DIRECTORIO DE PROFESORES DEL CURSO INTRODUCCION A LOS MICROPROCESADORES Y SUS APLICACIONES 1979.

ING. LUIS CORDERO BORBOA
Jefe de Laboratorio de Computación
Facultad de Ingeniería
UNAM
México 20, D.F.
Tel. 550.52.15 Ext.3750

M. EN C. MANUESL ESTEVE2 KUBLI
Departamento de Diseño
Centro de Instrumentos
UNAM
México 20, D.F.
Tel.550.52.15 Ext.4195

DR. ADOLFO GUZMAN ARENAS
Investigador
Instituto de Investigaciones en Matemáticas
Aplicadas y en Sistemas
UNAM
México 20, D.F.
Tel.550.52.15 Ext.4584 y 4585

M. EN C. MARCIAL PORTILLAROBERTSON (Coordinador)
Jefe de la Sección de Computación
Edificio de Ing. Mecánica y Eléctrica
Facultad de Ingeniería
UNAM
México 20, D.F.
Tel.550.52.15 Ext.3746

ING. DANIEL RIOS ZERTUCHE
Jefe del Porte Técnico
Coordinación de Informática
S.H.C.P.
Palacio Nal. Edif. 6-2°
México 1, D.F.
Tel.522.37.81

M. EN C. ARMANDO TORRES FENTANES
Investigador
Coordinador de Sistemas y Circuitos Electromecánicos
Depto. de Ing. Mec. y Eléctrica
Facultad de Ingeniería
UNAM
México 20, D.F.
Tel. 50.52.15 Ext.3746

INTRODUCCION A LOS MICROPROCESADORES Y SUS APLICACIONES

Fecha Fecha	Duración	Tema	Profesor
Agosto 10	17 a 21 h	CONCEPTOS BASICOS	M. EN C. MARCIAL PORTILLA ROBERTSON
Agosto 11	9 a 14 h	ESTRUCTURAS FUNDAMENTAL ES SEÑALES DE CONTROL	M. EN C. MARCIAL PORTILLA
Agosto 17	17 a 21 h	MODOS DE DIRECCIONAMIENTO	ING. LUIS CORDERO BORBOA
Agosto 18	9 a 14 h	CONJUNTO BASICO DE INSTRUCCION	M. EN C. MANUEL ESTEVEZ KUBLI
Agosto 24	17 a 21 h	PROGRAMA DE ENTRADAS Y SALIDAS E INTERFAZ	ING . DAN IEL RIOS ZERTUCHE
Agosto 25	9 a 14 h	ENSAMBLADOR Y PROGRAMACION	M. EN C. ARMANDO TORRES FENTANES
Agosto 31	17 a 21 h	PRACTICAS DE LABORATO RIO (CU)	M. EN C. ARMANDO TORRES
Septiembre 1°	9 a 14 h	PRACTICA LABORATORIO CU	M. EN C. MANUEL ESTEVEZ
Septiembre 7	17 a 21 h	APLICACIONES	DR. ADOLFO GUZMAN ARENA
Septiembre 8	9 a 14 h	PRACTICA LABORATORIO (CU)	ING. DANIEL RIOS ZERTUCHE

edcs.

centro de educación continua
división de estudios superiores
facultad de ingeniería, unam

INTRODUCCION A LOS MICROPROCESADORES Y SUS APLICACIONES

DISEÑO Y APLICACIONES DE MICROPROCESADORES

AGOSTO, 1979.

CHAPTER 1 OVERVIEW

Although the first microprocessor, the Intel 4004, appeared six years ago, the microprocessor field still seems new and exciting. The rapid advances in microprocessor technology, the creativity of the software designers, and the explosion of microprocessor applications have all contributed to this image. In the popular mind, the computer on a chip has been realized, and the availability of computing power to everyone is at hand. While this is not yet the case, it is true that few of us will long escape the impact of microprocessors. Microprocessors are scheduled for use in most automobiles in the late 70's and early 1980's, and the market for television modules is just now developing. For engineers and scientists, microprocessors will increasingly be employed in instruments, terminals, and a wide variety of military and industrial control and communication systems.

Components - Not Computers (A viewpoint)

There is a wide spectrum of views on microprocessors due in part to the wide variety of available microprocessors. The word "microcomputer" is frequently used to emphasize that the basic operation of these devices and their associated systems is nearly identical to that of conventional computers.* [1] On the other hand digital designers often view microprocessors as convenient replacements for hardwired digital subsystems. One should note that microprocessors are products of the components industry and not of the computer industry.

---

1. In this text, "microprocessor" will be used when describing the central processing unit of a "microprocessor system" (or "microcomputer").

Components manufacturers prefer to sell items in large quantities at low costs. Thus the types of processors that are available and that will become available are largely determined by a manufacturers' predictions about markets where large numbers of processors can be sold. Small users and hobbyists benefit from the "fallout" - the sheer availability of the components, but generally, they do not play an important role in the design process. Gordon Moore of Intel stated that: in its simplest terms, a microprocessor can be considered a "standard logic block whose particular functions can be determined after fabrication by software routines which the logic block would execute".[2] This does not imply that microprocessors cannot be made into sophisticated computing systems. The computer hobbyists have shown that this can be done. (A large fraction of the 25,000 hobbyist systems produced in 1977 by various companies (MITS, IMS, and Intel for example) will be purchased by industrial companies.) The amateurs may very well lead the way as amateurs radio operators once pioneered new techniques in radio communications. But, microprocessors are still most suitable for applications where they will be used in large numbers.

Head of the Family

Almost in parallel with the development of microprocessors, manufacturers developed ancillary chips that unburden the processor and simplify the software. In many cases, these chips rival the complexity of the processor; they are programmable and thus can be configured by the processor under software control to perform a particular task. Example are: parallel input and output chips, serial interface chips, disk interface chips, and number crunchers. Motorola developed its family chips together with the M6800 and described them all as a

---

2. Proc. of the IEEE, June, 1976.

"Microprocessor Family". Later, Intel developed its systems components for the 8080. The designer can expect a continuing growth in the family size for most microprocessors since the capabilities of the systems can be increased without moving to another processor and it concommitant software investments.

Rather special memory components have also been developed for microprocessors. These chips may have multiple chip selects that simplify the address decoding and may be organized so that a minimum number of chips can be used. The ubiquitous 2102 is organized as a 1024 by 1 bit chip (1024 words, 1 bit long), and a minimum memory for an 8-bit processor would require eight chips. This memory would have 1Kbyte capacity and might be too large for many microprocessor applications. Instead, one could use a Motorola 6810 which, in one chip, has 128 bytes. (In most microprocessor applications, the program is stored in read-only-memory and not in read/write memory.) Ultraviolet erasable read-only-memory (ROM) is also available for proto- type systems.

Finally, read/write and ROM memories have been combined with input/output circuits on the same chip. Obviously, the manufacturers are trying to provide chips that will simplify the design of microprocessor systems by minimizing the number of chips. Further, the load on the address and data buses is reduced and the overall reliability increased.

Computers are Smarter than Programmers

Software remains a costly feature of the design process for a microprocessor system. There are two reasons for this. First, microprocessors have a limited instructions set and limited addressing modes. The programmer simply must work harder and more carefully. Second, in contrast to computer manufacturers,

component manufacturers have been slow to develop comprehensive software aids because that software associated with development systems is so expensive compared to the component cost. Various estimates ranging from $20 -$40 per assembly language instruction have been made for typical programming tasks. This is to be contrasted with the $5 - $10 cost for the microprocessor.

Manufacturers have devloped some aids however. These include: cross assemblers, compilers, development systems with resident assemblers and debug aids, and software packages that allow the user to program entirely in "macros". The neophyte is frequently surprised and dismayed at the high costs of a development system, especially if he had been looking for an inexpensive substitute for a minicomputer. A designer can expect (or at least hope) that considerable progress in software for microprocessors will be made. Because micros will impact on many who otherwise would never touch a computer, there will be a continuing demand for better software, especially software that is transportable from processor to processor and that does not involve a fundamental understanding of the details of the processor architecture.

Challenges to the Designer

If only a few systems are needed for an application, a designer should probably select one of the single board systems (Intel 80/10, for example) containing the processor and slots for ROM, read/write memory, and input/output. Essentially the designer face the same problems (not challenges) as the hobbyist, and the relative cost of each system will be high.

The far more challenging task is the design for an application that will involve many processors (i.e., many identical microprocessor systems to be used

in exactly the same way). Here the designer must exersize careful judgement is the selection of the processor, memory, programmable input/output, board design, development system, etc. The large number of systems justifies considerable attention to the minimization of costs ,to testing and reliability, and to documentation. Since the same software will be used, the designer can try to optimize the software so that hardware costs can be minimized (e.g. memory size) through various tradeoffs in software. In this latter case, a designer should completely understand the components of the system and be able to cope with the software.

In this tutorial, we will attempt to study some of the fundamental ideas involved in microprocessor system design and to review the basic characteristics of microprocessor system components.

CHAPTER 2 BASIC MICROPROCESSOR ARCHITECTURE AND OPERATION

To illustrate the basic organization of a microprocessor, we will use the model shown in Fig. 2-1. This model resembles the 650X series of MOS Technology and the 6800 of Motorola; certain registers are not shown for clarity however. The sixteen address lines permit access to 65,536 memory location, although relatively few applications will require such a large memory. These locations contain the program, data storage, and input/output registers. Some of the newer and simpler microprocessors discussed in Chp. 5 have only eight address lines but time multiplex the sixteen address bits. Other processors time multiplex the data bus to obtain the full address information, and still others simply have fewer than sixteen address bits. These various schemes permit the use of other pins on the processor package for input/output, for extra control signals, or to permit a smaller package. The operation of some of the internal registers will be discussed in the following sections and described in detail at the end of the chapter where a specific processor is analyzed.

The operation of this processor is illustrated in Fig. 2-2. The program is stored in a sequence of memory locations in the read-only-memory (ROM). When the RESET signal is toggled low and then high, the contents of the program counter point to a particular memory location which must contain the first instruction (location 0 in this example). The first instruction is fetched when the contents of the PC flow on to the address bus to select location 0. The contents of that location, the op code, is brought into the instruction register of the processor, where the op code is decoded. The PC is automatically incremented so that its contents point to location 1. If only one byte is needed for the instruction, the instruction is executed. If more than one byte is needed,

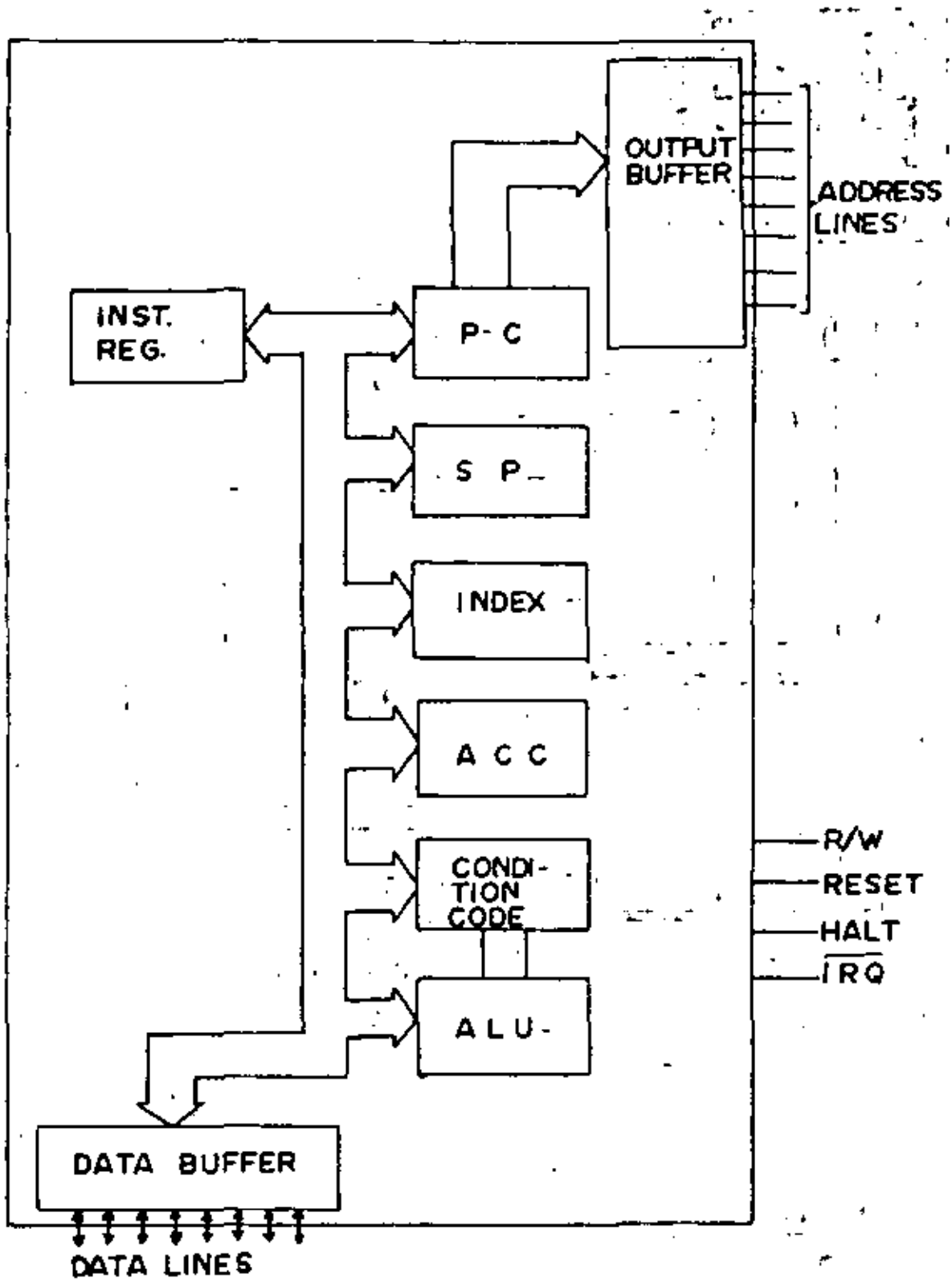

OUTPUT BUFFER
ADDRESS LINES
INST. REG.
P-C
S P
INDEX
A C C
CONDI- TION CODE
A L U
R/W
RESET
HALT
IRQ
DATA BUFFER
DATA LINES

Fig. 2-1

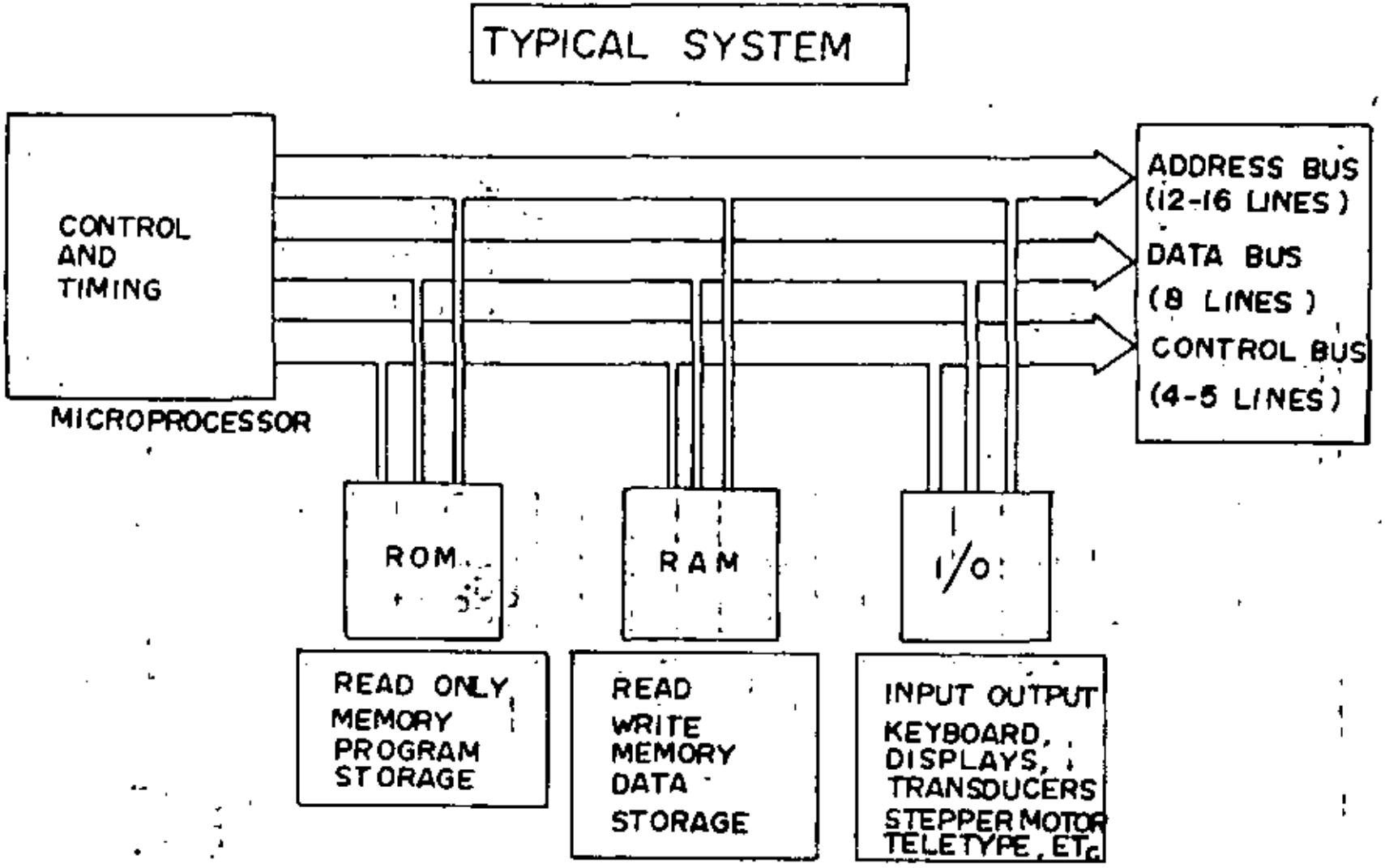

TYPICAL SYSTEM
CONTROL AND TIMING
MICROPROCESSOR
ADDRESS BUS (12-16 LINES)
DATA BUS (8 LINES)
CONTROL BUS (4-5 LINES)
ROM
RAM
I/O
READ ONLY MEMORY PROGRAM STORAGE
READ WRITE MEMORY DATA STORAGE
INPUT OUTPUT KEYBOARD, DISPLAYS, TRANSDUCERS STEPPER MOTOR TELETYPE, ETC.

Fig. 2-2

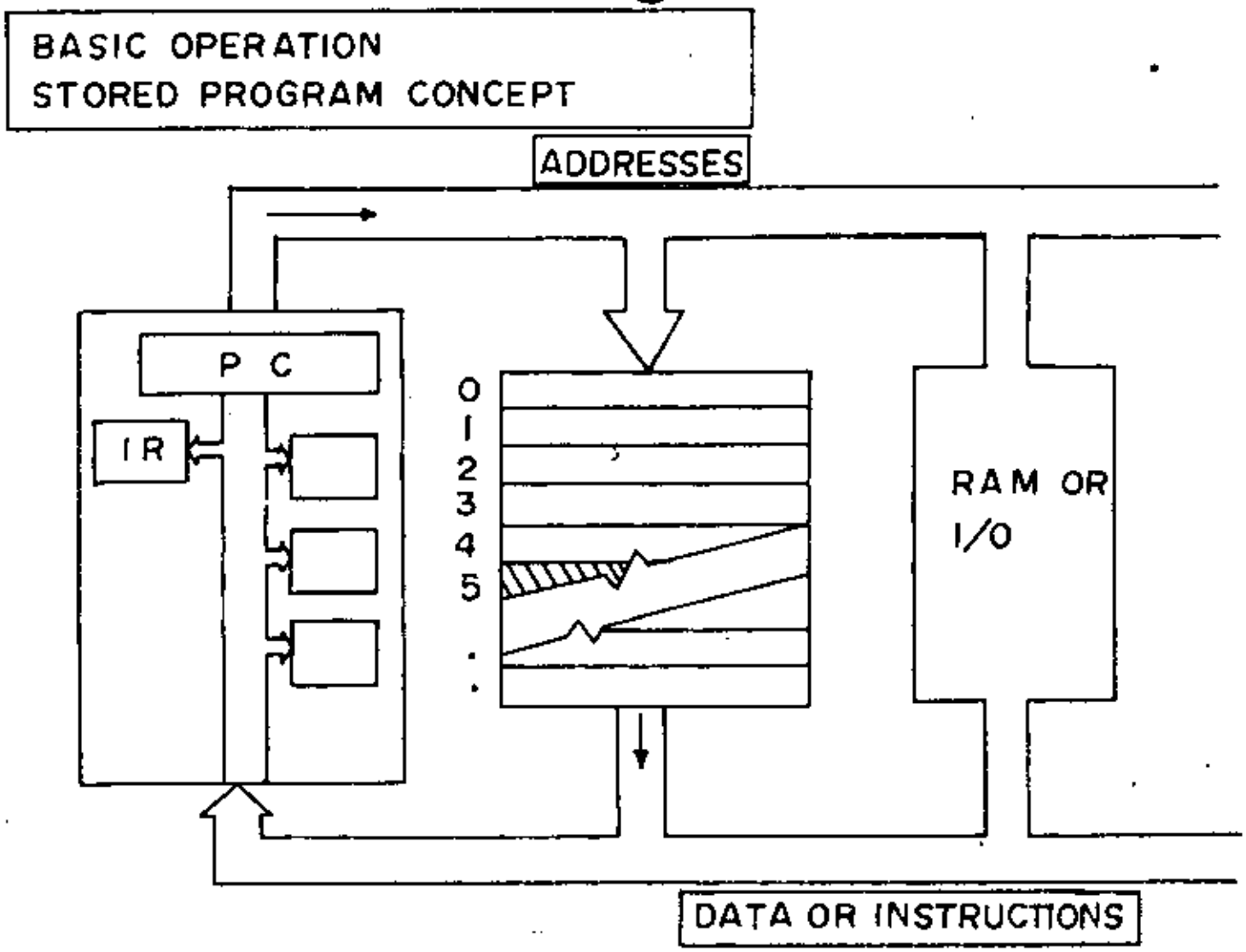

BASIC OPERATION
STORED PROGRAM CONCEPT
ADDRESSES
P C
I R
0
1
2
3
4
5
RAM OR
I/O
DATA OR INSTRUCTIONS

Fig. 2-2a

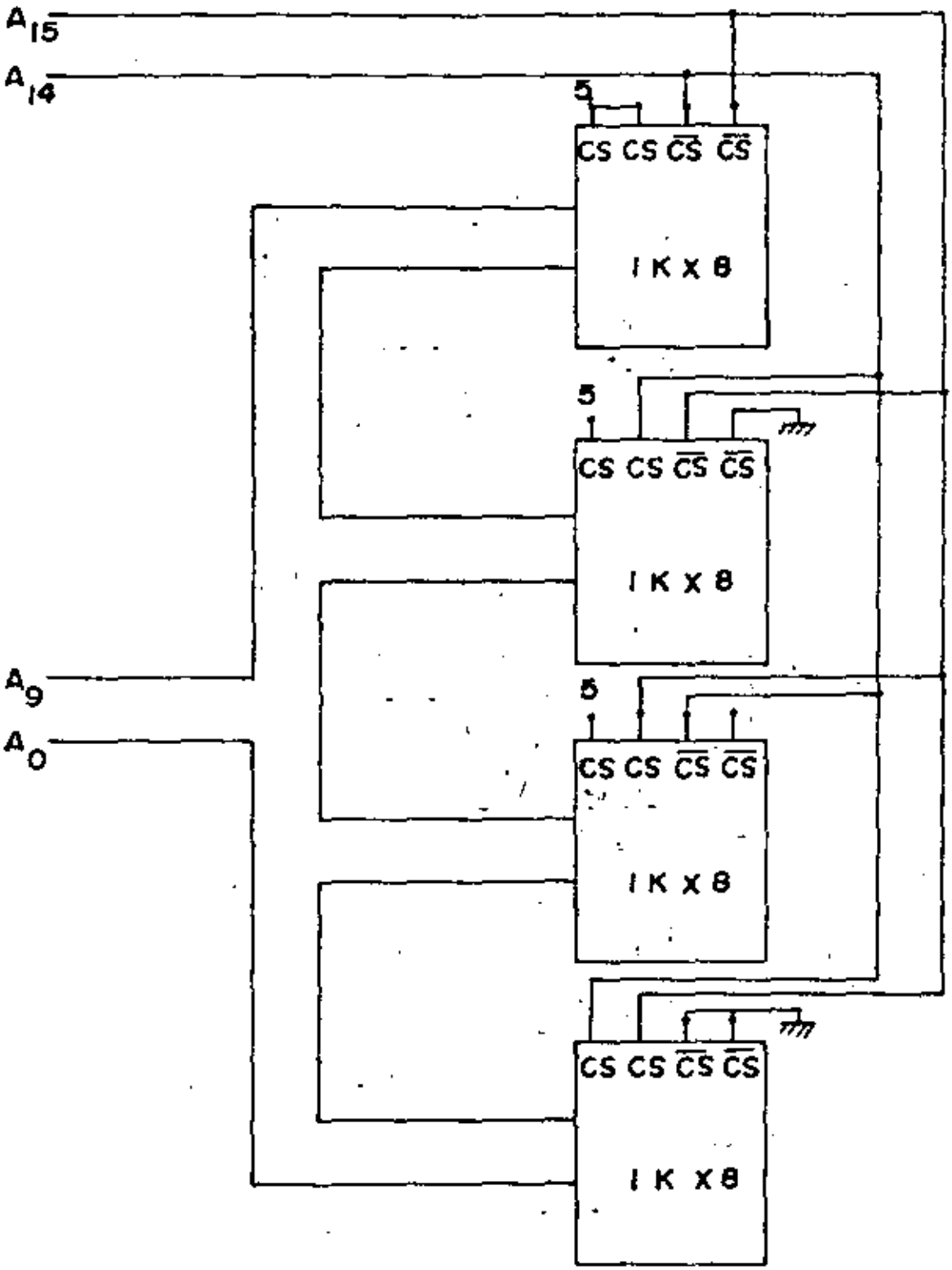
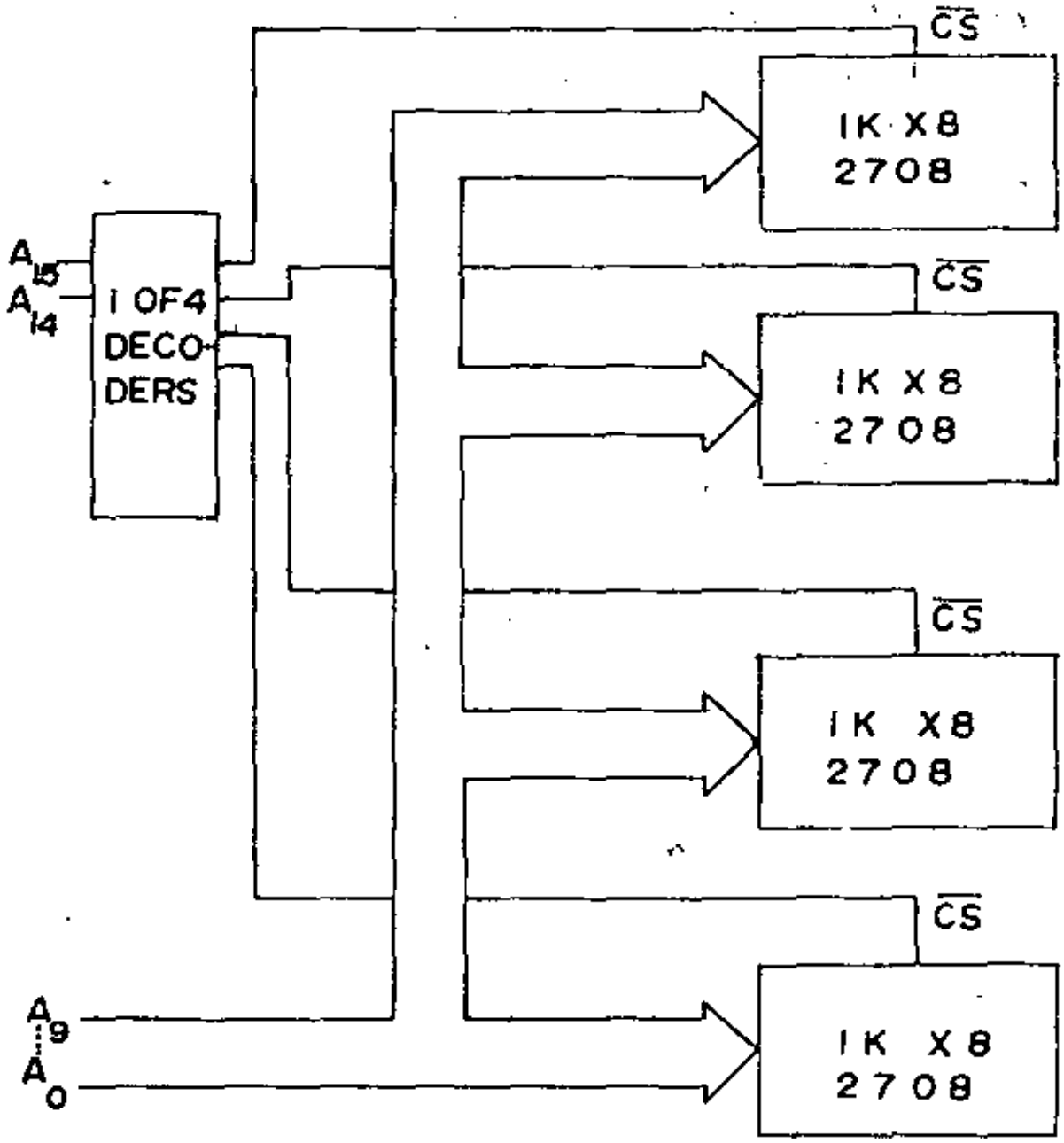

A15
A14
A9
A0
5
CS CS CS CS
1K X 8
5
CS CS CS CS
1K X 8
5
CS CS CS CS
1K X 8
CS CS CS CS
1K X 8

Fig. 2-3a



MEMORY MAP

A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0	
0	0	•	•	•	•	X	X	X	X	X	X	X	X	X	X	0000-03FF
0	1	•	•	•	•	X	X							X	X	0400-43FF
1	0	•	•	•	•	X	X									8000-83FF
1	1	•	•	•	•	X	X							X	X	C000-C3FF

• - DON'T CARE

Fig. 2-3b

the additional byte(s) is fetched before the instruction is executed. The additional byte is stored in the location(s) following the opcode, and as each is fetched, the PC is incremented so that at the end of the instruction, it points to the next instruction. The processor continues fetching and executing instructions sequentially until it is told to halt or until the PC is modified by an instruction so that it jumps to another location in the program. This operation is identical to that of nearly any other computer. The microprocessor is distinguished from conventional computers by some of the details of its operation. These details are discussed in the rest of the chapter and summarized at the end.

Address Decoding

Typically, some of the address lines are used for decoding; i.e., the selection of the appropriate chip(s) or I/O device. Two possible schemes are shown in Fig. 2-3 for two systems with 4K (4096) bytes of memory consisting of four 1K byte ROMs (2708). In Fig. 2-3a, the upper address lines are connected to a decoder chip which provides a 0 or LOW on only one of its output lines and thereby selects only one of the ROMs. In Fig. 2-3b, each memory chip is endowed with multiple chip selects (Motorola's family chips) and the address decoding is accomplished directly. This latter scheme is very useful if only a few chips will be used in the system. For larger memory systems, additional chips for address decoding are necessary.

The examples shown are called "limited address decoding" since the addresses of the memory locations in a given chip are not unique. Since bits 10 to 13 are not used in the decoding or selection of the chips, they can be arbitrarily set without affecting the operation of the system. In contrast,

minicomputers are "fully decoded", hence if the processor references locations where memory does not exist, a "trap" to a special program for such errors usually takes place. Generally, in microprocessor systems the processor may read and write in non-existent memory without impunity or it may address the same physical location through multiple addresses. Thus microprocessor programmers must be more careful because no "hand shaking" occurs between processor and memory to assure the processor that the memory exists or that the location is uniquely defined.

