


Figure 6.8 Interrupt daisy-chain protocol.

sees an active acknowledge signal (the status lines are guaranteed valid at the rising edge of $\overline{AS}$ during T1 of the acknowledge cycle), it pulls its IEO output low and releases the interrupt request line. When $\overline{DS}$ goes low later in the acknowledge cycle, the peripheral examines its IEI input. If IEI is high, the device accepts the acknowledge, and, if desired, places an identifier on the bus (a vector in the case of vectored interrupts) before $\overline{DS}$ rises. The device is now under service.

While the service routine for a peripheral is being executed, that peripheral will continue to hold its IEO output low, thereby preventing lower-priority devices on the daisy chain from requesting service via an interrupt. However, higher-priority devices can still assert interrupts; a peripheral knows that a higher-priority device on the daisy chain has preempted its service routine by asserting an interrupt if its IEI input goes low. When the servicing of a peripheral is completed, the peripheral returns to the quiescent state, wherein its IEO output follows its IEI input. [For Z8000 family peripherals, the CPU signals the peripheral that service is complete by resetting a bit called the interrupt-under-service bit in one of the peripheral's internal registers. This requires an I/O write to the peripheral at the end of the service

routine (that is, immediately before the interrupt return), as described in Chapter 12.]

For example, suppose that peripheral device C in Fig. 6.7 requires servicing from the CPU. If device C's IEI and $\overline{\text{INTACK}}$ inputs are both high, it can request the CPU's attention by pulling the interrupt line low. Sometime later, when the CPU enters the acknowledge cycle, device C will receive an active $\overline{\text{INTACK}}$ signal. Device C now pulls his IEO output low. When $\overline{\text{DS}}$ goes low, device C samples its IEI input. If IEI is low, then either device A or device B also asserted an interrupt before this acknowledge cycle began, saw the acknowledge, and pulled its IEO low; in this case, the higher-priority device accepts the acknowledge, and device C should let IEO follow IEI, wait until its IEI is high again, and then reassert the interrupt request. If device C's IEI input is high when $\overline{\text{DS}}$ goes low during the acknowledge cycle, device C is the highest-priority device requesting service, and is free to place an identifier or vector on the bus during this acknowledge cycle. While device C is being serviced, its IEO line is held low, preventing device D from making an interrupt request.

For this daisy-chain protocol to work properly, the delay in the interrupt acknowledge cycle between the rising edge of $\overline{\text{AS}}$ (acknowledge status valid) and the falling edge of $\overline{\text{DS}}$ (a device samples its IEI and accepts the acknowledge) must be long enough to allow a change in the IEO from the highest-priority device to propagate to the IEI of the lowest-priority device on the daisy chain. In other words, the IEI/IEO daisy chain must settle completely between the rising edge of $\overline{\text{AS}}$ and the falling edge of $\overline{\text{DS}}$ during the acknowledge cycle. For long daisy chains, this may require the addition of externally-generated wait states in the acknowledge cycle.

This type of priority arrangement of the peripherals via a hardware connection is called an interrupt-under-service daisy chain; devices on the chain are not permitted to request interrupts if a higher-priority device is being serviced. No separate priority control devices are needed; the priority of a given peripheral is determined solely by its physical position on the daisy chain. Four signals are required to implement the daisy chain for each Z8000 interrupt type: $\overline{\text{INT}}$, $\overline{\text{INTACK}}$, IEI, and IEO. These four lines can be replicated for each of the three interrupt types supported by the Z8000 CPU.

INTERRUPT ACKNOWLEDGE CYCLE

The interrupt acknowledge cycle is entered in response to an externally generated exception condition, that is, an interrupt or segmentation trap. This acknowledge cycle is used to identify the highest-priority device on an interrupt-under-service daisy chain, as described above, and to allow the CPU to receive an identifier word, which is saved on the stack with the program status of the interrupted task.

Figure 6.9 shows the timing of the interrupt acknowledge machine

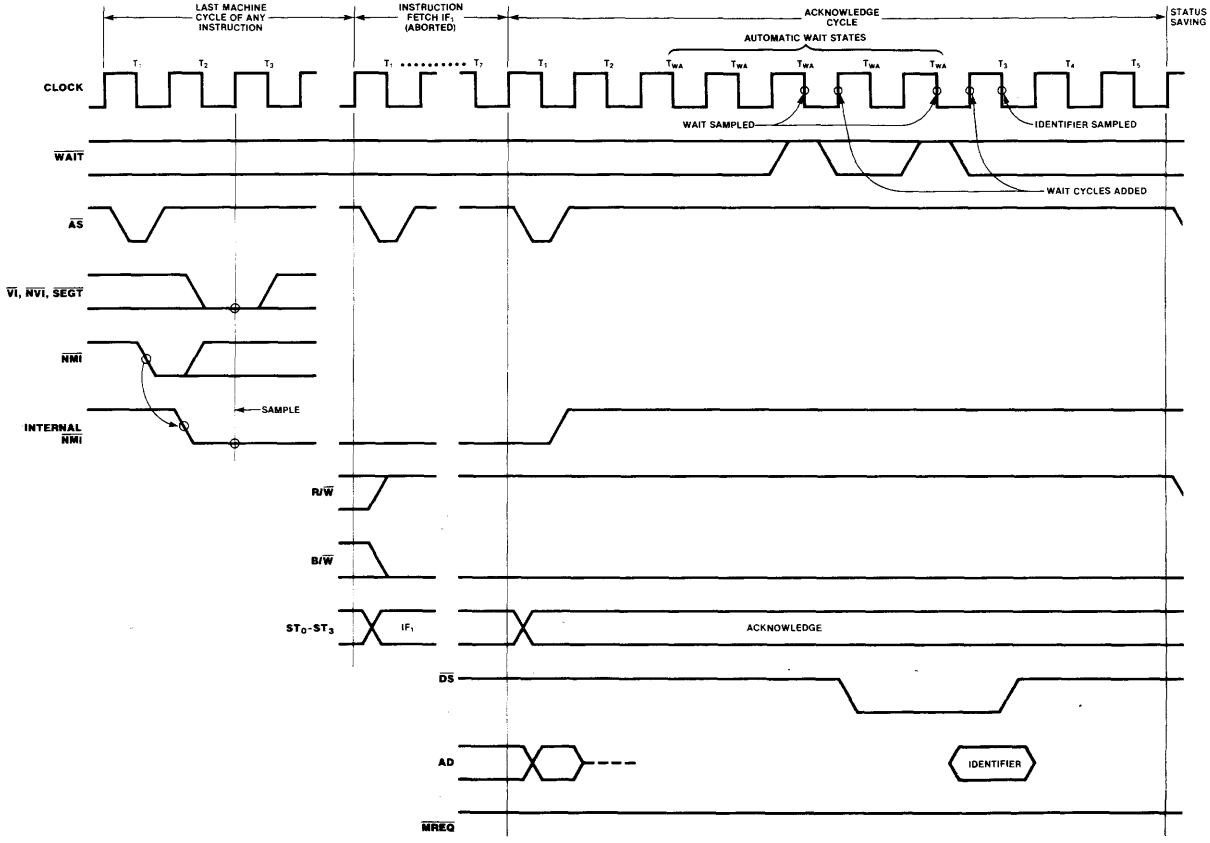


Figure 6.9 Interrupt acknowledge timing.

cycle. The interrupt and segmentation trap inputs to the CPU are sampled at the beginning of T3 of the last machine cycle of each instructions' execution. The $\overline{VI}$, $\overline{NVI}$, and $\overline{SEGT}$ inputs are level triggered; that is, the input pin must be low when sampled for the CPU to recognize the interrupt or trap request. The $\overline{NMI}$ is edge triggered; a negative edge on this input sets an $\overline{NMI}$ flip-flop internal to the CPU. This flip-flop is examined to determine if an $\overline{NMI}$ request is to be serviced. (Setup and hold-time requirements for these inputs are given in the AC timing characteristics chart in Appendix A.)

If an interrupt or trap condition is detected, the Z8000 will start the next instruction fetch, but this fetch will be abandoned before completion. All of the proper address and status information for the next instruction fetch will appear in T1, and the $\overline{AS}$ is pulsed. However, no active $\overline{DS}$ signal is generated, the instruction is never read from memory, and the PC is not updated. The CPU spends seven clock periods in this abandoned instruction fetch cycle; during this time, the CPU is resolving priority among competing events if more than one exception condition was detected, and decrementing the implied system stack pointer in preparation for saving the interrupted task's program status.

This abandoned instruction fetch cycle is followed by the actual interrupt acknowledge cycle. The CPU always switches to the system mode, and, for the Z8001, the segmented mode at the start of the acknowledge cycle. The CPU remains in these modes until it begins to execute the service routine. During T1 of the acknowledge cycle, all status signals, including the appropriate acknowledge code on the ST0–ST3 lines (Table 6.2), are output and guaranteed valid at the rising edge of $\overline{AS}$. The contents of the address/data bus are undetermined at this time; that is, no meaningful address information is output. $\overline{MREQ}$ is high, B/$\overline{W}$ is low, and R/$\overline{W}$ is high, indicating that the CPU is going to read a word from the interrupting device later in this cycle. During T2, the address/data bus is tri-stated by the CPU in anticipation of reading data placed on the bus by an external device in T3. Five wait states are automatically included in the interrupt acknowledge cycle. Assuming there are no externally-generated wait states, $\overline{DS}$ is lowered during the fourth wait state. The IEI/IEO daisy chain should be settled by this time; for Z8000 family peripherals, this normal timing would allow chains of about 10 devices. After the fifth automatic wait state, the T3 state is en-

TABLE 6.2 STATUS CODES FOR INTERRUPT AND TRAP ACKNOWLEDGMENTS

ST3–ST0	Acknowledge type
0100	Segment trap acknowledge
0101	Nonmaskable interrupt acknowledge
0110	Nonvectored interrupt acknowledge
0111	Vectored interrupt acknowledge

tered. The contents of the address/data bus are read by the CPU on the falling edge of the clock in the middle of T3. T4 and T5 clock periods are appended to this transaction, and are used internally to load the identifier read off the bus into a temporary storage register in the CPU. Thus the interrupt acknowledge machine cycle is always at least 10 clock cycles long.

The word of data read during T3 of the acknowledge cycle is the identifier word that is later saved on the system stack as part of the exception processing. Thus an interrupting device can send up to 16 bits of status information to the CPU as part of the acknowledge cycle. For vectored interrupts, the CPU uses the lower half of this word (the bits returned on the AD0-AD7 bus lines) as the interrupt vector. In the Z8001, the CPU will always set bit 0 of the vector to 0, regardless of the value returned during the acknowledge cycle, thereby guaranteeing that only even vector values are used when indexing into the Program Status Area. If several devices are connected on a daisy chain for the $\overline{\text{NMI}}$ or $\overline{\text{NVI}}$ interrupt inputs, the identifier word typically is used in the service routine to identify the device that asserted the interrupt. Of course, the peripheral does not have to return an identifier word during the acknowledge, except for vectored interrupts, where a vector is always required.

Additional wait states can be added to the interrupt acknowledge cycle by pulling the CPU's $\overline{\text{WAIT}}$ input low. During the middle of the third automatic wait state (before $\overline{\text{DS}}$ goes low), the $\overline{\text{WAIT}}$ input is sampled; if $\overline{\text{WAIT}}$ is active, an additional wait state is added before starting the clock period wherein $\overline{\text{DS}}$ goes low (Fig. 6.10). During that additional wait state, the $\overline{\text{WAIT}}$ input is again sampled, and further additional wait states are added until $\overline{\text{WAIT}}$ returns high. Therefore, the delay between the rising edge of $\overline{\text{AS}}$ and the falling edge of $\overline{\text{DS}}$ during the interrupt acknowledge can be arbitrarily long, allowing time for long IEI/IEO daisy chains to settle. Once $\overline{\text{DS}}$ goes low and the highest priority device that requested an interrupt has been selected, the $\overline{\text{WAIT}}$ input is again sampled during the last automatic wait state before T3. Additional wait states can be inserted here to allow the selected device additional time to place its identifier word or vector on the bus. Thus

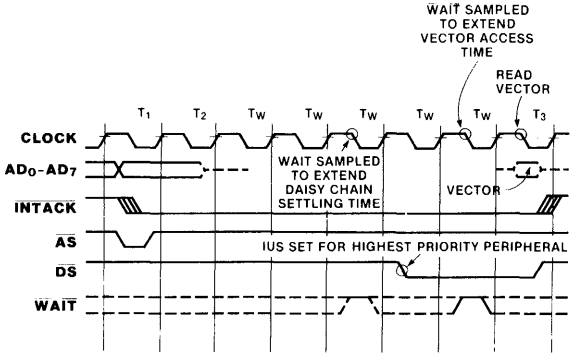


Figure 6.10 Wait states in the interrupt acknowledge cycle.

externally-generated wait states can be added at two points in the interrupt acknowledge cycle—before $\overline{DS}$ goes active, to allow time for long daisy chains to settle, and during $\overline{DS}$ active, to allow time for the selected device to place an identifier on the bus (Fig. 6.10).

Immediately following the acknowledge cycle, the CPU executes several memory access cycles. The program status information for the interrupted task is pushed onto the system stack in the following order: the PC offset, the PC segment number (Z8001 only), the FCW, and the identifier word. The new program status is read from the Program Status Area pointed to by the PSAP and loaded into the CPU's program status registers. All of these accesses are made in the system mode and, for the Z8001, the segmented mode. Then execution of the service routine is begun.

INTERRUPT RESPONSE TIME

Interrupt response time in a Z8000 system is dependent on the length of the instruction being executed when the interrupt occurs. If the interrupt occurs immediately after interrupts are sampled during the last machine cycle of an instruction, another complete instruction must be fetched and executed before interrupts are sampled again. Thus the maximum possible delay between an interrupt and its servicing depends on the longest instruction executed during interruptible portions of all programs in the system. Some Z8000 instructions can take relatively long times to execute; for example, the Divide Long (DIVL) instruction can take up to 749 clock cycles to execute in segmented mode. In systems where interrupt response time is critical, instructions such as DIVL should be avoided. (The Z8000 has several assembly language instructions that automatically repeat a given process for a set number of times, such as the block move instructions. These instructions are interruptible between each iteration of the instruction's execution.)

Once the interrupt has been sampled and the current instruction has completed, the remaining response time can be calculated. Table 6.3 lists the number of CPU clock cycles needed for each step of interrupt processing, excluding any externally-generated wait states during the interrupt acknowledge and memory access machine cycles. Thus the Z8001 takes at least 44 clock cycles from the end of the instruction being executed when the interrupt was sampled until the beginning of the fetch of the first instruction of the service routine. For the Z8002, at least 38 clock periods are required. Servicing of internally-generated traps (extended instruction, privileged instruction, and system call) requires 10 fewer clocks; the timing is similar except that the interrupt acknowledge cycle is not included in the response to an internal trap condition.

TABLE 6.3 TIME REQUIRED FOR INTERRUPT AND SEGMENTATION TRAP PROCESSING

Event	CPU clock periods[a]
Aborted instruction fetch cycle	7
Interrupt acknowledge cycle	10
Push PC offset on system stack	4
Push PC segment number[b]	3
Push FCW	4
Push identifier word	7
Fetch new FCW from Program Status Area	3
Fetch new PC segment number[b]	3
Fetch new PC offset	3

[a]Assumes no externally-generated wait states.
[b]Z8001 only.

SYSTEM CALL INSTRUCTION

Execution of the System Call (SC) instruction causes the CPU to process the system call trap. This instruction allows normal-mode programs to force execution of the service routine for the system call trap. This service routine could be a part of the operating system that performs system-mode type operations such as I/O for the normal-mode user.

The opcode for the SC instruction is one word long, where the upper byte is an EF (hex) and the lower byte is an immediate operand from 0 to 255 as specified by the programmer (Fig. 6.11). The entire SC opcode appears on the system stack as the identifier word during status saving for the system call stack (Fig. 6.12). The system call trap routine can read the iden-

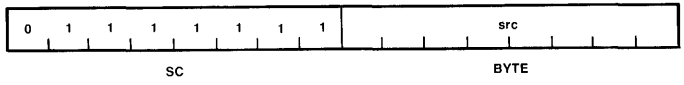


Figure 6.11 System call opcode.

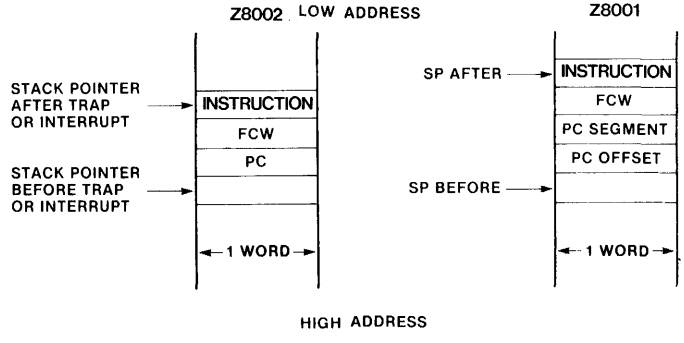


Figure 6.12 Saved program status for a system call trap.

tifier from the stack, and use it to define the action to be taken. In other words, a byte of data can be passed from the normal-mode program to the system call service routine in the opcode of the SC instruction itself. The service routine might use this byte parameter to determine the type of action that the normal-mode user is requesting via the system call. Thus the operating system could be written so that normal-mode users can use the system call to request any one of up to 256 actions available in the system call service routine.

SERVICE ROUTINES

When executing service routines, care must be taken not to change memory or register locations that, if altered, could prevent a successful return from the interrupt or trap. Before entering the service routine, the program status of the interrupted task is placed on the system stack using the implied stack pointers (R15 in the nonsegmented mode, RR14 in the segmented mode). The Interrupt Return (IRET) instruction is used to clear this information on the stack and return to the interrupted task at the end of the service routine. Therefore, the implied stack pointer must have the same value when the IRET is executed as it had when the service routine began. If the identifier word on the stack is to be accessed in the service routine via a pop of the stack, a word must be pushed back onto the stack to restore the stack pointer to its starting value. (The IRET must be executed in the segmented mode in the Z8001, as discussed previously.) Of course, the interrupted task's program status information on the stack cannot be altered if the interrupt task is to resume precisely where it was interrupted. Registers and memory locations that were being used by the interrupted task should not be unintentionally altered by the service routine. In short, execution of the interrupted task should produce the same results as if the task were not interrupted.

Often, a service routine will load all of the registers' contents into memory locations reserved for that purpose at the start of the service routine, use the registers, and then restore the original register contents before returning to the interrupted task. The Load Multiple (LDM) instruction in the Z8000 allows the user to store all of the general-purpose registers' contents into consecutive memory locations or fill all the registers from consecutive memory locations with one instruction.

If the system distinguishes between system-mode and normal-mode memory address spaces, the operating mode of the service routine will determine which memory areas can be accessed by that routine. If the service routine needs to read the identifier word from the system stack, that part of the routine will have to operate in the system mode. Segregated memory address spaces can affect parameter passing to a service routine. When a normal-mode program executes a system call, for example, 1 byte of data is

passed in the identifier word, as described above. If more data need to be passed in memory locations (such as a file to be output to a peripheral by the system call service routine), that memory would have to be accessible by both the normal-mode program, which sets up the data and makes the system call, and the system-mode service routine, which is to read the data. Therefore, for such a scheme to work, the system- and normal-mode memory address spaces must have some common memory locations (see Chapter 3).

HALT INSTRUCTION

The HALT instruction is a privileged instruction that suspends CPU operation until an exception condition occurs. Execution of a HALT puts the CPU in a continuous string of three-clock-period internal operation machine cycles (Fig. 6.13). (If automatic memory refresh is enabled, refresh cycles

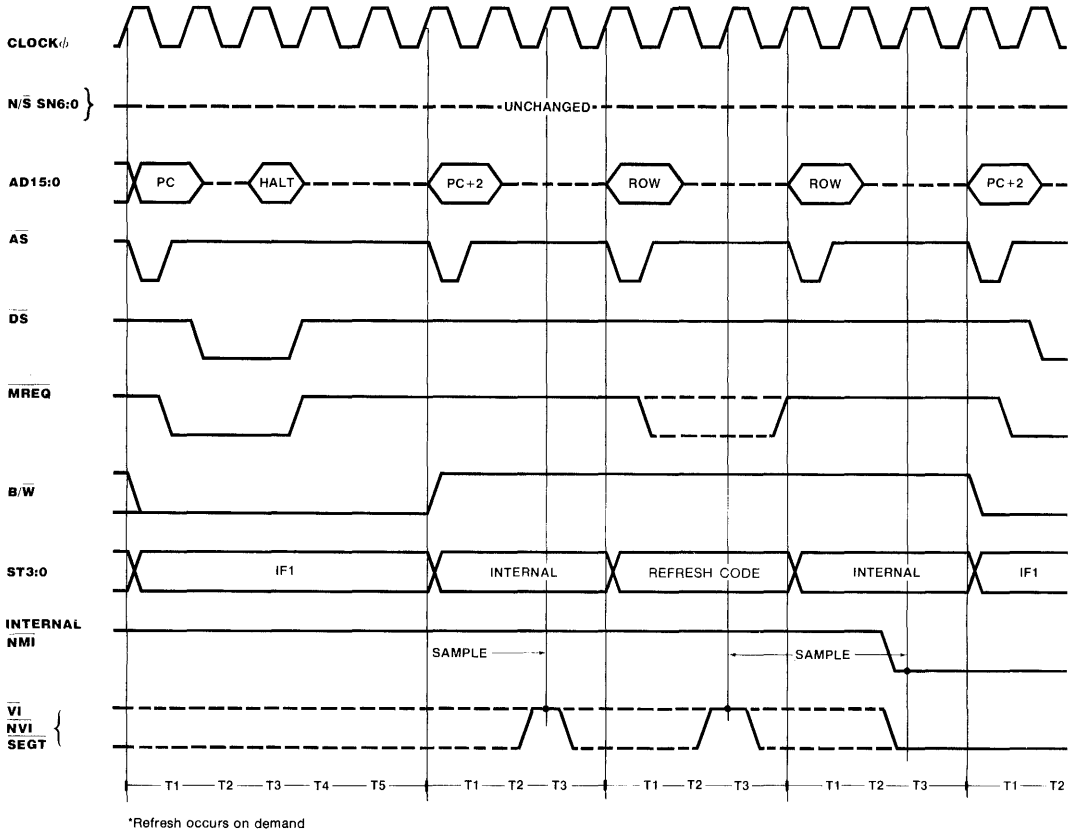


Figure 6.13 HALT instruction timing.

will be inserted as often as specified by the refresh rate counter portion of the refresh register.) The CPU remains in this state until an interrupt, segmentation trap, or reset occurs. After the exception is serviced, execution will resume with the instruction following the HALT. Typically, the HALT instruction is used to synchronize CPU operation with external events.

Execution of the HALT command consists of a five-clock instruction fetch cycle, followed by successive three-clock-period internal cycles (with refresh cycles inserted as dictated by the refresh register). At the start of T3 of each cycle, the interrupt and $\overline{\text{SEGT}}$ inputs are sampled. If an interrupt or $\overline{\text{SEGT}}$ is detected, the exception is processed in the normal fashion, with an acknowledge cycle, status saving, fetching of new program status, and execution of the service routine. The value of the PC that is saved during program status saving is the address of the instruction following the HALT. Thus the return from the service routine will return to the first instruction after the HALT. In this manner, the HALT can be used to stop program execution and resume it again based on some externally generated input.

HALT instructions also can be used to synchronize Z8000 CPUs in a multiple-processor system. The HALT instruction would be placed at the point in each processor's code where synchronization is desired. Before each HALT, an I/O instruction could be used to inform external hardware that the HALT has been reached. When all the processors have reached their HALT instruction, the external logic generates a pulse on an interrupt input to each processor, and they all resume program execution.

RESET

A hardware reset occurs at the end of any clock cycle when the $\overline{\text{RESET}}$ input to the CPU is low. The $\overline{\text{RESET}}$ input must be held low for at least five CPU clock cycles to initialize the CPU properly. A reset overrides all other considerations, including interrupts, traps, bus requests, and $\overline{\text{STOP}}$ inputs. Reset should be used to initialize the CPU as part of the system's power-up sequence.

Reset timing for the Z8002 is illustrated in Fig. 6.14. Within five clock cycles after $\overline{\text{RESET}}$ goes low, the address/data bus is tri-stated by the CPU; the $\overline{\text{AS}}$, $\overline{\text{DS}}$, $\overline{\text{MREQ}}$, $\overline{\text{MO}}$, $\overline{\text{BUSACK}}$, and ST0–ST3 signals are forced high; the SN0–SN6 segment number output goes low, and the R/$\overline{\text{W}}$, B/$\overline{\text{W}}$, and N/$\overline{\text{S}}$ lines are undefined (that is, they might be high or low). Three clock periods after $\overline{\text{RESET}}$ returns high, memory read cycles are executed in the system mode and the values read during those read cycles are loaded into the program status registers.

For the Z8002, two read cycles are executed; the first read fetches a word from memory location 2 (that is, 2 is the address emitted during T1 of the memory read) and loads it into the FCW, and the second read fetches a

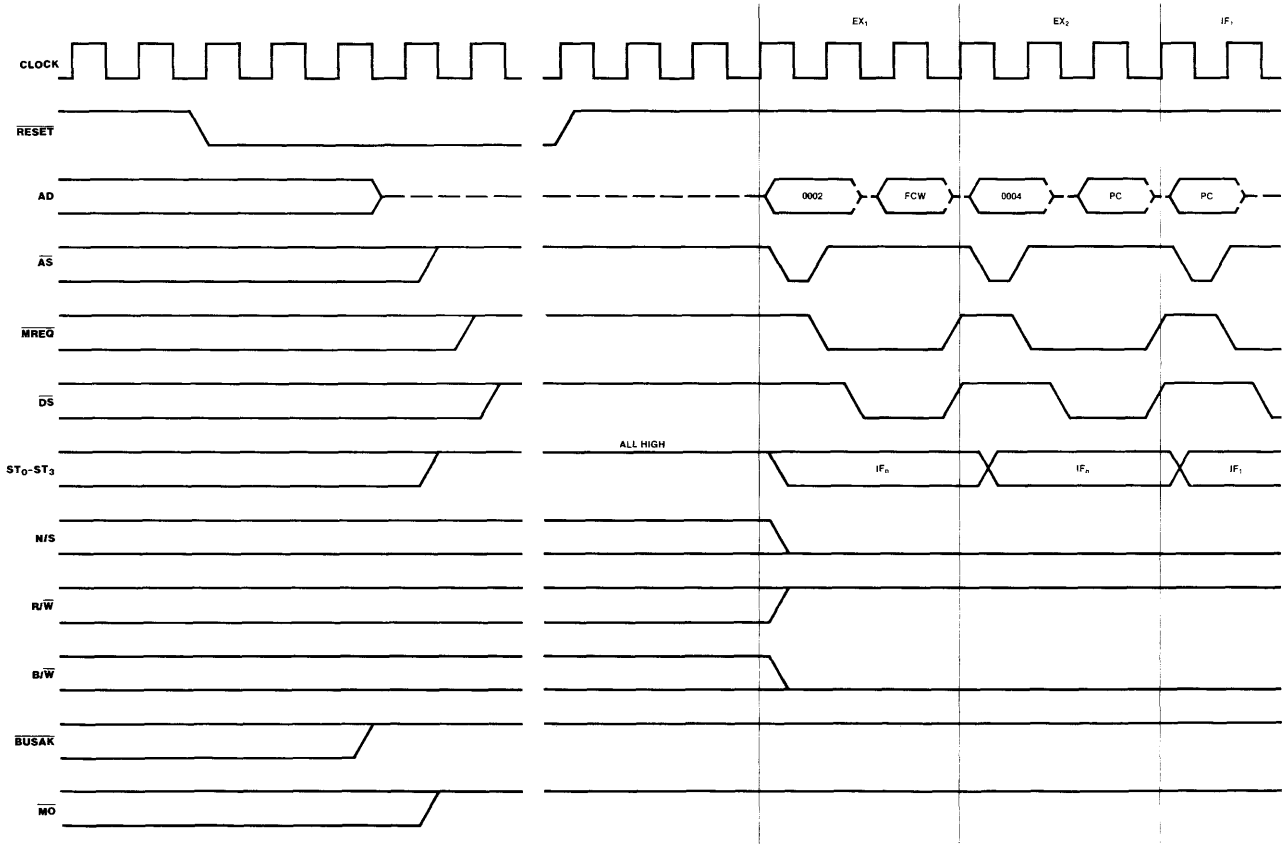


Figure 6.14 Z8002 reset timing.

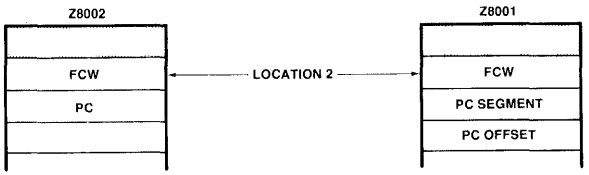


Figure 6.15 Memory locations accessed during a reset.

word from location 4 and loads it into the PC. These FCW and PC values are the program status for the first routine to be executed after the reset. Figure 6.14 shows the timing of these fetches, assuming that no active $\overline{\text{WAIT}}$ signals are generated during these memory accesses. The next machine cycle following these two reads will be an instruction fetch from the address loaded into the PC, under the operating modes specified in the FCW.

For the Z8001, three memory reads are executed as part of the reset sequence; the starting FCW is fetched from segment 0, location 2, the segment number for the initial PC value is fetched from segment 0, location 4, and the PC's offset is fetched from segment 0, location 6 (Fig. 6.15). The following machine cycle is the instruction fetch that starts the execution of the program at the address loaded into the PC, under the operating modes specified in the FCW.

These initial fetches of the FCW and PC values after a reset are made from the system-mode program memory address space (1100 status on ST0-ST3). These values in memory must be present at power-up, and, therefore, are implemented in nonvolatile memory (ROM or PROM). Since the first six memory locations for the Z8002 and the first eight locations in segment 0 for the Z8001 are dedicated to holding the program status values for a reset condition, these areas cannot be used as part of a Program Status Area. In other words, the Program Status Area cannot start at address 0 in a Z8002 system or segment 0, address 0 in a Z8001 system.

After a reset, the contents of all the CPU registers are undefined, except for the PC and FCW, and bit 15 of the refresh register. Bit 15 (the enable bit) of the refresh register is cleared to 0 by a reset, thereby turning off the automatic memory refresh mechanism. All interrupt and segmentation trap inputs are ignored during the reset processing, and any pending $\overline{\text{NMI}}$ requests are cleared (that is, the internal $\overline{\text{NMI}}$ request flip-flop is cleared). Interrupts are not sampled until the last machine cycle of the first instruction executed after the reset.

INITIALIZATION ROUTINES

After a reset, the first software program executed should be a routine that initializes the CPU control registers used in that system. The implied stack pointers (RR14 in the segmented mode, R15 in the nonsegmented mode)

must be initialized before the system processes interrupts, traps, or subroutine calls. The Program Status Area Pointer (PSAP) and Program Status Area must be initialized before interrupts or traps can be handled properly. (Often, Program Status Areas are implemented in nonvolatile ROM or PROM, so initialization is not required each time a reset occurs.) If automatic memory refreshes are needed for dynamic memories in the system, the rate and enable portions of the refresh register must be initialized.

An important practical consideration for Z8000 system initialization is the potential for a nonmaskable interrupt request shortly after a reset. Since the $\overline{\text{NMI}}$ cannot be disabled, the system must be ready to handle an $\overline{\text{NMI}}$ at any time. This is not possible, of course, since a finite amount of time is needed after a reset to initialize the implied stack pointer and PSAP, even if the Program Status Area is in ROM. An active $\overline{\text{NMI}}$ input before these initializations would be mishandled. Therefore, hardware external to the CPU is needed to delay $\overline{\text{NMI}}$ requests until after initializations are completed. This is true for any processor having a nonmaskable interrupt.

OTHER CONTEXT SWITCHES

Other Z8000 instructions besides the System Call can cause "context switches," in that they affect the program status registers. The Load Control (LDCTL) instruction can be used to load the FCW, thereby changing the CPU's operating modes. The Disable Interrupt (DI) and Enable Interrupt (EI) instructions change just the vectored and nonvectored interrupt enable bits in the FCW.

Complete context switches can be made with the Load Program Status (LDPS) instruction. This instruction loads a new FCW and PC from an area in memory that is formatted in the same manner as one block of the Program Status Area. In the nonsegmented mode, an FCW and PC value are loaded from memory; in the segmented mode, a reserved word, FCW, PC segment number, and PC offset address are loaded from memory (Fig. 6.16). These fetches are made from the data memory address space (1000 status on

Figure 6.16 Format of memory locations read during LDPS instruction execution.

ST0-ST3). The new value of the FCW does not become effective until the next instruction, so the status pins will not be affected by the new control bits until the LDPS instruction execution is completed. The next instruction executed is the instruction addressed by the new PC value.

Old program status is not saved by the LDPS instruction, meaning that this is a way of permanently switching program status. The LDPS instruction is useful for initiating normal-mode users' programs from the operating system, or for running a nonsegmented program on a Z8001. The segment-number portion of the PC is not affected by an LDPS instruction executed in the nonsegmented mode.

Of course, the LDCTL, EI, DI, and LDPS instructions are all privileged instructions that can be executed only during system-mode operation.

7

Bus and Resource Sharing

Besides interrupt requests, two other request buses can be used to control the sharing of resources in Z8000 systems. The address/data bus and its associated control and status signals are a resource that might be shared between processors and direct memory access (DMA) devices. Bus requests are made to the CPU when another device requires control of the bus; the Z-Bus bus request daisy chain allows multiple devices to share use of the bus in a well-defined manner. In multiprocessor systems, resources such as memory and I/O devices often are shared by two or more processors. For example, two separate Z8000-based systems might share a hard disk storage device, where only one processor may access the disk at a time. The Z-Bus multiprocessor resource sharing daisy chain defines the protocol for sharing a single resource among multiple Z8000 processors. Use of a hardware daisy-chain scheme to control bus and multiprocessor resource requests eliminates the need for separate priority controllers in the system.

BUS REQUESTS

In a given system, a CPU is designated as the default bus master; it uses the bus to fetch instructions and to transfer data to and from memory and peripheral devices as required during instruction execution. If another device, such as a DMA controller, needs to use the bus, that device must request control of the bus from the CPU.

Typically, bus requests are initiated by DMA controllers in a Z8000

system, where the DMA controllers are used for high-speed data transfers between memory and I/O devices. A peripheral device that needs servicing would request service from the DMA controller instead of interrupting the CPU. The DMA device requests control of the bus from the CPU, awaits acknowledgment of that request, performs the necessary data transfers, and then returns control of the bus to the CPU. Using a DMA controller to service a peripheral can have two advantages over CPU-controlled transfers between I/O and memory devices. First, the overhead involved in obtaining the bus for a DMA transfer via a bus request is less than the overhead involved in processing an interrupt in the CPU. Second, DMA controllers usually can transfer data between memory and I/O devices more efficiently (that is, faster) than the CPU. Of course, the bus requestor must use the address/data bus and its control and status lines in the exact same manner as the CPU to execute data transfers on the bus.

The bus request timing is illustrated in Fig. 7.1. A bus request is initi-

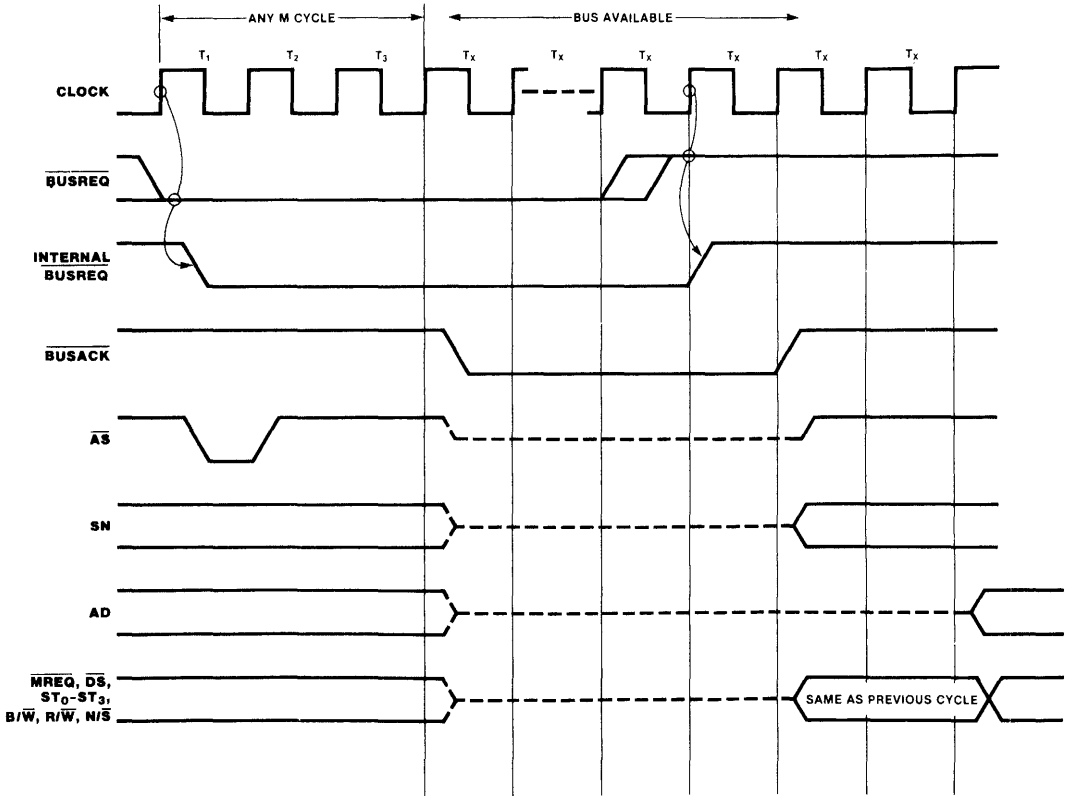


Figure 7.1 Bus request timing.

ated by pulling the $\overline{\text{BUSREQ}}$ input to the CPU low. The CPU samples this input at the start of each machine cycle, that is, at the rising edge of the clock of each T1 state. If $\overline{\text{BUSREQ}}$ is low at the start of the machine cycle, the CPU will relinquish the bus at the end of the bus transaction in that cycle (in other words, immediately after T3). The CPU gives up the bus by tri-stating the address/data bus (AD0-AD15), bus control ($\overline{\text{AS}}$, $\overline{\text{DS}}$, $\overline{\text{MREQ}}$), bus status (N/$\overline{\text{S}}$, B/$\overline{\text{W}}$, R/$\overline{\text{W}}$, ST0-ST3), and, for the Z8001, segment number (SN0-SN6) outputs. The CPU then pulls the $\overline{\text{BUSACK}}$ output low, signaling that the bus request has been acknowledged. The requesting device is now free to use the bus to initiate data transfers. For example, a DMA controller might transfer data between memory and I/O devices while in control of the bus. The bus requestor returns control of the bus to the CPU by deactivating the CPU's $\overline{\text{BUSREQ}}$ input. Once the CPU issues an active $\overline{\text{BUSACK}}$, it cannot regain control of the bus on its own; it must wait passively for the bus requestor to return control of the bus to the CPU by raising $\overline{\text{BUSREQ}}$. Two clock periods after $\overline{\text{BUSREQ}}$ goes high the CPU will raise $\overline{\text{BUSACK}}$ and regain control of the bus, with execution resuming at the point at which it was suspended by the bus request. Any device requiring control of the bus must wait at least two clock cycles after $\overline{\text{BUSREQ}}$ has risen (that is, until $\overline{\text{BUSACK}}$ returns high) before pulling $\overline{\text{BUSREQ}}$ down again.

Bus requests will always be acknowledged within two machine cycles after $\overline{\text{BUSREQ}}$ is active. Worst-case timing occurs if $\overline{\text{BUSREQ}}$ goes low immediately after the start of a machine cycle; that cycle will be executed, $\overline{\text{BUSREQ}}$ will be sensed at the start of the next cycle, and the bus request will be honored after that second cycle's bus transaction. During normal execution, the longest machine cycle is the eight-clock-period internal cycle, assuming that memory and I/O cycles are not longer than 8 clocks due to externally-generated wait states). During exception processing, the 10-clock-period interrupt acknowledge cycle is the longest possible machine cycle (again, excluding extra wait states). In most systems, worst-case response to a bus request would occur when the $\overline{\text{BUSREQ}}$ is pulled low right after the start of the aborted instruction fetch cycle during the response to an interrupt. The seven-clock-period aborted instruction fetch would execute, followed by 8 clock periods in the acknowledge cycle ($\overline{\text{BUSREQ}}$ would be sensed at the beginning of this cycle) before $\overline{\text{BUSACK}}$ would be returned. Thus the requesting device will always gain control of the bus soon after it pulls $\overline{\text{BUSREQ}}$ low in Z8000 systems.

BUS REQUEST PRIORITY DAISY CHAIN

If several devices in the system are capable of requesting control of the bus, these devices will share the $\overline{\text{BUSREQ}}$ input to the CPU. Arbitration between simultaneous requests from these devices is resolved via a priority

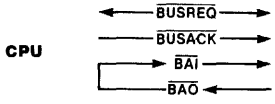


Figure 7.2 Z-Bus signals for bus requests.

daisy chain. Four signals are defined as part of the Z-Bus for handling bus requests and establishing the daisy chain: bus request ($\overline{\text{BUSREQ}}$), bus acknowledge ($\overline{\text{BUSACK}}$), bus acknowledge in ($\overline{\text{BAI}}$), and bus acknowledge out ($\overline{\text{BAO}}$) (Fig. 7.2).

Figure 7.3 is a block diagram of the bus request daisy chain. $\overline{\text{BUSREQ}}$ is driven by all bus requestors; a low on this line indicates that a bus requestor is trying to gain control of the bus. The $\overline{\text{BUSREQ}}$ signal serves two purposes: it is used as a CPU input to request control of the bus and as a status line to other bus requestors, indicating that some bus requestor is requesting or has control of the bus. $\overline{\text{BUSREQ}}$ must be bidirectional so that each device on the chain can monitor the activity of the other devices. $\overline{\text{BUSACK}}$ is a CPU output indicating that the CPU has relinquished bus control in response to a bus request. $\overline{\text{BAI}}$ and $\overline{\text{BAO}}$ are the signals that form the bus request daisy chain.

The protocol of the bus request daisy chain is outlined in Fig. 7.4. In the quiescent state (no bus requests being made and the CPU bus master has control of the bus) each device passes its $\overline{\text{BAI}}$ input to its $\overline{\text{BAO}}$ output. When a device needs to request control of the bus, it samples the $\overline{\text{BUSREQ}}$ line. A bus request can be made only if $\overline{\text{BUSREQ}}$ is initially high (that is, the CPU has control of the bus, and no other device is requesting the bus) and has been for two clock cycles. The request is made by pulling $\overline{\text{BUSREQ}}$ low. The requesting device also holds its $\overline{\text{BAO}}$ output high, thereby preventing bus acknowledgments from propagating to lower-priority devices, as explained below. The bus requestor gains control of the bus when its $\overline{\text{BAI}}$ input goes low. When the device is ready to return control of the bus to the CPU, it releases $\overline{\text{BUSREQ}}$ and allows its $\overline{\text{BAO}}$ output to follow its $\overline{\text{BAI}}$ input (that is, it returns to the quiescent state).

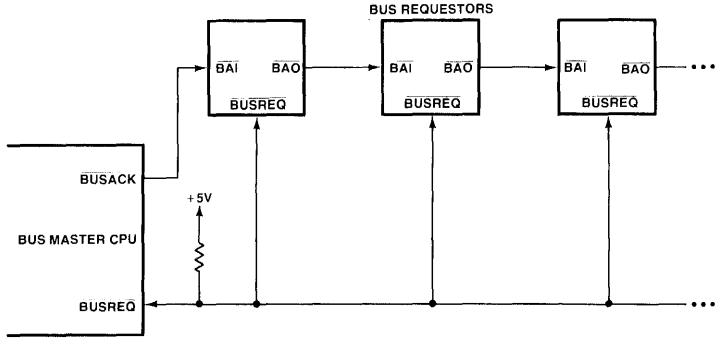


Figure 7.3 Z-Bus bus request daisy chain.

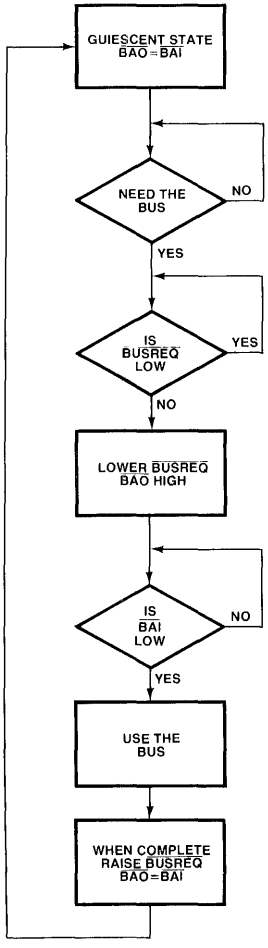


Figure 7.4 Bus request protocol.

Unlike the interrupt daisy chain, the bus request daisy chain is non-preemptive. Once a device gains control of the bus, no other devices on the chain can make a bus request until control is returned to the CPU, regardless of their position on the daisy chain. The daisy chain is used only to resolve the priority of simultaneous bus requests; if two devices request the bus at the same time, only the higher-priority device will receive the initial bus acknowledge on its $\overline{BAI}$ line, since the higher-priority device holds its $\overline{BAO}$ line high until it relinquishes control of the bus. After the higher-priority device returns to the quiescent state, the low $\overline{BUSACK}$ will propagate to the lower-priority device, that device will then be in control of the bus.

If the automatic memory refresh capabilities of the Z8000 CPUs are used to refresh dynamic memories, refresh cycles may be missed while DMA

devices have control of the bus. The CPU will remember the last two "missed" refresh row addresses and issue those refreshes immediately after regaining control of the bus. Care must be taken to ensure that bus requestors do not control the bus for a long enough period of time that the dynamic memory's contents are corrupted due to a lack of refreshes.

SHARED RESOURCE REQUESTS

The Z-Bus also includes signals for implementing a hardware daisy chain for requesting the use of a resource shared among several Z8000 processors. For example, several microcomputer systems might share a common disk drive or high-speed printer, where only one processor can use the shared resource at any one time. As each processor requires use of the shared resource, it must poll that resource to see if it is already being used by another processor. If not, the processor uses the resource, meanwhile locking out all other processors until it is finished. The shared resource could be an I/O device, a block of memory, or a shared bus.

Four Z-Bus signals make up the shared resource daisy chain: multi-micro request ($\overline{\text{MMRQ}}$), multi-micro status ($\overline{\text{MMST}}$) multi-micro acknowledge in ($\overline{\text{MMAI}}$), and multi-micro acknowledge out ($\overline{\text{MMAO}}$) (Fig. 7.5). Figure 7.6 shows a typical configuration of the daisy chain. The $\overline{\text{MMRQ}}$ signal can be driven by any device that can request use of the shared resource. A low on this line indicates that a request has been made or granted. The $\overline{\text{MMST}}$ is an input to each device on the daisy chain that indicates a request is pending or the shared resource is busy. $\overline{\text{MMAI}}$ and $\overline{\text{MMAO}}$ are the acknowledge signals that make up the daisy chain.

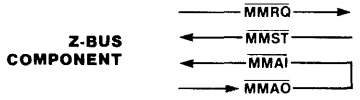


Figure 7.5 Z-Bus signals for sharing resources in a multiprocessor system.

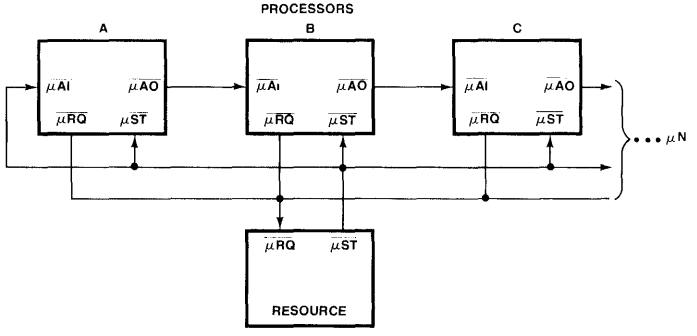


Figure 7.6 Z-Bus resource-sharing daisy chain.

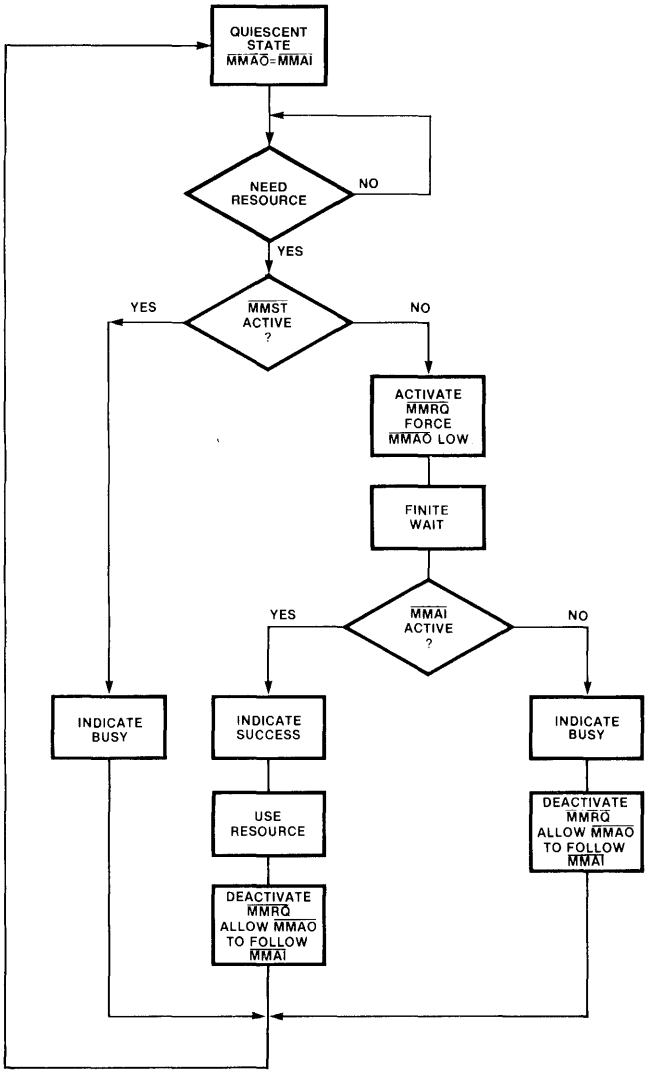


Figure 7.7 Multi-microprocessor resource-sharing protocol.

The protocol of resource requests is illustrated in Fig. 7.7. In the quiescent state (the resource is not busy and no requests are being made), each processor holds its $\overline{MMRQ}$ high and passes its $\overline{MMAI}$ input to its $\overline{MMAO}$ output. If $\overline{MMST}$ is high the resource is not busy and a device can request use of the resource by pulling $\overline{MMRQ}$ low. (Unlike the bus request protocol, no device on the shared resource daisy chain has control of the resource by default; every device must request the resource before using it.) Besides pulling $\overline{MMRQ}$ low, the requesting processor forces its $\overline{MMAO}$ output high, denying use of the resource to lower-priority devices on the daisy chain.

When the shared resource receives a request ($\overline{MMRQ}$ goes low), it pulls the $\overline{MMST}$ line low. This signal propagates through the $\overline{MMAI}/\overline{MMAO}$ daisy chain until reaching the highest-priority requestor. The requestor samples its $\overline{MMAI}$ input some finite amount of time after making the request; if $\overline{MMAI}$ is low, the requestor knows it has control of the shared resource. If $\overline{MMAI}$ is high, some higher-priority device on the daisy chain saw the acknowledge and is using the resource, so the original requestor returns to the quiescent state and tries again later. The delay between requesting the shared resource by pulling $\overline{MMRQ}$ low and checking for acknowledgment by sampling the $\overline{MMAI}$ input must be long enough to allow the $\overline{MMAI}/\overline{MMAO}$ daisy chain to settle. This time would be the sum of the worst-case $\overline{MMAI}$ to $\overline{MMAO}$ propagation delays for each device on the daisy chain. When the processor that has control of the shared resource is finished using the resource, it releases the resource by allowing $\overline{MMRQ}$ to go high and passing its $\overline{MMAI}$ input to its $\overline{MMAO}$ output. The shared resource lets the $\overline{MMST}$ line go high if $\overline{MMRQ}$ is high.

Like the bus request daisy chain, the shared resource request daisy chain is nonpreemptive. Once a processor gains control of the shared resource, no other device on the daisy chain can make a request until the shared resource is freed again. Priority on the daisy chain is important only in the case of two simultaneous requests. For example, if processor A and processor C of Fig 7.6 simultaneously pull their $\overline{MMRQ}$ lines low, only processor A will see a low on its $\overline{MMAI}$ input when it samples for acknowledgment, since processor A will hold its $\overline{MMAO}$ line high when it makes the request. When processor A has completed use of the resource, it will return to the quiescent state, and processor C can reassert its request.

All four lines in the shared resource request daisy chain are unidirectional, allowing the use of line drivers and receivers at each processor on the daisy chain. Furthermore, the delay for the daisy chain settling time can be set at each processor. Therefore, the devices on the chain can be separated by arbitrarily long distances.

A shared resource requestor can be any device capable of implementing the protocol. For Z8000 CPUs, the four lines of the resource request daisy chain are mapped into the multi-micro in ($\overline{MI}$) and multi-micro out ($\overline{MO}$) pins of the CPU. The logic for this mapping is shown in Fig. 7.8. During the quiescent state, the $\overline{MO}$ output is high; thus the $\overline{MI}$ input will reflect the state of the $\overline{MMST}$ line, and $\overline{MMAI}$ is gated out to $\overline{MMAO}$. $\overline{MMST}$ (on the $\overline{MI}$ input) can be read by the CPU to check the busy status of the resource. If the resource is not busy, the CPU pulls the $\overline{MO}$ output low. $\overline{MMAO}$ is forced high, and the $\overline{MMAI}$ signal is now gated to the CPU's $\overline{MI}$ input. After waiting for a predetermined delay time, the CPU reads $\overline{MMAI}$ (on the $\overline{MI}$ input). If it is low, the CPU knows that the request was successful, uses the resource, and sets $\overline{MO}$ high when completed. If $\overline{MMAI}$ is high when read, the request was not successfully acknowledged, and the CPU will set $\overline{MO}$ high

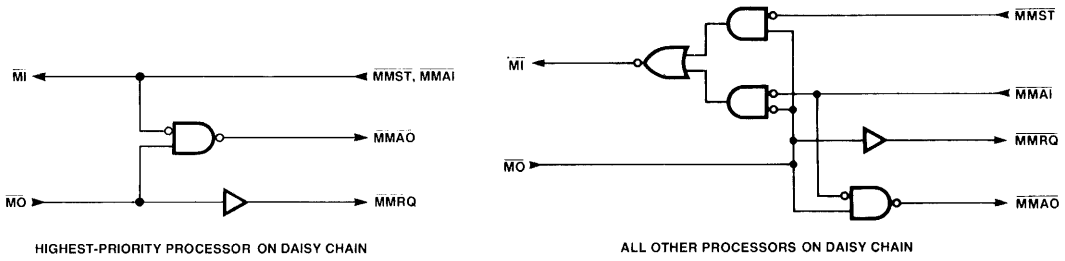


Figure 7.8 Logic for mapping the Z-Bus resource-sharing signals to the Z8000 CPU MO and MI pins.

and repeat the request later. (Alternatively, a 74LS157 multiplexor can be used in place of the combinational logic of Fig. 7.8, as illustrated in Fig. 7.9.)

If this configuration is used, one Z8000 instruction, the Multi-Micro Request (MREQ), will implement the whole protocol described above. The MREQ instruction has a 16-bit register as an operand. The contents of the register specified as the operand in the instruction will determine the delay between asserting the $\overline{\text{MMRQ}}$ and sampling the $\overline{\text{MMAI}}$ lines. After completion of the MREQ instruction, the S and Z flags in the FCW indicate whether or not the shared resource is available. Execution of the MREQ instruction is diagrammed in Fig. 7.10. First, the Z flag is cleared, and then the $\overline{\text{MI}}$ input is tested. If $\overline{\text{MI}}$ is low, indicating that $\overline{\text{MMST}}$ is low and the shared resource is busy, the S flag is cleared and $\overline{\text{MO}}$ remains high (no request is made). If $\overline{\text{MI}}$ is high, then $\overline{\text{MO}}$ is pulled low ($\overline{\text{MMRQ}}$ goes low) to initiate a request. Next, the contents of the register operand are decremented every seventh CPU clock cycle, until reaching zero. Then the $\overline{\text{MI}}$ input is sampled again, and the Z flag is set. If $\overline{\text{MI}}$ is high ($\overline{\text{MMAI}}$ high) the S flag is cleared and the $\overline{\text{MO}}$ line is pulled high (the request was not acknowledged). If $\overline{\text{MI}}$ is low, the S flag is set to 1, and the CPU now has control of the shared resource. Thus,

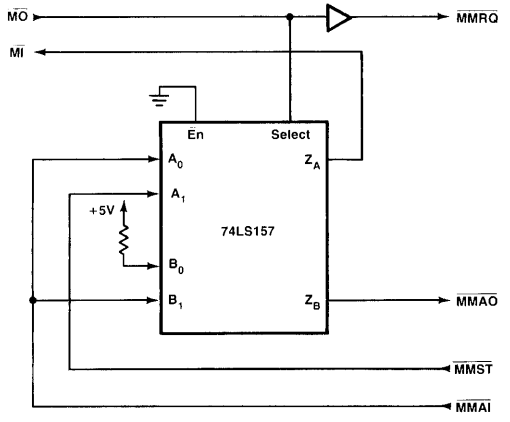


Figure 7.9 Mapping Z-Bus resource-request signals to a Z8000 CPU using a 2-to-1 multiplexer.

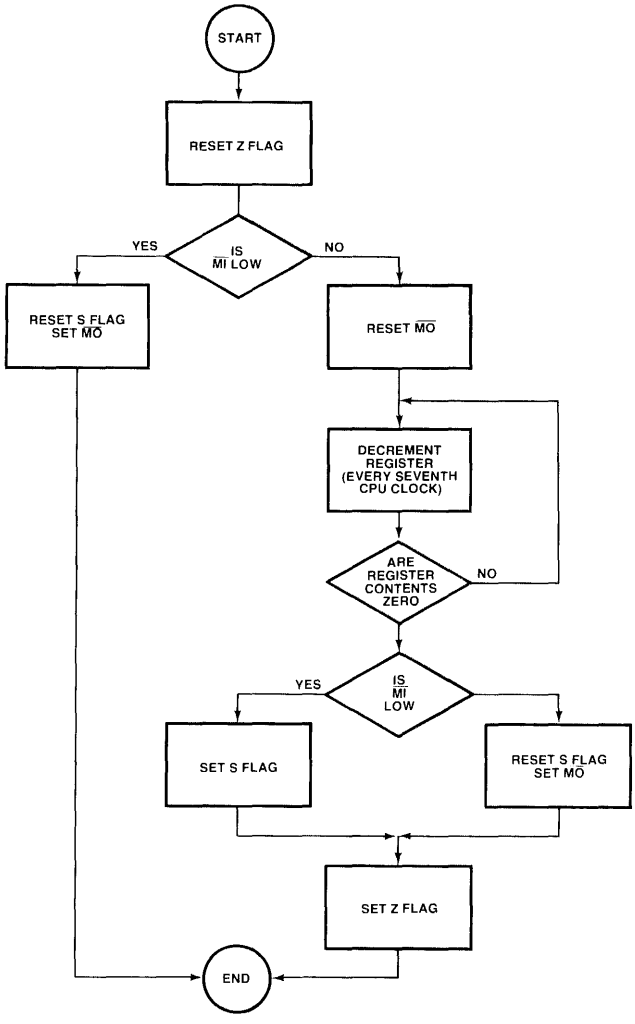


Figure 7.10 Execution of the MREQ instruction.

after execution of the MREQ, the Z flag indicates if a request was made, and the S flag indicates if the request was granted (Table 7.1). Another instruction, Multi-Micro Set (MSET), is used to write a 1 to $\overline{MO}$ after the CPU completes its use of the shared resource.

Two other instructions act on the $\overline{MO}$ and $\overline{MI}$ pins. The Multi-Micro Bit Test (MBIT) instruction reads the $\overline{MI}$ input and sets or resets the S flag if $\overline{MI}$ is high or low, respectively. The Multi-Micro Reset (MRES) instruction writes a 0 to the $\overline{MO}$ output. Thus MSET, MRES, and MBIT could be used to implement other, user-defined resource-sharing protocols, if so desired. If the system does not use the $\overline{MI}$ and $\overline{MO}$ pins for shared resource requests, they can be used as a single bit of input and output for other purposes.

TABLE 7.1 MEANING OF FLAGS AFTER THE EXECUTION OF A MREQ INSTRUCTION

S flag	Z flag	$\overline{MO}$	Indicates
0	0	High	Request not signaled (resource not available)
0	1	High	Request not granted (resource not available)
1	1	Low	Request granted (resource available)

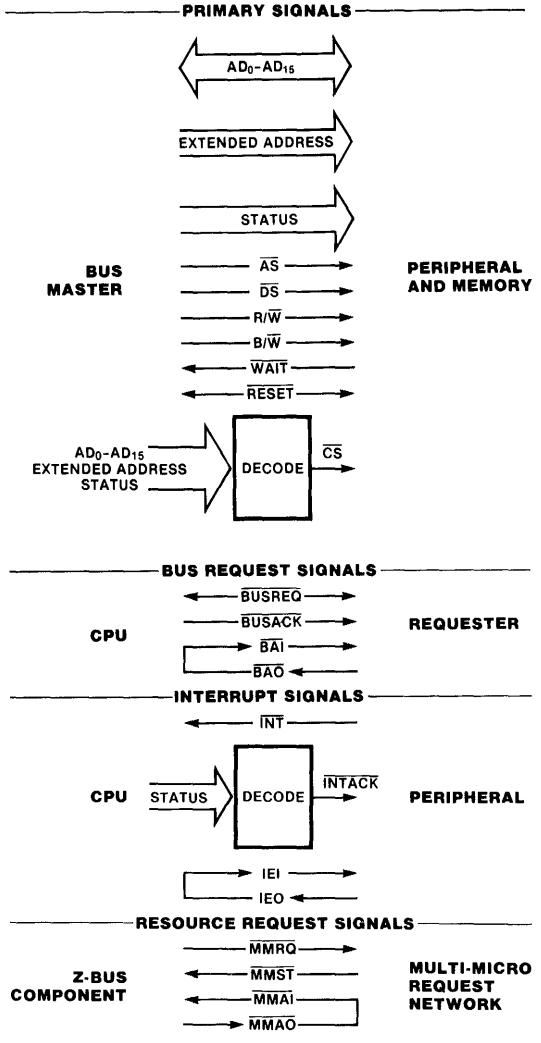


Figure 7.11 Z-Bus signals.

If the $\overline{MO}$ and $\overline{MI}$ pins are not used in the system at all, the MREQ instruction can be used to produce long, uninterruptible software delays. Execution of the MREQ instruction takes 10 + 7n clock cycles, where n is the contents of the word register specified as the operand in the MREQ instruction.

Z-BUS SIGNALS

All of the signals comprising the Z-Bus interface have now been discussed (Fig. 7.11). In review, the Z-Bus is a logical definition of the signals needed to link Z8000 family components in a computer system. Two kinds of operations can occur on the Z-Bus: transactions and requests. Four kinds of transactions are possible: memory, I/O, interrupt acknowledge, and null. (Null transactions occur during internal operation and memory refresh cycles, wherein no data are being transferred.) Only one transaction can occur at a time and it must be initiated by a bus master, that is, the device in control of the bus. There are three types of requests: interrupt, bus, and shared resource. A request can be initiated by a component that is not the bus master. For each type of request, a daisy-chain priority mechanism provides arbitration between simultaneous requests, eliminating the need for separate priority controllers.

Data transfers on the Z-Bus are asynchronous in the sense that Z-Bus components need not be synchronized with the CPU clock. The timing of data transfers is controlled by the $\overline{AS}$, $\overline{DS}$, and $\overline{WAIT}$ signals.

8

The Instruction Set

One feature of the Z8000 CPU architecture that sets it apart from previous generations of microprocessors is its large, powerful instruction set. The instruction set for the Z8000 includes 110 distinct instruction types. Many of these instructions act on byte, word, or long-word data operands. The operands in an instruction are specified using one of the Z8000's eight addressing modes. Combining the instruction types with the various data types and operand addressing modes supported by each instruction yields a set of over 400 different instructions.

The Z8000 CPUs have a full complement of data movement, arithmetic, logical, bit manipulation, rotate, shift, and I/O instructions. Instructions are included to manipulate strings of data, to facilitate multitasking operating systems, and to support typical high-level-language functions. When coupled with the other architectural features of the Z8000, such as the large memory address spaces, the Z8000 programmer is provided with computing power formerly available only on large mainframe computers.

ASSEMBLY LANGUAGE CONVENTIONS

Examples of assembly language instructions in this book will be written using the format accepted by the Zilog PLZ/ASM assembler. (PLZ/ASM is Zilog's structured assembly language that combines the assembly language instructions with some high-level constructs, allowing the programmer to write program code in a top-down, modular fashion.) Some of the conven-

TABLE 8.1 EXAMPLES OF ASSEMBLY LANGUAGE STATEMENTS IN PLZ/ASM

Label	Instruction mnemonic	Operands	Comments
START:	LD	R0, R2	! contents of R2 written to R0 !
Loop:	OUT	50, R5	! output R5 contents to port 50 !
	INCB	RL7	! increment register RL7 by 1 !
	NOP		! no operation !

tions for writing code in the PLZ/ASM language are described here, so that the programming examples shown in this chapter will be easily understood.