Memory Mapped vs. Isolated Input/Output

In most microprocessor systems, the input and output registers and their associated control and status registers are seen as memory locations by the processor. Thus the input and output registers and the ordinary memory locations occupy the same address space (i.e., they fall within the 65,536 locations addressable by a sixteen bit address bus of a typical processor). This scheme is called memory mapped I/O and is illustrated in Fig. 2-4. The advantages of this scheme is that one can apply any of the memory reference instructions to the input/output. Thus to create a pulse at an output bit, one can simply use the INC (increment) and DEC (decrement) commands which ordinarily increment and decrement memory locations. Some processors have separate I/O instructions; the 8080 is a good example. The 8080 generates input and output signals separate from the memory read and write signals. The input and output signals essentially are read and write signals for the I/O registers selected by eight of the address lines. The advantages of isolated I/O are that less address decoding is necessary and that the full address space can be used for memory. In isolated I/O, special instructions such as IN and OUT are used; these place the contents of the accumulator at the output register or place the contents of the input

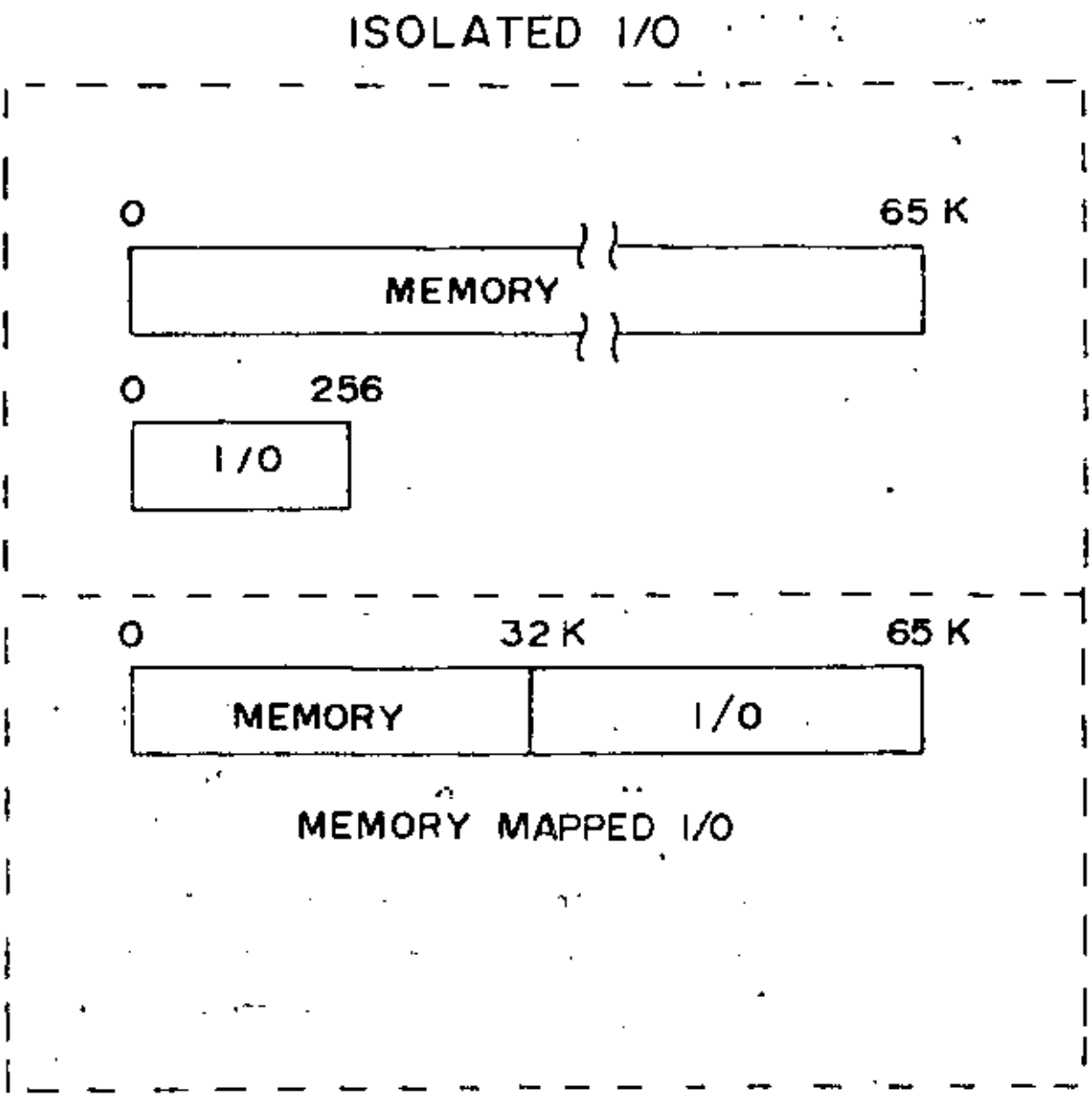
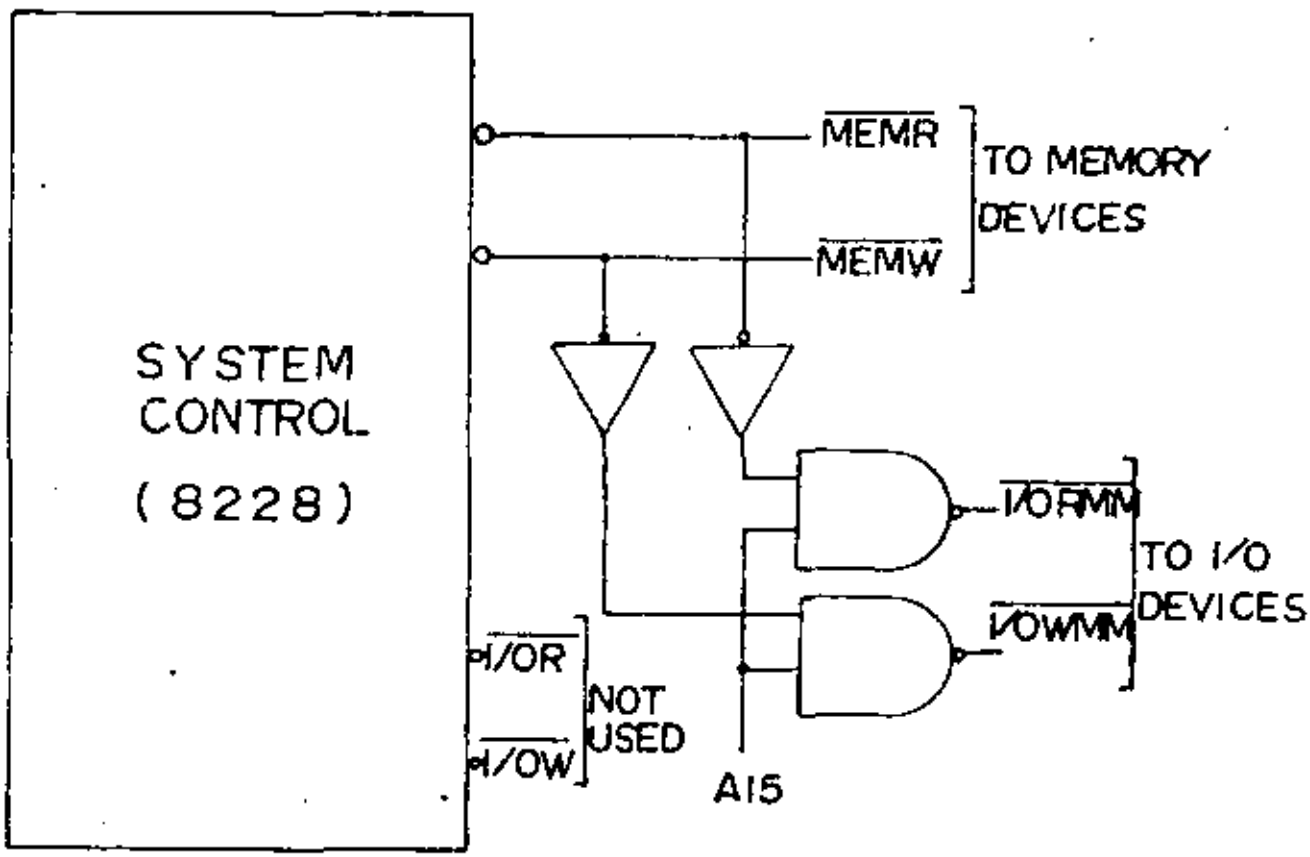

ISOLATED I/O
0
65 K
MEMORY
0
256
I/O
0
32 K
65 K
MEMORY
I/O
MEMORY MAPPED I/O

Fig. 2-4

register in the accumulator. The 8080 is easily converted to memory mapped as shown in Fig. 2-5 and as discussed in Intel's "8080 Microcomputer Systems User's Manual."

Isolated I/O is used in some of the newer processors such as Intel's 8048 and 8085. The 8085 is a later version of the 8080 chip which helps to minimize the chip count while keeping software compatibility. The 8080 is really a three chip processor because it requires a 8228 data bus chip to capture the status and a 8224 timing chip to generate the clock signals and synchronize control signals. A typical 8085 system is shown in Fig. 2-6, ; the 8085 will be used in addition to the 8080 for illustrations and examples. The 8048, a true microcomputer on a chip, is discussed later in Chp. 5 as one of the best, simpler processors. All of the Intel 80XX processors can be made compatible with simple logic circuits, and even other, quite distinct processors such as the 6800 or 6505 can be interfaced with Intel-like circuits.

A simple microprocessor system based on the MOS Technology 6502 is shown in Fig. 2-7 to illustrate address decoding and memory mapped I/O. A simple 8080 system is included in Appendix A along with other operation details. These simple examples show that the number of chips required to fabricate a working system is quite small. Yet quite sophisticated operations may be performed by these systems. In addition, these systems provide excellent I/O capabilities, typical of most microprocessor systems. In the 8085 system, the 8755 provides 2K bytes of ROM in addition to the I/O ports ad the 8155 provides 256 bytes of RAM. Both the ROM and RAM capacities are suitable for a large number of applications. Note that limited address decoding is provided by the 11th and 12th bits of the address bus. These systems require only a few resistors and capaci-



I/O DEVICES ARE SELECTED
WHEN A15=1

Fig. 2-5

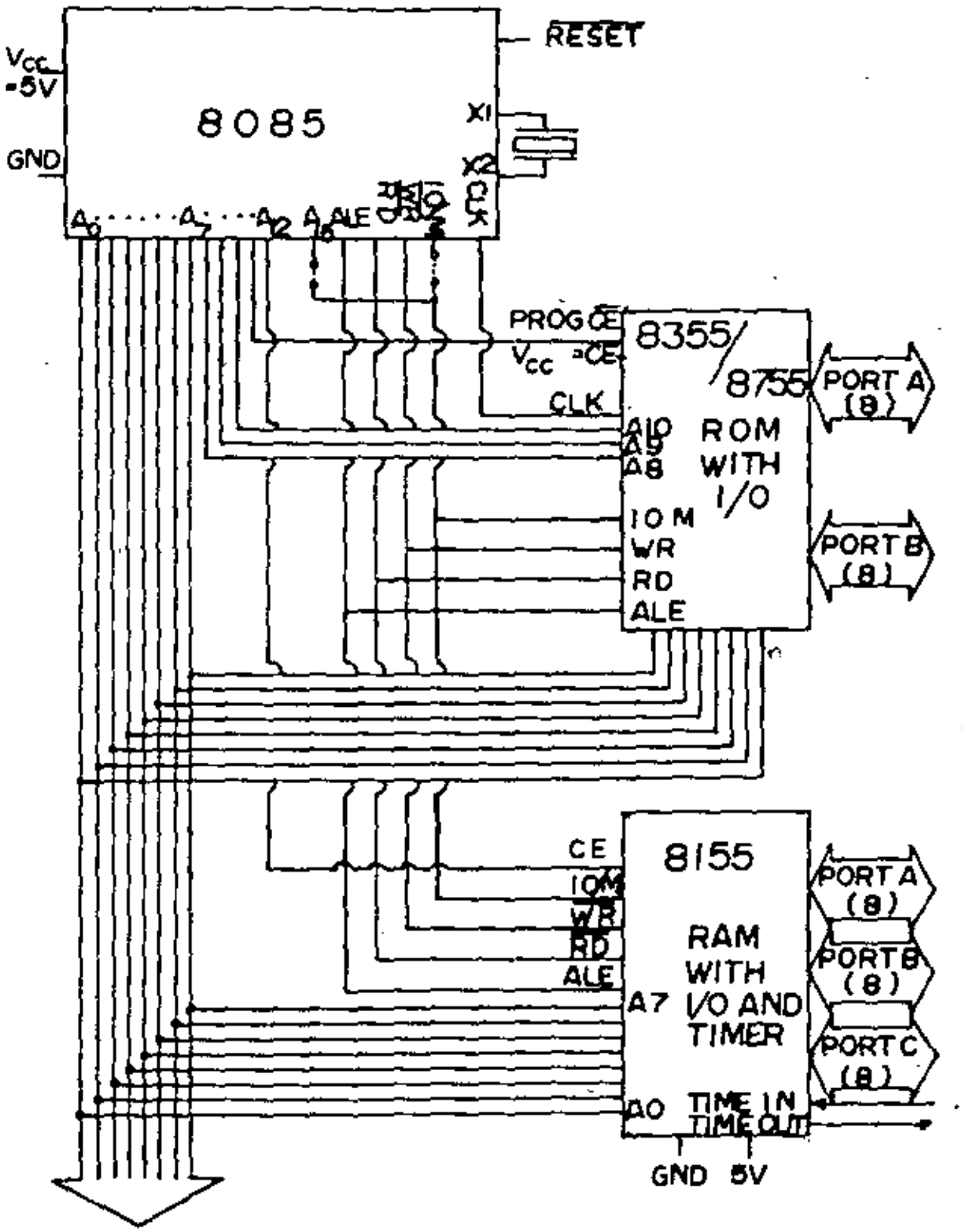
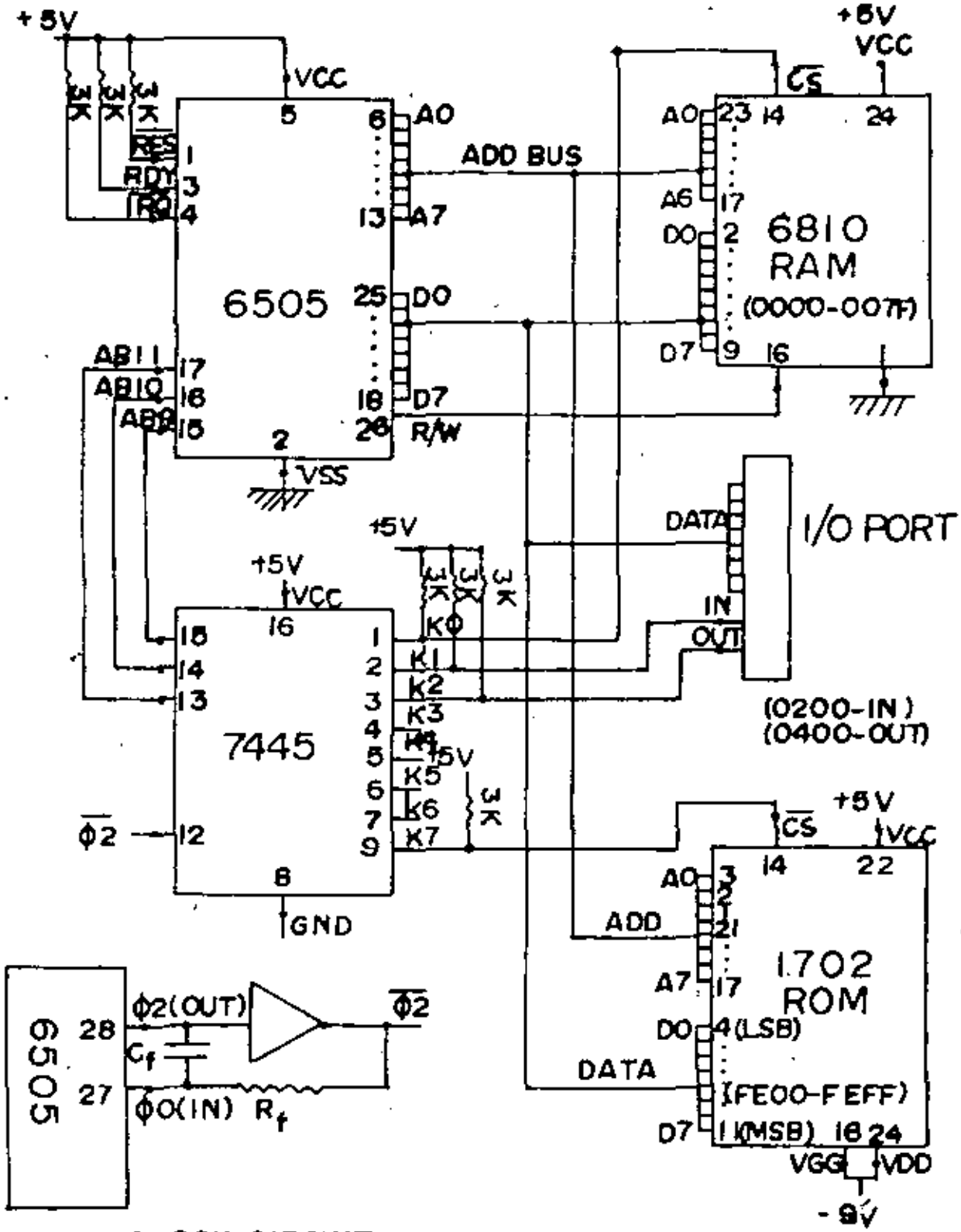

8085
RESET
Vcc
+5V
GND
X1
X2
CLK
A0
A7
A2
A8
ALE
IO/M
PROG CE
Vcc
CE
CLK
8355/
8755
A10
A9
A8
ROM
WITH
I/O
IO M
WR
RD
ALE
PORT A
(8)
PORT B
(8)
8155
CE
IO/M
WR
RD
ALE
A7
RAM
WITH
I/O AND
TIMER
A0
TIME IN
TIME OUT
PORT A
(8)
PORT B
(8)
PORT C
(8)
GND
5V

Fig. 2-6



CLOCK CIRCUIT

Fig. 2-7

tors in addition to the chips and crystal shown to fabricate a working system.

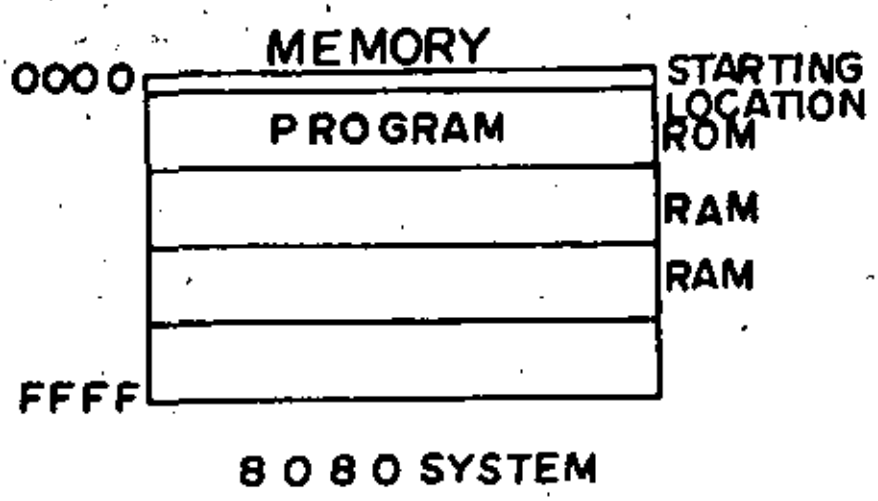
6505 systems uses a 1702 ROM with 256 bytes and a Motorola 6810 with 128 bytes. Address decoding is provided by the 7445 decoder chip. Again limited address decoding is used. The 6505 has only 12 address pins, and thus it can address only 4K bytes; but this capacity is very adequate for most systems. The 650X series from MOS Technology resembles the M6800. In some cases, a considerable simplification of the interfacing requirements has been achieved. Note that a very simple clock circuit is used and that the chip select of memory or I/O occurs with phase two of the clock. The system shown is probably the least expensive microprocessor system that can be assembled. It fits on wiring strips such as the Super-Strip and costs less than $30 is single quantities.

Restart Locations

Upon a RESET signal, the program counter of a processor is set to a specific state. In the 8080, the PC is cleared so that it points to location 0. The first instruction of the program must be in that location. Since the program is stored in ROM for nonvolatility (retains data without power), the ROM is usually located in the lower memory locations. In the 6800, the PC is loaded with the contents of location FFFE and location FFFF on RESET, thus the ROM in a 6800 system is usually located at the top of memory. In addition, the 6800 has addressing modes which makes special use of the lowest memory locations, and thus read/write memory (RAM) usually starts a location 0. (Fig. 2-8).

The Data Bus

The data bus of most processors is eight bits wide; a few such as the SP 1000 and the TI 9900 are sixteen bits wide. Eight bits permit the


MEMORY
0000
PROGRAM
STARTING LOCATION ROM
RAM
RAM
FFFF
8080 SYSTEM

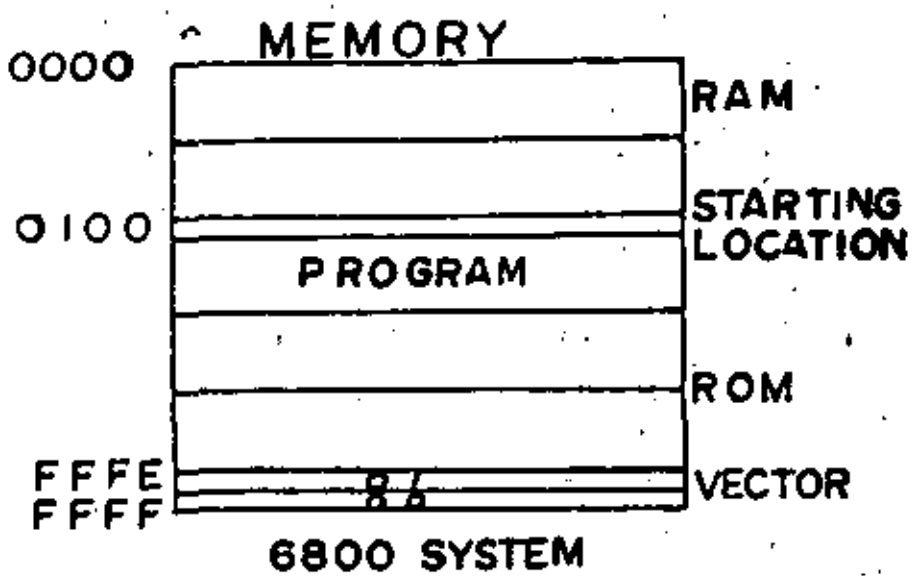

MEMORY
0000
RAM
0100
STARTING LOCATION
PROGRAM
ROM
FFFE
FFFF
VECTOR
6800 SYSTEM

Fig. 2-8

presentation of all of the alphanumeric characters; a byte is a natural unit for communication purposes. Most 8-bit processors have limited instruction sets because of the width limitation, the Z-80 perhaps is an exception. The 8080 uses 244 of the possible 256 op codes, the 6800 uses 195, and the 8048 uses 239 . In contrast to the address bus, the data bus is bidirectional, hence data can be read from or written into memory via the same lines.

In a typical system, the outputs of several memory or interface chips are connected to the data bus; however, only one device can place its contents on the bus at any given time. Thus most microprocessor system chips have "tri-state" or high impedance outputs. Unless a chip is selected by the address decoding process, its outputs remain in the off or floating state and do not affect the output. When selected, the outputs are either the 1 or 0 state.

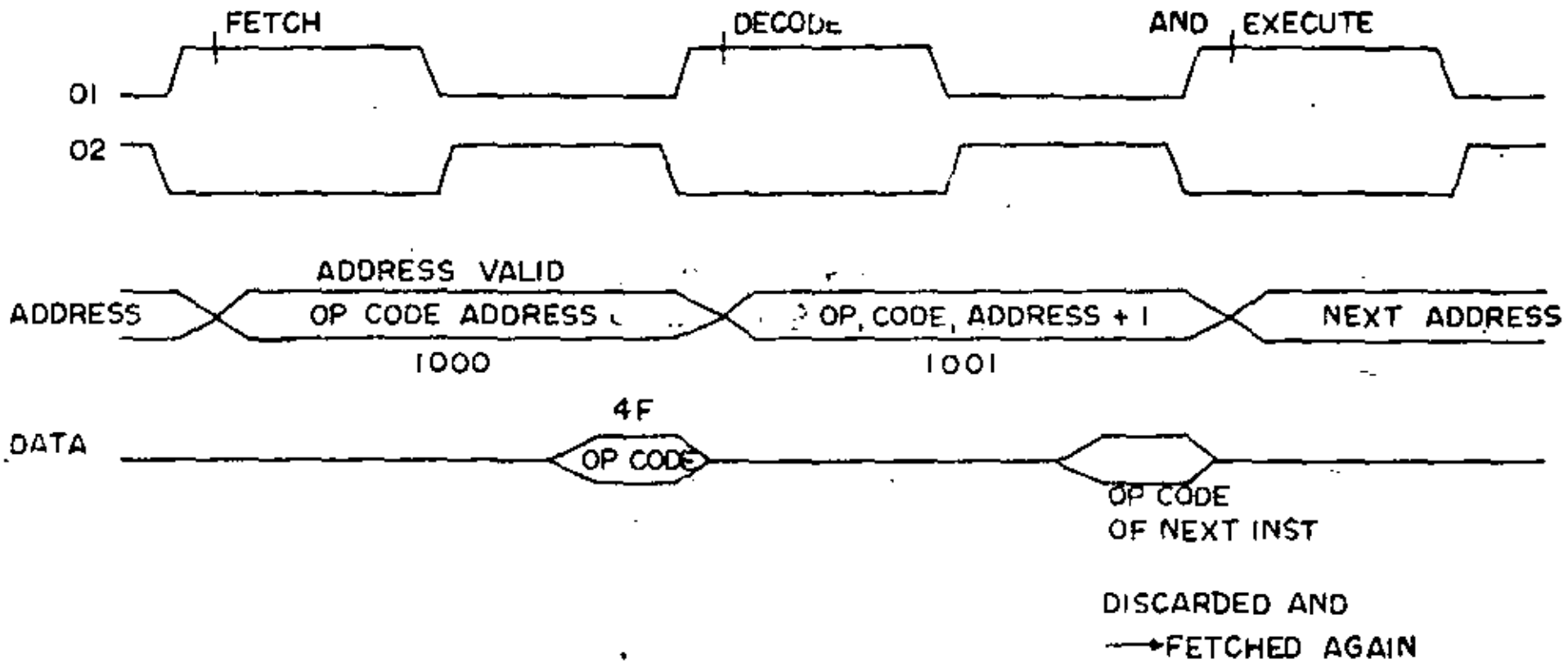
The data and address buses can drive only a limited number of chips. A rule of thumb is that ten family chips may be connected without buffering. Each memory or I/O device places a capacitance on the bus a permits a certain leakage current to flow. The address and data buses must be able to charge the parasitic capacitance of the wires and of the devices and must provide the leakage current in order for the processor to work at its rated speed. Special chips are available to buffer the buses. Intel designed the 8228 chip usually used with the 8080 so that it would buffer the data bus in addition to its other tasks. The data bus is the more difficult because it is bidirectional.

Timing and Clocks

Microprocessor operation is controlled by clock (oscillator) signals. In some cases, the oscillator is on the chip, and external components such a

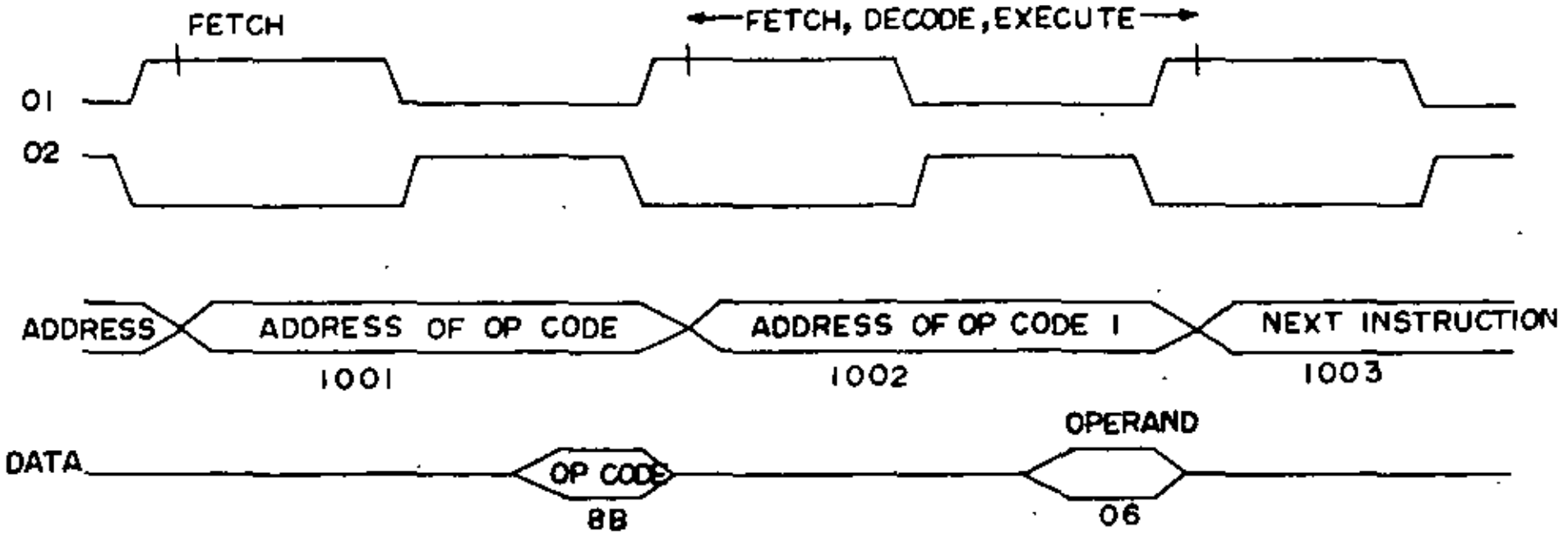
crystals or resistances and capacitances can be used to set the clock rate. In other cases, the clock signals must be provided by external signals. In the cases of the 6800 and the 8080, this is no mean task. There are restrictions on the voltage levels, risetimes, falltimes, overlap, etc. While the 6800 requires a 5 volt clock, it is not TTL compatible. The 8080 requires a high level clock (12 volts) and to achieve maximum speed, a difficult timing sequence (solved by the 8224).

Microprocessors have a variety of clock types: single phase, single phase static clocks, two phase clocks, etc. We will use the two phase clock as an illustration (6800 and 650X). The sequence is quite simple. During phase 1, address information is placed on the address bus to select some memory location or device. Control signals, discussed in a later section, are also set to establish a read or write condition. At the end of phase 2, the contents of the memory location are available to the processor in a read operation or the processor should place the oppropriate code on the data bus in a write operation. Thus phase one is associated with address information and phase two with data information. These operations are illustrated in Fig. 2-9. Referring to this figure and to our model processor, the first byte of the instruction, the op-code, is received at the end of phase two and directed into the instruction register where it is decoded to determine what operation is to be performed. Since the decoding will take some time and since the next byte in memory may be needed, that second byte is fetched in the next cycle. At the end of phase two of the second cycle, the next byte is available and the operation is now decoded. If only one byte is required for the instruction, the processor discards the second byte and executes the instruction. If two bytes were required, that second byte is now available. These cases are shown in Fig. 2-10. The execution of the instruction in most processors overlaps the fetching of the next



ONE BYTE INSTRUCTION
(CLEAR ACCUMULATOR)

Fig. 2-9



2 BYTE INSTRUCTION
(ADD 6 TO ACCB)

Fig. 2-10

instruction; the obviously save time and is called "pipelining". All of the ecoding of the op-code and execution of an instruction requires operations between internal registers in the processor. The nonoverlapping two phase clock provides four "edges" (rising and falling transitions) that clock or toggle these internal registers in addition to other timing tasks. If only a single phase clock is used, more clock cycles will be required to provide the "edges". An example is shown in Fig. 2-11 for the Intel 8085. Here, the data bus and the lower byte of the address bus share the same pins. The lower address bits are sent out first and must by latched using the ALE as the enabling pulse on D-type latches. Later, the data flows in or out of the processor in the normal manner. While this may seem cumbersome, the complete 8085 system is much simpler than a corresponding 8080 system.

The 6800 and 650X microprocessors have the very simple two phase clocks just discussed. The 8080 also has a two phase clock, but its operations are very much more complicated. A more or less complete discussion is included in Appendix A.

Clock Rates

The clock rate alone is not an indication of the speed of a processor. It is frequently and incorrectly stated that the 8080 is twice as fast as the 6800 because the former's clock rate is twice as fast as the latter's. In fact, the 8080 requires more clock cycles to accomplish the same task, and the two processors have very similar speeds. [3] Of course, for a given processor, a higher clock rate yields faster execution.

---

3. Clock rates are only one of many characteristics that are incorrectly compared even in the most recent literature.

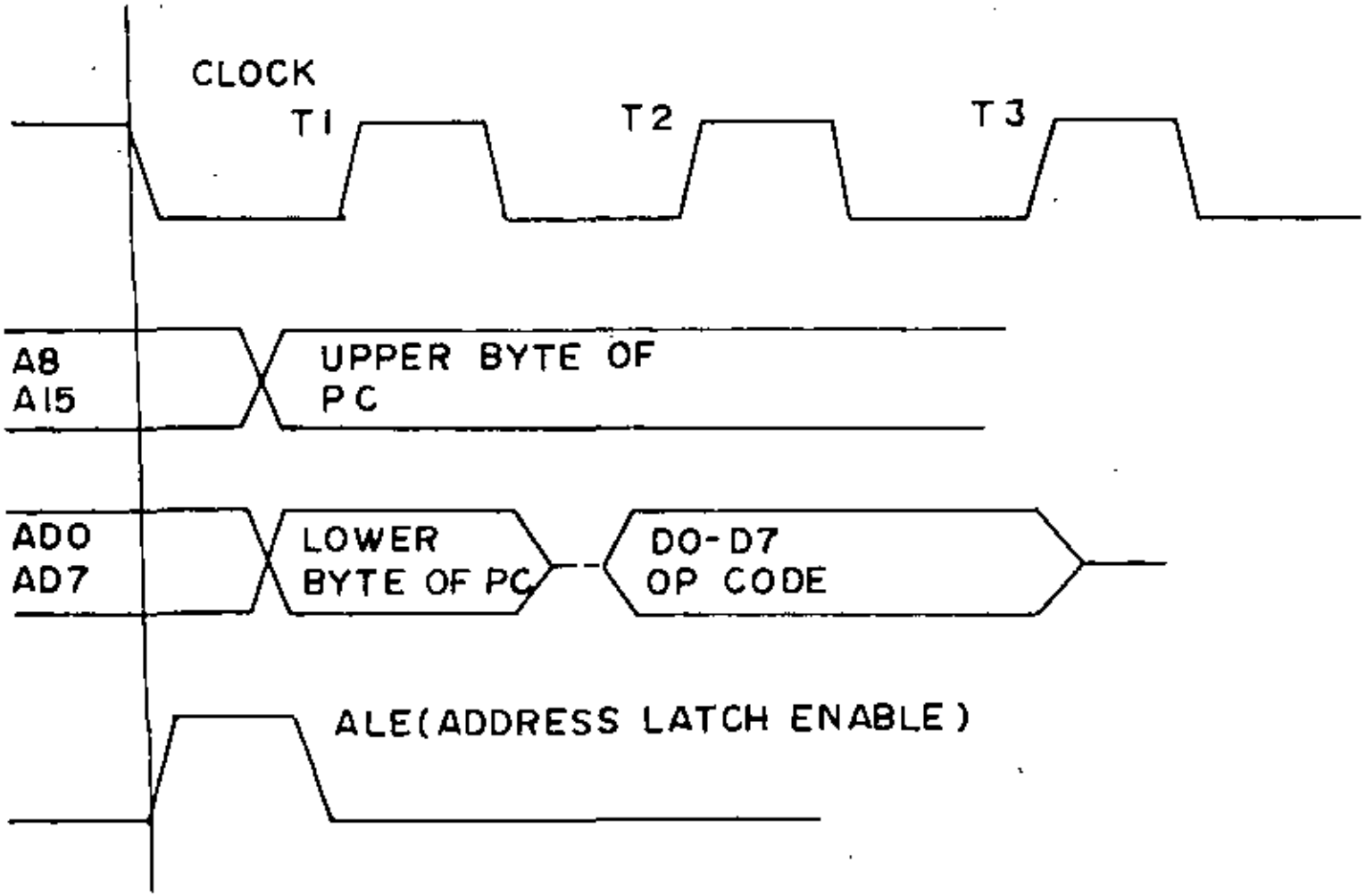

CLOCK
T1
T2
T3
A8
A15
UPPER BYTE OF
PC
AD0
AD7
LOWER
BYTE OF PC
D0-D7
OP CODE
ALE(ADDRESS LATCH ENABLE)

Fig. 2-11

The maximum clock rate is limited by propagation delays, access times, and other factors within the processor. The slowest rate at which the processor will operate depends upon the type of storage in the internal registers; viz. static or dynamic. In dynamic storage, the information is stored on capacitors which will eventually discharge unless the information is refreshed. Thus the minimum clock rate depends upon the rate of discharge. The slowest specified clock rate for the 6800 is 100KHz. At room temperature where the leakage is not particularly large, a 6505 can be operated at 5 KHZ before it ceases operation. The 8080 also has dynamic storage, and its details are included in App. A.

For static storage (the associated clock is sometimes called a "static clock") the clock may be slowed as much as desired. The RCA COSMAC (1802) has a static clock and thus may be single stepped using a debounced switch. Static and dynamic storage in the internal registers is, of course, similar to the static and dynamic memory discussed in Chp. 3.

Finally, some processors provide synchronizing or state signals that may be used for the control of peripherals or for interfacing. The 8080, for instance uses a "synch" signal; the RCA COSMAC state signals, and the 6800 has a VMA (valid memory address) signal. As discussed ealier, some processors do not have sixteen address pins, but time multiplex the address information or share the address bus with the data bus. In this case, the sixteen address bits are sent out in two bytes, and one of the bytes will have to be latched or held by external circuitry. A timing signal for the latch has to be provided such as the ALE on the 8085.

Control Bus

The last group of signals include read write lines, the ready or hold lines, RESET input, HALT, and IRQ, the interrupt request. These are often grouped together. The number of such lines and their special features vary widely from processor to processor. The interrupt feature is discussed in the next chapter. The Halt Line performs the obvious action of stopping the processor. A temporary halt may be used in some processors (e.g. the 6800) so that another device such as a direct memory access device (DMA) may control the memory or I/O. In other processors, a HOLD signal causes the processor to enter a waiting state and to release to bus to some other devices. Some of these features are discussed later in the next chapters.

The read and write signals fall into two groups. In the 6800 and 650X series, one single R/W contains both signals, and the convention is that the READ is high or 1 and the write is low or 0. This can lead to problems as discussed below. In the 8080 and the COSMAC and other processors,s the read and write signals are separate. For instance, the 8080 has a memory read and memory write lines that are inactive when they are high or 1. When a read or a write operation is desired, one of the lines goes low to signal the memory that a read or a write operation is occurring. See Fig. 2-12 .

A question of interest to most designers is, what is on the various buses? The manufacturers now provide that information, and the differences between processors is very interesting. The 8080 is discussed in an appendix, and we will simply state that it puts nothing on the bus that shouldn't be there. In the 6800, depending upon the operation, inappropriate information may be on the

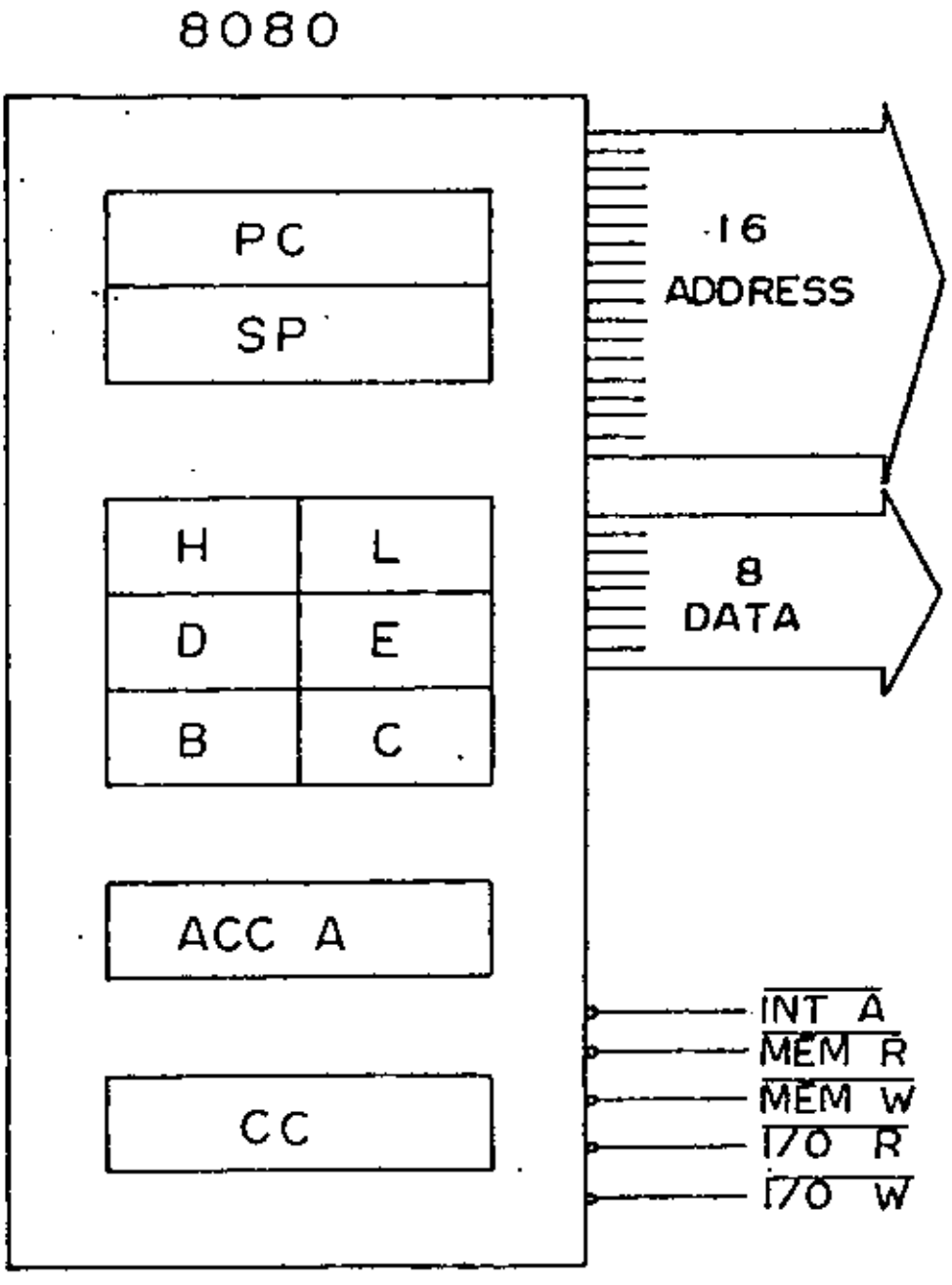

8080
PC
SP
H
L
D
E
B
C
ACC A
CC
16
ADDRESS
8
DATA
INT A
MEM R
MEM W
I/O R
I/O W

Fig. 2-12

address or data bus. Fortunately, a read operation normally occurs is such cases and that data is simply ignored. Motorola further provides a VMA signal (valid memory address) that is supposed to enable the memory and I/O registers. VMA is never asserted unless a valid memory is on the address bus. The 650X series has very similar properties. Cycle by cycle descriptions are shown in Fig. 2-13.

Examination of a Processor in Detail

In this section we will examine the instruction set of the 6800 processor and the details of its operation. This processor, as part of the Motorola Evaluation Kit II will be used in the tutorial. The designers of the 6800 initially attempted to produce a microprocessor with the same instruction set as the PDP-11 minicomputer; however the processor could not be realized on a single chip and marketing indicated that a two chip processor would not sell well. The final product does strongly resemble the PDP-11 (albeit 8-bits) and is perhaps the easiest microprocessor to program. Like the PDP-11 mini, the 6800 uses memory mapped I/O; that is, it has no separate I/O instructions. Thus any instructions that apply to memory locations can be applied to the I/O registers.

The internal registers in the 6800 are shown in Fig. 2-14. The two accumulators, the index register, the stack pointer, and the program counter are very conventional registers. The 6800, like almost every other processor, is a single operand machine: all arithmetic, logical, and data transfer instructions move through the accumulators and only one operand is manipulated at a time.

The 6800 is easy to program because it has a reasonable instruction set and it has good number of addressing modes. The modes include the immediate mode, the absolute mode (two forms), a true index mode, and the register mode.

CYCLE - BY- CYCLE OPERATION

EXTENDED ADDRESSING:

ADD @#XXXX

ADDRESS BUS	R/W	DATA BUS
Op code address	1	Op code
Op code address+1	1	Address of operand (high byte)
Op code address+2	1	Address of operand (low byte)
Address of operand	1	Operand data

DEC @#XXXX

ADDRESS BUS	R/W	DATA BUS
Op code address	1	Op code
Op code address+1	1	Address of operand (high byte)
Op code address+2	1	Address of operand (low byte)
Address of operand	1	Current Operand data
Address of operand	1	Irrelevant data (note)
Address of operand	0	New Operand data.

note: If VMA is used to enable memory, then VMA will be 0 during this cycle and the data bus will depend on capacitive decay.

Fig. 2-13

M6800 MICROPROCESSOR

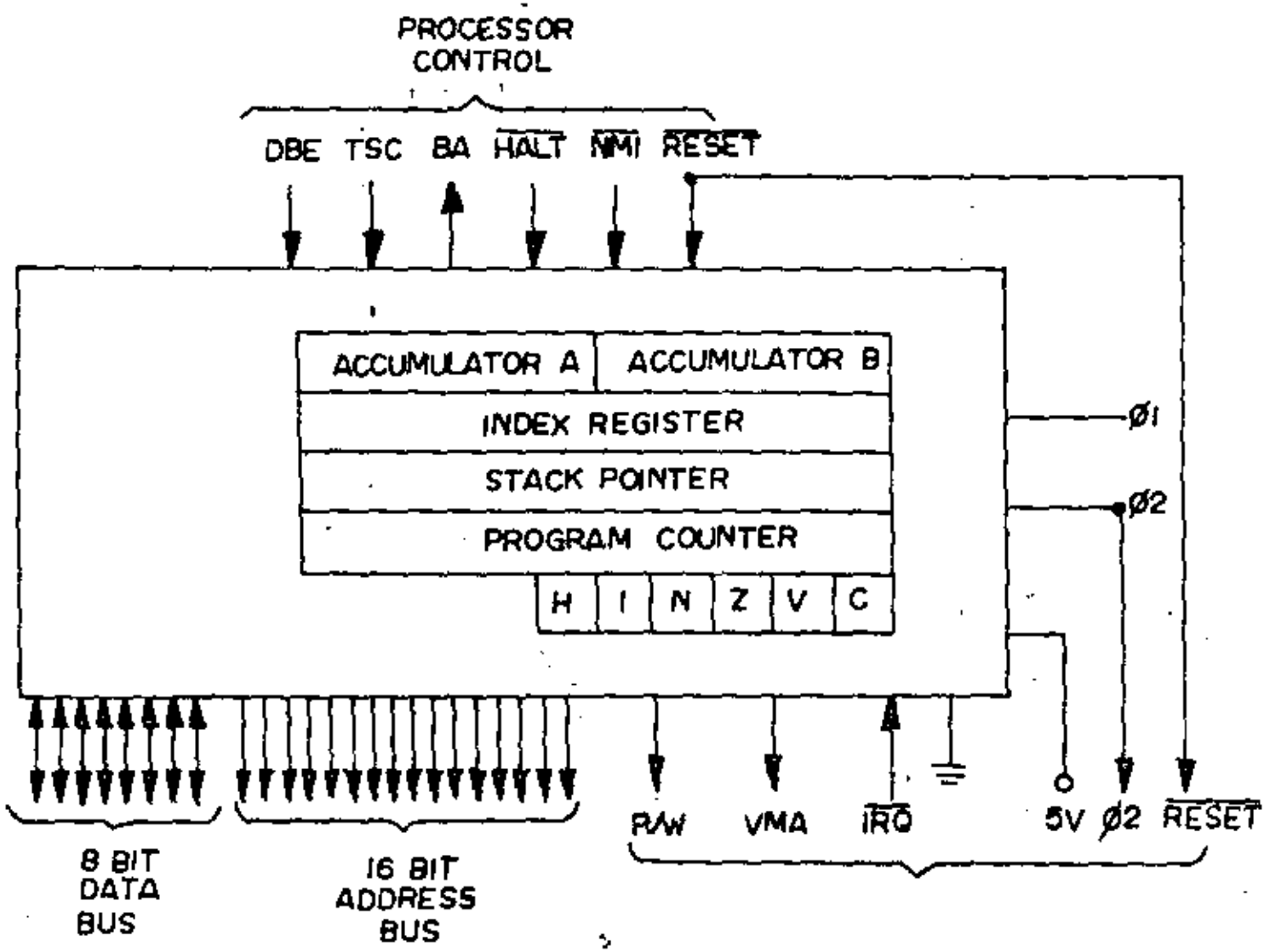


Fig. 2-14

(Motorola's names for these modes are not standard names.) Conditional branches use the relative mode. These modes are summarized in the following.

Immediate mode: In the immediate mode, the operand is stored in the location following the op-code. For example consider the instruction which adds the number 5 to accumulator A:

```
ADD #5, ACCA
```

This would be assembled in the following form:

```
ADD #5, ACCA    89    (op-code)
                05    (data)
```

Extended and Direct Modes: The modes are really absolute addressing modes, i.e., the absolute or numerical address of the operand is given in the following byte or bytes. In the extended mode, the specific address of the memory location or I/O register is given in the following two bytes (the full sixteen address bits), the high order byte first. For memory locations with addresses less than 256, the direct addressing mode may also be used. Here, only one byte, specifying one of the first 256 locations is used. This direct mode permits faster executing and, of course, saves memory locations. An example is a load the accumulator with the contents of a memory location:

```
LDA @# 167A
```

This instruction loads the accumulator with the contents of location 167A (hexadecimal), and would be assembled:

```
LDA @#167A      B6      (op-code)
                16
                7A      (low byte of address)
```

The 6800 has an index register which may be used to located the operands. This addressing mode uses an offset following the op-code to specify how many times the index register must be incremented to point to the address of the operand. This mode is especially useful when dealing with tables. Although some microprocessors are said to have an index register, they have in fact only a pointer register: one that points to a memory location only and has no provision for an index or offset. The 6800 has a true index register. For example, if the index register is currently pointing to location 8008, and one wishes to clear location 800A, one can use the following:

```
CLR 2(X)
```

which is assembled,

```
CLR 6F    (clear index mode)
    02    (index or offset)
```

The "implied" mode or more commonly the register mode is used when one wishes to deal with one of the internal registers: the accumulators, the index or stack

being executed.

pointer registers, or the condition code register.

Finally the relative mode is used for the branch commands and consists of adding the second byte of the instruction to the program counter to divert the program execution to another part of the program. The second byte is a 2's complements number, meaning that the contents of the program counter may be increased or decreased. An example is a branch from the current location back to a previous instruction, often in a loop. For instance, one may wish to test the state of a process by testing a done bit. If the done bit is the most significant bit of a register then the register's contents are a negative number when the done bit is set (=1). We suppose that the register is located at 8008 (hex).

```
LOOP:   TST @#8008
        BPL LOOP
```

Thus unless the done bit is set, the loop test the contents of 8008 and branches back for another test. This would be assembled:

```
LOOP:   7D      (test)
        80
        08      (the address)
        2A      (BPL, branch of plus)
        FB      (2's complement of -5)
```

Note the one branches back -5 because the program counter will point to the location following the offset of the branch command while the branch command is

The instruction set is displayed on the following pages and are grouped according to their function. Not all of the instructions may use all of the addressing modes; for example the clear (CLR) instruction cannot use the direct addressing mode but must use the extended addressing mode.

Data Handling Instructions

Function	Mnemonic	Operation
Load accum.	LDAA,LDAB	M ->A, M ->B
Push data	PSHA,PSHB	A -> stack B -> stack
Pull data	PULA,PULB	stack ->A stack ->B
Store accum.	STAA,STAB	A ->M, B->M
Transfer accum.	TAB,TBA	A ->B, B ->A

Data Handling Instructions II

Function	Mnemonic	Operation
Clear	CLR,CLRA,CLRB	00->M, A, B
Decrement	DEC,DECA,DECB	M - 1 -> M A - 1 -> A B - 1 -> B
Increment	INC,INCA,INCB	M + 1 -> M A + 1 -> A B + 1 -> B
Negate 2's complement	NEG,NEGA,NEGB	-M,-A,-B
Complement (1's)	COM,COMA,COMB	M, A, B

Data Handling Instruction III

Function	Mnemonic	Operation
Rotate left	ROL,ROLA,ROLB	c ->b0, b7 ->c
Rotate right	ROR,RORA,RORB	b0 ->c, c ->b7
Arithmetic shift left	ASL,ASLA,ASLB	b7 ->c, 0->b0
Arithmetic shift right	ASR,ASRA,ASRB	b7 ->b7, b0->c
Logic shift right	LSR,LSRA,LSRB	0 ->b7, b0->c

A - Accumulator A
B - Accumulator B
M - Memory location
c - carry bit
b0 - bit 0
b1 - bit 7

Arithmetic Instructions

Function	Mnemonic	Operation
Add	ADDA,ADDB	A + M -> A B + M -> B
Subtract	SUBA,SUBB	A - M -> A B - M -> B
Add with carry	ADCA,ADCB	A + M + c -> A B + M + c -> B
Subtract with carry	SBCA,SBCB	A - M - c -> A B - M - c -> B
ADD accumulators	ABA	A + B ->A
Subtract accms.	SBA	A - B -> A

Logic Instructions

Function	Mnemonic	Operation
AND	ANDA, ANDB	A M -> A B M -> B
Exclusive OR	EORA,EORB	A M -> A B M -> B
Inclusive OR	ORA,ORB	A + M -> A B + M -> B

+ - plus or OR

Conditional Branch and Jump Instructions

Function	Mnemonic	Condition
Branch always	BHA	none
Branch if = zero	BEQ	Z = 1
Branch if = zero	BNE	Z = 0
Branch if plus	BPL	N = 0
Branch if carry set	BCS	C = 1
. . .		
Branch to subroutine	BSR	
Jump	JMP	
No operation	NOP	

This is a partial list of the branch and jump commands; some depend upon the bits in the condition code register which are summarized below:

Bit no.		Condition Code
0	C	carry - borrow
1	V	overflow
2	Z	zero
3	N	negative
4	I	interrupt mask
5	H	half carry (for dec. arith.)

Chapter 3 MEMORIES - RAM AND ROM

Semiconductor memory development preceded that of microprocessor development and is equally important. Soon after the announcement of the first microprocessor, it was facetiously stated that the components manufacturers developed micros so that they could sell memory! In fact, the cost of the memory usually exceeds the cost of the microprocessor in a typical system. In this chapter, we will discuss read-only-memory, read/write memory, examine a simple memory system, and discuss direct memory access (DMA).

Read Only Memory: ROM

A microprocessor's program is usually stored in ROM since ROM is non-volatile; that is, it retains its contents without applied power. In prototype or evaluation systems, the program may be temporarily stored in read/write memory for easy testing and editing, and then the program may be moved to ROM. Electrically alterable ROM is also useful for systems where the ROM will be used in small quantities or where the program may be modified again. The well known 1702 uses a p-MOS technology[4] and contains 256 bytes. Recently the 2708, an n-channel ROM, has become available. It contains 1K bytes and is much easier to program than the 1702. Each of these ROMs is ultraviolet erasable and can be reprogrammed in an appropriate device external to the microprocessor system in which it will ultimately be used. This process takes at least 15 to 20 minutes including the erasure. Once programmed, the ROM chip is placed in the microprocessor system. If a large number of identical ROMs are to be used for many

4. p-MOS and n-MOS refer to p-channel and n-channel Metal Oxide Semiconductor transistors used in the memories.

identical systems, mask programmed ROMs are available from a number of manufacturers which are pin for pin compatible with the erasable ROMs. Even larger capacity mask programmed ROMs are available; e.g. the Intel 8316A with 16K bytes. Other types of programmable ROMs include fusible link ROMs in which a nichrome or silicon link is burned opened to store a 1 or 0 in the corresponding bit location. Fusible link ROMs are usually bipolar hence much faster than the MOS ROMs; but they are less practical since the faster speed is not necessary , they can be programmed only once, and the chip capacities are smaller. Intel, MOS Technology and others have combined ROMs and I/O circuits on the same chips. This helps to minimize the chip count for the system. Some of the ROMs are reprogrammable. The ultimate chip is the 8748, a member of the the MCS-48 family of Intel, which contains the processor, erasable ROM, read/write memory, interval/event timer, and copious I/O.

In addition to program storage, ROM can store tables of data, codes, monitor programs, assemblers, editors, etc. In evaluation kits and development systems, monitor programs assist the user in writing, editing, and executing programs.

Read/Write Memory: RAM

Actually, ROMs are random access memory (RAM), but "RAM" is usually reserved for read/write memory. Developments in RAM have proceeded as least as fast as microprocessor development. Memory chips with 16 K by one bit organizations are available; eight such chips would provide 16k bytes. In microprocessor systems, RAM is usually used for data storage, however the program may be stored in and executed from RAM. RAM is volatile, and thus requires battery backup if a power loss occurs. Many microprocessor systems require only a small

amount of RAM and thus component manufacturers have provided organizations that minimize the number of chips. Two example are shown in Fig. 3-1. The first is a Motorola 6810 with 128 bytes, and the second is an Intel 2111 with 256 4-bit nibbles. Two 2111 would provide 256 bytes. Larger amounts of memory may be obtained using 2102's (1K by 1 bit). For very large memory systems, larger capacity chips such as the 2107 (4K by 1 bit) and the newer 1K by 4bits static memory by Intel are also available.

Dynamic and Static RAM

Some of the larger capacity chips such as the 2107 have dynamic storage of bits. That is, the bits are stored on capacitors fabricated directly on the chip using the MOS technology. Since the data will eventually leak away, it is necesssary to detect the state of the bit before it is changed and restore the charge to the initial state: refreshing. The refreshing is accomplished using additional circuitry which addresses the chip and performs read operations on each location. Counter and timing circuits are needed to ensure that all locations (actually all columns in the array) are refreshed and to avoid conflict with processor access. Normally, the processor operation must be delayed or held in a state and be disconnected from the address bus to avoid conflicts. A refresh of each location must be done every 2 millsec. and thus the processor operation may be slowed by as much as 3%. The complexity of the refresh circuit does not increase with memory size, and thus dynamic memory is not the burden that one might think. In addition, dynamic memory uses less energy than static and is therefore more practical for battery backup. Finally, nearly all the very large capacity RAMs are dynamic, hence they are more suitable for large memory systems.

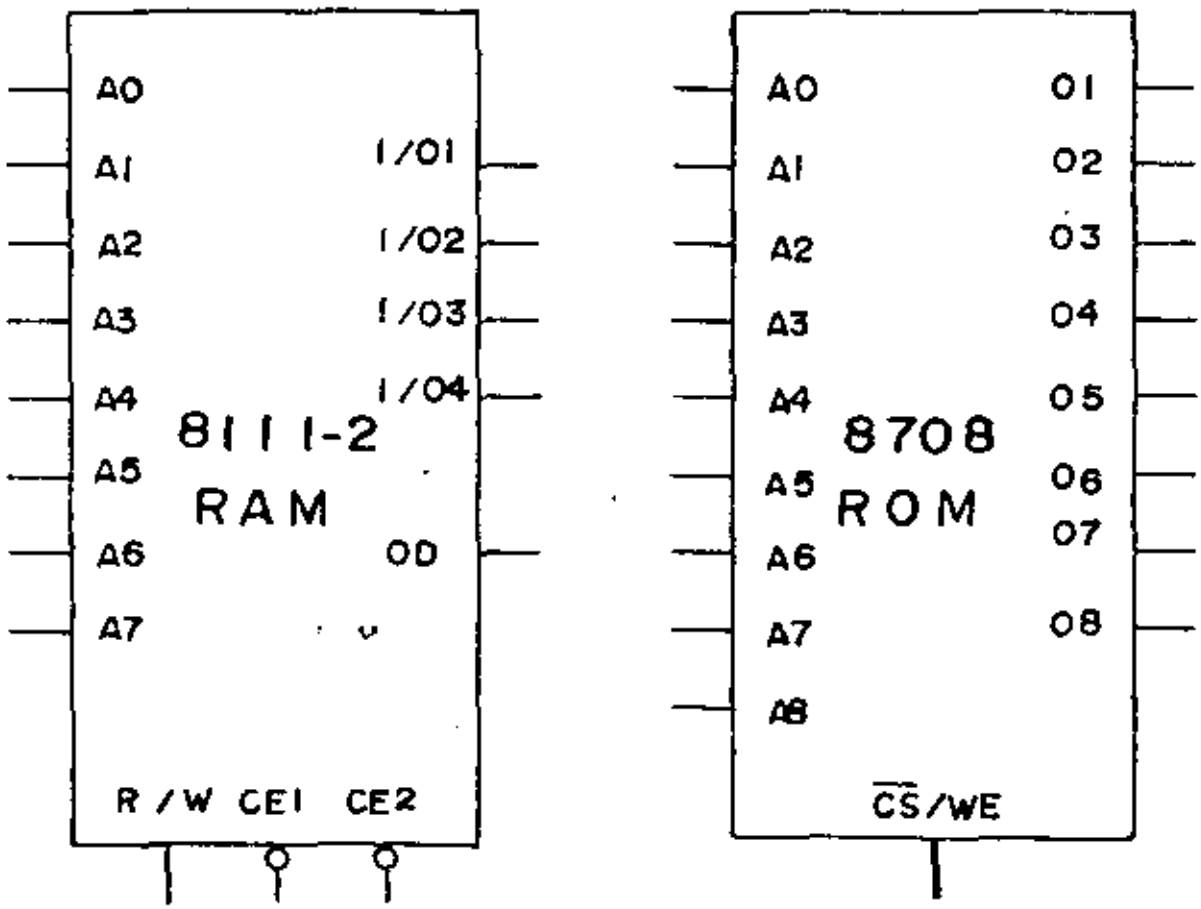

A0
A1
A2
A3
A4
A5
A6
A7
8111-2
RAM
I/O1
I/O2
I/O3
I/O4
OD
R/W CE1 CE2
A0
A1
A2
A3
A4
A5
A6
A7
A8
8708
ROM
O1
O2
O3
O4
O5
O6
O7
O8
CS/WE

Fig. 3-1a

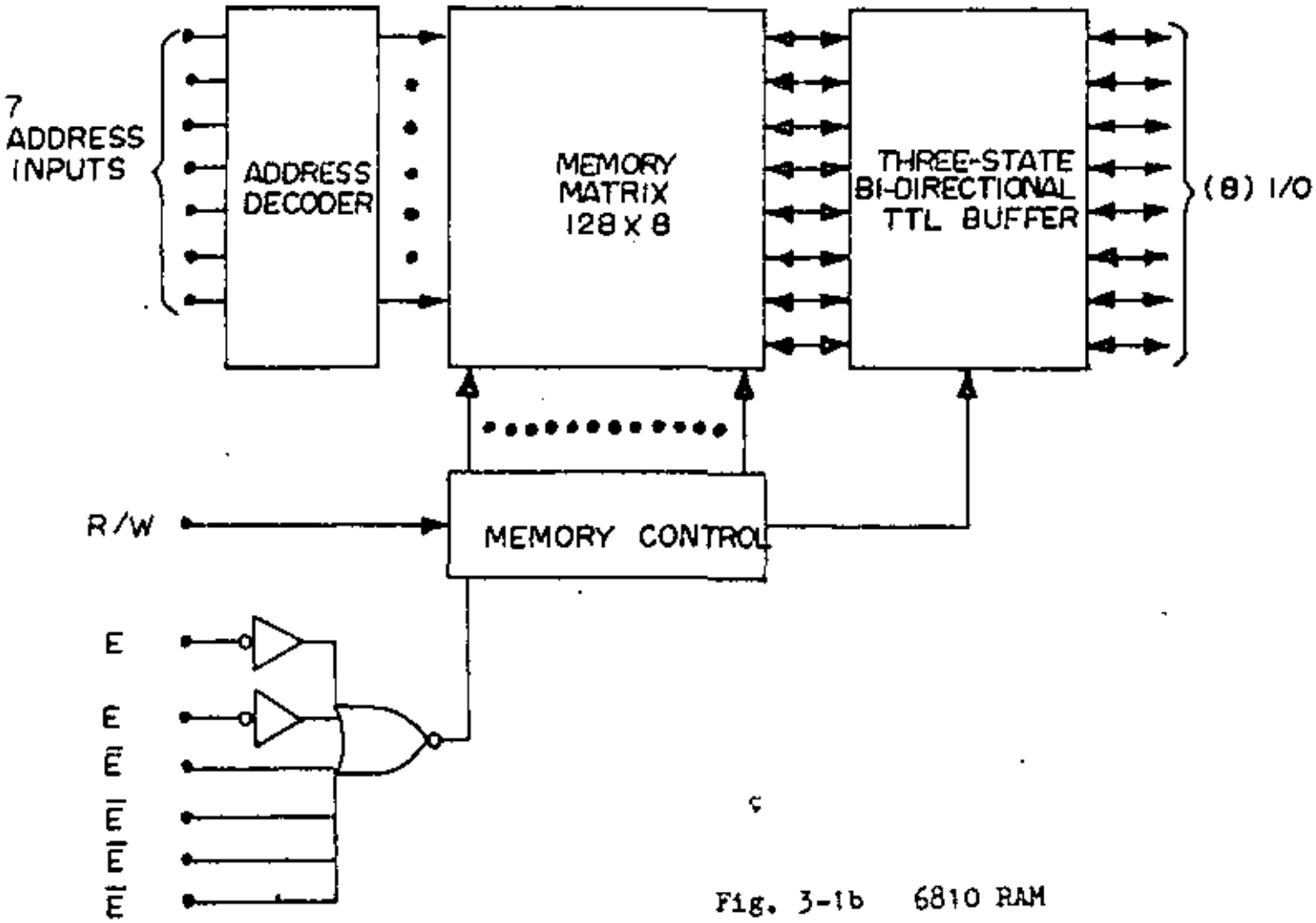


Fig. 3-1b 6810 RAM

The smaller capacity RAMs are static. Here the bit data is stored on flip-flops and the data will remain so long as power is applied. Power-down options are available which save power when the memory is not accessed by the processor. Static memory is easier to use than dynamic and is the obvious choice for small RAM systems.

Access Time and Cycle Time.

Recall that microprocessor timing is controlled by clock cycles. When the processor accesses memory, it sends out address information for chip selection and for byte location within a chip(s). At the end of phase two in our earlier example in Chp. 2, the processor expects valid data. Thus the memory must be fast enough to yield valid information. The time between the application of valid address and the data valid state is called the "access time" and must at least match that of the the processor. Shown in Fig. 3-2 are the timing data for a 2111. Note also that the chip select may occur after the address is applied. Usually the address information is involved in a chip select, but the phase two clock may also be involved in chip select. Thus a second time, the "chip enable to output" time is also important. Typical times are shown in the figure and in the associated table. Other times are included for completeness. Extremely fast memory, faster than 400 nsec would be a wasted expense for most systems.