Table 8.1 illustrates some statements written in PLZ/ASM. A statement consists of an assembly language instruction, the instruction's operands, and optionally, label and comment fields. Each of these fields must be separated by at least one delimiter; legal PLZ/ASM delimiters include spaces, commas, tabs, semicolons, carriage returns, line feeds, and form feeds. Several delimiters in a row are treated the same as just one delimiter. Each assembly language instruction is denoted by that instructions's mnemonic; for example, LD is the mnemonic for the Load instruction. Z8000 instructions may have anywhere from zero to four operands, depending on the instruction; in the case of multiple operands, the operands must be separated by delimiters. Labels are placed before the instruction's mnemonic, and are always followed by a colon. Comments may be placed anywhere that a delimiter may appear, and are enclosed in exclamation points. Labels and comments are always optional and are added by the programmer to make the program easier to read, maintain, and debug.

Often, symbolic names are used to represent particular memory locations or numeric constants in PLZ/ASM. These symbolic names, also called identifiers, are made up of alphabetic characters (both upper- and lowercase which are distinct), the digits 0 through 9, and the underline character (_). Symbolic identifiers can be up to 127 characters long, and must start with an alphabetic character.

Numbers can be written in decimal, hexadecimal, octal or binary notation, as shown in Table 8.2. A number without any prefix is interpreted as a decimal number, the "%" prefix designates a hexadecimal number, the "%(8)"

TABLE 8.2 REPRESENTATION OF NUMERIC CONSTANTS IN PLZ/ASM

Decimal	Hexadecimal	Octal	Binary
10	%A	%(8)12	%(2)1010
17	%11	%(8)21	%(2)10001
255	%FF	%(8)377	%(2)11111111

prefix designates an octal number, and the "%(2)" prefix designates a binary number.

When writing segmented addresses, the segment number is preceded by two "<" symbols and followed by two ">" symbols. The offset would follow immediately. For example, "<<5>>20" means segment 5, offset address 20 (the 5 and 20 are both decimal values in this case).

The general-purpose CPU registers are designated as follows: R0–R15 for the word registers; RH0–RH7 and RL0–RL7 for the byte registers; RR0, RR2, and so on, through RR14 for the long-word registers; RQ0, RQ4, RQ8, and RQ12 for the quad registers.

As with most assembly languages, the destination operand is placed before the source operand. Thus the statement

```
LD  R0, R2
```

means that the contents of register R2 are to be written into register R0. Some other assembler conventions, such as the designation of the operand addressing mode, will be described later.

CPU REGISTER USAGE

The 16 general-purpose CPU registers can each be used as an accumulator (the register holds the result of an artithmetic or logical operation), a pointer (the register holds a memory or I/O address), or an index (the register holds a value that is added to a base address to produce a memory address), with some minor exceptions. Register R0 cannot be used to hold a nonsegmented memory address, an I/O address, or an index, and RR0 cannot be used to hold a segmented memory address. Registers R15 in nonsegmented mode and RR14 in segmented mode are the implied stack pointers for subroutine calls and exception processing and, therefore, usually are not used as accumulators or indexes.

Nonsegmented memory addresses and I/O addresses are always 16 bits long and can be stored in any word register except R0. Segmented addresses are 23 bits long, and can be stored in any register pair except RR0. When storing a segmented address in a long-word register, the segment number is placed in bits 8–14 of the high-order word register of the register pair, and the offset address is stored in the low-order word register (Fig. 8.1).

The general philosophy of the instruction set is to have two operand instructions, with a register, memory location, I/O port, or immediate value

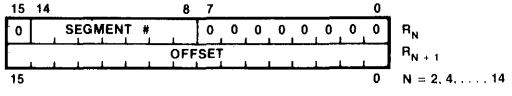


Figure 8.1 Format for storing segmented addresses in long-word registers.

for the source operand and a register as the destination. Thus most arithmetic, logical, shift, and rotate instructions have a register as the destination. Input instructions use a register for the destination and output instructions use a register for the source. Storing data in the registers allows shorter instructions and faster execution than when operands must be fetched from memory. However, some memory-to-memory and I/O-to-memory commands are included to improve program density; these commands normally are used to manipulate entire strings of data.

LONG AND SHORT OFFSET ADDRESSES

For two of the operand addressing modes, direct address (DA) and indexed (X), an I/O or memory address is included in the instruction itself. Nonsegmented memory addresses and I/O addresses are always 16 bits long, so these addresses occupy one word in the instruction's opcode. Segmented addresses are 23 bits long and can be represented two different ways within an instruction.

The most general format for designating a segmented address within an instruction is the long offset format, wherein the address is stored in two consecutive words (Fig. 8.2). The first word contains the segment number portion of the address in bits 8–14; bits 0–7 are reserved (set to 0's). Bit 15 is set to a 1, which designates the long offset format. The 16-bit offset address is placed in the next word of the instruction.

In the short offset format, the segmented address is compressed into one word in the instruction. Bit 15 is cleared to zero, bits 8–14 hold the segment number, and bits 0–7 hold the lower byte of the offset. The upper byte of the offset address is assumed to be all 0's. Thus the short offset format can be used only to address the first 256 bytes in a segment (that is, offset addresses 0 to 255 in the segment).

For the Zilog PLZ/ASM assembler, the short offset format is denoted by enclosing the segmented address within vertical bars when writing the instruction:

```
LD   R0, |«5»20 |    ! memory-to-register load, memory address is in short offset format !
LD   R0, «5» 300     ! memory-to-register load, memory address is in long offset format !
```

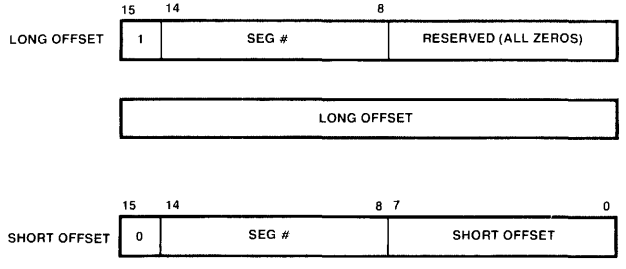


Figure 8.2 Formats for storing segmented addresses within an instruction's opcode.

The short offset format can be used only when a segmented address is included in the instruction's opcode. The short offset format cannot be used to store a segmented address in a CPU register.

ADDRESSING MODES

A Z8000 assembly language instruction is stored in memory as a consecutive list of one or more words that are accessed via the program counter during program execution. The word or words in memory making up the instruction are called an operation code (opcode). In addition to specifying what type of instruction is to be executed (for example, a load or add), the opcode also must specify the location of the data to be operated on, called the instruction's operands. Operands can reside in CPU registers, memory locations, or I/O ports. The method used to determine the location of an operand is called an addressing mode.

The Z8000 CPUs support eight different addressing modes: register (R), direct address (DA), immediate (IM), indirect register (IR), indexed (X), base address (BA), base indexed (BX), and relative address (RA). Not all instructions are capable of using every addressing mode to specify its operands. The addressing mode for an operand is explicitly specified when writing the instruction.

For the X, BA, BX, and RA addressing modes, the memory address of the operand is found by adding a 16-bit value, called an index or displacement, to a memory address, called the base address. The address yielded by this operation is called the effective address of the operand. In segmented-mode operation, this calculation of the operand's address applies only to the offset portion of the address; the segment number of the operand's effective address is always the segment number of the base address. Any carry resulting from adding the index to the offset address is ignored, rather than incrementing the segment number. In other words, the segment number and offset portions of the address are distinct and address calculations performed on the offset address will never alter the segment number.

REGISTER MODE

In the register (R) addressing mode, the operand is the contents of the CPU register specified in the instruction (Fig. 8.3). The register length (byte,

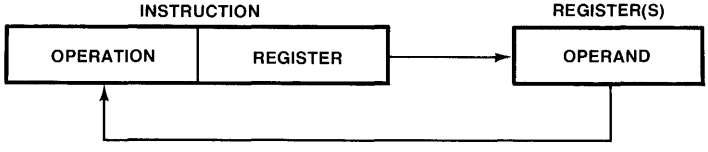


Figure 8.3 Register mode.

word, long word, or quad word) is determined by the instruction. The following statements use register addressing to specify both the source and destination operands:

```
LD   R2, R5          ! contents of R5 written into R2 !
ADDB  RH0, RL4       ! 8-bit addition of RH0 and RL4, sum goes in RH0 !
MULTL  RQ4, RL8      ! 32-bit multiply !
```

DIRECT ADDRESS MODE

For the direct address (DA) mode, the address of the operand is included in the instruction's opcode (Fig. 8.4). The address could be a memory address or an I/O address, as indicated by the instruction type. For 16-bit nonsegmented addresses and I/O port addresses, the address will be given in the second word of the instruction's opcode. Segmented addresses can be stored in an opcode in two ways: long offset format (the address is in the second and third words of the instruction's opcode) or short offset format (the address is in the second word of the intruction's opcode).

The operand specified by the DA mode can be in the I/O address space, special I/O address space, or data memory address space, depending on the instruction:

```
LD   R0, «10»%4AC0     ! segmented mode, contents of data memory location 4AC0 hex in
                         segment 10 are loaded into R0 !
SOUT   %00FF, R5       ! destination is port FF hex in special I/O address space !
```

Typically, the address is specified using a symbolic name rather than the numerical address:

```
LD   R0, TOTAL     ! TOTAL is previously defined symbolic name for some location in data
                     memory !
```

The instruction that uses DA mode to specify an operand could be a byte, word, or long-word instruction; the direct addressing logic is identical for each case. For byte instructions, a single byte in memory or an I/O port is accessed; for word instructions, the whole word is accessed. Long-word instructions will access the word whose address is given in DA mode and the next sequential word in memory.

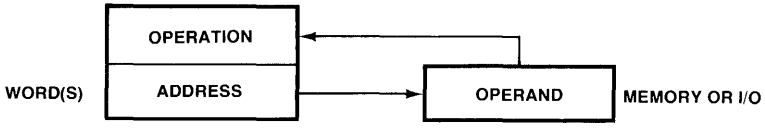


Figure 8.4 Direct address mode.

The DA mode also is used by the Jump and Call instructions to specify the address of the next instruction to be executed. In this case, the address in the instruction opcode is not used to reference a data operand in a memory or I/O location, but instead is used as an immediate value that is loaded into the program counter:

```
JUMP   %5000              ! jump to location 5000 hex !
CALL   SOME_ROUTINE       ! SOME_ROUTINE is the symbolic name for the starting ad-
                            dress of a subroutine !
```

IMMEDIATE MODE

Immediate (IM) mode addressing means that the operand appears within the instruction's opcode itself (Fig. 8.5). IM mode is the only addressing mode that does not involve specifying a register, memory, or I/O location for the

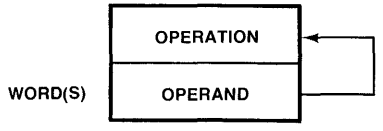


Figure 8.5 Immediate mode.

operand. Instructions that act on bytes or words provide the immediate operand as the second word in the opcode. For byte instructions, the immediate byte of data is duplicated in both bytes of the opcode's second word. Instructions that operate on long words provide the immediate operand as the second and third words of the opcode.

Since immediate operands are always part of the opcode, they are always located in the program memory address space. The "#" symbol is used to specify immediate-mode addressing. Immediate mode often is used to initialize memory and register locations:

```
LD   R0, #%5407     ! 5407 hex loaded into R0 !
ADDB RH8, #20       ! 20 added to contents of RH8 !
```

INDIRECT REGISTER MODE

For the indirect register (IR) mode, the operand resides at some memory or I/O location. The address of the operand is the contents of the register specified in the opcode. In other words, a register holds the address of the operand (Fig. 8.6). Of course, the appropriate address must be loaded into the register before IR mode is used. The instruction type determines if the address in the register is a memory or I/O address. Any word register except R0 can hold a nonsegmented memory address or an I/O address. Any register pair except RR0 can hold a segmented memory address.

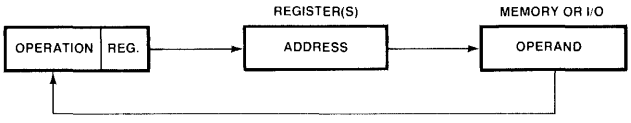


Figure 8.6 Indirect register mode.

Depending on the instruction, the operand specified by IR addressing could be in the I/O address space, special I/O address space, data memory address space, or stack memory address space. For memory accesses, if R15 in nonsegmented mode or RR14 in segmented mode is the register used to hold the operand's address, the operand will be in the stack memory address space (that is, the ST3-ST0 lines will be 1001 during the access); otherwise, the operand is in the data memory address space.

The "@" symbol in front of the register name is used to designate indirect register addressing:

```
LD   R5, @R2        ! nonsegmented mode, the destination uses IR addressing, the contents
                      of the data memory location whose address is in R2 is loaded into
                      R5 !
INB  RH0, @R1       ! R1 holds standard I/O port address !
LD   @RR14, R0      ! segmented mode, source uses IR addressing, source is in stack
                      memory !
```

INDEXED MODE

For the indexed (X) mode, the operand is located in a memory location. The address of the memory location is found by adding an address located in the instruction's opcode, called the base address, to the contents of a word register specified in the instruction, called an index (Fig. 8.7). The address formed by adding the index to the base address is called the effective address of the operand. Any general-purpose register except R0 can hold an index. The base address can be segmented or nonsegmented, depending on the current operating mode of the CPU; if segmented, the base address can be specified using the long or short offset formats.

When using the indexed addressing mode in segmented mode, the index is added to the offset portion of the base address, without affecting the segment number. The segment number of the effective address for the operand is always the segment number of the base address.

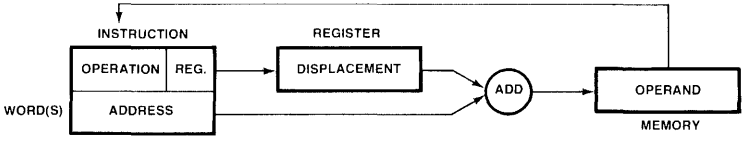


Figure 8.7 Indexed mode.

Operands specified using X mode are in the data memory address space, except when this mode is used with the Jump and Call instructions. For Jumps and Calls, the effective address is not used as the address of an operand in memory, but instead is used as an immediate value that is loaded into the program counter to execute the Jump or Call.

Indexed addressing is specified by designating the base address, followed by the register that holds the index enclosed in parentheses:

```
LD   R0, %4A90(R11)          ! nonsegmented mode, X mode for source operand !
LDB  RL3, LIST(R13)          ! LIST is symbolic name for some data variable !
JUMP DISPATCH_TABLE(R5)
```

Indexed addressing allows the programmer to access elements in tables of data, where the base address of the table is known, but the index of the particular element accessed is to be computed during program execution.

BASE ADDRESS MODE

Base address (BA) mode is the inverse of indexed mode; in BA mode, the specified register holds a base address and the index, or displacement, is a 16-bit value that is part of the instruction's opcode (Fig. 8.8). The operand's effective memory address is found by adding the index to the base address. In the nonsegmented mode, the base address can be in any register except R0; in the segmented mode, the base address can be in any register except RR0.

As with the indexed mode, the effective address calculation does not affect the segment number of the base address; the segment-number portion of the effective address will always be the segment number of the base address.

An operand specified using BA mode is in the stack memory address space if the base address is in R15 in the nonsegmented mode or RR14 in the segmented mode. Otherwise, the operand is in the data memory address space.

Base addressing is used only with the Load (LD) instruction. Base addressing is specified by stating the register that holds the base address, followed by an immediate index; the index is enclosed in parentheses:

```
LD   R1(#50), R8      ! nonsegmented mode, BA mode specifies the destination in data
                        memory !
LD   R0, RR14(#%A0)   ! segmented mode, BA mode specifies source in stack memory !
```

The base address mode is the complement of the indexed mode and allows access to data in a table where the index into the table is known, but the base address of the table is computed during program execution. For ex-

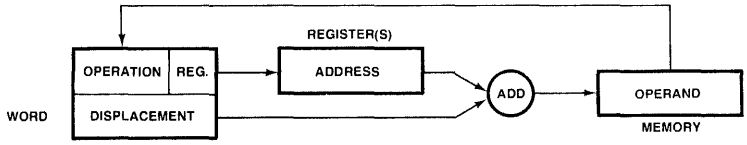


Figure 8.8 Base address mode.

ample, the BA mode might be used for accessing the corresponding elements in several different tables of data.

For the nonsegmented mode, the base address and indexed modes are equivalent; for each, a 16-bit value in a register is added to a 16-bit value given in the instruction's opcode to yield an effective memory address for the operand. The only difference occurs when R15 is used to hold the base address or index:

```
LD   R0, 50 (R15)
```

accesses data memory to obtain the source operand, whereas

```
LD   R0, R15 (#50)
```

would access stack memory to obtain the source operand. The indexed mode is used more frequently, however, since it is available in more instructions and results in faster execution than when base addressing is used.

BASE INDEXED MODE

The base indexed (BX) mode is a combination of the X and BA modes. For the BX mode, both the base address and index are held in registers (Fig. 8.9). In the nonsegmented mode, both the base address and the index are held in any word register except R0. In the segmented mode, the base address may be in any register pair except RR0 and the index may be in any word register except R0. The effective memory address of the operand is found by adding the index to the base address. The segment-number portion of the effective

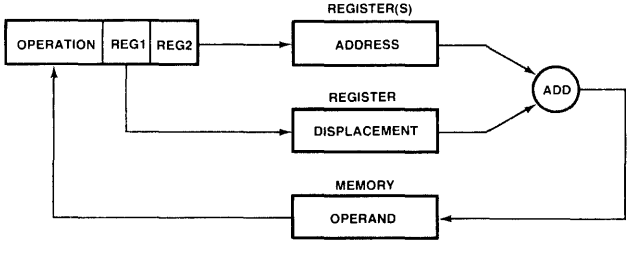


Figure 8.9 Base indexed mode.

address is always the segment number of the base address; the index is added to the offset only.

Operands specified using BX mode are in the stack memory address space if either R15 in the nonsegmented mode or RR14 in the segmented mode is used to hold the base address. Otherwise, the operand is in data memory.

The base indexed mode is used only with the Load (LD) instruction. When specifying the BX mode, the register that holds the base address is given, followed by the register holding the index; the index is enclosed in parentheses:

```
LD   R0, R2(R5)          ! nonsegmented mode, BX mode designates source !
LDB  RH2, RR14(R9)       ! segmented mode, source is in stack memory !
```

RELATIVE ADDRESS MODE

In the relative address (RA) mode, the operand is at a memory address that is calculated relative to the current program counter value. A displacement is given in the instruction's opcode; this displacement is a two's-complement number that is added to or subtracted from the program counter to yield the address of the operand (Fig. 8.10). The maximum size of the displacement depends on the instruction used. The PC value used is the address of the next memory location following the currently executing instruction. In the segmented mode, the calculation of the operand's effective address will not affect the segment-number portion of the address; that is, the operand is always in the segment determined by the segment number in the PC.

An operand specified by RA mode is always in the program memory address space. Relative addressing allows references to memory locations that are a short distance backward or forward from the current PC value. It is used by the Jump Relative (JR), Call Relative (CALR), Decrement and Jump if Not Zero (DJNZ, DBJNZ), Load Relative (LDR, LDRB, LDRL), and Load Address Relative (LDAR) instructions. Typically, the programmer uses label names in the program to specify the location to be referenced using RA mode and lets the assembler calculate the displacement value that becomes part of the instruction's opcode:

```
LDR   R5, LOOP      ! LOOP is a label in the program !
```

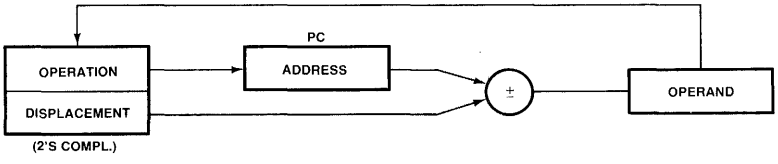


Figure 8.10 Relative address mode.

For the JR and CALR instructions, the effective address is not used to reference a memory location, but instead is used as an immediate value that is loaded into the program counter.

USE OF THE ADDRESSING MODES

Not every instruction in the Z8000 is capable of supporting every addressing mode; only the Load (LD) instruction can do so. However, the architecture is very regular in that the five main addressing modes—R, DA, IM, IR, and X—are available in most instructions that can reference operands in memory (that is, all the arithmetic and logical instructions).

Use of the R, DA, and IM modes implies that the programmer knows the exact location or value of the operand at assembly time. In contrast, use of the IR, X, BA, and BX modes to specify an operand's address involves storing the address or index value in a register. Since registers' contents can be varied during program execution, these modes allow run-time calculation of the operand's address; that is, the effective address of the operand will depend on the results of some previous operations executed when running the program.

The Z8000's addressing modes provide considerable flexibility in operand addressing. It is, of course, up to the programmer to make the most effective use of the addressing modes.

IMPLIED ADDRESSING MODES

Besides the eight addressing modes described above, the Z8000 has two additional modes that are implied by the use of certain instructions, auto-increment and auto-decrement. Several instructions manipulate entire strings of data in memory. These instructions address source and destination strings in memory using indirect register addressing; that is, the address of the element in the string currently being acted on is in a register. After each iteration of the instruction's execution, the operand's address is incremented or decremented, as specified by the instruction type, so that the register now points to the next element of the string, ready for the next iteration of the instruction's execution.

ASSEMBLY LANGUAGE INSTRUCTIONS

The Z8000 CPU's assembly language instruction set can be segregated into 12 categories of instructions: data movement, arithmetic, logical, bit manipulation, rotate and shift, program control, block move, block compare, block translate, I/O, special I/O, and CPU control instructions. Tables 8.4 through

8.15 list the instructions in each category, including the mnemonic, operands, addressing modes for the operands, clock cycles required to execute, and a description of the operation for each instruction.

Many of the instructions can operate on byte, word, or long-word data types. In general, if an instruction's mnemonic ends with a "B" suffix, it is a byte instruction; if the instruction's mnemonic ends with an "L" suffix, it is a long-word instruction; otherwise, it is a word instruction.

The number of CPU clock cycles required for execution of an instruction depends on the data type for that instruction and the segmentation mode of the CPU. In Tables 8.4 through 8.15, NS means nonsegmented mode, SS means segmented mode using short offset format, and SL means segmented mode using long offset format. The execution times are calculated assuming that there are no externally-generated wait states during memory and I/O accesses.

The carry, zero, sign, and overflow/parity flags in the FCW are used to control the operation of certain "conditional" instructions, such as the Jump (JP) instruction. The operation of these instructions depends on the condition of these four flags. Sixteen different combinations of these flag settings are encoded in a 4-bit field in the opcode called a condition code. The mnemonics for the condition codes and the flag settings they represent are listed in Table 8.3. Although there are only 16 unique conditions, the

TABLE 8.3 CONDITION CODES

Code	Meaning	Flag setting
F	Always false	—
	Always true	—
Z	Zero	Z = 1
NZ	Not zero	Z = 0
C	Carry	C = 1
NC	No carry	C = 0
PL	Plus	S = 0
MI	Minus	S = 1
NE	Not equal	Z = 0
EQ	Equal	Z = 1
OV	Overflow	V = 1
NOV	No overflow	V = 0
PE	Parity even	P = 1
PO	Parity odd	P = 0
GE	Greater than or equal	(S XOR V) = 0
LT	Less than	(S XOR V) = 1
GT	Greater than	(Z OR (S XOR V)) = 0
LE	Less than or equal	(Z OR (S XOR V)) = 1
UGE	Unsigned greater than or equal	C = 0
ULT	Unsigned less than	C = 1
UGT	Unsigned greater than	((C = 0) AND (Z = 0)) = 1
ULE	Unsigned less than or equal	(C OR Z) = 1

Zilog PLZ/ASM assembler recognizes more than 16 condition code mnemonics; in some cases, two different mnemonics correspond to identical flag settings (Z and EQ, for example). If no mnemonic for a condition code is given in a conditional instruction, the "always true" condition is assumed.

DATA MOVEMENT INSTRUCTIONS

The data movement instructions, listed in Table 8.4, provide one of three functions: (1) load a register with data from a register or memory location, (2) load a memory location with data from a register, or (3) load a register or memory location with an immediate value. These instructions do not affect the flags in the FCW.

The Clear instructions (CLR, CLRB) are used to clear a byte or word register or memory location to zero. This is functionally equivalent to a Load instruction with an immediate operand of zero.

The Exchange instructions (EX, EXB) are used to swap the contents of two registers, or a register and a memory location. A temporary storage register internal to the CPU is used to implement the swap. The Exchange instruction is useful for converting Z80 or other microprocessor code into Z8000 code, since the Z8000 uses the opposite convention of odd/even memory addressing of bytes in words than the Z80.

The Load instructions (LD, LDB, LDL) provide for transferring data between memory and register locations. Note that no memory-to-memory loads are included in this group of instructions.

The Load Address instruction (LDA) loads the address of the source operand into the destination register. The contents of the source are not accessed; the effective address computation corresponding to the specified addressing mode is made, and that effective address, not the data at that address, is written into the destination. The destination must be a word register in nonsegmented mode, and a long-word register in segmented mode.

The Load Address Relative instruction (LDAR) is similar to the LDA instruction, except that it supports the relative address mode. The displacement can range from - 32768 to +32767 and is added to the current program counter value to yield the effective address that is loaded into the destination.

The Load Constant instruction (LDK) is a short, fast instruction for loading small numeric constants into a word register. The source operand must be an immediate value between 0 and 15. The high-order 12 bits of the destination register are cleared to zeros.

The Load Multiple instruction (LDM) provides for efficient saving and restoring of registers' contents and can significantly lower the overhead required for procedure calls and other context switches. This instruction

TABLE 8.4 DATA MOVEMENT INSTRUCTIONS

Mnemonics	Operands	Address modes	Clock cycles: Word/byte NS	Word/byte SS	Word/byte SL	Long word NS	Long word SS	Long word SL	Operation
CLR, CLRB	dst	R	7	—	—				Clear
		IR	8	—	—				dst ← 0
		DA	11	12	14				
		X	12	12	15				
EX, EXB	R, src	R	6	—	—				Exchange
		IR	12	—	—				R ↔ src
		DA	15	16	18				
		X	16	16	19				
LD, LDB, LDL	R, src	R	3	—	—	5	—	—	Load into Register
		IM	7	—	—	11	—	—	R ← src
		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)
		DA	11	12	14	14	15	17	dst ← R
		X	12	12	15	15	15	18	
		BA	14	—	—	17	—	—	
		BX	14	—	—	17	—	—	
LD, LDB	dst, IM	IR	11	—	—				Load Immediate into Memory
		DA	14	15	17				dst ← IM
		X	15	15	18				
LDA	R, src	DA	12	13	15				Load Address
		X	13	13	16				R ← source address
		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 Multiple
		DA	14	15	17	+3n			R ← src (n consecutive words)
		X	15	15	18				(n = 1 . . . 16)
LDM	dst, R, n	IR	11	—	—				Load Multiple (Store Multiple)
		DA	14	15	17	+3n			dst ← R (n consecutive words)
		X	15	15	18				(n = 1 . . . 16)
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	8	—	—	12	—	—	Pop
		IR	12	—	—	19	—	—	dst ← IR
		DA	16	16	18	23	23	25	Autoincrement contents of R
		X	16	16	19	23	23	26	
PUSH, PUSHL	IR, src	R	9	—	—	12	—	—	Push
		IM	12	—	—	—	—	—	Autodecrement contents of R
		IR	13	—	—	20	—	—	IR ← src
		DA	14	14	16	21	21	23	
		X	14	14	17	21	21	24	

allows any contiguous group of 1 to 16 general-purpose registers to be transferred to or from a block of consecutive memory locations.

The Load Relative instructions (LDR, LDRB, LDRL) allow data transfers between the registers and program memory locations. The operand specified using relative addressing can be displaced -32768 to +32767 addresses from the current PC value. The displacement is added to the PC to yield the effective address of the operand in program memory.

The Pop (POP, POPL) and Push (PUSH, PUSHL) instructions support stack operations. The register to be used as the stack pointer (the source for pops and destination for pushes) is designated using the IR addressing mode; any general-purpose word register except R0 can be used as a stack pointer in the nonsegmented mode, and any register pair except RR0 can be a stack pointer in the segmented mode. Byte operations are not allowed; a stack pointer should always have an even address, since only words and long words can be written and read from stacks. The register being used as the stack pointer is automatically incremented by two after popping a word, or by four after popping a long word; the stack pointer is automatically decremented before a push. In the segmented mode, the segment-number portion of the address in the register pair used as a stack pointer is not affected by the automatic increment and decrement operations.

ARITHMETIC INSTRUCTIONS

Table 8.5 lists the Z8000 instructions that perform integer arithmetic. The basic instructions use standard two's-complement binary format for representing integers, but support is provided for BCD arithmetic as well. Most instructions in this group perform a binary operation between a register's contents and a source operand designated by one of the five basic addressing modes (R, DA, IM, IR, and X). The result is loaded into a register. These instructions set the flags in the FCW to the appropriate values depending on the result of the arithmetic operation. The P/V flag is used to indicate overflow for these instructions and is called the V flag.

The Add (ADD, ADDB, ADDL) and Subtract (SUB, SUBB, SUBL) instructions perform basic binary addition and subtraction. Multiple-precision operations can be implemented using the Add with Carry (ADC, ADCB) and Subtract with Carry (SBC, SBCB) instructions, but these instructions support only register addressing for both the source and destination.

The Compare instructions (CP, CPB, CPL) allow comparison of a register's contents to another register's contents, a memory location's contents, or an immediate value. The Compare instructions do not affect either operand but set the flags based on the result that occurs when the source is subtracted from the destination.

The Decimal Adjust instruction (DAB) operates only on byte registers

TABLE 8.5 ARITHMETIC INSTRUCTIONS

Mnemonics	Operands	Address modes	Clock cycles: Word/byte NS	Word/byte SS	Word/byte SL	Long word NS	Long word SS	Long word SL	Operation
ADC, ADCB	R, src	R	5	—	—				Add with Carry R ← R + src + carry
ADD, ADDB, ADDL	R, src	R	4	—	—	8	—	—	Add R ← R + src
		IM	7	—	—	14	—	—	
		IR	7	—	—	14	—	—	
		DA	9	10	12	15	16	18	
		X	10	10	13	16	16	19	
CP, CPB, CPL	R, src	R	4	—	—	8	—	—	Compare with Register R − src
		IM	7	—	—	14	—	—	
		IR	7	—	—	14	—	—	
		DA	9	10	12	15	16	18	
		X	10	10	13	16	16	19	
CP, CPB	dst, IM	IR	11	—	—				Compare with Immediate dst − IM
		DA	14	15	17				
		X	15	15	18				
DAB	dst	R	5	—	—				Decimal Adjust
DEC, DECB	dst, n	R	4	—	—				Decrement by n dst ← dst − n (n = 1 . . . 16)
		IR	11	—	—				
		DA	13	14	16				
		X	14	14	17				
DIV, DIVL	R, src	R	107	—	—	744	—	—	Divide (signed) Word: $R_{n+1} \leftarrow R_{n, n+1}$ + src $R_n \leftarrow$ remainder Long word: $R_{n+2, n+3} \leftarrow$ $\leftarrow R_{n \ldots n+3}$ + src $R_{n, n+1} \leftarrow$ remainder
		IM	107	—	—	744	—	—	
		IR	107	107	107	744	744	744	
		DA	108	109	111	745	746	748	
		X	109	109	112	746	746	749	
EXTS, EXTSB, EXTSL	dst	R	11	—	—	11	—	—	Extend Sign Extend sign of low-order half of dst through high-order half of dst
INC, INCB,	dst, n	R	4	—	—				Increment by n dst ← dst + n (n = 1, . . . 16)
		IR	11	—	—				
		DA	13	14	16				
		X	14	14	17				
MULT, MULTL	R, src	R	70	—	—	282[a]	—	—	Multiply (signed) Word: $R_{n, n+1} \leftarrow R_{n+1} \cdot$ src Long word: $R_{n \ldots n+3} \leftarrow$ $\leftarrow R_{n+2, n+3}$
		IM	70	—	—	282[a]	—	—	
		IR	70	—	—	282[a]	—	—	
		DA	71	72	74	283[a]	284[a]	286[a]	
		X	72	72	75	284[a]	284[a]	287[a]	
NEG, NEGB	dst	R	7	—	—				Negate dst ← 0 − dst
		IR	12	—	—				
		DA	15	16	18				
		X	16	16	19				
SBC, SBCB	R, src	R	5	—	—				Subtract with Carry R ← R − src − carry
SUB, SUBB, SUBL	R, src	R	4	—	—	8	—	—	Subtract R ← R − src
		IM	7	—	—	14	—	—	
		IR	7	—	—	14	—	—	
		DA	9	10	12	15	16	18	
		X	10	10	13	16	16	19	

[a]Plus seven cycles for each 1 in the multiplicand.

and is used to implement BCD arithmetic. Two BCD digits can be packed into a byte register, one per nibble. Byte registers so formatted can be added or subtracted using the binary ADDB and SUBB instructions; these instructions would be followed by a DAB. The DAB instruction adjusts the destination register back into BCD format using the D and H flags.

The Decrement (DEC, DECB) and Increment (INC, INCB) instructions are used to decrement or increment a register or memory location by an immediate value between 1 and 16.

The Extend Sign instructions (EXTS, EXTSB, EXTSL) are used to convert a small signed operand (in a register) to a larger signed operand, by copying the sign bit (most significant bit) of the low-order half of the destination to all the bits in the high-order half of the destination. Thus an 8-bit signed integer can be converted to 16 bits, a 16-bit integer to 32 bits, or a 32-bit integer to 64 bits.

The Divide instructions (DIV, DIVL) perform signed two's-complement division on word or long-word operands. The DIV instruction requires a long-word register as the destination and a word operand as the source. The 32-bit destination is divided by the source; the quotient is written into the low-order half of the destination and the remainder is loaded into the high-order half of the destination. For example:

```
DIV  RR2, R5
```

would divide the contents of RR2 by the contents of R5 and, after the division, the quotient would be in R3 and the remainder in R2. Similarly, the DIVL instruction requires a quad register as the destination and a long-word register as the source:

```
DIVL  RQ4, RR12
```

After the division, RR4 holds the remainder and RR6 holds the quotient.

The Multiply instructions (MULT, MULTL) perform signed two's-complement multiplication of word and long-word operands. MULT multiplies two 16-bit words and produces a 32-bit result; MULTL multiplies two 32-bit long words and produces a 64-bit result.

The Negate instructions (NEG, NEGB) perform a two's-complement negation on the contents of a register or memory location.

LOGICAL INSTRUCTIONS

Instructions that perform logical operations are listed in Table 8.6. These instructions set the Z and S flags based on the result of the logical operation. The byte instructions also use the P/V flag as a parity flag; the P flag is set if the result has even parity.

TABLE 8.6 LOGICAL INSTRUCTIONS

Mnemonics	Operands	Address modes	Clock cycles: Word/byte NS	Word/byte SS	Word/byte SL	Long word NS	Long word SS	Long word SL	Operation
AND	R, src	R	4	—	—				AND
ANDB		IM	7	—	—				R ← R AND src
		IR	7	—	—				
		DA	9	10	12				
		X	10	10	13				
COM,	dst	R	7	—	—				Complement
COMB		IR	12	—	—				dst ← NOT dst
		DA	15	16	18				
		X	16	16	19				
OR,	R, src	R	4	—	—				OR
ORB		IM	7	—	—				R ← R OR src
		IR	7	—	—				
		DA	9	10	12				
		X	10	10	13				
TCC,	cc, dst	R	5	—	—				Test Condition Code
TCCB									Set LSB if cc is true
TEST,	dst	R	7	—	—	13	—	—	Test
TESTB,		IR	8	—	—	13	—	—	dst OR 0
TESTL		DA	11	12	14	16	17	19	
		X	12	12	15	17	17	20	
XOR,	R, src	R	4	—	—				Exclusive OR
XORB		IM	7	—	—				R ← R XOR src
		IR	7	—	—				
		DA	9	10	12				
		X	10	10	13				

The two-operand instructions, And (AND, ANDB), Or (OR, ORB), and Exclusive-Or (XOR, XORB) perform the specified logical operation on the corresponding bits of the source and destination operands and load the result into the destination register.

The Complement instructions (COM, COMB) are a one's-complement operation; that is, all the bits holding 1's in the destination are changed to 0, and vice versa.

The Test instructions (TEST, TESTB, TESTL) perform a logical Or beween the destination and zero; the flags are set accordingly. This instruction is used to set the Z, S, and, for TESTB, P flags to reflect the contents of the destination: however, the destination itself is not affected.

The Test Condition Code instructions (TCC, TCCB) are used to create Boolean data variables based on the current flag settings. The flags in the FCW are checked to see if the specified condition code is true. If so, the least significant bit (bit 0) of the destination is set; if not, the destination is not affected. Bits other than bit 0 in the destination are never changed by a Test Condition Code instruction.

Except for TESTL, long-word operands are not supported by the logical instructions. However, logical operations on long words are easily implemented with pairs of instructions.

BIT MANIPULATION INSTRUCTIONS

The bit manipulation instructions (Table 8.7) are used to set, reset, or test individual bits in registers or memory locations. With most other processors, bit manipulation must be done using the logical operations with appropriate masks, which is awkward and inefficient.

The Bit Test instructions (BIT, BITB) test the specified bit in the destination for a 1 or 0, and set the Z flag appropriately. If testing a bit in a memory location, the number of the bit to be tested is given as an immediate operand between 0 and 7 for BITB and between 0 and 15 for BIT. If testing a bit in a register, the number of the bit to be tested can be an immediate operand or the contents of a word register.

In a similar manner, the Set Bit (SET, SETB) and Reset Bit (RES, RESB) instructions are used to set or reset any bit in a register or memory location. For destination operands in memory, the number of the bit to be set or reset is given as an immediate operand; for bits in registers, the number of the bit can be given as an immediate operand or the contents of a word register.

The Test and Set instruction (TSET) is provided to support imple-

TABLE 8.7 BIT MANIPULATION INSTRUCTIONS

			Clock cycles						
			Word/byte			Long word			
Mnemonics	Operands	Address modes	NS	SS	SL	NS	SS	SL	Operation
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
SET, SETB	dst, b	R	4	—	—				Set Bit Static Set dst bit specified by b
		IR	11	—	—				
		DA	13	14	16				
		X	14	14	17				
SET, SETB	dst, R	R	10	—	—				Set Bit Dynamic Set dst bit specified by contents of R
TSET, TSETB	dst	R	7	—	—				Test and Set S flag ← MSB of dst dst ← all 1s
		IR	11	—	—				
		DA	14	15	17				
		X	15	15	18				

mentation of multitasking operating systems. The most significant bit of the destination is copied into the S flag and then every bit in the destination is set to a 1.

The TSET instruction provides a locking mechanism for allocating resources within a multitasking system. For example, suppose that a Z8000-based system with a printer as an output peripheral is executing two users' tasks on a time-sharing basis. A register or memory location, called a semaphore, is used to indicate when a particular task is using the printer. The semaphore is tested each time a task requires the printer. If the semaphore is all 0's, the printer is not busy; if the semaphore is all 1's, the printer is busy (that is, it has been allocated for use by a particular task). When user A needs the printer, the operating system checks the semaphore using the TSET instruction and, if the printer is not busy, assigns the resource to task A. When task A is finished using the printer, the operating system resets the semaphore. Thus if user A is outputting a file, that output operation is completed before user B can access the printer.

The TSET instruction allows the operating system to check and set the semaphore in one uninterruptible step. In fact, even bus requests will not be honored in the time between checking the most significant bit in the semaphore and writing all 1's to the semaphore. If the test and set operations are executed in two separate instructions, conflicts are possible if an interrupt, bus request, or task switching under operating system control occurs between the two instructions.

ROTATE AND SHIFT INSTRUCTIONS

The rotate and shift instructions (Table 8.8) are used to rotate bits in byte or word registers and shift bits in byte, word, or long-word registers.

With the Rotate Left (RL, RLB) and Rotate Right (RR, RRB) instructions, bits in byte or word registers can be rotated by one or two bit positions. For Rotate Left, the most significant bit of the operand is loaded into the C flag as well as rotating into the least significant bit; for Rotate Right, the least significant bit is written to the C flag. Rotates that include the carry flag are also available: Rotate Left through Carry (RLC, RLCB) and Rotate Right through Carry (RRC, RRCB), as illustrated in Fig. 8.11.

The Rotate Left Digit (RLDB) and Rotate Right Digit (RRDB) instructions operate only on byte registers. BCD digits in the source register can be rotated right or left with these instructions (Fig. 8.12). For RLDB, the lower nibble of the source is moved to the upper nibble of the source; the upper nibble of the source is moved to the lower nibble of the destination; the lower nibble of the destination is moved to the lower nibble of the source. The upper nibble of the destination is unaffected. The RRDB instruction rotates nibble right in a similar fashion. If strings of BCD digits are to be

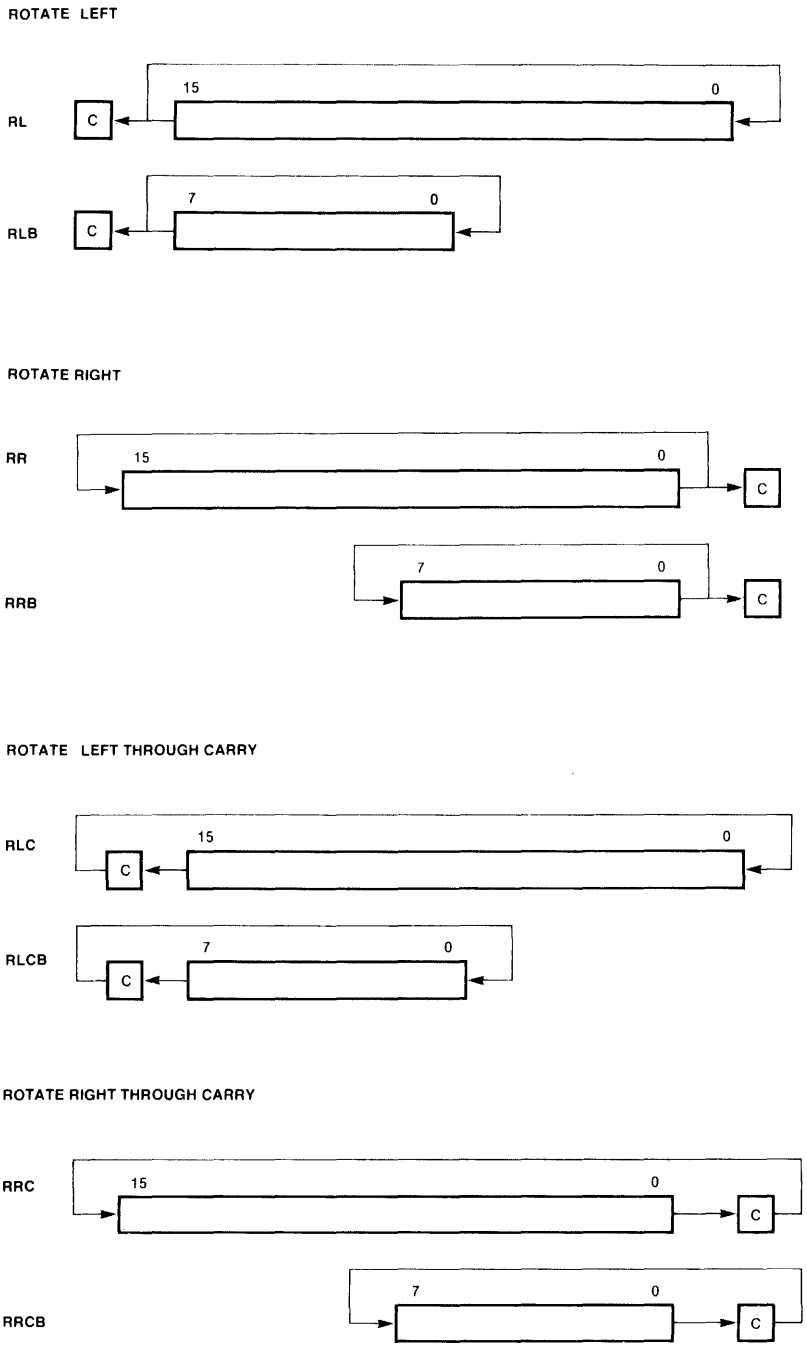


Figure 8.11 Rotate instructions.

TABLE 8.8 ROTATE AND SHIFT INSTRUCTIONS

Mnemonics	Operands	Address modes	Clock cycles: Word/byte NS	Word/byte SS	Word/byte SL	Long word NS	Long word SS	Long word SL	Operation
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

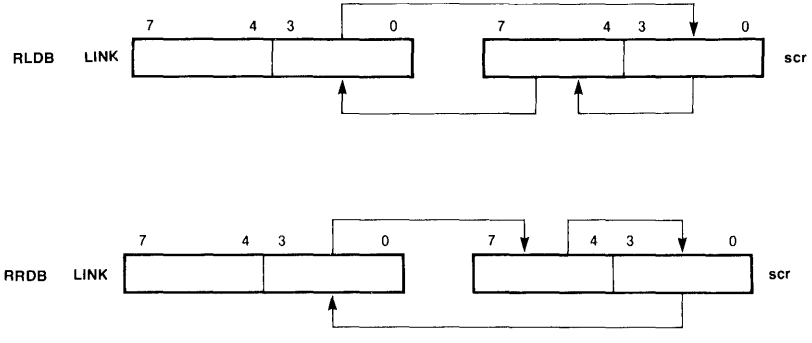


Figure 8.12 Rotate Left Digit and Rotate Right Digit instructions.

rotated, the destination register serves as a link between successive bytes of the string. This is analogous to the use of the carry flag in the RLC and RRC instructions.

A full set of both arithmetic and logical shifts is provided. For the arithmetic shifts, the most significant bit (the sign bit for signed two's-complement integers) is preserved when shifting right and 0's are shifted into the least significant bit when shifting left. For logical shifts, 0's are shifted into the most significant bit when shifting right and the least significant bit when shifting left (Fig. 8.13).

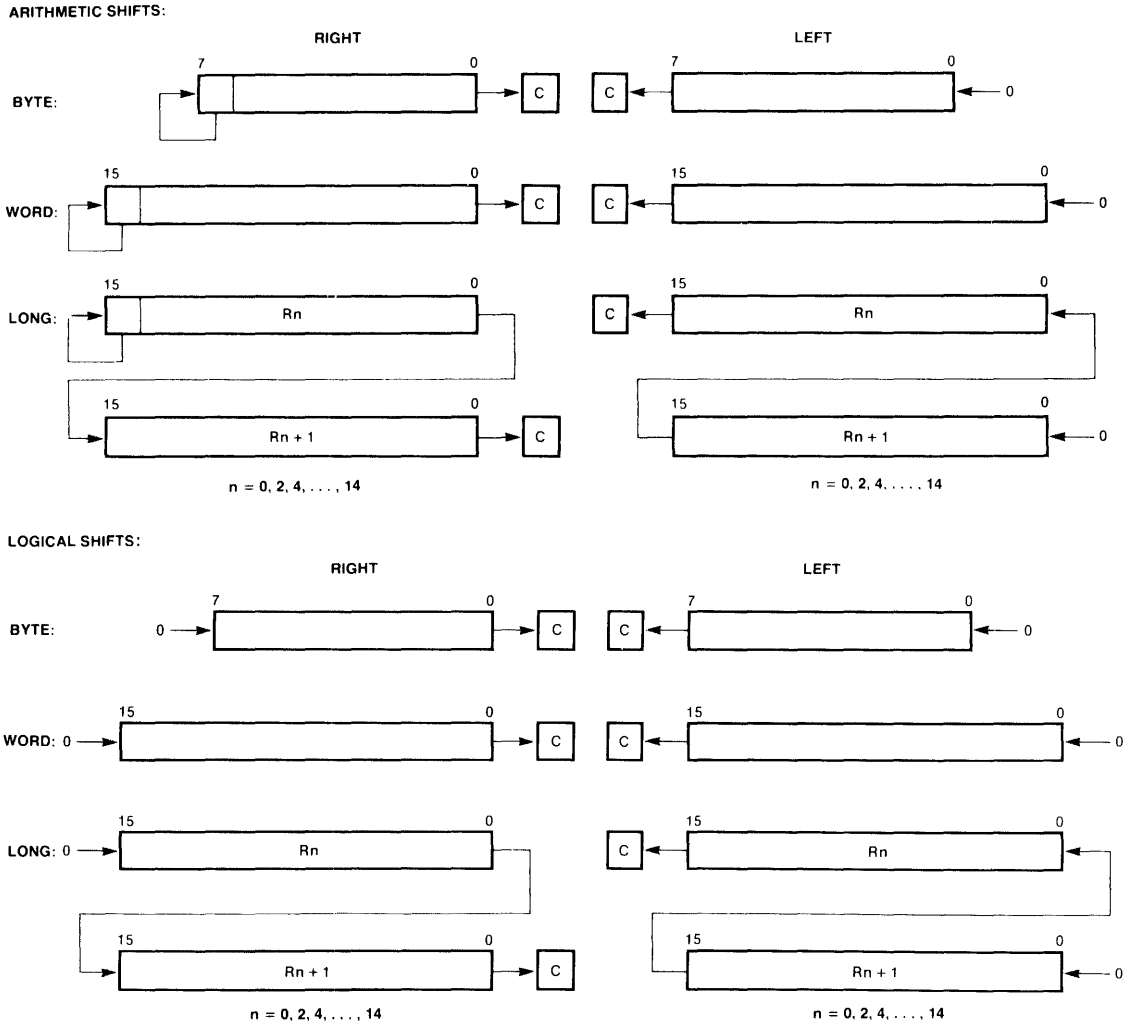


Figure 8.13 Arithmetic and logical shifts.

The Shift Dynamic Arithmetic (SDA, SDAB, SDAL) and Shift Dynamic Logical (SDL, SDLB, SDLL) instructions allow right or left shifts by the number of bits specified by the contents of the source operand, which is always a word register. The source is treated as a signed two's-complement value; positive values specify a left shift, whereas negative values specify a right shift. The shift count can range from -8 to +8 for byte operations, -16 to +16 for word operations, and -32 to +32 for long-word operations.

The remaining shift instructions require an immediate operand as the source. This operand determines the number of bit positions to be shifted, and range from 0 to 8 for byte operations, 0 to 16 for word operations, and 0 to 32 for long-word operations. These instructions include the Shift Left Arithmetic (SLA, SLAB, SLAL), Shift Left Logical (SLL, SLLB, SLLL), Shift Right Arithmetic (SRA, SRAB, SRAL), and Shift Right Logical (SRL, SRLB, SRLL) instructions. The only difference between the arithmetic and logical left shifts is the setting of the V flag.

PROGRAM CONTROL INSTRUCTIONS

The program control instructions (Table 8.9) are instructions that change the value of the program counter.

When the Call (CALL) or Call Relative (CALR) instruction is executed, the current PC value is pushed onto the stack using the implied stack pointer (R15 in nonsegmented mode, RR14 in segmented mode). The PC value pushed is the address of the next instruction following the subroutine call. The specified destination address is then loaded into the PC; the PC then points to the first instruction in the subroutine. For the CALR instruction, the destination address is calculated using relative addressing and must be in the range -4092 to +4098 bytes from the start of the CALR instruction. The CALR has a shorter opcode than the CALL instruction.

At the end of the procedure entered with a CALL or CALR, a Return (RET) instruction will pop the old PC value off the stack and resume execution at the instruction following the subroutine call, if the specified condition code is satisfied by the flags. If the condition code is not satisfied, the next instruction following the RET is executed.

The Decrement and Jump if Not Zero instructions (DJNZ, DBJNZ) are used to control execution of program loops. A word register (for DJNZ) or byte register (for DBJNZ) is used as a loop counter. A destination address is calculated using relative addressing. Execution of the DJNZ will cause the register contents to be decremented by one. If the contents of the register are not zero after the decrement, the destination address is loaded into the PC. When the register contents reach zero, control falls through to the next instruction after the DJNZ. The destination address must be in the range of -252 to +2 bytes from the start of the DJNZ instruction. Thus this instruction cannot be used to jump in a forward direction.

TABLE 8.9 PROGRAM CONTROL INSTRUCTIONS

Mnemonics[a]	Operands	Address modes	Clock cycles: Word/byte NS	Word/byte SS	Word/byte SL	Long word NS	Long word SS	Long word 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*	—	—	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 Conditional 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

[a] An asterisk indicates a privileged instruction.

The DJNZ and DBJNZ instructions provide a simple, efficient loop control method for Z8000 programs. For example, the code needed to execute a program loop exactly 50 times would read as follows:

```
        LDB  RH0, #50
             :
LOOP:   ! BODY OF THE LOOP !
             :
        DBJNZ RH0, LOOP
```

Although a decrement is performed during execution of the DJNZ, the flags are not affected by this instruction. Thus this instruction can control loops used to implement multiple-precision arithmetic operations, without

having to save the flags before checking the end-of-loop condition and restoring them afterwards.

The Jump (JP) and Jump Relative (JR) instructions load the PC with the destination address if the condition code is true. If the condition code is not satisfied by the flags, the instruction following the jump will be executed. For the JR instruction, relative addressing is used to calculate the destination address, which must be in the range of -254 to +256 bytes from the start of the JR instruction.

The Interrupt Return instruction (IRET) is used to return to the interrupted task after executing an interrupt or trap service routine, as described in Chapter 6. The identifier word associated with the interrupt or trap is popped from the system stack and discarded. Then the old program status values for the interrupted task are popped and loaded into the FCW and PC. The new value for the FCW is not effective until the next instruction, so the CPU's status pins will not be affected by the new control bits until after the IRET instruction execution is complete. IRET is a privileged instruction and, therefore, can only be executed in system mode. A Z8001 must be in segmented mode when an IRET is performed.

The System Call instruction (SC) causes the CPU to process a system call trap, as described in Chapter 6. This instruction allows normal-mode users access to the operating system functions in the system call trap service routine. The SC instruction requires an immediate operand between 0 and 255; this operand is encoded into the opcode that is saved on the stack as the identifier word when processing the system call trap.

BLOCK MOVE INSTRUCTIONS

The block move instructions (Table 8.10) provide memory-to-memory transfers wherein a byte or word string of data of any length up to 64K bytes can be transferred. In these instructions, the address in memory of the source and destination operands are stored in registers (word registers in the nonsegmented mode, register pairs in the segmented mode) and indirect register addressing is used. The registers used as memory pointers are automatically incremented or decremented during instruction execution; thus after each element of the string is transferred, the pointer register is updated to address the next element of the string.

The Load and Decrement instructions (LDD, LDDB) transfer a byte or word of data, and then decrement the source and destination registers (by 1 for LDDB and by 2 for LDD). The Load and Increment instructions (LDI, LDIB) increment the registers that hold the source and destination addresses after performing the transfer. (In segmented mode, only the low-order half of the register pair that holds the memory address is incremented or decremented; the segment number is not affected by the address calculations.) These instructions typically are used in a program loop, where a string of

TABLE 8.10 BLOCK MOVE INSTRUCTIONS

Mnemonics	Operands	Address modes	Clock cycles: Word/byte NS	Word/byte SS	Word/byte SL	Long word NS	Long word SS	Long word SL	Operation
LDD, LDDB	dst, src, R	IR	20	—	—				Load and Decrement dst ← src Autodecrement dst and src addresses R ← R − 1
LDDR, LDDRB	dst, src, R	IR	(11 + 9n)						Load, Decrement and Repeat dst ← src Autodecrement dst and src addresses R ← R − 1 Repeat until R = 0
LDI, LDIB	dst, src, R	IR	20	—	—				Load and Increment dst ← src Autoincrement dst and src addresses R ← R − 1
LDIR, LDIRB	dst, src, R	IR	(11 + 9n)						Load, Increment and Repeat dst ← src Autoincrement dst and src addresses R ← R − 1 Repeat until R = 0

data is being moved in memory and other operations are contained in the loop. The third operand in these instructions is a word register that holds a count that is decremented each time the instruction is executed.

The Load, Increment and Repeat (LDIR, LDIRB) and Load, Decrement and Repeat (LDDR, LDDRB) instructions are automatically repeating forms of the block transfer instructions. These instructions are, in essence, a one-instruction loop. The third operand is a word register that holds the count of how many times the instruction is to be executed. Thus a block of 64K bytes of data can be relocated within memory with one Z8000 instruction.

The automatically repeating forms of the block move instructions are interruptible at their elementary level, that is, after each iteration of the byte or word transfer. The address saved on the stack during exception processing would be the address of the block move instruction itself. Of course, the service routine should not alter any of the registers being used by the block move to hold memory addresses or the repetition count.

BLOCK COMPARE INSTRUCTIONS

Two types of block compare instructions are provided: one for comparing the elements of a string of bytes or words in memory to the contents of a

register, and another for comparing the corresponding elements of two strings in memory (Table 8.11).

The Compare and Decrement instructions (CPD, CPDB) use four operands. The first is the register whose contents are compared to the memory location specified by the second operand using indirect register addressing. The third operand is a word register that holds a count and the fourth is a condition code specifying the flag settings to be examined after each compare operation. Each time the instruction is executed, the register used to hold the memory address in the string is decremented. The Compare and Increment instructions (CPI, CPIB) are similar, with the memory address being incremented after each execution. These instructions would be used in a program loop to compare a string of data in memory to the destination register's contents. The Z flag is set if the condition code is satisfied as the result of a comparison.

The Compare, Decrement and Repeat (CPDR, CPDRB) and Compare, Increment and Repeat (CPIR, CPIRB) are automatically repeating forms of these instructions. The instruction repeats until the condition code is met during a comparison or the count is exhausted. The Z and V flags indicate which condition caused the instruction to terminate. The repeating instructions are interruptible after each iteration of the instruction.

The Compare String and Decrement instructions (CPSD, CPSDB) are used to compare corresponding elements in two strings of data in memory. Both strings are referenced using indirect register addressing. The third operand is a word register that holds a count and the fourth operand is the condition code for the comparison. After each execution, the Z flag indicates if the condition code is met, and the addresses in the source and destination registers are decremented to point to the next element in their respective strings. The Compare String and Increment instructions (CPSI, CPSIB) are similar, with the addresses in the registers being incremented after each execution. These instructions typically are used in program loops that operate on two strings.

The Compare String, Decrement and Repeat (CPSDR, CPSDRB) and Compare String, Increment and Repeat (CPSIR, CPSIRB) are automatically repeating forms of the string compare. The instruction repeats until the condition code is met or the count is exhausted, as indicated by the Z and V flags. These repeating instructions are interruptible after each iteration of their execution.

BLOCK TRANSLATE INSTRUCTIONS

The block translate instructions (Table 8.12) operate only on byte strings in memory. One set of instructions is used to translate a string of bytes from one code to another; another set of block translate instructions is used to

TABLE 8.11 BLOCK COMPARE INSTRUCTIONS

Mnemonics	Operands	Address modes	Clock cycles: Word/byte NS	Word/byte SS	Word/byte SL	Long word NS	Long word SS	Long word SL	Operation
CPD, CPDB	R_X, src, R_Y, cc	IR	20	—	—				Compare and Decrement R_X − src Autodecrement src address $R_Y \leftarrow R_Y - 1$
CPDR, CPDRB	R_X, src, R_Y, cc	IR	(11 + 9n)						Compare, Decrement and Repeat R_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, Decrement 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, Increment and Repeat dst − src Autoincrement dst and src addresses R ← R − 1 Repeat until cc is true or R = 0

TABLE 8.12 BLOCK TRANSLATE INSTRUCTIONS

Mnemonics	Operands	Address modes	Clock cycles: Word/byte NS	Word/byte SS	Word/byte SL	Long word NS	Long word SS	Long word SL	Operation
TRDB	dst, src, R	IR	25	—	—				Translate and Decrement dst ← src (dst) Autodecrement dst address R ← R − 1
TRDRB	dst, src, R	IR	(11 + 14n)						Translate, Decrement and Repeat dst ← src (dst) Autodecrement dst address R ← R − 1 Repeat until R = 0
TRIB	dst, src, R	IR	25	—	—				Translate and Increment dst ← src (dst) Autoincrement dst address R ← R − 1
TRIRB	dst, src, R	IR	(11 + 14n)						Translate, Increment and Repeat dst ← src (dst) Autoincrement dst address R ← R − 1 Repeat until R = 0
TRTDB	src1, src2, R	IR	25	—	—				Translate and Test, Decrement RH1 ← src 2 (src 1) Autodecrement src 1 address R ← R − 1
TRTDRB	src1, src2, R	IR	(11 + 14n)						Translate and Test, Decrement and Repeat RH1 ← src 2 (src 1) Autodecrement src 1 address R ← R − 1 Repeat until R = 0 or RH1 = 0
TRTIB	src1, src2, R	IR	25	—	—				Translate and Test, Increment RH1 ← src 2 (src 1) Autoincrement src 1 address R ← R − 1
TRTIRB	src1, src2, R	IR	(11 + 14n)						Translate and Test, Increment and Repeat RH1 ← src 2 (src 1) Autoincrement src 1 address R ← R − 1 Repeat until R = 0 or RH1 = 0

scan a string of bytes for elements with a special meaning. All of the block translate instructions use byte register RH1 as a temporary storage area during their execution.

The Translate and Decrement instruction (TRDB) has three operands. The source and destination operands are in memory and are specified using indirect register addressing. The third operand is a count that is decremented each time the instruction is executed. The location addressed by the destination register is called the target byte and its contents are used as an index into a table of translation values whose base address is in the source register. The element of the table whose address is found by adding the target byte to the base address of the table replaces the target byte at the destination address. The destination register is then decremented by one to point to the next element of the string to be translated.

Suppose that a string of bytes holding ASCII characters is to be translated into the corresponding EBCDIC characters. A table is built in memory wherein each EBCDIC character is placed at the table location corresponding to the ASCII value for that character. For example, the thirtieth (hexadecimal) element of the table should contain the EBCDIC code for a "0," since 30 (hex) is the ASCII code for a "0." The base address (that is, lowest address) of the table is loaded into the source register for the TRDB instruction. The highest address in the ASCII string to be translated is loaded into the destination register and the TRDB instruction is used to perform the translation. The destination register is automatically decremented to point to the next byte to be translated.

The Translate and Increment instruction (TRIB) is similar, with the destination register being incremented instead of decremented after each execution. TRDB and TRIB typically are used in a program loop to translate an entire string of data.

The Translate, Decrement and Repeat (TRDRB) and Translate, Increment and Repeat (TRIRB) are automatically repeating forms of the translate instructions. The instruction repeats until the contents of the count register reach zero. These instructions are interruptible after each iteration of the operation.

The Translate, Test and Decrement instruction (TRTDB) works in a similar manner and is used to scan a string for special characters. The contents of the location addressed by the first source register is used as an index into a table of values whose base address is contained in the second source register. This element of the table is loaded into RH1 and, if it is zero, the Z flag is set. The contents of the location addressed by the source are not altered by this instruction. The first source register is then decremented to point to the next element of the string being scanned. This instruction typically is used in a loop, where a string is scanned until a nonzero value is found in the table. The Translate, Test and Increment instruction (TRTIB) is similar, except the first source address is incremented.

The repeating forms of these instructions are Translate, Test, Decrement and Repeat (TRTDRB) and Translate, Test, Increment and Repeat (TRTIRB). These instructions repeat until either the count reaches zero or a nonzero value in the table is accessed and loaded into RH1. Thus the user can build a table that marks special characters by having a nonzero value in the appropriate position in the table. A string of characters can be scanned with one of these instructions, searching for the special characters. The repeating forms are interruptible after each iteration of the execution.

I/O INSTRUCTIONS

The instructions used to access the standard I/O address space are listed in Table 8.13. All I/O instructions are privileged, meaning that they can be executed in the system mode only.

The Input instructions (IN, INB) are used to input a byte or word of data from an I/O device. The destination is always a register. The I/O port address can be specified using the direct address or indirect register addressing modes. No flags are affected.

For the Input and Decrement instructions (IND, INDB), the destination is a memory location and the source is an I/O port. Both the memory and I/O port address are specified using indirect register addressing. After execution, the destination register is decremented to point at the next lower memory location, in preparation for another execution of the instruction. Thus this instruction would be used in a program loop to load strings of data from an I/O port to consecutive memory locations. The third operand is a word register that holds a count that is decremented each time the instruction is executed. The Input and Increment instructions (INI, INIB) are similar, with the destination register being incremented each time the instruction is executed.

The Input, Decrement and Repeat (INDR, INDRB) and Input, Increment and Repeat (INIR, INIRB) are automatically repeating forms of the input instruction. The instruction repeats until the count reaches zero; therefore, up to 64K bytes of data can be read from an I/O port and placed in memory with one instruction.

The Output instruction (OUT, OUTB) is used to write a byte or word of data from a register to a peripheral device. Block output instructions are available for writing data in consecutive memory locations to an I/O port. These are analogous to the input commands, and include the Output and Decrement (OUTD, OUTDB), Output and Increment (OUTI, OUTIB), Output, Decrement and Repeat (OTDR, OTDRB), and Output, Increment and Repeat (OTIR, OTIRB) commands.