Simple Memory System

Shown in Fig. 3-3 is a simple system using one ROM and two 2111 RAMs. The memory is connected to an 8080 bus and some address decoding is accomplished by

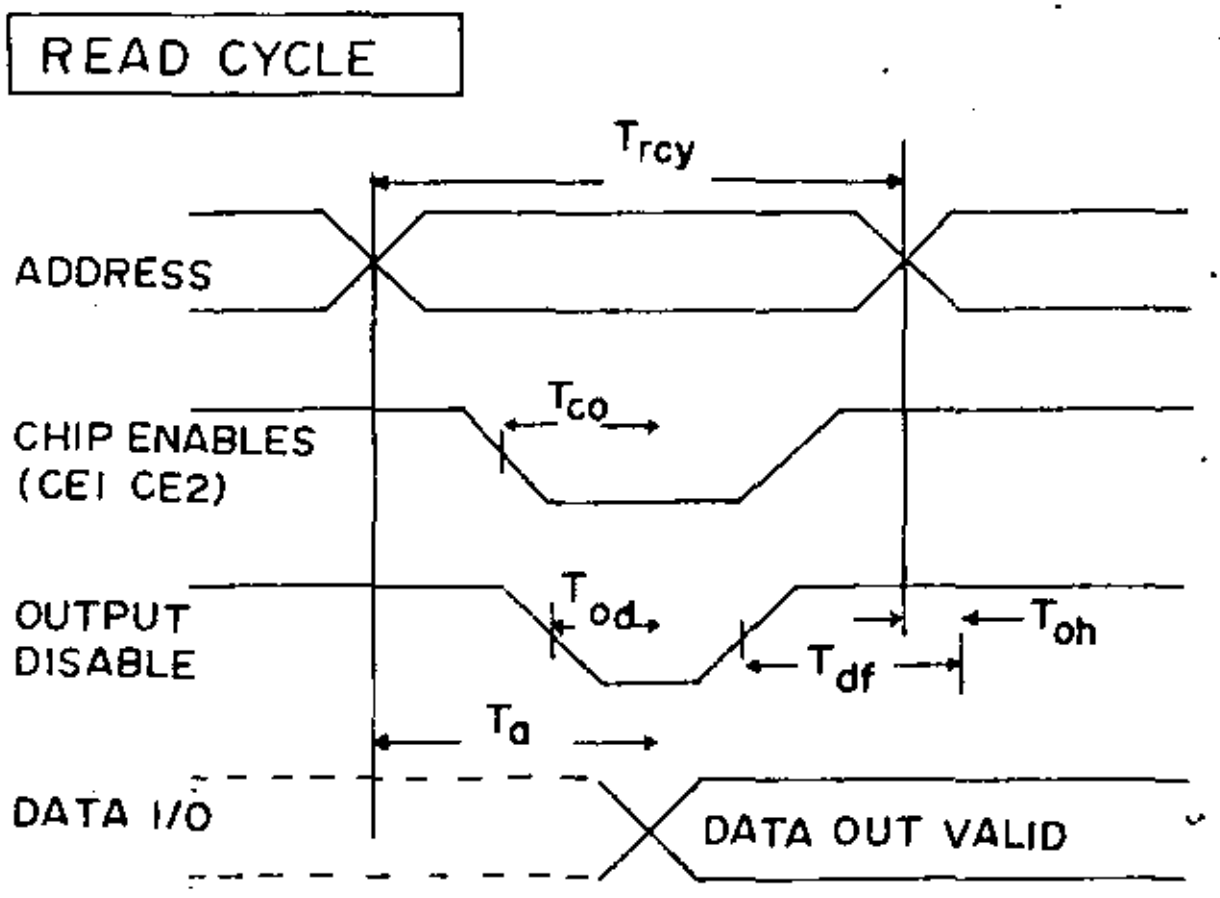

READ CYCLE
Trcy
ADDRESS
Tco
CHIP ENABLES
(CE1 CE2)
OUTPUT
DISABLE
Tod
Toh
Tdf
Ta
DATA I/O
DATA OUT VALID

Fig. 3-2a

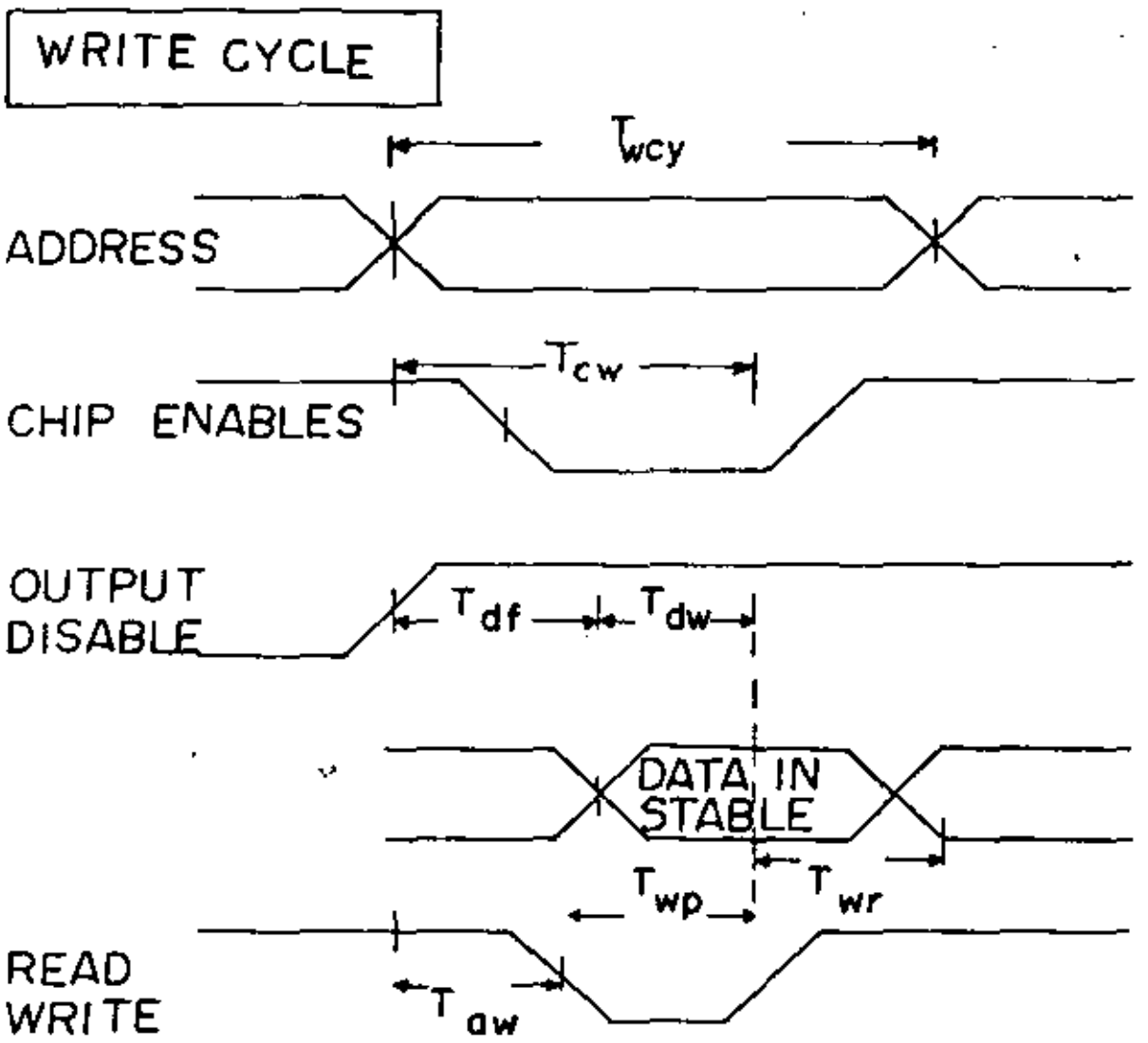

WRITE CYCLE
T_{wcy}
ADDRESS
T_{cw}
CHIP ENABLES
OUTPUT DISABLE
T_{df}
T_{dw}
DATA IN STABLE
T_{wp}
T_{wr}
READ WRITE
T_{aw}

Fig. 3-2b

TIMING DATA FOR THE 8111 RAM

		Min.	Max.	
t_{rcy}	Read Cycle	850		nsec.
t_a	Access time		850	
t_{co}	Chip enable to output		550	
t_{wcy}	Write Cycle	850		nsec.
t_{aw}	Write delay	150		
t_{wc}	Chip enable to output	750		

Fig. 3-2c

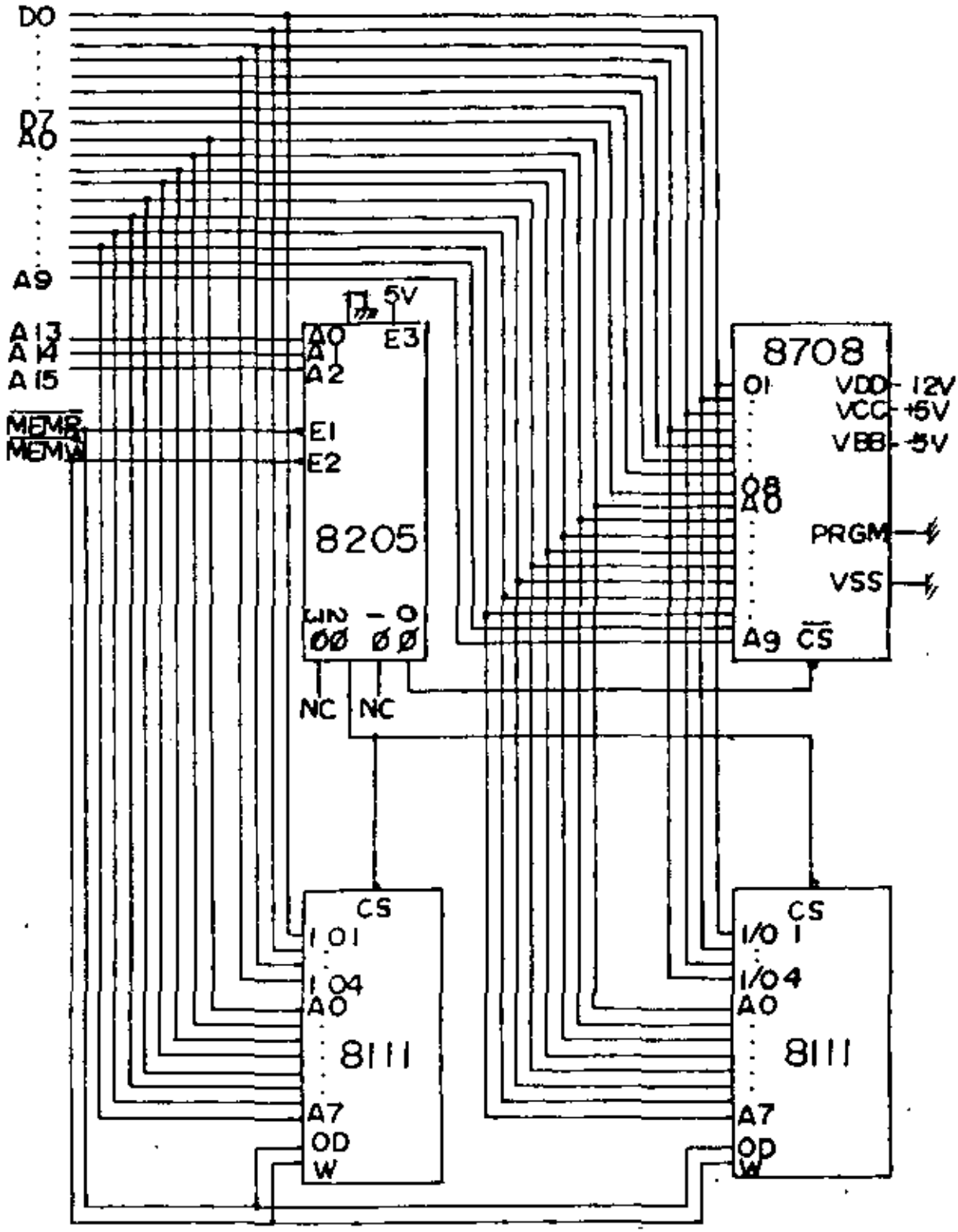

D0
D7
A0
A9
A13
A14
A15
MEMR
MEMW
5V
A0
A1
A2
E3
E1
E2
8205
GND
O0 O1 O2 O3
NC
NC
8708
O1
VDD 12V
VCC +5V
VBB -5V
O8
A0
PRGM
VSS
A9 CS
CS
I/O1
I/O4
A0
8111
A7
OD
W
CS
I/O1
I/O4
A0
8111
A7
OD
W

Fig. 3-3

the 8205 chip which supplies a low on one of its outputs according to the input pattern. That low selects the appropriate chip or chips. For this system, the ROM locations start at 0000 and the 256 bits of RAM start at 4000. More complete system drawings can be found in the appendices of the Intel SDK-80 Manual.

Direct Memory Access

Direct Memory Access (DMA) is related to input/output and thus could be discussed in the following chapter as well. Under programmed I/O, a substantial time (at least several microseconds) are required for data transfers to memory; further, the processor must be dedicated to the I/O process. In many applications, fast transfer of large amounts of data to memory are required. It may be that the data generation rate of the process is inherently fast or it may be desired to transfer the data so that the processor is not tied up or held in a waiting state for long times. An excellent discussion of various techniques can be found in Motorola's Applications Manual for the 6800. There are direct techniques where the processor is halted or held in a waiting state so that the address and data lines are not controlled by the processor. An external circuit or device then can take direct control of the buses and deposit the data in memory. The 8080 has a HOLD signal input that accomplishes just that task. The 6800 is only slightly more complicated in its HALT state. In both, the memory may be written fast as the cycle time of the memory permits (one application where fast RAM is useful). This scheme does not require complex hardware. Another techniques uses "cycle stealing" which does not require a complete shutdown of the processor or complex circuitry. However, the data transfer rates are correspondingly less. An unusual scheme involving the 6800 is "multiplexed DMA" and is illustrated in Fig. 3-4. This scheme is possible because of the simple, symmetrical clock timing of the 6800. These techniques are summarized

MULTIPLEXED DMA/MPU OPERATION

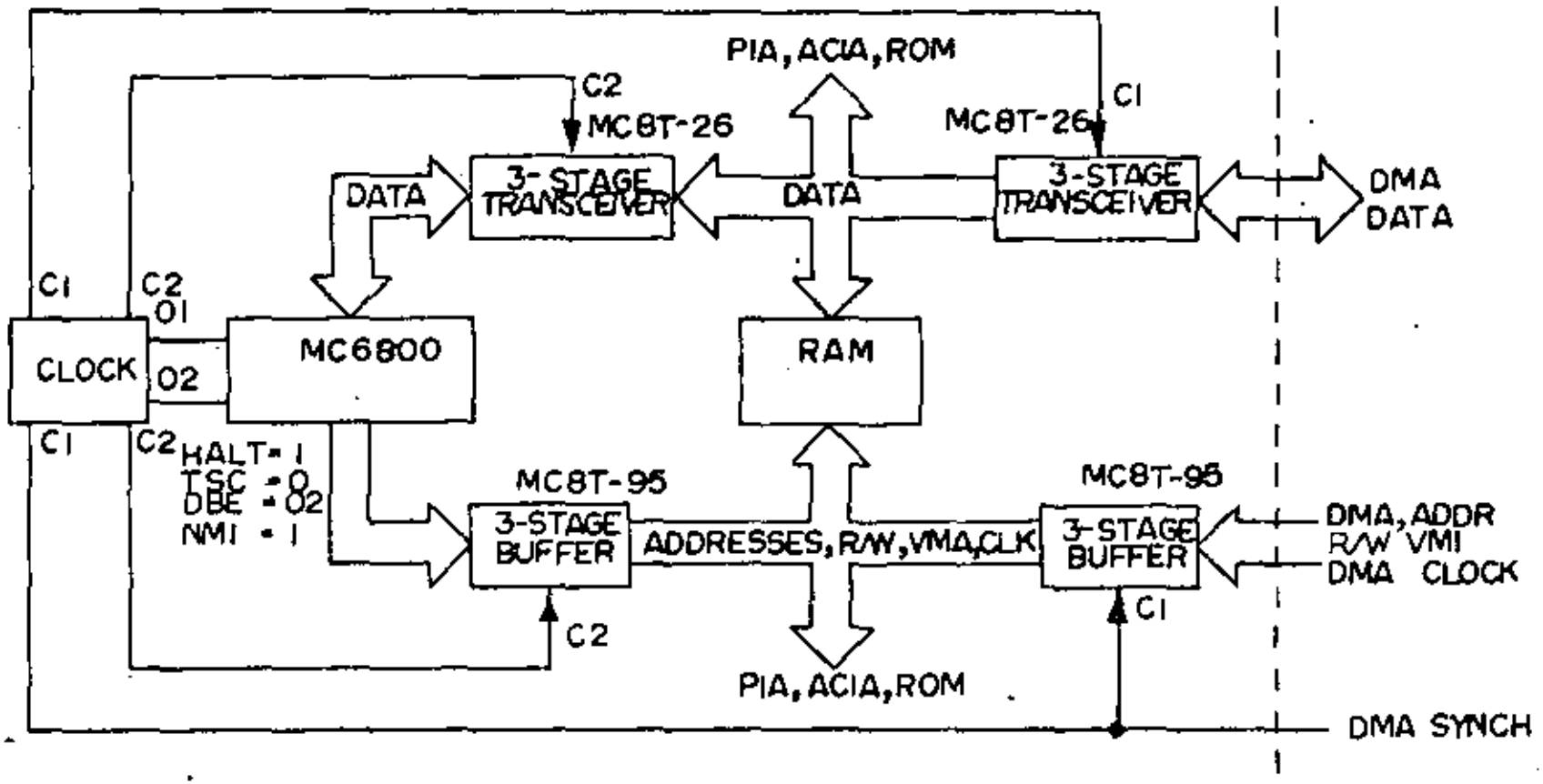


Fig. 3-4

in the associated table.

Applications for DMA include disk interfaces where data transfer in blocks at high speeds is required and a variety of data acquisition systems. One example is a transient recorder in which the digital data is stored directly in the RAM and then the processor "slowly" displays all or parts of the pulse under program control. Hardwired systems cost up to $10,000; but a microprocessor system could reduce much of the cost and provide additional features.

DMA can also be accomplished by some of the programmable chips discussed in the next chapter; however, the data rates are not as high as those in which the processor is halted or held in a HOLD state.

DMA TECHNIQUES

Technique	Maximum Data Rate	MPU program Execution	Hardware Complexity
Halt Processor	1 byte/microsec.	0	Lowest
Cycle Steal	1 byte/2.5 microsec.	1cycle/5 microsec.	Medium
Multiplexed DMA	1 byte/1.2 microsec.	1cycle/1.2 microsec.	Highest
Software (PIA)	1 byte/14 microsec.	Dedicated to Aquisition	Lowest

Notes:

These examples hold for the 6800 processor and are described in Motorola's Applications Manual. The DMA rate for the Halt case is limited by memory speed.

CHAPTER 4

PROGRAMMABLE MICROPROCESSOR SYSTEM CHIPS - INPUT/OUTPUT

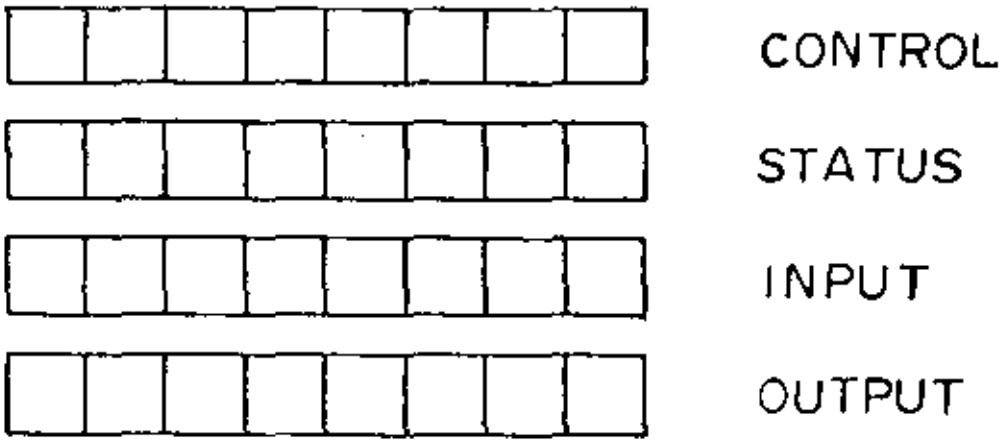
When microprocessors first became available, the input and output circuits has to be deisgned using SSI logic. Not only was this task time consuming, but the number of chips in the system increased a great deal. Motorola, and later

Intel and others, eased these problems by providing LSI chips to handle I/O. Recently, additional LSI chips have become available that augment the processor's capabilities in other ways. Almost all of these newer chips are programmable, and thus they can be configured to suit a given task. Standard microprocessor systems can therefore be fabricated for a number of applications thereby reducing production and documentation costs. Intel, Motorola, are many other manufacturers have produced a number of systems that can be programmed by the user for various tasks. The 80/10 and 80/20 systems of Intel are good examples. These systems contain a variety of I/O circuits in addition to the processor and memory, and they can be expanded to include additional memory and I/O. These small systems are particularly useful for applications where only a few systems are required. For applications that require a larger number of processors, several manufacuters will provide custom designs. Whether a designer chooses these systems or uses one of his own, he will encounter these programmable chips.

In this chapter, we will discuss basic input/output chip, and then examine some specific examples of parallel and serial I/O. Interrupt programming, including priority interrupt schemes will then be discussed and illustrated. Finally other programmable chips will be surveyed.

Basic Input/Output

Programmable chips may be descibed using the model shown in Fig. 4-1. Not all of the registers are found in every chip; some combine the registers, others may simply lack one or more of the registers. The chip is programmed by depositing the appropriate number in the control register. The control register may appear as a memory location to the processor (memory mapped I/O) or as one of



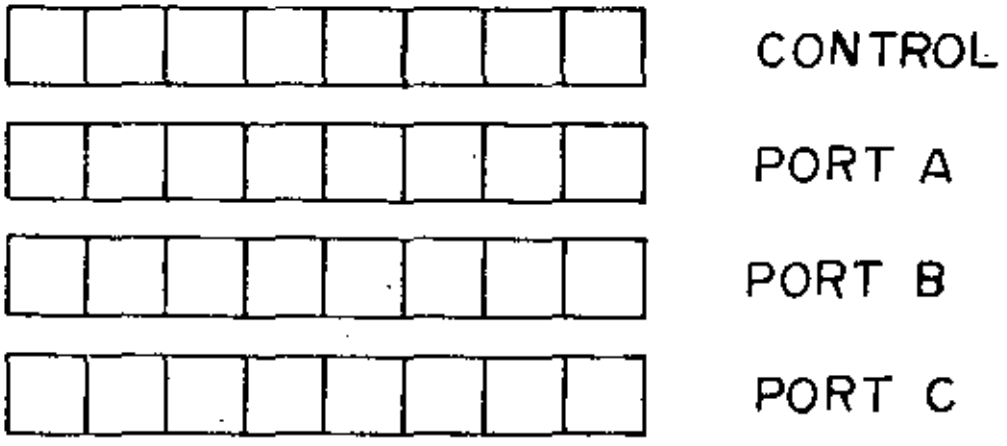


Fig. 4-1

the I/O registers (isolated I/O). The contents of the control register determine the roles that the other register can play. The contents may specify which register is an input register or an output register; they may enable or disable interrupt requests from the chip; or they may specify the type of input (e.g. latching with handshaking) or output (synchronous or asynchronous).

One of the more interesting examples is Intel's 8255 Programmable Peripheral Interface (PPI)[5] shown in Fig. 4-2. This device contains no status register. The four internal registers are selected by the two lines connected to the address bits A0 and A1. The control register is "write only"; that is, it cannot be read by the processor. On reset, all of the registers are cleared. The chip may be operated in three modes. Mode 0 is called basic input and output. In this mode, ports A and B may be either input or output ports. Port C is split and either half may be an input or output. In this configuration, the output lines will hold or latch the data in the output registers, and the input registers will reflect the state of the input lines. Further, any of the eight bits of port C may be individually set or reset using a single output instruction which writes into the control register. The control register is however not permanently modified by a bit set/reset operation. Details of the control register operation are shown in Fig. 4-3.

Keyboard Encoding - an Application Example

A simple scheme for encoding an array of switches using the 8255 is shown in Fig. 4-4. Of course, keyboards may be purchased with LSI encoder chips, however, encoding by the microprocessor may reduce the cost of the total system.

---

5. I don't make up these names, I just use them.

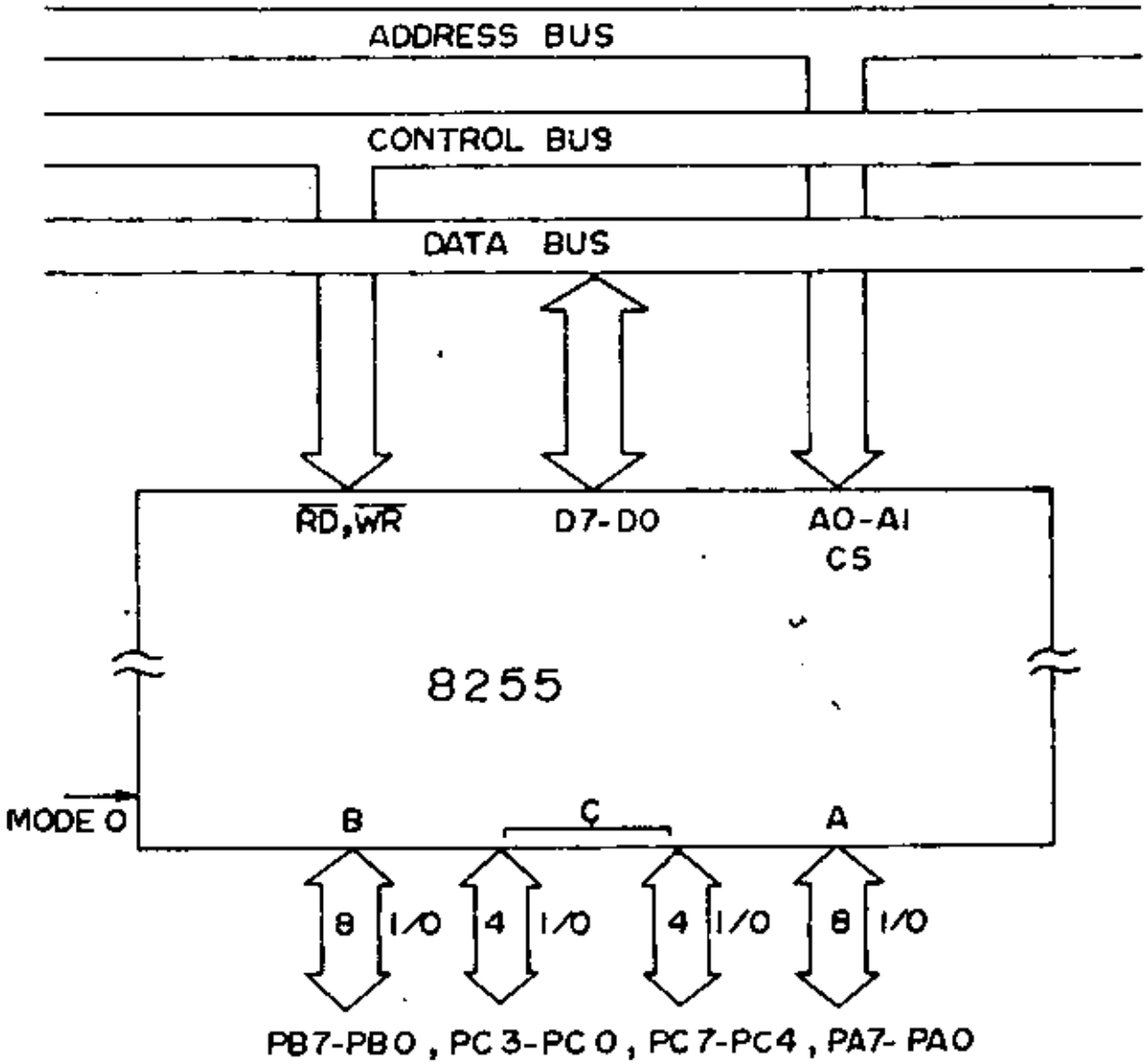

ADDRESS BUS
CONTROL BUS
DATA BUS
RD, WR
D7-D0
A0-A1
CS
8255
MODE 0
B
C
A
8 I/O
4 I/O
4 I/O
8 I/O
PB7-PB0 , PC3-PC0 , PC7-PC4 , PA7-PA0

Fig. 4-2

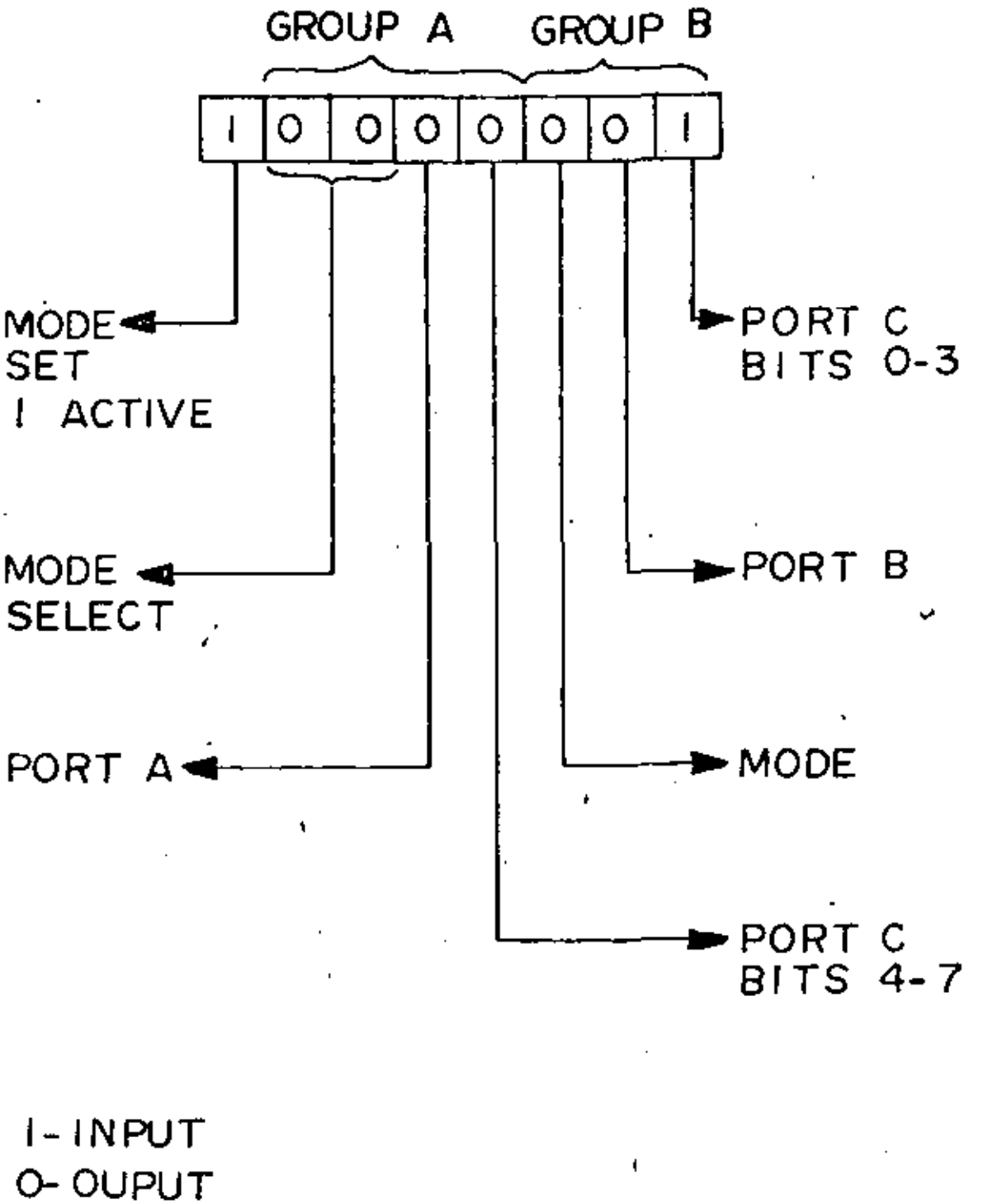

GROUP A
GROUP B
1 0 0 0 0 0 0 1
MODE SET I ACTIVE
MODE SELECT
PORT A
PORT C BITS 0-3
PORT B
MODE
PORT C BITS 4-7
I - INPUT
O - OUPUT

Fig. 4-3

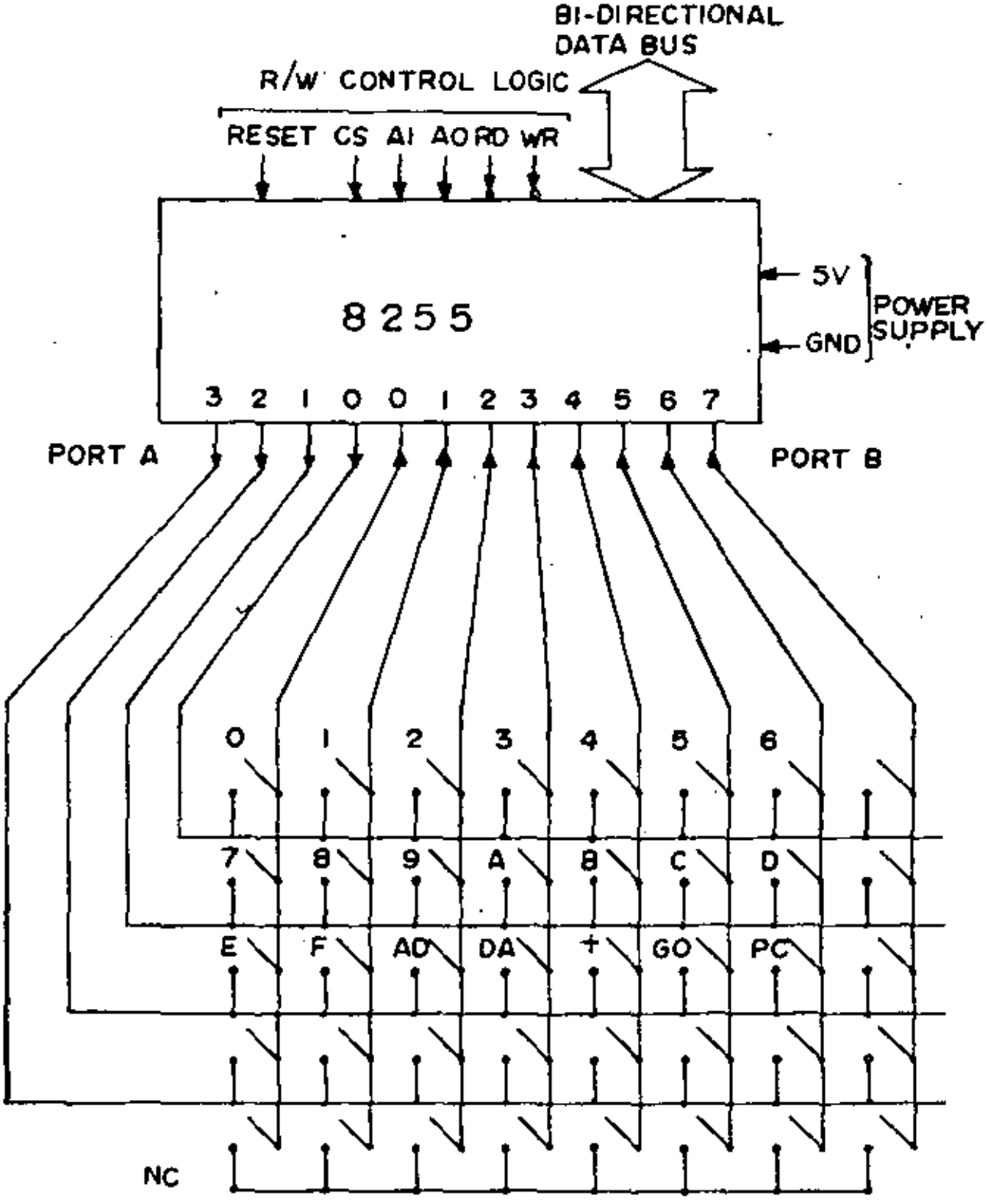

BI-DIRECTIONAL
DATA BUS
R/W CONTROL LOGIC
RESET CS A1 A0 RD WR
8255
5V
GND
POWER
SUPPLY
3 2 1 0 0 1 2 3 4 5 6 7
PORT A
PORT B
0 1 2 3 4 5 6
7 8 9 A B C D
E F AD DA + GO PC
NC

Fig. 4-4

This is very practical if the processor will frequently be waiting for an input from the keyboard anyway, and if the extra software does not require additional memory chips thus cancelling the saving of the encoder chip. Some of the simple evaluation kits such as the KIM-1 and the M6800D2 use this scheme. In this application, a pattern of 1's and a 0 will be applied from the A port, and if a key is closed in the row in which the 0 is applied, one of the port B inputs will be pulled low. If no switches are closed, the inputs to port B will all be high (+5 volts). The processor, under program control, thus shifts or rotates the single 0 among the port A outputs and checks the state of the B input. When a low is found at the input, the processor can use the states of port A and B to determine which key was closed. For example. the processor can use the B input data and the A output data to construct an address in a table stored in memory and find the corresponding ASCII character. An outline of the 8080 code is hown in Fig. 4-5.

8080 CODE FOR SWITCH DECODING

In the following, it is assumed that the 8255 registers have the addresses: Port A: F4; Port B: F5; Port C: F6; Control Register: F7.

A program fragment is shown; the complete program would be too complicated to be shown here.

```
MVI A, #82          ;load accum with 82
OUT F7              ;initialize the control register,
                     Port A, bits 0-3 outputs, Port B input,
                     Port C incidentally set as input
MVI A, #FE
OUT F4              ;Put zero in bit 0 of Port A
IN F5               ;read Port B
XRI #FF             ;1's complement the accum.
JNZ HIT             ;Hit is the routine for the table lookup
                     If zero, no key was closed, and the
                     zero bit of Port A should be shifted
.
.
.
.
```

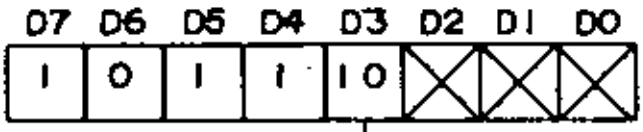
Fig. 4-5

Latched Inputs and Handshaking -Another Mode

In some applications, it is necessary to latch input data since the data may be valid only for a short time. Usually some timing signal or "strobe" is available to indicate when the data are valid. The processor may be occupied with some task which cannot be interrupted or the data valid dstate may be too short for the processor to capture the data. The 8255 may be used in mode 1 as shown in Fig. 4-6 for these cases. (If the processor is not otherwise occupied and if the data valid state exists for at least several microseconds, the processor can check the state of the strobe input and capture the data easily in mode 0 of the previous section.) In the latched input case of Fig. 4-6, the strobe signal loads the data into the input latch. Subsequently the 8255 issue the input buffer full (IBF) signal. If the interrupt enable is set using the bit set of pin 4 of port C, an interrupt request will be generated on pin 3 of port C. The strobe and the IBF constitute the handshaking; the IBF signal is useful for indicating to the inputting device that the data has not been read by the processor. When the contents of port A are read, the IBF signal is reset. Port A and port B may both be operated in this mode. Another use of Mode 1 is strobed output. Here data are placed at the output of A or B and an output buffer full (OBF) signal is issued by the 8255. When the output device connected to the 8255 accepts the data, it should issue an acknowledge signal that can in turn generate an interrupt request. (Interrupts are discussed later in this chapter.)

The 8255 can be operated in still another mode which can provide synchronous input or output. Details can be found in the 8080 Microcomputer Systems User's Manual. In summary, the 8255 provides 24 input or output lines and can

CONTROL WORD



PC 67
1 = INPUT
0 = OUTPUT

MODE 1 (PORT A)

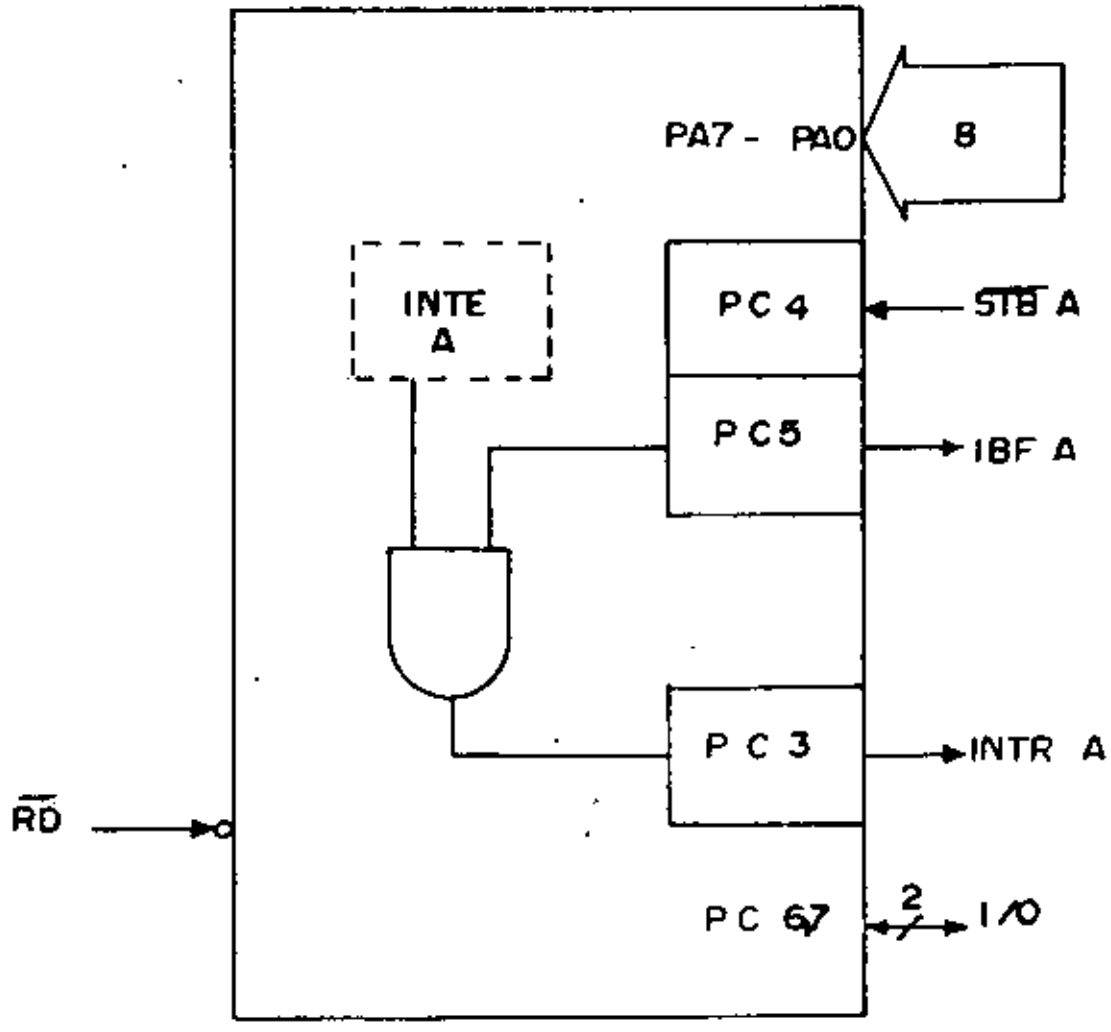


Fig. 4-6

be operated in a number of modes.

There are a number of other chips from Intel and other manufacturers that resemble the 8255. With the new 8085, an 8155 has been made available which has many of the 8255 features: two 8-bit A and B ports, 6-bit A CA port, basic and strobed input/output. In addition, it has a 14-bit binary down counter, and 256 bytes of RAM, and may be interfaced with the 8080.

Data Direction Registers

On some parallel I/O chips, individual bits may be designated as either an input or an output bit. One example is the Motorola PIA (6820). Here the chip may be divided into two parts, A and B. Each side or part is very similar as shown in Fig. 4-7. (The A output bits are standard TTL compatible, but the B output bits may source up to 1 milliampere, adequate for the base drive of a transistor. The Intel 8255 outputs also provide the 1ma. drive.) Although there are six internal registers, only four are addressable at any time. The data direction registers specify which bits are outputs or inputs. A 1 in a bit of the data direction register defines the corresponding external line as an output; a 0 defines an input. The control registers define and enable the interrupt states and with pins R0 and R1, the register select pins, select the I/O register or the data direction register. Specifically, it the 2nd bit of the control register is a zero, then (R0,R1) = (0,0) selects the data direction register. If the 2nd bit is one, (0,0) selects the I/O register. The control registers are always selectable. The register select is indicated in Fig. 4-8. While cumbersome, this scheme permits a pin that would ordinarily be used as register select, to be used for some other useful task. MOS Technology manufactures an excellent variation of the PIA: the 6530. It contains not only the

INTERNAL ADDRESSING OF THE PIA

RS1	RS0	Control Register Bit		Location Selected
		CRA-2	CRB-2	
0	0	1	x	Peripheral Register A
0	0	0	x	Data Direction Register A
0	1	x	x	Control Register A
1	0	x	1	Peripheral Register B
1	0	x	0	Data Direction Register B
1	1	x	x	Control Register B

RS0 and RS1 are the register select pins normally connected to address pins A0 and A1. CRA-2 and CRB-2 are bit 2 of the control registers.

Data Acquisition: Analog to Digital Conversion

In this example, we will consider the simple task of controlling an A/D converter and storing the binary data in consecutive memory locations. Note that the 6800 uses memory mapped I/O, and thus the registers in the PIA are addressed as memory locations. The A/D converts analog signals (usually a voltage or a current) into a binary number. When the A/D is issued a command pulse (0 to 1 to 0), it begins the conversion. When the conversion is finished, a "done" signal is issued by the A/D. Some A/D's perform the conversion very quickly, perhaps in a few microseconds. Other may take as long as milliseconds. If the A/D is fast and frequent conversions are required, the processor will have to be devoted to the tasks. If the A/D is slow, the processor may be used for some other task and then come back for the data or be interrupted from the other task to take care of the conversion and data. In this latter case, interrupt processing may be useful. Here we will consider the case where the processor is handling only the conversion. In Fig. 4-9, the flow chart and instructions for the 6800 are given.

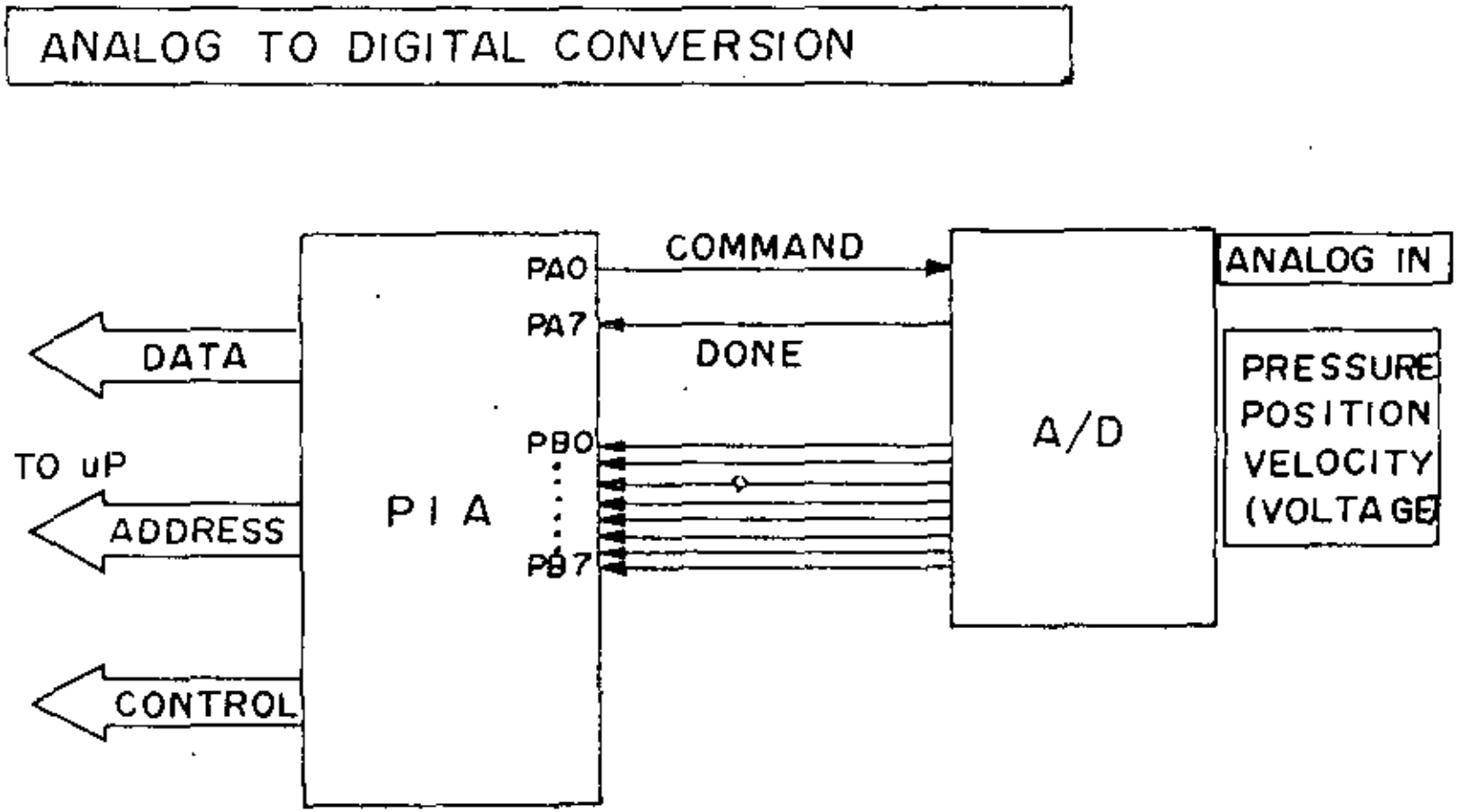

ANALOG TO DIGITAL CONVERSION
DATA
TO uP
ADDRESS
CONTROL
P I A
PA0
PA7
PB0
PB7
COMMAND
DONE
A/D
ANALOG IN
PRESSURE
POSITION
VELOCITY
(VOLTAGE

Fig. Fig. 4-8

6800 Code for the A/D Example

```
      LDAA    #01             ;initialize PIA, bit 0 an output
      STAA    @# 8008         ; store in data direction register
      LDAA    #04
      STAA    @# 8009         ;set control register A
      STAA    @# 800B         ;set control register B
LOOP: INC @#8008
      DEC @#8008              ;pulse the command line
DONE: TST @#8008              ;is the done bit set?
      BPL DONE                ;if not, go back to DONE
      LDAA @#800A
      STAA (using index reg.); index register used to store data
      INX                     ;step index register to next location
      CPI #xxxx               ;all data acquired?
      BNE LOOP                ;if not, go back to LOOP
```

Fig. 4-9

The A/D is one example of a common task where the processor must examine a strobe bit and store the date for a certain number of samples. A question of important is, how fast can the processor acquire data while keeping track of data samples and the count? The code for the 6800 is shown in Fig. 4-10. The processor may handle more than one such task. The Motorola Applications Book describes and analyzes this multiple task.

DATA TRANSFER PROGRAM

This program segment is intended to indicate the maximum rate at which a 6800 may acquire data. Here, we are given a control register (CNTRL) which indicated that status: if bit 7 is high, the data word (DTWR) is ready to be read. The index register has been preloaded with the number of words to be acquired andi is decremented until it is zero in which case the acquisition is done.

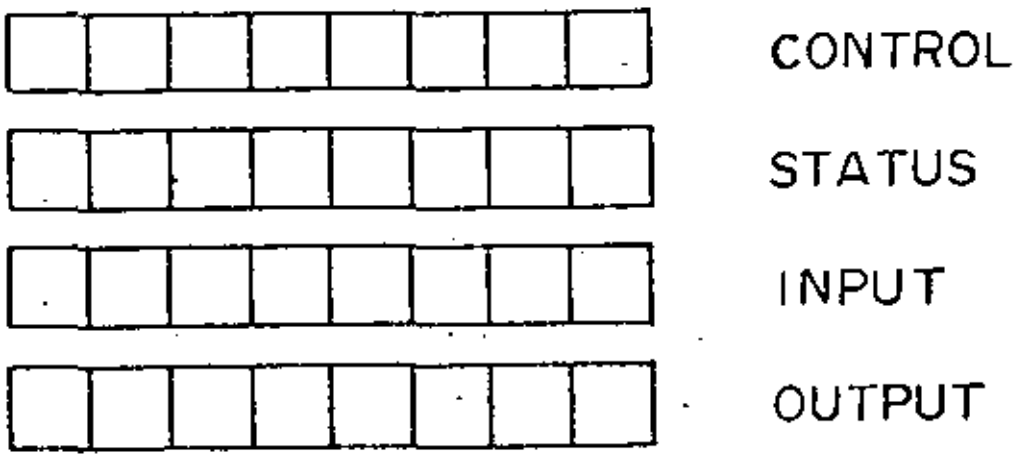
```
LOOP:   LDAA    CNTRL       ;load control word      4 cycles
        BPL     LOOP        ;data not ready         4
        LDAA    DTWD        ;get data into accum A. 4
        PSHA                ;store on stack         4
        DEX                 ;Index reg. has  count  4
        BNE     LOOP        ;return for next word   4
```

Fig. 4-10

Here it is shown that 24 clock cycles are required to test the status, acquire the data, and count the number of acquisitions. For a 6800, 24 cycles correspond to 24 microseconds or 12 microseconds depending on which speed processor is used. While adequate for many tasks, this acquisition rate is too slow for other tasks. It is sufficiently fast to acquire data from a floppy disc if the data is available in bytes, but too slow to provide the bit to byte interfacing for the disc itself. It is fast enough to acquire data from a cassette tape recorder bit by bit.

Serial Input/Output Chips

Not long ago, communication between the processor and a teletype or CRT was handled by a U/ART, a p-MOS LSI chip which converts parallel data from the processor into serial data and converts serial data from the terminal into parallel data. This chip is "programmed" by connecting jumpers to +5 volts or to ground. It is very flexible and in fact is still used (in hobby computers and in the LSI-11 for example. Again, Motorola was the first to produce a microprocessor compatible chip that can be programmed by the processor and not by external connections. These chips have various names: ACIA (asynchronous communication interface adapter), USART (Intel's universal synchronous asynchronous receiver transmitter). A typical version is shown in Fig. 4-11. The four internal registers are selected by the register select line and the read/write line. The control register and the transmitter buffer are write only and the status register and read buffer are read only. The chip is programmed by depositing the appropriate code in the control register. Specific bits determine the number of stop bits, parity, number of data bits, and together with the transmitter and receiver clocks, the baud rate. An interrupt on the reception or complete



CONTROL REGISTER - WRITE ONLY

STATUS REGISTER - READ ONLY

I/O REGISTERS - DOUBLE BUFFERRED

Fig. 4-11

transmission of a character may also be provided. Thus the processor may perform some other task while the transmission of a character is carried out or while waiting for a character to be received. The function of the individual bits is illustrated in Fig. 4-12. The input and output of the ACIA are serial strings of bits shown in Fig. 4-13. The start bit and stop bits act as timing signals for the teletype or CRT.

The ACIA can be connected to a modem (modulator-demodulator) chip so that the serial information can be transmitted over telephone lines. The modem provides the frequency shift keying (FSK) over the lines; it can be operated in an originate or an answer mode. It originates 1's and 0's (called marks and spaces) at 1270 and 1070 Hz respectively and answers at 2225 and 2025 Hz repectively.

The U/ART or ACIA type chips are not always satisfactory for teletype communication since there is no provision for reader control. [6] Shown in Fig. 4-14 is a scheme used by Motorola in their evaluation kits. Certain output bits send and receive the serial data and others turn on or off the reader relay. However, in this scheme, the processor is responsible for converting the parallel character into serial bits. It can do this by shifting the bits out one by one. The processor will therefore have to dedicate itself to this task. If the processor is supposed to handle other tasks, the ACIA or its equivalent is necessary. (Transmission to a teletype requires 100 milliseconds per character or about 9 milleseconds per bit. This slow rate would permit some time for execution of a task, but for higher baud rates, the processor would have to spend

---

6. In some teletypes, provision is made for control of the paper tape reader using special coding bars. Certain control characters then can turn the reader on and off.

TRANSMIT DATA REG.
RS · R/W
WRITE ONLY

7	6	5	4	3	2	1	0
DB 7	DB 6	DB 5	DB 4	DB 3	DB 2	DB 1	DB 0

RECEIVE DATA REG.
RS · R/W
READ ONLY

7	6	5	4	3	2	1	0
DB 7	DB 6	DB 5	DB4	DB 3	DB 2	DB 1	DB 0

CONTROL REG
RS · R/W
WRITE ONLY

7	6 5	4	3 2	1 0
INT. ENABLE	TRANSMIT CONTROL	# OF STOP BIT	WORD SELECT BIT PARITY	COUNTER DEVIDE (÷1 ÷16 RESET)

STATUS REG
RS · R/W
READ ONLY

7	6	5	4	3 2	1 0
INT. REQ.	PARITY ERROR	REC. OVERRUN	FRAME ERROR	MODEM STATUS	TRANS-MIT DATA REG

Fig. 4-12

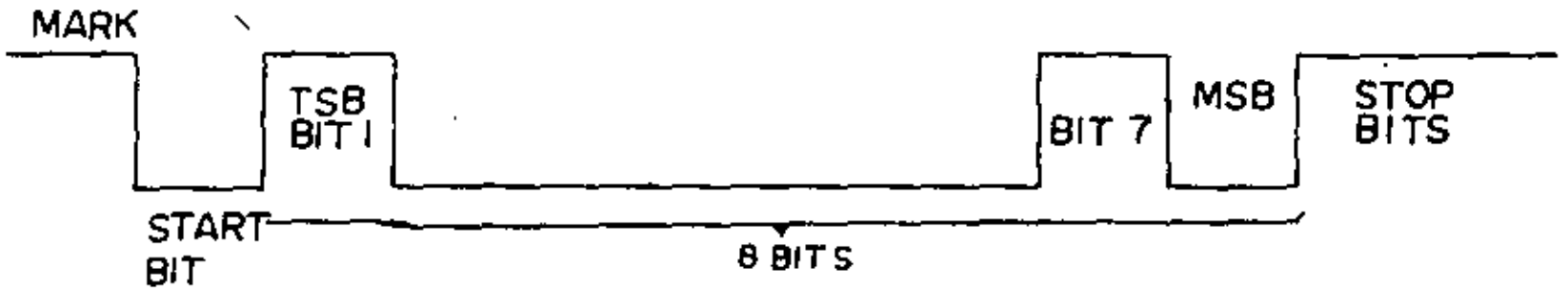


Fig. 4-13

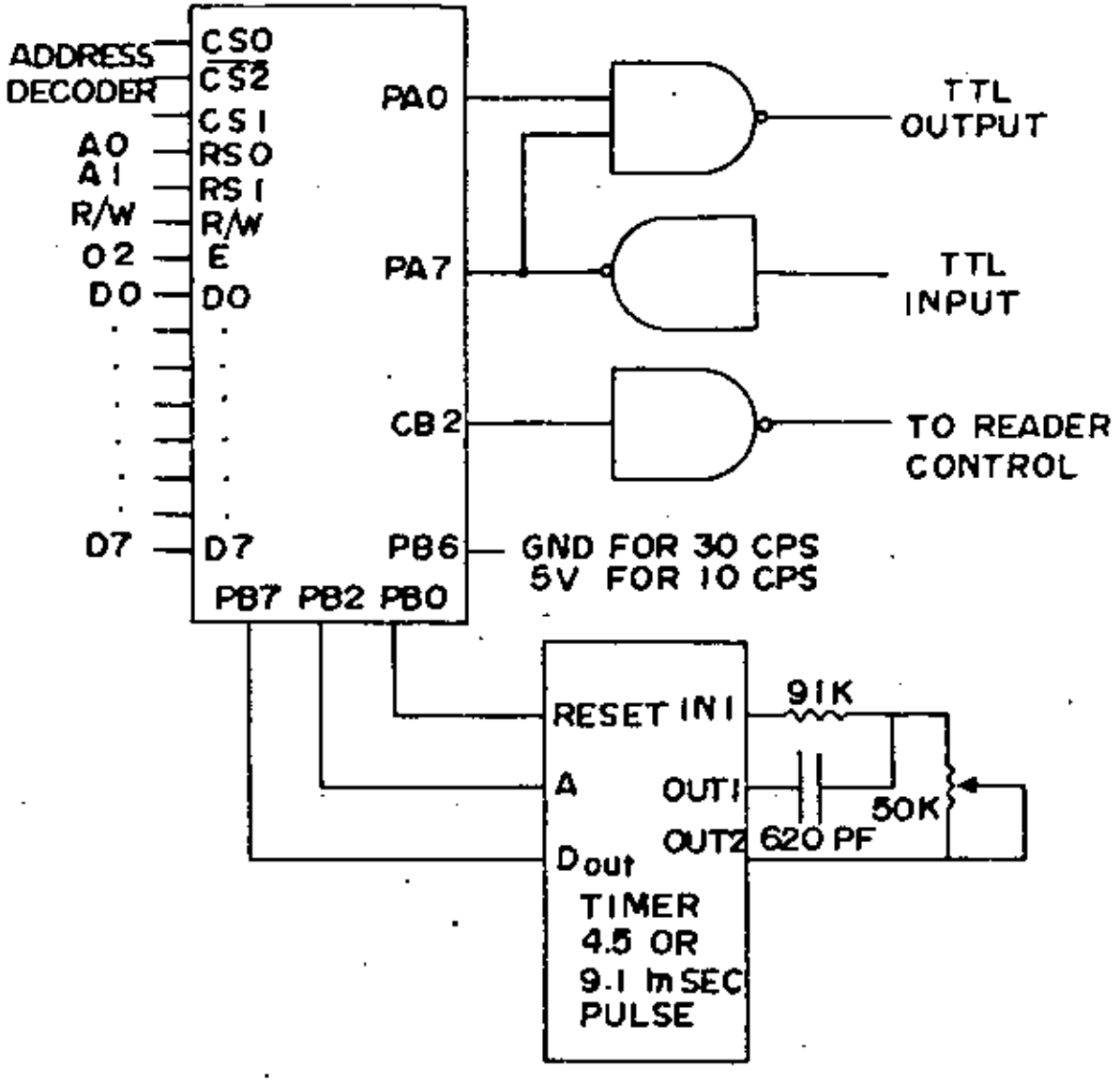

ADDRESS DECODER
CS0
CS2
CS1
RS0
RS1
R/W
E
DO
D7
A0
A1
R/W
O2
DO
D7
PA0
PA7
CB2
PB6
PB7 PB2 PB0
TTL OUTPUT
TTL INPUT
TO READER CONTROL
GND FOR 30 CPS
5V FOR 10 CPS
RESET IN1
A
Dout
OUT1
OUT2
91K
620 PF
50K
TIMER
4.5 OR
9.1 mSEC
PULSE

Fig. 4-14

most of its time shifting and timing the bits.)

Interrupt Handling

Interrupts provide a processor with the capacity to handle more than one programming task. Usually one task is the "main program" and the other(s) is a service routine which is performed on request. An example is a microprocessor program where the processor is supposed to monitor a process (and perhaps also provide control signals) and print information or data on a terminal. Here, the processor may, under program control, enable interrupts on the serial communication interface so that the processor may monitor the process most of the time and only occasionally divert to the terminal program and transmit the next character out. If the monitoring program cannot be interrupted at certain times, the processor can disable the interrupt and devote full time to its main program. When the interrupt is enabled, the processor on interrupt request finishes its current instruction, saves enough information so that it can return to the main program (e.g. the program counter and the condition codes), and jumps to the interrupt routine. On entering the interrupt routine, the processor disables further interrupts and begins execution. When finished, it returns to the main program by restoring the saved program counter and any other registers information saved before leaving the main program. The processor may enable an interrupt on entry to an interrupt routine and thus interrupts of interrupts (nesting) may occur.

Minicomputers such as the PDP-11 handle interrupts in sophisticated ways. These processors are relatively expensive and powerful, thus multitask programming is very common. The interrupt capabilties of most microprocessor are similar to those of minicomputers (or can, by additional circuitry, be made

similar), however the designer should consider the possibility of using several processors rather than coping with complicated multitask programming. Microprocessor are less powerful than minicomputers, the software presents more difficulties, and they cost much less. Of course, if the data or control rates are very slow and simple, a microprocessor can handle many tasks.

Typical Interrupt Procedures

Most interrupts are maskable, that is, the microprocessor's condition code register contains a flag bit which enables or disables (masks) interrupt requests. When the interrupt in not masked, and an interrupt request is made, the processor finishes its current instruction and inters the service routine. The flow chart of a 6800 is shown in Fig. 4-15. Note that there are two interrupt requests: a nonmaskable interrupt (one that cannot be disabled by the processor) and a regular interrupt request. We will discuss only the latter. When the processor finishes an instruction, it checks for a halt signal and then an interrupt request. If the interrupt request is asserted, it then checks the mask. If the mask is set, the processor continues with the next instruction. If the request is not masked, the processor saves all of the internal registers on the stack (excluding of course the contents of the stack pointer) and goes to locations FFF8 and FFF9 for the new program counter. If several devices are capable of interrupting the processor, the service program must POLL the devices to determine which device created the interrupt request. A "priority" of the requests may be established through the order in which the devices are checked. In the 6800 family, the I/O devices have a status register or status/control register which contains a flag bit that signals an interrupt request. Thus when an I/O device is programmed, interrupt enable bits in the control register may be set to permit interrupts by that I/O device.(Fig. 4-16). The Motorola scheme

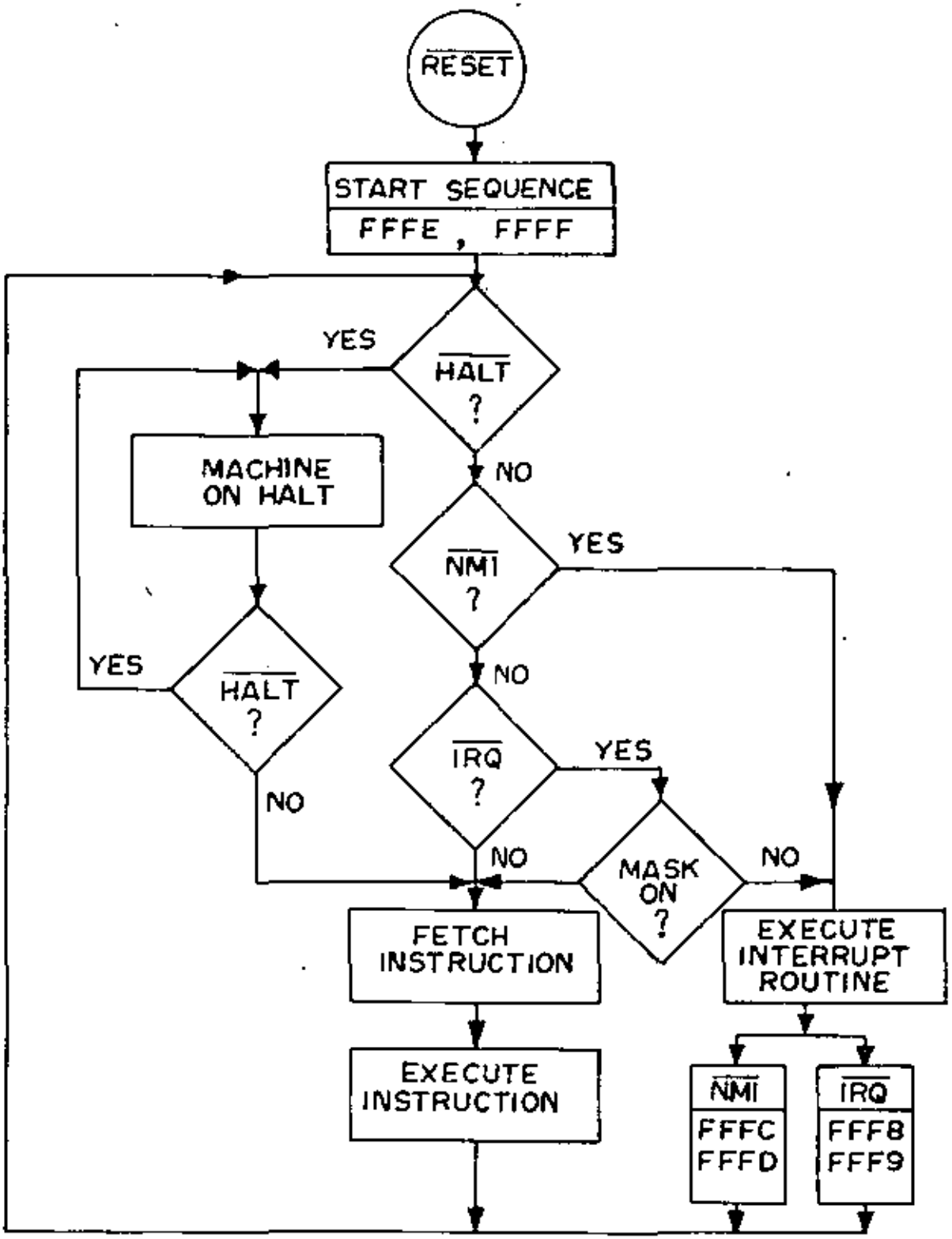

RESET
START SEQUENCE
FFFE , FFFF
HALT ?
YES
NO
MACHINE ON HALT
NMI ?
YES
NO
HALT ?
YES
NO
IRQ ?
YES
NO
MASK ON ?
NO
FETCH INSTRUCTION
EXECUTE INSTRUCTION
EXECUTE INTERRUPT ROUTINE
NMI
FFFC
FFFD
IRQ
FFF8
FFF9

Fig. 4-15

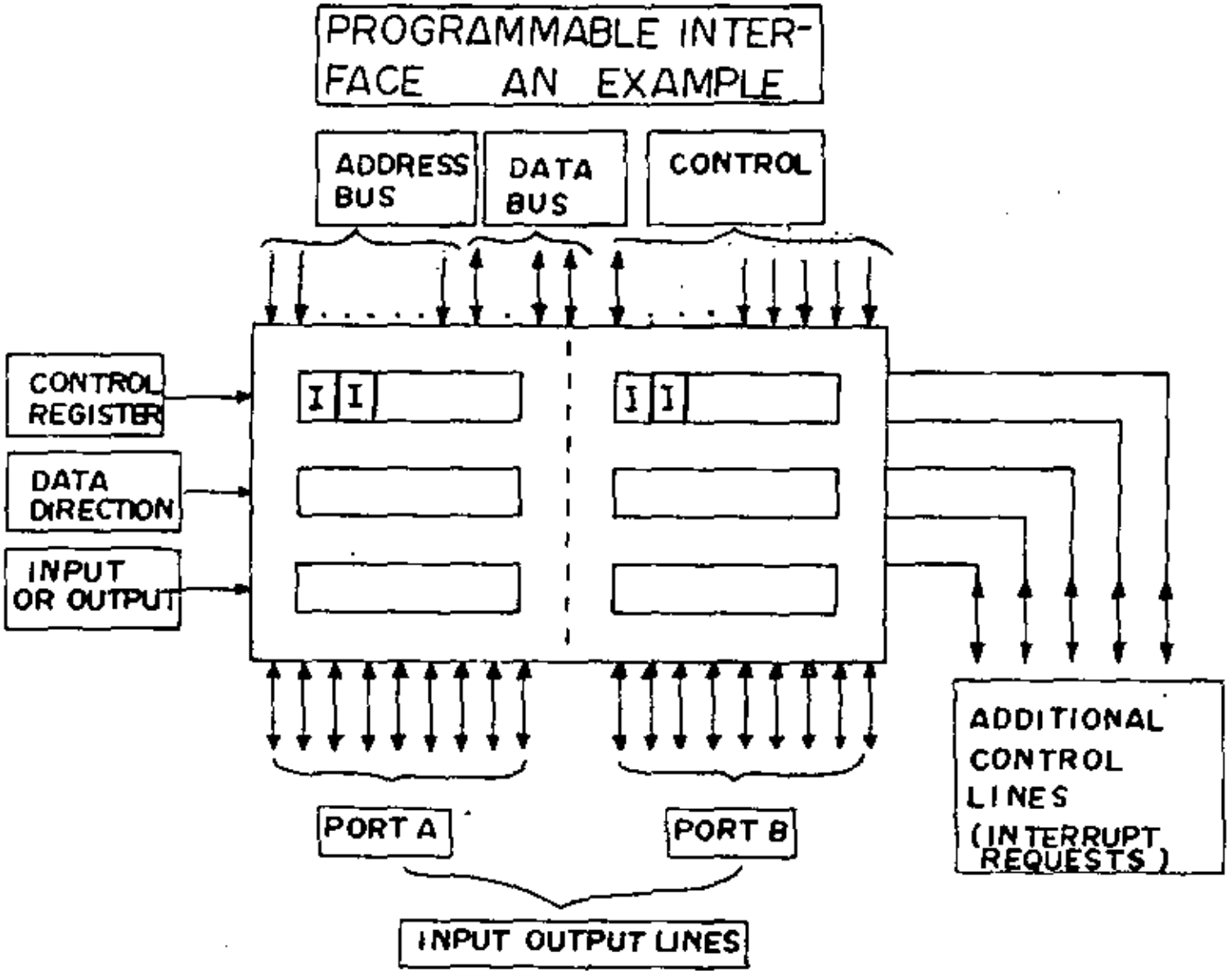

PROGRAMMABLE INTER-
FACE AN EXAMPLE
ADDRESS BUS
DATA BUS
CONTROL
CONTROL REGISTER
DATA DIRECTION
INPUT OR OUTPUT
I I
I I
ADDITIONAL CONTROL LINES (INTERRUPT REQUESTS)
PORT A
PORT B
INPUT OUTPUT LINES

Fig. 4-16

minimizes the hardware for interrupt processing, however, the search for the interrupting device does require additional time. Considerable time may be saved if the processor is told or can quickly determine which device is interrupting and thereby go directly to the service routing. This is called VECTOR interrupt.