All the repeating I/O instructions are interruptible after each iteration of the instruction. For byte I/O operations, the I/O port address will deter-

TABLE 8.13 I/O INSTRUCTIONS

Mnemonics[a]	Operands	Address modes	Clock cycles: Word/byte NS	Word/byte SS	Word/byte SL	Long word NS	Long word SS	Long word SL	Operation
IN*, INB*	R, src	IR DA	10 12	— —	— —				Input R ← src
IND*, INDB*	dst, src, R	IR	21	—	—				Input and Decrement dst ← src Autodecrement dst address R ← R − 1
INDR*, INDRB*	dst, src, R	IR	(11 + 10n)						Input, Decrement and Repeat dst ← src Autodecrement dst address R ← R − 1 Repeat until R = 0
INI*, INIB*	dst, src, R	IR	21	—	—				Input and Increment dst ← src Autoincrement dst address R ← R − 1
INIR*, INIRB*	dst, src, R	IR	(11 + 10n)						Input, Increment and Repeat dst ← src Autoincrement dst address R ← R − 1 Repeat until R = 0
OUT*, OUTB*	dst, R	IR DA	10 12	— —	— —				Output dst ← R
OUTD*, OUTDB*	dst, src, R	IR	21	—	—				Output and Decrement dst ← src Autodecrement src address R ← R − 1
OTDR*, OTDRB*	dst, src, R	IR	(11 + 10n)						Output, Decrement and Repeat dst ← src Autodecrement src address R ← R − 1 Repeat until R = 0
OUTI*, OUTIB*	dst, src, R	IR	21	—	—				Output and Increment dst ← src Autoincrement src address R ← R − 1
OTIR*, OTIRB*	dst, src, R	IR	(11 + 10n)						Output, Increment and Repeat dst ← src Autoincrement src address R ← R − 1 Repeat until R = 0

[a] As indicated by the asterisks, all I/O instructions are privileged.

mine which half of the address/data bus is used for the byte data transfer, as explained in Chapter 4. Byte data are transferred on AD0–AD7 for odd port addresses, and on AD8–AD15 for even port addresses.

SPECIAL I/O INSTRUCTIONS

The special I/O instructions (Table 8.14) are used to access the peripherals in the special I/O address space and are identical in format to the I/O instructions described above. In fact, the only difference between executing an I/O instruction and the corresponding special I/O instruction is the status code emitted on the ST0–ST3 lines during I/O access machine cycles. Note that the Special Input (SIN) and Special Output (SOUT) instructions do not support the indirect register addressing mode for specifying I/O port addresses. All special I/O instructions are privileged instructions.

CPU CONTROL INSTRUCTIONS

The CPU control instructions (Table 8.15) are instructions that operate on the CPU control registers (the FCW, PC, and refresh register) or perform other CPU-related functions.

Several instructions can be used to manipulate the flags in the FCW. The Set Flag instruction (SETFLG) is used to load a 1 into any combination of the C, S, Z, or P/V flags. The operands are the flags themselves; one, two, three, or four operands can be specified, in any order:

```
SETFLG  Z, V      ! set the zero and overflow flags !
```

In a similar manner, the Reset Flag instruction (RESFLG) is used to clear any combination of the C, S, Z, or P/V flags to zero. The complement flag instruction (COMFLG) complements any combination of those four flags (that is, each 1 is changed to a 0, and vice versa).

The entire low-order byte of the FCW (the byte that holds the flags) can be read to or written from any byte register using the Load Control Byte instruction (LDCTLB), as follows:

```
LDCTLB  RH0, FLAGS     ! load the flag byte of the FCW into RH0 !
LDCTLB  FLAGS, RL6     ! load contents of RL6 into low half of FCW !
```

The Disable Interrupt instruction (DI) is used to disable vectored interrupts and nonvectored interrupts by writing 0's to the appropriate bits in the FCW. Either vectored interrupts, nonvectored interrupts, or both can be

TABLE 8.14 SPECIAL I/O INSTRUCTIONS

Mnemonics[a]	Operands	Address modes	Clock cycles: Word/byte NS	Word/byte SS	Word/byte SL	Long word NS	Long word SS	Long word SL	Operation
SIN*, SINB*	R, src	DA	12	—	—				Special Input R ← src
SIND*, SINDB*	dst, src, R	IR	21	—	—				Special Input and Decrement dst ← src Autodecrement dst address R ← R − 1
SINDR*, SINDRB*	dst, src, R	IR	(11 + 10n)						Special Input, Decrement and Repeat dst ← src Autodecrement dst address R ← R − 1 Repeat until R = 0
SINI*, SINIB*	dst, src, R	IR	21	—	—				Special Input and Increment dst ← src Autoincrement dst address R ← R − 1
SINIR*, SINIRB*	dst, src, R	IR	(11 + 10n)						Special Input, Increment and Repeat dst ← src Autoincrement dst address R ← R − 1 Repeat until R = 0
SOUT*, SOUTB*	dst, src	DA	12	—	—				Special Output dst ← src
SOUTD*, SOUTDB*	dst, src, R	IR	21	—	—				Special Output and Decrement dst ← src Autodecrement src address R ← R − 1
SOTDR*, SOTDRB*	dst, src, R	IR	(11 + 10n)						Special Output, Decrement and Repeat dst ← src Autodecrement src address R ← R − 1 Repeat until R = 0
SOUTI*, SOUTIB*	dst, src, R	IR	21	—	—				Special Output and Increment dst ← src Autoincrement src address R ← R − 1
SOTIR*, SOTIRB*	dst, src, R	R	(11 + 10n)						Special Output, Increment and Repeat dst ← src Autoincrement src address R ← R − 1 Repeat until R = 0

[a] All special I/O instructions are privileged.

TABLE 8.15 CPU CONTROL INSTRUCTIONS

Mnemonics[a]	Operands	Address modes	Clock cycles: Word/byte NS	Word/byte SS	Word/byte SL	Long word NS	Long word SS	Long word SL	Operation
COMFLG	flags	—	7	—	—				Complement Flag (Any combination of C, Z, S, P/V)
DI*	int	—	7	—	—				Disable Interrupt (Any combination of NVI, VI)
EI*	int	—	7	—	—				Enable Interrupt (Any combination of NVI, VI)
HALT*	—	—	(8 + 3n)						HALT
LDCTL*	CTLR, src	R	7	—	—				Load into Control Register CTLR ← src
LDCTL*	dst, CTLR	R	7	—	—				Load from Control Register dst ← CTLR
LDCTLB	flags, src	R	7	—	—				Load into Flag Byte Register FLGR ← src
LDCTLB	dst, flags	R	7	—	—				Load from Flag Byte Register dst ← FLGR
LDPS*	src	IR	12	—	16				Load Program Status PS ← src
		DA	16	20	22				
		X	17	20	23				
MBIT*	—	—	7	—	—				Test Multi-Micro Bit Set S if $\overline{M}_I$ is Low; reset S if $\overline{M}_I$ is High
MREQ*	dst	R	(12 + 7n)						Multi-Micro Request
MRES*	—	—	5	—	—				Multi-Micro Reset
MSET*	—	—	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)

[a] An asterisk indicates a privileged instruction.

disabled with a single DI instruction:

```
DI  NVI, VI     ! disable vectored and nonvectored interrupts !
```

The Enable Interrupt instruction (EI) is the complement of the DI instruction. It is used to enable vectored and nonvectored interrupts by setting the appropriate bits in the FCW to 1's. EI and DI are privileged instructions.

The LDCTL instruction (LDCTL) is a privileged instruction that is used

to transfer words between any general-purpose register and the CPU control registers. LDCTL can read or write to the FCW, refresh register, PSAP segment number (Z8001 only), or PSAP offset. When loading control registers, bits marked "reserved" in the destination should be loaded with 0's. The value of those bits when read from control registers and loaded into general-purpose registers is undefined. Only the row counter portion (bits 0-8) of the refresh register can be read. The normal-mode implied stack pointer can be accessed while in the system mode with the LDCTL instruction. This allows the operating system to initialize the implied stack pointers for normal-mode users. Table 8.16 shows examples of the LDCTL instruction.

The Load Program Status instruction (LDPS) loads new program status (an FCW and PC value) from the memory area specified as the operand, as discussed in Chapter 6. The old program status is not saved, so execution of the LDPS instruction causes a permanent change in the program environment. LDPS is a privileged instruction.

The Multi-Micro Request (MREQ), Multi-Micro Bit (MBIT), Multi-Micro Set (MSET), and Multi-Micro Reset (MRES) instructions are privileged instructions that act on the $\overline{MI}$ and $\overline{MO}$ CPU signals. These signals usually are used to implement resource sharing in multiprocessor systems, as described in Chapter 7.

TABLE 8.16 USE OF THE LDCTL INSTRUCTION

```
LDCTL   FCW, R0          ! write to FCW !
LDCTL   R0, FCW          ! read from FCW !

LDCTL   REFRESH, R0      ! write to refresh register !
LDCTL   R0, REFRESH      ! read from refresh register—can only read row address portion
                           (bits 0-8) !

LDCTL   PSAPSEG, R0      ! write segment number in program status area pointer
LDCTL   R0, PSAPSEG      ! read segment number portion of PSAP !

LDCTL   PSAPOFF, R0      ! write to offset portion of PSAP !
LDCTL   R0, PSAPOFF      ! read offset portion of PSAP !

LDCTL   NSPSEG, R0       ! write segment-number portion of normal-mode stack pointer
                           (normal-mode R14) while in system mode !

LDCTL   R0, NSPSEG       ! read segment number in normal-mode stack pointer while in
                           system mode !

LDCTL   NSPOFF, R0       ! write offset portion of normal-mode stack pointer (normal-
                           mode R15) while in system mode !

LDCTL   R0, NSPOFF       ! read offset in normal-mode stack pointer while in system
                           mode !
```

The No Operation instruction (NOP) does not perform any operations, as the name implies.

The Z8000 instruction set can be extended by the addition of Extended Processing Units (EPUs) to a system. Opcodes that begin with 0E, 0F, 4E, 4F, 8E, and 8F (hexadecimal) are extended instructions that are executed by EPUs. The five basic addressing modes (R, DA, IM, IR, and X) can be used in extended instructions to specify data operands. The extended processor architecture is described in Chapter 10.

9

The Z8010 Memory Management Unit

The Z8001 CPU is capable of addressing up to 8 megabytes of memory per memory address space. The 8M bytes of memory in each address space are partitioned into 128 segments, where each segment can hold up to 64K bytes of memory. This large addressing capability and the other powerful features of the Z8000 architecture allow the Z8001 CPU to be used in microcomputer systems that support sophisticated operating systems, complex programs, large data bases, and the use of high-level languages.

Within a computer system, the operating system controls the allocation of resources among the programming tasks being executed on the system, as described in Chapter 1. One of the major resources in a computer system is its memory. The efficient allocation of memory resources is critical, especially in systems where a large number of tasks are competing for the use of a limited amount of physical memory. Furthermore, the operating system may want to protect the integrity of the system by limiting access to various portions of memory. For example, users' programs might not be allowed to access the memory that holds the code for the operating system. Thus the management of memory resources in a computer system involves both the allocation of memory for the various tasks executing on the system and the protection of memory to prevent illegal accesses and maintain system integrity. For Z8001-based systems, the Z8010 Memory Management Unit (MMU) is a programmable device that can be used to support memory management functions.

MEMORY ALLOCATION

The Z8010 MMU facilitates control of memory resource allocation by translating logical addresses to physical addresses. Logical addresses are the mem-

ory addresses manipulated within a program and emitted by the CPU during execution. Physical addresses are the memory addresses seen at the memory control logic and input to the acutal memory devices (RAM, ROM, or PROM). In simple systems without memory management capability, the logical and physical addresses are the same. In more complex systems, a memory manager is used to translate logical addresses into different physical addresses. The memory management unit stands between the CPU and memory in a system, accepting logical addresses from the CPU and outputting the corresponding physical addresses to memory. Thus a given programming task can be relocated anywhere within physical memory by the memory manager through address translation.

This separation of logical and physical addresses is necessary in most multitasking systems. The amount of memory in a computer system is finite. In multitasking systems, the memory requirements for all the tasks in the system usually exceeds the amount of physical memory in that system. However, not every task's code and data are needed in main memory at the same time; secondary storage devices such as tapes and disks can be used to store tasks not currently being executed. When a task's code or data is needed, it can be written from the secondary storage device into memory. Of course, the operating system must keep track of which memory areas are being used for each task running on the system at any given time. With a memory manager that provides address translation, a new task can be placed in any "open" area of memory (that is, an area not currently being used by some other task).

The ability to relocate tasks anywhere within memory can significantly increase the performance and flexibility of a system. An individual user does not need to be concerned that the logical addresses within his program are the same as the logical addresses for another program on the system; the memory manager will route each task's logical addresses to different areas of physical memory. The cumbersome techniques of reserving fixed areas of memory for overlays in a large program can be replaced by more efficient algorithms using the memory manager to relocate program code. Separating logical and physical addresses also facilitates the sharing of code or data between two different tasks, since two or more logical addresses can be mapped to the same physical address.

SEGMENTATION AND MEMORY ALLOCATION

Complex programs written as one large, monolithic block are difficult, if not impossible, to write, debug, and maintain. Modern structured programming techniques require programmers to partition large programs into smaller, easily-managed subparts, with each subpart having a well-defined interface to other parts of the program. Highly structured programming languages such as Pascal and Ada are based on such a scheme. The segmentation of mem-

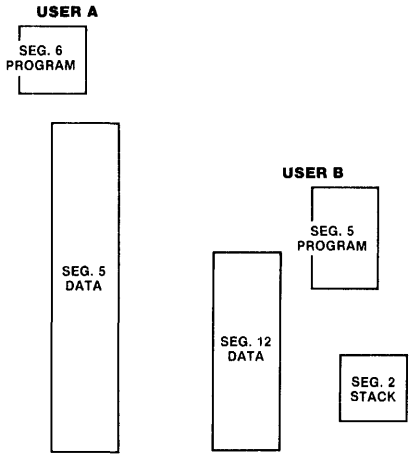


Figure 9.1 Two users' logical address spaces.

ory spaces in a Z8001-based system provides support at the hardware level for partitioning programs; each code, data, and stack area in a program can reside in its own memory segment.

Furthermore, memory segmentation provides a basis for address relocation within the memory manager. The Z8010 MMU translates logical to physical addresses on a segment-by-segment basis. For example, Fig. 9.1 shows the logical addresses for two users of a Z8001 system. User A has specified that task A's program code resides in segment 6, and the data in segment 5. User B has specified segment 5 for task B's code, segment 12 for the data, and segment 2 for the stack. Note that both users have named one of their segments "segment 5," but they refer to completely different memory areas. Figure 9.2 illustrates one way that these users' segments could be mapped into physical memory. The dashed lines indicate the mapping of logical to physical addresses by the memory manager. The segments are logically distinct; a reference to one segment cannot inadvertently result in an access to another segment.

In the the Z8010 MMU, each memory segment is assigned to some area of physical memory (Fig. 9.3). Segments can be of variable size, up to 64K bytes per segment. Address translation is performed by adding the offset portion of the logical address to the starting physical address of the segment. Thus when a logical address of the form ⟨⟨A⟩⟩B is emitted by the CPU, the segment name "A" is used by the MMU to determine where segment A begins in physical memory. If segment A resides in locations 10000 to 25000, the physical address corresponding to logical address ⟨⟨A⟩⟩B would be location 10000 + B. In other words, the logical segment number is just a name corresponding to some physical address in memory, and the logical offset is a displacement from that address.

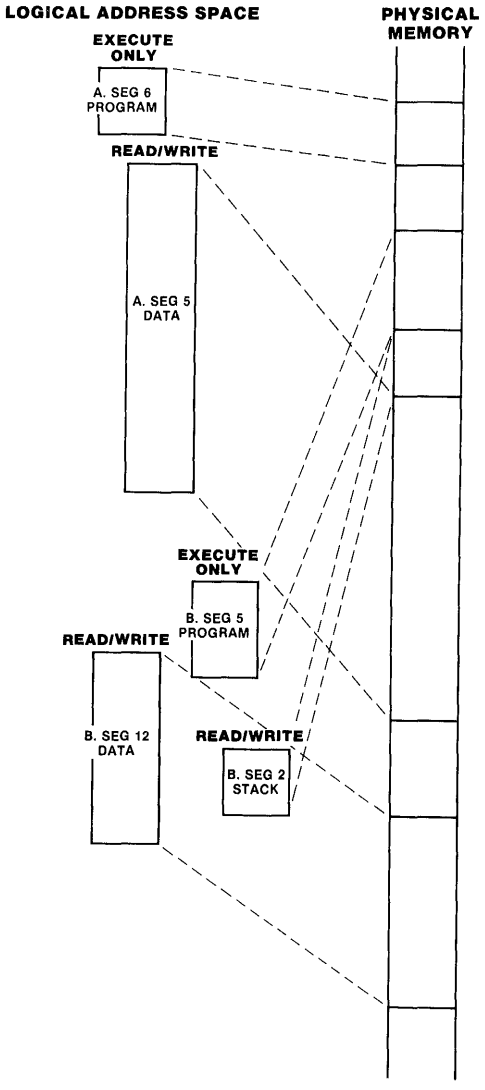


Figure 9.2 Mapping logical addresses to physical addresses.

Within the Z8010 MMU, special registers are used to hold the starting address in physical memory for each segment. The offset address emitted by the CPU during T1 of a memory access cycle is added to this starting address for the segment to produce the physical address output by the MMU. This address translation is similar to indexed addressing, where the contents of the MMU register that holds the starting address for the segment is used as a base address and the offset portion of the logical address is added to that base to yield the physical address.

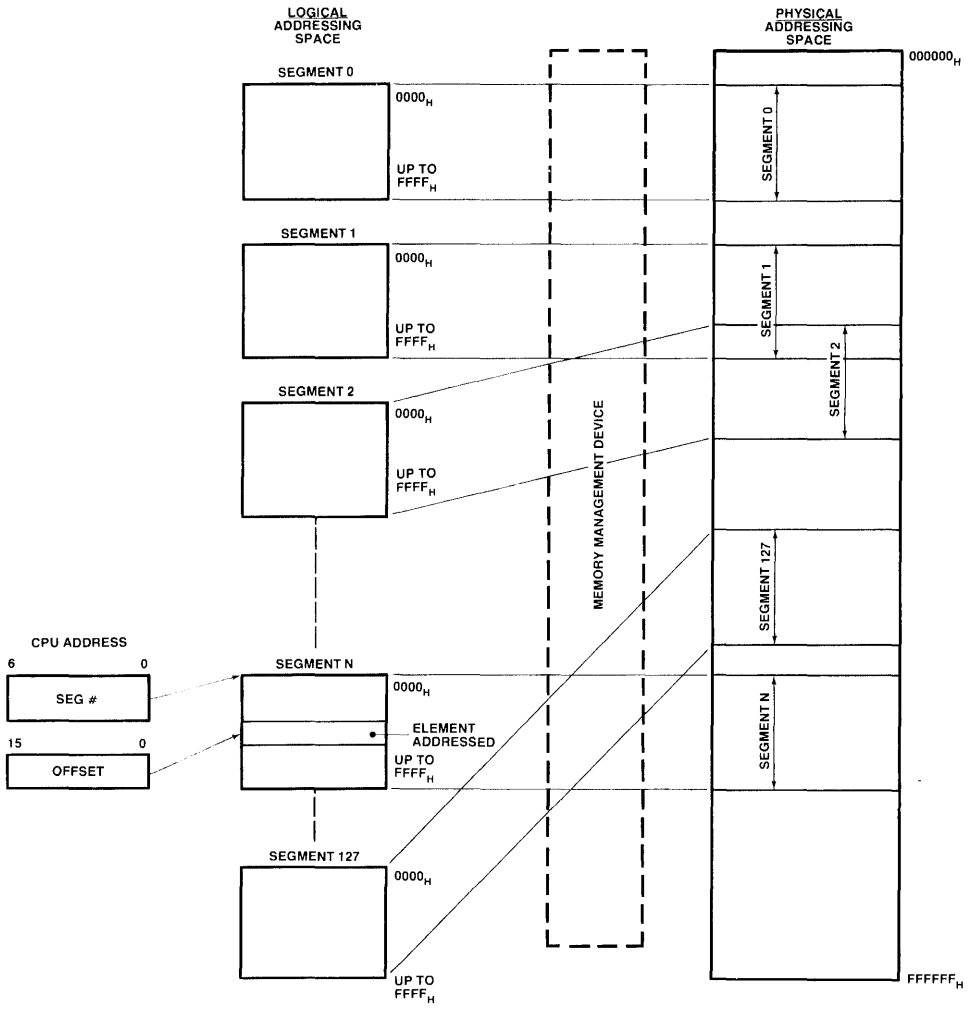


Figure 9.3 Memory mapping of segments with a memory manager.

MEMORY PROTECTION

Once the program has been partitioned into separate memory segments, it is highly desirable to assign attributes to each segment. For example, attributes have been associated with each segment in Fig. 9.2 based on how that segment is to be used during task execution; segments that hold program code are designated "execute only," and segments that hold data are "read/write" segments. Since the MMU is between the CPU and memory, it can block illegal memory accesses based on the attributes assigned to each segment. Thus

users A and B are prevented from executing a data segment or writing into a code segment.

More specifically, memory protection by the Z8010 MMU is accomplished by memory attribute checking. Attributes are assigned on a segment-by-segment basis. A segment's attributes determine who can access that segment and what types of accesses are allowed. Each memory reference is checked to ensure that the task has the right to access the data in that fashion.

A number of attributes can be associated with a segment and checked during accesses to that segment. The length of a segment can be designated in the MMU, and references to each segment are checked to ensure that they fall within the boundaries of that segment. Segments can be assigned a "read-only" attribute; this is used to prevent modification of data elements or to protect the integrity of program code that is not self-modifying. The "execute-only" attribute means that the segment can be accessed only during instruction fetch cycles (including loads that use the relative addressing mode) and is useful for guarding proprietary software. The "system-only" attribute prevents normal-mode programs from accessing segments that are reserved for operating system code or data. To check these attributes, the MMU must sample the status lines from the Z8001 CPU that define the type of the current transaction.

The Z8010 MMU stores each logical segment's attributes in an internal register (one per segment) and checks those attributes each time the segment is accessed. If a memory access that violates the attributes for a segment is detected, the MMU notifies the CPU by making a segmentation trap request on the Z8001's $\overline{\text{SEGT}}$ input. The segmentation trap service routine can read status registers in the MMU to determine the exact cause of the trap. When asserting a segment trap request, the MMU also generates a signal to memory ($\overline{\text{SUP}}$) that can be used by memory control logic to inhibit an erroneous memory write.

The MMU stores other status information together with the attributes for each segment. This information includes flags that indicate if a segment has been referenced or modified while resident in main memory. If a segment is to be written to a secondary storage device, such as a disk, in order to make room in main memory for another task, these flags will indicate if that segment was modified since the last time it was read into memory from the disk. If the segment was modified, the updated version of that segment must be written to the disk; if not, the copy of the segment's contents on the disk is still valid and rewriting the segment to the disk is not necessary. Obviously, such status information can improve the performance of the entire system.

In summary, memory management involves the allocation and protection of the system's memory resources. In Z8001-based systems, a memory management system can be implemented using operating system software and the Z8010 Memory Management Unit. The Z8010 MMU controls memory allo-

cation by translating logical addresses from the CPU into physical addresses for the memory devices. Memory protection is provided by attribute checking in the MMU.

The segmented address translation mechanism with attribute checking provides all the benefits of an efficient memory management system. Memory can be allocated dynamically during task execution; that is, a task may be located anywhere in physical memory and even moved when its execution is suspended. Moving tasks to different locations requires only changing the address mapping within the MMU. This flexibility is possible since the program deals exclusively with logical addresses that are independent from the physical addresses accessed during execution. Furthermore, sharing of common memory areas by different tasks is accomplished easily by mapping each task's logical addresses to the same physical address.

Obvious execution errors can be avoided through the assignment of attributes to memory segments. The MMU will notify the CPU when illegal accesses are attempted, such as exceeding the boundaries of a segment or writing to read-only memory. The segregation of the operating system from the users' applications programs is facilitated by the "system-only" attribute.

Segmentation and memory management support the development of large, complex programs and systems. The concept of segmentation corresponds to the concept of structured programming, where each procedure and data structure is associated with a distinct segment of memory and each segment is assigned its own attributes that govern its use. A task accesses a particular procedure or data structure by referring to its logical segment number; that segment could be relocated into any appropriate area of physical memory. Access to each segment is restricted through the attribute-checking mechanism of the MMU, thereby protecting system integrity.

Z8010 MMU ARCHITECTURE

The Z8010 Memory Management Unit is a 48-pin LSI device that operates from a single +5-Vpower supply; 4 MHz, 6MHz, and 10 MHz versions are available. The MMU is used in conjunction with the Z8001 CPU to provide dynamic segment relocation and memory protection features within a microcomputer system. A single MMU can manage 64 segments of memory; pairs of MMUs support the full 128 segments available in a Z8001 memory address space. Any number of MMUs may be included in one system. For example, a system might include two MMUs to support 128 segments for normal-mode users and another two MMUs to support 128 segments for operating system software. Physically, the MMU is placed between the CPU and memory within a system; logical addresses are input to the Z8010 and physical addresses are output to memory. Figure 9.4 shows a simple, single-MMU system. Only memory addresses are translated by the MMU; I/O addresses and data bypass this component.

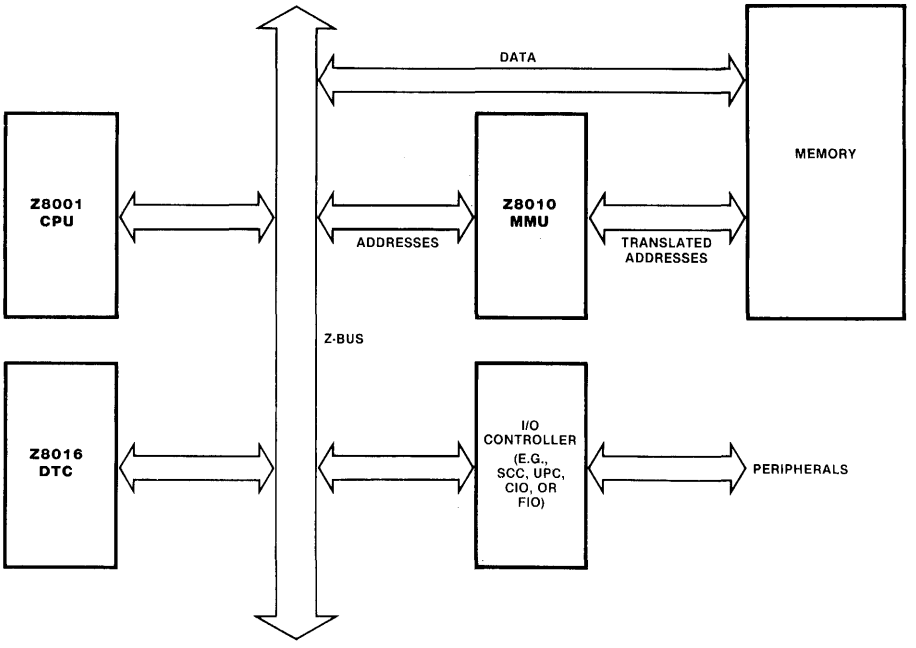


Figure 9.4 Z8010 MMU in a Z8001 system.

The MMU translates the 23-bit logical address (7-bit segment number and 16-bit offset) from the CPU into a 24-bit physical address. Thus the 8M bytes of logical addresses in a Z8001 memory space can be mapped into 16M bytes of addresses in physical memory. Address translation and attribute checking take place on a segment-by-segment basis. A translation table with one entry per segment is used to perform the address translation. The attributes for each segment also are stored in a table, with one entry per segment. As the address translation occurs, the attributes are checked against the status information for the memory access from the CPU. If a violation of the attributes is detected, the MMU notifies the Z8001 via the segmentation trap ($\overline{\text{SEGT}}$) signal.

Figure 9.5 shows the functional pin-out and pin assignments for the Z8010. The MMU receives the upper half of the address/data bus (AD8-AD15), the segment number, the status signals, and the bus timing signals from the CPU. The remaining signals are control lines, including a chip select ($\overline{\text{CS}}$), DMA synchronization strobe (DMASYNC), reset, and the clock. The MMU is not a Z-Bus-compatible part, but instead is considered an extension of the processor. As such, it must receive the same clock as the CPU in order to synchronize the address translation process with the CPU's memory access timing. MMU outputs include 16 bits of physical address informa-

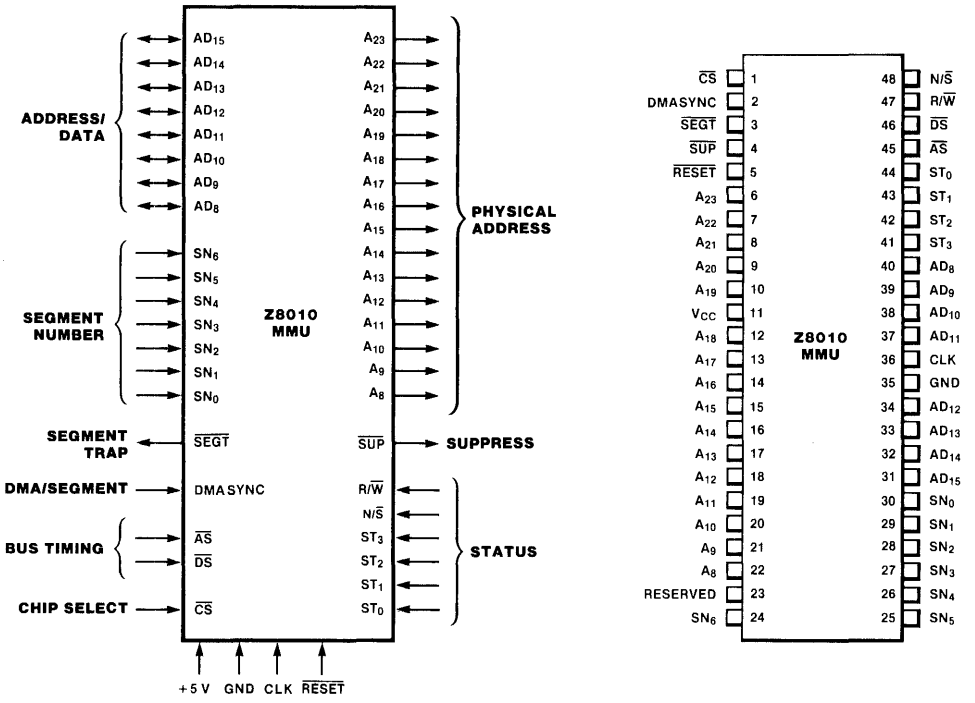


Figure 9.5 Z8010 MMU pinout.

tion (A8-A23), the segmentation trap request line ($\overline{SEGT}$), and a memory suppress signal ($\overline{SUP}$).

For each bus transaction initiated by the CPU or a DMA device, the Z8010 MMU will enter one of three functional states. The first is the memory management state; for a certain subset of memory transactions, the MMU will translate the logical address to a physical address and check the attributes for that access. The second state is a command state; the MMU will interpret special I/O transactions as commands if the MMU's chip select ($\overline{CS}$) input is active. These commands allow the CPU to read from and write to the MMU's registers. The third state is a quiescent state wherein the MMU ignores the transaction and tri-states its address outputs. The MMU ignores all standard I/O, internal, and refresh cycles and a subset of all memory transactions. The MMU also ignores special I/O transactions if $\overline{CS}$ is not active. While in the command or quiescent state, the MMU address outputs are tri-stated.

Figure 9.6 illustrates the interface between the Z8001 CPU and the Z8010 MMU. The MMU selects which of the three states it should enter for a given transaction based on the bus status information on the STO-ST3, R/$\overline{W}$, and N/$\overline{S}$ signals during T1 of a CPU or DMA cycle. If an address translation is to be performed, the MMU uses the segment number to access an internal

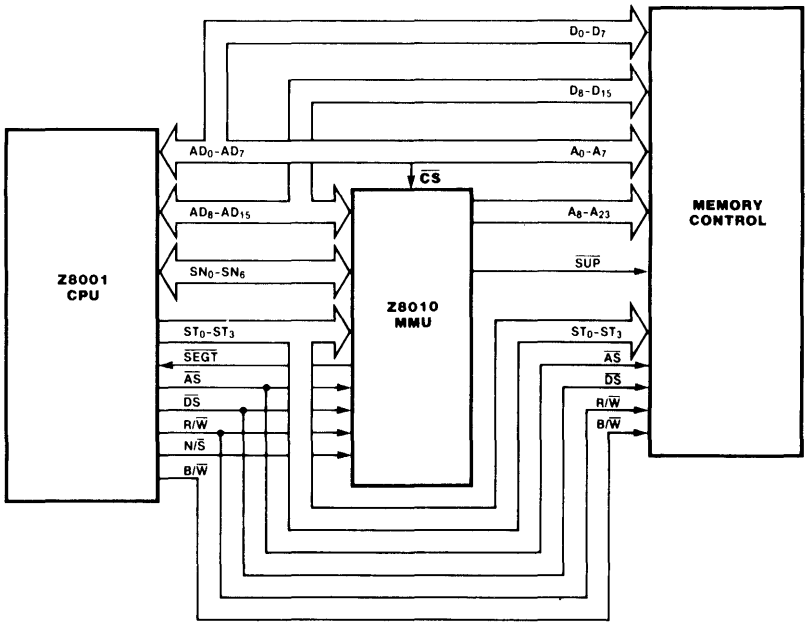


Figure 9.6 Z8001-Z8010 interface.

table of registers. These registers, called segment descriptor registers, contain the starting location (called the base address) of each segment in physical memory, the size of the segment, and the segment's attributes. Segments may be of any size from 256 bytes to 64K bytes, in increments of 256 bytes (that is, 256 bytes, or 512 bytes, or 768 bytes, and so on up to 65,536 bytes). The eight least signficiant bits of the base address are always 0's and are not stored in the segment descriptor registers. Thus segments must begin at some 256-byte boundary in physical memory. Since the low-order 8 bits of the base address are always 0's, the low-order 8 bits of the offset address are not included in the address calculation internal to the MMU. Instead, they are concatenated with the 16 bits output by the MMU to yield a 24-bit physical address (Fig. 9.7).

Therefore, only the upper half of the address/data bus, AD8-AD15, are MMU inputs; only the upper 16 bits of the physical address, A8-A23, are outputs. The low-order 8 bits of the physical address is always the same as the low-order eight bits of the logical address. This scheme saves 16 pins on the MMU package (eight inputs and eight outputs) at the expense of restricting segment sizes to multiples of 256 bytes (since the lowest 8 bits can address a block of 256 bytes). Of course, all 16 bits of the address/data bus must still be propagated to the memory control logic since data always bypass the MMU.

The segment number (SN0-SN7) from the CPU is used to select a seg-

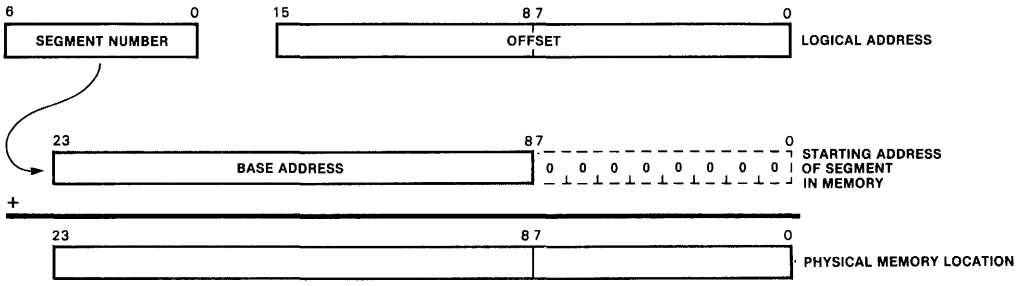


Figure 9.7 Generation of a physical address from a logical address.

ment descriptor register during the address translation process. The segment number is output by the CPU earlier than the offset address during a memory access. This allows the MMU to access its internal segment descriptor register for that segment and select the base address before the offset appears, thereby minimizing the delay between a valid offset address at the MMU's inputs and a valid physical address at the MMU's outputs.

The rising edge of $\overline{AS}$ indicates that the offset address and status signals are valid at the MMU's inputs. $\overline{AS}$ and $\overline{DS}$ provide the bus timing for data transactions between the CPU and MMU during special I/O transactions that access the MMU's registers and during interrupt acknowledge cycles resulting from segmentation traps.

The chip select ($\overline{CS}$) is used to select an MMU during a special I/O instruction. The CPU uses special I/O transactions to read from and write to the MMU's internal registers.

The DMA synchronization strobe (DMASYNC) input signals whether the current cycle was initiated by the CPU or a DMA controller. A low indicates that a DMA device has control of the bus.

When the MMU enters the memory management state during a memory transaction, the segment's attributes are checked against the current bus status. If a violation occurs (for example, the R/$\overline{W}$ line is low, indicating a write transaction, but the segment is assigned the "read-only" attribute), the MMU pulls the segment trap ($\overline{SEGT}$) signal low, forcing a segmentation trap at the CPU. The suppress ($\overline{SUP}$) signal also is pulled low; suppress can be used in the memory control logic to block illegal memory accesses.

The MMU contains a number of internal registers that are used to control MMU operation and to implement the address translation and attribute checking processes. There are three types of registers: segment descriptor registers, control registers, and status registers. A segment descriptor register determines the base address, size, and attributes for a particular segment. There are 64 segment descriptor registers—one for each segment handled by the MMU. One control register, the mode register, determines which memory accesses, if any, will put the MMU in the memory management state. Two

other control registers are used to access the segment descriptor registers, the segment address register (SAR), and the descriptor selection counter register (DSC). Six status registers can be read by the CPU when a segmentation trap occurs to determine the cause of the trap. The violation-type register identifies the type of violation caused by the attempted memory access. The violation segment and violation offset registers hold the segment number and upper byte of the offset address of the memory access that caused the violation. The instruction segment number and instruction offset registers hold the segment number and upper byte of the offset address of the first word in the instruction that was executing when the violation occurred. The bus cycle status register holds the bus status conditions at the time of the violation.

The MMU is controlled via 22 commands issued as special I/O instructions by the Z8001 CPU. With these commands, system software can read from or write to the MMU's registers. Data are transferred between the CPU and MMU one byte at a time on the AD8-AD15 bus lines.

SEGMENT DESCRIPTOR REGISTERS

There are 64 segment descriptor registers in the MMU, one for each segment whose access is controlled by the MMU. Two MMUs are required to handle all 128 possible segment numbers from the Z8001. The mode register is programmed so that an MMU handles segment numbers 0-63 or segment numbers 64-127.

Each of the 64 segment descriptor registers contains information related to the address translation and protection attributes for one segment. A segment descriptor register is 32 bits wide and has three fields: a 16-bit base address field, an 8-bit limit field, and an 8-bit attribute field (Fig. 9.8).

The base address field holds the upper 16 bits of the 24-bit physical address that is the starting address in physical memory for that segment. The low-order byte of the base address is always all 0's (Fig. 9.9). During a memory access, the logical offset address from the CPU is added to this base ad-

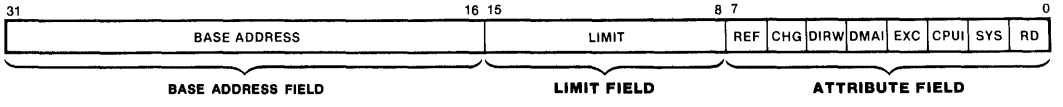


Figure 9.8 Segment descriptor register.

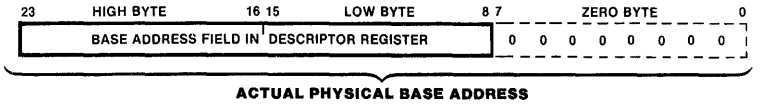


Figure 9.9 Base address for a segment.

dress to yield the corresponding physical address in memory. The two bytes of the base address in the segment descriptor register are read or written one byte at a time when these registers are accessed.

The limit field is an 8-bit field that determines the size of the memory segment. Segments may be any length from 256 bytes to 64K bytes, in increments of 256 bytes. If the limit field contains the number N, then the segment contains either N+1 blocks of 256 bytes or 256-N blocks of 256 bytes, depending on the DIRW bit in the segment descriptor's attribute field, as explained below. Each time an address translation occurs, the upper byte of the logical offset address is compared to this limit field to ensure that the access falls within the limits of that segment; if not, a segment-length violation is detected.

Each of the 8 bits in the attribute field is a flag for a particular attribute or status condition. Five bits determine the protection attributes assigned to the segment, one bit specifies the orientation of the segment, and two bits hold status information recording the types of accesses that have been made in that segment.

Bit 0 of the attribute field is the read-only (RD) bit. When this bit is set, the segment may be accessed only by memory reads (instruction fetches or data reads). Writes are prohibited and an attempted write access will cause a violation condition. This attribute is useful for protecting executable code from inadvertent writes. It is also used to protect critical data that is not to be modified by a particular user.

Bit 1 of the attribute field is the system-only (SYS) bit. When this bit is set, the segment may be accessed only when the CPU is in the system mode; normal-mode accesses are prohibited. This attribute is useful when an MMU receives logical addresses from both the operating system and users' programs. Normal-mode users are prevented from accessing segments containing code or data reserved for the operating system, even if the normal-mode programs generate the correct logical addresses for those segments. For example, I/O routines might be in a segment with the system-only attribute and normal-mode users would be unable to access them directly.

Bit 2, the CPU-inhibit (CPUI) bit, indicates that the segment cannot be referenced by the CPU. When set, all CPU accesses to the segment are prohibited, but DMA controllers can still access the segment. This flag is useful for preventing programs from accessing segments that are being altered by a DMA operation. For example, a DMA controller may be filling a segment with data from a disk drive, interleaving its operation with the CPU. The segment being loaded by the DMA device should not be accessed by the CPU until the DMA operation is complete.

Bit 3 of the attribute field is the execute-only (EXC) bit. When this bit is set, the segment may be referenced only during instruction fetch cycles (including PC-relative loads). Access during any other type of cycle, such as

a data read, is prohibited. This attribute is useful for preventing programs from reading or copying proprietary code.

Bit 4 is the DMA-inhibit (DMAI) bit. When set, the segment cannot be accessed by DMA devices; only CPU accesses are allowed. This attribute can be used to prevent DMA controllers from altering a segment that is being used by a currently executing task.

The DMAI and CPUI bits can be used together to designate a segment that does not exist in physical memory at a given time. If both bits are set, neither the CPU nor DMA controllers can access the segment; an attempted access will cause a violation condition.

Bit 5 of the segment descriptor register's attribute field is the direction and warning flag (DIRW). This bit specifies the orientation of the segment. When this flag is set, the segment's memory locations are organized in descending order, from the segment's base address +65535 down to the limit. With this flag set, the segment contains 256-N blocks of 256 bytes (or 128-N blocks of 128 words), where N is the number in the segment descriptor's limit field (Fig. 9.10). This flag is used for segments that hold stacks, since stacks grow downward in memory.

When this bit is 0, the segment's memory locations are organized in ascending order, starting at the segment's base address. A limit field of N produces a segment with N+1 blocks of 256 bytes. An access beyond this limit is prohibited and will cause a violation condition.

Setting the DIRW bit to 1 activates a special warning feature as well as specifying a downward orientation for the segment. A write to the lowest valid block of 256 bytes in such a segment causes a write warning condition. For a write warning, a segmentation trap request is made but the suppress signal to memory is not activated. Thus the access is successful, but the CPU is interrupted. The write warning signals the CPU that the stack may soon overflow its allotted memory space. In response, the operating system could allocate more physical memory for that stack by increasing the limits of the

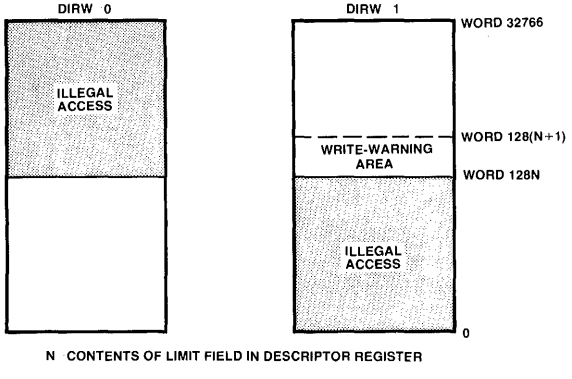


Figure 9.10 Effect of the DIRW bit on segment orientation and size.

segment. This allows stack areas to be allocated dynamically as the need arises while executing tasks. Of course, an access beyond the limits of the segment still causes a violation condition.

Bits 6 and 7 of the attribute field are status bits that indicate the type of accesses made to the segment. Bit 6 is the changed (CHG) bit and is set if the segment has been modified by a CPU or DMA controller access. The CHG bit is set if any write access is made, provided that the access did not cause a violation condition. This flag is useful for indicating when a segment has been modified in the case where the segment must be written to a secondary storage device such as a disk. Segments that have not been changed need not be copied back to the disk if a copy already exists on the disk; this may occur when a task is suspended and removed from memory to make room for another task.

Bit 7 is the referenced (REF) flag and indicates that an access has been made to the segment. This bit is set whenever the segment is read or written by a CPU or DMA device, provided that the access did not cause a violation condition. This flag is useful in determining which segments have not been used in cases where the operating system must select segments to be swapped out of physical memory.

CONTROL REGISTERS

Three 8-bit control registers direct the functioning of the MMU. The mode register controls the enabling of the MMU and the address translation function. The segment address register and descriptor selection counter are pointers into the table of segment descriptor registers.

The mode register contains a 3-bit identification code and five control bits, as illustrated in Fig. 9.11. The identification code (ID code) is used during an acknowledge of a segmentation trap to indicate which MMU or MMUs generated the trap in systems with eight or fewer MMUs. During the acknowledge cycle for a segmentation trap, the MMU uses the ID field in the mode register to select one of the AD8-AD15 lines on the address/data bus; an ID code of 000 corresponds to AD8, an ID code of 001 corresponds to AD9, and so on. Each enabled MMU should have a unique ID code. If an MMU requests a trap, it outputs a 1 on the appropriate line on the address/data bus as determined by the ID code during the acknowledge cycle; otherwise, it outputs a 0 on that line during the acknowledge cycle. The other address/data bus lines are not driven by this MMU, but might correspond to the ID field of another MMU in the system. Thus the upper byte of the identifier word read by the CPU during the segmentation trap acknowledge cycle

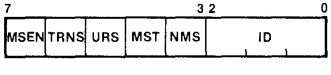


Figure 9.11 MMU mode register.

will indicate which MMUs generated the trap. One memory access could result in multiple violations in different MMUs, so the segment trap service routine may have to deal with several MMUs when processing the trap.

The control bits in the MMU mode register provide a means for selectively enabling MMUs in a multiple MMU system. As mentioned previously, an MMU will enter the memory management state for some memory transactions, but remain in the quiescent state for other memory accesses; the mode register determines the subset of memory accesses that the MMU will handle.

Bit 7 of the mode register is a master enable (MSEN) bit. When set to a 1, the MMU can perform address translation and attribute-checking functions. When cleared to 0, the MMU is disabled and its A8–A23 outputs are tri-stated.

Bit 6, the translate (TRNS) bit, enables the MMU's address translation capability. When this bit is 0 and the MSEN bit is 1, the MMU does not perform address translations or attribute checking, but instead passes the logical address unchanged to physical memory without protection checking. The AD8-AD15 logical address inputs are passed directly to the A8-A15 MMU outputs; the SN0-SN6 segment number inputs are passed to the A16-A22 outputs; A23 is set to 0. This is called the transparent mode for the MMU, since the system now operates as if the MMU were not present. When the TRNS bit is set to 1, the MMU performs address translation and attribute checking for some memory transactions, as determined by the other control bits.

Bit 5 of the mode register is the upper range select (URS) bit. When this bit is cleared, the MMU handles segments 0-63; when set, the MMU handles segments 64-127. Thus for the MMU to enter the memory management state during a given memory transaction, the most significant bit of the segment number, SN6, must match the URS bit. If not, the MMU remains in the quiescent state with its address outputs tri-stated.

Bit 4, the multiple segment table (MST) bit, and bit 5, the normal-mode select (NMS) bit, work together to allow an MMU to be dedicated only to system-mode or normal-mode operation. If separate MMUs are used for the system and normal modes, the MST bit is set in the MMUs. If the MST bit is a 1, the NMS bit determines if the MMU responds to system-mode or normal-mode memory accesses, as indicated by the N/$\overline{S}$ signal from the CPU. When MST is set, the N/$\overline{S}$ MMU input must match the NMS bit in order for the MMU to enter the memory management state; otherwise, the MMU remains in the quiescent state with its address outputs tri-stated. Thus an MMU with MST set and NMS cleared would handle address translation for system-mode memory accesses and another MMU with the MST and NMS set would handle the translation for normal-mode memory accesses. When the MST bit is 0, the MMU responds to all appropriate memory transactions regardless of the operating mode; in this configuration, the system-mode-only attribute can be

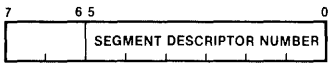


Figure 9.12 Segment address register.

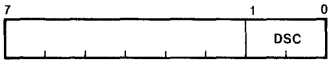


Figure 9.13 Descriptor selection counter register.

used to protect operating system code and data segments from normal-mode users.

During a memory access, an MMU enters the memory management state, wherein it performs address translation and attribute checking if the MSEN and TRNS bits are both 1's, the URS bit is the same as the SN6 input, and the MST bit is 0 or the MST bit is 1 and the NMS bit is the same as the $N/\overline{S}$ input. For all other memory transactions, the MMU is in the quiescent state. In multiple MMU systems, a given memory transaction should cause only one MMU in the system to enter the memory management state, with all other MMUs remaining in the quiescent state.

The segment address register (SAR) is a pointer into the table of 64 segment descriptor registers. Bits 6 and 7 of the SAR are not used (Fig. 9.12). Commands to the MMU that access the segment descriptor registers use the SAR to select one of the 64 descriptors.

The descriptor selection counter register (DSC) points to one of four bytes in a segment descriptor register during accesses to the descriptors. Only bits 0 and 1 of the DSC are used (Fig. 9.13). A value of 00 (binary) in the DSC indicates the high byte of the descriptor's base address field, a 01 in the DSC indicates the low byte of the base address field, a 10 indicates the limit field, and an 11 indicates the attribute field.

Commands to the MMU can read or write only one byte at a time, since the MMU is connected to only one half of the address/data bus. Together, the SAR and DSC point to one byte in the table of 64 segment descriptor registers. For example, if the SAR and DSC are both 0 and the URS bit in the mode register is a 1, the selected byte is the high-order byte in the base address for segment 64. The SAR and DSC are automatically incremented for several MMU commands, allowing the descriptor registers to be accessed as a block using the Z8001's automatically repeating special I/O instructions.

ADDRESS TRANSLATION

When the MMU enters the memory management state, two operations occur simultaneously: the logical address is translated to a physical address and the status information for the memory access is compared to the segment's attributes in the appropriate segment descriptor register.

The address translation process involves using the segment number in the logical address to select one of 64 segment descriptor registers in the MMU.

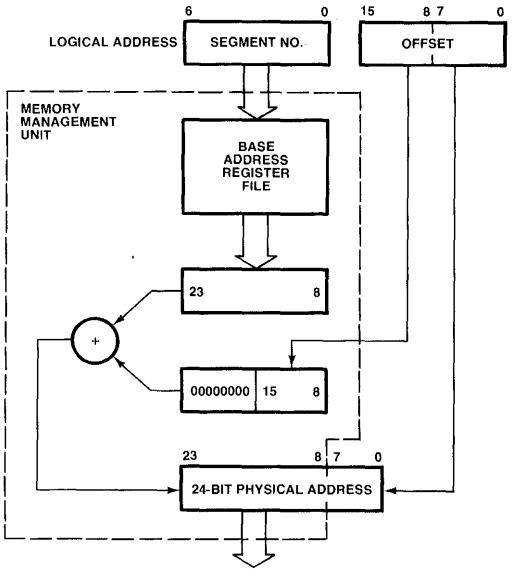


Figure 9.14 Address translation in the MMU.

The contents of the chosen descriptor register's base address field determines the starting address in physical memory for that segment. The MMU's SN0-SN5 segment number inputs are used as a pointer into the bank of segment descriptor registers to select one segment descriptor. (SN6 is used in conjunction with the URS bit in the mode register to select the MMU, as described previously.) The high-order byte of the offset address (emitted by the CPU on AD8-AD15) is then added to the base address in the selected segment descriptor register to yield physical address bits A8-A23. This result is concatenated to the low-order byte of the logical offset address to form the 24-bit physical address (Fig. 9.14). The low-order byte of the offset address is not processed by the MMU.

This address translation process is equivalent to adding the offset portion of the logical address to the starting physical address of the segment (Fig. 9.15). The eight least significant bits of the base address for each segment are

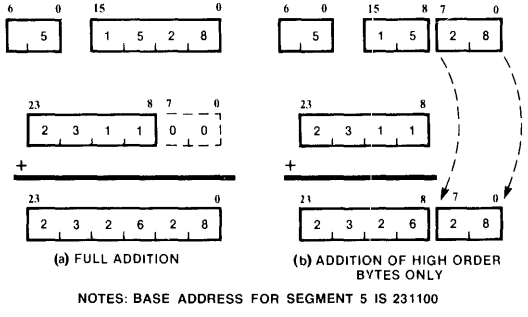


Figure 9.15 Example of address translation process illustrated in two ways.

assumed to be all 0's and are not stored in the segment descriptor registers. Thus the low-order byte of the physical address is always the same as the low-order byte of the offset portion of the corresponding logical address.

VIOLATION TYPES AND STATUS REGISTERS

While the address translation process is occurring, the MMU also compares the attributes in the selected segment descriptor register with the status signals from the CPU. If a violation occurs, such as an attempted write to a segment with the read-only attribute, an active segmentation trap and/or suppress signal is output by the MMU. Furthermore, the offset portion of the logical address is compared to the limit field in the segment's descriptor register and a trap and suppress is generated if the access violates the limits of the segment. The translated physical address is output regardless of whether or not a violation is detected.

The MMU will generate a segmentation trap request to the CPU by pulling its $\overline{\text{SEGT}}$ output low for one of two reasons: detection of an access violation or detection of a write warning in a segment whose DIRW bit is set. In the event of an access violation, the MMU's suppress ($\overline{\text{SUP}}$) output is pulled low also; this signal can be used as part of the memory control logic to inhibit the memory transaction, thereby preserving the integrity of memory contents during illegal accesses.

When the MMU detects an access violation or write warning, it stores status information about the transaction in six 8-bit status registers. These registers can be read by the CPU when servicing segmentation traps to determine the cause of the trap. Five registers indicate the memory address, bus status, and current instruction being executed when the violation occurred. The sixth status register, the violation-type register (VTR), contains eight flags that describe the violation type (Fig. 9.16).

Five flags in the VTR signal violations during memory accesses. Bit 0 of the VTR is the read-only violation (RDV) flag. The RDV flag is set if a write is attempted to a segment having the read-only attribute. Bit 1 is the system violation (SYSV) flag and is set if a normal-mode program attempts to access a segment with the system-only attribute. Bit 2 of the VTR, the segment-length violation (SLV) flag, is set when an attempt is made to access a location outside the legal limits of the segment. This violation is detected by comparing the offset in the logical address to the limit field of the segment descriptor register. Bit 3 is the CPU-inhibit violation (CPUIV) flag and is set if the CPU attempts to access a segment whose CPU-inhibit bit is set. Bit 4 of the VTR is the execute-only violation (EXCV) flag; this flag

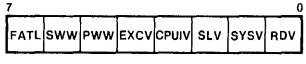


Figure 9.16 Violation-type register.

is set if an access other than an instruction fetch is attempted to a segment with the execute-only attribute.

DMA-inhibit violations also are possible and occur when a DMA controller attempts to access a segment whose DMA-inhibit bit is set. However, there is no corresponding flag in the VTR since violations during DMA accesses do not cause segmentation traps.

Bits 5 and 6 of the VTR deal with write warnings. Setting the DIRW bit in a segment descriptor's attribute field defines that segment as a stack segment that grows downward in memory. An access into the last 256 bytes (that is, the 256 bytes with the lowest physical addresses) in the segment causes a write warning. A write warning does not signal an illegal access but instead warns the system of an impending stack overflow problem. Write warnings can set three different flags in the VTR, depending on the conditions at the time the write warning occurs. If no other flag in the VTR is set, a write warning sets bit 5 in the VTR, the primary write warning (PWW) flag. If any one of the RDV, SYSV, SLV, CPUIV, EXCV, or PWW flags is set because of the execution of a previous instruction and the write warning is the result of an access to the system stack memory address space, then bit 6, the secondary write warning (SWW) flag, is set. If any of those flags in the VTR is set and the write warning is the result of an access to any memory address space except system stack memory, bit 7 of the VTR, the fatal (FATL) flag, is set.

The FATL flag is set whenever a flag already is set in the VTR due to a violation from a previously executed instruction and another violation other than a secondary write warning is detected. This violation must occur during an instruction subsequent to the instruction that caused the first flag in the VTR to be set. If several violations occur during the execution of the same instruction, several flags may be set in the VTR, but the FATL flag is not set. Thus the FATL flag usually indicates a violation that occurs while attempting to service the segmentation trap generated by a previous violation. Once the FATL flag is set, subsequent violations will not cause segmentation traps until the FATL flag has been reset.

When an access violation or write warning occurs, two status registers are used to hold the logical memory address that was being accessed when the violation was detected. The violation segment number register holds the segment number and the violation offset register holds the high-order byte of the logical offset address (Fig. 9.17). External circuitry is required if the lower byte of the offset address is to be saved. If the violation occurred during an instruction fetch, these registers hold the logical address of the word

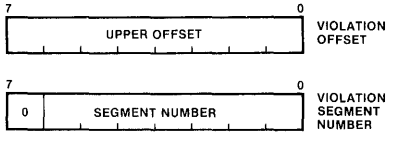


Figure 9.17 Violation segment number and violation offset registers.

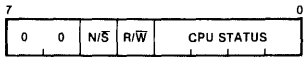
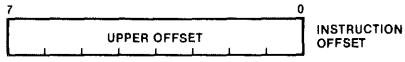


Figure 9.18 Bus cycle status register.



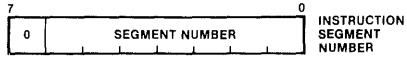


Figure 9.19 Instruction offset and instruction segment number registers.

in the instruction's opcode that was being accessed; otherwise, they hold the logical address of the data item that was being accessed.

The bus cycle status register holds the bus status information at the time of the violation or write warning, including the state of the ST0-ST3 status lines, the R/$\overline{W}$ line, and the N/$\overline{S}$ line (Fig. 9.18).

The last two status registers hold the logical address of the first word of the last instruction fetched before the violation or write warning occurred. The instruction segment number register holds the segment number and the instruction offset register holds the upper byte of the offset address (Fig. 9.19). External circuitry is required if the lower byte of the offset is to be saved. If the violation occurred while fetching the first word of an instruction, these two registers would hold the first word of the previous instruction. Otherwise, these registers will contain the logical address of the first word of the instruction that specified the access that caused the violation.

Status information is stored in the six status registers only for violations or write warnings resulting from an attempted memory access by the CPU. Violations that occur while a DMA device accesses memory will cause $\overline{SUP}$ to be asserted, but no trap is generated and the status registers are not altered. Thus if DMA and CPU operations are interleaved and a DMA transaction causes a violation while the CPU is executing a segmentation trap service routine, the MMU's status registers retain the status information being used by the CPU's service routine.

TRAPS AND SUPPRESSES

The MMU responds to violations and write warnings with two different output signals: segmentation trap ($\overline{SEGT}$) and suppress ($\overline{SUP}$). An active $\overline{SEGT}$ signal causes the CPU to service a segmentation trap. The $\overline{SUP}$ signal is used to block illegal memory accesses; for example, $\overline{SUP}$ can be used to gate the $\overline{DS}$ signal to memory so that accesses attempted while $\overline{SUP}$ is low will not be completed. $\overline{SUP}$ also can be used to trigger external hardware that saves the low-order byte of the offset address for the access that caused the violation. Both $\overline{SEGT}$ and $\overline{SUP}$ are open-drain signals; the $\overline{SEGT}$ and $\overline{SUP}$ outputs from

TABLE 9.1 MMU RESPONSE TO VIOLATIONS AND WRITE WARNINGS

	CPU	DMA
Violation	Trap and suppress	Suppress only
Write Warning	Trap only	No signal

several different MMUs can be tied together to form one signal to the CPU or memory control logic.

An MMU activates the $\overline{\text{SEGT}}$ and/or $\overline{\text{SUP}}$ outputs depending on the device that is making the access and the type of violation that occurred, as outlined in Table 9.1. Suppress is not asserted during write warnings, since write warnings only indicate a potential stack overflow problem in the future, not an illegal access now. DMA-generated memory accesses do not generate segmentation traps, since traps interrupt the CPU and not the DMA controller. DMA write warnings are not signaled at all; DMA devices rarely access memory segments that are being used as stacks.

The $\overline{\text{SEGT}}$ and $\overline{\text{SUP}}$ signals both are asserted during T2 of the memory access cycle, if appropriate (Fig. 9.20). The $\overline{\text{SEGT}}$ signal stays low until a segmentation trap acknowledge signal is detected on the ST0–ST3 status lines. $\overline{\text{SUP}}$ is asserted throughout the data transfer portion of the transaction that caused the violation and for all subsequent CPU memory accesses until the end of the current instruction. Intervening DMA transactions will not be suppressed, however, unless they also generate a violation. Violations during DMA transactions cause $\overline{\text{SUP}}$ to be active only during that transaction.

If the FATL flag in the MMU's VTR is set, indicating that a violation was detected before a previous violation was processed, $\overline{\text{SEGT}}$ will not be asserted for subsequent violations until FATL is reset; $\overline{\text{SUP}}$, however, is generated for each violation even if FATL is set.

If the SWW (secondary write warning) flag in the VTR is set, subsequent write warnings while accessing system stack memory do not generate an active $\overline{\text{SEGT}}$ signal. This prevents the system from repeated interruptions while trying to process the initial write warning.

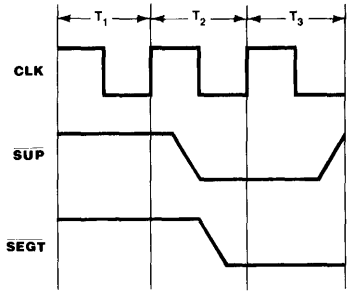


Figure 9.20 Timing of SUP and SEGT signals.

The Z8001 processes a segmentation trap request from an MMU in the same manner as an interrupt. The next instruction fetch is started but aborted, and an interrupt acknowledge cycle is entered. If the MMU detects a violation during the aborted instruction fetch cycle, $\overline{SUP}$ will be asserted but $\overline{SEGT}$ will not. During T3 of the acknowledge cycle, the CPU reads an identifier word from the bus; the upper byte of this identifier word will indicate which MMU or MMUs asserted the trap, as previously discussed. After the acknowledge cycle, the CPU saves the program status for the interrupted task on the system stack. If a write warning is generated while program status is being saved, the SWW flag in the MMU is set and another trap request is made ($\overline{SEGT}$ is asserted again). Servicing this second trap will cause another write warning when program status is saved, but $\overline{SEGT}$ will not be asserted again, since SWW is set already. After saving the old program status, the new program status for the service routine is fetched from the Program Status Area and the service routine is executed. If another violation occurs while fetching new program status or early in the service routine (that is, before the VTR is reset), the FATL flag is set. Subsequent violations cause $\overline{SUP}$ to be asserted but not $\overline{SEGT}$. Thus the FATL and SWW flags prevent a segmentation trap service routine from repeatedly interrupting itself to process a trap it created.

The service routine for a segmentation trap should examine the identifier word on the stack to determine which MMUs detected a violation. Then the VTR in each of those MMUs is checked to determine the cause of the trap. The FATL flag should be tested first to see if multiple violations have occurred. The SWW flag is examined next to determine if more space is needed for the system stack. Finally, the original violation that caused the trap is processed and the VTR is cleared before returning to the interrupted routine. The flags in the VTR are reset by explicit commands from the CPU.

MMU COMMANDS

The CPU can access the MMU's registers via special I/O instructions. When an MMU detects special I/O status on the ST0-ST3 status lines and the MMU's chip-select ($\overline{CS}$) input is active, it accepts and processes a command. These commands allow the CPU to read all MMU registers, write to the segment descriptor, mode, segment address, and descriptor selection counter registers, or reset the violation-type register. Data read from or written to the MMU are transferred one byte at a time on the AD8-AD15 bus lines.