Vector Interrupt Example

In the 8080, the processor is fed a particular address corresponding to the particular interrupt service routine. This saves time at the expense of additional hardware. The 8080 itself allows eight possible interrupts. When the interrupt request is received and the interrupt enable bit is set, the 8080 expects to see a RESTART instruction.

11NNN111

When this restart command is received, the PC is saved on the stack, and the processor will go to location 8*NNN (locations 0,8,16,32,40,48 and 56) for the beginning of the interrupt routine. [7] Location 0 corresponds to reset External circuitry is required for the generation of the RST command. For a single interrupt, this can be accomplished using an 8-bit input port that is enabled when the processor acknowledges an interrupt request (INTA). The circuitry for several devices is considerably more complicated. Intel produced the 8214 priority control unit shown in Fig.4-16, which fed a particular RST command for each request on one of the inputs to the 8214. If two devices simultaneously

---

7. Note that the 8080 is essentially coded in octal rather than in hexadecimal, and thus when the instructions are expressed in hexadecimal, they have a rather chaotic relationship.

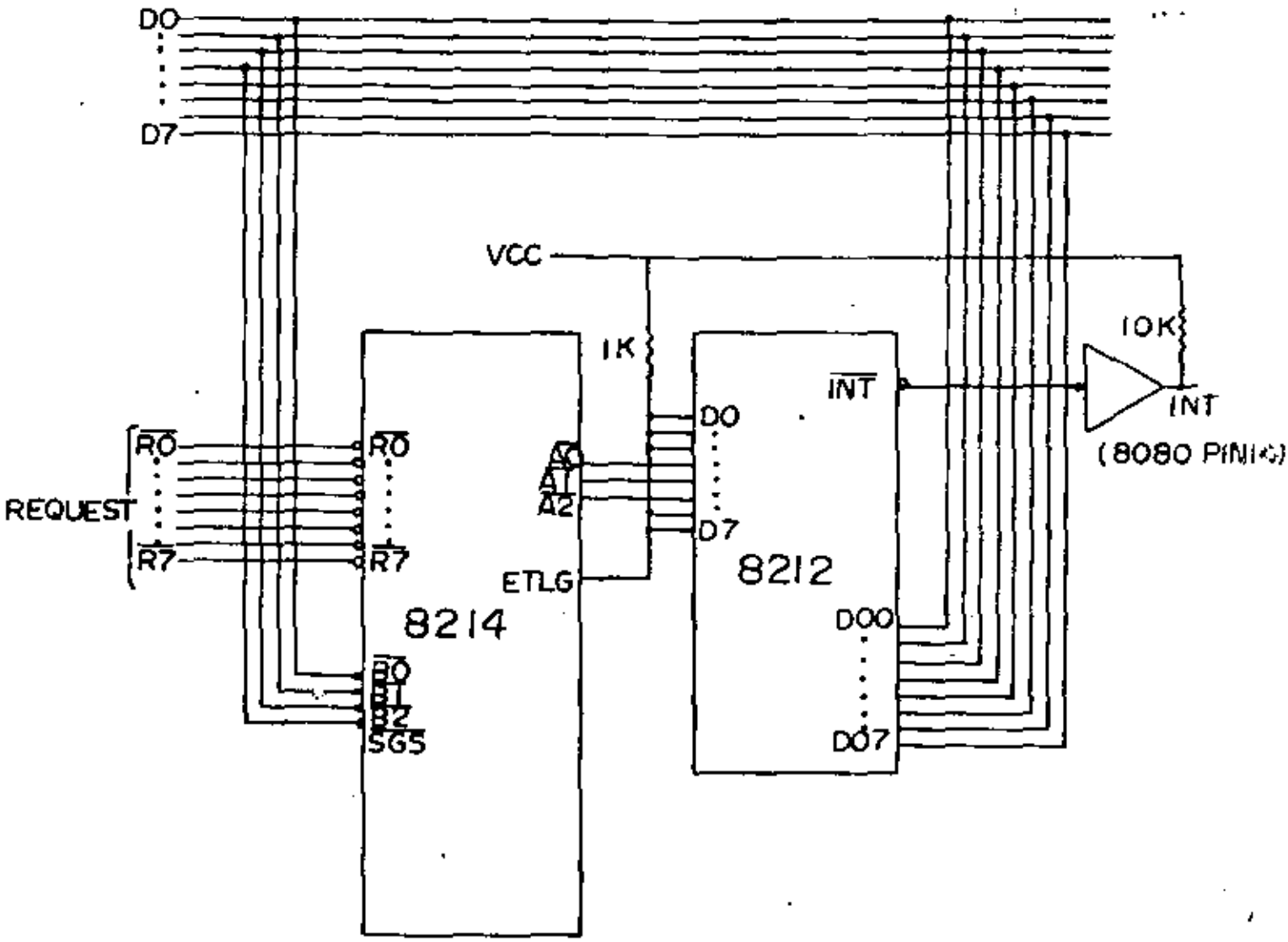

D0
D7
VCC
1K
10K
INT
INT
(8080 PINIC)
REQUEST
R0
R7
8214
A0
A1
A2
ETLG
SGS
8212
DO
D7
DOO
DO7

Fig. 4-16

request an interrupt, the one with higher priority will be recognized first. Again, the 8214 issues the appropriate RST instruction on (INTA) from the processor.

Intel's 8259 provides programmable priority interrupt which can be cascaded to 64 levels. On INTA from the 8228, the 8259 will issue a CALL (Subroutine) op-code which in turn will cause the 8228 to issue two more INTA to fetch the complete 3 byte subroutine call instruction. The 8259 may be programmed so that any memory location may be specified for the start of the interrupt routine and so that desired priority is established.

Other Programmable Chips

Other chips include "number crunchers" which are essentially programmable calculators for processors. Specific chips have been announced for the 8080 and for the SC/MP of National. When the cruncher finishes with the calculation, it can interrupt the processor in the manner described earlier. Floppy disk interface chips, special serial and parallel interface chips, crt controllers, keyboard/display interface chips and SDLC protocol controller chips have also been developed. Basically their operation is determined by a control register and simple commands from the processor accomplish the desired operation(s). As discussed earlier, these chips extend the capabilities of the processor, unburden the processor, and simplify the software and hardware design. In this section we will only look at a direct memory access (DMA) chip that simplifies the software and hardware of the DMA problem discussed in Chap. 3 (Memories).

The DMA chip in question is the 8257 DMA controlled of Intel. This device permits four different channel inputs to memory. It autocatically generates

sequential address locations for the input or output and uses the processor HOLD feature discussed earlier. Priorities may be set for each of the channels or a rotating priority may be established. Finally, a "terminal count" or the number of bytes that are to be transferred in a single request may be programmed. A disadvantage of using this chip is that a maximum transfer rate of 1 byte/4usec. is possible. For many instrumentation tasks, this rate is much too slow. A hardwired DMA unit as discussed earlier, can handle data as fast as the memory access time will allow. However, in this latter case, that hardware complexity is much greater than the complexity of a single chip.

CHAPTER 5. SIMPLE PROCESSORS

In this chapter, we will discuss the Intel 4048, the RCA 1802, the National SC/MP, and to a limited degree, the Intel 8085. I have classified these processors as simple because they provide I/O lines on the CPU chip, and thus they can be used to design systems with very few components. The processors are not necessarily limited either in the degree of system complexity or in the degree of system expansion or capabilities however. I have, perhaps unfairly, not included the F8 in this chapter. In most respects, the operation of these processors is similar to that discussed in Chap 2, and the interfacing of the processors to their "family" members is very similar to those techniques discussed in the chapters related to I/O or memory. Only those features of these simple processors and some applications will be illustrated here.

The 8048 - A One-Chip Microcomputer Intel's 8048 is the newest and perhaps the premier member of this group of simple processors. It has 1Kbyte of ROM, 64 bytes of RAM, a time/event counter, and 27 I/O lines. The 8748 version has eraseable ROM (PROM) while the 8048 is mask programmed at the factory. A schematic is shown in Fig. 5-1.

In developing this processor, Intel seems to have departed from the 8008-8080-8085 series of software compatibility. But, the processor may be interfaced to the 8080/8085 family or system components discussed in the previous chapter on I/O in addition to its own, specially developed system chips. Any of the internal RAM memory locations can be addressed using one of the two

8048

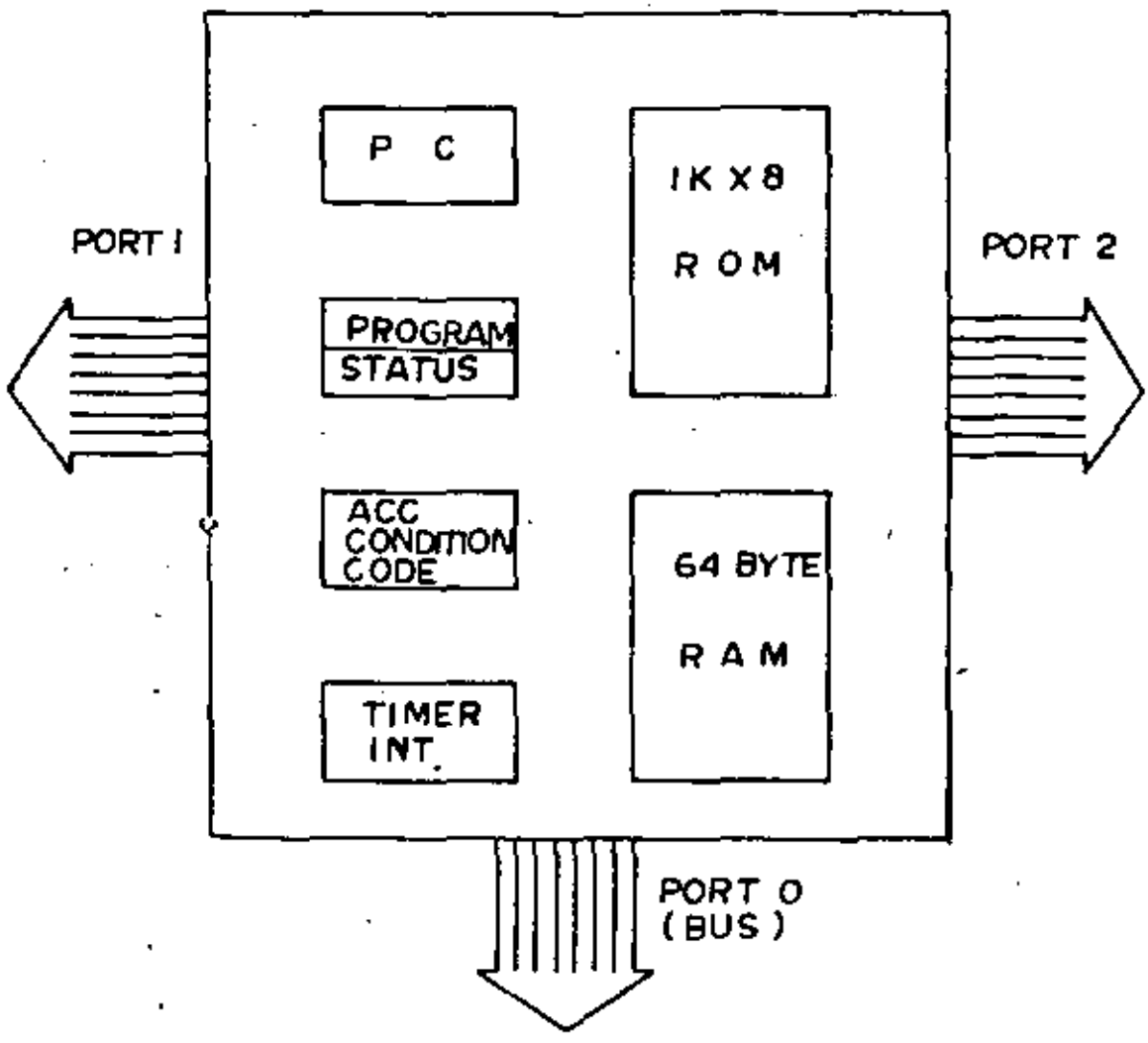

P C
1K X 8
ROM
PORT 1
PORT 2
PROGRAM
STATUS
ACC
CONDITION
CODE
64 BYTE
RAM
TIMER
INT.
PORT 0
(BUS)

Fig. 5-1

registers, R0 or R1. In addition, any of the first eight registers can be directly addressed using several instructions. These are called "working" registers. A second bank or working registers can be selected by using a register bank switch instruction. Additional, external RAM and ROM may be added using conventional memory or using the special I/O chips that contain RAM or ROM.

The instruction set contains many conventional operations: loading and storing the accumulator, conditional-jumps on flags or condition codes, etc. A new instruction that I have not seen before is a conditional jump instruction depending upon is a specified bit in the accumulator is set. This instruction could be very useful in quickly handling some I/O problems.

The 8048 has three I/O ports or buses. Two are called "quasi-bidirectional ports" because they may act as latching output port or non-latching input ports without defining instructions or programming. This feature is possible because of a special hardware design. Thus one simply connects inputs to the lines as desired and connects others as outputs. The third bus is simply called "bus" and is a bidirectional bus in the usual manner. It is through this bus that system expansion is accomplished.

A simple application of the 8048 is shown in Fig. 5-2 and 5-3. Here, the 8048 is used as an intelligent terminal; a "video display module"[8] is used for the display. The modules display the characters on standard monitors in a variety of ways. For example 32 characters by 16 lines, 80 characters by 25 lines, etc. are available The module appears to the processor as standard RAM; hence it can be read and written. The monitor displays the converted ASCII data

8. Available from Matrox Electronic Systems and other companies.

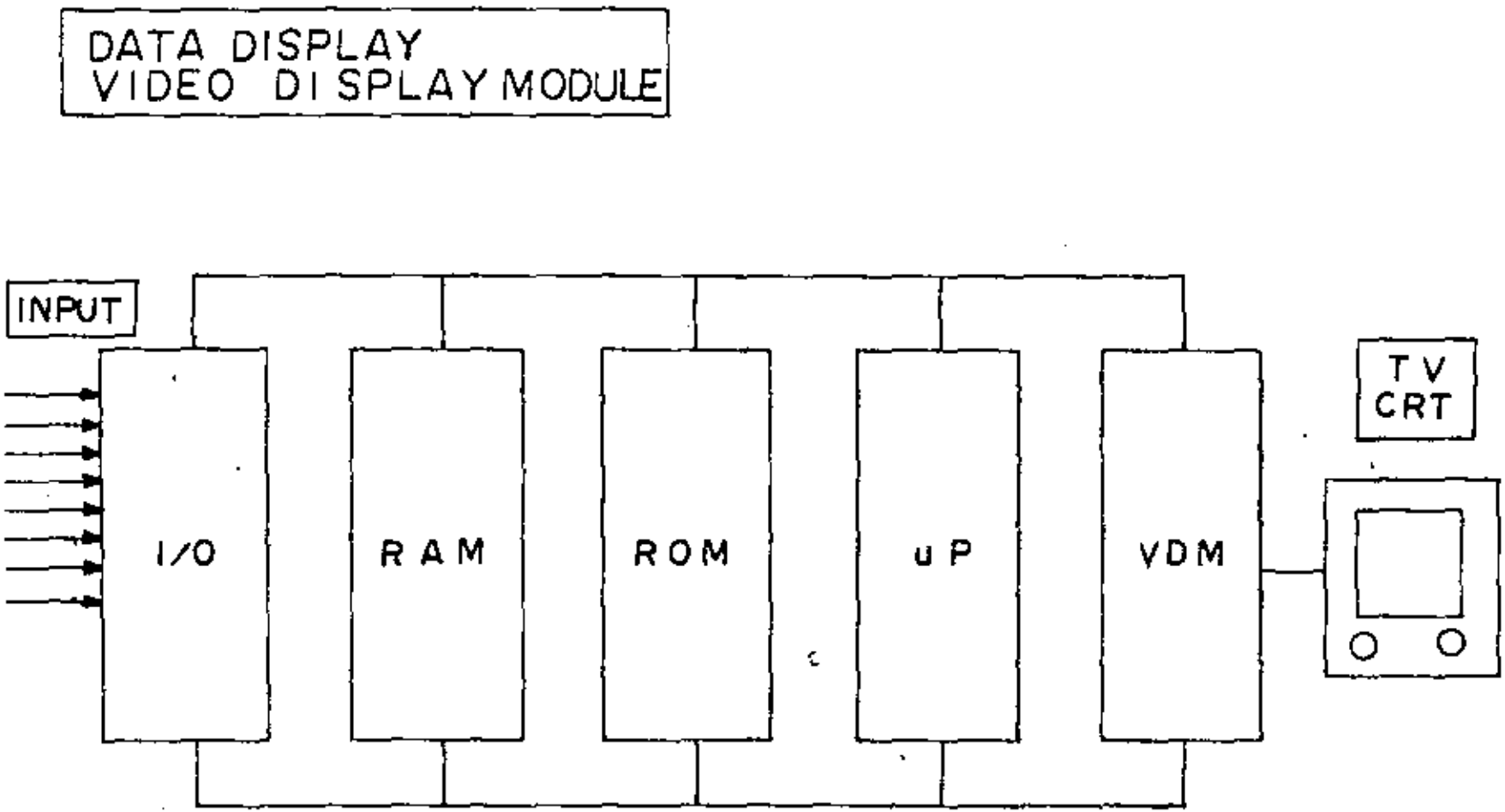
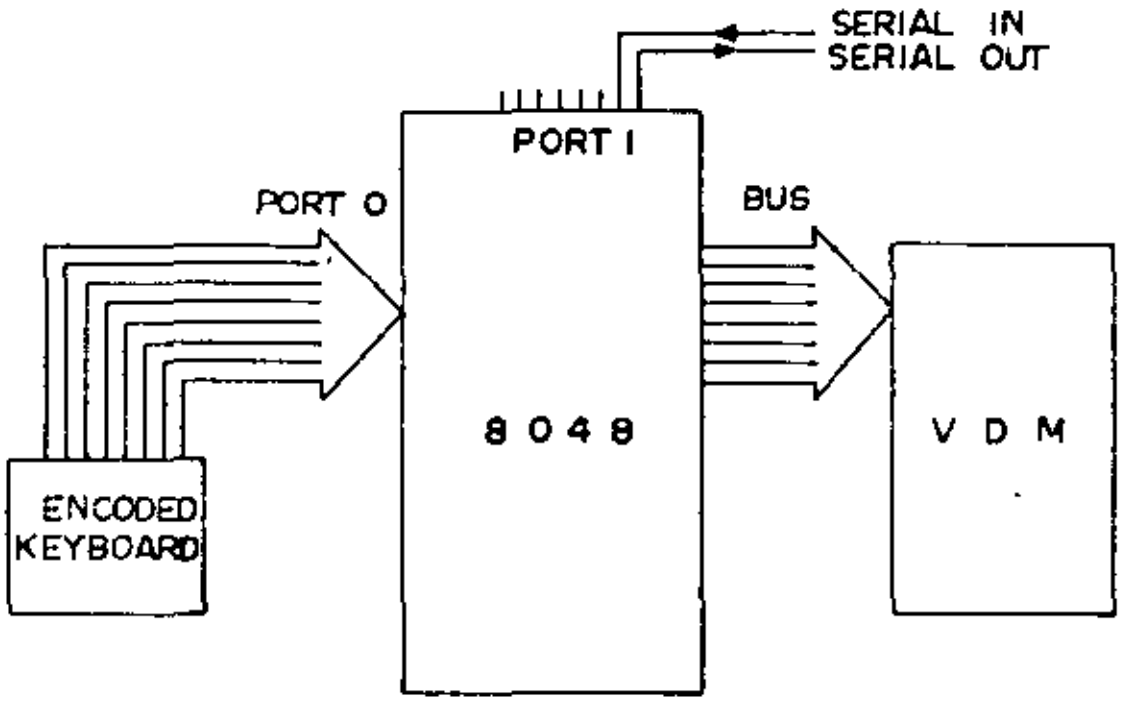

DATA DISPLAY
VIDEO DISPLAY MODULE
INPUT
I/O
RAM
ROM
uP
VDM
TV
CRT

Fig. 5-2



8048 APPLICATION

Fig. 5-3

of the RAM. A keyboard may be interfaced to the 8048 using the 8279 keyboard/display or a standard, encoded keyboard may be connected to one of the quasi-bidirectional ports since the VDM already supplies the display. The serial I/O can be provided by using the other quasi-bidirectional port. To generate to proper baud and handle the serial I/O, the processor may use its internal counter for timing.

The 1Kbyte ROM and 64 bytes of RAM are adequate for the necessary program. The processor must spend most if its time monitoring the serial input and output and echoing that response on the VDM. When a key is struck, an interrupt will occur that results in a read of the keyboard encoding chip and transmission of the character via the serial output. Since the internal timer increments every 80 microsec., this scheme is satisfactory up to 600 baud. For higher baud, a serial interface chip such as the 6800 discussed in the previous chapter is required.

Intel's 8085

The 8085 was mentioned in Chp.3 as an extension or simplication of the 8080. In the 8080, status information must be placed on the bus and be captured using an additional chip, the 8228. The 8228 possesses some additional features that make it very useful, but if one is trying to use the minimum number of components, the 8085 system is better. The 8085 has all of the instructions of the 8080 and can be used in similar systems but is not bus compatible. One must use some logic chips to connect the 8085 to an 8080 bus.

The 8085 offers some simplication of system design, especially if the system does not require many priority interrupt request lines. For instance, the

8085 has five interrupt request lines on the chip:

INTR- general purpose interrupt request

RST5.5

RST6.5

RST7.5 These are all interrupt requests which result in internal restarts, restarts at particular locations. RST7.5 has the highest priority, RST5.5 the lowest.

TRAP A nonmaskable restart instruction that restarts at 24(Hex). It is not affected by any interrupt masks and can be used for power down or other "catastrophic" error

The 8085 also has two I/O pins on the package, a serial in and a serial out pin. The output is set by setting bits 6 and 7 of the accumulator to the appropriate state and then using a SIM, set interrupt masks. The serial in datum is read by a RIM, read interrupt mask, and found in bit 7.

The COSMAC: RCA 1802

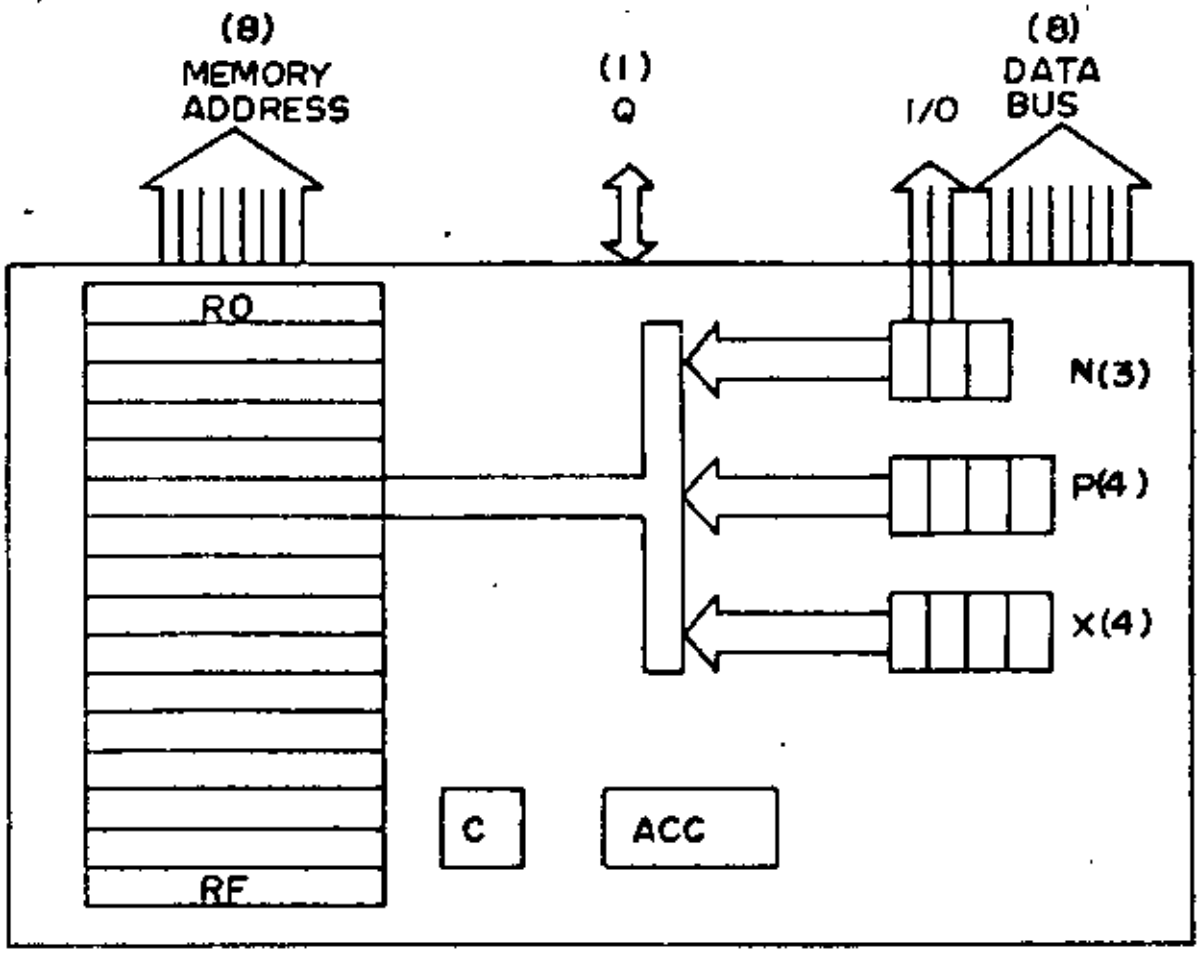
Actually, RCA probably produced the first 8-bit processor but delayed its announcements and production because it had recently divested its computer division and apparently was not willing to invest in another "computer". The COSMAC (a name which is not always used to describe the 1802 even in RCA literature) is

a CMOS (complimentary MOS logic) processor, and thus has some unique features. First, the processor can be operated with a wide range of voltages: 3 - 15 volts. Second, the processor requires extremely low power- microwatts instead of hundreds of milliwatts for more conventional processors. Thus the processor and its associated chips can be operated in applications requiring battery power or very low power input much more successfully than, say, the 8080.

Other special features of the COSMAC include a static clock (no minimum clock frequency); simple input/output either memory mapped, on-chip DMA, or I/O lines on the chip; and 16 16-bit registers, any one of which may be the program counter, index register, or DMA pointer. The processor is thus I/O oriented. Each of these I/O features will be examined in the following.

The processor has four input flags which which may be tested and used for conditional branches and it has 3 output lines which may be used for simplified address decoding or device selection or for additional I/O lines. Finally, a one-bit I/O line is available for serial input/output. Some details are shown in Fig. 5-4.

The processor has a DMA request input that essentially provides "cycle stealing" capabilities. If the request is made frequently enough or if it is continuously made, the processor will continue to provide DMA in or DMA out. This feature is very useful for loading memory with the desired program and then executing the program. RCA has made available a "Microtutor" that uses this scheme. The DMA feature is handled by Register 0; i.e. R0 points to the memory locations that are loaded through the DMA requests and is automatically stepped to the next memory location. On RESET, R0 also serves as the program counter, but any register may thereafter be designated the program counter.



COSMAC

Fig. 5-4

Any of the sixteen 16-bit register may be designated as an index register. Basic input and output is handled through an IN or an OUT instruction which uses an index register as a pointer to input or output a byte. The index register is automatically stepped in this procedure. Further a 3 bit code is specified on the output bits for device selection. The byte is placed on the data bus by the processor in an output operation, or the processor reads the byte on the input operation. With additional external logic, the I/O capabilities may be greatly expanded. HCA has also developed peripheral chips for this processor. The additional I/O lines are possible because only 8 pins are used for the address bits. The 16 address bits are time multiplexed; an additional bit is provided so that the first (most significant) address bits can be latched. This address multiplexing is common on many of the simpler processors.

The SC/MP: National's SCAMP

The original SC/MP was fabricated using a p-MOS technology, and thus it was quite slow: 32 microseconds per instruction. A new n-MOS version is now available and is correspondingly faster. The SC/MP has 12 address bits on the chip but can multiplex the additional four for a full 16-bit address using four bits on the data bus. A timing or latching signal is issued by the processor when the additional four bits are valid. The internal register organization is illustrated in Fig. 5-5. The program counter is hardwired as pointer register 0. However, the contents of the PC can be exchanged with the contents of any pointer register using a single instruction. This feature leads to an interesting way to handle subrountines: if the pointer registers are pointing to the beginning of a subrountine, then by exchanging contents, the processor executes the subroutine. The return from subroutine is accomplished by another exchange

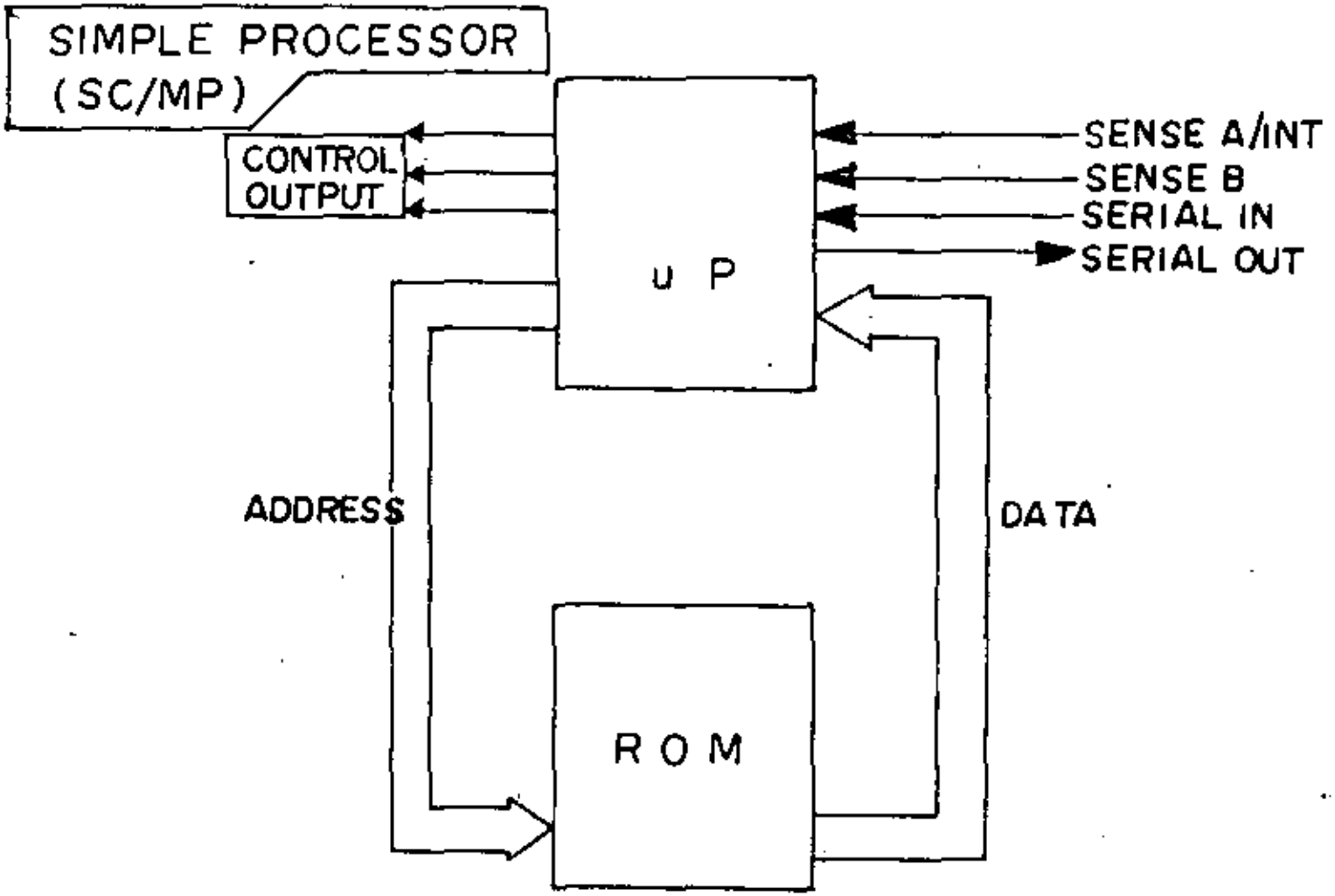

SIMPLE PROCESSOR
(SC/MP)
CONTROL
OUTPUT
u P
SENSE A/INT
SENSE B
SERIAL IN
SERIAL OUT
ADDRESS
DATA
R O M

Fig. 5-5a

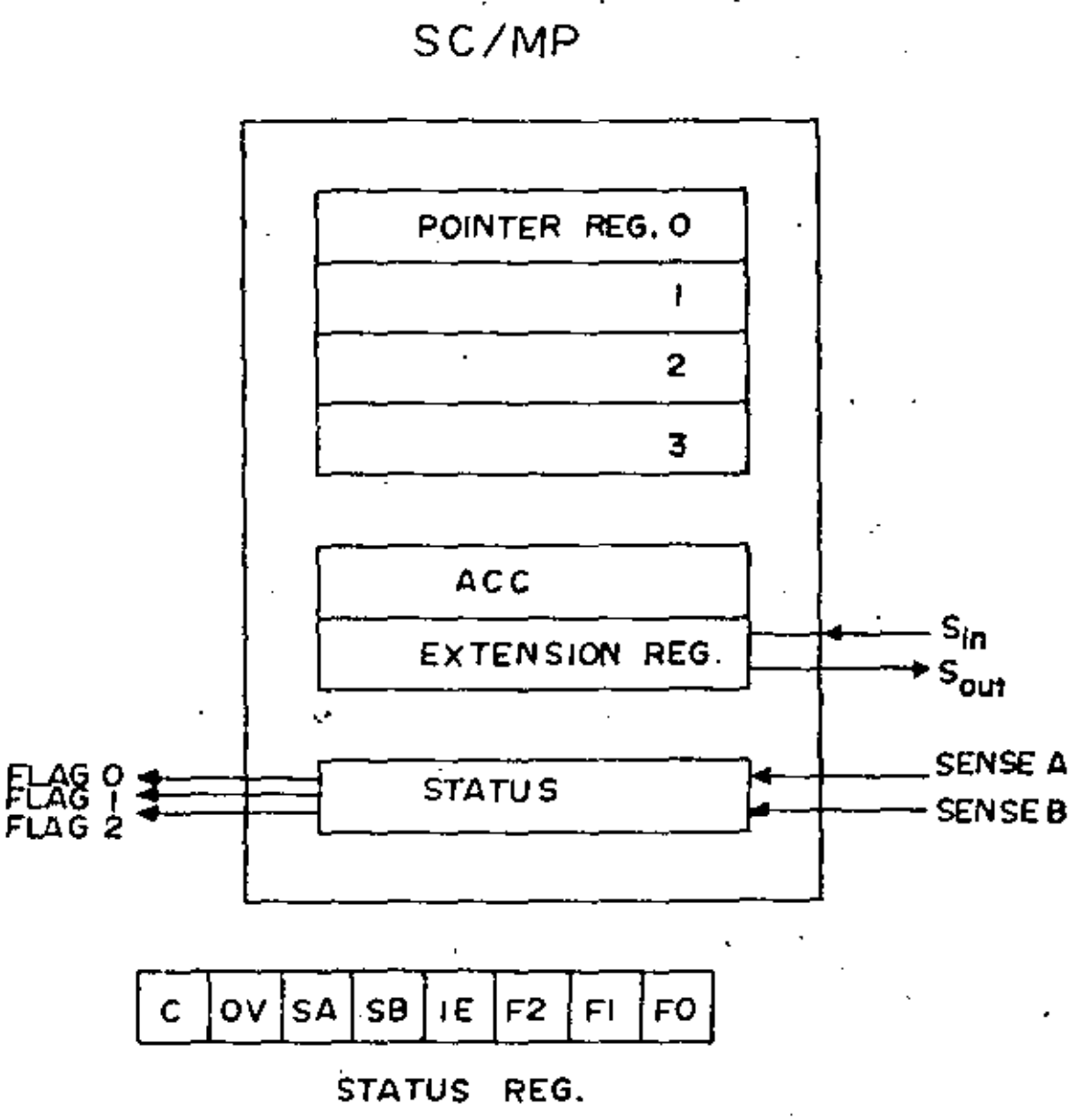

SC/MP
POINTER REG. 0
1
2
3
ACC
EXTENSION REG.
S_{in}
S_{out}
FLAG 0
FLAG 1
FLAG 2
STATUS
SENSE A
SENSE B
C
OV
SA
SB
IE
F2
F1
F0
STATUS REG.

Fig. 5-5b

of contents. In addition to a conventional accumulator, an extension register is avaliable for arithmetic, logical, and data-transfer instructions. Its use will be illustrated below.

The SC/MP has several I/O lines on the chip. First, it has serial in and serial out pins. Under software control, these signals can be shifted in and out of the extension register using one byte instructions. Second, there are three flag outputs that can be used for almost any purpose. These flags are set through "copy" instructions which copy the contents of the accumulator into the status register and vice versa. Two additional input bits, Sense A and Sense B are available for user purposes, and Sense A may be used for interrupt requests. The sense bits are available to the processor through the status register.

The SC/MP has an unique DELAY command which can, through a two byte instruction, delay from tens of microseconds to seconds without using programmed delay loops.

A simple application for the SC/MP is shown in Fig. 5-5. This application is a microprocessor controlled welding system. Here the processor must control the output positions of the welding head, perhaps through stepping motors, the welding current, and in turn acqire enough information about the processes so that a desired accuracy may be obtained. Since one might wish to direct the processor to perform a variety of welds, provisions are made for keyboard entry by the operation. The processor also echoes the operator's commands and provides information on the state of the system when it is operating. While this system would have once been complicated, microprocessors and their associated components would greatly simplify a design. Almost any processor would do, certainly one that is I/O oriented would help. A processor with family chips that

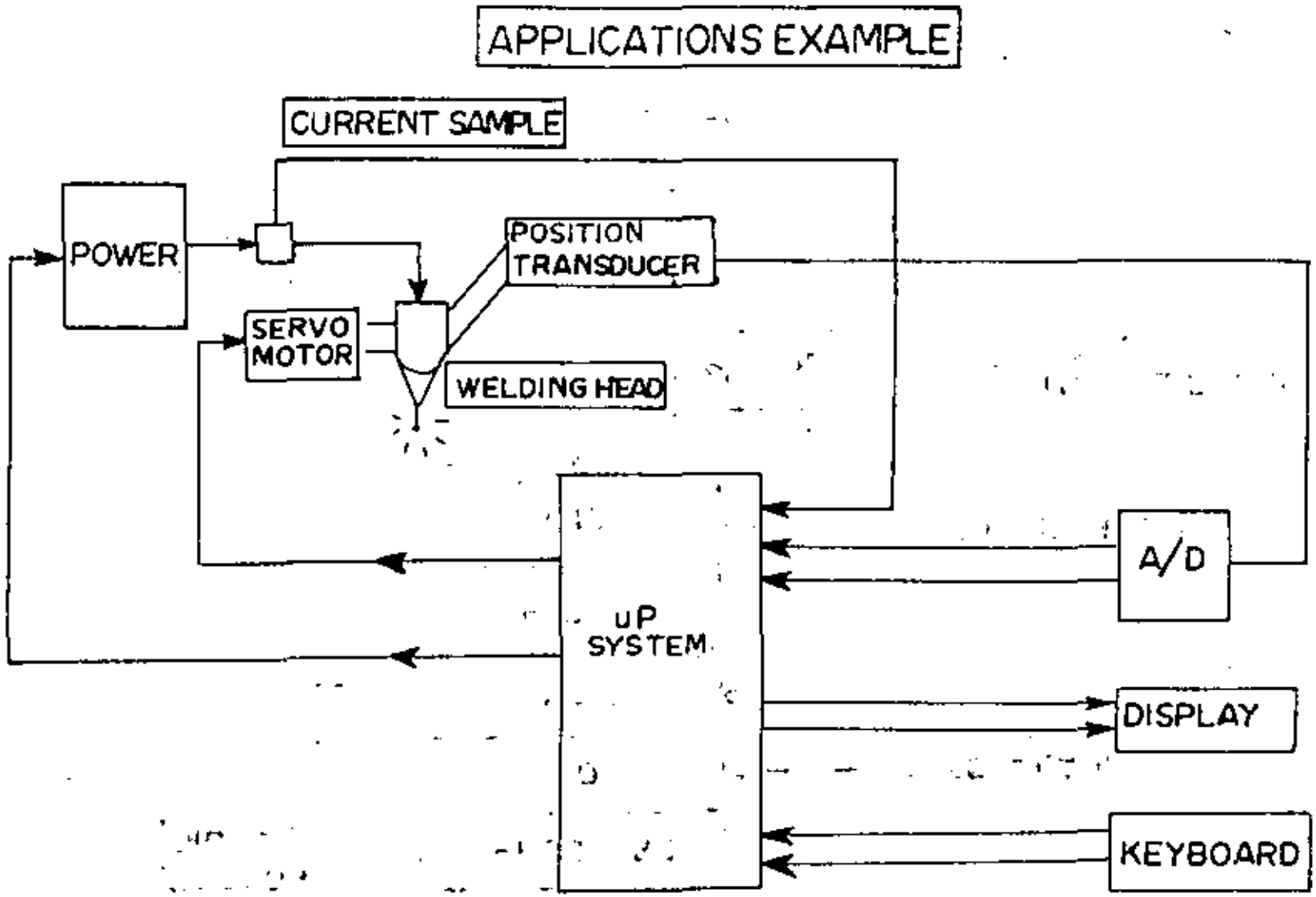

APPLICATIONS EXAMPLE
CURRENT SAMPLE
POWER
POSITION TRANSDUCER
SERVO MOTOR
WELDING HEAD
A/D
uP SYSTEM
DISPLAY
KEYBOARD

Fig. 5-6

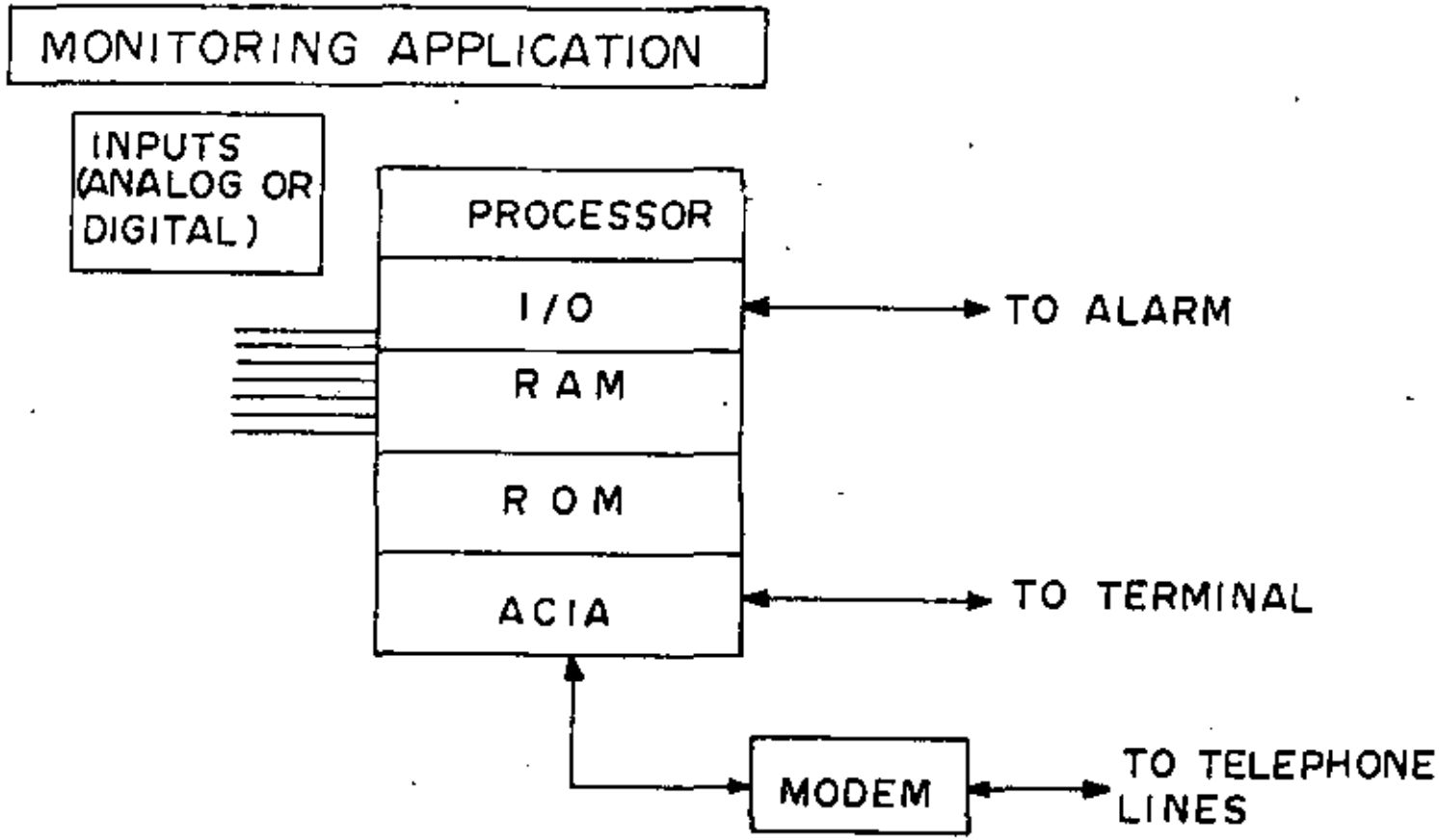

MONITORING APPLICATION
INPUTS (ANALOG OR DIGITAL)
PROCESSOR
I/O
RAM
ROM
ACIA
TO ALARM
TO TERMINAL
MODEM
TO TELEPHONE LINES

Fig. 5-7

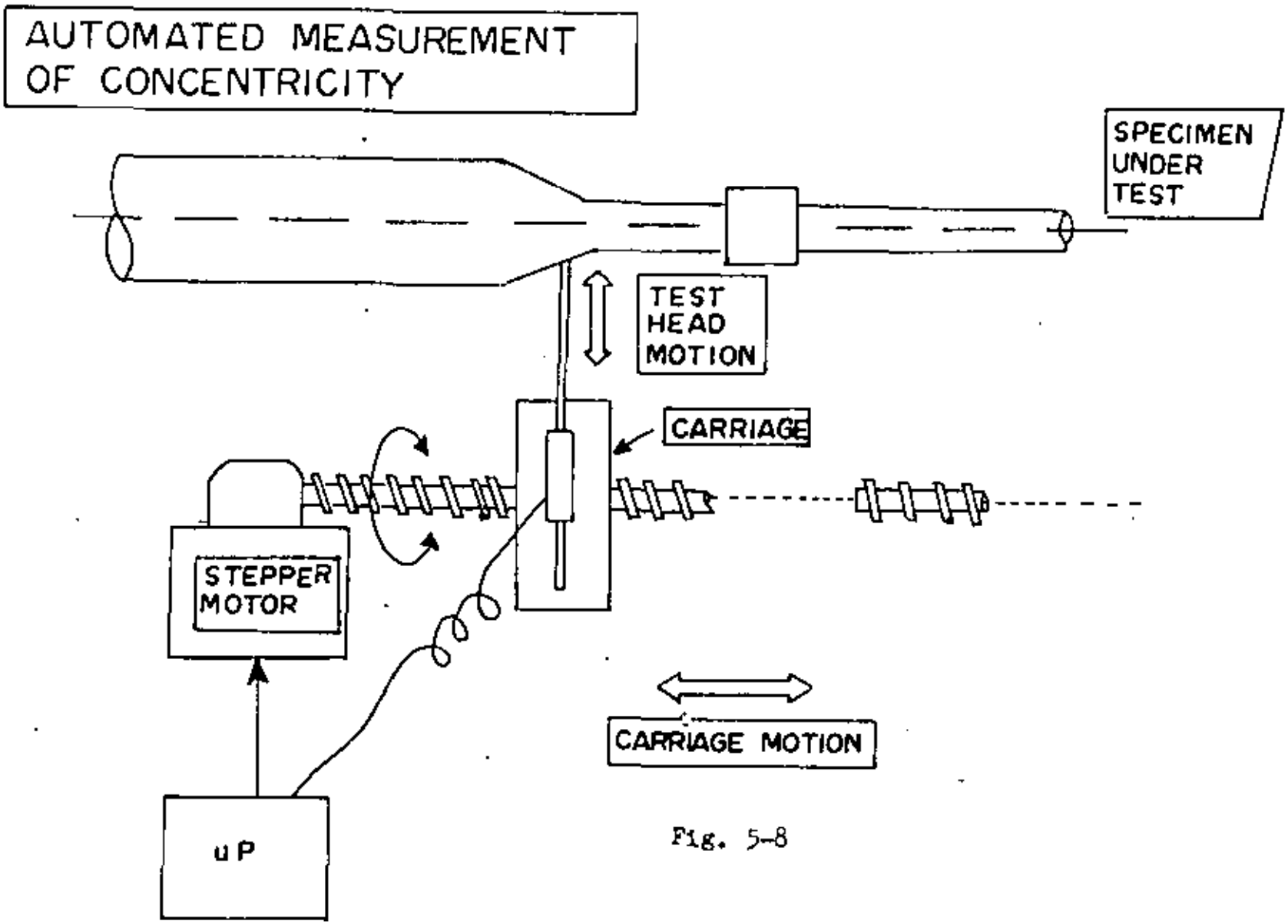

AUTOMATED MEASUREMENT
OF CONCENTRICITY
SPECIMEN
UNDER
TEST
TEST
HEAD
MOTION
CARRIAGE
STEPPER
MOTOR
CARRIAGE MOTION
uP

Fig. 5-8

would help with the acquisition and operator I/O would help minimize the chip numbers.

In the end, the software, or the programming of the processor would be the costliest part of the design - for any processor. These troubles are discussed in the next chapter. But even with these costs, microprocessors usually represent less than 10% of the total cost of a system. Designers should not therefore put the cart before the horse - modify the overall system design to simplify the microprocessor design.

SOFTWARE AIDS AND DEVELOPMENT SYSTEMS

CHAPTER 6

Many software aids for microprocessors resembles those available for minicomputers. Manufacturers of micros and others have written assemblers, cross assemblers, simulators, and compilers. These are available at additional costs to the purchaser of the processors. These aids are generally not used on typical microprocessor systems. Instead, the software is assembled and often tested on a larger computer system or on a special microprocessor development system. Logic analyzers are also available for microprocessors that help to solve hardware bugs. All of these aids are generally expensive compared to the final, individual microprocessor systems. These high costs are justified because software costs are very high otherwise, and because the costs are amortized over many systems and many designs.

In this chapter, we will examine various microprocessor systems that help with the development of microprocessor software and discuss the various software aids associated with each system.

Evaluation Systems

Microprocessor manufacturers have developed inexpensive evaluation kits or systems for users who wish to explore simple microprocessor operation. The Motorola Evaluation Kit II used in this tutorial is a good example. These systems interface with a terminal or with a keyboard on the system and allow the user to deposit, run, debug, and perhaps store his program on a cassette. The system is controlled by a program, the monitor, stored in a ROM. These systems can be expanded to include more memory and more I/O, however, it is impractical in most cases to add assemblers, disks, etc. These systems cost from $100 to $300, and are often available only as a kit. Other popular evaluation kits include the SDK-80 from Intel, and the KIM-1 from MOS Technology. While I have heard of a firm that used the SDK-80 in a product, I believe that evaulation kits are really suitable for learning about microprocessor and especially in educational programs.

Hobby Systems

This text is not directed toward the design of hobby systems such as the Altair by MITS or IMSAI's 8080 system. They are mentioned here because their costs are modest, and because these systems are occasionally used for inexpensive development of software. In contrast to the evaluation systems, the hobby systems can be easily expanded. Many have assemblers and debugging aids that greatly ease the programming process. Most of the systems can be interfaced to floppy disks where the software can be stored, edited, assembled, etc. High level languages have also been developed for these systems; BASIC in some form is the most popular.

Most of the use of the hObby systems are most suitable as a "personal computer": for computing by those who do not have easy access to larger systems. The costs of these systems varies from $600 to $3000 depending upon the equipment added to the basic processor.

Development Systems

The advent of microprocessor systems led to a demand for a new design tool, a tool that would greatly ease the testing of programs and testing of the user's hardware systems. One answer to the demand has been development systems. These systems contain system programs such as an editor, assembler, compiler, and debugging aids. Provisions for the testing of prototype systems and logic analyzers have also been added. Intel, Motorola, and others have produced development systems for software development and debugging of hardware systems. These systems cost from $5000 to $20,000 depending on the options. In Intel's "INTELLEC Microcomputer Development System" (MDS), software for the 8080, 8048, 8085, and 3001 slice systems can be developed. The MDS provides a basic bus and interfacing to disks and holds the appropriate modules for each processor. Intel has developed an "in circuit emulator" in which connection is made to the user's system through the processor's socket, and the actual processor is emulated using the software in the MDS. This system allows the testing of software in the user's sytems and the checking of hardware operation. Recently, Intel has introduced a resident compiler, PL/M, which can simplifiy the programming for a particular application, and which can be evaluated and developed on the MDS system. Compilers are discussed below.

Motorola has produced an "EXORciser" that performs many of the same

operations as the MDS system. As in the Intel system, it accomodates a number of different systems, and can be expanded to additional memory, ROM prorrammers, disk interfaces etc. Both the Intel and the Motorola systems deal only with their respective processors. Recently, Tektronix has developed its system, the 8002 system, which can accomodate a number of different processors: 6800, 8080, Z-80, and will be expanded to include four others. This system will be available at the tutorial and will be discussed below.

The 8002 system consists of a dual-floppy disk, a terminal, and the 8002 mainframe. The first two are standard components and need no elaboration. The mainframe contains the system processor, a Z-80, 16K bytes of system memory, an assembler processor, and an emulator processor. 16K bytes of RAM are available for the users program and the RAm may be expanded to 64K bytes. The assembler processor run the Tektronix assembler while the system processor handles all of the I/O to the disks. The emulator processor is a circuit board which uses the particular commercail processor of interest to the user. It runs the user-program while the system processor program detects program errors and runtime errors.

As shown in 6-1, one can start with the software design and produce a source program which in turn may be edited by the 8002 editing program. The source program may be assembled via the assembler processor and tested using the emulator processor. In that testing, the emulator processor executes the user program from the emulator memory and also provides the I/O.

In the hardware design process, one may start with a breadboard or a prototype system. Using a special prototype probe, the hardware system may be checked using the emulator processor. The processor is removed from the

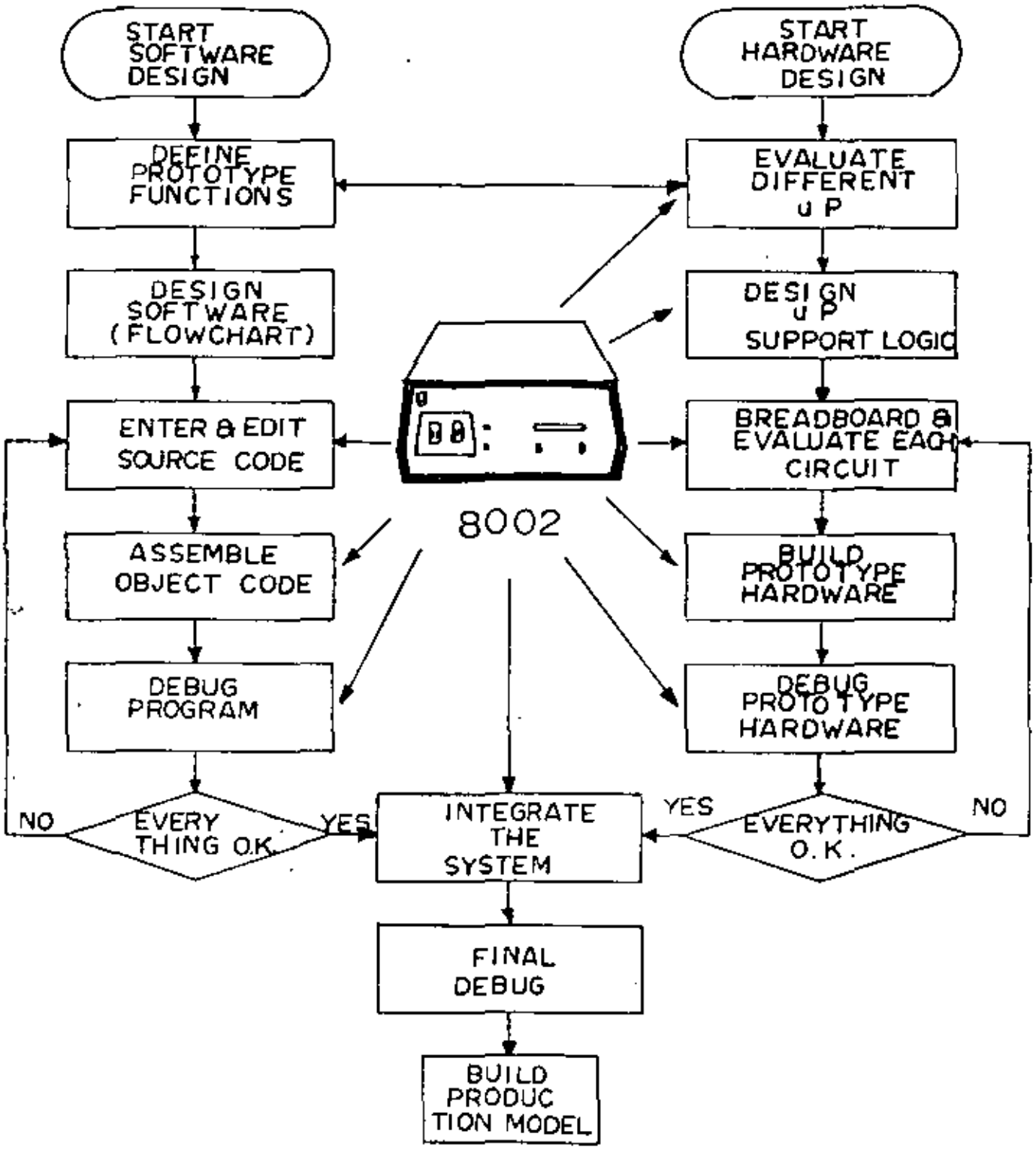

START SOFTWARE DESIGN
START HARDWARE DESIGN
DEFINE PROTOTYPE FUNCTIONS
EVALUATE DIFFERENT u P
DESIGN SOFTWARE (FLOWCHART)
DESIGN u P SUPPORT LOGIC
ENTER & EDIT SOURCE CODE
BREADBOARD & EVALUATE EACH CIRCUIT
8002
ASSEMBLE OBJECT CODE
BUILD PROTOTYPE HARDWARE
DEBUG PROGRAM
DEBUG PROTOTYPE HARDWARE
NO
EVERY THING O.K.
YES
INTEGRATE THE SYSTEM
YES
EVERYTHING O.K.
NO
FINAL DEBUG
BUILD PRODUC TION MODEL

Fig. 6-1

breadboard or prototype system and the prototype probe is inserted in its place. In this mode, the emulator processor runs the hardware while the system processor performs debugging and run time checks. Breakpoints may be inserted, new instruction added or substituted, and timing check in loops or from one part of the user program to another. The user program may be run from the emulator processor, from the prototype system, or both. A special mapping command exists to select which part of the memory and I/O space resides in the emulator memory and which part in the prototype board. A ROM programmer is also available for programming the hardware system's memory.

Finally, a logic analyzer facility is available for system testing. Software debugging is successful only when the hardware is working properly. To check hardware, the analyzer displays the states of the address bus, the data bus, the control bus, and eight other user selected points. The last 100 such states are stored and displayed as a result of a pre-trigger, a post-trigger, or a variable center trigger.

This development system was described in some detail because it offers most of the design aids available in a single system. The design aids are not necessarily unique however; Intel offers an ICE system which is very similar to the emulator processor and the prototype probe. The high initial cost for this system -about $18,000 - is justified if designs using several different processors are necessary. The same system instructions and usage prevails over all of the processors, and is easier than using development systems from different manufacturers.

Time Sharing Systems

Time sharing systems are available to nearly everyone, although the connect time costs may be very high in a few cases. Many organizations have their own computer systems and offer this service at little or no costs. Microprocessor manufacturers have offered compilers, cross-assemblers, and simulators/emulators for time sharing services. Normally, the assemblers are written in Fortran, and the compilers are written in PL/1 (Intel's PL/M, Motorola's MPL). The cost of such programs is about $1000. These programs may be placed on the host computers and source programs may be assembled or compiled and partially debugged. This approach is, of course, considerably cheaper in terms of initial costs, but if extensive software assembly and simulation is done on a time sharing system, the costs may exceed that of a development system.

The assemblers are fairly standard. Some people have trouble with assembly language programming, and many problems are difficult to write in terms of assembly language instructions. Thus, there has been considerable interest in compilers which allow the programmer to write in a high level language. Intel was the first to offer a compiler and chose PL/1 as the basic language, calling their compilers the PL/M compiler. Later, Motorola also wrote a compiler in PL/1 and called it MPL. Especially at first, compilers were relatively controversial. They were accused of generating inefficient code thus requiring greater amounts of memory for the final microprocessor program. Also it is necessary to write the I/O parts in assembly language anyway. The compilers have been improved and indeed offer a number of advantages. First, compilers save a great deal of programming time; claims of one-fifth the time compared to assembly language programming have been made. Second, compilers make good sense if frequent reprogramming is necessary or if one wants to quickly obtain prototype software. Third, compilers are useful in low volume systems where it is to

costly to optimize in assembly language. Normally, the speed with which a program will execute depends upon a very few portions of the program: a few loops perhaps. If the compiler can be linked to an assembly language program, as most can, these few portions can be optimized and the rest of the program written using assembly language.

Intel offers their compiler as a resident program in their Intellec system (MDS) and claims that this offers considerable savings over timesharing services. Intel also states that the realiability of programs written in PL/M have better software reliability.

Macros and other aids

Programming still seems to be a barrier in the use of computer systems for many people. While high level languages help, they are not always the answer. Macros represent at least a partial solution. A macro is essentially a string of assembly language instructions which represents a single operation or task. An example might be a microprocessor controlled pin-ball machine[9] in which the designers (who are not computer experts) wish to program a variety of games. The manufacturer of the processors (if the quantity is very promising) or some other organization might write a series of Macros, each of which performs some task: check the tilt, count the points, check for bonus points, etc. The designer of the game could then simple combine the macros and a few conditional branches to form a specific game. The macros can be named in obvious ways. Essentially, then, macros can be used by non-expert programmers to deal with programmable systems such as microprocessor based systems.

---

9. Mentioned at an IEEE Microprocessor Workshop by a National Semiconductor speaker

The use of macros to simplify programming is similar to "modular software" [10] where each module is a short, debugged, program that accomplishes one task. Software modules can be compared to standard hardware modules that are used again and again in design of microprocessor systems. One needs, of course, a software design system that will support the modules or macros.

Microprocessors will soon play an important role in "personal computing" that far exceeds its present hobby status. Simplified programming languages beyond Macros will have to be developed to satisfy this market. Assembly language and probably BASIC are out of the question. Of course the languages will have to be closely related to the system that the processor controls. This text was prepared using only a few system calls - the rest: the editing, inserting, appending, even checking the spelling was accomplished with a few system calls. Such a system is used by a large number of people, some of whom know very little about programming.

---

10. Intel article reprint AR-30 from Electronics, Sept. 30, 1976.

APPENDIX A THE 8080 MICROPROCESSOR

The 8080 is best appreciated and understood if it is remembered that it is the first of the "2nd generation" processors. Its predessesors were the 4004 and the 8008, quite limited processors by comparison. The programming model of the 8080 is shown in Fig. A-1. The program counter, the stack pointer, and the accumulator are quite standard registers. The condition code register has the usual bits plus a parity bit that is set when the modulo 2 sum of the bits in the result of the operation is zero. I have never used this feature, and thus I cannot give an example of the application of the parity bit. The register pair, H & L, is frequently used to point to a memory location, but it is not an index register. Registers D & E and B & C, as pairs, also can point to memory and used in some instructions. The registers may also be used for scratch pad purposes.

The 8080 has a number of addressing modes: absolute (exact address), register indirect (location pointed to by register pair), register (op-code specifies one of the internal registers or register pairs), and immediate(operand follows op-code). All conditional branches, called conditional jumps in the 8080, use absolute addressing. The register pair H L is more frequently used for memory-accumulator transfers or ALU operations. The condition codes are not modified by incrementing or decrementing register pairs and there is no operation to compare H L to determine if a transfer a data is complete. This is the principal shortcoming of the instruction set.

The 8080's clock cycles are more complex than most other processors'. Each

instruction cycle consists of a number of machine cycles. Each machine cycle consists of a number of clock cycles. The number of machine cycles in an instruction cycle is equal to the number of memory references in the instruction. For example, if the instruction is a one byte instruction such as increment the accumulator, one machine cycle is needed to fetch the instruction. If we are going to add a number to the accumulator using the immediate mode, two cycles are needed, one to fetch the op-code, and one to fetch the operand. The first machine cycle is either four or five clock cycles and the rest are three clock cycles. Shown in Fig. A-2 is a typical machine cycle (the first in an instruction). In T1, the address of the op-code is placed on the address bus; in T2, status information is placed on the data bus; in T3, the op-code is supposed to be available from memory; and in T4, the opcode is decoded. If the instruction is only a single byte instruction, a T5 period will follow for execution. If more than one byte is required for the instruction, T4 will be followed by another series of T1, T2, and T3 to fetch the next byte. Thus each cycle consists of a single series of T1,T2,T3,T4 (possibly T5) and a series of T1,T2,T3 for each additional access to memory.

The status information supplied by the processor during each T2 is transitory and must be latched. It consists of the memory read and write or input or output control signals. It must further be decoded to be useful. Intel now supplies the 8228 which automatically latches and decodes the status information and also supplies the interrupt capabilities discussed earlier in the main text. The 8080 is probably not now used without the 8228 in most systems.

The high level clocks (greater than 5 volts) required for the 8080 are also hard to generated for the following reasons. The phase 1 and phase 2 signals are not symmetrical if the processor is to operated at maximum speeds. Further there are synchronization problems with the READY signal. Intel has provided an 8224 clock driver for this purpose that solves all of the problems. Thus the 8080 cpu could be considered a three-chip processor (in fairness the clock generation of the 6800 is nontrivial).

The complete processor is shown in Fig. A-3. The 8085 offers considerable ease in design, since it does not need any special clock or status latches. It is almost certainly the processor of choice for most future systems if one wishes to stay in the 8080/8085 software (a considerable investment).