When a special I/O instruction is used to send a command to the MMU, the MMU interprets the high-order byte of the port address output by the CPU during T1 of the I/O machine cycle as a command opcode. The low-order byte of the special I/O port address can be decoded to generate the chip-select signals to the system's MMUs (Fig. 9.21). The MMU's $\overline{CS}$ input is

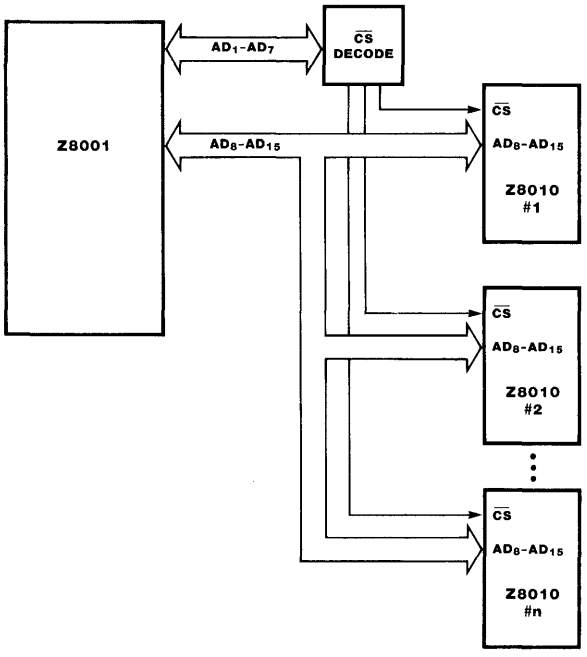


Figure 9.21 Decoding address lines to generate chip selects for MMUs.

used only to send commands to the MMU and is not involved in the MMU's address translation process during memory accesses. Any data associated with the MMU command are transferred between the CPU and MMU during T3 of the I/O cycle, just as if the CPU were talking to a peripheral device. In other words, the upper byte of the special I/O address is the MMU command opcode that enters the MMU on AD8-AD15. The chip-select signal to the appropriate MMU or MMUs is generated from the low-order byte of the I/O address (Fig. 9.22). AD0 must be 0, since any byte data transfer for the command occurs on the upper half of the address/data bus (see Chapter 4). Thus AD1-AD7 are decoded to generate $\overline{CS}$ signals to the MMUs.

For systems with seven or fewer MMUs, the simplest encoding method is to assign ADi (i = 1 to 7) as the chip select for MMU #i (Fig. 9.23). With this scheme, more than one MMU can be selected to receive a given command. This configuration is assumed in subsequent programming examples in this chapter.

Table 9.2 lists all the MMU commands. These commands fall into two basic categories: read/write commands and set/reset commands.

The read/write MMU commands are used to read or write the MMU's internal registers. Special input instructions are used to read MMU register

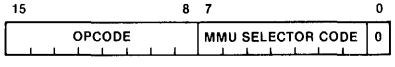


Figure 9.22 Format of a special I/O address used as an MMU command.

TABLE 9.2 MMU COMMANDS

Opcode	Operation
Read/Write commands	
Segment descriptor registers	
08	Read/Write Base Field in Descriptor
09	Read/Write Limit Field in Descriptor
0A	Read/Write Attribute Field in Descriptor
0B	Read/Write Descriptor (all fields)
0C	Read/Write Base Field and Increment SAR
0D	Read/Write Limit Field and Increment SAR
0E	Read/Write Attribute Field and Increment SAR
0F	Read/Write Descriptor and Increment SAR
Control registers	
00	Read/Write Mode Register
01	Read/Write Segment Address Register
20	Read/Write Descriptor Selector Counter Register
Status registers	
02	Read Violation Type Register
03	Read Violation Segment Number Register
04	Read Violation Offset (high-byte) Register
05	Read Bus Cycle Status Register
06	Read Instruction Segment Number Register
07	Read Instruction Offset (high-byte) Register
Set/Reset commands	
Segment descriptor registers	
15	Set All CPU-Inhibit Flags
16	Set All DMA-Inhibit Flags
Violation status registers	
11	Reset Violation Type Register
13	Reset SWW Flag in VTR
14	Reset FATL Flag in VTR
Reserved	
10	Not assigned
12	Not assigned
17-1F	Not assigned
21-FF	Not assigned

contents to the CPU; special output instructions are used to write into MMU registers from the CPU. The mode register, segment address register, descriptor selection counter register, and all the segment descriptor registers can be read or written with these commands. The status registers can be read only. For example, the instruction

```
SOUTB  %00FC, RH0
```

will load the contents of CPU register RH0 into the mode register of the MMU that is chip selected when AD1 is low (MMU #1 in Fig. 9.22). Several of these instructions automatically increment the SAR and DSC pointers

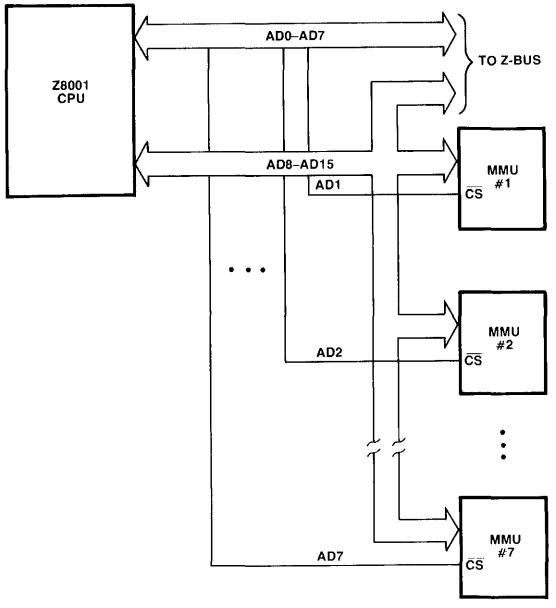


Figure 9.23 Simple chip select encoding for seven MMUs.

into the bank of segment descriptor registers, allowing use of repeating block move special I/O instructions to fill the 64 descriptors. For example, the Read/Write Descriptor and Increment SAR command (opcode %0F) accesses the four bytes of a descriptor register and then increments the SAR to point to the next descriptor. If the data to be loaded into all 64 descriptor registers in an MMU are stored in memory as shown in Fig. 9.24, the descriptors could be initialized in the MMU with the following code:

```
CLR     R0
SOUTB   %01FC, RH0          ! clear the SAR in MMU #1 !
LDA     RR4, DESCRIPTORS    ! segmented mode; DESCRIPTORS is symbolic name
                              for starting address of table in memory !
LD      R0, #256            ! count register !
LD      R1, #%0FFC          ! I/O port address is MMU command to load de-
                              scriptors of MMU #1 !
SOTIRB  @R1, @RR4, R0       ! Load all descriptors in MMU #1 !
```

Note that the six status registers are read-only, with the exception of the VTR, which can be reset.

The set/reset commands are used to set or reset certain fields in MMU registers. For these commands, special output instructions are used, and no data are transferred:

```
SOUTB   %11FC, RH0      ! reset the VTR in MMU #1 !
```

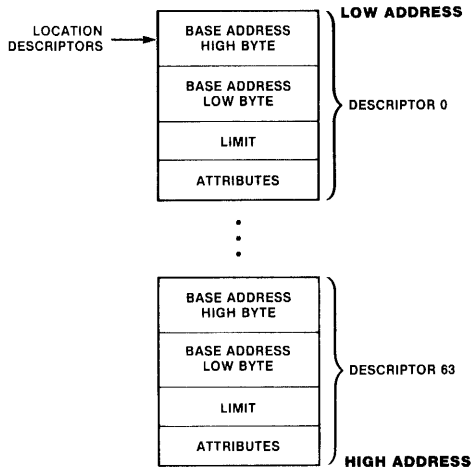


Figure 9.24 Format of data in memory to be loaded into an MMU's segment descriptor registers.

The destination address is the MMU command word, and the source can be any arbitrary CPU register. The contents of the source register are placed onto the bus during T3 of the I/O cycle but are ignored by the MMU.

RESETS

The Z8010 MMU is reset by pulling its $\overline{\text{RESET}}$ input low. A reset clears the mode, descriptor selection counter, and violation-type registers. The contents of all other registers are undefined after a reset. If $\overline{\text{CS}}$ is high while $\overline{\text{RESET}}$ is asserted, the master enable flag (MSEN) in the mode register is cleared and the MMU is disabled. The address outputs are tri-stated and the $\overline{\text{SUP}}$ and $\overline{\text{SEGT}}$ open-drain outputs are not driven. To enable the MMU, the CPU must write to the mode register and set the MSEN bit.

If $\overline{\text{CS}}$ is low while $\overline{\text{RESET}}$ is asserted, the MSEN bit in the mode register is set and the translate bit (TRNS) is cleared, thereby enabling the MMU but putting it in the transparent mode. The logical address inputs are passed directly through to the physical address outputs without translation. One MMU in a system should be reset in this manner so the CPU can access memory and execute an initialization routine. The initialization routine would be in absolute memory locations in memory (that is, the initialization routine is not relocatable). The initialization routine should include commands to program the system's MMUs.

MULTIPLE MMU SYSTEMS

The architecture of the Z8010 MMU supports system configurations that include more than one MMU. Multiple MMUs can be used to support all 128 possible segment numbers rather than the 64 segments managed by one MMU, or to support multiple translation tables in a multitasking system.

A single-MMU system is restricted to handling only 64 logical segment numbers—either segments 0-63 or segments 64-127. If the CPU generates a logical address with a segment number outside the range handled by the MMU, the MMU remains in the quiescent state and no physical address is output to memory. Single-MMU systems require external hardware to detect erroneous segment numbers outside the range handled by the MMU and generate $\overline{SEGT}$ and $\overline{SUP}$ signals if an illegal segment number is encountered.

A two-MMU system would be capable of handling all 128 segment numbers from the Z8001 CPU. The URS flag in the mode register is used to determine which MMU handles which segment numbers. Figure 9.25 is a

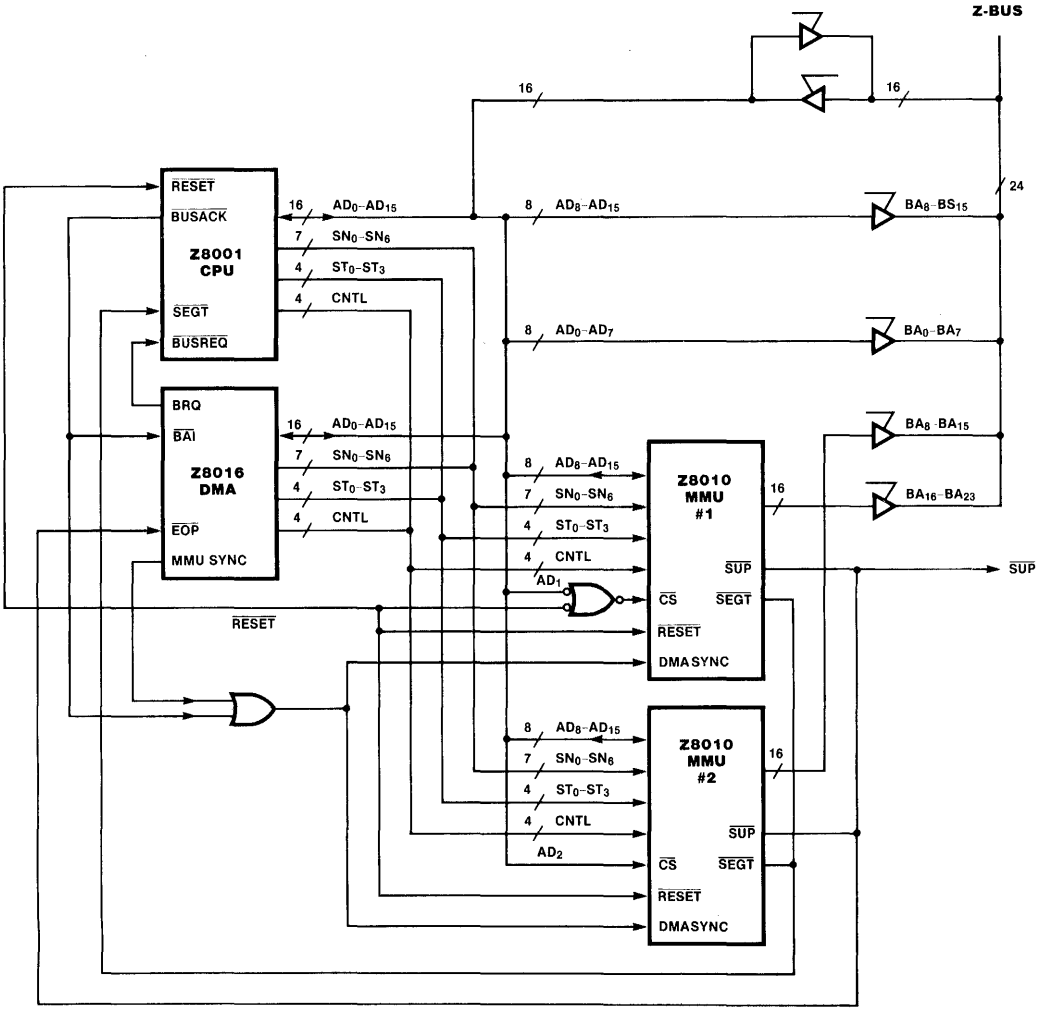


Figure 9.25 Dual-MMU system.

block diagram of a dual-MMU system with a Z8016 DMA controller. When a reset occurs, MMU #1 enters the transparent mode and MMU #2 is disabled.

Additional MMUs can be added to the system to implement multiple translation tables. The MST and NMS flags in the MMU's mode registers allow separate address translation tables for system-mode and normal-mode operations. Furthermore, separate translation tables for different users' tasks could be built in separate MMUs, and the appropriate MMUs enabled or disabled with the MSEN flag as part of the task switching process in the operating system. Alternatively, one MMU could handle all users' tasks and be completely reprogrammed during the task switching process. Thus a trade-off must be made between the number of MMUs in the system and the frequency of reprogramming each MMU.

THE MMU AND MEMORY ACCESS TIME

During a memory access, the Z8001 CPU outputs the segment-number portion of the logical address early in the machine cycle. (In fact, the segment number is emitted during the last timing state of the preceding cycle, as discussed in Chapter 5.) This allows the MMU to use the segment number to select the appropriate segment descriptor register before the offset portion of the address is available. Later, in T1 of the memory access cycle, the CPU outputs the offset address. The MMU adds the upper byte of this offset to the base address in the segment descriptor register to yield the physical address while checking the status lines with the segment's assigned attributes. This addition process does take some time, of course, so there is some delay between when the CPU issues the logical offset address and the MMU outputs a valid physical address. This delay shortens the memory access time for the transaction. Furthermore, the rising edge of $\overline{AS}$ from the CPU no longer indicates a valid physical memory address; a "delayed" address strobe may be needed to indicate a valid address at the memory control logic and the hardware to generate this delayed address strobe added to the system.

MMUs AND VIRTUAL MEMORIES

When a system has a physical memory address space that is smaller than its logical address space it is called a virtual memory system. In a virtual memory system, operating system software, memory management hardware, and a secondary storage device, such as a disk, are used to make physically addressable memory appear larger than it really is for users' programs. All the segments for a particular users' program might not fit into memory at any one time. Segments that are "missing" in memory are so marked in the memory manager; for the Z8010 MMU, the CPU-inhibit flag could be used.

A reference to such a segment would cause a trap; the trap service routine would move the "missing" segment from the disk to memory, swapping out some segment already in memory by writing it to the disk. The MMU is reprogrammed to reflect this change, and the users' program is restarted at the instruction that caused the trap. All of this memory manipulation would be transparent to the user executing an applications program; thus the system would appear to each user as having a physical memory address space as large as the logical address space.

In a virtual memory system, the operating system must keep track of which segments have been used, how often, and in what way. A segment that has not been referenced at all since last being loaded into memory is a likely candidate for removal from physical memory when another segment must be swapped into memory from the disk. The most frequently accessed segments should be kept in memory at all times if possible. The REF and CHG flags in the MMU's segment descriptor registers can aid in this process.

However, the Z8001-Z8010 combination does not completely support virtual memory implementations. If the Z8001 references a segment that does not exist in physical memory, the MMU will respond with active $\overline{\text{SEGT}}$ and $\overline{\text{SUP}}$ signals. In the case of a memory write, the $\overline{\text{SUP}}$ signal can be used to block the memory access, thereby protecting the integrity of memory contents. However, the Z8001 will complete the execution of the current instruction before recognizing the segmentation trap request. The critical case occurs when the CPU is executing an arithmetic or logical instruction where the source is in memory but the destination is a CPU register, such as

```
ADD  R0, DATA
```

where DATA is the symbolic name for some memory location. If the fetch to the location called DATA causes an MMU violation and trap request, the instruction will be completed before the trap is recognized and processed. Even though a suppress is sent to memory, the CPU will read something on the bus during T3 of the data memory access and add whatever is read to the contents of register R0. Thus the contents of CPU registers can be corrupted.

However, true virtual memory systems can be implemented with the Z8001 and Z8010 by adding additional hardware to the system. This extra hardware would need to force a predetermined value on the bus whenever an MMU violation occurs, as indicated by the $\overline{\text{SUP}}$ signal. The value placed on the bus would depend on the type of instruction being executed. For example, if all 0's were forced on the bus during the data fetch for the ADD instruction described above, the contents of R0 would not be corrupted, and the service routine could force a restart at that instruction after making the appropriate changes to physical memory and the disk. In the case of a violation during an IF1 cycle, the extra logic could force the opcode for a NOP instruction onto the bus.

(Another upward-compatible member of the Z8000 family of processors, the Z8003, has an instruction abort feature that permits the implementation of virtual memory systems without this additional hardware. In Z8003-based systems, the memory manager can force the CPU to abort the execution of an instruction, thereby protecting the integrity of CPU registers. This abort sequence leaves the Z8003 CPU in a well-defined state, allowing a software recovery. Thus the Z8003 is called a Virtual Memory Central Processing Unit.)

Within a system, memory segments are continuous blocks of physical memory with sizes varying from 256 bytes to 64K bytes. If a virtual memory system services many tasks, segments of widely ranging sizes may be swapped into and out of physical memory. Whatever procedure is used to control this swapping process, "holes" inevitably develop in physical memory between segments (that is, areas of memory not assigned to any segment and not big enough to form another segment). Occasionally, the memory management software may need to coalesce several of these "holes" into a useful block of memory by reassigning existing segments to new physical addresses. In general, this is a difficult task that can consume considerable execution time.

To aid in this process, most virtual memory systems divide physical memory into sections called pages. Pages are fixed-size blocks of memory (as opposed to segments, which are of variable size) and typically range from tens to hundreds of words, depending on system requirements. Paging does require a large investment in hardware and software, however, since relocation, property, and usage data must be maintained for each page.

The Z8001 and Z8010 do not support paging directly in their architecture. However, paging can be designed into a Z8001 operating system, thereby realizing the advantages of both memory paging and memory segmentation. If paging is designed into a Z8001-Z8010 system, page sizes of some multiple of 256 bytes are easiest to implement, since each segment could then contain an integral number of pages.

[Another memory management unit, the Z8015 Paged MMU (PMMU), has been designed for use with the Z8003 CPU in virtual memory systems. Each Z8015 PMMU can manage 64 2048-byte memory pages. The Z8015 generates the instruction abort signal to the Z8003 CPU if an access is made to a page that is not present in main memory.]

10

Extended Processor Units

Additions can be made to the basic instruction set of the Z8000 microprocessors through the use of Extended Processor Units (EPUs). An Extended Processor Unit is an LSI device dedicated to performing complex, time-consuming tasks in order to unburden the CPU. Typical tasks suited for implementation in EPUs include floating-point arithmetic, data base management, graphics support, networking, and communications interfaces—in short, any computing task that might be performed in dedicated hardware. Up to four EPUs can be included in a Z8000-based system.

EPUs perform their tasks on data resident in their internal registers. The CPU is responsible for moving data into and out of the EPUs and for instructing the EPUs as to what operations are to be performed. Special instructions called extended instructions are processed by the EPUs; when the CPU encounters an extended instruction it performs any specified data transactions, but otherwise assumes that the instruction will be recognized and handled by an EPU. Thus by adding EPUs to a system, the instruction set is expanded to include the extended instructions applicable to the EPUs used, thereby boosting the processing power of the whole system.

CPU-EPU INTERFACE

Extended Processor Units connect directly to the Z-bus; no extra external logic is needed to interface an EPU to a Z8000-based system. As the CPU fetches and executes instructions, the EPUs monitor the bus. When an ex-

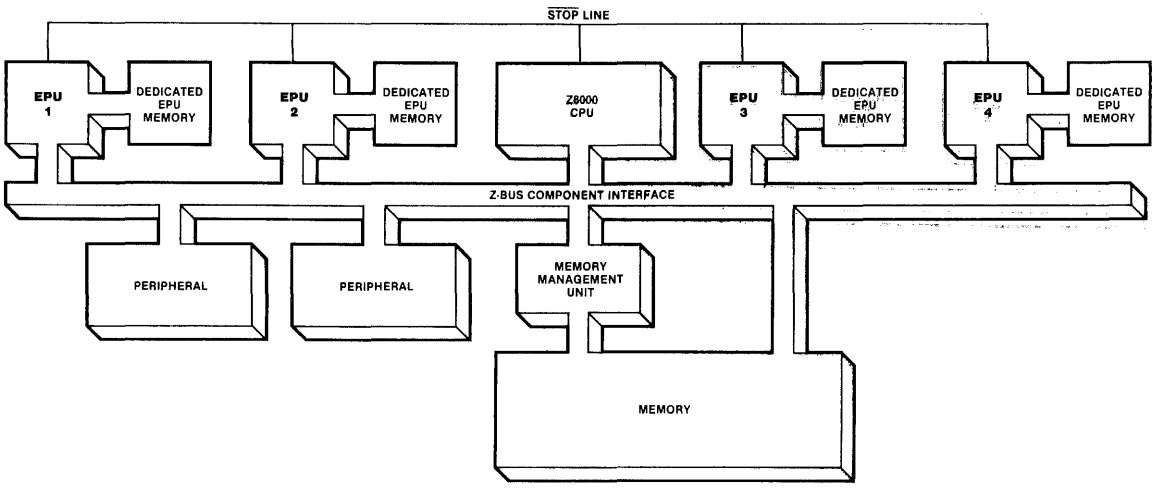


Figure 10.1 Typical Z8000 configuration with 10 EPUs.

tended instruction is encountered, the appropriate EPU responds by executing the instruction; this may involve having the EPU send or receive data or status information on the address/data bus during a bus transaction. Protection against overlapping instructions is provided by the $\overline{\text{STOP}}$ signal. $\overline{\text{STOP}}$ is an EPU output that is pulled low if the EPU is requested to perform an operation before completing a previous operation. An active $\overline{\text{STOP}}$ input to the CPU puts the processor in a state wherein it executes refresh cycles continuously until $\overline{\text{STOP}}$ returns high, effectively halting the processor. Figure 10.1 is a block diagram of a Z8000 system with four EPUs.

The CPU and EPUs work together like one processing unit. The CPU supplies all the address and status information for fetching instructions and reading or writing data to memory. The EPUs monitor these transactions, accepting or supplying data as required. Each EPU must continuously monitor the address/data bus and its associated control and status lines from the CPU to know when to participate in EPU-to-memory or EPU-to-CPU data transactions. A system with EPUs can be thought of as a system whose processor consists of 1 + N devices, where N is the number of EPUs in the system. Thus EPUs provide a means of adding power to the system's processor in a modular fashion.

EXTENDED INSTRUCTIONS

Instructions with an opcode whose first word has an upper byte of 0E, 0F, 4E, 4F, 8E, or 8F (hexadecimal) are extended instructions reserved for use by an EPU. All extended instructions have opcodes that are two words long.

If the EPA bit in the CPU's flag and control word (FCW) is a 1 and an extended instruction opcode is encountered, the CPU will assume that there are EPUs in the system and treat the instruction accordingly, as described below. If the EPA bit in the FCW is 0, indicating that there are no EPUs in the system, encountering an extended instruction opcode will cause an extended instruction trap. If desired, the action of an EPU can be simulated in software in the extended instruction trap service routine. This software trap mechanism facilitates the design of systems in which EPUs are not present initially but may be added later. The "extended" function is included in the operating system software as the extended instruction trap service routine; this routine can be deleted when the EPU is added to the system and the EPA bit set to a 1. This change would be transparent to users' applications executing on the system.

If EPUs are present in the system and the EPA bit in the CPU is a 1, the CPU is responsible for delivering instructions and data to the EPUs. There are four kinds of extended instructions: instructions that transfer data between an EPU and memory, instructions that transfer data between an EPU and the CPU, instructions that transfer status information between an EPU and the CPU's flag and control word, and instructions that specify internal operations in an EPU.

In order to determine which transactions to participate in, an EPU must monitor the address/data bus and its associated status and control signals. When the CPU fetches the first word of an instruction, as indicated by IF1 status on the ST0–ST3 lines, each EPU examines the instruction's opcode. If an extended instruction is found, each EPU checks a 2-bit identification field in the opcode to see if the instruction is intended for that particular EPU. Thus up to four EPUs can be interfaced to a single CPU. The EPU selected must also capture the second word of the extended instruction's opcode; the fetch of the second word of the opcode is the next nonrefresh CPU transaction. From this two-word opcode, the EPU determines if it will participate in any subsequent data transactions and, if so, how many transactions are involved.

If the extended instruction calls for a data transfer between an EPU and memory, the CPU supplies the address, status, and control information for each transfer. During the memory accesses, the CPU will tri-state its AD0–AD15 lines while $\overline{DS}$ is low, so that the EPU can send or receive data on the bus. The CPU can use the indirect register, direct address, or indexed addressing modes to calculate the memory addresses for the transactions, as specified by the instruction. The EPU must supply the data (if the R/$\overline{W}$ line is low, indicating a memory write) or capture the data (if the R/$\overline{W}$ line is high, indicating a memory read) just as if it were part of the CPU. EPU-to-memory data transactions are always word transfers (B/$\overline{W}$ is low). Up to 16 words can be transferred between an EPU and memory as the result of a single extended instruction, as specified in the instruction's opcode.

If the instruction involves a data transfer between the EPU and the CPU's general-purpose registers, the CPU controls the transaction and a 1110 code (CPU-EPU transfer status) is emitted on the ST0-ST3 lines. The timing of the transaction is identical to a three-clock-cycle memory access, except that no address is emitted by the CPU. CPU-EPU transactions are always word transfers; up to 16 words can be transferred as a result of an extended instruction.

Similarly, status information can be transferred between an EPU and the lower byte of the CPU's FCW (the CPU flags). CPU-EPU status transfers are always byte transfers and occur on the AD0-AD7 bus lines. The contents of CPU register R0 are destroyed during this transaction. This type of transaction is useful when the program must branch on the results of an EPU operation.

Extended instructions can also specify internal operations for EPUs, wherein the EPU operates on data in its internal registers. No data transactions are involved in the execution of such an instruction. The CPU can continue to fetch and execute subsequent instructions while the EPU is involved with an internal operation. Thus processing can proceed simultaneously in both the CPU and the EPUs. If a second extended instruction for a particular EPU is fetched before an earlier instruction for that EPU has completed execution, the EPU must activate the $\overline{\text{STOP}}$ line, stopping the CPU. The EPU releases the $\overline{\text{STOP}}$ signal when it completes execution of the first instruction, and execution proceeds as if the CPU were not temporarily halted. The $\overline{\text{STOP}}$ line provides synchronization between the CPU and the EPUs, preventing EPUs from missing instructions because they were busy executing earlier instructions. Of course, all the EPUs in a system may be executing their own internal operations simultaneously.

In order to monitor instruction fetches and participate in data transfers, the EPUs must also monitor the $\overline{\text{BUSACK}}$ CPU output to verify that transactions are initiated by the CPU. EPUs will ignore all bus transactions while $\overline{\text{BUSACK}}$ is low.

Thus EPU instructions are processed "in-line" with Z8000 instructions, providing parallel processing capability while eliminating the system software and bus contention problems inherent in other multiprocessor and co-processor schemes, such as master-slave arrangements. The processing power of a Z8000-based system can be extended in an upward-compatible manner by the addition of EPUs.

STOP TIMING

The $\overline{\text{STOP}}$ input to the CPU is used to synchronize EPU and CPU execution, as described above. A low level on the $\overline{\text{STOP}}$ input forces the CPU into a state wherein it continuously executes refresh machine cycles. The $\overline{\text{STOP}}$ in-

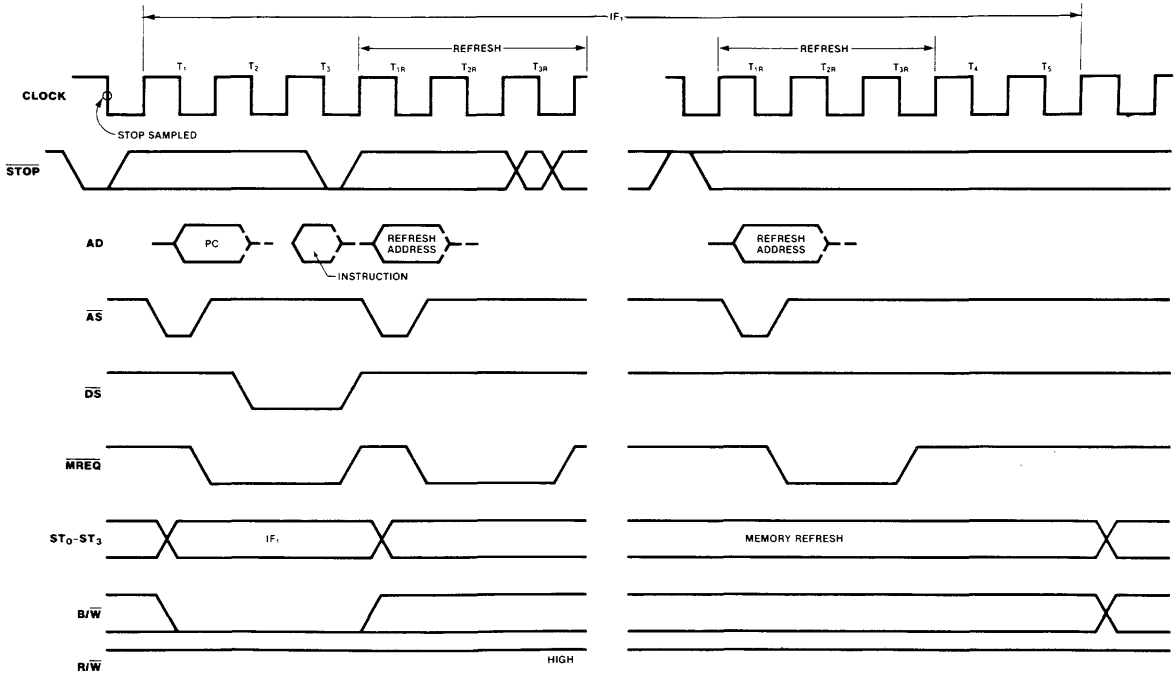


Figure 10.2 STOP timing.

put is sampled by the CPU on the falling edge of the clock in the clock period preceding an IF1 cycle, as shown in Fig. 10.2. If $\overline{\text{STOP}}$ is low, a continuous stream of refresh cycles is entered after T3 of the instruction fetch. During each of the refresh cycles, $\overline{\text{STOP}}$ is sampled again on the falling edge of the clock in the middle of T3. When $\overline{\text{STOP}}$ is found to be high, one more refresh cycle is executed and then any remaining T states in the IF1 cycle are completed. From there, execution proceeds normally. Bus requests are honored while $\overline{\text{STOP}}$ is active.

If the EPA bit in the CPU's FCW is set, the $\overline{\text{STOP}}$ input also is sampled on the falling edge of the clock preceding the fetch of the second word of an extended instruction. (The CPU recognizes that it is processing an extended instruction after the first word of the instruction is fetched.) Thus the $\overline{\text{STOP}}$ line can be activated by an EPU if the CPU fetches an extended instruction for that EPU before the EPU has finished processing an earlier extended instruction.

The continuous refresh operation while $\overline{\text{STOP}}$ is low does not use the rate counter in the CPU's refresh register. The row counter is incremented by two after each refresh cycle. Thus refreshes do not occur on demand; instead, a new refresh is emitted every three clock cycles. Therefore, higher-than-normal heat dissipation may occur in dynamic memories while $\overline{\text{STOP}}$ is low. Long and frequent stops can be avoided by writing program code so that extended instructions for a given EPU are not closely spaced.

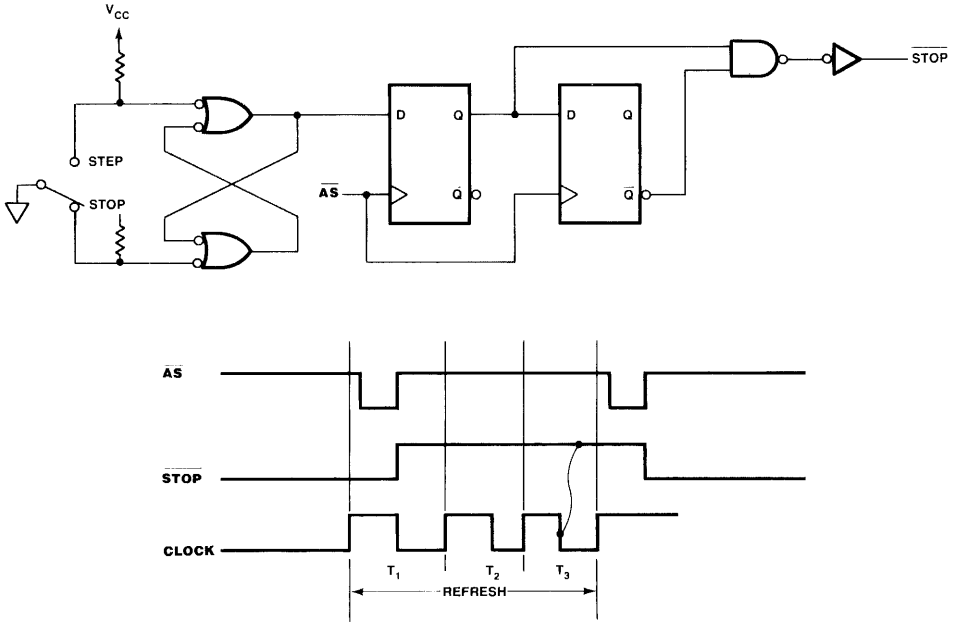


Figure 10.3 Z8000 single-step circuit using the $\overline{\text{STOP}}$ input.

The $\overline{\text{STOP}}$ input also can be used to externally single-step a Z8000 CPU. A circuit for single-stepping the CPU one instruction at a time is illustrated in Fig. 10.3. The cross-coupled NOR gates act as a switch debounce. Each time the switch is pulled from the 0 position (STOP) to the 1 position (STEP), $\overline{\text{STOP}}$ is forced high for one machine cycle. This allows the CPU to exit the "stopped" state and complete execution of the current instruction. $\overline{\text{STOP}}$ will return low before the next instruction is fetched, so that instruction will not execute until the switch is toggled again. Thus single-step execution is realized. However, if there are dynamic memories in the system that use the Z8000's automatic memory refresh capabilities, heat dissipation problems may occur due to the large amount of time spent in the "stopped" state.

11

A Z8000 Design Example

A small but powerful single-board microcomputer system can be developed using the Z8002 microprocessor, Z80-family peripherals, PROMs, dynamic RAMs, and some TTL support devices. Figure 11.1 is a block diagram of a system that includes up to 32K bytes of PROM, 32K bytes of RAM, two Z8420 Parallel I/O Controllers (PIOs), a Z8430 Counter/Timer Circuit (CTC), and a Z8440 Serial I/O Controller (SIO). (The PIO, CTC, and SIO are all Z80-family peripherals. The Z80 is an 8-bit microprocessor.) The two PIOs provide four 8-bit bidirectional parallel ports with handshake control. Two independent full-duplex serial I/O channels are implemented with the SIO. The CTC includes four 8-bit counter/timers for simple counting tasks and generation of the baud-rate clocks for the serial channels. A set of eight switches that can be read by the CPU is included as a fifth peripheral device. A crystal-controlled timing circuit provides the timing signals for the system's components.

(The system of Fig. 11.1 is similar in structure to the Zilog Z8000 Development Module. The Z8000 Development Module is a single-board microcomputer intended to support evaluation of the Z8001 and Z8002 microprocessors. A monitor program, stored in EPROM, is available for this product to aid in the debugging and evaluation of users' applications programs.)

CLOCK GENERATION

The Z8001 and Z8002 require a continuously running clock with a frequency between 500 kHz and 4 MHz; the Z8001A and Z8002A require a

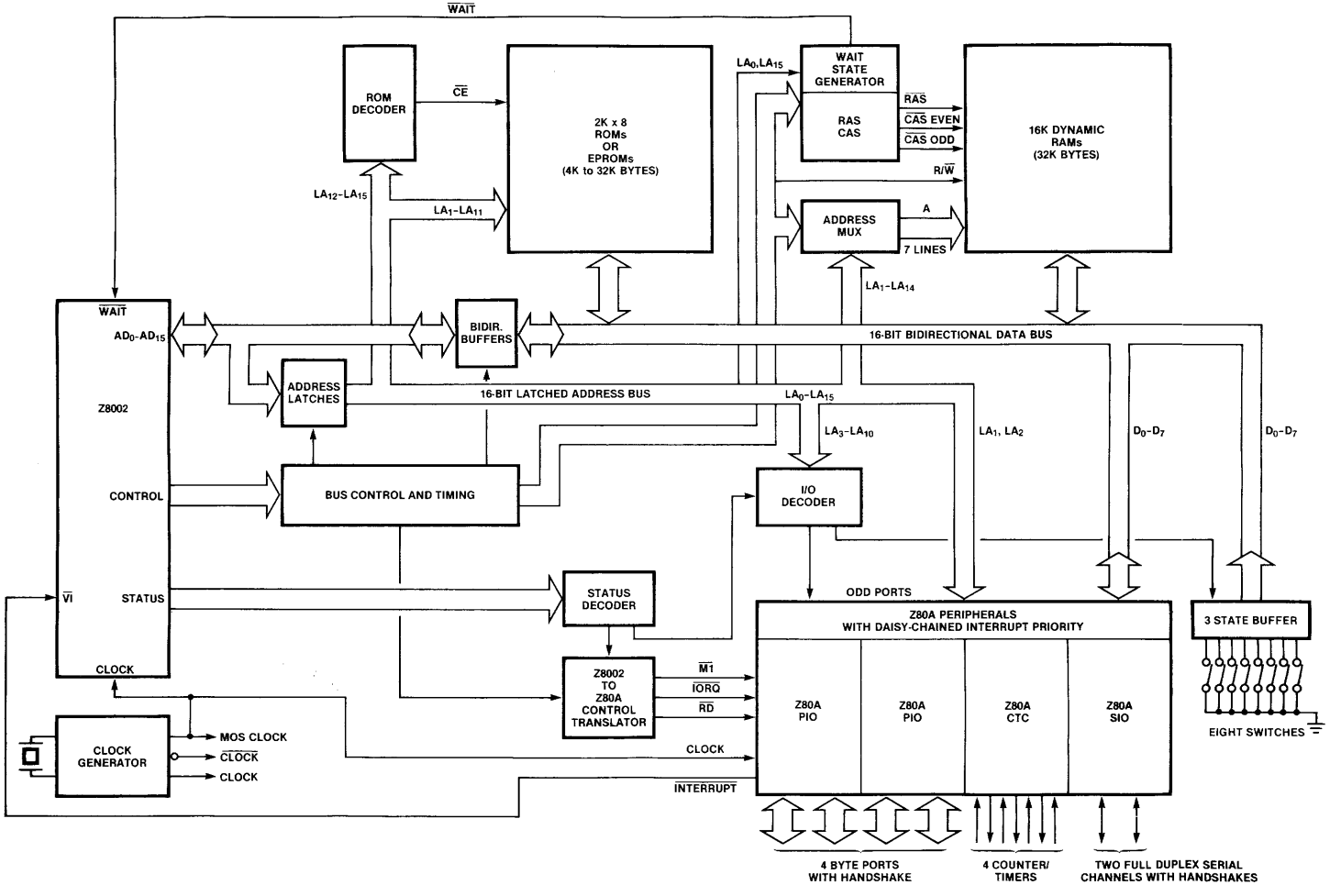


Figure 11.1 Block diagram of a Z8002-based single-board microcomputer.

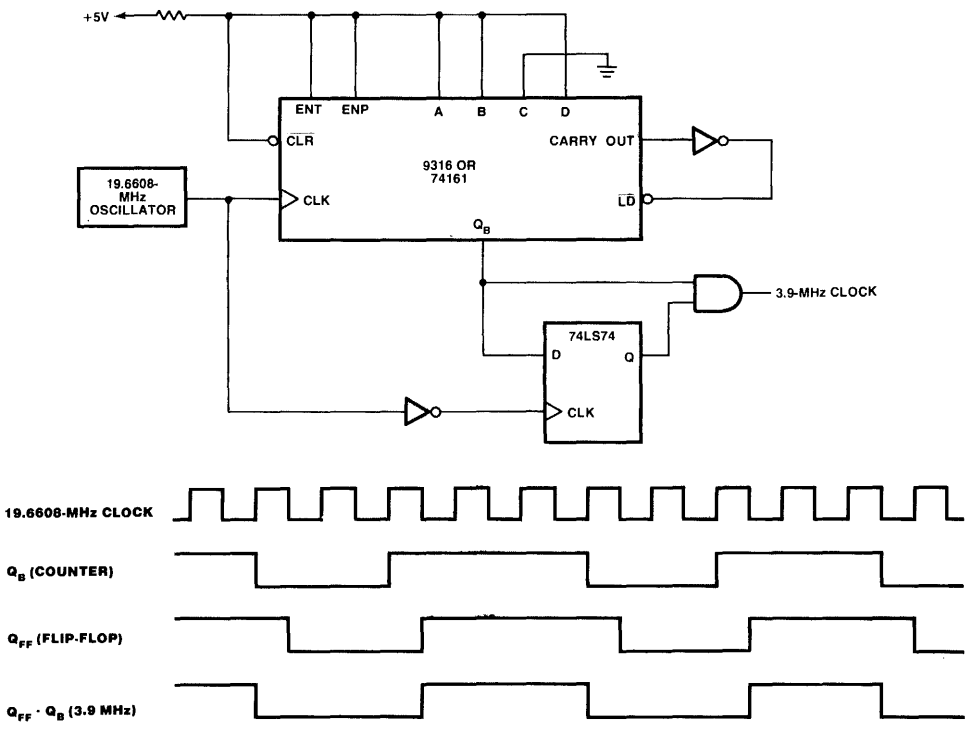


Figure 11.2 Generation of a 3.9-MHz clock from a 19.6608-MHz oscillator.

clock frequency between 500 kHz and 6 MHz. Most Z8000 applications call for a high-performance system with a clock frequency approaching the maximum limit. Clocks usually are generated by dividing the output of a crystal oscillator using flip-flops (as in Fig. 1.14) and/or TTL counter chips. For example, a divide-by-5 circuit can be used to generate a 3.93-MHz clock from a 19.6608-MHz crystal, as illustrated in Fig. 11.2. (19.6608 MHz is a frequency commonly used to generate baud clocks for serial channels.) The Z8000 CPU's clock input is not TTL compatible because of its level, rise-time, and fall-time requirements. An active driver circuit such as the one illustrated in Fig. 1.14 is required for the CPU clock.

CPU BUS BUFFERING

As with most MOS devices, the Z8000 CPU outputs have limited drive capability. The Z8000 microprocessors sink a maximum of 2.0 mA for output signals and source a maximum of 250 μA for input signals while maintaining standard TTL levels (0.4 V maximum for a logical 0, 2.4 V minimum for

a logical 1). Thus a Z8000 CPU output can drive only one standard TTL load or five LS-TTL loads. Output delays are specified for a 50-pF load and increase by approximately 0.1 ns/pf for additional capacitive loading. Since the address/data bus and its control and status signals usually are propagated to several memory and I/O control devices, all but the simplest of systems require buffering of the CPU outputs.

The 16 address/data bus signals are bidirectional. Address and data information are CPU outputs during write operations; addresses are CPU outputs and data are CPU inputs during read operations. Thus buffers for the address/data bus must be bidirectional. Two approaches are possible: have the buffers for the bus point away from the CPU as a default (that is, treat the CPU pins as outputs and drive the system bus with the buffer), with the buffer direction reversed (that is, pointing toward the CPU) only during data reads, or have the buffers point toward the CPU as a default and drive the system bus only when the CPU outputs addresses and during data writes. The first choice, with the buffers pointing away from the CPU as a default, is preferable for two reasons. First, the CPU pins will not have to sink current from the buffer as often, since the buffer will normally treat the CPU pins as outputs, thereby minimizing heat dissipation in the CPU. Second, possible bus contention problems when using an in-circuit emulation device for system debug will be avoided. (Bus contention problems occur when both the emulation hardware and target system's bus buffers are simultaneously driving the address/data bus lines to the CPU.) Thus the bidirectional driver for the address/data bus should drive the CPU signals onto the system's bus except during data reads, as indicated by a high on the R/$\overline{W}$ pin and an active $\overline{DS}$ signal.

If bus sharing is allowed in the system, the CPU must relinquish control of the bus when a bus request is made. The bus buffer must provide for other devices, such as a DMA controller, driving the address/data bus whenever $\overline{BUSACK}$ is low. Therefore, a bus buffer whose outputs can be tristated is required.

Possible choices for address/data bus buffering include the SN74LS243 Quad Non-Inverting Transceiver (Fig. 11.3) and the SN74LS245 Octal Non-Inverting Transceiver (Fig. 11.4). The transceivers are controlled by the

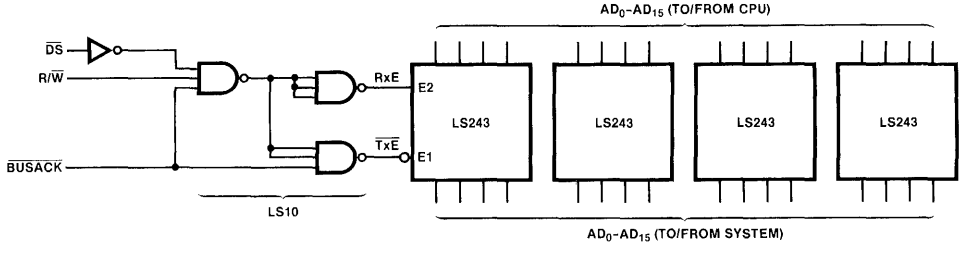


Figure 11.3 Address/data bus buffering using SN74LS243 transceivers.

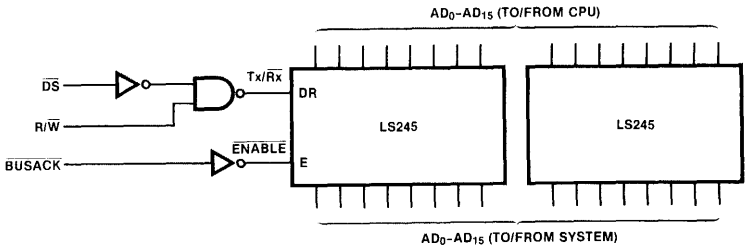


Figure 11.4 Address/data bus buffering using SN74LS245 transceivers.

TABLE 11.1 CONTROL SIGNALS FOR ADDRESS/DATA BUS BUFFERING

$\overline{\text{BUSACK}}$	R/$\overline{\text{W}}$	$\overline{\text{DS}}$	
H	H	L	Enable Receiver (input Data into CPU)
H	H	H	Enable Transmitter (output Address or Data from CPU)
H	L	H	
H	L	L	
L	X	X	Disable Transceiver

$\overline{\text{BUSACK}}$, R/$\overline{\text{W}}$, and $\overline{\text{DS}}$ CPU outputs, as described in Table 11.1. The transmit function of the driver is enabled normally, the receive function is enabled during data reads (R/$\overline{\text{W}}$ high, $\overline{\text{DS}}$ low), and the drivers are tri-stated when $\overline{\text{BUSACK}}$ is active.

The bus control and bus status CPU outputs also are usually propagated to several memory and I/O controllers and, therefore, must be buffered. These signals are always CPU outputs, so a unidirectional driver is adequate. If bus sharing is allowed in the system, the CPU must be able to relinquish control of these signals as well as the address/data bus. A tri-state driver that can be disabled when $\overline{\text{BUSACK}}$ is low is required.

An SN74LS365, SN74LS367, or SN74LS244 tri-state buffer would be adequate for driving the $\overline{\text{AS}}$, $\overline{\text{DS}}$, $\overline{\text{MREQ}}$, R/$\overline{\text{W}}$, N/$\overline{\text{S}}$, and B/$\overline{\text{W}}$ CPU outputs. Figure 11.5 shows control signal buffering with an SN74LS365 device.

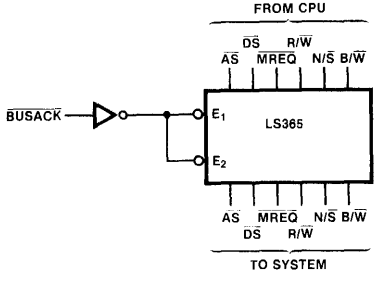


Figure 11.5 Control signal buffering using an SN74LS365 buffer.

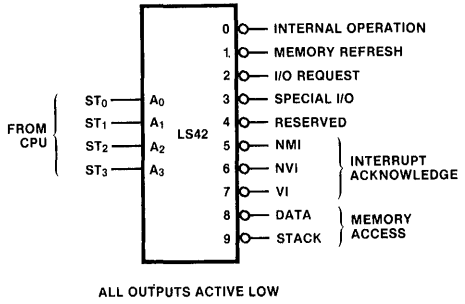


Figure 11.6 Status decoding with an SN74LS42 decoder.

The ST0–ST3 status lines typically drive only one circuit—a decoder that generates the status signals for memory and I/O accesses. A 4-to-16 decoder such as the SN74154 or two 3-to-8 decoders such as the SN74LS138 can be used to decode all 16 possible status conditions. For simple systems without EPUs, only the first 10 status codes need to be decoded. An SN74-LS42 1-of-10 decoder can be used, as in Fig. 11.6. If none of the SN74LS42 decoder outputs are active during a transaction, a program memory access is assumed.

If bus sharing is allowed, the ST0–ST3 status signals may originate from devices other than the CPU. Several decoding schemes are possible. For example, entirely separate decoders might be used for CPU and DMA operations, and the $\overline{\text{BUSACK}}$ signal used to enable the appropriate decoder. Alternatively, the ST0–ST3 status signals from the CPU can be driven by a tri-state buffer that is disabled when $\overline{\text{BUSACK}}$ is active, allowing DMA devices to control those status lines before they are decoded.

The remaining CPU outputs, $\overline{\text{BUSACK}}$ and $\overline{\text{MO}}$, need to be buffered if they are to drive more than one TTL load. Any unidirectional driver is adequate. For the Z8001, the segment number outputs require a unidirectional driver that can be tri-stated when $\overline{\text{BUSACK}}$ is low.

ADDRESS LATCHING

Most semiconductor memories demand that their address inputs remain fixed throughout a memory access; most I/O devices require that their chip select inputs (which are generated by decoding the address) remain active throughout an I/O access. As a result, the Z8000 CPU's address/data bus must be demultiplexed in most systems. Often, systems will latch the address at the beginning of each bus transaction and propagate the latched address to all the memory and I/O controllers. $\overline{\text{AS}}$ is the obvious choice for the control of the address latch. The falling edge of $\overline{\text{AS}}$ cannot be used to clock edge-triggered latches since addresses are not guaranteed to be valid when $\overline{\text{AS}}$ goes low. The rising edge of $\overline{\text{AS}}$ can be used to latch addresses, but addresses are valid before $\overline{\text{AS}}$ rises, so this would delay address availability

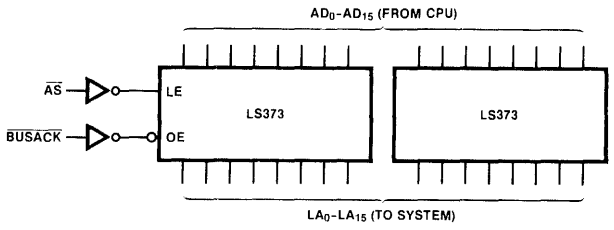


Figure 11.7 Address latching with SN74LS373 latches.

for memory and I/O controllers. Transparent latches that propagate their inputs to their outputs when the control signal is low but hold the outputs fixed when the control signal goes high are a better choice. The rising edge of $\overline{AS}$ could then still be used to signal valid address information at the memory and I/O controllers themselves.

Figure 11.7 shows two SN74LS373 Octal Transparent Latches used for address latching. Depending on the system configuration, these latches might be tri-stated when $\overline{BUSACK}$ goes low. The latched address (LA0-LA15) is propagated to all memory and I/O control logic.

MEMORY INTERFACING

Most microprocessor systems use both volatile (RAM) and nonvolatile (ROM, PROM, or EPROM) memories for program and data storage. Since the Z8000 CPUs read program status information after a reset from locations 0002, 0004, and (for the Z8001 only) 0006, nonvolatile memory usually occupies the lowest address locations in the system.

The Z8000 interface to nonvolatile memories is straightforward, as illustrated in Chapter 3. The single-board system of Fig. 11.1 uses 2K × 8 PROMs, addressed by LA1-LA11. LA0 can be ignored, since Z8000 systems always read a full word during memory reads. The upper address bits are decoded to select particular memory devices using SN74LS138 3-to-8 Decoders or similar devices. LA15 is used to distinguish between the 32K-byte PROM memory space and the 32K-byte RAM memory space. The access time for nonvolatile memories is usually longer than the default memory access time for a Z8000 CPU, so wait states must be added for each PROM access.

Dynamic RAMs such as the 4116 16K × 1 RAM can be used for read/write random-access storage. Most dynamic RAMs use address multiplexing to reduce the package pin count. For example, standard 16K × 1 dynamic RAMs require 14 address inputs. The address is input through seven pins on each memory chip in two steps. First, seven address signals are placed on the memory's address inputs and a clocking signal called row address strobe ($\overline{RAS}$) is lowered. Then the other seven address lines are routed to the memory's address inputs and a second clocking signal called column address

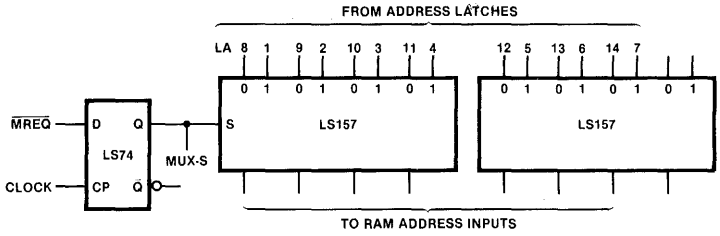


Figure 11.8 Address multiplexing for a dynamic RAM interface.

strobe ($\overline{CAS}$) is lowered. Memory refresh is implemented with a cycle having an active $\overline{RAS}$ but no $\overline{CAS}$. An entire row of memory cells (that is, all the memory locations having seven common address bits) are refreshed by one refresh operation. Interfacing such devices to Z8000 systems involves multiplexing the address and generating the $\overline{RAS}$ and $\overline{CAS}$ signals.

For the system of Fig. 11.1, LA1–LA14 are used as the address inputs to the dynamic RAMs, with LA15 selecting between the PROM and RAM memory areas. LA0 is used only during byte accesses to select one byte of the addressed word. Two 74LS157 Quad 2-to-1 Multiplexors can be used to route the LA1–LA14 address lines into the seven address inputs of the RAMs (Fig. 11.8). $\overline{MREQ}$ is synchronized with the rising edge of the CPU clock to provide the control signal to the multiplexor.

The $\overline{RAS}$ and $\overline{CAS}$ strobes must be timed carefully with respect to the latched address and multiplexor control signals. $\overline{MREQ}$ can be used as $\overline{RAS}$ and $\overline{DS}$ as $\overline{CAS}$; this would, however, considerably shorten the memory access time. The access time would be the delay between $\overline{DS}$ going low and the time that valid data must be valid on the bus in T3 (about 205 ns for a memory read in a 4-MHz system). To allow a longer access time, some synchronous TTL logic can be used to generate the strobes.

One possible configuration is given in Fig. 11.9. $\overline{RAS}$ goes low on the falling edge of the CPU clock in the middle of T1, after $\overline{AS}$ goes low. (In a 4-MHz system, the address emitted by the CPU is guaranteed to be valid within 100 ns after the start of T1; therefore, the address is valid at least 25 ns before $\overline{RAS}$ is active. Of course, delays through intervening latches and buffers must be taken into consideration in an actual system design to assure that the address is valid before an active $\overline{RAS}$.) $\overline{RAS}$ is generated from an SN74LS109 Dual J-K Flip-Flop triggered by an inverted CPU clock signal and stays low for two clock periods. The $\overline{CAS}$ is generated by an SN74LS139 Dual 1-of-4 Decoder. One decoder is used to produce $\overline{CAS}$ by ANDing MUX-S, LA15, and a high R/$\overline{W}$ line or low $\overline{DS}$ line. The other decoder controls the routing of $\overline{CAS}$ to the even or odd byte bank of memory. Both banks are selected except during byte writes, when LA0 is used to select only the even or odd bank. During a read operation, $\overline{CAS}$ becomes active at the beginning of T2 when MUX-S goes high (that is, on the rising edge of the

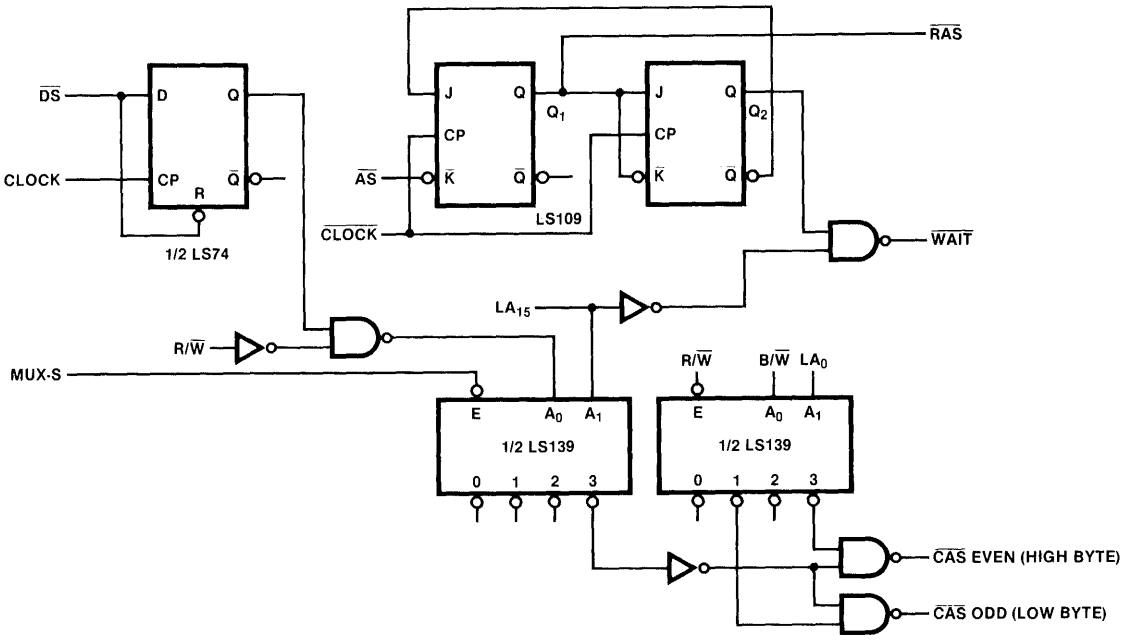


Figure 11.9 RAS and CAS signal generation.

clock after $\overline{\text{MREQ}}$ is active). The second seven address lines are routed to the memory inputs at the same time. During writes, $\overline{\text{CAS}}$ is delayed until $\overline{\text{DS}}$ goes low, guaranteeing that the data to be output are valid before $\overline{\text{CAS}}$ is active. The SN74LS74 Flip-Flop stretches $\overline{\text{CAS}}$ during write operations, as required by slower memories. The timing of $\overline{\text{RAS}}$ and $\overline{\text{CAS}}$ for read and write operations using this circuit is illustrated in Fig. 11.10. $\overline{\text{CAS}}$ is generated only during memory accesses (that is, only when $\overline{\text{MREQ}}$ is active).

PERIPHERAL INTERFACING

Z-Bus-compatible peripherals are available for use in Z8000-based systems, as described in Chapters 12 and 13. However, other peripherals also can be interfaced to the Z-Bus. For example, Z80-family peripherals are easily connected to Z8000 systems with some TTL logic. Z80 peripherals are all byte peripherals; since they are capable of returning a vector during an interrupt acknowledge sequence, they are usually connected to the lower half of the Z-Bus when used in Z8000 systems.

Figure 11.11 shows the timing of Z80 instruction fetch, I/O read, I/O write, and interrupt acknowledge cycles. An active $\overline{\text{M1}}$ signal normally indicates an instruction fetch, but when used in conjunction with $\overline{\text{IORQ}}$ (I/O re-

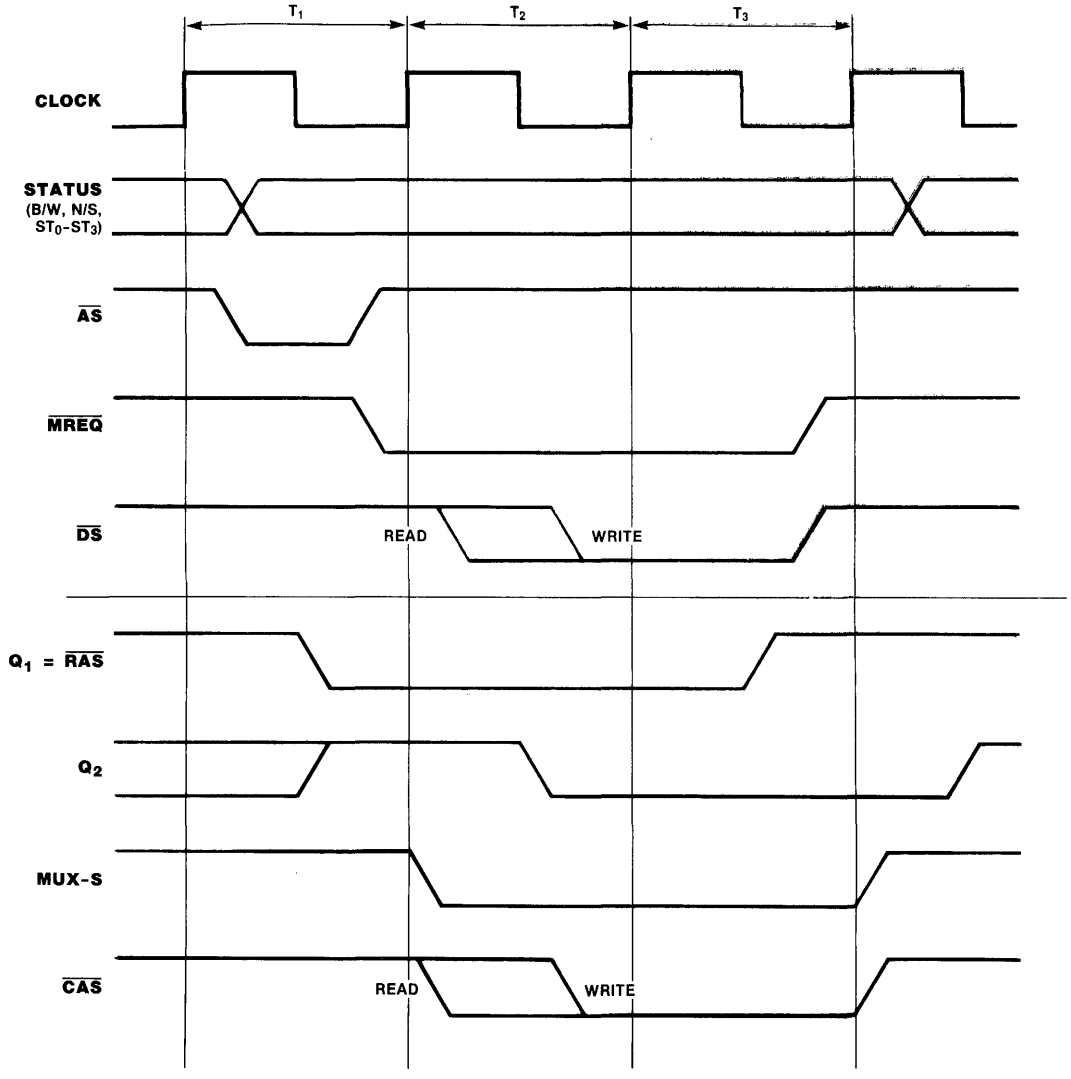


Figure 11.10 Timing of the RAS and CAS signals for dynamic RAM memory accesses.

qest) it indicates an interrupt acknowledge. An active $\overline{IORQ}$ signal without an active $\overline{M1}$, indicates an I/O access. $\overline{RD}$ is used to determine the direction of a data transfer. Interfacing Z80 peripherals to the Z-Bus involves generating the $\overline{M1}$, $\overline{IORQ}$, and $\overline{RD}$ signals to the peripherals from the Z-Bus control and status lines. Four different kinds of operations must be considered: I/O writes, I/O reads, interrupt acknowledges, and interrupt returns.

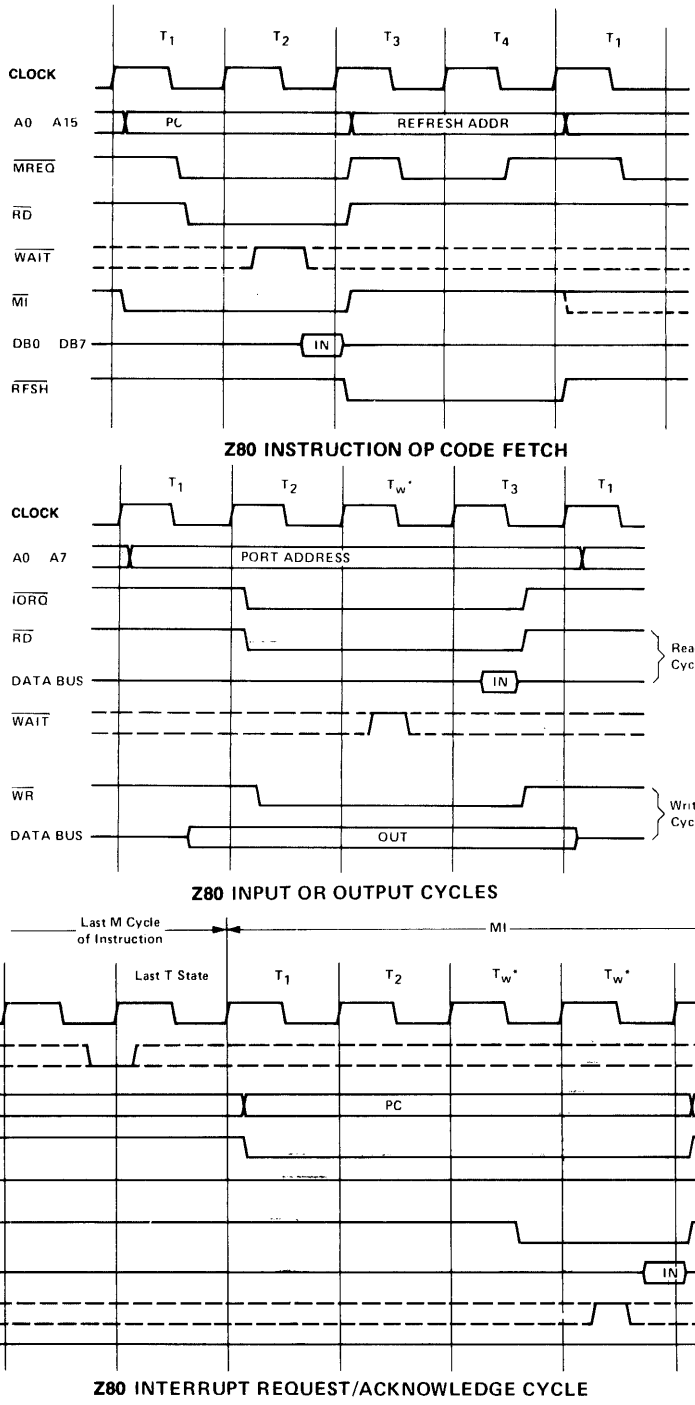


Figure 11.11 Timing of Z80 CPU machine cycles.

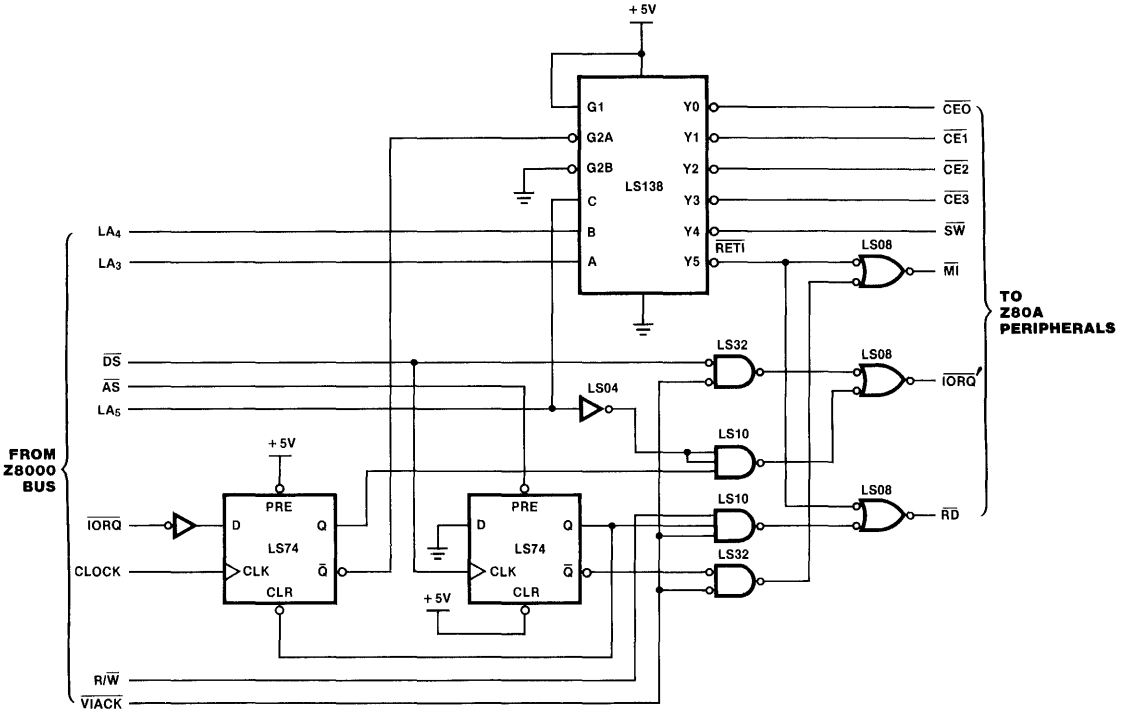


Figure 11.12 Interfacing Z80 peripherals to the Z-Bus.

One possible implementation is shown in Fig. 11.12. An SN74LS138 is used to decode address bits LA3, LA4, and LA5 to generate chip selects to the system's I/O devices. For the system of Fig. 11.1, six peripherals can be accessed: the four Z80 peripherals, the switches, and a special port used to simulate the Z80's interrupt return process (called the RETI port). If more peripherals were in the system, the upper bits of the address would be included in the I/O port decode logic. LA0 cannot be decoded as part of an I/O address, however; since this system's peripherals are all byte peripherals on the lower half of the address/data bus, they must all have odd port addresses. The I/O chip enable logic is activated only during I/O cycles, as indicated by standard I/O status from the ST0-ST3 status decoder ($\overline{IORQ}$ low, where $\overline{IORQ}$ is generated by the status decoder in Fig. 11.6).

Read and write operations are straightforward. A write operation to The peripheral occurs when $\overline{IORQ'}$ is low and $\overline{RD}$ is high; a read from the peripheral occurs when both $\overline{IORQ'}$ and $\overline{RD}$ are low. The I/O status line from the status decoder ($\overline{IORQ}$) is synchronized with the CPU clock and gated with $\overline{DS}$ to generate $\overline{IORQ'}$ to the Z80 peripheral. The $\overline{RD}$ signal is generated from the Z8000 CPU's R/$\overline{W}$ status line. (The Z80 peripherals are

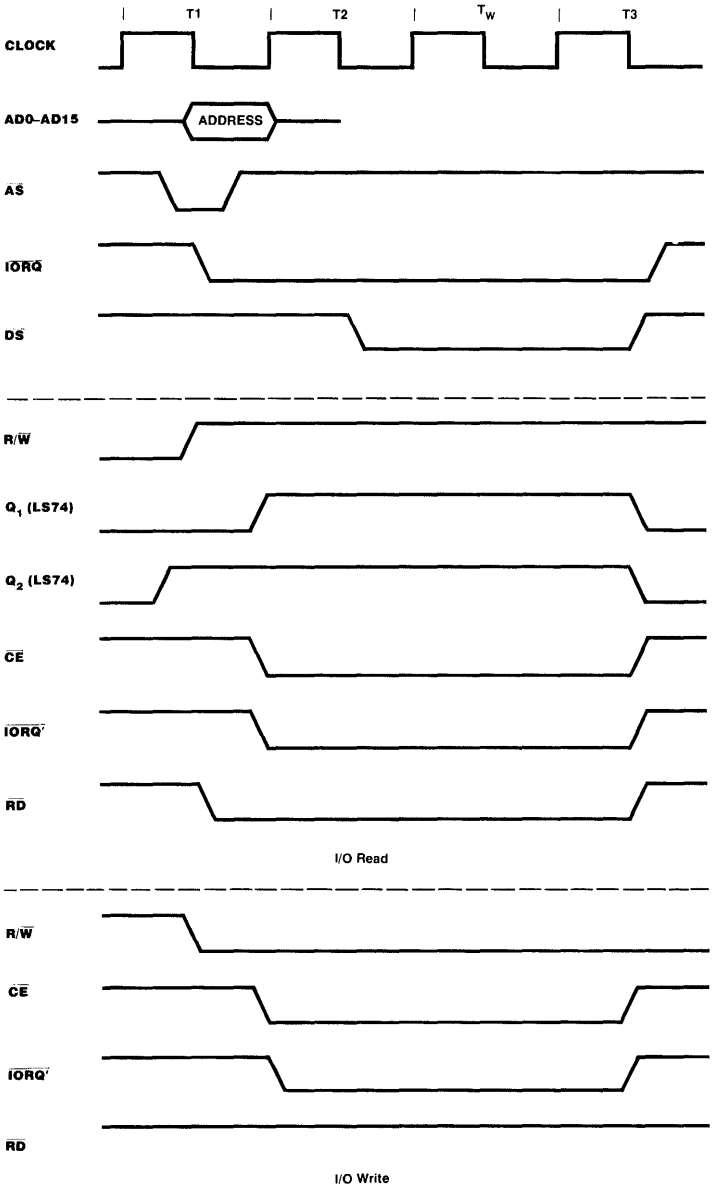


Figure 11.13 Z80 peripheral interfacing read and write timing.

addressed when LA5 is low in this example.) The timing diagram for Z80 peripheral reads and writes using this circuit is illustrated in Fig. 11.13.

Some additional logic is needed to make the Z80 peripherals compatible with the Z8000 interrupt structure. The interrupt outputs from the

peripherals are used to drive a Z8000 CPU interrupt request line; priority among the peripherals is determined by an IEI-IE0 hardware daisy chain similar to that of Z8000 peripherals. The Z80 CPU has no dedicated interrupt acknowledge output. Z80 peripherals are acknowledged when $\overline{IORQ}$ is active during an $\overline{M1}$ cycle (Fig. 11.11).

The circuit of Fig. 11.12 generates the $\overline{M1}$, $\overline{IORQ'}$, and $\overline{RD}$ signals required by the Z80 peripherals during an interrupt acknowledge cycle. $\overline{VIACK}$ is the appropriate interrupt acknowledge signal from the ST0-ST3 status decoder. A timing diagram for the acknowledge cycle using this circuit is given in Fig. 11.14.

At the end of the service routine for an interrupt generated by a Z80 peripheral, the interrupt-under-service flag in the peripheral must be reset. The Z80 CPU's interrupt return instruction (RETI) has an opcode of ED4D (hex); a Z80 fetch of this instruction takes two consecutive instruction fetch cycles. A Z80 peripheral will monitor the data bus while under service and recognize when the RETI instruction is fetched from memory. When the "ED4D" code is sensed, the interrupt-under-service latch in the peripheral is reset.

In a Z8000 system with Z80 peripherals, the Z80's RETI instruction fetch sequence must be simulated at the end of the peripheral's service routine with a combination of hardware and software. The necessary hardware is part of the circuit of Fig. 11.12. A special I/O port address, the "RETI"

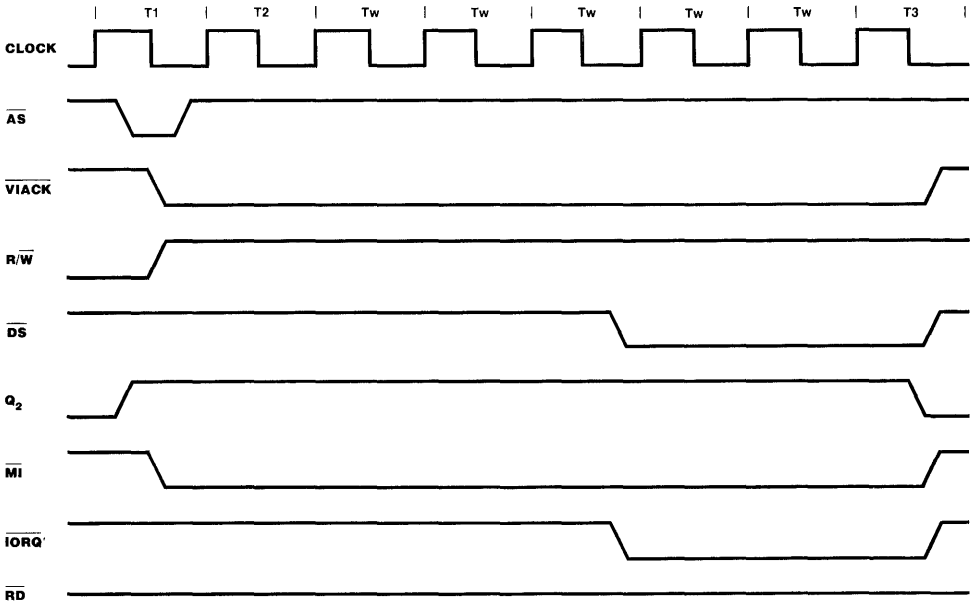


Figure 11.14 Z-80 peripheral interface interrupt acknowledge timing.

port, is used to place an "ED" and "4D" code on the peripherals data inputs while manipulating the $\overline{MI}$ and $\overline{RD}$ lines to simulate Z80 instruction fetches. The software routine is as follows:

```
DI      VI,   NVI      ! disable interrupts !
LDB     RL1,  #%ED     ! load first byte of RETI opcode !
OUTB    RETI, RL1      ! RETI is port address for I/O port used to simulate Z80 instruc-
                         tion fetch !
LDB     RL1,  #%4D     ! load second byte of RETI opcode !
OUTB    RETI, RL1      ! output second byte and simulate Z80 I-fetch !
EI      VI,   NVI      ! enable interrupts !
IRET                   ! end service routine !
```

The two simulated instruction fetches for the Z80 RETI must be consecutive operations at the peripheral and, therefore, interrupts of the Z8000 CPU should be disabled while this sequence is executed, as shown. The timing of the signals to the Z80 peripherals during this simulated RETI sequence is illustrated in Fig. 11.15. (In a system with bus sharing, bus requests also must

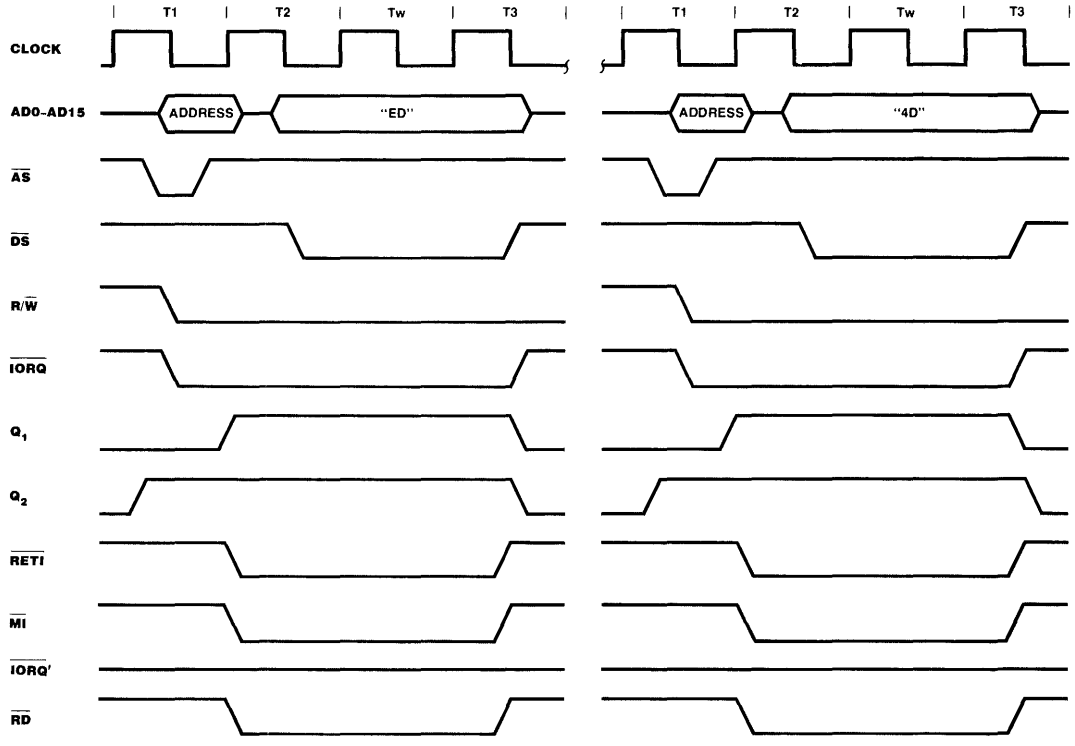


Figure 11.15 Z80 peripheral interface return from interrupt (RETI) timing.

be disabled between the two output operations that simulate the Z80 RETI sequence. This might require some additional hardware to gate the BUSREQ input to the CPU.)

The address/data bus can be connected directly to the D0-D7 data lines of the Z80 peripherals. In the system of Fig. 11.1, latched address lines LA1 and LA2 are used as the Port Select ($\overline{A}$/B) and Control/Data (C/$\overline{D}$) signals for the PIO and SIO, and as Channel Select (CS0 and CS1) for the CTC. The interrupt outputs are all connected to the appropriate interrupt request input of the Z8000 CPU. The IEI-IEO daisy chain is used to establish interrupt priorities among the Z80 peripherals.

The Z8420 PIO needs an active $\overline{M1}$ pulse to enable its internal interrupt circuitry. This is accomplished by writing 0's to the "RETI" port used to simulate the Z80 RETI instruction fetch after the PIO interrupts have been enabled.

In summary, a small, powerful microcomputer system can be implemented with a Z8000 CPU, memory, peripheral devices, and a small amount of TTL logic. Buffering of the CPU outputs is necessary in all but the smallest systems. Interfacing the Z8000 CPU to nonvolatile memory is straightforward; interfacing to dynamic RAMs requires the generation of the proper timing strobes. If peripherals from other microprocessor families are used, the control signals for those peripherals must be generated from the Z8000 CPU's bus control and status signals. Z80 family peripherals interface easily to Z8000 systems, largely due to the similarity in the Z80 and Z8000 interrupt structures.

12

Z8000 Family Devices

The Z8001 and Z8002 CPUs are just two members of a family of devices designed to interact over the Z-Bus. Other members of the Z8000 family include several Z-Bus-compatible peripherals, a DMA controller, and a Z-Bus memory.

The Z-Bus peripheral chips are powerful, multifunction devices that can be configured under program control for a particular application. All of the Z-Bus peripherals share a common interrupt structure and can be used in a priority interrupt or polled environment. The functions of each device are controlled by using I/O commands to access the peripheral's internal registers; each register has its own I/O port address. The Z8036 Counter/Timer and Parallel I/O Unit (CIO) contains three parallel ports and three counter/timers; it also can be used as a priority interrupt controller. The Z8038 FIFO Input/Output Unit (FIO) is a byte-wide first-in/first-out buffer for interfacing asynchronous devices in a single or multiprocessor system. The buffer depth is expandable with the Z8060 FIFO Buffer Unit. The Z8030 Serial Communications Controller (SCC) is a dual-channel serial I/O unit that supports all popular synchronous and asynchronous communications protocols. The Z8065 Burst Error Processor (BEP) provides error correction and detection capabilities for high-speed data transfers. The Z8068 Data Ciphering Processor (DCP) encrypts or decrypts data using the National Bureau of Standards encryption algorithms. The Z8052 CRT Controller (CRTC) can be used to control a variety of CRT displays. These peripherals each perform complicated interfacing tasks, thereby unburdening the CPU and increasing system throughput.

The Z8016 Direct Memory Access Transfer Controller (DTC) is both a

Z-Bus requester and a Z-Bus peripheral. The DTC is programmed by the CPU via I/O operations and can interrupt the CPU like a peripheral device, but acts as a bus master when executing DMA transfers.

The Z6132 Quasi-Static RAM is a 4K × 8 Z-Bus-compatible memory device that is easily interfaced to Z8000 systems.

[Two other Z-Bus components, the Z8 single-chip microcomputer and the Universal Peripheral Controller (UPC), a slave microcomputer, are described in Chapter 13.]

Z-BUS PERIPHERAL INTERFACE

All of the Z-Bus peripherals are byte peripherals, with the exception of the Z8052 CRT controller. Figure 12.1 illustrates the signals used to interface a byte peripheral to the Z-Bus. One-half of the address/data bus provides up to 8 bits of address information for directly addressing the peripheral's internal registers and an 8-bit data path for data transfers between the peripheral and the CPU. (Typically, the lower half of the address/data bus is used, since interrupt vectors must be placed on the lower half of the bus when using vectored interrupts.) Timing of the data transfers is controlled by the address strobe ($\overline{AS}$) and data strobe ($\overline{DS}$), and the direction of transfers is determined by the R/W signal (Fig. 12.2). The chip select ($\overline{CS}$) for a peripheral is decoded from the I/O port address during I/O accesses and is latched internally by the peripheral on the rising edge of $\overline{AS}$. Resets are implemented when both the $\overline{AS}$ and $\overline{DS}$ inputs to the peripheral are low simultaneously. (In normal operation, $\overline{AS}$ active and $\overline{DS}$ active are mutually exclusive events.) The $\overline{INT}$, $\overline{INTACK}$, IEI, and IEO signals interface the peripheral to the Z-Bus interrupt structure.

Other signals, such as $\overline{WAIT}$, also might be part of the peripheral-to-Z-Bus interface, depending on the application. Some of the Z-Bus peripherals

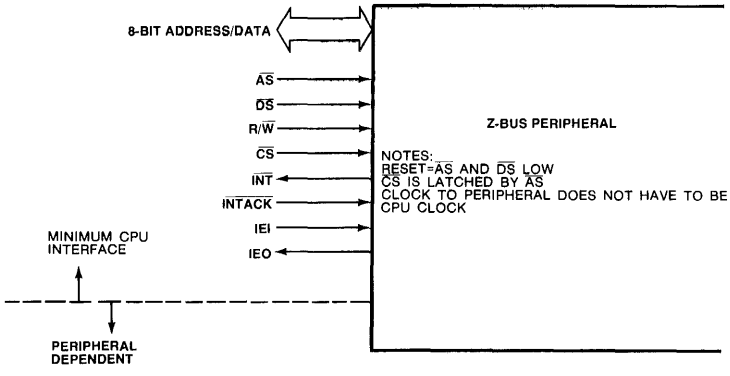


Figure 12.1 Z-Bus peripheral interface signals.

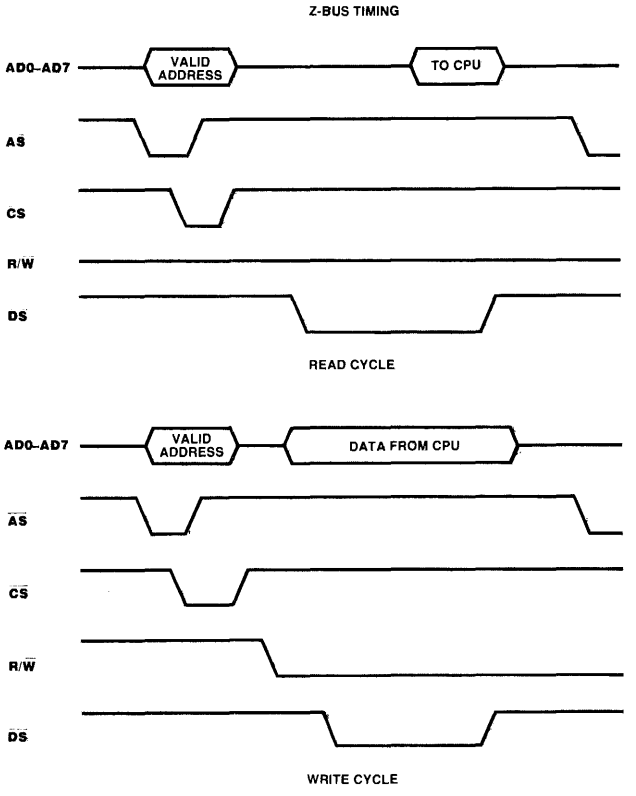


Figure 12.2 Z-Bus to peripheral interface timing.

have a clock input; the peripheral's clock does not have to be synchronized with the CPU clock in any way.

(Most of these peripherals also are available in a version that easily interfaces to systems with separate, nonmultiplexed address and data buses, such as Z80-based systems.)

Each Z-Bus peripheral implements several functions and is programmable for a particular application. A peripheral may have up to 128 internal registers that can be read or written by the CPU using I/O instructions. Each register has its own I/O port address; thus each peripheral occupies a block of port addresses in the system. The address of the register being accessed is sampled on the peripheral's address/data bus inputs on the rising edge of $\overline{AS}$. A programmable option allows the user to decide if the least signficant bit of the register address is on the AD0 or AD1 bus line. For byte transfers in Z8000-based systems, the value of AD0 when the CPU emits the address determines which half of the bus will be used to transfer the byte of data. For example, byte peripherals connected to the lower half of the bus always have odd addresses. Therefore, in Z8000 systems, the least significant bit of the peripheral's register address should be on the AD1 line, not AD0.

If several Z-Bus peripherals share a common interrupt request line to the CPU, the IEI-IEO daisy chain is used to establish the relative priority of those peripherals. When one or more peripherals request the CPU's attention via an interrupt, the interrupt acknowledge cycle is used to select the peripheral whose interrupt is to be serviced and to obtain a vector or identifier word, as described in Chapter 6.

Each of the Z-Bus peripheral devices can have several sources of interrupt internal to that chip. For peripherals with multiple interrupt sources on one chip, priority is established with an IEI-IEO daisy chain internal to the device. This prioritization order is fixed and cannot be altered by the user. Figure 12.3 illustrates the Z-Bus interrupt structure for several peripherals sharing an interrupt line, and for one peripheral with several sources of interrupts internal to that device.

For every interrupt source on a Z-Bus peripheral there are three bits within the device's internal registers that control the interrupt logic. The Interrupt Enable (IE) bit is set to enable or reset to disable that particular interrupt source. The Interrupt Pending (IP) bit is set when the device requires servicing and reset when the interrupt is serviced. The Interrupt Under Service (IUS) bit indicates when the interrupt is being serviced and must be reset by the programmer upon completion of the service routine.

A Z-Bus peripheral has one or more registers that hold an interrupt vector that is read by the CPU during the interrupt acknowledge cycle. Each interrupt source on a device is associated with a vector and each vector can have one or more interrupt sources associated with it. If more than one interrupt source is associated with a single vector, some bits in the vector can be encoded to identify which source caused the interrupt. A bit called the Vector Includes Status (VIS) bit is used to enable or disable this encoding function.

Each peripheral also has three programmable bits that control the interrupt logic for all interrupt sources on the device. The Master Enable (MIE) bit is used to enable or disable all interrupt sources on the chip. The Disable Lower Chain (DLC) bit is used to force the peripheral's IEO output to 0, thereby disabling interrupts from peripherals of lower priority on the daisy chain. The No Vector (NV) bit is set if a vector is not to be placed on the bus during the interrupt acknowledge sequence.

Figure 12.4 illustrates the interrupt daisy-chain protocol as it applies to the Z-Bus peripherals. An interrupt source with an interrupt pending (IP = 1) requests an interrupt by pulling $\overline{\text{INT}}$ low if the IE bit for that source and the MIE bit for that device are both set, that interrupt source is not already under service (IUS = 0), no higher-priority device's interrupt sources are being serviced (IEI = 1), and an interrupt acknowledge cycle is not currently being executed (INTACK = 1). After the CPU samples the active interrupt request, an interrupt acknowledge cycle is executed, as indicated by $\overline{\text{INTACK}}$ going

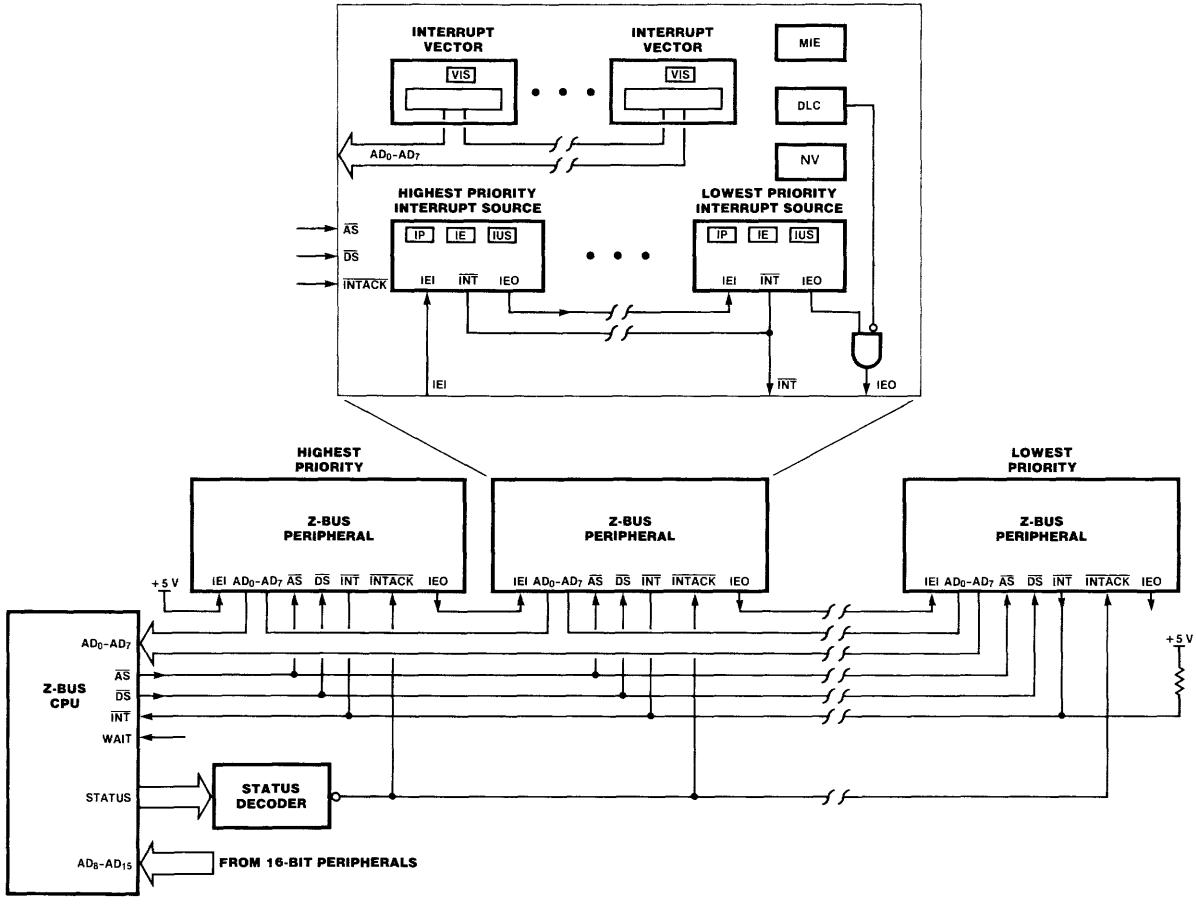


Figure 12.3 Z-Bus peripheral interrupt structure.

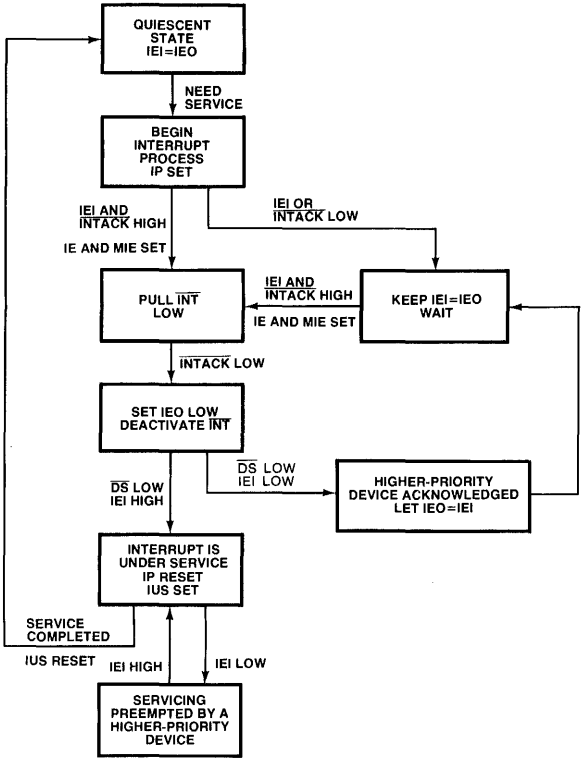


Figure 12.4 Peripheral interrupt protocol.

low. When $\overline{\text{INTACK}}$ is active, all interrupt sources with an interrupt pending (the IP, IE, and MIE bits are all 1's) or under service (the IUS bit = 1) hold their IEO outputs low. When $\overline{\text{DS}}$ goes low during the acknowledge cycle, only the highest-priority interrupt source with an interrupt pending (IP = 1) should have a high IEI input; this is the interrupt being acknowledged. The IP bit is reset for that interrupt source, the IUS bit is set, and, if the NV bit = 0, the appropriate vector is placed on the bus to be read by the CPU. If the NV bit is 1, the peripheral's AD0-AD7 pins are left floating, allowing external circuitry to place a vector on the bus, if so desired. While servicing of that interrupt is in progress, as indicated by IUS = 1, IEO is held low, thereby disabling interrupts from lower-priority devices. When servicing is completed, the IUS bit must be reset. The CPU resets the IUS bit by an explicit I/O write to the register in the peripheral that contains the IUS bit; this I/O write operation usually is executed immediately preceding the interrupt return in the service routine. (In most cases, the IP bit is not reset automatically during the acknowledge sequence, and also must be reset via an explicit write to the appropriate register in the peripheral.)

A polled interrupt scheme can be implemented by disabling interrupts using the MIE bit in each peripheral. The registers containing the IP bits are

read by the CPU via I/O read operations to detect pending interrupts. The IP bits must be reset by writes to the same registers.

Z8036 CIO

The Z8036 Counter/Timer and Parallel I/O Unit (CIO) contains three parallel ports and three programmable counter/timers, satisfying most parallel I/O and counter/timer needs in Z8000 systems. There are five distinct interrupt sources in the CIO, and three separate interrupt vectors. The configuration of the CIO is controlled by 48 directly addressable read/write registers. The pin assignments for this 40-pin device are given in Fig. 12.5. The CIO operates from a single +5-V power supply and draws a maximum of 250 mA. Figure 12.6 is a block diagram of the CIO.

The CIO's parallel I/O capabilities consist of two 8-bit general-purpose ports and one 4-bit special-purpose port. The two 8-bit ports, ports A and B, can be linked together to form a single 16-bit port. Either port can be configured as a byte port (that is, an entire byte of input or output) or as a bit port (the direction of each bit is programmable). Figure 12.7 shows a block diagram of these ports. Optionally, port B pins can be used to provide external access to counter/timers 1 and 2. With this exception, ports A and B are identical.

When configured as byte ports, ports A and B can be input, output, or bidirectional ports. I/O operations can be interrupt driven; both port A and

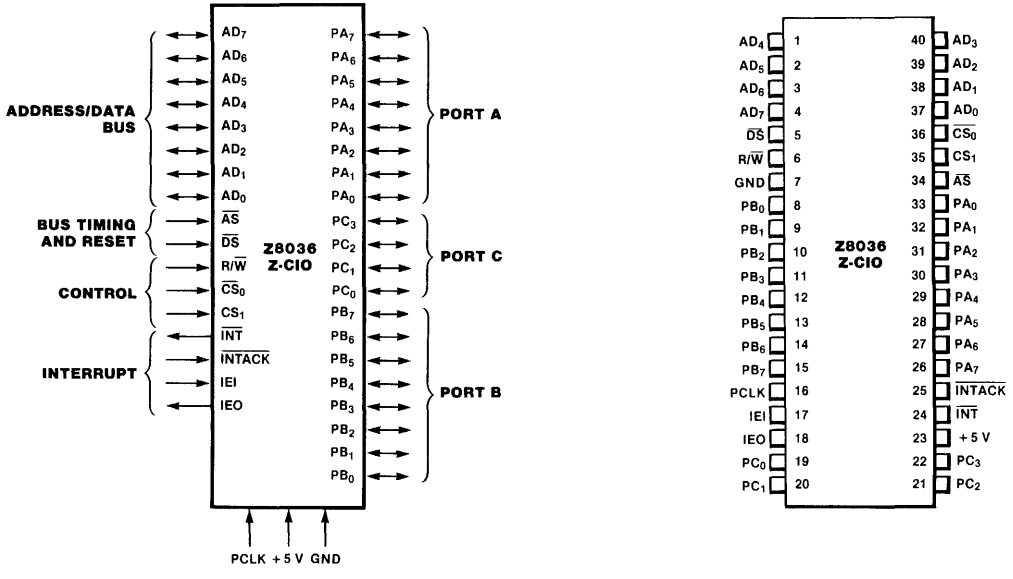


Figure 12.5 Z8036 CIO pin assignments.

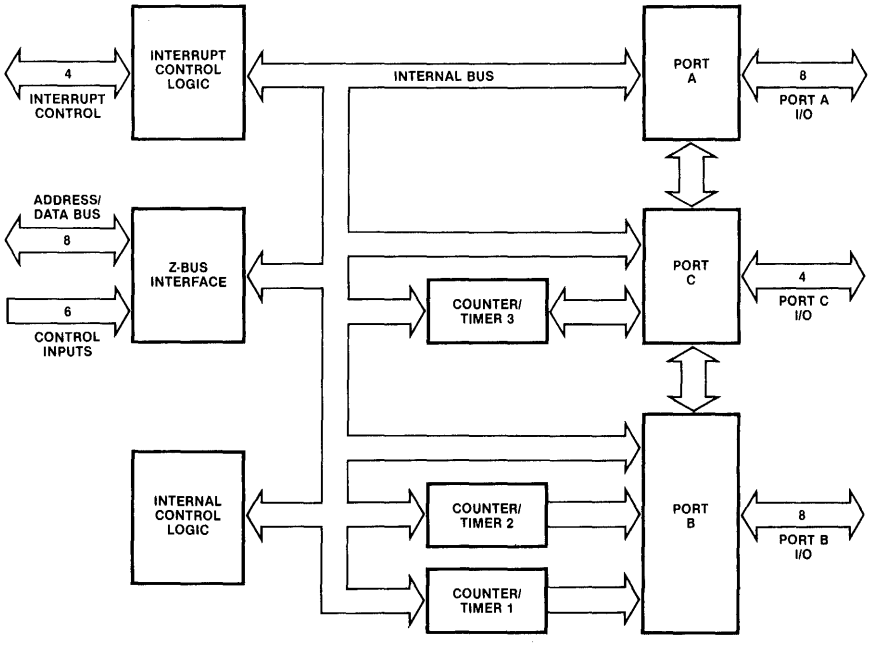


Figure 12.6 Z8036 CIO block diagram.

port B are an interrupt source. Byte ports can be single- or double-buffered, and the interrupt logic programmed to interrupt for every byte transferred or for every second byte transferred, accordingly. Optionally, port C signals can be used as handshake lines to control I/O operations on ports A and B. Four kinds of handshakes are available: interlocked, strobed, pulsed, and three-wire (Fig. 12.8). The interlocked, strobed, and pulsed handshakes are implemented with two signals: acknowledge in ($\overline{\text{ACKIN}}$) and ready for data (RFD) for input handshakes, and acknowledge in and data available ($\overline{\text{DAV}}$) for output handshakes. The three-wire handshake [data available ($\overline{\text{DAV}}$), ready for data (RFD), and data accepted (DAC)] is compatible with the handshake in the IEEE-488 bus specification. For output ports, a programmable 4-bit deskew timer is available for determining the delay between valid data being output and the falling edge of the $\overline{\text{DAV}}$ handshake line. Outputs can be programmed as open-drain or active signals. Data polarity is programmable on a bit-by-bit basis.

When port A or port B is used as a bit port, both data direction and data polarity are programmable on a bit-by-bit basis. Optionally, inputs can be "one's catchers" (that is, programmed to remain at a logical 1 level until read if a low-to-high transition occurs) and outputs can be open-drain or active.

Pattern recognition logic is available for both ports A and B, regardless of whether they are used as byte or bit ports, allowing interrupt generation

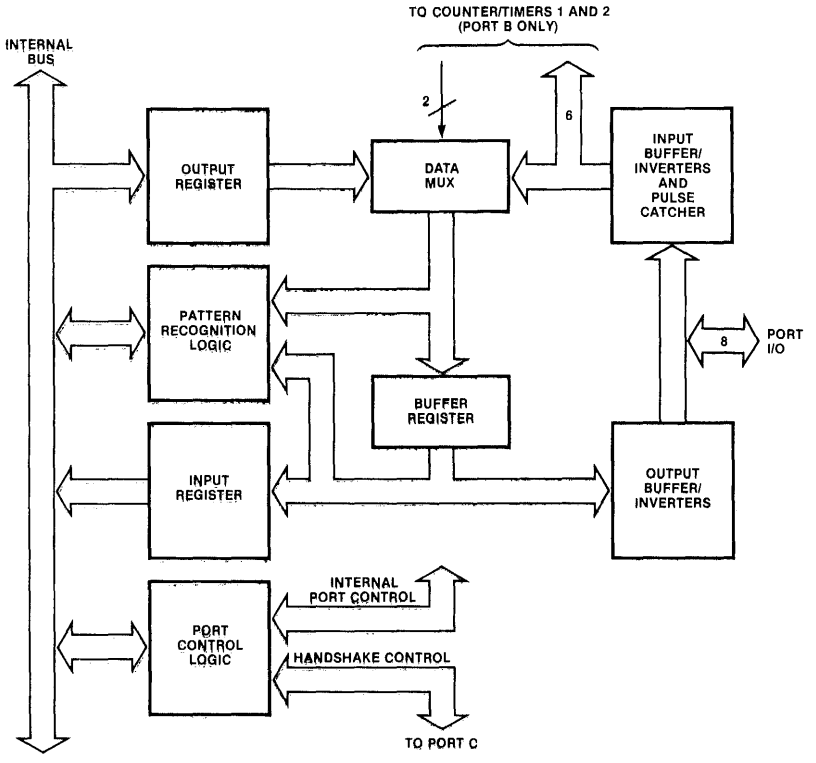
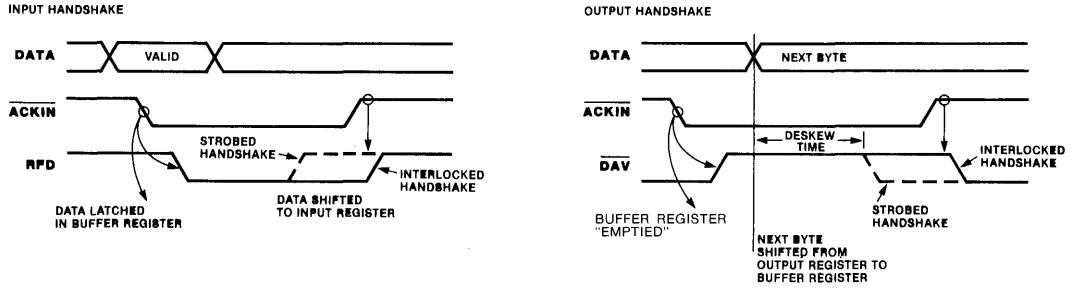


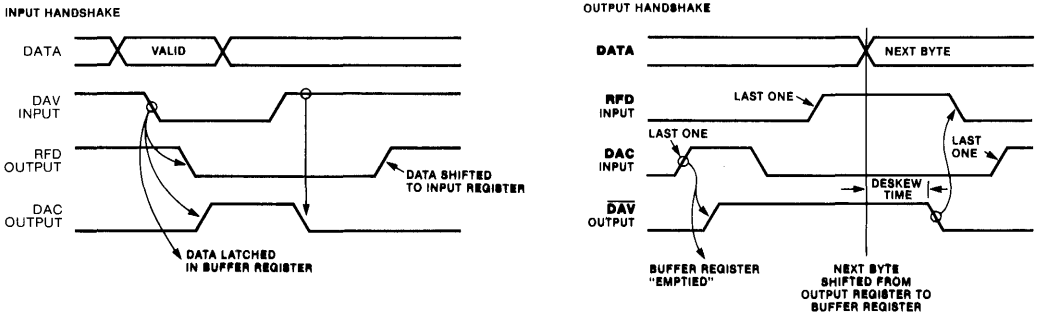
Figure 12.7 CIO ports A and B diagram.

when a specific pattern is detected at the port. The pattern can be specified for each bit as a 1, 0, rising edge, falling edge, or any transition. Individual bits can be masked if they are not to be included in the pattern match. Three pattern-match modes are available: AND, OR, and OR-Priority Encoded Vector. In the AND mode, a pattern match is defined as the simultaneous satisfaction of all nonmasked bit specifications. In the OR and OR-Priority Encoded Vector modes, the satisfaction of any one nonmasked bit specification constitutes a pattern match.

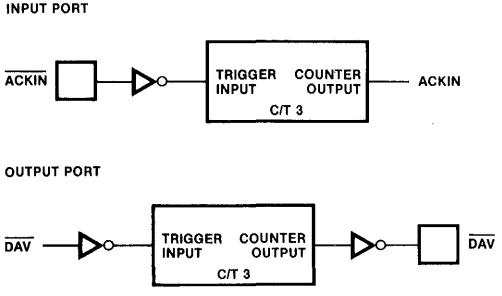
The OR-Priority Encoded Vector mode allows the CIO to be used as an interrupt priority controller. In this mode, the IP bit is set when a pattern match occurs and cannot be cleared until a match is no longer present. If the interrupt vector is allowed to include status information (VIS = 1), the vector returned during the interrupt acknowledge cycle indicates the highest priority bit matching its pattern-match specification at the time of the acknowledge; this may or may not be the bit that originally caused the pattern-match interrupt. Bit 7 has the highest priority and bit 0 the lowest. Thus a CIO port could accept interrupt inputs from other devices on an input port and



Interlocked and Strobed Handshakes



3-Wire Handshake



Pulsed Handshake

Figure 12.8 Ports A and B handshakes.

act as an interrupt priority controller for those devices. For example, if a low level indicates an interrupt request from devices #0 through #7 in Fig. 12.9, the pattern match logic for CIO port A would be set to match on 0's in OR-Priority Encoded Vector mode. An active $\overline{INT}$ from any device would cause the CIO to interrupt the CPU. The vector returned to the CPU would indi-

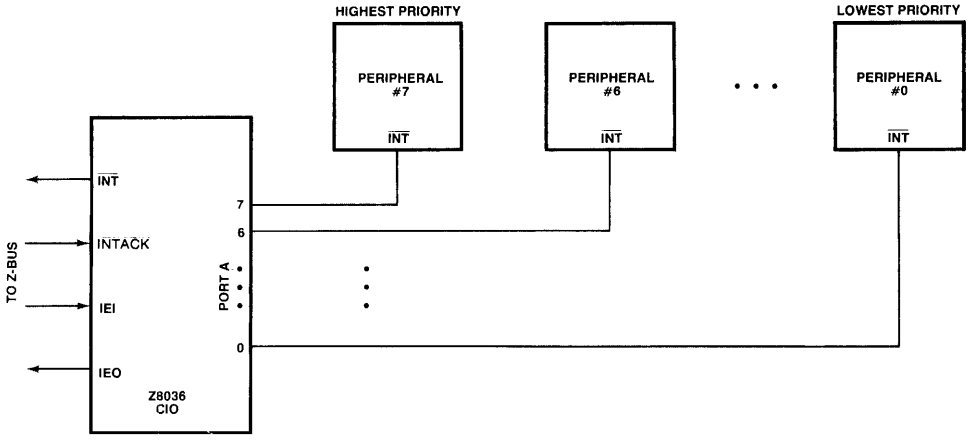


Figure 12.9 CIO as an interrupt controller.

cate the highest-priority device with an active $\overline{\text{INT}}$ signal at the time of the acknowledge. When desired, interrupts from selected devices could be disabled by masking the appropriate bits in the CIO's pattern match logic. Thus the CIO is used as an interrupt controller. Furthermore, this provides an easy method for interfacing non-Z8000 family peripherals to the Z-Bus interrupt structure.

The function of the 4-bit special-purpose port, port C, depends on the configuration of ports A and B (Table 12.1 and Fig. 12.10). Port C provides the handshake lines for ports A and B and the data direction line for a bidirectional port. One bit of port C can be programmed as a $\overline{\text{WAIT}}$ signal to the CPU or a REQUEST signal to a DMA controller, thereby allowing block

TABLE 12.1 CIO PORT C BIT UTILIZATION

Port A/B configuration	PC_3	PC_2	PC_1	PC_0
Ports A and B: bit ports	Bit I/O	Bit I/O	Bit I/O	Bit I/O
Port A: input or output port (interlocked, strobed, or pulsed handshake)[a]	RFD or $\overline{\text{DAV}}$	$\overline{\text{ACKIN}}$	REQUEST/$\overline{\text{WAIT}}$ or bit I/O	Bit I/O
Port B: input or output port (interlocked, strobed, or pulsed handshake)[a]	REQUEST/$\overline{\text{WAIT}}$ or bit I/O	Bit I/O	RFD or $\overline{\text{DAV}}$	$\overline{\text{ACKIN}}$
Port A or B: input port (three-wire handshake)	RFD (output)	$\overline{\text{DAV}}$ (input)	REQUEST/$\overline{\text{WAIT}}$ or bit I/O	DAC (output)
Port A or B: output port (three-wire handshake)	$\overline{\text{DAV}}$ (output)	DAC (input)	REQUEST/$\overline{\text{WAIT}}$ or bit I/O	RFD (input)
Port A or B: bidirectional port (interlocked or strobed handshake)	RFD or $\overline{\text{DAV}}$	$\overline{\text{ACKIN}}$	REQUEST/$\overline{\text{WAIT}}$ or bit I/O	IN/$\overline{\text{OUT}}$

[a]Both ports A and B can be specified input or output with interlocked, strobed, or pulsed handshake at the same time if neither uses REQUEST/$\overline{\text{WAIT}}$.

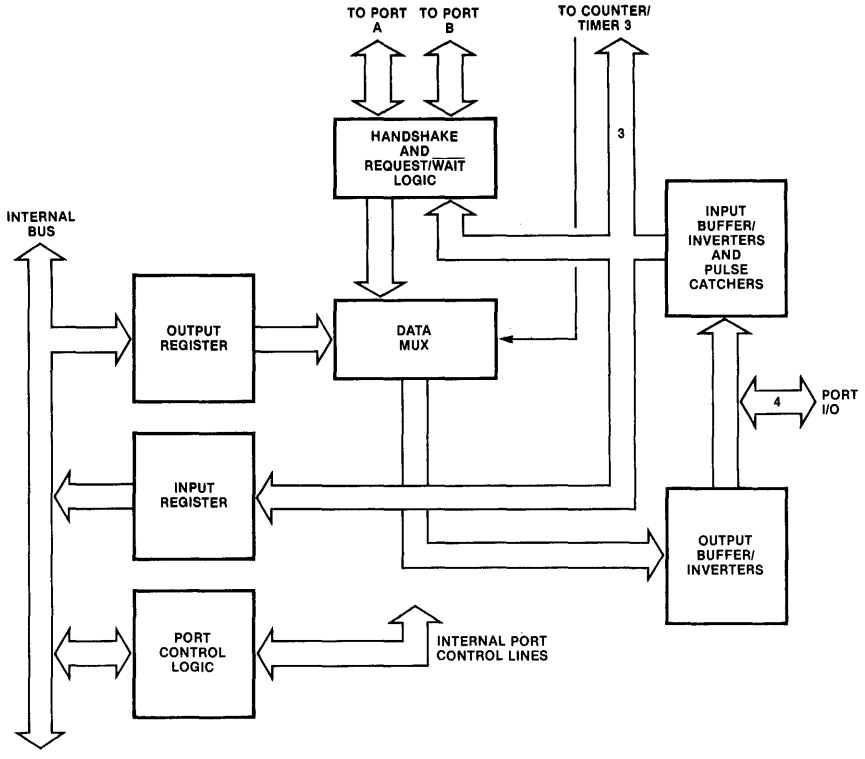


Figure 12.10 CIO port C block diagram.

transfers to or from the CIO. Any port C pins not used for those functions can be used as I/O lines with programmable data direction and polarity. Optionally, inputs can be "one's catchers" and outputs can be open-drain or active. The port C pins also can be used to provide external access to counter/timer 3.

The three programmable counter/timers are 16-bit down counters (see Fig. 12.11). Up to four external I/O lines can be used to control each counter/timer: counter input, gate input, trigger input, and counter/timer output. These external access lines are provided by port B and port C pins (Table 12.2). Optionally, the device's clock input (PCLK/2) can be used to drive any counter/timer. If the counter/timer output is routed to an external pin, three output waveforms are available: pulse, one-shot, and square wave (Fig. 12.12). The end-of-count condition can be used to generate an interrupt. The counter/timers can be run in single-cycle or continuous modes; if a trigger input is employed, retriggerable or nonretriggerable operation can be specified. The current count can be read at any time. Typical applications for the counter/

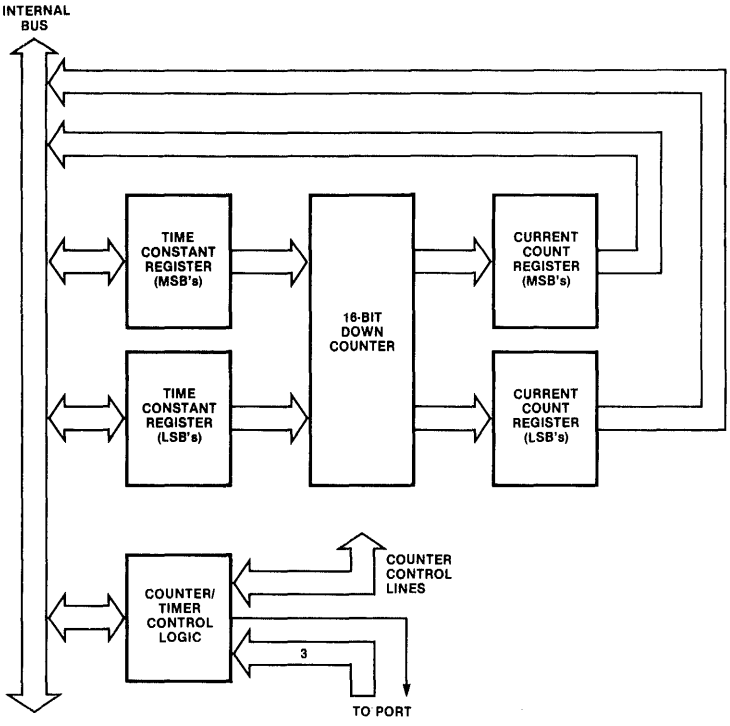


Figure 12.11 CIO counter/timer block diagram.

timers include event counters, pulse train generators, event duration timers, watchdog timers, and baud-rate clock generators.

The five interrupt sources for the CIO are, in priority order, counter/timer 3, port A, counter/timer 2, port B, and counter/timer 1. Each source has its own IE, IP, and IUS bits to control that interrupt. Three interrupt vectors can be specified: one for port A, one for port B, and one shared by all three counter/timers. The vectors can be encoded with status information to identify further the event that caused the interrupt. For polled operations a

TABLE 12.2 CIO COUNTER/TIMER EXTERNAL ACCESS

Function	C/T_1	C/T_2	C/T_3
Counter/timer output	PB 4	PB 0	PC 0
Counter input	PB 5	PB 1	PC 1
Trigger input	PB 6	PB 2	PC 2
Gate input	PB 7	PB 3	PC 3

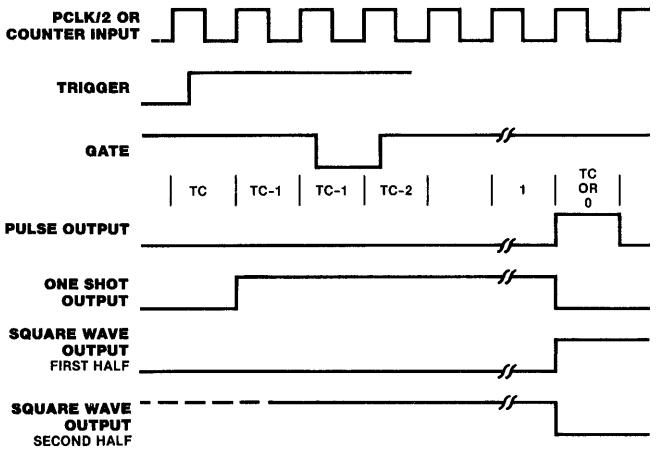


Figure 12.12 Counter/timer output waveforms.

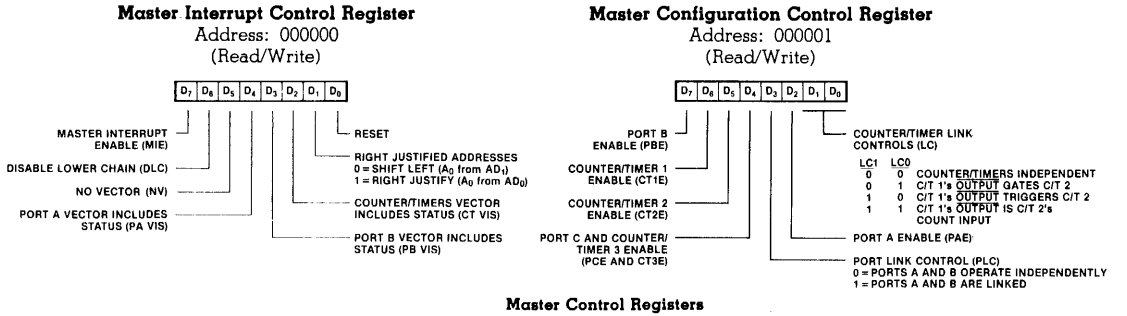
special register, called the current vector register, holds the vector that would have been returned if hardware interrupts were used.

Figure 12.13 shows all 48 CIO registers. Programming the CIO involves loading the registers with the appropriate bit pattern to implement the desired operation. Addressing of the registers is determined by the Right Justify Address (RJA) bit in the Master Interrupt Control register. If RJA is 0, the address bits on AD1-AD6 during $\overline{AS}$ active in an I/O cycle are decoded as the register address if the CIO is chip selected. When RJA is 1, the register address is decoded on AD0-AD5. The 6-bit register addresses are given in Fig. 12.13.

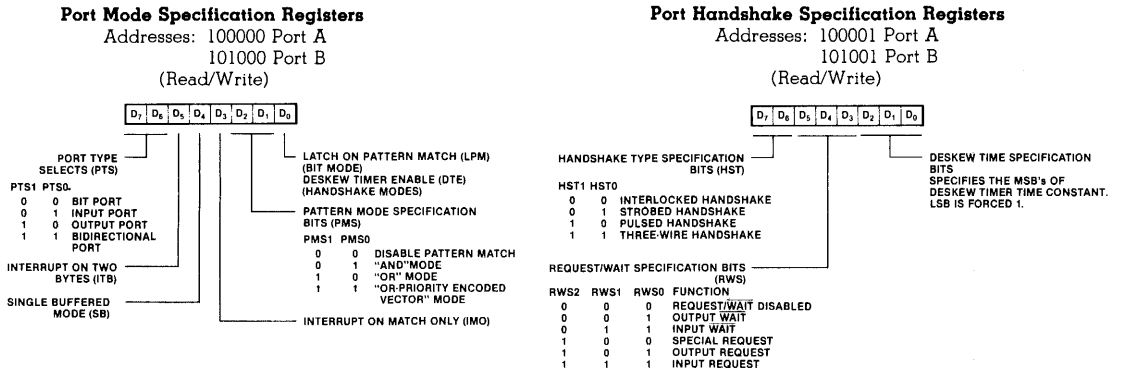
A typical Z-Bus to CIO interface is diagrammed in Fig. 12.14. The CIO's address/data pins are connected to the lower half of the Z-Bus address/data bus. The address bit on AD0 should always be a 1 for byte transfers on the lower half of the bus, so the RJA bit should be a 0 and AD1-AD6 provide the register addresses. AD7-AD15 are decoded to provide chip selects during I/O transactions (with $\overline{\text{I/O STATUS}}$ decoded from the ST0-ST3 lines). The CIO is chip selected by I/O port address %FF80 to %FFFF; however, only odd addresses are used and only 48 of those addresses actually access a CIO register. An active $\overline{\text{RESET}}$ signal resets the CIO by pulling $\overline{AS}$ and $\overline{DS}$ low simultaneously.

Z8038 FIO

The Z8038 FIFO Input/Output Interface Unit (FIO) contains a 128-byte first-in/first-out buffer that provides an asynchronous CPU-to-CPU or CPU-to-peripheral interface. One side of the FIO, the port 1 side, can be configured as a Z-bus or general-purpose nonmultiplexed bus interface to a microprocessor;



Master Control Registers



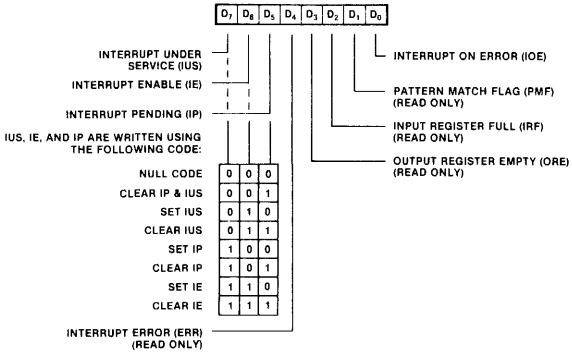


Figure 12.13 Z8036 CIO registers.

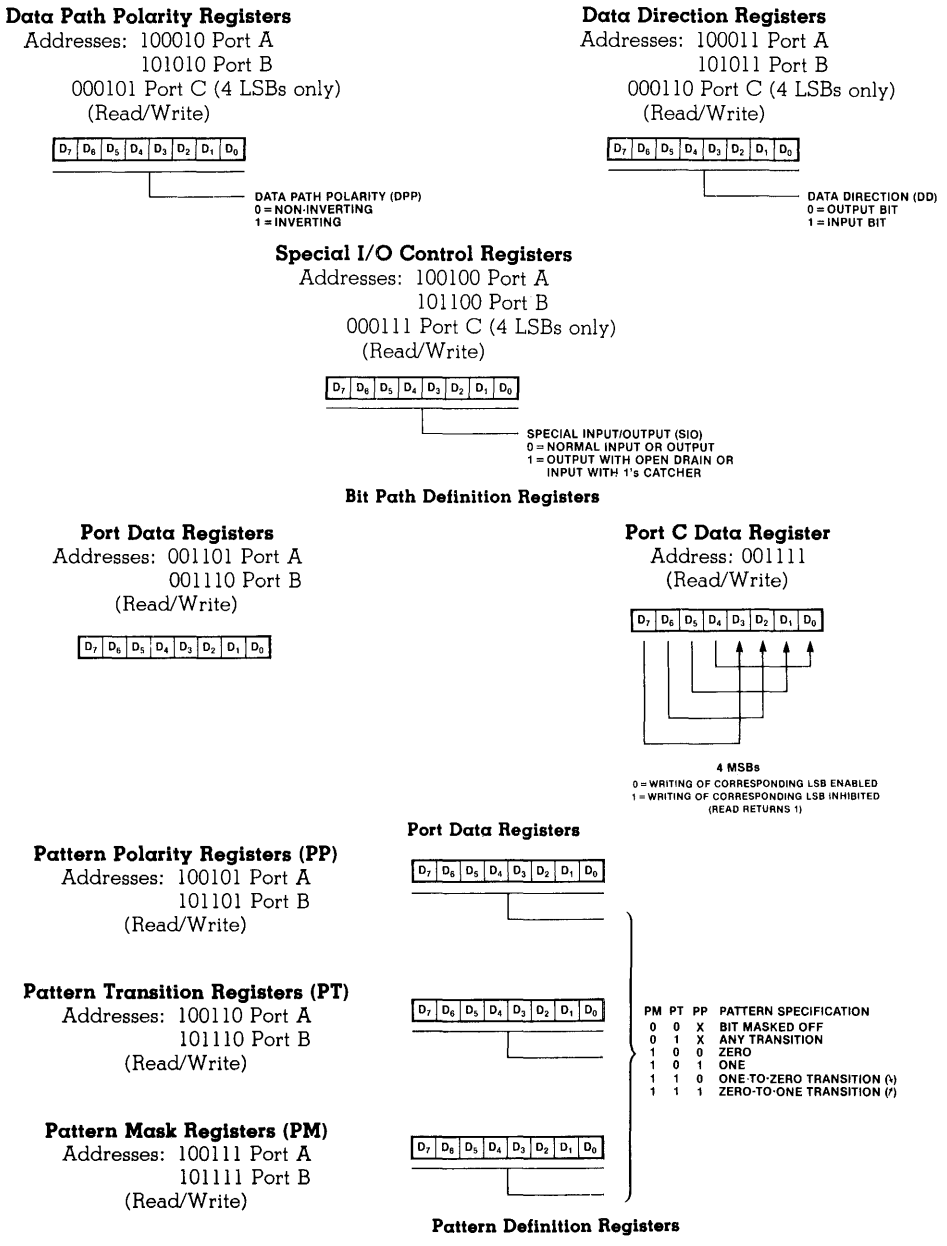
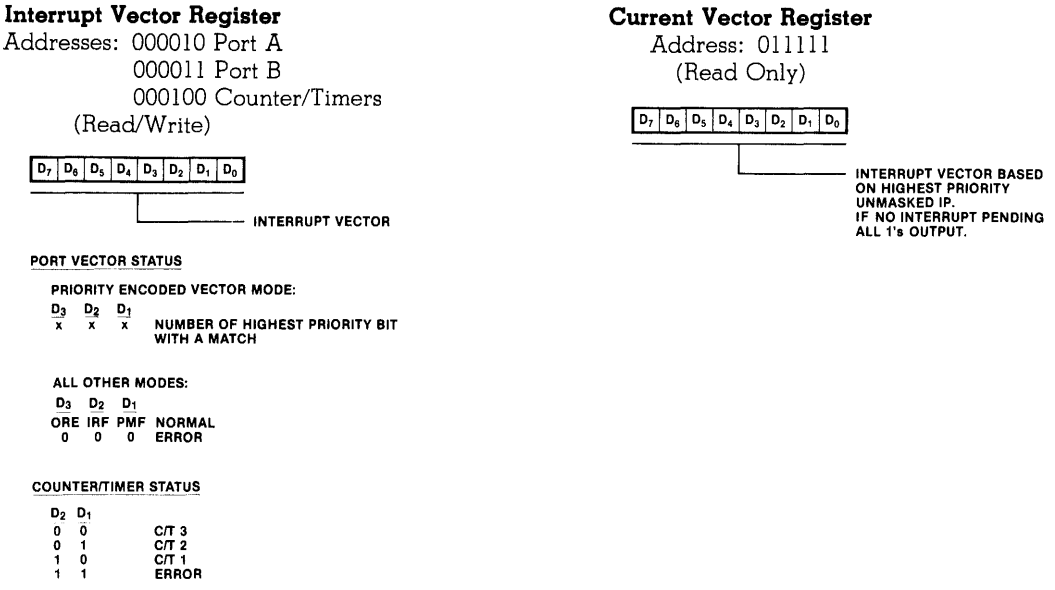


Figure 12.13 *Continued*



Interrupt Vector Registers

Figure 12.13 *Continued*

the other side (port 2 side) can be configured as a Z-Bus interface, nonmultiplexed bus interface, two-wire handshake I/O interface, or three-wire handshake I/O interface. Thus dissimilar CPUs or CPUs and peripherals running with different speeds or protocols can be linked, allowing asynchronous data transfers and improving I/O overhead. The FIO is a 40-pin device that requires a single +5-V supply and draws a maximum of 250 mA.

A block diagram of the FIO is given in Fig. 12.15. The port 1 side is always a processor interface and can be configured as a Z-Bus interface (connected to either the lower or upper half of the bus) or a nonmultiplexed bus interface. The nonmultiplexed bus is a general-purpose microprocessor interface with eight data lines, chip enable ($\overline{CE}$), read ($\overline{RD}$), write ($\overline{WR}$), and control/data (C/$\overline{D}$) signals. The timing for data transfers on this bus is illustrated in Fig. 12.16. The C/$\overline{D}$ signal determines if the current bus transfer involves a control register in the FIO or the FIFO data buffer itself. This bus configuration easily interfaces to microprocessors with separate, nonmultiplexed address and data buses, such as the Z80, 8080, and 6800. The configuration of the port 1 side is determined by the condition of two pins, M0 and M1 (Table 12.3). The port 2 side can be a Z-Bus, nonmultiplexed bus, two-wire handshake, or three-wire handshake interface, as determined by two bits (B0 and B1) in an internal register that is programmed from the port 1 side. Fig-

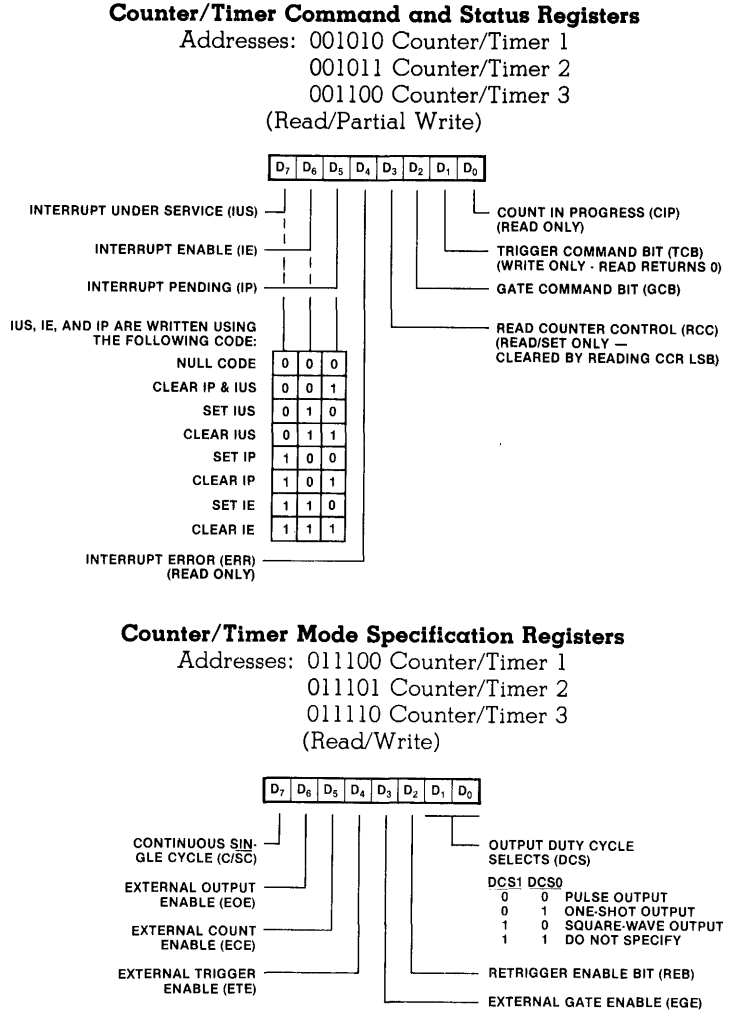


Figure 12.13 *Continued*

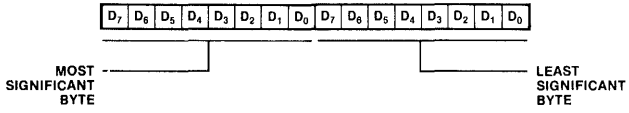
ure 12.17 shows the pin-out of the FIO and Table 12.4 describes the pin assignment for each possible type of interface.

Pattern recognition logic is included for both sides of the FIO and is capable of generating an interrupt when a specific data pattern is written to or read from the FIFO buffer. The pattern can be specified for each bit as a 1 or a 0; individual bits can be masked off, if so desired.

Special message registers, also called mailbox registers, can be used to pass information between CPUs if the FIO is used as a CPU-to-CPU interface.

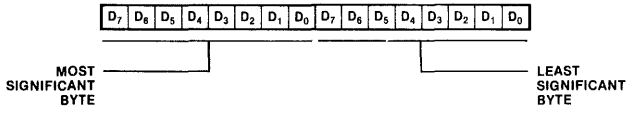
Counter/Timer Current Count Registers

Addresses: 010000 Counter/Timer 1's MSB
010001 Counter/Timer 1's LSB
010010 Counter/Timer 2's MSB
010011 Counter/Timer 2's LSB
010100 Counter/Timer 3's MSB
010101 Counter/Timer 3's LSB
(Read Only)



Counter/Timer Time Constant Registers

Addresses: 010110 Counter/Timer 1's MSB
010111 Counter/Timer 1's LSB
011000 Counter/Timer 2's MSB
011001 Counter/Timer 2's LSB
011010 Counter/Timer 3's MSB
011011 Counter/Timer 3's LSB
(Read/Write)



Counter/Timer Registers

Figure 12.13 *Continued*

TABLE 12.3 Z8038 FIO OPERATING MODES

Mode	M_1	M_0	B_1	B_0	Port 1	Port 2
0	0	0	0	0	Z-Bus low byte	Z-Bus low byte
1	0	0	0	1	Z-Bus low byte	Non-Z-Bus
2	0	0	1	0	Z-Bus low byte	Three-wire handshake
3	0	0	1	1	Z-Bus low byte	Two-wire handshake
4	0	1	0	0	Z-Bus high byte	Z-Bus high byte
5	0	1	0	1	Z-Bus high byte	Non-Z-Bus
6	0	1	1	0	Z-Bus high byte	Three-wire handshake
7	0	1	1	1	Z-Bus high byte	Two-wire handshake
8	1	0	0	0	Non-Z-Bus	Z-Bus low byte
9	1	0	0	1	Non-Z-Bus	Non-Z-Bus
10	1	0	1	0	Non-Z-Bus	Three-wire handshake
11	1	0	1	1	Non-Z-Bus	Two-wire handshake

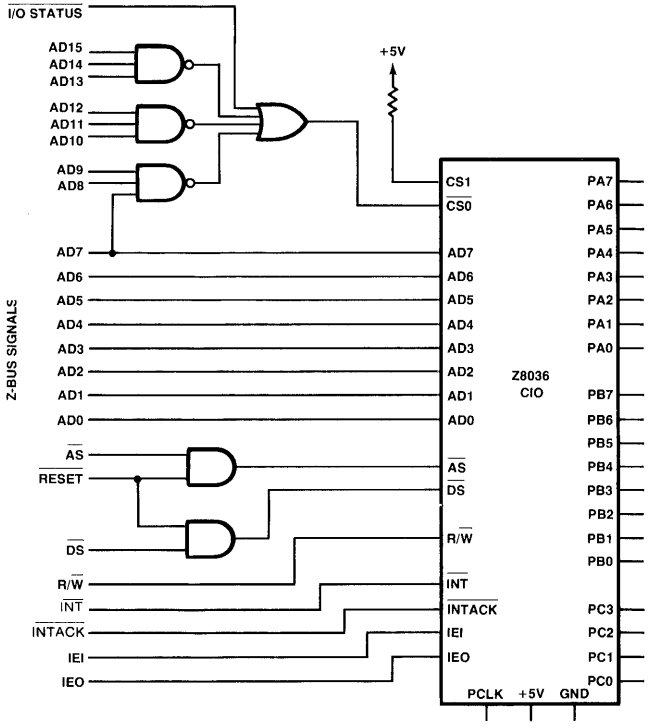


Figure 12.14 Typical Z-Bus to CIO interface.

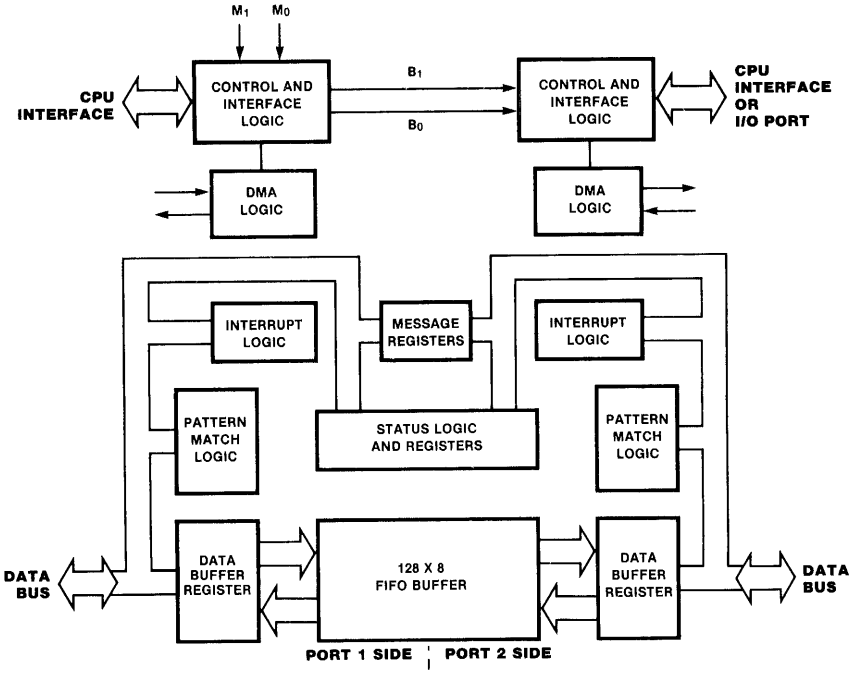
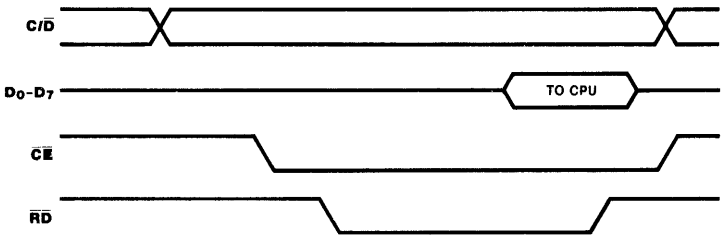
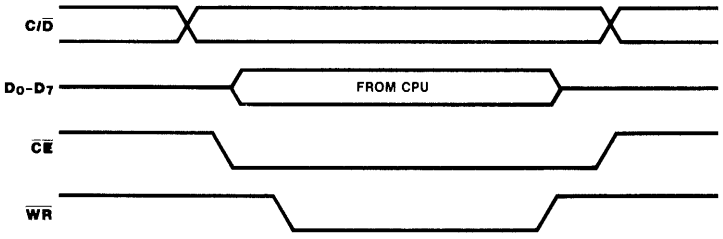


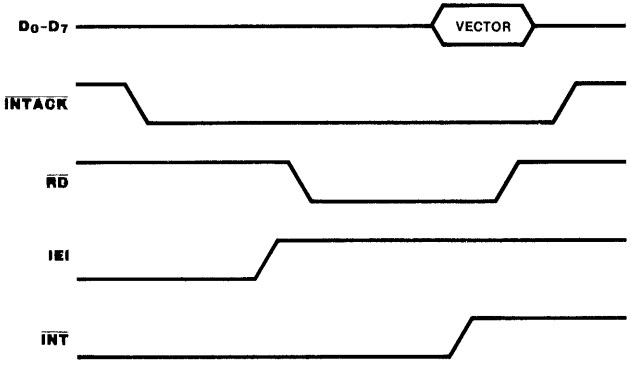
Figure 12.15 Z8038 FIO block diagram.



Non-Z-BUS Read Cycle Timing



Non-Z-BUS Write Cycle Timing

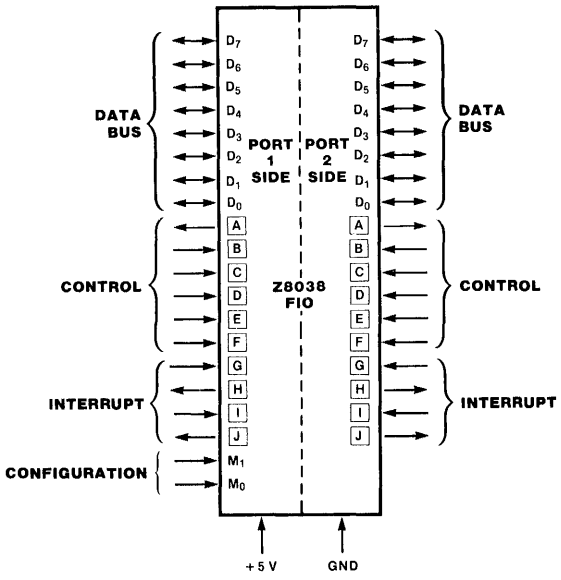


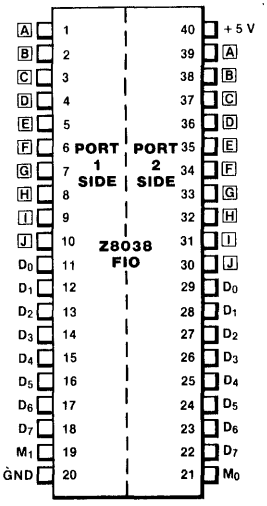
Non-Z-BUS Interrupt Acknowledge Cycle

Figure 12.16 Nonmultiplexed bus interface timing.

These mailbox registers allow control information to be transferred between the CPUs without using or affecting the FIFO buffer. The transmitting CPU can interrupt the receiving CPU by loading a byte into the mailbox register.

The data transfer logic of the FIO has been specially designed to work with DMA controllers for high-speed data transfers between the FIFO buffer





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

*2 side only.

Figure 12.17 Z8038 FIO pin assignments.

and the system's memory. A special control register, the Byte Count Comparison register, can be used to send a request to the DMA device when a given number of bytes is in the FIFO buffer. For the input side of the FIFO buffer, the request ($\overline{REQ}$) signal to the DMA controller becomes active when the number of bytes in the FIFO buffer is equal in value to the Byte Count Comparison register and stays active until the buffer is full (Fig. 12.18). For

TABLE 12.4 Z8038 FIO PIN FUNCTIONS

Pin signals	Pin names	Pin numbers		Mode	Signal description
		Pins Common to Both Sides			
M_0	M_0	21			M_1 and M_0 program port 1 side CPU interface
M_1	M_1	19			
+5 V dc	+5 V dc	40			Dc power source
GND	Gnd	20			Dc power ground
		Z-Bus Low-Byte Mode			
		Port 1	Port 2		
AD_0-AD_7 (address/data)	D_0-D_7	11-18	29-22		Multiplexed bidirectional address/data lines, Z-Bus compatible
$\overline{REQ}/\overline{WAIT}$ (request/wait)	A	1	39		Output, active low, REQUEST (ready) line for DMA transfer; $\overline{WAIT}$ line (open-drain) output for synchronized CPU and FIO data transfers
$\overline{DMASTB}$ (direct memory access strobe)	B	2	38		Input, active low; strobes DMA data to and from the FIFO buffer
$\overline{DS}$ (data strobe)	C	3	37		Input, active low; provides timing for data transfer to or from FIO
R/$\overline{W}$ (read/write)	D	4	36		Input: active high signals CPU read from FIO; active low signals CPU write to FIO
$\overline{CS}$ (chip select)	E	5	35		Input, active low; enables FIO; latched on the rising edge of $\overline{AS}$
$\overline{AS}$ (address strobe)	F	6	34		Input, active low; addresses, $\overline{CS}$ and $\overline{INTACK}$ sampled while $\overline{AS}$ low
$\overline{INTACK}$ (interrupt acknowledge)	G	7	33		Input, active low; acknowledges an interrupt; latched on the rising edge of $\overline{AS}$
IEO (interrupt enable out)	H	8	32		Output, active high; sends interrupt enable to lower priority device IEI pin
IEI (interrupt enable in)	I	9	31		Input, active high; receives interrupt enable from higher-priority-device IEO signal
$\overline{INT}$ (interrupt)	J	10	30		Output, open drain, active low; signals FIO interrupt request to CPU

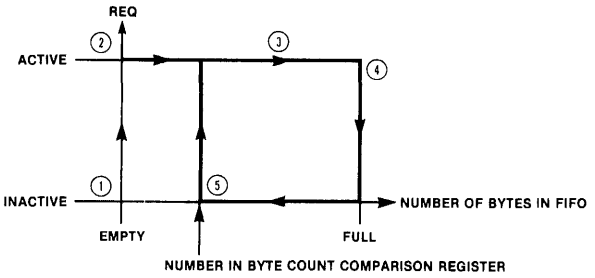
		Z-Bus High-Byte Mode		
AD_0-AD_7 (address/data)	D_0-D_7	11-18	29-22	Multiplexed bidirectional address/data lines, Z-Bus compatible
$\overline{REQ}/\overline{WAIT}$ (request/wait)	A	1	39	Output, active low, REQUEST (ready) line for DMA transfer; $\overline{WAIT}$ line (open-drain) output for synchronized CPU and FIO data transfers
$\overline{DMASTB}$ (direct memory access strobe)	B	2	38	Input, active low; strobes DMA data to and from the FIFO buffer
$\overline{DS}$ (data strobe)	C	3	37	Input, active low; provides timing for transfer of data to or from FIO
$R/\overline{W}$ (read/write)	D	4	36	Input active high; signals CPU read from FIO; active low signals CPU write to FIO
$\overline{CS}$ (chip select)	E	5	35	Input, active low; enables FIO; latched on the rising edge of $\overline{AS}$
$\overline{AS}$ (address strobe)	F	6	34	Input, active low; addresses $\overline{CS}$ and $\overline{INTACK}$ are sampled while $\overline{AS}$ is low
A_0 (address bit 0)	G	7	33	Input, active high; with A_1, A_2, and A_3, addresses FIO internal registers
A_1 (address bit 1)	H	8	32	Input, active high; with A_0, A_2, and A_3, addresses FIO internal registers
A_2 (address bit 2)	I	9	31	Input, active high; with A_0, A_1, and A_3, addresses FIO internal registers
A_3 (address Bit 3)	J	10	30	Input, active high; with A_0, A_1, and A_2, addresses FIO internal registers
		Non-Z-Bus Mode		
D_0-D_7 (data)	D_0-D_7	11-18	29-22	Bidirectional data bus
$\overline{REQ}/\overline{WT}$ (request/wait)	A	1	39	Output, active low, REQUEST (ready) line for DMA transfer; $\overline{WAIT}$ line (open-drain) output for synchronized CPU and FIO data transfer
$\overline{DACK}$ (DMA acknowledge)	B	2	38	Input, active low; DMA acknowledge
$\overline{RD}$ (read)	C	3	37	Input, active low; signals CPU read from FIO

TABLE 12.4 Z8038 FIO PIN FUNCTIONS *Continued*

Pin signals	Pin names	Pin numbers		Mode	Signal description
			Non-Z-Bus Mode *Continued*		
$\overline{WR}$ (write)	D	4	36		Input, active low; signals CPU write to FIO
$\overline{CE}$ (chip select)	E	5	35		Input, active low; used to select FIO
C/$\overline{D}$ (control/data)	F	6	34		Input, active high; identifies control byte on D_0-D_7; active low identifies data byte on D_0-D_7
$\overline{INTACK}$ (interrupt acknowledge)	G	7	33		Input, active low; acknowledges an interrupt
IEO (interrupt enable out)	H	8	32		Output, active high; sends interrupt enable to lower priority device IEI pin
IEI (interrupt enable in)	I	9	31		Input, active high; receives interrupt enable from higher-priority-device IEO signal
$\overline{INT}$ (interrupt)	J	10	30		Output, open drain, active low; signals FIO interrupt to CPU
			Port 2—I/O Port Mode		
D_0-D_7 (data)	D_0-D_7	29-22		Two-wire HS[a] Three-wire HS	Bidirectional data bus
RFD/$\overline{DAV}$ (ready for data/data available)	A	39		Two-wire HS Three-wire HS	Output, RFD active high; signals peripherals that FIO is ready to receive data; $\overline{DAV}$ active low signals that FIO is ready to send data to peripherals
$\overline{ACKIN}$ (acknowledge input)	B	38		Two-wire HS	Input, active low; signals FIO that output data are received by peripherals or that input data is valid
$\overline{DAV}$/DAC (data available/data accepted)	B	38		Three-wire HS	Input: $\overline{DAV}$ (active low) signals that data are valid on bus; DAC (active high) signals that output data are accepted by peripherals
FULL	C	37		Two-wire HS	Output, open drain, active high; signals that FIO buffer is full
DAC/RFD (data accepted/ready for data)	C	37		Three-wire HS	Direction controlled by internal programming; both active high: DAC (an output) signals that FIO has received data from peripheral;

				RFD (an input) signals that the listeners are ready for data
EMPTY	D	36	Two-wire HS Three-wire HS	Output, open drain, active high; signals that FIFO buffer is empty
$\overline{\text{CLEAR}}$	E	35	Two-wire HS Three-wire HS	Programmable input or output, active low; clears all data from FIFO buffer
DATA DIR (data direction)	F	34	Two-wire HS Three-wire HS	Programmable input or output: active high signals data input to Port 2; low signals data output from Port 2
IN_0	G	33	Two-wire HS Three-wire HS	Input line to D_0 of control register 3
OUT_1	H	32	Two-wire HS Three-wire HS	Output line from D_1 of control register 3
$\overline{\text{OE}}$ (output enable)	I	31	Two-wire HS Three-wire HS	Input, active low; when low, enables bus drivers; when high, floats bus drivers at high impedance
OUT_3	J	30	Two-wire HS Three-wire HS	Output line from D_3 of control register 3

[a]HS, handshake.

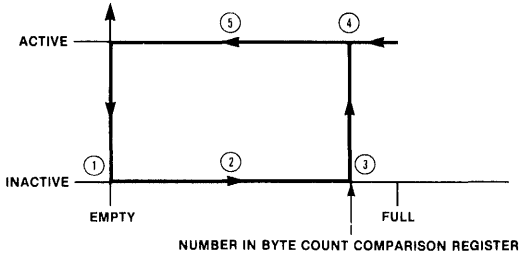


NOTES:
1. FIFO empty.
2. REQUEST enabled, FIO requests DMA transfer.
3. DMA transfers data into the FIO.
4. FIFO full, REQUEST inactive.
5. The FIFO empties from the opposite port until the number of bytes in the FIFO buffer is the same as the number programmed in the Byte Count Comparison register.

Figure 12.18 DMA-controlled writes to a FIO.

the output side of the FIO, the $\overline{REQ}$ pin is inactive until the number of bytes in the FIFO buffer equals the value in the Byte Count Comparison register. $\overline{REQ}$ then goes active and stays active until the buffer is empty (Fig. 12.19). A $\overline{WAIT}$ signal can be programmed to synchronize CPU-controlled block transfers.

Special control signals can be used to clear the FIFO buffer or change the direction of data flow in the buffer. The clear and data direction functions are controlled by the port 1 side as a default, but control of these functions can be passed to the port 2 side if desired. For CPU-to-CPU interfaces, if the controlling CPU changes the direction of the buffer, the other CPU is notified via an interrupt.

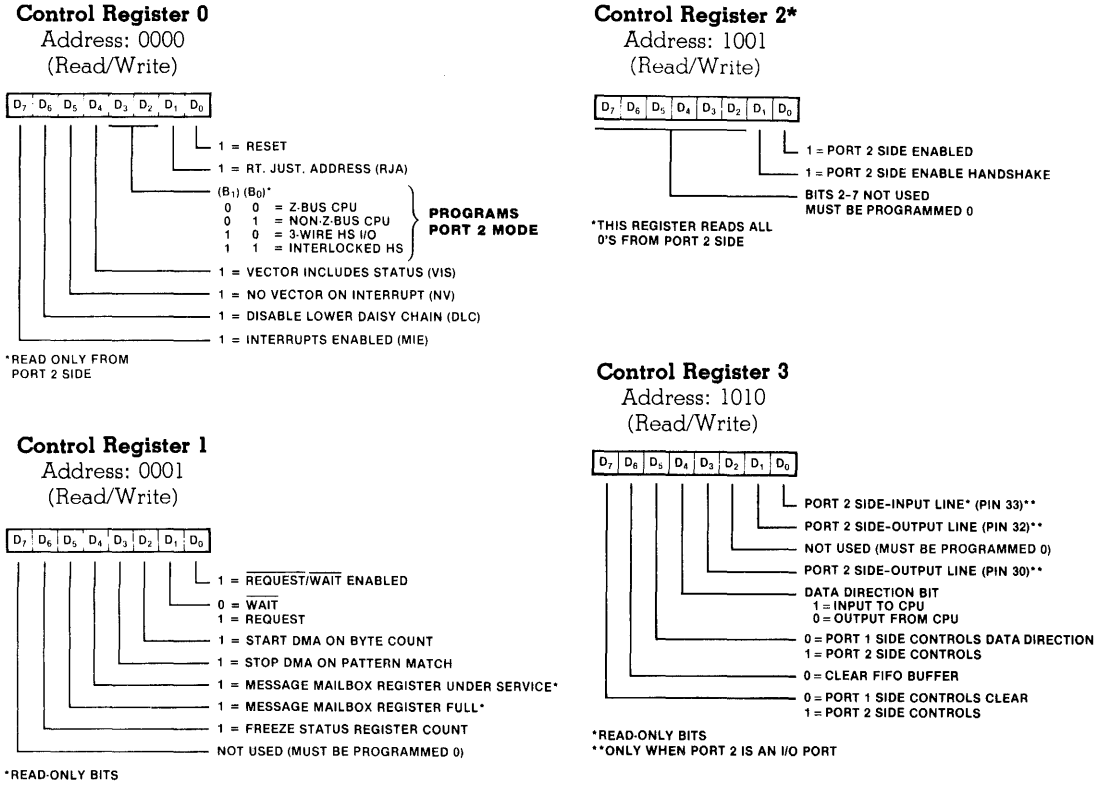


NOTES:
1. FIFO empty.
2. CPU/DMA fills FIFO buffer from the opposite port.
3. Number of bytes in FIFO buffer is the same as the number of bytes programmed in the Byte Count Comparison register.
4. REQUEST goes active.
5. DMA transfers data out of FIFO until it is empty.

Figure 12.19 DMA-controlled reads from an FIO.

Each side of the FIO has seven sources of interrupt. They are, in priority order, the mailbox register, change in data direction, pattern match, status match (number of bytes in the FIFO buffer equals the value in the Byte Count Comparison register), overflow/underflow error, buffer full, and buffer empty. Each interrupt source has its own IE, IP, and IUS bits for controlling that interrupt. Each side of the FIO has one interrupt vector; that vector can include encoded status information identifying the interrupt source.

Each side of the FIO has 16 addressable read/write registers. One of these, Control Register 2, is not used on the port 2 side. The RJA bit in Control Register 0 determines how the registers are addressed. When RJA = 0, address bus bits AD1-AD4 are used for register addressing; when RJA = 1, address bus bits AD0-AD3 are used. Figure 12.20 shows all the FIO registers.



Control Registers

Figure 12.20 Z8038 FIO registers.

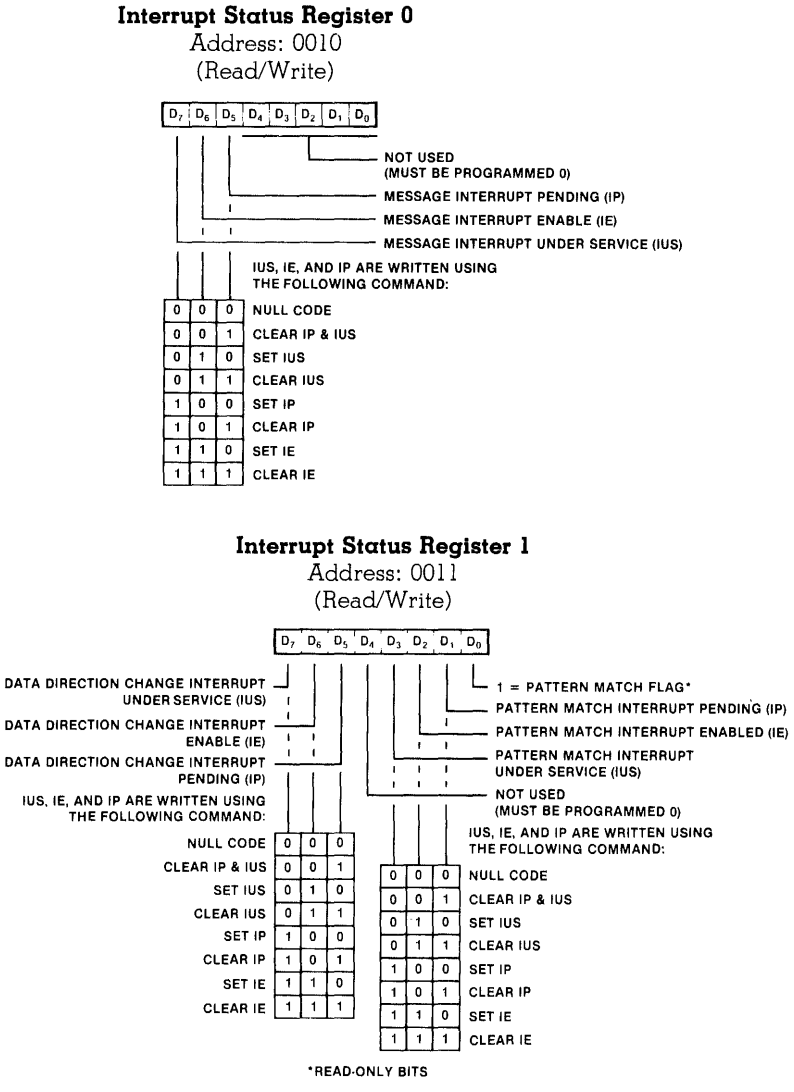
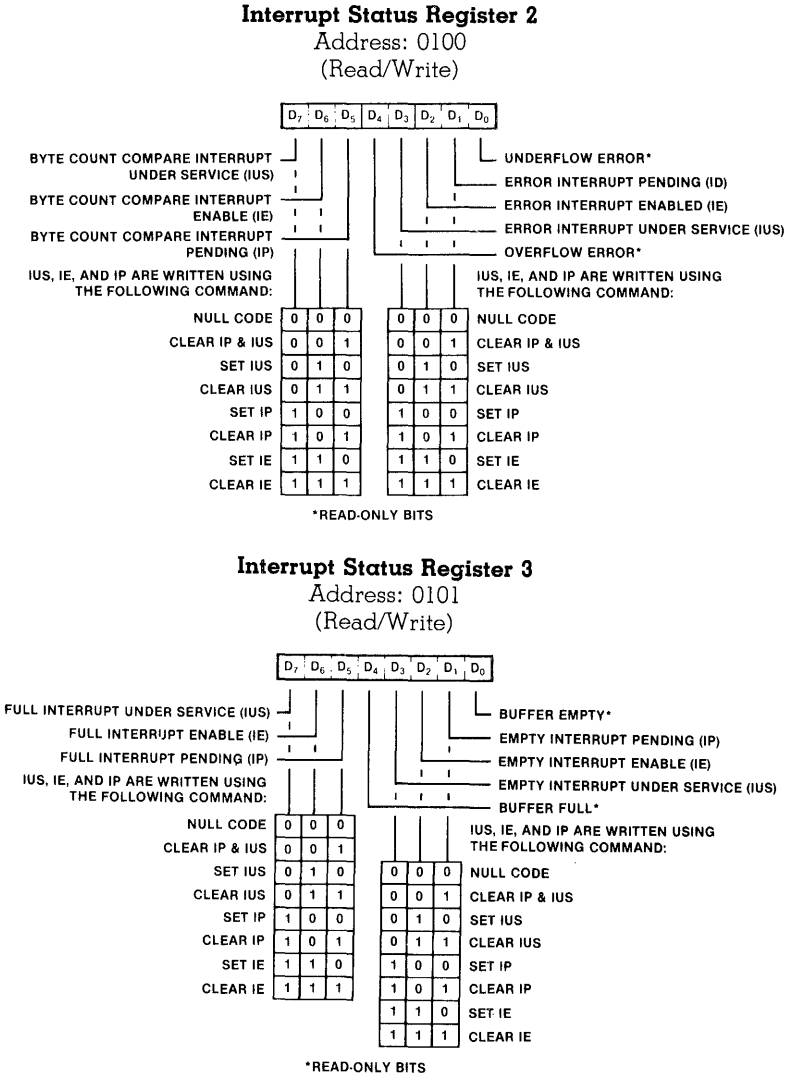


Figure 12.20 *Continued*

FIFO BUFFER EXPANSION

Both the depth and width of FIFO buffers in a system can be expanded easily with the Z8038 FIO and an auxiliary part, the Z8060 FIFO. The Z8060 FIFO is a 128 × 8 FIFO buffer with a two-wire interlocked handshake interface on both sides of the buffer (Fig. 12.21).

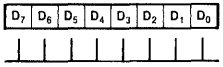


Interrupt Status Registers

Figure 12.20 *Continued*

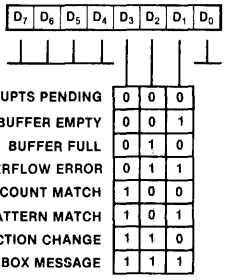
The buffer depth is expanded by cascading Z8038 FIOs and Z8060 FIFOs. Communication between these devices is via the two-wire interlocked handshake. For example, Fig. 12.22 illustrates a 512-byte CPU-to-CPU FIFO buffer interface consisting of two Z8038's and two Z8060's.

Byte Count Register
Address: 0111
(Read Only)

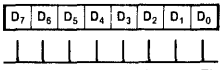


REFLECTS NUMBER OF BYTES IN BUFFER

Interrupt Vector Register
Address: 0110
(Read/Write)

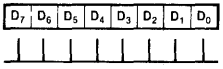


Pattern Match Register
Address: 1101
(Read/Write)



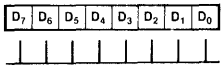
STORES BYTE COMPARED WITH BYTE IN DATA BUFFER REGISTER

Pattern Mask Register
Address: 1110
(Read/Write)



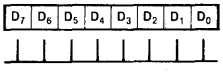
IF SET, BITS 0-7 MASK BITS 0-7 IN PATTERN MATCH REGISTER. MATCH OCCURS WHEN ALL NON-MASKED BITS AGREE.

Data Buffer Register
Address: 1111
(Read/Write)



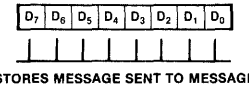
CONTAINS THE BYTE TRANSFERRED TO OR FROM FIFO BUFFER RAM

Byte Count Comparison Register
Address: 1000
(Read/Write)



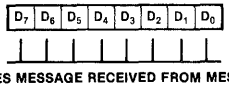
CONTAINS VALUE COMPARED TO BYTE COUNT REGISTER TO ISSUE INTERRUPTS ON MATCH (BIT 7 ALWAYS 0.)

Message Out Register
Address: 1011
(Read/Write)



STORES MESSAGE SENT TO MESSAGE IN REGISTER ON OPPOSITE PORT OF FIO

Message In Register
Address: 1100
(Read Only)



STORES MESSAGE RECEIVED FROM MESSAGE OUT REGISTER ON OPPOSITE PORT OF CPU

Figure 12.20 *Continued*

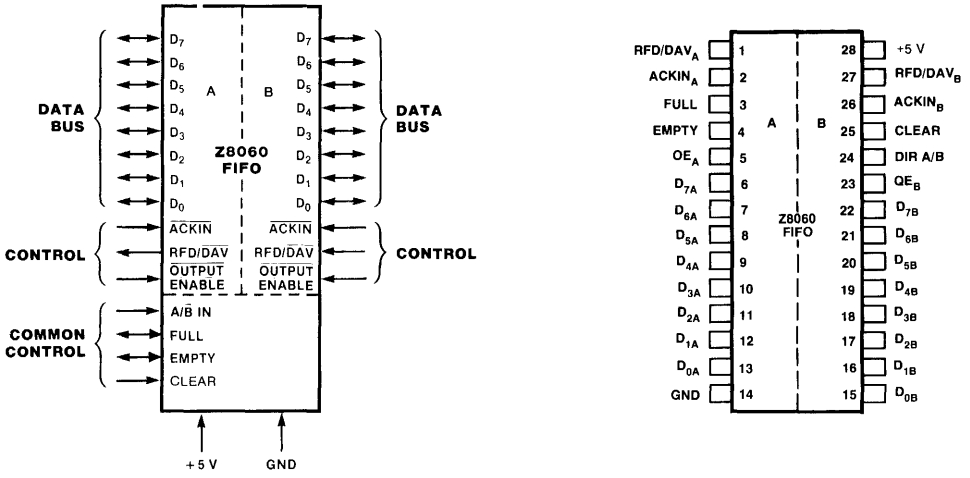


Figure 12.21 Z8060 FIFO pin assignments.

Buffer width is expanded using multiple Z8038 FIOs, as in Fig. 12.23. Two Z8038's are connected to a 16-bit microprocessor bus to implement a 16-bit wide FIFO buffer. If a CPU-to-peripheral word buffer is desired, some external logic will be necessary to synchronize the handshake signals from both Z8038's.

Z8030 SCC

The Z8030 Serial Communications Controller (SCC) is a dual-channel programmable data communications device that supports a wide variety of serial communication protocols. The SCC controls two independent full-duplex serial channels (called channel A and channel B) with data transfer rates up

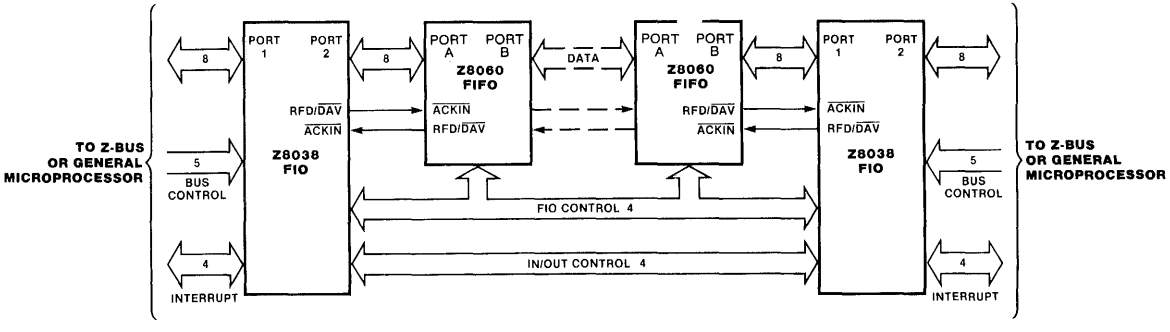


Figure 12.22 512-byte buffer using Z8038's and Z8060's.

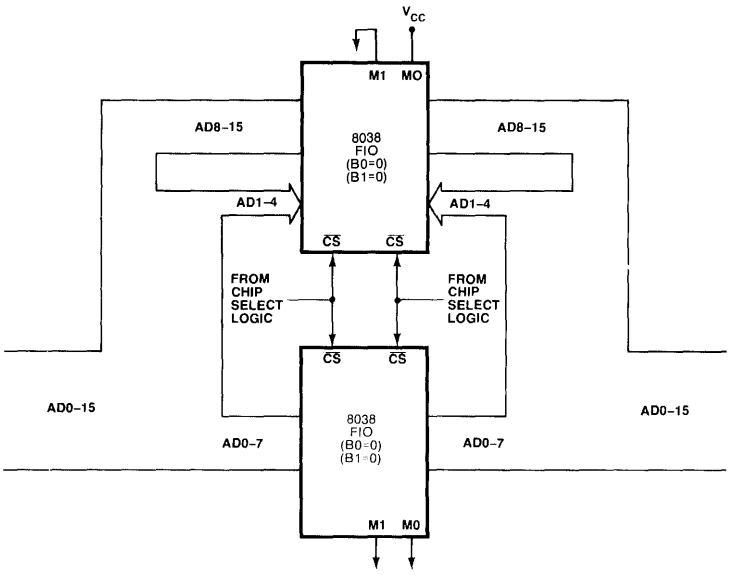


Figure 12.23 16-bit-wide buffer between two Z-Bus processors using Z8038's.

to 1 megabit/second. Each channel has its own crystal oscillator, baud rate generator, and digital phase-locked-loop circuitry for clock generation and recovery. Asynchronous, byte-oriented synchronous, and bit-oriented synchronous protocols are supported. Facilities are included for data integrity checking and modem controls. Thus the SCC is suitable for virtually any serial data communications application. The SCC is a 40-pin device that requires a single +5-V power supply and draws a maximum of 250 mA (Fig. 12.24 and 12.25).

When used for asynchronous communications, the SCC can be programmed for anywhere from 5 to 8 data bits per character (plus, optionally, a parity bit). The transmitter can supply one, one-and-a-half, or two stop bits per character and can provide a break output at any time. Automatic odd- or even-parity generation and checking can be specified. Framing and overrun errors are automatically detected. The transmit and receive clocks need not be symmetric, and data rates of 1, $\frac{1}{16}$, $\frac{1}{32}$, or $\frac{1}{64}$ of the clock rate are allowed.

For synchronous communications, both byte-oriented and bit-oriented protocols are supported. Cyclic Redundancy Codes (CRC) are used for error detection to assure data integrity. Both the CRC-16 ($x^{16} + x^{15} + x^2 + 1$) and CRC-CCITT ($x^{16} + x^{12} + x^5 + 1$) error-checking polynomials are supported. The CRC generator (for transmit operations) and CRC checker (for receive operations) may be preset to all 1's or all 0's.

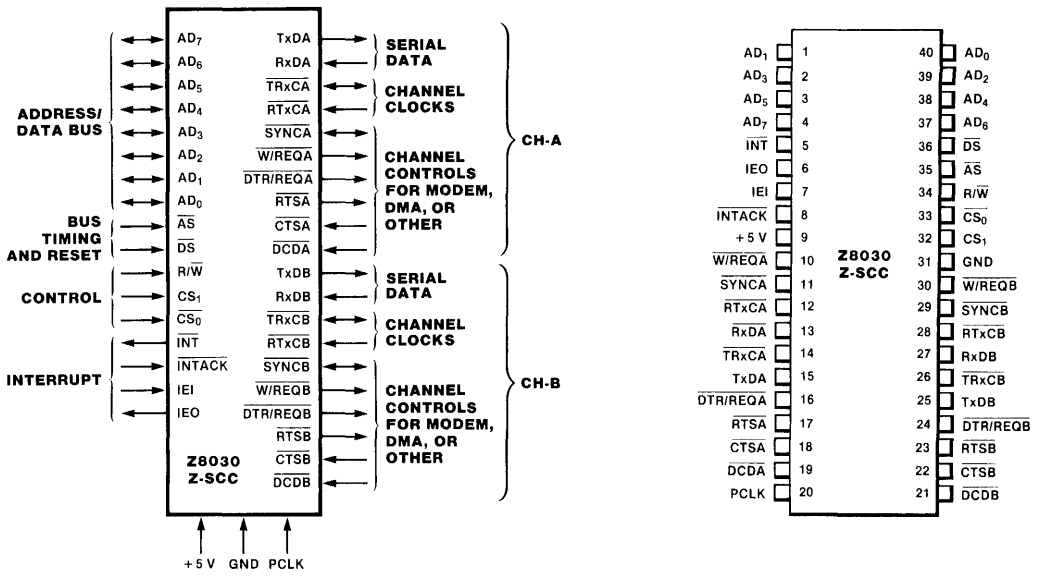


Figure 12.24 Z8030 SCC pin assignments.

Byte-oriented protocols, such as Monosync and Bisync, can have character synchronization with a 6-bit, 8-bit, 12-bit, or 16-bit synchronization pattern or an external synchronization signal. Leading synchronization characters are automatically deleted from the data stream without interrupting the CPU.

Bit-oriented protocols, such as SDLC and HDLC, are supported by features including automatic flag sending, automatic zero insertion and deletion, and abort sequence generation and checking. At the end of a message, the SCC automatically transmits the CRC code and trailing flags when the transmitter underruns. Address field recognition also is provided; the receiver can be programmed to respond for frames addresses by a byte, or 4 bits within a byte, or a user-selected address, or a global broadcast address. The number of address bytes can be extended under software control. At the end of a transmission, the status of the received frame is available in a status register. SDLC loop mode operations also are supported by the SCC; the SCC can perform the functions of a controller or secondary station in an SDLC loop.

The SCC has two special modes useful for system debugging called Auto Echo and Local Loopback. In Auto Echo mode, received data are automatically routed to the transmit data line; thus the SCC continuously transmits what it receives. In Local Loopback mode, transmitted data are automatically routed to the receive data line; thus the SCC reads the data it is transmitting.

Data may be encoded in any of four ways using the SCC: FM0 (biphase space), FM1 (biphase mark), NRZ, or NRZI encoding (see Fig. 12.26). For

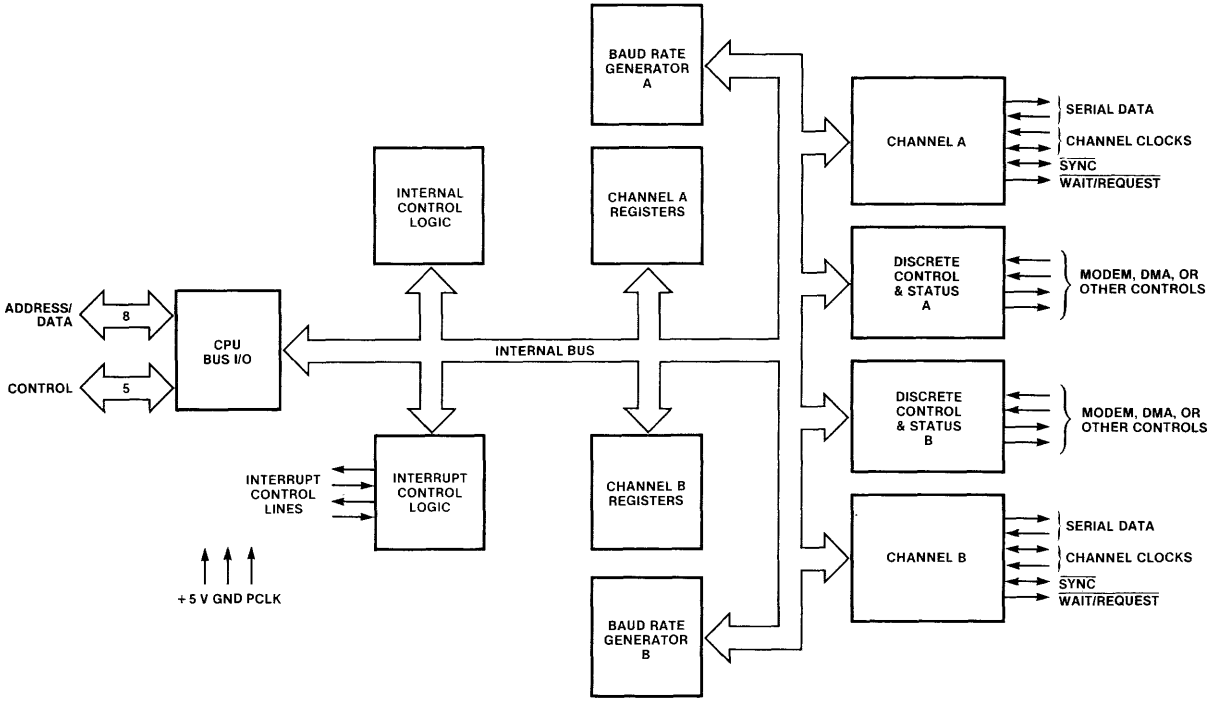


Figure 12.25 Z8030 SCC block diagram.

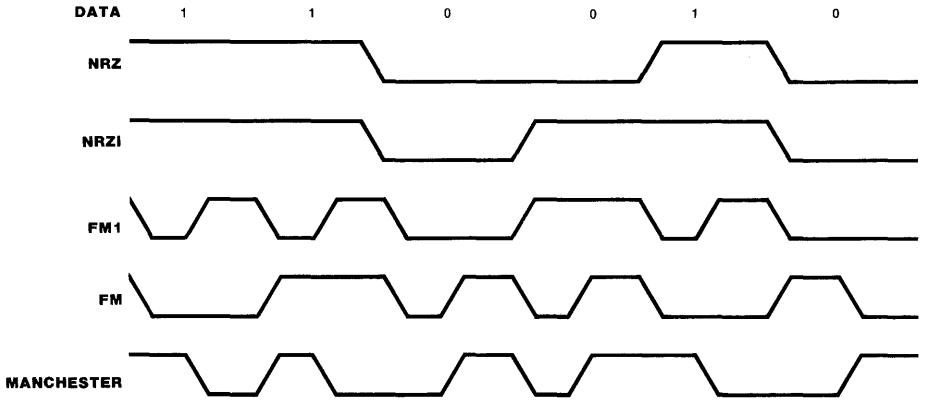


Figure 12.26 Data encoding in the SCC.

FM0 and FM1 encoding, a transition occurs at the beginning of every bit cell. In FM0 encoding, a 0 is represented by another transition at the center of the cell and a 1 is represented by no additional transfers in the cell. FM1 encoding is the inverse; a 1 is represented by another transition at the center of the cell and a 0 is represented by no further transitions in the cell. For NRZ encoding, a 1 is represented by a high level and a 0 by a low level. In NRZI encoding, a 1 is represented by no change in level and a 0 by a change in level. In addition, the SCC can decode Manchester (biphase level) encoded data. Manchester data encoding always produces a transition in the center of the cell. If the transition is from 0 to 1, the bit is a 0; if the transition is from 1 to 0, the bit is a 1.

Each channel has its own baud rate generator consisting of a 16-bit programmable down counter. The output of the baud rate generator can be used as a transmit clock, receive clock, or the input to the phase-locked-loop circuit. The digital phase-locked-loop circuitry can be used to recover clock information from NRZI, FM0, or FM1 encoded data. This clock can then be used as a transmit or receive clock.

Each SCC channel has three sources of interrupts: receive interrupts, transmit interrupts, and external/status condition interrupts (in that priority order). Channel A interrupts have priority over channel B interrupts. Each interrupt has its own IP, IUS, and IE control bits. One interrupt vector is provided; that vector can be encoded with status information to identify the interrupt source. A receiver can interrupt the CPU in three ways: interrupt on the first received character or special receive conditions, interrupt on all received characters or special receive conditions, or interrupt only on special receive conditions. The special receive conditions include overrun, parity, and framing errors and the end-of-frame condition (SDLC mode). The trans-

mit interrupt is activated when the transmit buffer becomes empty. External/status interrupts are caused by active levels on the $\overline{CTS}$, $\overline{DCD}$, or $\overline{SYNC}$ pins, transmit underruns, a break condition (asynchronous modes), an abort sequence (SDLC mode), an end-of-poll sequence (SDLC loop mode), or a zero count in the baud rate generator. Support is provided for DMA or CPU-controlled high-speed data transfers.

Receive data are routed through a 3-byte FIFO buffer, providing additional time for the CPU to service a receive interrupt. For each character received, status information indicating if an error was detected while receiving that character is available; this status information is stored in its own 3-byte FIFO. The transmit buffer is 20 bits long.

The SCC contains 14 write registers and nine read registers per channel. Two other write registers are shared by both channels. The register configurations are given in Fig. 12.27.

Z8065 BEP

The Z8065 Burst Error Processor (BEP) provides error correction and detection for applications involving high-speed data transfers, such as high-performance disk systems. The BEP can detect errors in data streams up to 585K bits long and at data rates up to 20 megabits/second. The pin assignments for the BEP are given in Fig. 12.28.

The BEP uses any one of four different cyclic redundancy codes, called Fire codes, to detect and locate errors (Table 12.5). Three different operations can be performed: writing data, reading data, and correcting data. During writes, the BEP computes a check word by dividing the data stream by the selected polynomial; the remainder is a check code that is appended to the data stream. When reading, the stream of data and check bytes is divided by the polynomial to get a syndrome. If the syndrome is not 0, an error is detected. Two read modes are provided, normal and high speed; the read mode determines the correction methodology if an error is found. For all but the 48-bit polynomial, an error in the data stream can be located using one of two methods, the "full-period clock around" method (normal reads) or the "Chinese remainder theorem" method (high-speed reads). The "reciprocal poly-

TABLE 12.5 POLYNOMIALS SUPPORTED BY THE Z8065 BEP

Polynomial	Number of check bits	Maximum data length (bits)	Correctable burst error length
56-bit	56	585,442	11
32-bit	32	42,987	11
35-bit	35	94,185	12
48-bit	48	$13 \times (2^{35} - 1)$	7

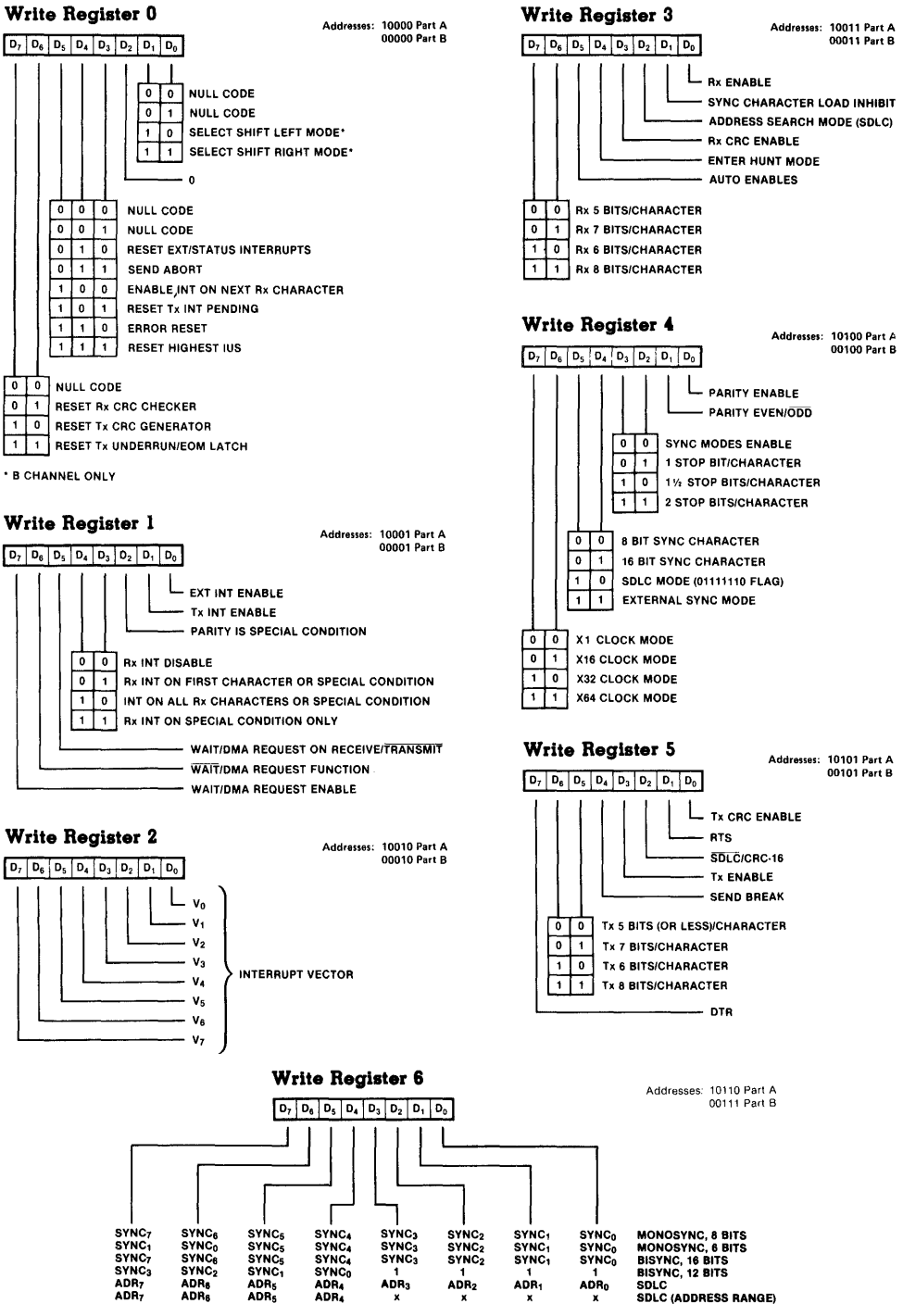
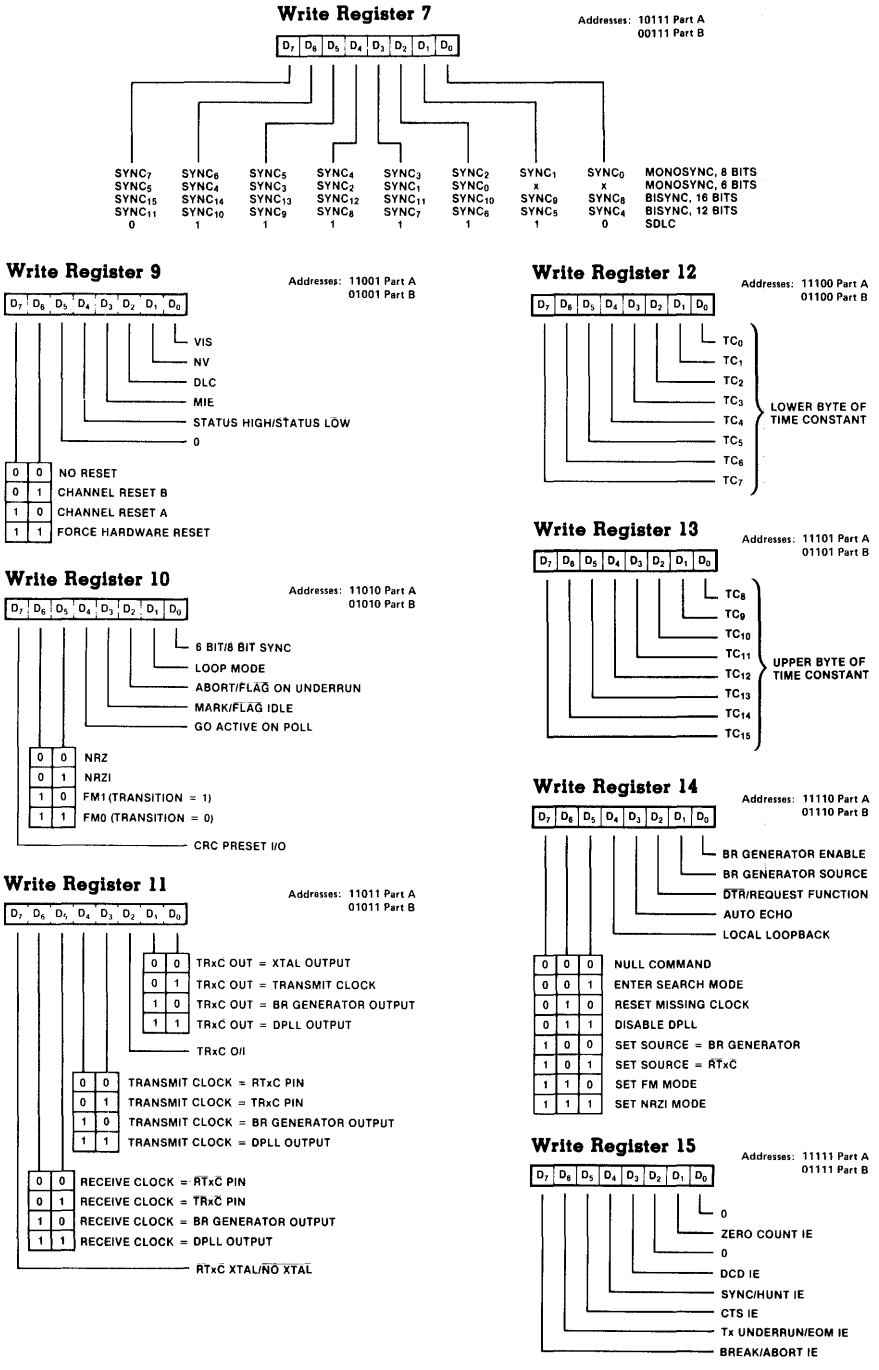


Figure 12.27 Z8030 SCC registers.



NOTE: WRITE/READ REGISTER 8 IS THE DATA REGISTER.

Figure 12.27 *Continued*

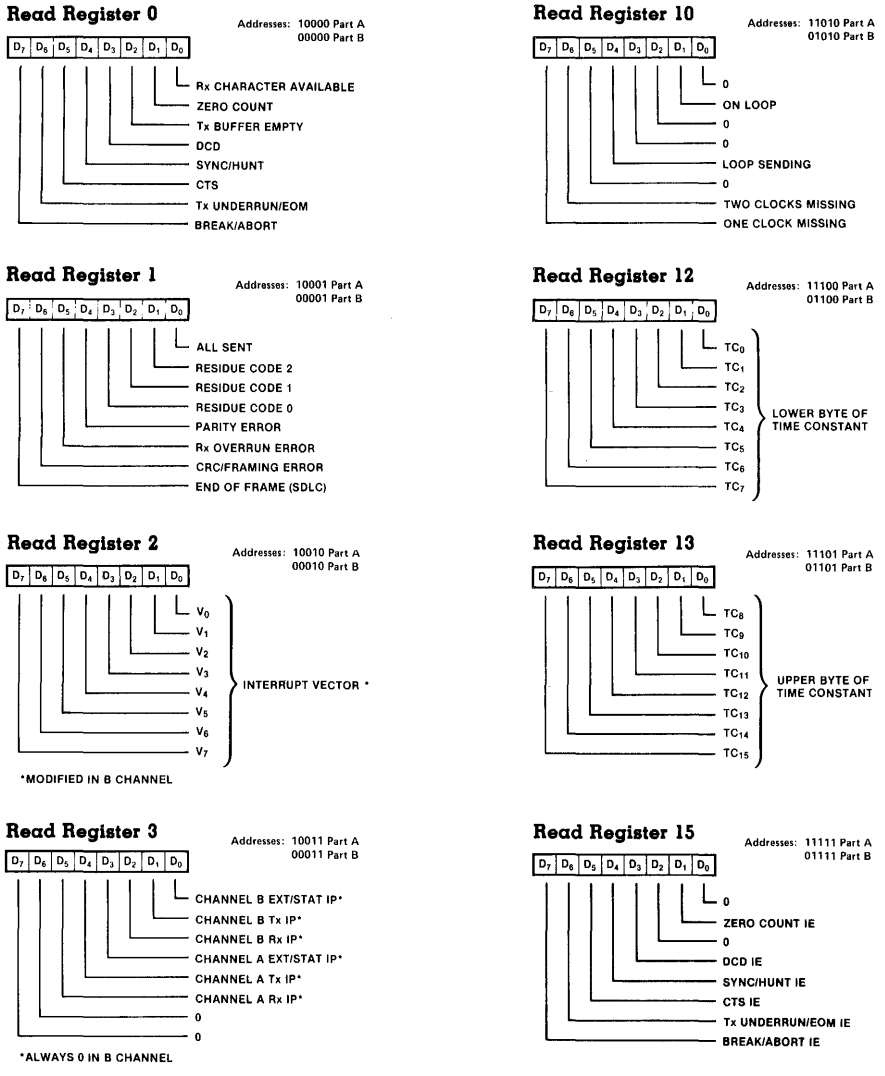


Figure 12.27 *Continued*

nomial" error-correction method is used with the 48-bit Fire code. These correction algorithms extract the error pattern in the data stream for external correction.

The major sections of the BEP are illustrated in Fig. 12.29. Data are input to the BEP one byte at a time and divided by the appropriate polynomial in the Polynomial Divide Matrix. The Register Array contains 56 flip-flops used

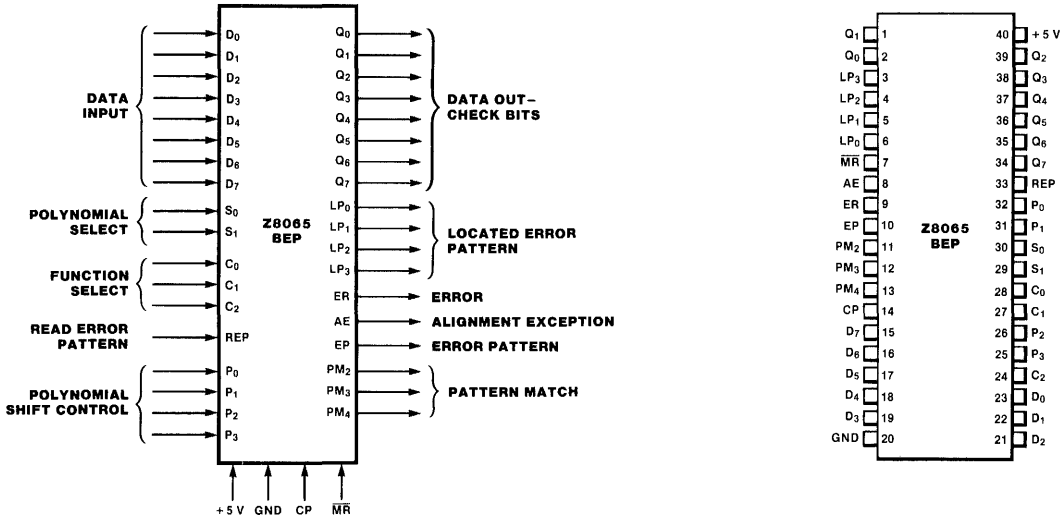


Figure 12.28 Z8065 BEP pin assignments.

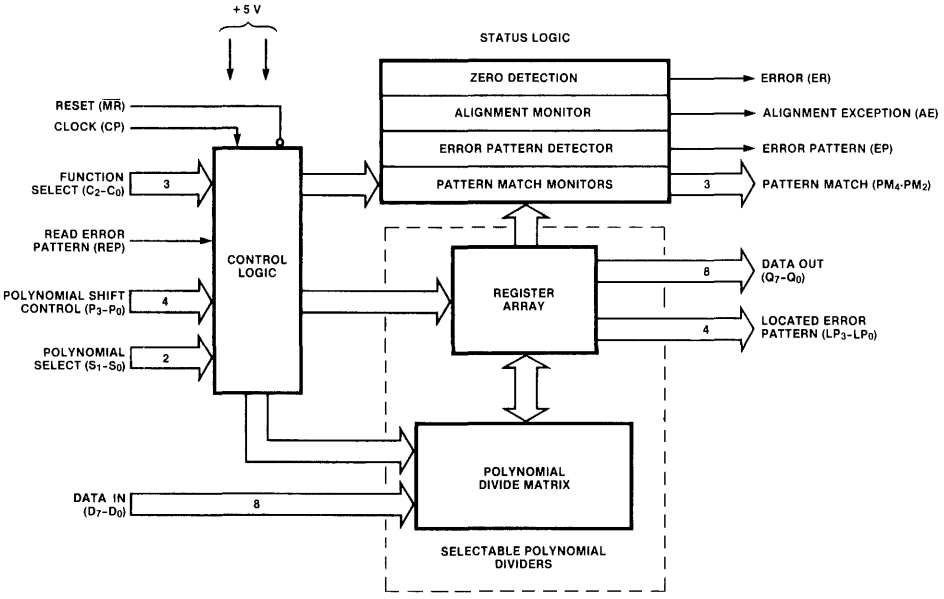


Figure 12.29 Z8065 BEP block diagram.

for check bit computation and error pattern extraction. The Control Logic contains the timing, gating, and reset circuitry for the device.

Z8068 DCP

The Z8068 Data Ciphering Processor (DCP) is a data encryption/decryption device that conforms to the National Bureau of Standards Data Encryption Standard (Federal Information Processing Standards Publication 46). Data rates up to 1 megabyte/second can be obtained. The pin assignments for this 40-pin device are shown in Fig. 12.30.

The DCP provides three ciphering options: Electronic Code Book, Chain Block Cipher, and Cipher Feedback. Electronic Code Book is a relatively straightforward cipher used for disk systems and similar applications. Chain Block Cipher encryption involves a feedback step wherein the ciphering of a data block is dependent on the previous data block and is commonly used in high-speed telecommunications applications. Cipher Feedback encryption involves both a feedback path and a pseudo-random binary stream that is exclusive-ORed to the text to be encrypted; it is used in low-speed byte-oriented applications.

Three separate 8-bit ports can be used for the cypher key, clear data, and encrypted data (Fig. 12.31). The DCP can be used as an encrypting or decrypt-

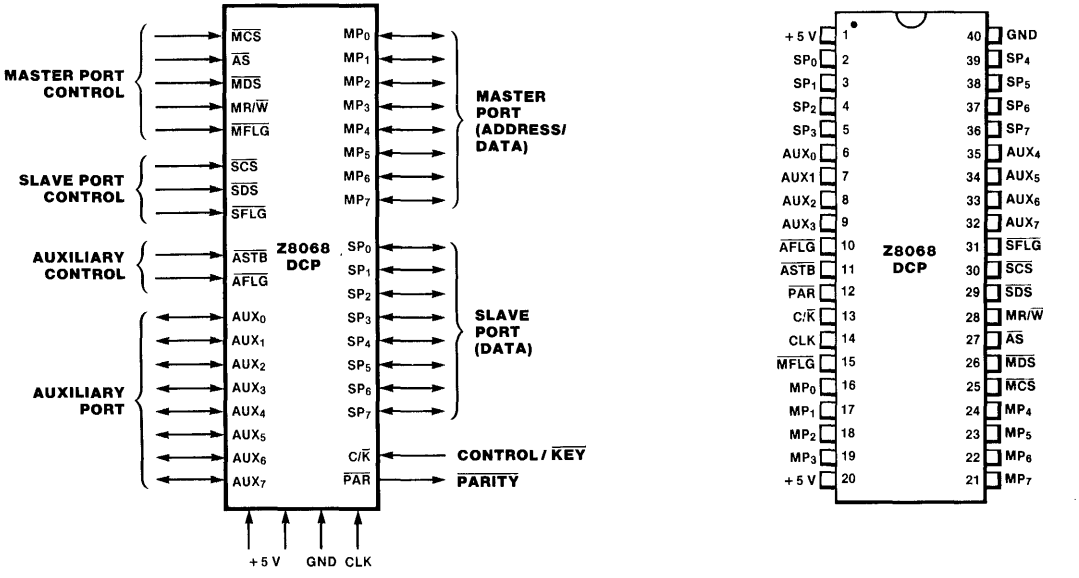


Figure 12.30 Z8068 DCP pin assignments.

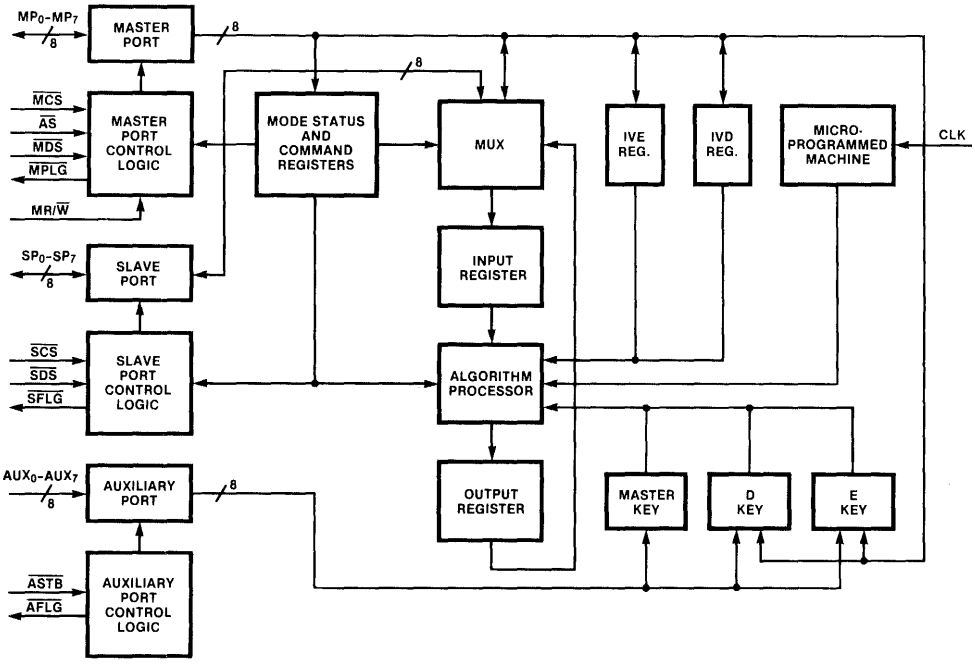


Figure 12.31 Z8068 DCP block diagram.

ing device in a single-port configuration (the master port is used for both clear and encrypted data) or a dual-port configuration (the master port is used for clear data and the slave port for encrypted data, or vice versa). Input, output, and ciphering of data are performed concurrently, thereby maximizing data throughput. The input and output registers each hold 8 bytes and data are encrypted or decrypted in 64-bit blocks.

Z8052 CRTC

The Z8052 CRT Controller (CRTC) is a general-purpose controller for raster-scan CRT displays. The CRTC is a word peripheral with an on-board DMA controller capable of addressing up to 64K bytes of the system's memory. The CRTC is a 48-pin device that operates from a single 5-V supply.

Designed to interface the Z8000 to a wide variety of CRT displays, the Z8052 includes numerous advanced features such as vertical and horizontal split-screen capability, multiple cursors, blinking cursors or characters with programmable blink rates, character vertical shifting (subscripts and superscripts), variable number of scan lines per row, and variable row lengths. Fifteen character attributes are specified on a character-by-character basis; char-

acters and their attributes are stored in a 132 × 22 buffer. Simple line graphics also can be implemented with the line attributes provided. The CRTC can be operated in a slave mode that allows expansion of the character buffer using multiple CRTCs.

Z8016 DTC

The Z8016 Direct Memory Access Transfer Controller (DTC) is a high-speed dual-channel DMA controller that interfaces to the Z-Bus as both a bus requestor and a programmable peripheral device. Each of the DTC's two channels can transfer data blocks between memory and a peripheral, two memory areas, or two peripherals. Memory-to-memory transfer rates up to 2 megabytes/second and memory-to-peripheral or peripheral-to-peripheral transfer rates up to 1.3 megabytes/second are possible. The DTC is housed in a 48-pin package and requires a single 5-V power supply (Fig. 12.32).

The segment number lines, address/data bus, and associated status and control signals are bidirectional at the DTC. These signals are inputs to the DTC when the CPU has control of the bus so that the DTC can be pro-

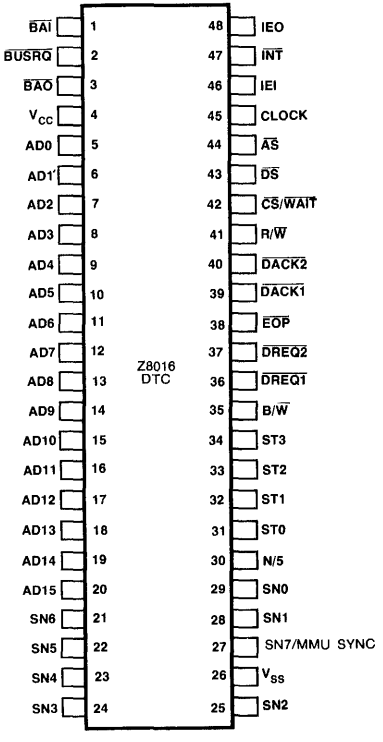


Figure 12.32 Z8016 DTC pin assignments.

grammed via I/O commands from the CPU; these signals are outputs when the DTC has control of the bus so that the DTC can control data transfers on the bus. The $\overline{BAI}$, $\overline{BAO}$, and $\overline{BUSRQ}$ signals are used to interface the DTC to a bus request daisy chain, as described in Chapter 7. IEI, IEO, and $\overline{INT}$ signals are available for attaching to an interrupt daisy chain; interrupt acknowledges are decoded internally from the ST0-ST3 inputs. The DMA request signals ($\overline{DREQ1}$ and $\overline{DREQ2}$, one for each channel) are inputs that can be used to initiate a DMA operation with an external signal. The DMA acknowledge signals ($\overline{DACK1}$ and $\overline{DACK2}$) indicate when a channel is performing a DMA operation. The end-of-process ($\overline{EOP}$) signal is a bidirectional line that the DTC uses to signal the end of a DMA operation. An external device (such as the MMU's $\overline{SUP}$ signal) also can terminate a DMA operation by pulling the $\overline{EOP}$ line low. The $\overline{CS}/\overline{WAIT}$ pin is used as a chip select when the DTC does not control the bus for sending commands to the DTC. When the DTC does control the bus, this pin is a $\overline{WAIT}$ input for controlling the timing of memory and I/O accesses initiated by the DTC.

In a system containing Z8010 MMUs, the Z8016 DTC can be used in two different manners. The DTC can be connected directly to the memory control logic and deal with physical memory addresses (Fig. 12.33) or the DTC can be interfaced to memory through the MMUs and use logical memory addresses (Fig. 12.34). For logical addresses, the DTC uses the SN0-SN6 and AD0-AD15 lines for the 23-bit logical address; the MMU Sync signal is sent

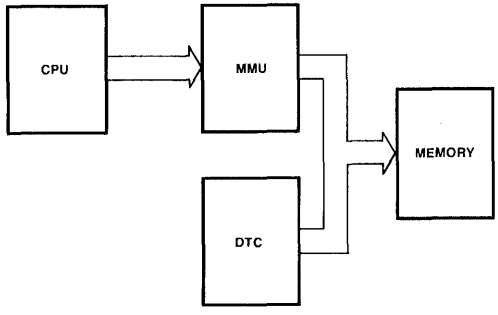


Figure 12.33 System with the DTC using physical memory addresses.

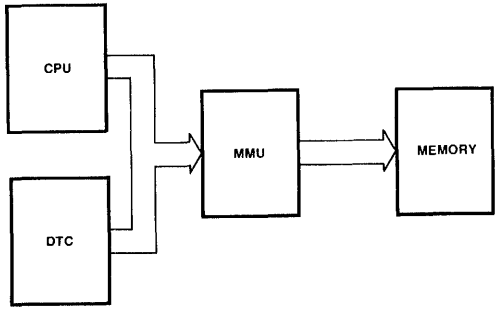


Figure 12.34 System with the DTC using logical memory addresses.

to the MMU's DMA Sync input to allow the MMU to differentiate between CPU- and DTC-controlled memory accesses. For physical addresses, the AD0-AD15 lines hold the 16 least significant bits and the SN0-SN7 lines hold the 8 most significant bits of the physical memory address.

DTC-controlled data transfers can be byte or word oriented. The DTC can be programmed to perform byte/word funneling for transfers between byte or word peripherals and/or memory. For transfers between a byte source and a word destination, two bytes are read in consecutive accesses from the source and sent as a word to the destination. For transfers between a word source and a byte destination, the word is read from the source and sent to the destination with two consecutive byte writes.

Pattern match capability is included in the Z8016, allowing search and transfer-and-search operations. Search operations read data from the source until a match to the specified pattern is found. Transfer-and-search operations transfer data between a source and destination until the specified data pattern is encountered.

The DTC provides for wait states during transactions with slow memories or peripherals under both hardware and software control. In addition to the hardware $\overline{\text{WAIT}}$ signal, the DTC can be programmed to automatically insert zero, one, two, or four wait states when accessing a particular source or destination device.

The Z8016 contains 20 status and control registers in each DMA channel. Three additional registers are used to control the overall operation of the device. To minimize CPU overhead, the DTC can be programmed to load many of its own registers from memory via DMA operations; the CPU only has to load the memory address of the control parameter table and issue a command to start this operation. This operation also can be performed at the end of some other DMA process, allowing the DTC to automatically chain its own operations without CPU intervention. Alternatively, the DTC can be programmed to interrupt the CPU at the termination of a DMA operation.

Z6132 RAM

The Z6132 Quasi-Static Random-Access Memory is a Z-Bus-compatible 4K × 8 dynamic RAM with on-board self-refresh capability. External refresh circuitry is not needed; thus the Z6132 combines the convenience of a static RAM with the density and low-power consumption of a dynamic RAM. The RAM is packaged as a 28-pin device and is pin-compatible with 2716/2732-type EPROMs (Fig. 12.35).

The Z6132 is organized as two separate blocks of memory with independent row address buffers and decoders but common column address decoders and data lines (Fig. 12.36). The row address decoders are addressed by either the A1-A7 address lines or an internal 7-bit refresh counter. During memory

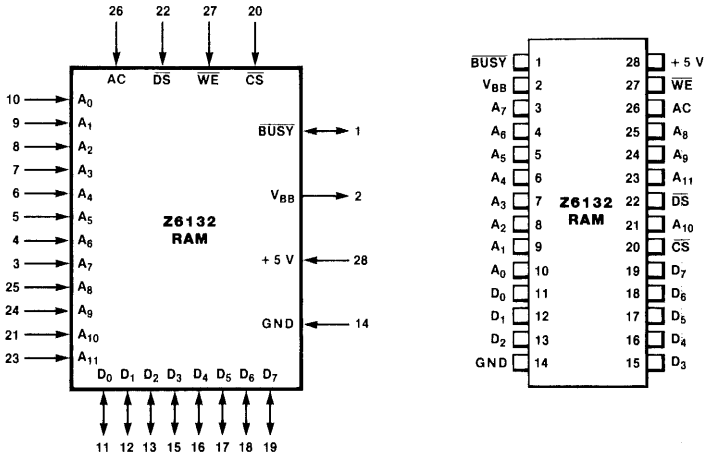


Figure 12.35 Z6132 Quasi-Static RAM pin assignments.

accesses, address input A0 selects one of the two blocks; meanwhile, the other block is refreshed using the refresh counter.

The timing of Z6132 memory accesses is described briefly in Chapter 3. A memory cycle starts on the rising edge of address clock (AC); this edge

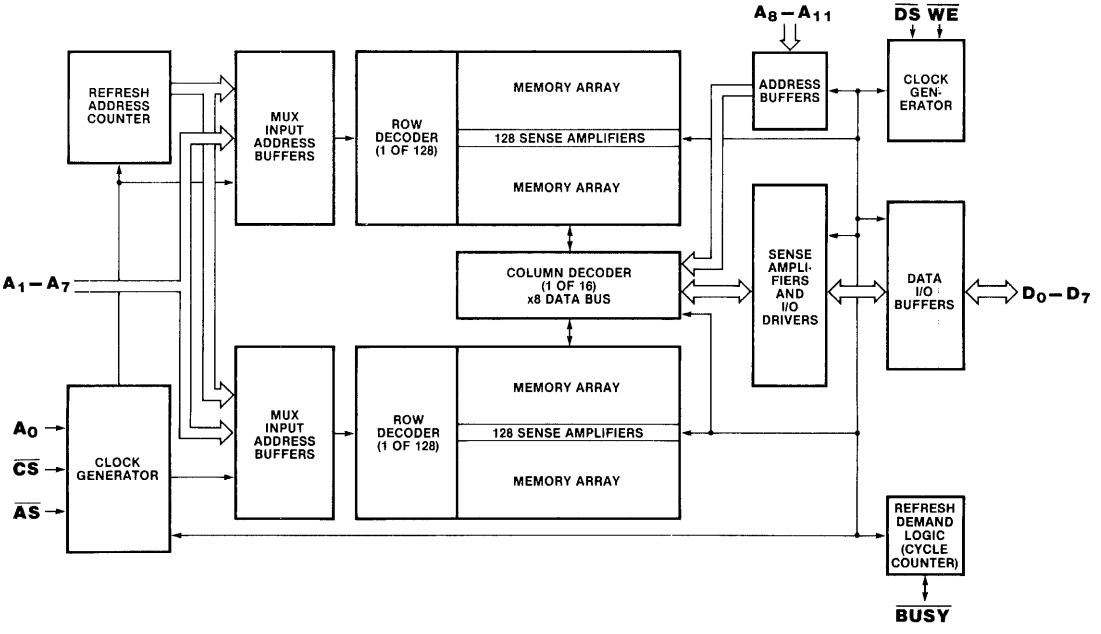


Figure 12.36 Z6132 RAM block diagram.

latches the chip select ($\overline{CS}$), write enable ($\overline{WE}$), and A0 signals. If the chip is not selected ($\overline{CS}$ high), all other inputs are ignored for the remainder of the cycle and both memory blocks in the chip are refreshed using the 7-bit refresh counter. If the chip is selected ($\overline{CS}$ low), the A0-A11 inputs are latched internally, where A0 determines the block addresses by A1-A11. If $\overline{WE}$ is high, indicating a read cycle, a subseqent low level on the data strobe ($\overline{DS}$) input activates the D0-D7 data outputs after a specified delay from the rising edge of AC or falling edge of $\overline{DS}$, whichever comes later. Thus $\overline{DS}$ is used as an output enable during read operations. If $\overline{WE}$ is low, indicating a write cycle, the falling edge of $\overline{DS}$ loads the data on the D0-D7 inputs into the addressed memory location (Figure 12.37).

Every dynamic memory cell in the Z6132 must be refreshed at least every 2 ms. Each of the two memory blocks contains 16,384 cells and requires

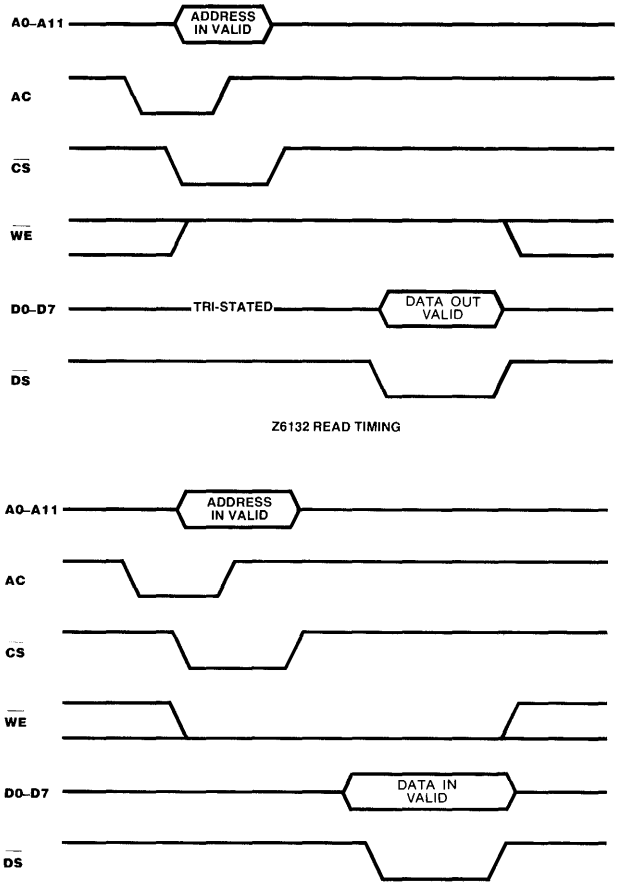


Figure 12.37 Z6132 timing.

128 refresh cycles to completely refresh the block. Two user-selectable refresh modes are provided, the long-cycle-time mode and the short-cycle-time mode.

The long-cycle-time mode is selected by pulling the $\overline{\text{BUSY}}$ pin low. In this mode, every memory cycle is followed by a refresh operation on both blocks of memory in the Z6132. There must be at least 128 address clock (AC) signals in any 2-ms period. The long-cycle-time mode is most practical in applications where the cycle time exceeds 700 ns.

The short-cycle-time refresh mode is selected by pulling the $\overline{\text{BUSY}}$ pin high through a pull-up resistor. In this mode, the Z6132 performs a refresh operation on the memory block that is not being accessed. If the chip is not selected ($\overline{\text{CS}}$ high), both blocks are refreshed. If the chip is selected, only the block that is not addressed by A0 is refreshed; the refresh occurs simultaneously with the access of the other block. This scheme takes advantage of the sequential nature of most memory addressing; normally, this odd/even refresh scheme will provide 128 refresh operations to each block within 2 ms. In the unlikely event of 17 consecutive all odd (A0 = 1) or all even (A0 = 0) accesses, the refresh operation will automatically request one long memory cycle to append a refresh operation to the appropriate block. The $\overline{\text{BUSY}}$ line is pulled low during this cycle; the $\overline{\text{BUSY}}$ pins from all the system's Z6132's can be OR-tied together and fed into the CPU's $\overline{\text{WAIT}}$ input. This is the refresh method appropriate for most Z8000-based systems.

Thus the Z6132 is well suited for microprocessor applications where its byte-wide organization, self-refresh capability, and Z-Bus interface logic serve to simplify design and reduce system's parts count.

13

Z-Bus Microcomputers

A single-chip microcomputer is an entire computer (CPU, memory, and I/O) incorporated on a single integrated-circuit chip. The Z8 family of microcomputers are stand-alone single-chip microcomputers that can access memory external to the chip via a Z-Bus interface. Thus the Z8 can be interfaced easily to a Z8000 system in a distributed processing application. The Universal Peripheral Controller (UPC) is an intelligent peripheral controller for Z8000 systems based on the Z8 architecture. The Z8 and UPC architectures provide powerful processing features, including fast execution times, efficient memory usage, sophisticated interrupt structures, I/O and bit manipulation capabilities, and powerful instruction set.

Z8 ARCHITECTURAL OVERVIEW

Figure 13.1 shows the pin-out of the Z8601, one member of the Z8 family of single-chip microcomputers. The Z8601 has four 8-bit I/O ports (ports 0, 1, 2, and 3), address strobe ($\overline{AS}$), data strobe ($\overline{DS}$), and read/write (R/$\overline{W}$ signals for interfacing to memory external to the Z8, and two pins (XTAL1 and XTAL2) for connecting a series-resonant crystal or single-phase clock (8 MHz maximum) to an on-chip oscillator. The processor runs at one-half the speed of the external crystal or clock (that is, 4-MHz maximum internal clock).

A block diagram of the Z8601 is given in Fig. 13.2. On-board memory consists of 2K bytes of mask-programmable ROM and 144 byte registers, including 124 general-purpose registers, 4 I/O port registers, and 16 status

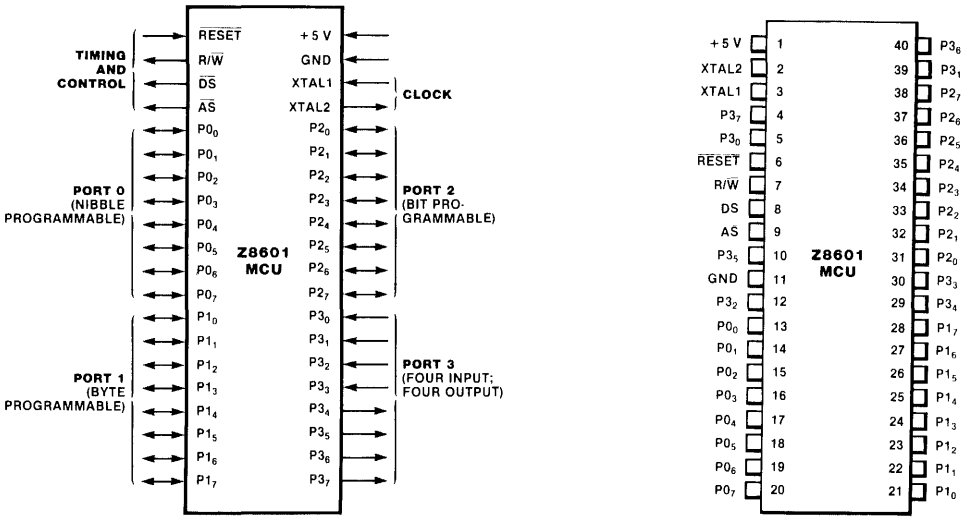


Figure 13.1 Z8601 microcomputer pin assignments.

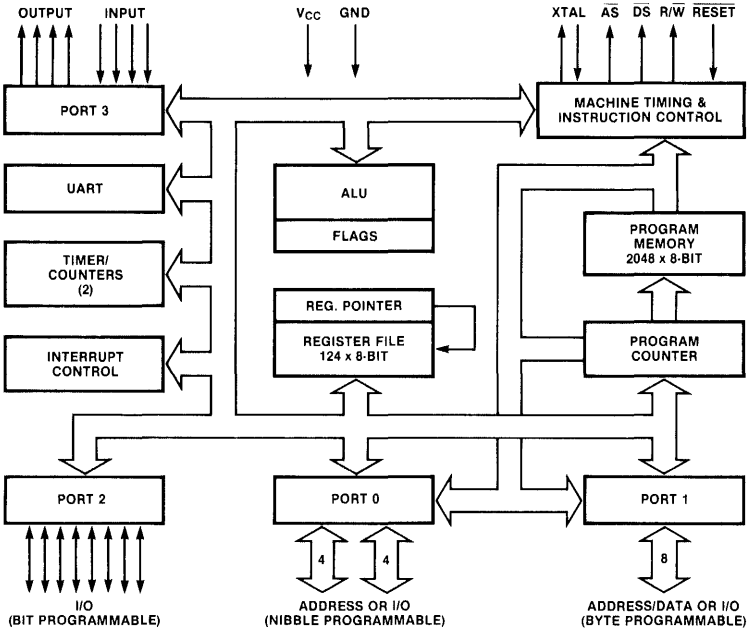


Figure 13.2 Z8601 microcomputer block diagram.

and control registers. The Z8 can be interfaced to up to 124K bytes of external memory as a programmable option. Memory addresses are 16 bits long, register addresses are 8 bits long, and data transfers are always 8-bit (byte) transfers. The Z8's programmable options can be configured for a particular application by writing to the status/control registers. Port 1 can be used as a byte-programmable I/O port or as an 8-bit multiplexed address/data bus to external memory. Port 0 is a nibble-programmable I/O port or additional address bits for interfacing to external memory. Port 2 is a bit-programmable I/O port. Port 3 is always 4 bits of input and 4 bits of output; port 3 pins can be used for several control functions, including handshake signals for the other ports, interrupt request inputs, serial I/O lines, and external access to the Z8's counter/timers. An on-board full duplex UART (universal asynchronous receiver/transmitter) provides serial I/O capability. Two programmable counter/timers with several user-selectable modes also are provided.

Since the same port 0 and port 1 pins can be used as I/O ports or as an interface to memory external to the Z8, a Z8 system can have several different configurations. For an I/O intensive application with a relatively short program, all four ports are used as I/O ports, thereby providing 32 bits of I/O (Fig. 13.3). The program code resides in the 2K bytes of on-board ROM and data resides in the general-purpose registers. If slightly more memory is needed, port 1 can be configured as an 8-bit multiplexed address/data bus to memory external to the Z8; thus 256 bytes of additional memory can be accessed (Fig. 13.4). Ports 0, 2, and 3 still are available for I/O in this configuration. At the expense of using one bit of port 3 as a status line called data memory select ($\overline{DM}$), separate program and data memory areas can be defined in external memory. Therefore, this configuration could have 256 bytes of

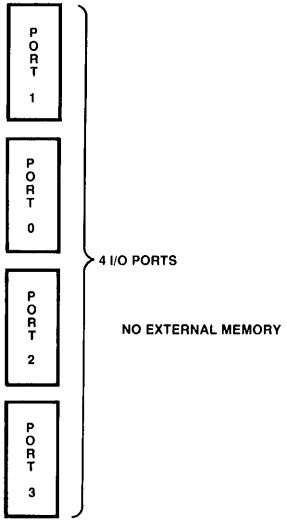


Figure 13.3 Z8 configuration for I/O intensive applications.

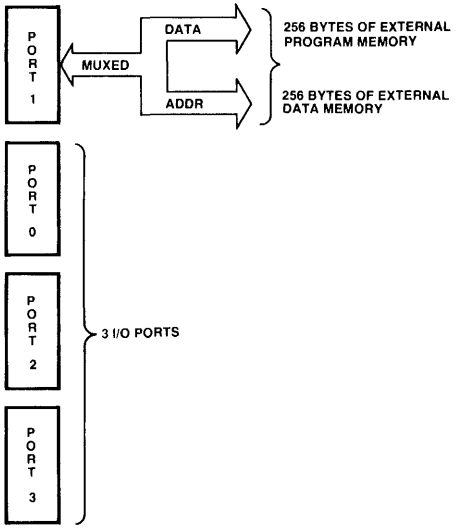


Figure 13.4 Z8 configuration for I/O intensive applications requiring some external memory.

program memory and 256 bytes of data memory external to the Z8. For a more memory intensive application, the lower 4 bits of port 0 can provide 4 more bits of address to external memory; when combined with port 1, 12 address lines to external memory allow an interface to 4K bytes of program memory and 4K bytes of data memory external to the Z8 (Fig. 13.5). Ports 2, 3, and one-half of port 0 are still available as I/O pins. In a very memory intensive application, all of ports 0 and 1 can be used to access external memory. This results in 16-bit addresses to external memory, allowing 62K bytes

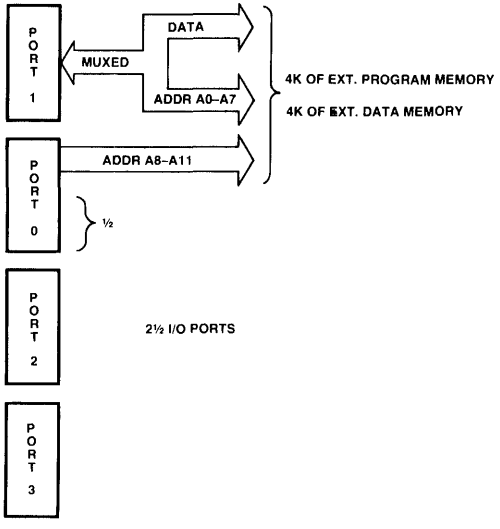


Figure 13.5 Z8 configuration with interface to 4K bytes of external memory.

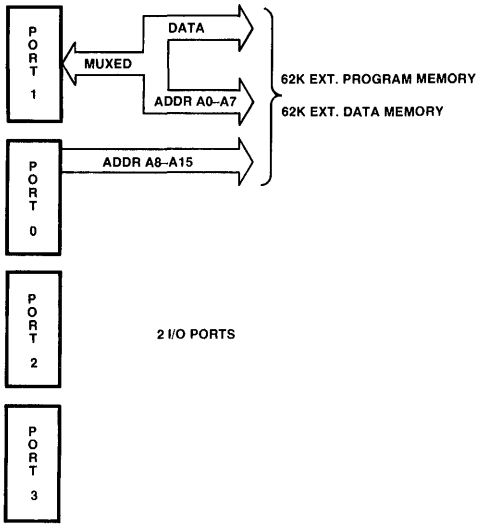


Figure 13.6 Z8 configuration for memory intensive applications.

of program memory and 62K bytes of data memory external to the Z8 (Fig. 13.6). (The first 2K of addresses, 0 to 7FF hexadecimal, are reserved for the on-board ROM.) Thus the Z8 can handle a wide range of both memory-intensive and I/O-intensive applications.

Z8 MEMORY SPACES

Three different memory address spaces are available in Z8 systems: program memory, data memory, and the registers.

Program memory is memory that can be accessed during instruction fetches. The Z8's 16-bit program counter can address 64K bytes of program memory. The first 2048 bytes of the Z8061's program memory resides in the mask-programmable ROM on the chip (Fig. 13.7). Up to 62K bytes of external program memory can be added to a Z8601-based system provided that ports 1 and 0 are configured to act as an address/data bus to external memory.

The first 12 bytes of program memory are used to hold six 16-bit interrupt vectors, where each vector corresponds to an interrupt source. When an interrupt occurs, program control is passed to the service routine whose starting address is stored in the appropriate vector location. Resetting the Z8 forces the program counter to location 12 (0C hexadecimal), the first program memory location available for user's code.

External program memory can be implemented in any combination of ROM and RAM. Program memory can be used to hold data as well as program code and is accessed by the Load Control (LDC) instruction.

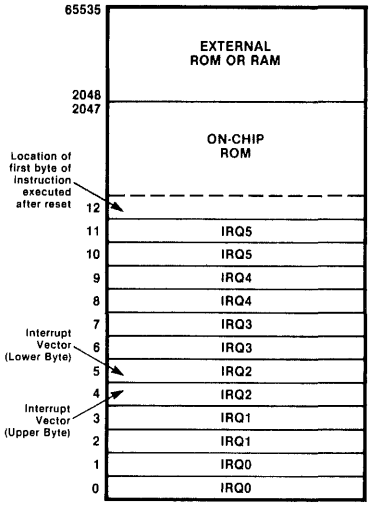


Figure 13.7 Z8 program memory.

Data memory, on the other hand, can hold only data; instruction fetches access only program memory. Data memory is always external to the Z8 chip (Fig. 13.8); up to 62K bytes of data memory can be included in a system, depending on the configuration of ports 1 and 0. Separate program and data areas in external memory are implemented by programming bit 4 of port 3 to be the data memory select ($\overline{DM}$) signal. When the line is low, data memory is being accessed; when $\overline{DM}$ is high, program memory is being accessed. Thus the $\overline{DM}$ signal can be used as part of the chip select logic to external memory to segregate the program and data memory areas (Fig 13.9). The state of the

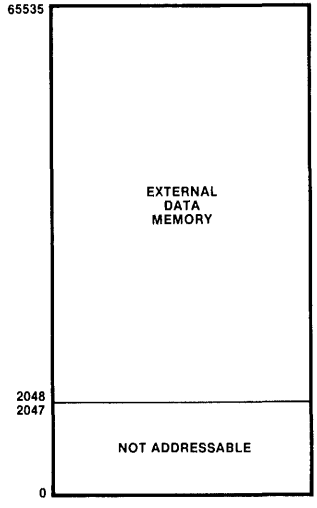


Figure 13.8 Z8 data memory.

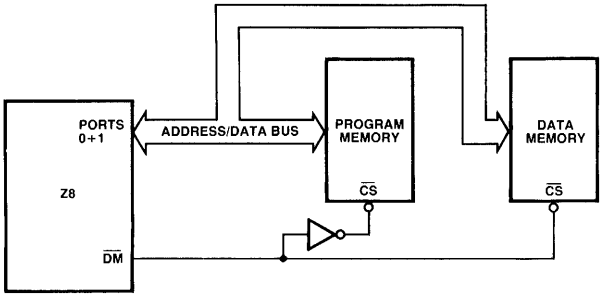


Figure 13.9 Using the $\overline{DM}$ signal to segregate external program and data memory.

$\overline{DM}$ signal when accessing a data operand during instruction execution depends on the instruction being executed; the Load Control (LDC) instruction accesses program memory and the Load External (LDE) instruction accesses data memory. In other words, the instruction used determines which memory space is being accessed.

The 144-byte register file includes 4 I/O port registers (R0-R3), 16 status and control registers (R240-R255), and 124 general-purpose registers (R4-R127), as illustrated in Fig. 13.10. I/O ports 0 to 3 are accessed by reads and writes to registers 0 to 3, respectively. Thus there are no explicit I/O instructions in the Z8 instruction set; any instruction that acts on a register also can act on an I/O port. The general-purpose registers can be used as ac-

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

Figure 13.10 Z8 registers.

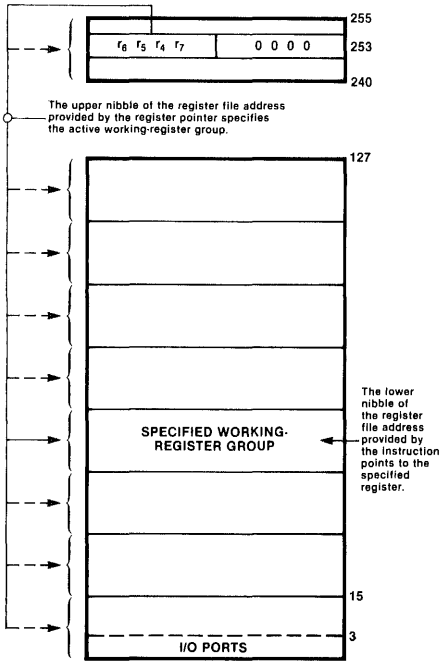


Figure 13.11 Z8 register pointer.

cumulators, address pointers, or index registers. The status and control registers are used to configure the Z8's programmable options and to hold status information, such as the state of the ALU flags.

Z8 registers can be accessed directly or indirectly using 8-bit register addresses. However, one of the control registers (R253, the Register Pointer) allows 4-bit register addresses, resulting in shorter and faster instructions. The registers are divided into nine groups of 16 registers each (Fig. 13.11). The Register Pointer holds the starting address of one of these nine groups; the 16 registers in the specified group are called "working registers" and can be accessed with 4-bit register addresses.

Either the internal register file or external data memory can be used to hold the Z8's stack. An 8-bit stack pointer (R255) is used if the stack is in the registers; a 16-bit stack pointer (R254 and R255) is used if the stack is in data memory.

The Z8's flags consist of carry, zero, sign, overflow, half-carry, and decimal-adjust flags, just as in the Z8000. These flags are held in register R252.

Z8 I/O PORTS

The Z8's 32 I/O lines are configured as four 8-bit parallel I/O ports, called ports 0, 1, 2, and 3. The ports also can be programmed to provide an inter-

face to external memory, serial I/O, handshakes for parallel I/O, status signals, access to the counter/timers, or interrupt request inputs.

Port 1 can be configured as a byte I/O port or as a time-multiplexed address/data bus to external memory. As an I/O port, port 1 can be a byte of input or a byte of output. Optionally, transfers with this port can be controlled by a two-wire interlocked handshake; bits 3 and 4 of port 3 provide the handshake signals. The port is accessed via reads and writes to register R1.

To interface to external memory, port 1 must be configured as an 8-bit time-multiplexed address/data bus. The $\overline{AS}$, $\overline{DS}$, and R/$\overline{W}$ signals are used to control data transfers on this bus. Port 1 and these control signals can be placed in a high-impedance state to allow bus sharing in Z8 systems.

Port 0 can be used as a nibble-programmable I/O port or as additional address lines for interfacing to external memory. When configured as an I/O port, the two nibbles of port 0 can be independently programmed as inputs or outputs. Optionally, bits 2 and 5 of port 3 can provide an interlocked handshake for port 0 I/O operations; the direction of the handshake is the direction of port 0's upper nibble.

For external memory interfacing, port 0 can provide four additional address lines (lower nibble only) or eight address lines (both nibbles). If only the lower nibble is required for memory addressing, the upper nibble still can be used for I/O. The port 0 lines defined as address bits can be placed in the high-impedance state along with the port 1 pins for bus-sharing applications.

The configuration of ports 0 and 1 is controlled by register R248, the Port 0-1 Mode register (Fig. 13.12).

When external memory is included in a Z8 system, ports 1 and 0 are configured to provide the desired number of address bits. The timing for external memory accesses is the standard Z-Bus timing, as illustrated in Fig. 13.13. The clock shown is the external clock (8 MHz maximum); the processor runs at one-half that speed. During T1, the address is emitted and $\overline{AS}$ is pulsed; the

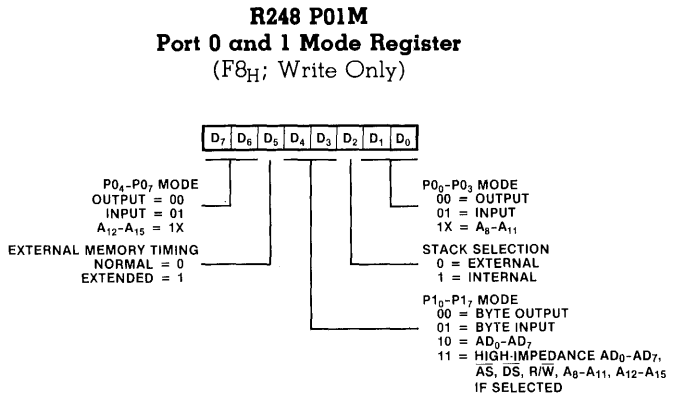


Figure 13.12 Port 0-1 mode register.

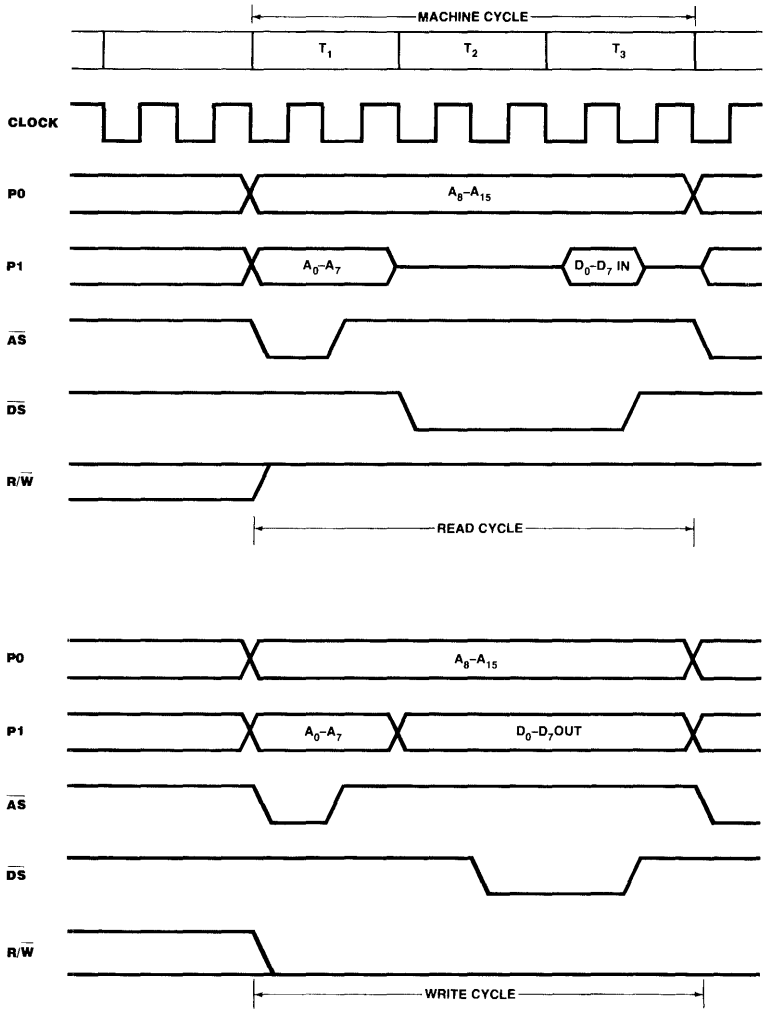


Figure 13.13 Z8 external memory interface timing.

address and R/$\overline{W}$ signal are guaranteed valid on the rising edge of $\overline{AS}$. During T2, the multiplexed portion of the bus (the port 1 pins) is cleared of the address. For reads, $\overline{DS}$ is lowered in anticipation of receiving data from memory; for writes, the data are emitted before $\overline{DS}$ goes active. During T3, the data are read into the CPU or the write is completed. The rising edge of $\overline{DS}$ marks the end of the data transfer. For a 4-MHz system, the worst-case memory access time would be 320 ns. As an option, extended memory timing can be selected by setting bit 5 of the Port 0-1 Mode register. With extended memory timing, one wait state is inserted between T2 and T3 of each external

R246 P2M
Port 2 Mode Register
($F6_H$; Write Only)

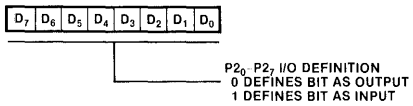


Figure 13.14 Port 2 Mode register.

R247 P3M
Port 3 Mode Register
($F7_H$; Write Only)

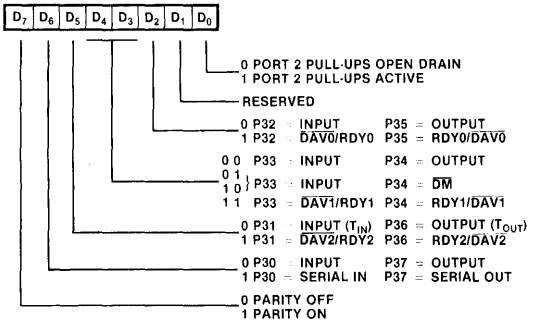


Figure 13.15 Port 3 Mode register.

memory access, thereby increasing the access time by a full clock period of the internal processor clock.

Port 2 is always a bit-programmable I/O port wherein each bit can be individually programmed as an input or output line using R246, the Port 2 Mode register (Fig. 13.14). Any pins defined as outputs can be active or open-drain outputs. Optionally, bits 1 and 6 of port 3 can provide an interlocked handshake for port 2; the direction of the handshake is the direction of bit 7 in port 2.

Port 3 consists of four input pins (bits 0-3) and four output pins (bits 4-7). These pins can be used as I/O lines or control signals, as determined by R247, the Port 3 Mode register (Fig. 13.15). Port 3 pins can provide the handshake signals for the other ports, serial I/O lines (bit 0 for serial in, bit 7 for serial out), timer input and output signals, four external interrupt requests, and the data memory select ($\overline{DM}$) signal.

Z8 COUNTER/TIMERS

The Z8 has two independent 8-bit programmable counter/timers, T0 and T1. Each counter/timer has a programmable prescaler and count register; one mode register controls the configuration of both counter/timers (Fig. 13.16). T0 is

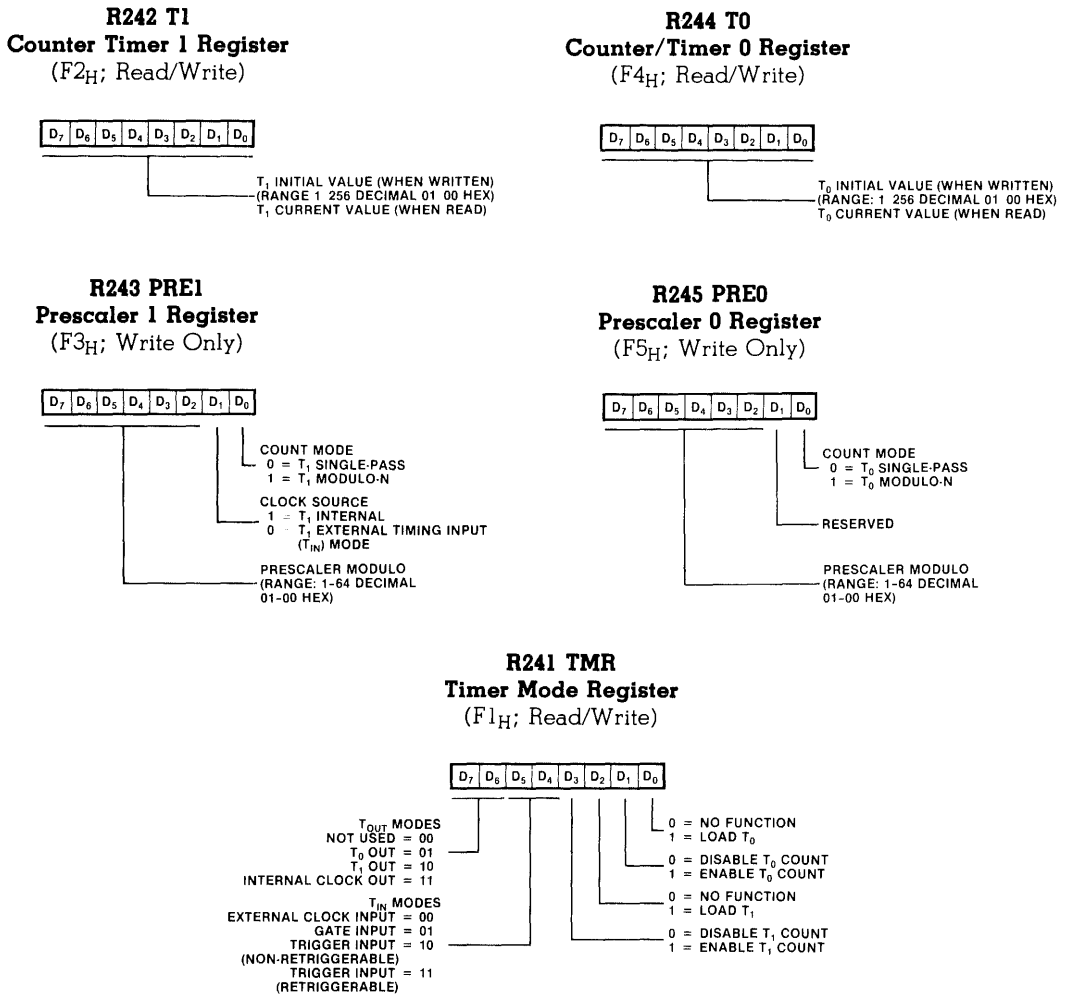


Figure 13.16 Counter/timer control and status registers.

driven by the internal processor clock divided by 4 (that is, a 1-MHz clock if an 8-MHz external crystal is used); T1 can be driven by this internal clock or an external signal from port 3, bit 1. Bit 1 of port 3 can be used as a gate or trigger for T1, also. Either counter/timer can generate an interrupt at the end-of-count condition and/or have its output routed to bit 6 of port 3, which toggles at the end-of-count. Thus the counter/timers can be used for a variety of applications, including event counters, watch-dog timers, delay timers, and square-wave generators.

The 6-bit prescaler is a clock divider that can divide the input clock to the counter/timer by any value from 1 to 64. The output of the prescaler

drives the down-counter, decrementing the count. The current count can be read at any time without disturbing the counting process. Either single-cycle mode (counter stops upon reaching zero) or modulo-n mode (the counter reloads the initial count value upon reaching end-of-count) can be specified.

Z8 SERIAL I/O

The Z8 contains a full-duplex serial asynchronous receiver/transmitter that is enabled by setting bit 6 of the Port 3 Mode register. If this serial I/O device is used, counter/timer T0 must be used as the baud rate clock generator. Data rates up to 62.5K bits/second are possible. Bit 0 of port 3 acts as the serial-in line and bit 7 of port 3 is the serial-out line. Received data are read from register R240 and data to be transmitted are written to register R240. Optionally, an odd-parity generator and checker is available.

The Z8 transmits 8-bit characters with one start bit and two stop bits. If parity is enabled, the eighth bit will be replaced by an odd-parity bit. Received data must be formatted as 8-bit characters with a start bit and at least one stop bit. If parity is enabled, the eighth bit of a received character is replaced by a parity error flag. Separate interrupt requests may be generated upon transmitting or receiving a character. The transmitter is single-buffered and the receiver is double-buffered with no overwrite protection provided in the hardware.

Z8 INTERRUPTS

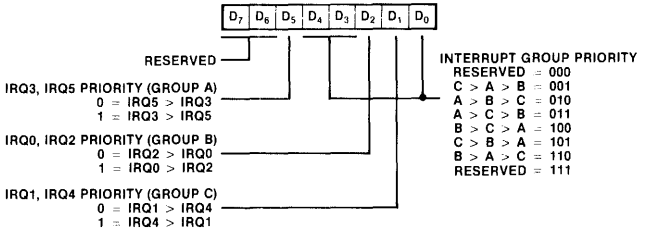
The Z8 provides for six different interrupts from eight possible sources: the four port 3 inputs, the two counter/timers, the UART's receiver, and the UART's transmitter (Table 13.1). Three registers control the interrupt structure: the Interrupt Priority, Interrupt Request, and Interrupt Mask registers (Fig. 13.17). The Interrupt Mask register globally or individually enables or disables the six interrupt requests. When more than one interrupt is pending, the contents of the Interrupt Priority register determines which interrupt request is serviced first.

The Z8's interrupt processing mechanism is diagrammed in Fig. 13.18. The appropriate bit in the Interrupt Request register is set when the event corresponding to that interrupt request occurs. If that interrupt is enabled in the Interrupt Mask register, interrupt processing begins at the end of the current instruction's execution. The Interrupt Priority register is used to determine priority in the case of simultaneous requests. The current program counter and flags (R252) are pushed onto the stack, interrupts are disabled, and the service routine pointed to by the appropriate interrupt vector is executed. The six interrupt vectors are in the first 12 bytes of program memory.

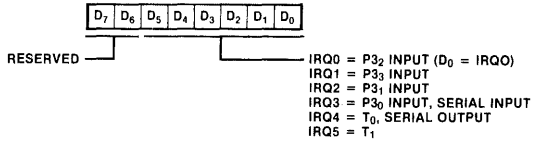
TABLE 13.1 Z8 INTERRUPT REQUESTS

Name	Source	Vector location in program memory	Comments
IRQ0	Port 3, bit 2	0, 1	Negative-edge triggered; can be port 0 handshake
IRQ1	Port 3, bit 3	2, 3	Negative-edge triggered; can be port 1 handshake
IRQ2	Port 3, bit 1	4, 5	Negative-edge triggered; can be port 2 handshake or timer in signal
IRQ3	Port 3, bit 0 or serial in	6, 7	Negative-edge triggered if port 3 input
IRQ4	T0 end-of-count or serial out	8, 9	
IRQ5	T1 end-of-count	10, 11	

R249 IPR
Interrupt Priority Register
($F9_H$; Write Only)



R250 IRQ
Interrupt Request Register
(FA_H; Read/Write)



R251 IMR
Interrupt Mask Register
(FB_H; Read/Write)

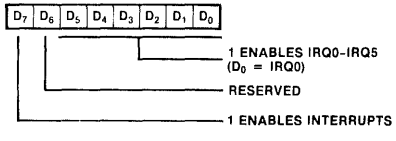


Figure 13.17 Interrupt control and status registers.

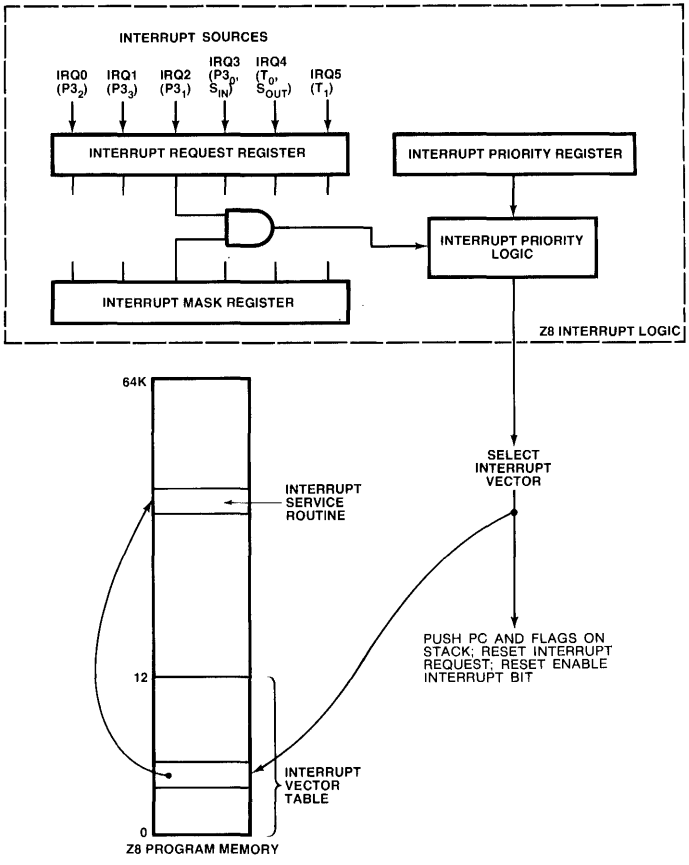


Figure 13.18 Z8 interrupt processing.

Nested interrupts are possible by reenabling interrupts within the service routine. The Interrupt Return (IRET) instruction restores the program counter and flags of the interrupted program upon the completion of the service routine.

Polled interrupt systems also are supported. In a polled system, the interrupts to be polled are disabled in the Interrupt Mask register. The Interrupt Request register is read at predetermined intervals to determine which interrupt requests require servicing.

Z8 INSTRUCTION SET

The Z8's assembly language instruction set is optimized for high code density and fast execution time. The Z8 features 43 instruction types and six operand addressing modes.

The operand addressing modes include register, indirect register, direct address, relative address, indexed, and immediate modes. In register mode, the operand value is the contents of the register specified in the instruction. Registers can be used in pairs to hold 16-bit values or memory addresses. Indirect register addressing means that a register or register pair holds the address of the location whose contents is to be used as the operand. Both the registers and memory can be accessed using indirect register mode. In fact, data operands in program or data memory can be accessed only with indirect register mode. Direct addressing is used by the Jump and Call instructions to designate the address to be loaded into the program counter; in this mode, the destination address is given in the instruction itself. Relative addressing means that an offset is specified in the instruction; this offset is added to the current program counter contents to form the destination address. An indexed address consists of a register address offset by the contents of a designated working register (the index). Indexing is allowed only within the registers and is supported only by the Load instruction. For immediate mode, the operand value is supplied in the instruction itself. These addressing modes are very similar in operation to the corresponding addressing modes in the Z8000 CPUs (see Chapter 8).

Table 13.2 is a list of the Z8 assembly language instruction set. The Z8's Jump and Jump Relative instructions use the same set of condition codes as the Z8000 (See Table 8.3). The Z8 is a register-oriented processor; arithmetic and logical operations can be performed only on data in registers. Three different load instructions are provided. Load (LD) is a register-to-register or immediate-to-register load. Load Constant (LDC) is for data transfers between a register and program memory. Load External (LDE) is for data transfers between a register and data memory. The block transfer instructions (LDCI and LDEI) are used within program loops to move entire blocks of data between the registers and program or data memory. LDC, LDE, LDCI, and LDEI all use indirect register addressing to access memory. The Pop and Push instructions might access the registers or external data memory, depending on the location of the stack. Similarly, the Call and Interrupt Return instructions access either the registers or data memory when pushing or popping status information. All other instructions operate on the registers only. The Test Under Mask (TM) and Test Complement Under Mask (TCM) instructions allow bit testing of register contents. The Decimal Adjust (DA) instruction is used

TABLE 13.2 Z8 INSTRUCTION SET

Instruction	Operand(s)	Name of instruction
Load		
CLR	dst	Clear
LD	dst, src	Load
LDC	dst, src	Load Constant
LDE	dst, src	Load External Data

TABLE 13.2 Z8 Instruction Set (*Continued*)

Instruction	Operand(s)	Name of instruction
POP	dst	Pop
PUSH	src	Push
Arithmetic		
ADC	dst, src	Add with Carry
ADD	dst, src	Add
CP	dst, src	Compare
DA	dst	Decimal Adjust
DEC	dst	Decrement
DECW	dst	Decrement Word
INC	dst	Increment
INCW	dst	Increment Word
SBC	dst, src	Subtract with Carry
SUB	dst, src	Subtract
Logical		
AND	dst, src	Logical And
COM	dst	Complement
OR	dst, src	Logical Or
XOR	dst, src	Logical Exclusive Or
Program control		
CALL	dst	Call Procedure
DJNZ	r, dst	Decrement and Jump if Nonzero
IRET		Interrupt Return
JP	cc, dst	Jump
JR	cc, dst	Jump Relative
RET		Return
Bit manipulation		
TCM	dst, src	Test Complement under Mask
TM	dst, src	Test under Mask
Block transfer		
LDCI	dst, src	Load Constant Autoincrement
LDEI	dst, src	Load External Data Autoincrement
Rotate and Shift		
RL	dst	Rotate Left
RLC	dst	Rotate Left through Carry
RR	dst	Rotate Right
RRC	dst	Rotate Right through Carry
SRA	dst	Shift Right Arithmetic
SWAP	dst	Swap Nibbles
CPU control		
CCF		Complement Carry Flag
DI		Disable Interrupts
EI		Enable Interrupts
NOP		No Operation
RCF		Reset Carry Flag
SCF		Set Carry Flag
SRP	src	Set Register Pointer

to perform arithmetic on binary-coded-decimal data, in the same manner as the Z8000's DAB instruction.

Z8 CONFIGURATIONS

Several different product configurations are available within the Z8 family (Table 13.3). The Z8601 is a 40-pin device with 2K bytes of mask-programmable ROM, as previously described. The Z8602 is a 64-pin development version of the Z8601 that allows the user to prototype the system in hardware without mask-programming the code. The Z8602 is identical in function to the Z8601 except that the 2K bytes of internal ROM are removed, the ROM address and data lines are brought out to the additional pins, and control signals for accessing the first 2K bytes of program memory external to the chip have been added. Thus the program memory is implemented external to the Z8602 with an EPROM or PROM. The Z8603 is a combination of the Z8601 and Z8602 wherein the first 2K bytes of program memory are external to the device. The Z8603 Protopak is a 40-pin chip that is pin-compatible with the Z8601; the Z8603 carries a 24-pin socket in "piggyback" manner for the EPROM that holds the first 2K bytes of program memory (Fig. 13.19). Thus the Z8603 allows the user to design a printed circuit board for the 40-pin mask-programmable Z8601 and use the Z8603 Protopak to build prototype and pilot production units.

The Z8611 is identical to the Z8601 except that 4K bytes of mask-programmable ROM is provided on the chip. Up to 60K bytes of program memory and 60K bytes of data memory can be accessed external to the Z8611 if ports 0 and 1 are configured appropriately. The Z8612 is the 64-pin proto-

TABLE 13.3 Z8 PRODUCT CONFIGURATIONS

Part number	Package	Description
Z8601	40-pin	2K bytes of internal mask-programmable ROM
Z8602	64-pin	No internal ROM; interface to 2K bytes of external ROM/PROM
Z8603	40-pin Protopak	No internal ROM; 24-pin socket for 2K bytes of external EPROM
Z8611	40-pin	4K bytes of internal mask-programmable ROM
Z8612	64-pin	No internal ROM; interface to 4K bytes of external ROM/PROM
Z8613	40-pin Protopak	No internal ROM; 24-pin socket for 4K bytes of external EPROM
Z8681	40-pin	No internal ROM; port 1 pins an interface to external memory
Z8671	40-pin	Z8601 with a BASIC debugger/interpreter

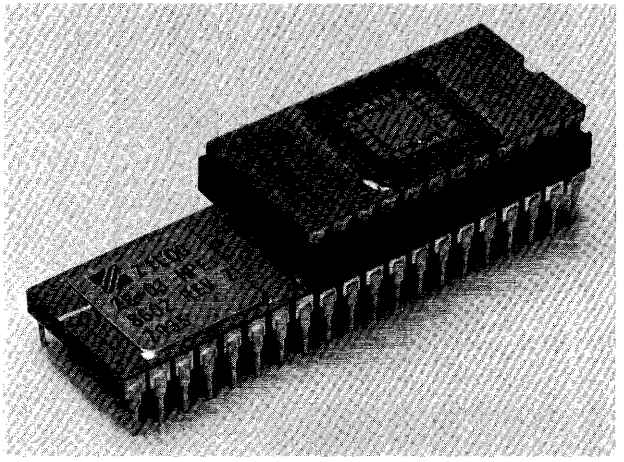


Figure 13.19 Z8 Protopak package.

typing version of the Z8611 with all memory external to the chip; the Z8613 is the 40-pin Protopak version of the Z8611.

The Z8681 is a "ROMless" version of the Z8 with no program memory on board the device. Port 1 is always configured as an address/data bus to external memory; port 0 is still nibble-programmable as I/O or additional address lines. Therefore, the Z8681 can address up to 64K of program memory and 64K of data memory.

The Z8671 microcomputer is a Z8601 with a BASIC interpreter and debugger already programmed into the 2K bytes of mask-programmable ROM. The BASIC language used is a subset of Dartmouth BASIC.

Z8000-Z8 INTERFACING

A Z8 microcomputer could be used as another processor in a Z8000-based system in distributed processing applications. For example, a Z8 might be used as a front-end I/O processor dedicated to data handling and formatting for a specific I/O device in the system.

Z8's can be interfaced to Z8000 systems in a number of ways. The most straightforward method is to treat the Z8 as an I/O device for a Z8000 CPU using a Z8036 CIO or Z8038 FIO as the interfacing device (Fig. 13.20). A Z8 port with interlocking handshake interfaces directly to the interlocked handshake of a CIO or FIO port. The other Z8 ports could be used to interface to I/O devices or access external memory.

Since the Z8 uses Z-Bus timing for accessing external memory, a Z8 microcomputer could be used as a bus requestor in a Z8000 system, where the Z8 directly accesses one segment of the Z8000's memory. In Fig. 13.21, one port 3 output bit is used as the bus request signal to the Z8000 CPU and

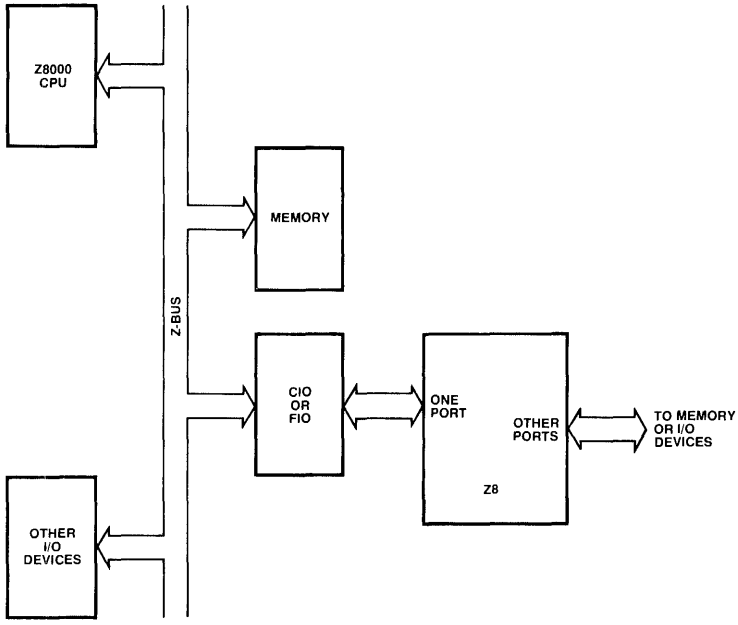


Figure 13.20 Z8000-to-Z8 interface with a CIO or FIO.

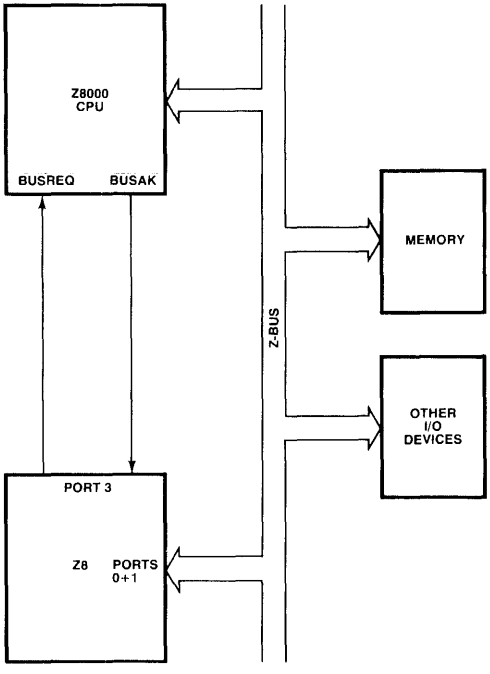


Figure 13.21 Z8000/Z8 bus sharing.

one port 3 input bit is the bus acknowledge. Ports 0 and 1 are defined as an address/data bus to external memory; these are normally held in the tri-state mode. When the Z8 gains control of the bus via a bus request, it can make byte access to the Z8000's memory using LDC, LDE, PUSH, or POP instructions. When the Z8 is not using the Z8000's bus, the Z8 still can be executing from its internal ROM.

UPC ARCHITECTURAL OVERVIEW

The Universal Peripheral Controller (UPC) is a slave microcomputer that can be used as an intelligent peripheral controller in Z8000 systems. The UPC is a complete microcomputer based on the Z8 architecture with its own CPU, memory, and I/O ports on the chip; a Z-Bus interface allows the UPC to act as a byte peripheral in a Z8000-based system. A Z8000 CPU can send and receive byte data from the UPC by reading and writing to the UPC's internal registers via I/O operations. Thus the UPC can unburden the master CPU by handling tasks such as data translation and formatting, arithmetic, and I/O device control.

The UPC is available in several different product configurations. Figure 13.22 shows the pin assignments for the Z8090, a 40-pin UPC with 2K bytes of mask-programmable ROM on board the chip. The Z8090 has three 8-bit parallel ports (ports 1, 2, and 3) a complete Z-Bus peripheral interface (consisting of AD0-AD7, $\overline{AS}$, $\overline{DS}$, R/$\overline{W}$, $\overline{CS}$, and $\overline{WAIT}$), and a TTL-compatible

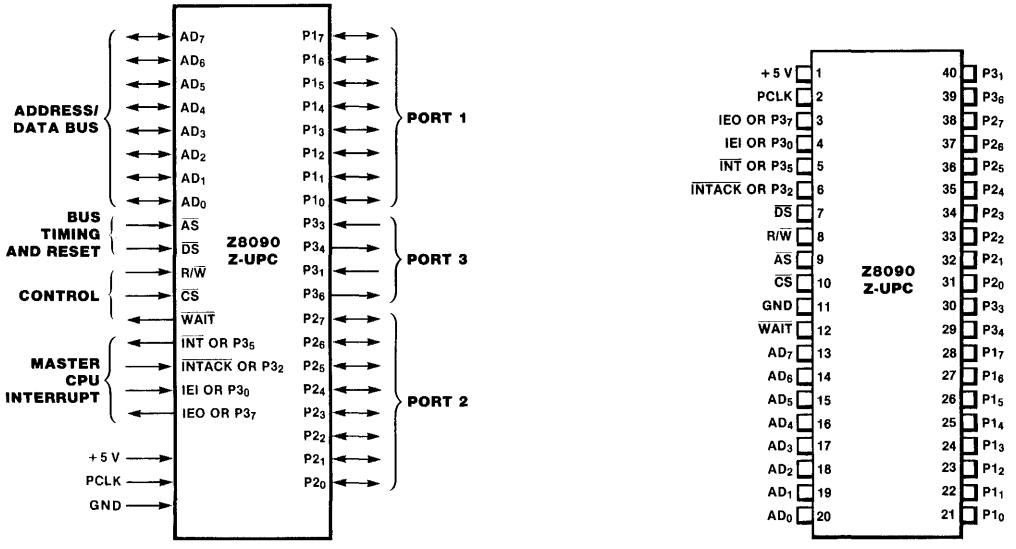


Figure 13.22 Z8090 UPC pin assignments.

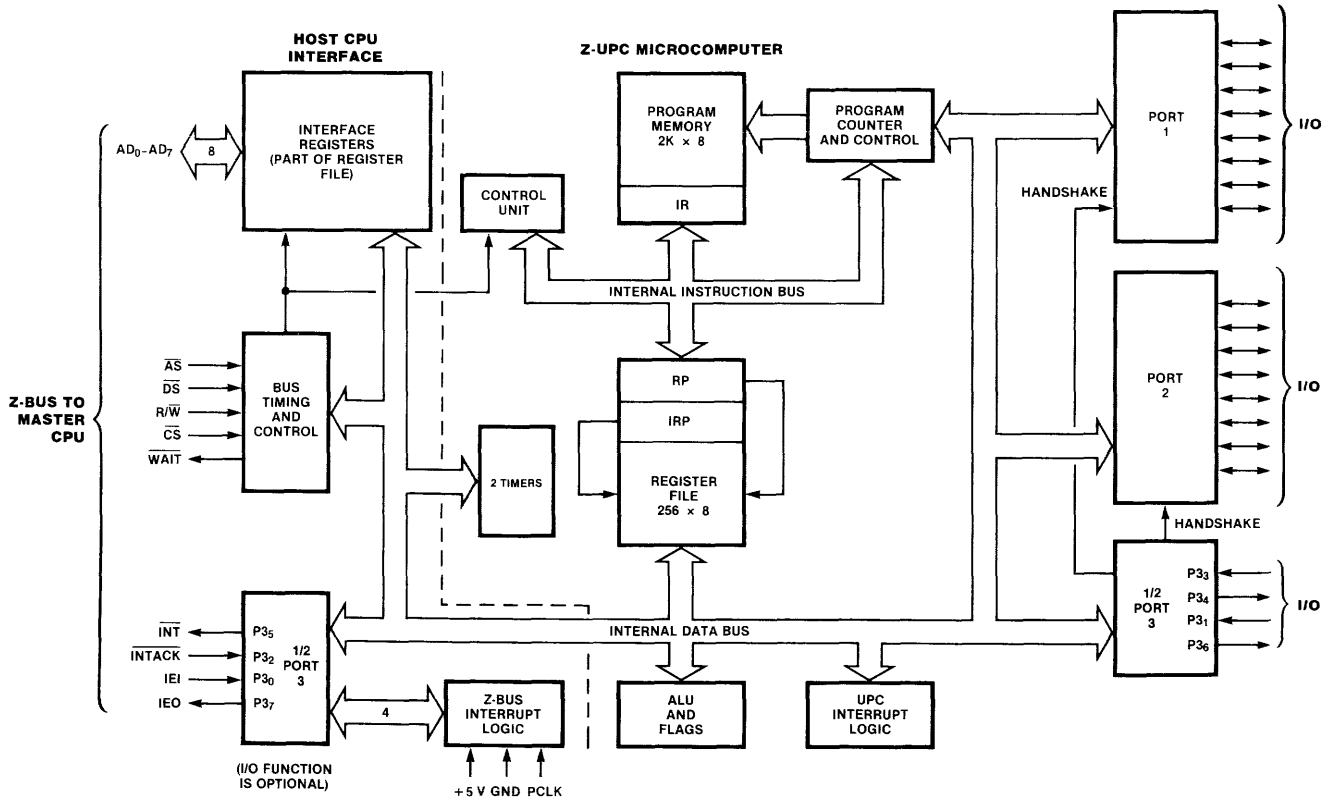


Figure 13.23 Z8090 UPC block diagram.

clock input. Optionally, one-half of port 3 can be used as the interrupt signals ($\overline{INT}$, $\overline{INTACK}$, IEI, and IEO) for the Z-Bus interface. The maximum clock frequency is 4 MHz; the UPC's clock does not need to be synchronized to the master CPU's clock.

A block diagram of the Z8090 is given in Fig. 13.23. The Z8090's memory consists of 2K bytes of mask-programmable ROM and 256 byte registers, including 234 general-purpose registers, 19 status and control registers, and 3 I/O port registers. Ports 1 and 2 are bit-programmable; port 3 consists of 4 input bits and 4 output bits. Port 3 pins can be used for special control functions, including an interrupt interface to the master CPU, handshakes for ports 1 and 2, and interrupt request inputs for the UPC. Unlike the Z8, no mechanism is provided for interfacing to memory external to the Z8090. The UPC's two programmable counter/timers are identical to those in the Z8. The UPC's instruction set also is identical to that of the Z8.

UPC MEMORY SPACES

Two different memory address spaces are available in the UPC, program memory and the registers.

The Z8090 UPC contains 2K bytes of mask-programmable ROM for program memory (that is, memory for holding program code). The UPC's program counter is 16 bits long; however, performance at program addresses above 2K is not defined. The first 12 bytes of program memory are reserved for six 16-bit interrupt vectors (Fig. 13.24). Resetting the UPC forces the program counter to location 12, the first program memory location available for user's code.

The UPC's 256-byte register file consists of 234 general-purpose registers, 3 I/O port registers, and 19 status/control registers (Fig. 13.25). I/O ports are accessed via reads and writes to registers 1, 2, and 3. The general-

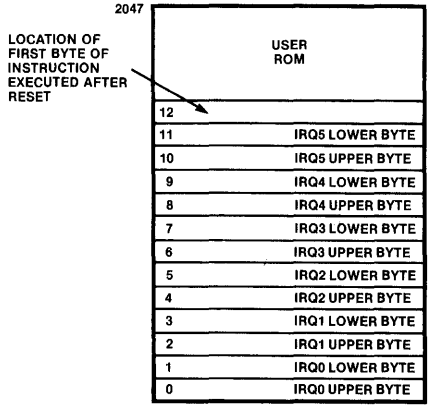


Figure 13.24 UPC program memory.

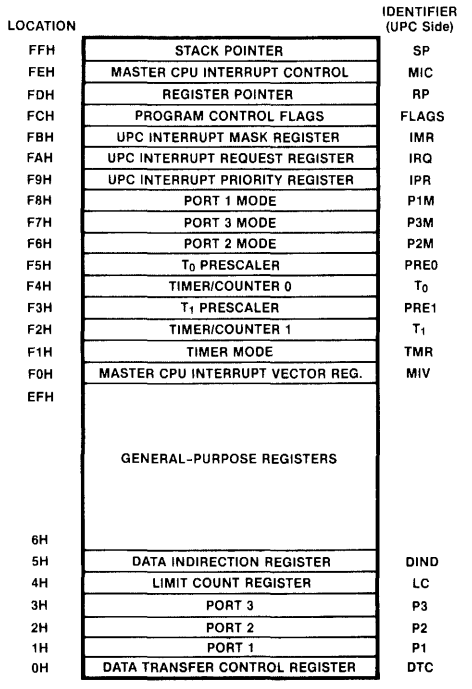


Figure 13.25 UPC register file.

purpose registers can be used as accumulators, address pointers, data buffers, or index registers. The status and control registers are used for configuring the UPC's programmable options, controlling the communication between the UPC and the master CPU, and storing status information.

An 8-bit stack pointer is used for addressing the stack; the stack always resides in the general-purpose register file. As in the Z8, a register pointer addresses the starting point of the 16 working registers. The flag register holds the ALU flags: carry, sign, overflow, zero, decimal adjust, and half-carry. Three registers control the UPC interrupt structure, three others determine the I/O port configuration, and five are used for programming the two counter/timers. The Master CPU Interrupt Control register controls the interrupts to the master CPU and the Master CPU Interrupt Vector register holds the vector that is returned when the master CPU processes an interrupt from the UPC. The Data Transfer Control, Limit Count, and Data Indirection registers are used to control transactions between the UPC and master CPU, as described later.

UPC I/O PORTS

The UPC's 24 I/O lines are organized as three 8-bit parallel ports, ports 1, 2, and 3. Their configuration is determined by the three I/O port mode registers

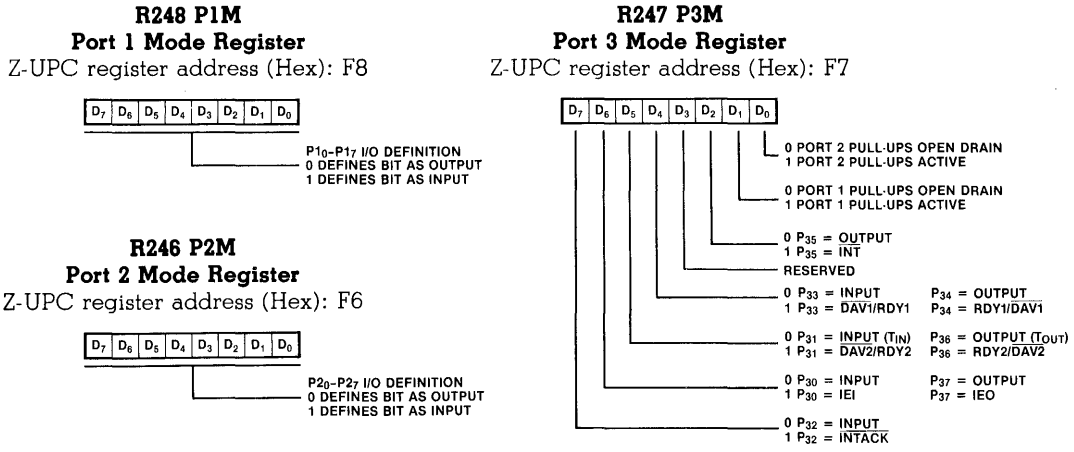


Figure 13.26 Port Mode registers.

(Fig. 13.26). Ports 1 and 2 are bit-programmable ports wherein each bit can be individually programmed as an input or output. Each bit specified as an output can be an active or open-drain output. Optionally, bits 3 and 4 of port 3 can be interlocked handshake signals for port 1 and bits 1 and 6 of port 3 can be the handshake signals for port 2.

Port 3 always has 4 input and 4 output bits. Bits 0, 2, 5, and 7 can be used as the interrupt interface to the master Z8000 CPU (the IEI, $\overline{\text{INTACK}}$, $\overline{\text{INT}}$, and IEO signals, respectively). The other bits in port 3 can be handshake lines for ports 1 and 2, external access for the counter/timers, or bits of I/O. Input bits 0, 1, and 3 of port 3 can be used as UPC interrupt requests regardless of their configuration.

UPC INTERRUPTS

The UPC provides for six different interrupts from eight different sources, as listed in Table 13.4: three port 3 inputs (bits 0, 1, and 3), the two counter/timers, and three master CPU data transfer status bits called end-of-message (EOM), transfer error (XERR), and limit error (LERR). The interrupt mechanism is identical to that of the Z8, with three registers controlling the interrupt structure: the Interrupt Priority, Interrupt Mask, and Interrupt Request registers.

CPU-UPC COMMUNICATION

The UPC is a peripheral for Z8000 systems; the master Z8000 CPU uses I/O operations to read or write to the UPC's register file. All communication be-

TABLE 13.4 UPC INTERRUPT REQUESTS

Name	Source	Vector location in memory	Comments
IRQ0	EOM, XERR, or LERR	0, 1	Data Transfer Control register status bits
IRQ1	Port 3, bit 3	2, 3	Negative-edge triggered; can be port 1 handshake
IRQ2	Port 3, bit 1	4, 5	Negative-edge triggered; can be port 2 handshake or timer in
IRQ3	Port 3, bit 0	6, 7	Negative-edge triggered; can be IEI signal
IRQ4	T0 end-of-count	8, 9	
IRQ5	T1 end-of-count	10, 11	

tween the master CPU and the UPC is initiated by the CPU; the CPU can execute reads and writes to the UPC but the UPC cannot read or write to the master CPU. However, the UPC can issue an interrupt request to gain the CPU's attention.

The master CPU can access the UPC's registers in two ways, direct access and block access. Access by the master CPU is controlled by the UPC, though, in that the UPC determines when a CPU access is allowed, providing software independence between the master CPU and the UPC. The UPC sets the transfer enable (EDX) bit in the Master Interrupt Control register (Fig. 13.27) to enable the CPU to access the UPC register file and resets that bit to disable CPU accesses. Only the UPC can write to the EDX bit. When the CPU completes a transaction with the UPC, the CPU notifies the UPC by setting the end-of-message (EOM) bit in the same register.

The master CPU can directly access 19 of the UPC's registers. Three registers—the Master Interrupt Control, Master Interrupt Vector, and Data Transfer Control registers—are mapped directly into the CPU's I/O address space.

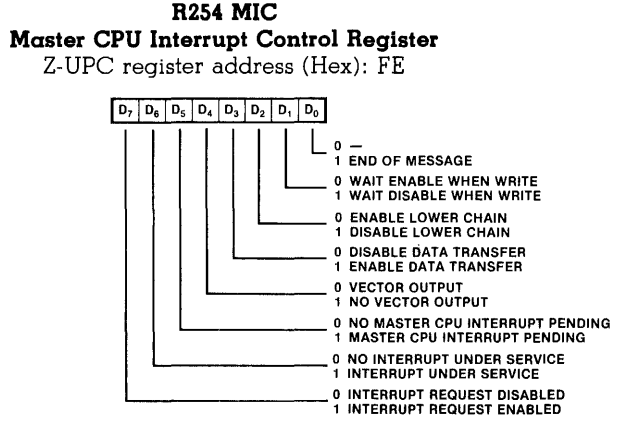


Figure 13.27 Master Interrupt Control register.

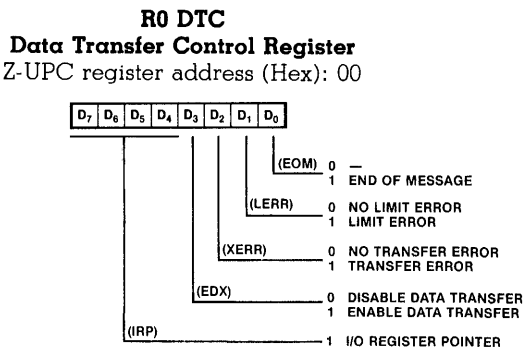


Figure 13.28 Data Transfer Control register.

In other words, each of these registers has a unique port address in the Z8000 system. The Master Interrupt Control register (Fig. 13.27) contains the interrupt enable (IE), interrupt pending (IP), interrupt under service (IUS), no vector (NV), and disable lower chain (DLC) bits that control this interrupt source according to the Z-Bus protocols described in Chapter 12. The Master Interrupt Vector register holds the vector that is read by the CPU during an interrupt acknowledge cycle. The Data Transfer Control register (Fig. 13.28) holds an I/O register pointer and four status bits.

The other 16 registers that can be directly accessed by the master CPU are the 16 contiguous registers designated by the I/O register pointer in the upper nibble of the Data Transfer Control register. The I/O register pointer, like the Register Pointer, selects a group of 16 contiguous registers in the UPC's register file. The actual register address accessed is determined by concatenating the 4-bit I/O register pointer with the four least significant bits of the port address emitted by the master CPU. Thus these 16 registers each have a unique I/O port address in the Z8000 system.

The master CPU also can access UPC registers indirectly using the block access method. This transfer method is controlled by the Data Indirection and Limit Count registers in the UPC (Fig. 13.29). The block access method allows the master CPU to access an entire block of UPC registers using one port address. The address of the first UPC register in the block to be accessed is held in the Data Indirection register and the number of registers in the block is in the Limit Count register. After each master CPU read or write using block access mode, the contents of the Data Indirection register are incremented automatically (so that it points to the next register in the block) and the contents of the Limit Count register are decremented. Thus a Z8000 block I/O instruction can access an entire block of UPC registers.

If the master CPU attempts to access the UPC's registers when the transfer enable (EDX) bit in the Master Interrupt Control register is reset or if the CPU attempts a block access when the Limit Count register is zero, the access is not completed. The transfer error (XERR) bit in the Data Transfer Control register (Fig. 13.28) indicates that if a CPU access was attempted

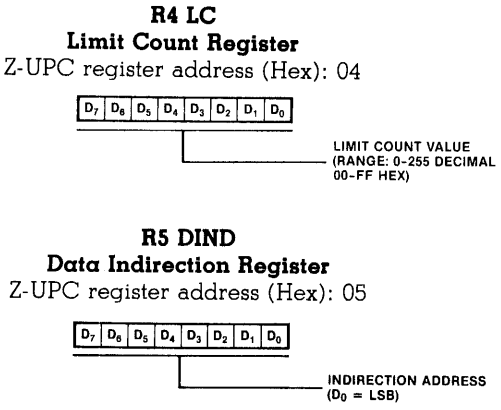


Figure 13.29 Data Indirection and Limit Count registers.

while transfers were disabled (EDX = 0); the limit error (LERR) bit in the same register is set if a block access was attempted while the Limit Count register contained 0. The end-of-message (EOM) and transfer enable (EDX) bits in the Data Transfer Control register are status bits indicating the state of the corresponding control bits in the Master Interrupt Control registers. The setting of the XERR, LERR, or EOM bits in the Data Transfer Control register generates an interrupt request to the UPC.

A typical master CPU-to-UPC data transfer protocol is diagrammed in Fig. 13.30. When the UPC is prepared for a CPU access, the UPC sets the transfer enable bit in the Master Interrupt Control register. The UPC can gain the CPU's attention via an interrupt request by setting the interrupt pending (IP) bit in that same register. Sometime later, the master CPU services the

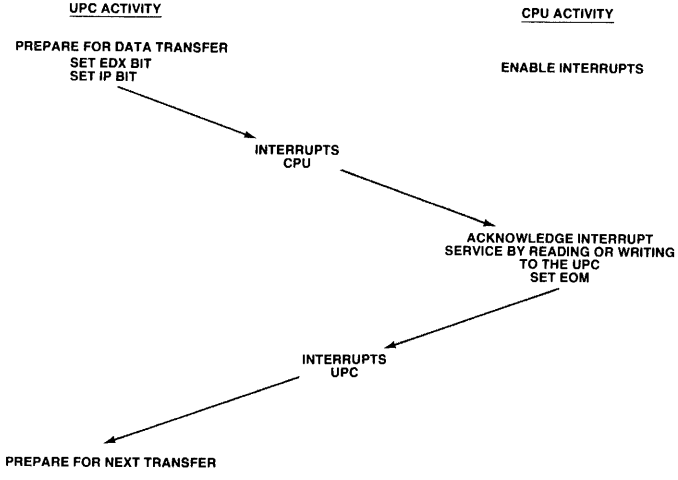


Figure 13.30 Typical master CPU-UPC data transfer protocol.

interrupt by accessing the UPC, using either the direct access or block access method. When this access is completed, the CPU sets the end-of-message bit via a write to the Master Interrupt Control register in the UPC. This, in turn, interrupts the UPC, thereby informing the UPC that the CPU-UPC transfer is completed.

UPC PRODUCT CONFIGURATIONS

The UPC is available in five different product configurations (Table 13.5). The Z8090 has 2K bytes of mask-programmable ROM on board the chip for its program memory, as described previously. A 64-pin version, the Z8091, allows the user to prototype the system in hardware without mask-programming the code. The Z8091 is identical in function to the Z8090 except that the mask-programmable ROM is removed. The additional 24 pins are used as address, data, and control lines to an external ROM, PROM, or EPROM; the

TABLE 13.5 UPC PRODUCT CONFIGURATIONS

Part number	Package	Description
Z8090	40-pin	2K bytes of internal mask-programmable ROM
Z8091	64-pin	No internal ROM; interface to 4K bytes of external ROM/PROM
Z8092	64-pin	36 byte internal ROM holds bootstrap program; interface to 4K bytes of external RAM
Z8093	40-pin Protopak	No internal ROM; 24-pin socket for 4K bytes of external EPROM
Z8094	40-pin Protopak	36 byte internal ROM holds bootstrap program; 24-pin socket for 4K bytes of external RAM

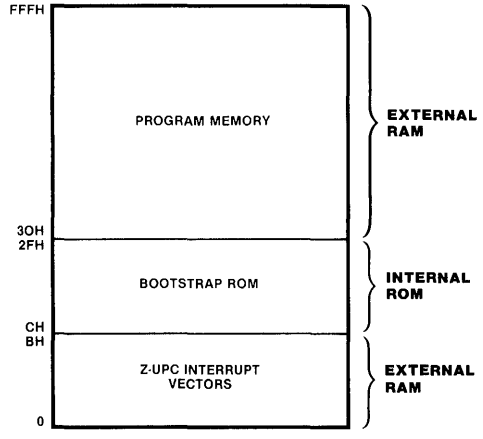


Figure 13.31 Z8092 UPC RAM memory map.

Z8091 can access 4K bytes of program memory external to the chip. The Z8093 Protopak is a combination of the Z8090 and Z8091—a 40-pin package with program memory external to the chip. The Z8093 carries a 24-pin socket in "piggyback" manner for an EPROM that holds the program code (Fig. 13.19).

The Z8092 is a 64-pin UPC with an interface to 4K bytes of external RAM. Thirty-six bytes of ROM are retained on board the chip; this ROM contains a program that allows the user to download code from the master CPU into the UPC's RAM. The internal ROM occupies addresses 0C-2F hexadecimal (Fig. 13.31). The Z8094 is a 40-pin protopak version of the Z8092 wherein the socket for the RAM device is "piggybacked" onto the 40-pin package.

Thus the Z8000 family of components includes processors, memory management devices, peripherals, memories, and single-chip microcomputers. These components are linked via a set of signals called the Z-Bus, providing powerful solutions to a wide variety of applications.

APPENDIX A

Z8000 CPU DC and AC Electrical Characteristics

Absolute Maximum Ratings

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

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

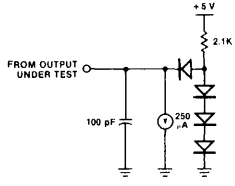
Storage Temperature . . . . . -65°C to +150°C

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

Standard Test Conditions

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

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



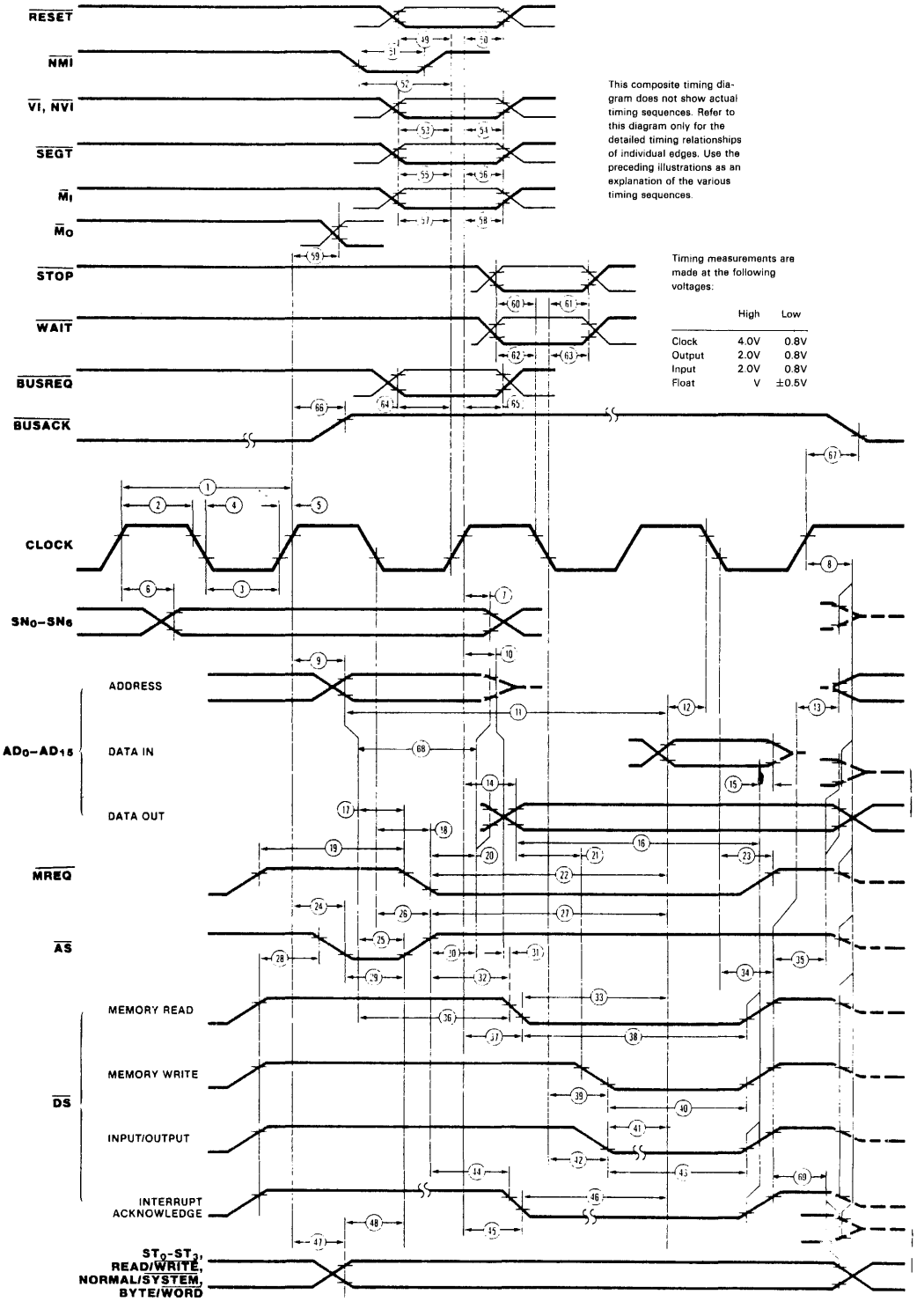
All ac parameters assume a load capacitance of 100 pF max. except for parameter 6 (50 pF max). Timing references between two output signals assume a load difference of 50 pF max.

DC Characteristics

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

Note: Contact Zilog, Inc., for latest information on AC and DC characteristics.

COMPOSITE AC TIMING DIAGRAM



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

Timing measurements are made at the following voltages:

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

AC CHARACTERISTICS

No.	Symbol	Parameter	Z8001/Z8002 Min(ns)	Z8001/Z8002 Max(ns)	Z8001A/Z8002A Min(ns)	Z8001A/Z8002A Max(ns)	Z8001B/Z8002B Min(ns)	Z8001B/Z8002B Max(ns)
1	TcC	Clock Cycle Time	250	2000	165	2000	100	2000
2	TwCh	Clock Width (High)	105	2000	70	2000	40	
3	TwCl	Clock Width (Low)	105	2000	70	2000	40	
4	TfC	Clock Fall Time		20		10		10
5	TrC	Clock Rise Time		20		15		10
6	TdC(SNv)	Clock ↑ to Segment Number Valid (50 pf load)		130		110		70
7	TdC(SNn)	Clock ↑ to Segment Number Not Valid	20		10		5	
8	TdC(Bz)	Clock ↑ to Bus Float		65		55		40
9	TdC(A)	Clock ↑ to Address Valid		100		75		50
10	TdC(Az)	Clock ↑ to Address Float		65		55		40
11	TdA(DR)	Address Valid to Read Data Required Valid		475		305*		180
12	TsDR(C)	Read Data to Clock ↓ Setup Time	30		20		10	
13	TdDS(A)	$\overline{DS}$ ↑ to Address Active	80		45		20*	
14	TdC(DW)	Clock ↑ to Write Data Valid		100		75		50
15	ThDR(DS)	Read Data to $\overline{DS}$ ↑ Hold Time	0		0		0	
16	TdDW(DS)	Write Data Valid to $\overline{DS}$ ↑ Delay	295*		195*		110*	
17	TdA(MR)	Address Valid to $\overline{MREQ}$ ↓ Delay	(55)*		(35)*		20*	
18	TdC(MR)	Clock ↓ to $\overline{MREQ}$ ↓ Delay		80		70		40
19	TwMRh	$\overline{MREQ}$ Width (High)	210*		135*		80*	
20	TdMR(A)	$\overline{MREQ}$ ↓ to Address Not Active	70*		35*		20*	
21	TdDW(DSW)	Write Data Valid to $\overline{DS}$ ↓ (Write) Delay	55*		35*		15*	
22	TdMR(DR)	$\overline{MREQ}$ ↓ to Read Data Required Valid	375		230		140*	
23	TdC(MR)	Clock ↓ $\overline{MREQ}$ ↑ Delay		80		60		45
24	TdC(ASf)	Clock ↑ to $\overline{AS}$ ↓ Delay		80		60		40
25	TdA(AS)	Address Valid to $\overline{AS}$ ↑ Delay	55*		35*		20*	
26	TdC(ASr)	Clock ↓ to $\overline{AS}$ ↑ Delay		90		80		40
27	TdAS(DR)	$\overline{AS}$ ↑ to Read Data Required Valid	360		220		140*	
28	TdDS(AS)	$\overline{DS}$ ↑ to $\overline{AS}$ ↓ Delay	70*		35*		15*	
29	TwAS	$\overline{AS}$ Width (Low)	85*		55*		30*	
30	TdAS(A)	$\overline{AS}$ ↑ to Address Not Active Delay	70		45		20*	
31	TdAz(DSR)	Address Float to $\overline{DS}$ (Read) ↓ Delay	0		0		0	
32	TdAS(DSR)	$\overline{AS}$ ↑ to $\overline{DS}$ (Read) ↓ Delay	80		55		30*	
33	TdDSR(DR)	$\overline{DS}$ (Read) ↓ to Read Data Required Valid	205		130		70*	
34	TdC(DSr)	Clock ↓ to $\overline{DS}$ ↑ Delay		70		65		45
35	TdDS(DW)	$\overline{DS}$ ↑ to Write Data Not Valid	75*		45*		25*	
36	TdA(DSR)	Address Valid to $\overline{DS}$ (Read) ↓ Delay	180*		110*		65*	
37	TdC(DSR)	Clock ↑ to $\overline{DS}$ (Read) ↓ Delay		120		85		60
38	TwDSR	$\overline{DS}$ (Read) Width (Low)	275*		185*		110*	

39	TdC(DSW)	Clock ↓ to $\overline{DS}$ (Write) ↓ Delay		95		80		60
40	TwDSW	$\overline{DS}$ (Write) Width (Low)	185*		110		75*	
41	TdDSI(DR)	$\overline{DS}$ (I/O) ↓ to Read Data Required Valid	330		210		120*	
42	TdC(DSf)	Clock ↓ to $\overline{DS}$ (I/O) ↓ Delay		120		90		60
43	TwDS	$\overline{DS}$ (I/O) Width (Low)	410*		255*		160*	
44	TdAS(DSA)	$\overline{AS}$ ↑ to $\overline{DS}$ (Acknowledge) ↓ Delay	1065*		690*		410*	
45	TdC(DSA)	Clock ↑ to $\overline{DS}$ (Acknowledge) ↓ Delay		120		85		65
46	TdDSA(DR)	$\overline{DS}$ (Acknowledge) ↓ to Read Data Required Delay	455		295		165*	
47	TdC(S)	Clock ↑ to Status Valid Delay		110		85		60
48	TdS(AS)	Status Valid to $\overline{AS}$ ↑ Delay	50		30*		10*	
49	TsR(C)	$\overline{RESET}$ to Clock ↑ Setup Time	180		70		50	
50	ThR(C)	$\overline{RESET}$ to Clock ↑ Hold Time	0		0		0	
51	TwNMI	$\overline{NMI}$ Width (Low)	100		70		50	
52	TsNMI(C)	$\overline{NMI}$ to Clock ↑ Setup Time	140		70		50	
53	TsVI(C)	$\overline{VI}$,$\overline{NVI}$ to Clock ↑ Setup Time	110		50		40	
54	ThVI(C)	$\overline{VI}$,$\overline{NVI}$ to Clock ↑ Hold Time	0		20		0	
55	TsSGT(C)	$\overline{SEGT}$ to Clock ↑ Setup Time	70		55		40	
56	ThSGT(C)	$\overline{SEGT}$ to Clock ↑ Hold Time	0		0		0	
57	TsMI(C)	$\overline{MI}$ to Clock ↑ Setup Time	180		110		80	
58	ThMI(C)	$\overline{MI}$ to Clock ↑ Hold Time	0		0		0	
59	TdC(MO)	Clock ↑ to MO Delay		120		85		70
60	TsSTP(C)	$\overline{STOP}$ to Clock ↓ Setup Time	140		80		50	
61	ThSTP(C)	$\overline{STOP}$ to Clock ↓ Hold Time	0		0		0	
62	TsW(C)	$\overline{WAIT}$ to Clock ↓ Setup Time	50		30		20	
63	ThW(C)	$\overline{WAIT}$ to Clock ↓ Hold Time	10		10		5	
64	TsBRQ(C)	$\overline{BUSREQ}$ to Clock ↑ Setup Time	90		80		60	
65	ThBRQ(C)	$\overline{BUSREQ}$ to Clock ↑ Hold Time	10		10		5	
66	TdC(BAKr)	Clock ↑ to $\overline{BUSACK}$ ↑ Delay		100		75		60
67	TdC(BAKf)	Clock ↑ to $\overline{BUSACK}$ ↓ Delay		100		75		60
68	TwA	Address Valid Width	150*		95*		50*	
69	TdDS(S)	$\overline{DS}$ ↑ to STATUS Not Valid	80*		55*		30*	

*Clock-cycle-time-dependent characteristics. See table on following page.

CLOCK-CYCLE-TIME-DEPENDENT CHARACTERISTICS

Number	Symbol	Z8001/Z8002 Equation	Z8001A/Z8002A Equation	Z8001B/Z8002B Equation
11	TdA(DR)	2TcC + TwCh - 130 ns	2TcC + TwCh - 95 ns	2TcC + TwCh - 60 ns
13	TdDS(A)	TwCl - 25 ns	TWCl - 25 ns	TwCl - 20 ns
16	TdDW(DS)	TcC + TwCh - 60 ns	TcC + TwCh - 40 ns	TcC + TwCh - 30 ns
17	TdA(MR)	TwCh - 50 ns	TwCh - 35 ns	TwCh - 20 ns
19	TwMRh	TcC - 40 ns	TcC - 30 ns	TcC - 20 ns
20	TdMR(A)	TwCl - 35 ns	TwCl - 35 ns	TwCl - 20 ns
21	TdDW(DSW)	TwCh - 50 ns	TwCh - 35 ns	TwCh - 25 ns
22	TdMR(DR)	2TcC - 130 ns	2TcC - 100 ns	2TcC - 60 ns
25	TdA(AS)	TwCh - 50 ns	TwCh - 35 ns	TwCh - 20 ns
27	TdAS(DR)	2TcC - 140 ns	2TcC - 110 ns	2TcC - 60 ns
28	TdDS(AS)	TwCl - 35 ns	TwCl - 35 ns	TwCl - 25 ns
29	TwAS	TwCh - 20 ns	TwCh - 15 ns	TwCh - 10 ns
30	TdAS(A)	TwCl - 35 ns	TwCl - 25 ns	TwCl - 20 ns
32	TdAS(DSR)	TwCl - 25 ns	TwCl - 15 ns	TwCl - 10 ns
33	TdDSR(DR)	TcC + TwCh - 150 ns	TcC + TwCh - 105 ns	TcC + TwCh - 70 ns
35	TdDS(DW)	TwCl - 30 ns	TwCl - 25 ns	TwCl - 15 ns
36	TdA(DSR)	TcC - 70 ns	TcC - 55 ns	TcC - 35 ns
38	TwDSR	TcC + TwCh - 80 ns	TcC + TwCh - 50 ns	TcC + TwCh - 30 ns
40	TwDSW	TcC - 65 ns	TcC - 55 ns	TcC - 25 ns
41	TdDSI(DR)	2TcC - 170 ns	2TcC - 120	2TcC - 80 ns
43	TwDS	2TcC - 90 ns	2TcC - 75 ns	2TcC - 40 ns
44	TdAS(DSA)	4TcC + TwCl - 40 ns	4TcC + TwCl - 40 ns	4TcC + TwCl - 30 ns
46	TdDSA(DR)	2TcC + TwCh - 150 ns	2TcC + TwCh - 105 ns	2TcC + TwCh - 75 ns
48	TdS(AS)	TwCh - 55 ns	TwCh - 40 ns	TwCh - 30 ns
68	TwA	TcC - 90 ns	TcC - 70 ns	TcC - 50 ns
69	TdDS(S)	TwCl - 25 ns	TwCl - 15 ns	TwCl - 10 ns

APPENDIX B

Glossary

Access time: The time required to read or write data to a device (memory or a peripheral), measured from when the address of the device is available until when the data are actually read from or written into the device.

Accumulator: A register within a central processing unit (CPU) that can hold the result of an arithmetic or logical operation.

Acknowledge cycle: A CPU machine cycle entered as a response to an interrupt or trap. The Z8000 CPUs read an identifier word from the interrupting device during this cycle.

Address: A number that identifies a particular register, memory location, or peripheral device.

Address space: A set of addresses that are accessed in a similar manner. The Z8000 CPUs can access six memory address spaces (normal-mode program, normal-mode data, normal-mode stack, system-mode program, system-mode data, and system-mode stack) and two I/O address spaces (standard I/O and special I/O).

Addressing modes: The method used to specify the address of an operand within an instruction.

Applications programs: A program designed to do a task other than controlling the resources within a computer system. Applications programs typically run in the normal mode on a Z8000 system.

Arithmetic and logical unit (ALU): The part of the central processing unit that contains the logic for performing arithmetic and logical operations on data.

Assembler: A computer program that translates assembly language code into machine language. An assembler generally translates symbolic codes, such as instruction mnemonics, into the opcodes that are executed by the processor.

Asynchronous: Not related to or dependent on a specific time period or clock frequency; having no fixed relationship in time.

Attribute: A characteristic or feature of a particular entity. The memory manager assigns attributes such as read-only or execute-only to memory segments.

Autodecrement: An operand addressing method wherein the contents of a specified register are decremented and used as the address of an operand during instruction execution.

Autoincrement: An operand addressing method wherein the contents of a specified register are incremented and used as the address of an operand during instruction execution.

Base address: A number that appears as an address in an instruction, but serves as a starting point for an effective address calculation. (*See* base addressing mode, base indexed addressing mode, and indexed addressing mode.)

Base addressing mode: An operand addressing mode wherein the base address is held in a register and the displacement is given in the instruction's opcode. The effective address of the operand is found by adding the displacement to the base address.

Base indexed addressing mode: An operand addressing mode wherein the base address and the displacement are held in registers. The effective address of the operand is found by adding the displacement to the base address.

Bidirectional: Pertaining to a bus structure wherein a single signal line can transmit signals in either direction. For example, the Z8000 CPU's address/data bus pins are bidirectional since data can be transmitted or received by the CPU.

Binary-coded decimal (BCD): A notation in which the 10 decimal digits (0–9) are encoded in 4-bit binary fields. BCD notation often is used to process numbers in base-10 format.

Bit: A binary digit; one unit of data in binary notation. A bit can have one of two values, 0 or 1.

Bus: A group of signal lines that connect devices in a system; a path over which information is transferred.

Bus master: The device in a system that controls the bus. A bus master must be capable of initiating transactions.

Bus request: A request for control of the bus initiated by a device other than the bus master.

Bus transaction: (*See* transaction.)

Byte: A field of eight contiguous bits that is operated on as a unit.

Central processing unit (CPU): The primary functioning unit of a computer, consisting of an ALU, control logic for decoding and executing instructions and controlling program flow, and registers.

Comment: An optional field within a statement in a program that contains an identification or explanation for a particular step in the program, but has no effect on the operation of the computer when the program is being executed. Comments are used for documentation purposes only.

Condition code: A specific Boolean function of the ALU flags tested during the execution of a conditional instruction.

Conditional instruction: An instruction that can take more than one action based on the current condition of the ALU flags.

Context switch: A switch from one programming task to another, usually as the result of an interrupt or trap condition.

Cyclic redundancy code (CRC): A code used to check data integrity during data transmissions. CRCs typically are used in serial communications. The transmitted data bit stream is divided by a polynomial and the remainder transmitted as the check field. The receiver compares the transmitted check field with its own computed remainder to verify that the data received were valid.

Daisy chain: A method of propagating signals along a bus wherein the priority of devices is determined by the physical position of each device on the bus.

Data: Basic elements of information that can be processed by a computer.

Destination: The register, memory location, or device to which data are to be transferred.

Direct address mode: An operand addressing mode wherein the address of the operand is given in the instruction's opcode.

Direct memory access (DMA): A method of accessing individual memory locations without using the CPU. DMA circuits are used in systems that require data transfers at faster rates than those obtained by going through the CPU.

Displacement: A number that is added to a base address during an effective address calculation. (*See* base addressing mode, base indexed addressing mode, indexed addressing mode, and relative addressing mode.)

Distributed processing: A design technique in multiprocessor systems wherein each processor in the system has its own specific task assigned to it.

Dynamic RAM: A random access memory on which a special operation

called a refresh must be performed at periodic intervals to preserve data integrity.

Effective address: The actual address of a data operand. Often, the effective address is calculated during instruction execution by adding a displacement or index to a base address.

Exception condition: A condition that causes the CPU to discontinue the current programming task (perhaps only temporarily), such as an interrupt, trap, or reset.

Extended instruction: A special Z8000 instruction intended for use with Extended Processor Units.

Extended instruction trap: A trap that is caused by attempting to execute an extended instruction in a system without Extended Processor Units.

Extended Processor Unit (EPU): A large-scale integrated-circuit chip that contains a processor to perform dedicated tasks within a Z8000 system.

First-in-first-out (FIFO) buffer: A data buffer in which the data are read from the buffer in the same order as they were written to the buffer. The first element of data to be written to the buffer is the first element to be read out.

Flag: One bit of information used to indicate that some particular condition occurred. For example, the Z8000's zero flag indicates when an ALU operation yields a zero for a result.

Flag and control word (FCW): A register in the Z8000 CPU that contains the ALU flags and control bits that determine the processor's operating modes.

Full duplex: A method of data transmission wherein each end can simultaneously transmit and receive.

Handshake: A sequence of signals that provide a protocol for transferring data between devices. Typically, handshakes are used for asynchronous interfaces where each signal requires a response in order to complete a data transfer.

Identifier word: A status word that is pushed onto the stack when the Z8000 CPU processes an interrupt or trap. For internal events (traps other than the segmentation trap), the identifier word is the first word of the instruction that caused the trap. For external events (interrupts and segmentation traps), the identifier word is the status word read from the interrupting device during the acknowledge cycle.

Immediate addressing mode: An operand addressing mode wherein the operand is given within the instruction's opcode itself.

Implied stack pointer: In Z8000 systems, the stack pointer used when saving program status during exception processing and subroutine calls. R15 is the implied stack pointer for nonsegmented-mode operation; RR14 is the implied stack pointer for segmented-mode operation.

Index: A number used to identify a particular element in an array or table. The index, also called the displacement, is added to the base address to determine the effective address of an operand. (*See* base indexed addressing mode and indexed addressing mode.)

Indexed addressing mode: An operand addressing mode wherein the operand address is calculated by adding the index, which is the contents of a register, to a base address given in the instruction's opcode.

Indirect register addressing mode: An operand addressing mode wherein the address of the operand is the contents of the register specified in the instruction's opcode.

Input: The process of reading data from a peripheral device.

Input/output (I/O) device: (*See* peripheral.)

Input/output (I/O) transaction: A transaction that transfers data between the CPU and a peripheral device.

Instruction: The specification of an operation to be performed by a computer and the operands for that operation.

Instruction fetch: A memory read operation wherein the data read from memory comprise the opcode of an instruction that is to be executed.

Instruction pre-fetch: A processor timing scheme wherein the opcode for the next instruction is fetched while the previous instruction is still being executed.

Instruction set: The set of all instructions that can be executed by a given processor.

Interlocked handshake: A handshake protocol for transferring data between two devices using two control signals. A change in level of one control signal requires the appropriate response on the other signal for the transfer to be completed.

Internal operation cycle: A Z8000 machine cycle during which the processor performs an operation internal to the CPU and no data transfer occurs on the bus.

Interrupt: An event that changes the normal flow of instruction execution due to a signal generated external to the CPU. The flow of instruction execution is broken in a manner that allows it to be resumed from that point at a later time. Interrupts provide a means for peripheral devices to gain the CPU's attention.

Interrupt acknowledge cycle: (*See* acknowledge cycle.)

Interrupt service routine: (*See* service routine.)

Interrupt-under-service daisy chain: A method of determining priority among peripherals sharing an interrupt request line wherein the service routine for a device cannot be interrupted by an interrupt request from a lower-priority device.

Interrupt vector: Data read from the interrupting device during an acknowledge cycle that determine the location of the service routine. (*See* vectored interrupts.)

Label: An optional field within a statement in a program that allows a symbolic name (the label) to be associated with the memory address of the code generated by that statement.

Logical address: A memory address that is manipulated by the programmer, used in instructions, and output by the CPU during program execution.

Long offset address: A segmented address given in an instruction that occupies two words in the instruction's opcode, one word for the segment number and one word for the offset.

Long word: A field of 32 contiguous bits that is operated on as a unit.

Machine cycle: One basic CPU operation involving a single transaction on the bus. For Z8000 CPUs, one machine cycle is the time from the falling edge of $\overline{AS}$ to the next falling edge of $\overline{AS}$.

Machine language: Binary code that can be read directly and used by a computer.

Main memory: Memory within a system that can be directly accessed using memory access cycles. Main memory is typically made up of semiconductor memories such as ROMs and RAMs.

Mask-programmable ROM: Read-only memory whose contents are determined by a photolithographic mask used to manufacture the part.

Memory: A device into which information can be written and then retrieved at a later time; an information storage device.

Memory access time: (*See* access time.)

Memory cycle time: The time between the start of one memory access and the start of the next memory access.

Memory management: The process of controlling memory allocation and protection by mapping physical addresses to logical addresses and performing attribute checking.

Memory manager: The hardware and software in a system that controls memory management.

Memory-mapped I/O: A technique that allows peripheral devices to be accessed as if they were memory locations in main memory.

Memory refresh cycle: A CPU machine cycle dedicated to performing a refresh operation on dynamic RAMs.

Memory segment: (*See* segment.)

Memory transaction: A transaction involving a transfer of data between the CPU and main memory.

Microprocessor: A central processing unit built with large-scale integrated circuits and usually contained on one chip.

Mnemonic: An abbreviation or acronym. Mnemonics often are used to represent assembly language instructions when writing programs.

Modem: Acronym for modulator/demodulator. A modem is a device that converts data from the digital form used by computers to an analog form used for data transmission, and vice versa.

Multiprocessing: Using two or more processors in a computer system.

Multitasking: The ability of a computer system to handle multiple programming tasks simultaneously by overlapping or interleaving their execution, as in a time-sharing system (also called multiprogramming).

Nibble: A field of four contiguous bits operated on as a unit.

Nonmaskable interrupt: An interrupt that cannot be disabled internal to the CPU.

Nonsegmented mode: An operating mode of the Z8000 CPUs in which memory addresses are treated as 16-bit fields. For the Z8001, all memory accesses are made with the same segment number while in nonsegmented mode.

Nonvectored interrupt: An interrupt with only one possible service routine whose location is not dependent on the identifier word read from the interrrupting device. (*See* vectored interrupts.)

Normal mode: An operating mode of the Z8000 CPUs in which certain instructions, called privileged instructions, cannot be executed. Applications programs typically run in normal mode.

Offset: In Z8001 systems, the portion of a memory address that appears on the 16-bit address/data bus during a memory access.

Opcode: Acronym for operation code; an instruction written in machine language. Opcodes are read from memory during instruction fetches.

Operand: An item of data to be operated on during instruction execution.

Operating mode: The method or manner of operation within a Z8000 CPU. The Z8000 CPUs can execute in system mode or normal mode. The Z8001 also can operate in segmented mode or nonsegmented mode.

Operating system: Software in a system dedicated to controlling the system's resources in a manner that permits applications programs to interface with the hardware in an efficient and safe manner. For Z8000 systems, operating system software typically is executed while in system mode.

Output: The process of transferring data to a peripheral device.

Page: A fixed-size block of memory. Memory is divided into pages to facilitate memory management in virtual memory systems.

Peripheral: A device used to read data into or write data out of a computer system. Line printers, CRT terminals, and card readers are all examples of peripherals.

Physical Address: The address used to access a particular memory location; the address seen at the memory device.

PLZ/ASM: A programming language developed at Zilog, Inc., that allows the use of structured programming techniques when writing assembly language programs.

Priority interrupt controller: A device that determines the relative priority for servicing peripherals when they send interrupt requests to the CPU.

Privileged instruction: A Z8000 instruction that cannot be executed in the normal mode. Privileged instructions are instructions that change the processor state or perform I/O transactions.

Privileged instruction trap: A trap that is caused by attempting to execute a privileged instruction during normal-mode operation.

Processor: A device capable of receiving data, performing arithmetic and logical operations on the data, and storing the results. A CPU is a type of processor.

Program: A set of instructions that performs a particular function when executed on a computer.

Program counter (PC): A CPU control register that holds the memory address of the next instruction to be executed.

Program status area: An area of memory in a Z8000 system that holds the values that are loaded into the program status registers in order to execute the appropriate interrupt or trap service routine during exception processing.

Program status area pointer (PSAP): A CPU control register that holds the starting address in memory of the program status area.

Program status registers: The CPU control registers that define the running environment of the processor. For the Z8000 CPUs, the program status registers are the program counter (PC) and flag-and-control word (FCW).

Programming task: One program operating on its data.

Protocol: The rules or conventions used between devices and processes for exchanging information.

Protopak: A type of package for an integrated circuit wherein a socket for another chip is embedded onto the package.

Pulsed handshake: A handshake protocol for transferring data between two devices using two control signals. Long pulses on the two control signals are used to control the data transfer.

Quad word: A field of 64 contiguous bits that is operated on as a unit.

Quasi-static RAM: A dynamic RAM chip that contains its own refresh logic on the integrated circuit chip with the RAM. (Also called pseudo-static RAM.)

Quiescent: Inactive or dormant. For example, a quiescent peripheral device is one that is not asserting an interrupt or undergoing service from the CPU.

Refresh: To restore information that fades away if left alone. For example, dynamic memories must be refreshed periodically in order to retain their contents.

Refresh cycle: (*See* memory refresh cycle.)

Refresh register: A CPU control register whose contents determine if and how often memory refresh cycles occur.

Register: A storage location within a CPU.

Register addressing mode: An operand addressing mode wherein the operand is the contents of the register specified in the instruction's opcode.

Register pair: In the Z8000 CPUs, one of eight pairs of general-purpose word registers; a 32-bit register.

Relative address mode: An operand addressing mode wherein the effective address of the operand is found by adding a displacement given in the instruction's opcode to the current program counter contents.

Request: A signal or message used by a device to indicate the need for some action or resource.

Reset: To return a device to an initial state. For the Z8000 CPUs, a reset operation initializes the program status registers.

Resource: An asset or device within a computer system that can be allocated to a particular task.

Resource request: A request by a particular processor to use a resource shared by several processors in a multiprocessor system.

Secondary storage: Storage devices that are not directly addressable using memory access cycles. Disks and cartridge tapes are examples of secondary storage devices.

Segment: A block of memory that can be assigned common attributes by the memory manager. In Z8001 systems, memory segments can be up to 64K bytes long.

Segmentation: The process of dividing memory into distinct areas, called segments, where each area can be assigned its own attributes and is referred to by its own segment number.

Segmentation trap: A trap that is initiated by the memory manager when a memory violation is detected.

Segmented address: In Z8001 CPUs, a 23-bit memory address that consists of a 7-bit segment number and a 16-bit offset, where the segment number and offset are distinct parts of the memory address.

Segmented mode: An operating mode of the Z8001 CPU in which memory addresses are treated as 23-bit segmented addresses.

Segment number: In Z8001 CPUs, the portion of the memory address that is output on the SN0–SN6 lines during a memory access. Each segment number specifies a particular memory segment.

Semaphore: A storage location used as a Boolean variable to synchronize the use of resources among multiple programming tasks. A semaphore ensures that a shared resource is allocated to only one task at any given time.

Service routine: Program code that is executed in response to an interrupt or trap.

Short offset address: A segmented address given within an instruction that occupies only one word in the instruction's opcode. Short offset addresses can be used to access the first 256 bytes of a memory segment.

Single-chip microcomputer: An entire computer including CPU, memory, and I/O devices on a single integrated-circuit chip.

Source: The register, memory location, or device from which data are being read.

Stack: An area of memory used for temporary storage and subroutine linkages. A stack uses the first-in–last-out method for storing and retrieving data; the last data written onto the stack will be the first data read from the stack.

Stack pointer: A register that holds the address of the top of the stack.

Static RAM: Random access memory that retains its contents without the need for refresh cycles.

Stop request: A request made by activating the $\overline{\text{STOP}}$ line to a Z8000 in order to suspend CPU activity.

Strobed handshake: A handshake protocol for transferring data between two devices using two control signals. Short pulses on the two signals are used to control the data transfer.

Synchronous: Related to or dependent on a specific clock signal; having a fixed relationship in time.

System call trap: A trap that is caused by the execution of a System Call instruction.

System mode: An operating mode of the Z8000 CPUs in which all instructions, including privileged instructions, can be executed. Operating systems software typically runs in system mode.

Task: (*See* programming task.)

Three-wire handshake: A handshake protocol for transferring data between two or more devices using three control signals. A change in level on one signal requires the appropriate response on the other signals for the transfer to be completed.

Time-multiplexed: A bus structure wherein the same signal lines serve more than one purpose at different times. For example, the Z8000 CPU's address/data bus pins hold addresses at some times and data at other times.

Transaction: A basic bus operation involving the transfer of one byte or word of data between the CPU and a memory or peripheral device.

Trap: A condition that occurs at the end of a Z8000 instruction that caused an illegal operation, similar to an interrupt. (*See* extended instruction trap, privileged instruction trap, segmentation trap, and system call trap.)

Tri-state: An output mode of a logic device wherein the output is held in a high-impedance state and does not affect the logic level on the line. Tri-state outputs are useful when several devices are connected to the same signal line but only one device controls the logic level on that line at a given time.

Unidirectional: Pertaining to a bus structure wherein a single conductor transmits signals in only one direction. For example, the Z8000 CPU's status lines are unidirectional since the status information is always a CPU output.

Vector: (*See* interrupt vector.)

Vectored interrupt: An interrupt with several possible service routines. The service routine executed as a result of a particular vectored interrupt request depends on the value of the interrupt vector read from the interrupting device during the acknowledge cycle.

Violation: An error condition detected by the memory manager when an illegal memory access is attempted, such as an attempted write to a memory segment with the read-only attribute.

Virtual memory system: A system in which the logical memory address space is larger than the physical memory address space. In virtual memory systems, secondary storage devices are used as an extension of main memory, thus giving the appearance to the user of a larger main memory area than actually exists.

Wait state: A clock period that is added to a memory or I/O transaction due to an active $\overline{\text{WAIT}}$ signal. Wait states are used to prolong memory and I/O transactions to devices with long access times.

Word: A field of 16 contiguous bits that is operated on as a unit.

Write warning: A condition that is detected by the memory manager when the amount of available space in a stack area of memory goes below a certain limit. Write warnings signal a potential memory allocation problem.

Z-Bus: The logical definition of the set of signals needed to interconnect the components in the Z8000 family within a computer system.

APPENDIX C

Bibliography

LEVENTHAL, LANCE, ADAM OSBORNE, and CHUCK COLLINS, *Z8000 Assembly Language Programming*, Osborne/McGraw-Hill, Berkeley, CA, 1980.

MATEOSIAN, RICHARD, *Programming the Z8000*, Sybex, Inc., Berkeley, CA, 1980.

SIPPL, CHARLES J., and ROGER J. SIPPL, *Computer Dictionary*, Howard W. Sams & Co., Indianapolis, Ind., 1980.

ZARELLA, JOHN, *Operating Systems Concepts and Principles*, Microcomputer Applications, 1979.

ZARELLA, JOHN, *System Architecture*, Microcomputer Applications, 1980.

1981 Data Book, Zilog, Inc., document 00-2034-01, 1981.

Z8000 CPU Technical Manual, Zilog, Inc., document 00-2010-C0, 1981.

Z8000 PLZ/ASM Assembly Language Programming Manual, Zilog, Inc., document 03-3055-02, 1980.

A Small Z8000 System Application Note, Zilog, Inc., document 03-8060-02, 1980.

Z8010 MMU Technical Manual, Zilog, Inc., document 00-2015-A0, 1981.

Z8 Microcomputer Technical Manual, Zilog, Inc., document 03-3047-02, 1977.

Z8 PLZ/ASM Assembly Language Programming Manual, Zilog, Inc., document 03-3023-03, 1980.

Z-UPC Universal Peripheral Controller Technical Manual, Zilog, Inc., document 00-2055-A0, 1980.

Z-UPC Assembly Language Programming Manual, Zilog, Inc., document 03-3145-A0, 1981.

Index

involved in hardware designs that use Z8000 family com-
ware designs for Z8000-based systems, this
Z8000 Microprocessor: A Design Handbook,
addition to your
hardware engineer this book is a guide to interfacing the
memory and peripheral devices. The software engi-
Z8000's architecture and its relation to the micro-

The Z8000 Microprocessor: A Design Handbook,
design with the Z8001 and Z8002 microproc-
family of parts are discussed.

Among its features, the book

- and Z8002 microprocessors from both an architectural
viewpoint;
- architecture concepts such as operating systems
- family components often used in Z8000-based systems
Memory Management Unit, peripherals, memories, and
- overview of the instruction set.

to anyone interested in learning about the Z8000 who has
microprocessors and is familiar with concepts such as
computers, and interrupts.

PRENTICE-HALL, INC., ENGLEWOOD CLIFFS, NEW JERSEY 07632

ISBN 0-13-983734-5