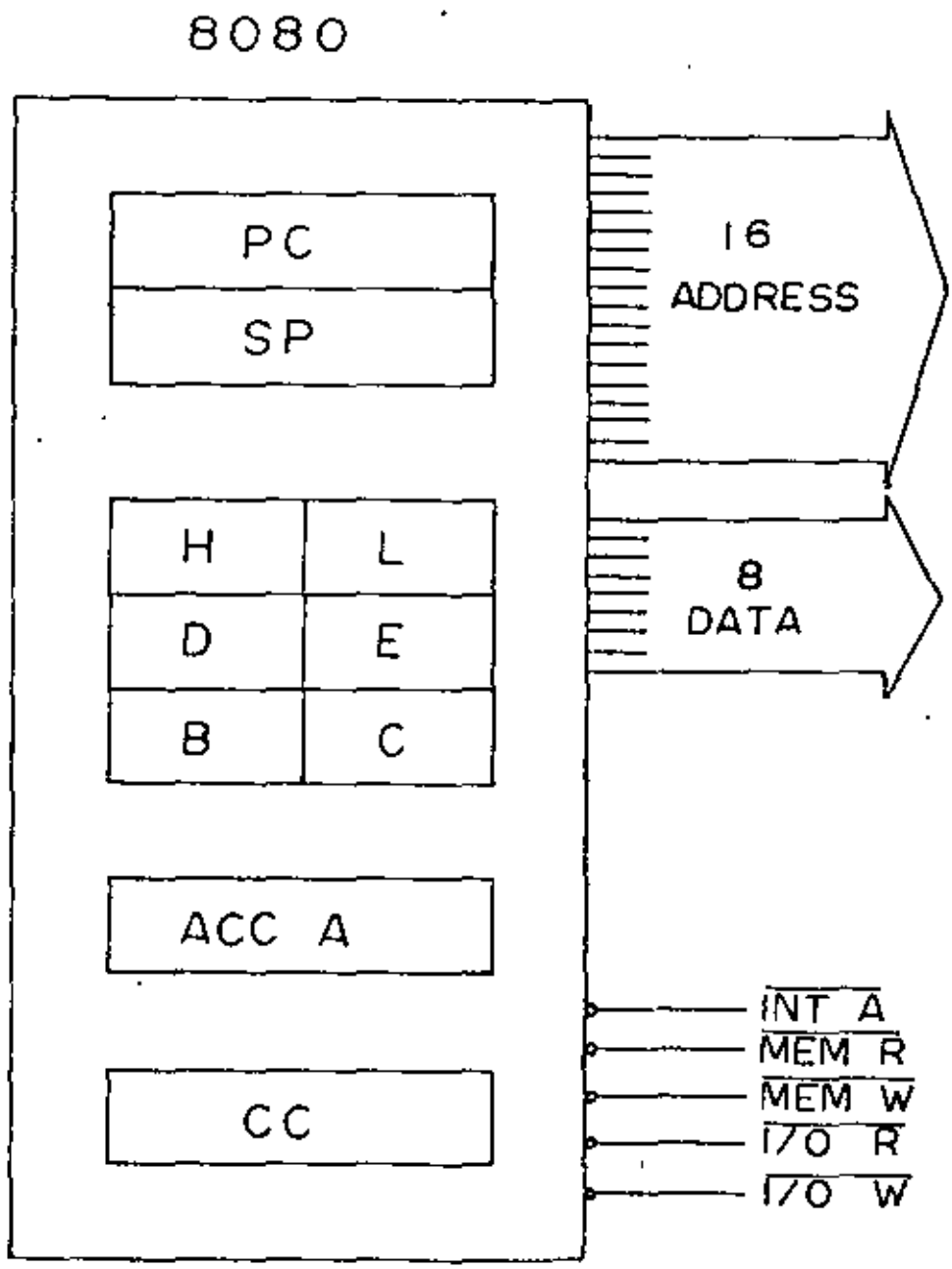
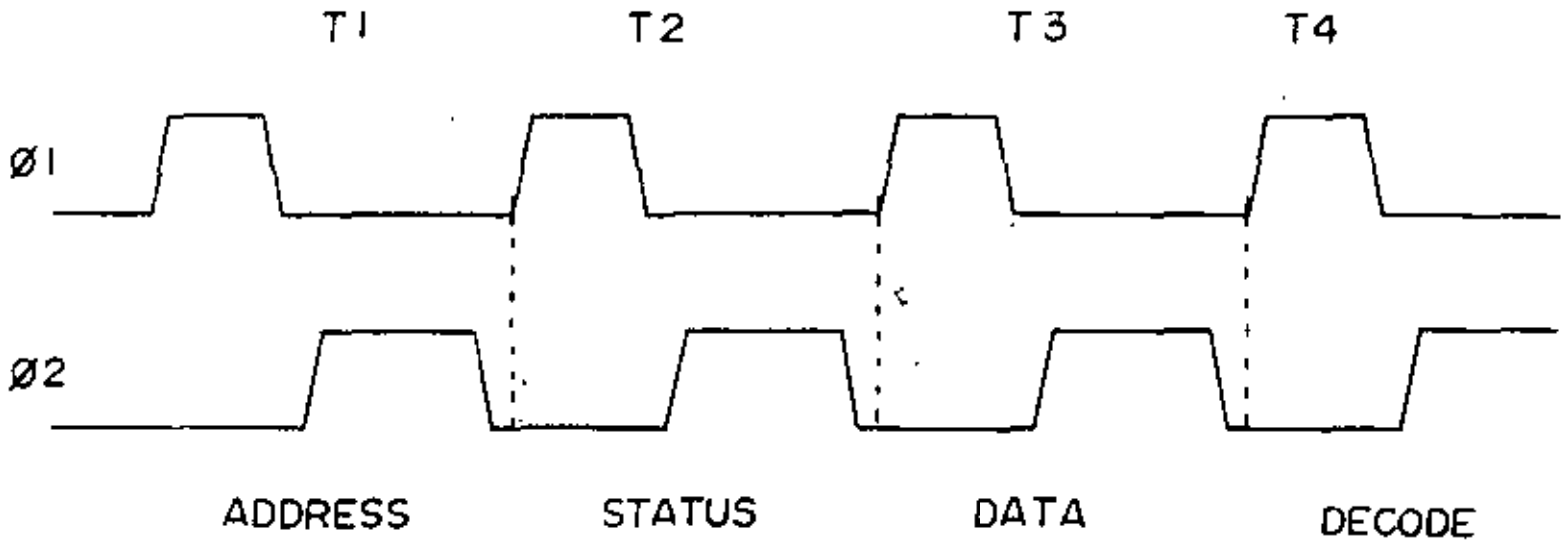

8080
PC
SP
H
L
D
E
B
C
ACC A
CC
16
ADDRESS
8
DATA
INT A
MEM R
MEM W
I/O R
I/O W

Fig. A-1



CLOCK CYCLES FOR THE 8080

Fig. A-2

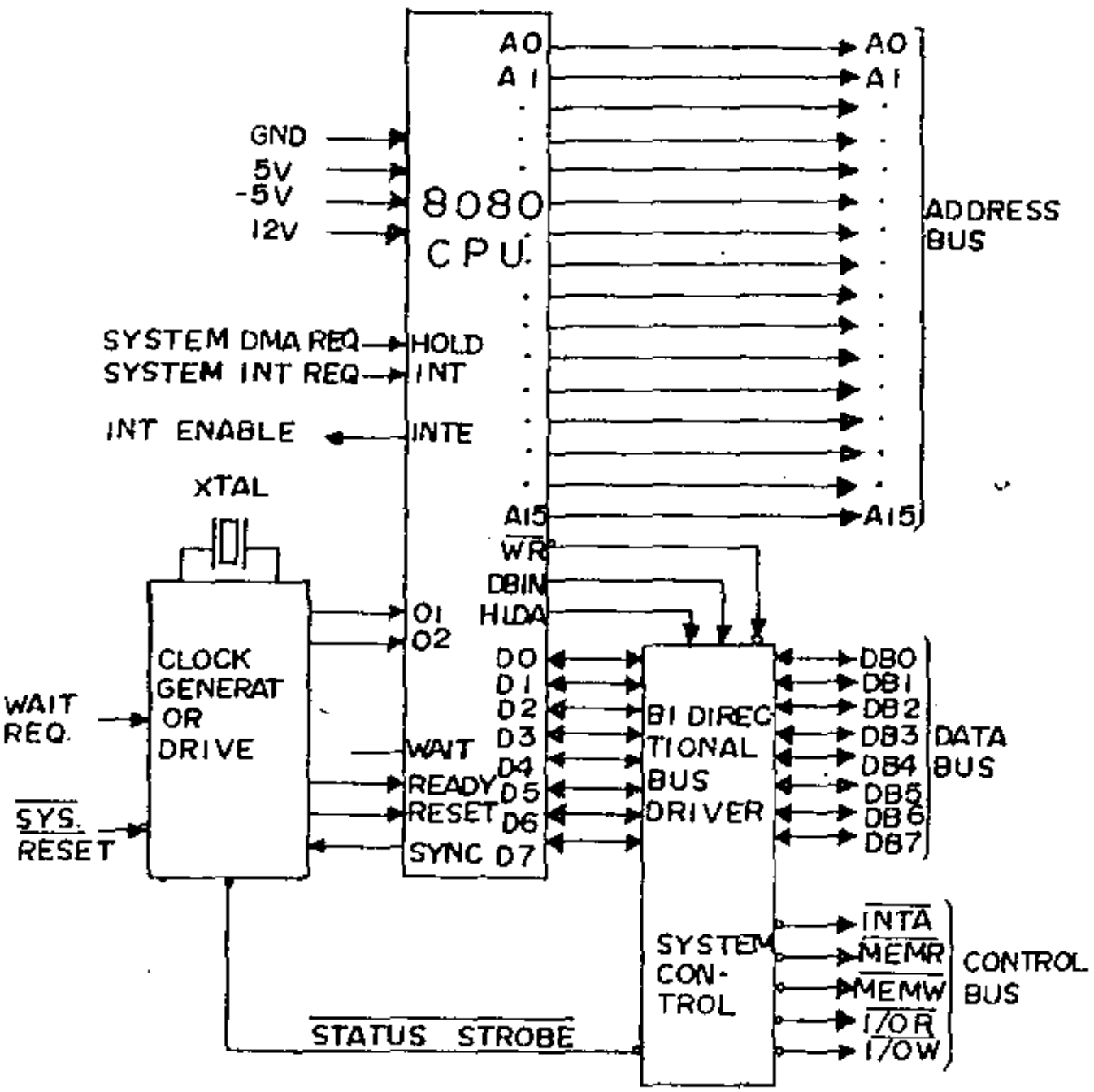

A0
A1
A15
GND
5V
-5V
12V
8080
CPU
ADDRESS
BUS
SYSTEM DMA REQ
SYSTEM INT REQ
HOLD
INT
INT ENABLE
INTE
XTAL
WR
DBIN
HLDA
O1
O2
CLOCK
GENERAT
OR
DRIVE
WAIT
REQ.
SYS.
RESET
WAIT
READY
RESET
SYNC
D0
D1
D2
D3
D4
D5
D6
D7
BI DIREC
TIONAL
BUS
DRIVER
DB0
DB1
DB2
DB3
DB4
DB5
DB6
DB7
DATA
BUS
STATUS STROBE
SYSTEM
CON-
TROL
INTA
MEMR
MEMW
I/OR
I/OW
CONTROL
BUS

Fig. A-3

centro de educación continua
división de estudios superiores
facultad de ingeniería, unam

INTRODUCCION A LOS MICROPROCESADORES Y SUS APLICACIONES

MANUAL DEL PROCESADOR 280

AGOSTO, 1979

MANUEL ESTEVEZ 1

Z80®-CPU
Z80A®-CPU

Technical Manual

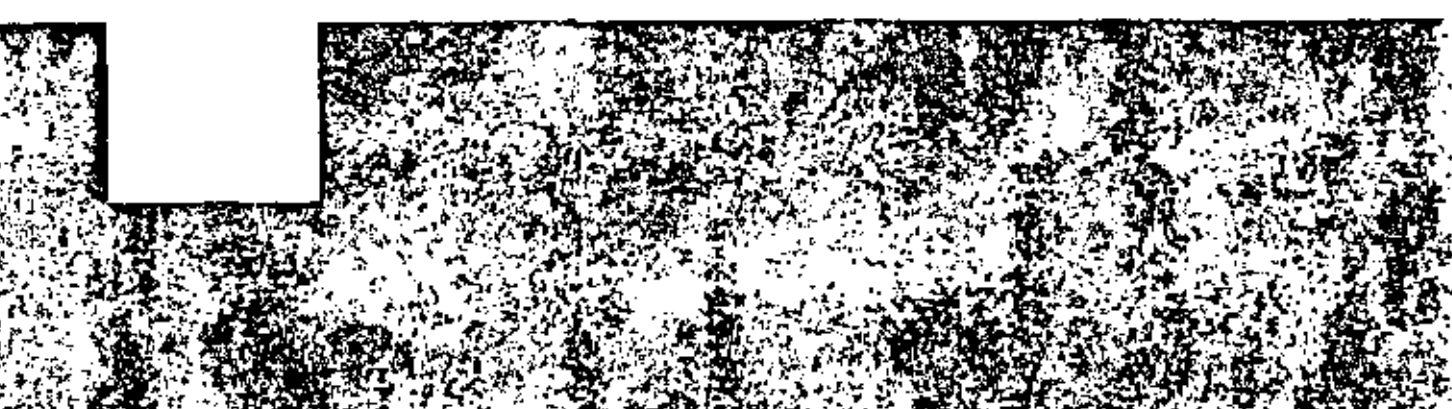


TABLE OF CONTENTS

1.0 INTRODUCTION

The term "microcomputer" has been used to describe virtually every type of small computing device designed within the last few years. This term has been applied to everything from simple "microprogrammed" controllers constructed out of TTL MSI up to low end minicomputers with a portion of the CPU constructed out of TTL LSI "bit slices." However, the major impact of the LSI technology within the last few years has been with MOS LSI. With this technology, it is possible to fabricate complete and very powerful computer systems with only a few MOS LSI components.

The Zilog Z-80 family of components is a significant advancement in the state-of-the art of microcomputers. These components can be configured with any type of standard semiconductor memory to generate computer systems with an extremely wide range of capabilities. For example, as few as two LSI circuits and three standard TTL MSI packages can be combined to form a simple controller. With additional memory and I/O devices a computer can be constructed with capabilities that only a minicomputer could previously deliver. This wide range of computational power allows standard modules to be constructed by a user that can satisfy the requirements of an extremely wide range of applications.

The major reason for MOS LSI domination of the microcomputer market is the low cost of these few LSI components. For example, MOS LSI microcomputers have already replaced TTL logic in such applications as terminal controllers, peripheral device controllers, traffic signal controllers, point of sale terminals, intelligent terminals and test systems. In fact the MOS LSI microcomputer is finding its way into almost every product that now uses electronics and it is even replacing many mechanical systems such as weight scales and automobile controls.

The MOS LSI microcomputer market is already well established and new products using them are being developed at an extraordinary rate. The Zilog Z-80 component set has been designed to fit into this market through the following factors:

1. The Z-80 is fully software compatible with the popular 8080A CPU offered from several sources. Existing designs can be easily converted to include the Z-80 as a superior alternative.
2. The Z-80 component set is superior in both software and hardware capabilities to any other microcomputer system on the market. These capabilities provide the user with significantly lower hardware and software development costs while also allowing him to offer additional features in his system.
3. For increased throughput the Z80A operating at a 4 MHZ clock rate offers the user significant speed advantages over competitive products.
4. A complete product line including full software support with strong emphasis on high level languages and a disk-based development system with advanced real-time debug capabilities is offered to enable the user to easily develop new products.

Microcomputer systems are extremely simple to construct using Z-80 components. Any such system consists of three parts:

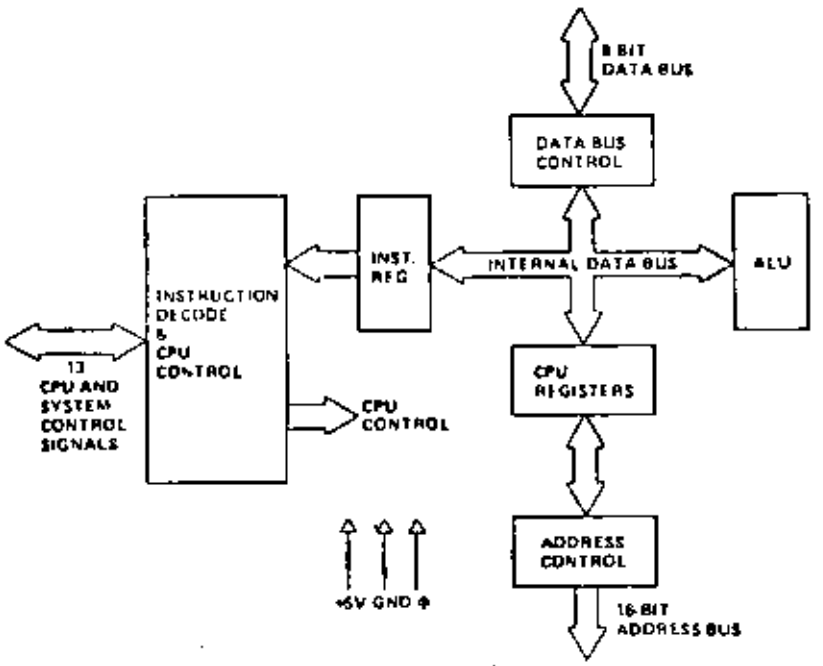
1. CPU (Central Processing Unit)
2. Memory
3. Interface Circuits to peripheral devices

The CPU is the heart of the system. Its function is to obtain instructions from the memory and perform the desired operations. The memory is used to contain instructions and in most cases data that is to be processed. For example, a typical instruction sequence may be to read data from a specific peripheral device, store it in a location in memory, check the parity and write it out to another peripheral device. Note that the Zilog component set includes the CPU and various general purpose I/O device controllers, while a wide range of memory devices may be used from any source. Thus, all required components can be connected together in a very simple manner with virtually no other external logic. The user's effort then becomes primarily one of software development. That is, the user can concentrate on describing his problem and translating it into a series of instructions that can be loaded into the microcomputer memory. Zilog is dedicated to making this step of software generation as simple as possible. A good example of this is our

assembly language in which a simple mnemonic is used to represent every instruction that the CPU can perform. This language is self documenting in such a way that from the mnemonic the user can understand exactly what the instruction is doing without constantly checking back to a complex cross listing.

2.0 Z-80 CPU ARCHITECTURE

A block diagram of the internal architecture of the Z-80 CPU is shown in figure 2.0-1. The diagram shows all of the major elements in the CPU and it should be referred to throughout the following description.



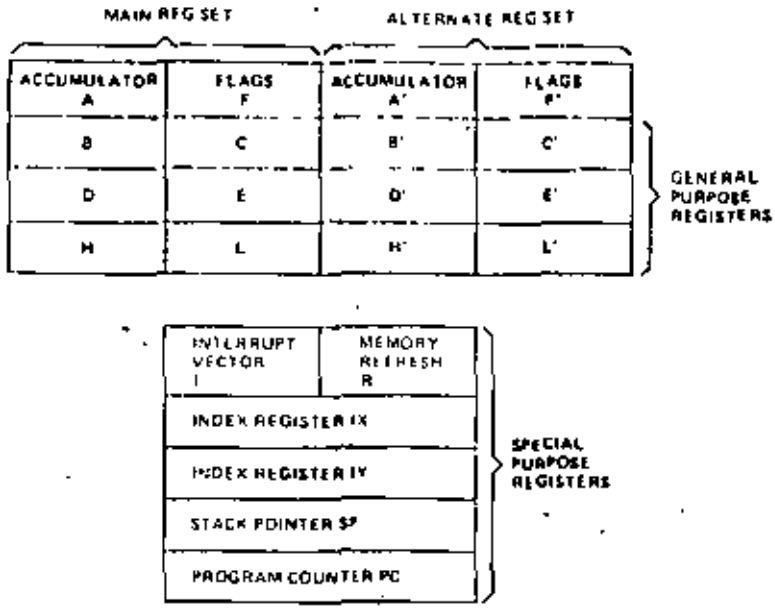
Z-80 CPU BLOCK DIAGRAM
FIGURE 2.0-1

2.1 CPU REGISTERS

The Z-80 CPU contains 208 bits of R/W memory that are accessible to the programmer. Figure 2.0-2 illustrates how this memory is configured into eighteen 8-bit registers and four 16-bit registers. All Z-80 registers are implemented using static RAM. The registers include two sets of six general purpose registers that may be used individually as 8-bit registers or in pairs as 16-bit registers. There are also two sets of accumulator and flag registers.

Special Purpose Registers

1. Program Counter (PC). The program counter holds the 16-bit address of the current instruction being fetched from memory. The PC is automatically incremented after its contents have been transferred to the address lines. When a program jump occurs the new value is automatically placed in the PC, overriding the incrementer.
2. Stack Pointer (SP). The stack pointer holds the 16-bit address of the current top of a stack located anywhere in external system RAM memory. The external stack memory is organized as a last-in first-out (LIFO) file. Data can be pushed onto the stack from specific CPU registers or popped off of the stack into specific CPU registers through the execution of PUSH and POP instructions. The data popped from the stack is always the last data pushed onto it. The stack allows simple implementation of multiple level interrupts, unlimited subroutine nesting and simplification of many types of data manipulation.



Z-80 CPU REGISTER CONFIGURATION
FIGURE 2.0-2

3. **Two Index Registers (IX & IY).** The two independent index registers hold a 16-bit base address that is used in indexed addressing modes. In this mode, an index register is used as a base to point to a region in memory from which data is to be stored or retrieved. An additional byte is included in indexed instructions to specify a displacement from this base. This displacement is specified as a two's complement signed integer. This mode of addressing greatly simplifies many types of programs, especially where tables of data are used.

4. **Interrupt Page Address Register (I).** The Z-80 CPU can be operated in a mode where an indirect call to any memory location can be achieved in response to an interrupt. The I Register is used for this purpose to store the high order 8-bits of the indirect address while the interrupting device provides the lower 8-bits of the address. This feature allows interrupt routines to be dynamically located anywhere in memory with absolute minimal access time to the routine.

5. **Memory Refresh Register (R).** The Z-80 CPU contains a memory refresh counter to enable dynamic memories to be used with the same ease as static memories. Seven bits of this 8 bit register are automatically incremented after each instruction fetch. The eighth bit will remain as programmed as the result of an LD R, A instruction. The data in the refresh counter is sent out on the lower portion of the address bus along with a refresh control signal while the CPU is decoding and executing the fetched instruction. This mode of refresh is totally transparent to the programmer and does not slow down the CPU operation. The programmer can load the R register for testing purposes, but this register is normally not used by the programmer. During refresh, the contents of the I register are placed on the upper 8 bits of the address bus.

Accumulator and Flag Registers

The CPU includes two independent 8-bit accumulators and associated 8-bit flag registers. The accumulator holds the results of 8-bit arithmetic or logical operations while the flag register indicates specific conditions for 8 or 16-bit operations, such as indicating whether or not the result of an operation is equal to zero. The programmer selects the accumulator and flag pair that he wishes to work with with a single exchange instruction so that he may easily work with either pair.

General Purpose Registers

There are two matched sets of general purpose registers, each set containing six 8-bit registers that may be used individually as 8-bit registers or as 16-bit register pairs by the programmer. One set is called BC, DE and HL while the complementary set is called BC', DE' and HL'. At any one time the programmer can select either set of registers to work with through a single exchange command for the entire set. In systems where fast interrupt response is required, one set of general purpose registers and an accumulator/flag register may be reserved for handling this very fast routine. Only a simple exchange commands need be executed to go between the routines. This greatly reduces interrupt service time by eliminating the requirement for saving and retrieving register contents in the external stack during interrupt or subroutine processing. These general purpose registers are used for a wide range of applications by the programmer. They also simplify programming, especially in ROM based systems where little external read/write memory is available.

2.2 ARITHMETIC & LOGIC UNIT (ALU)

The 8-bit arithmetic and logical instructions of the CPU are executed in the ALU. Internally the ALU communicates with the registers and the external data bus on the internal data bus. The type of functions performed by the ALU include:

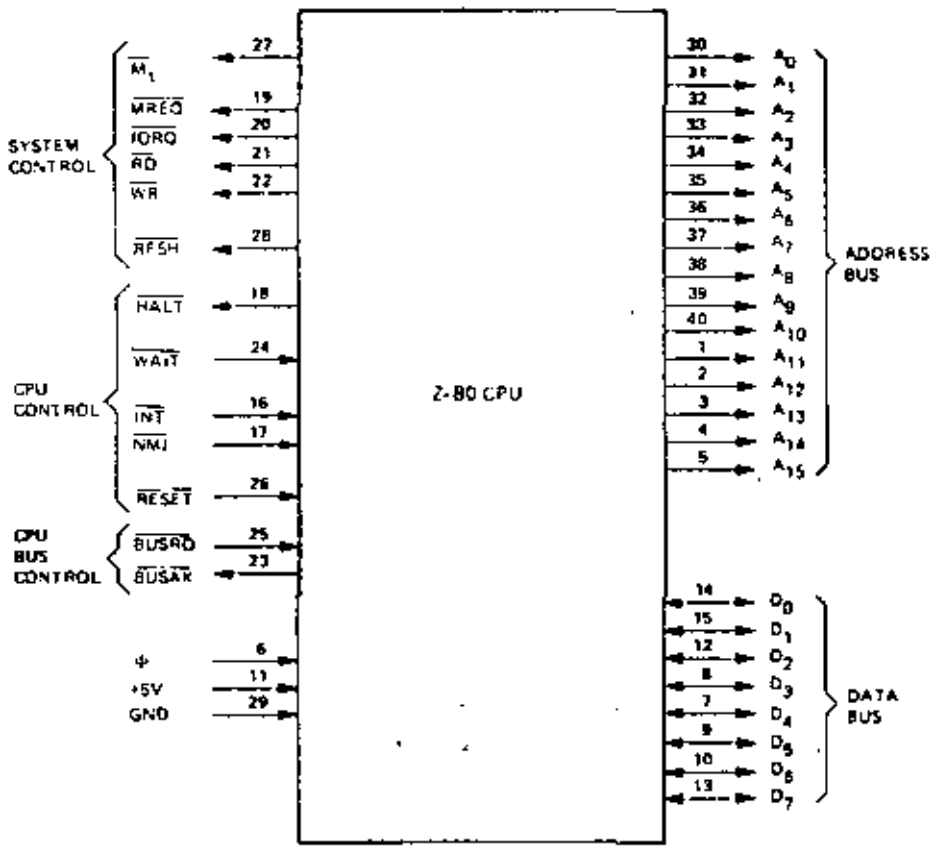
Add	Left or right shifts or rotates (arithmetic and logical)
Subtract	Increment
Logical AND	Decrement
Logical OR	Set bit
Logical Exclusive OR	Reset bit
Compare	Test bit

2.3 INSTRUCTION REGISTER AND CPU CONTROL

As each instruction is fetched from memory, it is placed in the instruction register and decoded. The control sections performs this function and then generates and supplies all of the control signals necessary to read or write data from or to the registers, control the ALU and provide all required external control signals.

3.0 Z-80 CPU PIN DESCRIPTION

The Z-80 CPU is packaged in an industry standard 40 pin Dual In-Line Package. The I/O pins are shown in figure 3.0-1 and the function of each is described below.



Z-80 PIN CONFIGURATION
FIGURE 3.0-1

A_0-A_{15} (Address Bus)

Tri-state output, active high. A_0-A_{15} constitute a 16-bit address bus. The address bus provides the address for memory (up to 64K bytes) data exchanges and for I/O device data exchanges. I/O addressing uses the 8 lower address bits to allow the user to directly select up to 256 input or 256 output ports. A_0 is the least significant address bit. During refresh time, the lower 7 bits contain a valid refresh address.

D_0-D_7 (Data Bus)

Tri-state input/output, active high. D_0-D_7 constitute an 8-bit bidirectional data bus. The data bus is used for data exchanges with memory and I/O devices.

$\overline{M_1}$ (Machine Cycle one)

Output, active low. $\overline{M_1}$ indicates that the current machine cycle is the OP code fetch cycle of an instruction execution. Note that during execution of 2-byte op-codes, $\overline{M1}$ is generated as each op code byte is fetched. These two byte op-codes always begin with CBH, DDH, EDH or FDH. $\overline{M1}$ also occurs with $\overline{IORQ}$ to indicate an interrupt acknowledge cycle.

$\overline{MREQ}$ (Memory Request)

Tri-state output, active low. The memory request signal indicates that the address bus holds a valid address for a memory read or memory write operation.

$\overline{IORQ}$ (Input/Output Request)	Tri-state output, active low. The $\overline{IORQ}$ signal indicates that the lower half of the address bus holds a valid I/O address for a I/O read or write operation. An $\overline{IORQ}$ signal is also generated with an $\overline{M1}$ signal when an interrupt is being acknowledged to indicate that an interrupt response vector can be placed on the data bus. Interrupt Acknowledge operations occur during M_1 time while I/O operations never occur during M_1 time.
$\overline{RD}$ (Memory Read)	Tri-state output, active low. $\overline{RD}$ indicates that the CPU wants to read data from memory or an I/O device. The addressed I/O device or memory should use this signal to gate data onto the CPU data bus.
$\overline{WR}$ (Memory Write)	Tri-state output, active low. $\overline{WR}$ indicates that the CPU data bus holds valid data to be stored in the addressed memory or I/O device.
$\overline{RFSH}$ (Refresh)	Output, active low. $\overline{RFSH}$ indicates that the lower 7 bits of the address bus contain a refresh address for dynamic memories and the current $\overline{MREQ}$ signal should be used to do a refresh read to all dynamic memories.
$\overline{HALT}$ (Halt state)	Output, active low. $\overline{HALT}$ indicates that the CPU has executed a HALT software instruction and is awaiting either a non maskable or a maskable interrupt (with the mask enabled) before operation can resume. While halted, the CPU executes NOP's to maintain memory refresh activity.
$\overline{WAIT}$ (Wait)	Input, active low. $\overline{WAIT}$ indicates to the Z-80 CPU that the addressed memory or I/O devices are not ready for a data transfer. The CPU continues to enter wait states for as long as this signal is active. This signal allows memory or I/O devices of any speed to be synchronized to the CPU.
$\overline{INT}$ (Interrupt Request)	Input, active low. The Interrupt Request signal is generated by I/O devices. A request will be honored at the end of the current instruction if the internal software controlled interrupt enable flip-flop (IFF) is enabled and if the $\overline{BUSRQ}$ signal is not active. When the CPU accepts the interrupt, an acknowledge signal ($\overline{IORQ}$ during M_1 time) is sent out at the beginning of the next instruction cycle. The CPU can respond to an interrupt in three different modes that are described in detail in section 5.4 (CPU Control Instructions).
$\overline{NMI}$ (Non Maskable Interrupt)	Input, negative edge triggered. The non maskable interrupt request line has a higher priority than $\overline{INT}$ and is always recognized at the end of the current instruction, independent of the status of the interrupt enable flip-flop. $\overline{NMI}$ automatically forces the Z-80 CPU to restart to location 0066_H. The program counter is automatically saved in the external stack so that the user can return to the program that was interrupted. Note that continuous WAIT cycles can prevent the current instruction from ending, and that a $\overline{BUSRQ}$ will override a $\overline{NMI}$.

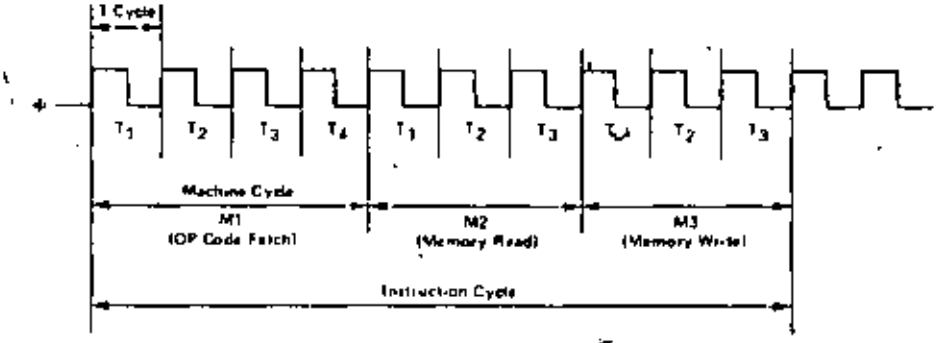
$\overline{\text{RESET}}$	Input, active low. $\overline{\text{RESET}}$ forces the program counter to zero and initializes the CPU. The CPU initialization includes: 1) Disable the interrupt enable flip-flop 2) Set Register I = 00_H 3) Set Register R = 00_H 4) Set Interrupt Mode 0 During reset time, the address bus and data bus go to a high impedance state and all control output signals go to the inactive state.
$\overline{\text{BUSRQ}}$ (Bus Request)	Input, active low. The bus request signal is used to request the CPU address bus, data bus and tri-state output control signals to go to a high impedance state so that other devices can control these buses. When $\overline{\text{BUSRQ}}$ is activated, the CPU will set these buses to a high impedance state as soon as the current CPU machine cycle is terminated.
$\overline{\text{BUSAK}}$ (Bus Acknowledge)	Output, active low. Bus acknowledge is used to indicate to the requesting device that the CPU address bus, data bus and tri-state control bus signals have been set to their high impedance state and the external device can now control these signals.
Φ	Single phase TTL level clock which requires only a 330 ohm pull-up resistor to +5 volts to meet all clock requirements.

4.0 CPU TIMING

The Z-80 CPU executes instructions by stepping through a very precise set of a few basic operations. These include:

Memory read or write
I/O device read or write
Interrupt acknowledge

All instructions are merely a series of these basic operations. Each of these basic operations can take from three to six clock periods to complete or they can be lengthened to synchronize the CPU to the speed of external devices. The basic clock periods are referred to as T cycles and the basic operations are referred to as M (for machine) cycles. Figure 4.0-0 illustrates how a typical instruction will be merely a series of specific M and T cycles. Notice that this instruction consists of three machine cycles (M1, M2 and M3). The first machine cycle of any instruction is a fetch cycle which is four, five or six T cycles long (unless lengthened by the wait signal which will be fully described in the next section). The fetch cycle (M1) is used to fetch the OP code of the next instruction to be executed. Subsequent machine cycles move data between the CPU and memory or I/O devices and they may have anywhere from three to five T cycles (again they may be lengthened by wait states to synchronize the external devices to the CPU). The following paragraphs describe the timing which occurs within any of the basic machine cycles. In section 7, the exact timing for each instruction is specified.



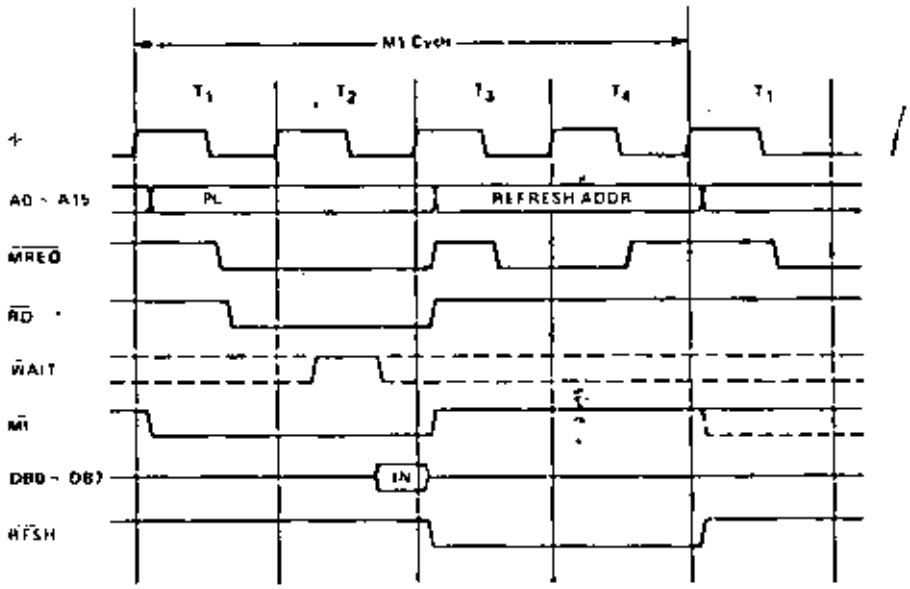
BASIC CPU TIMING EXAMPLE
FIGURE 4.0-0

All CPU timing can be broken down into a few very simple timing diagrams as shown in figure 4.0-1 through 4.0-7. These diagrams show the following basic operations with and without wait states (wait states are added to synchronize the CPU to slow memory or I/O devices).

4.0-1. Instruction OP code fetch (M1 cycle)
4.0-2. Memory data read or write cycles
4.0-3. I/O read or write cycles
4.0-4. Bus Request/Acknowledge Cycle
4.0-5. Interrupt Request/Acknowledge Cycle
4.0-6. Non maskable Interrupt Request/Acknowledge Cycle
4.0-7. Exit from a HALT instruction

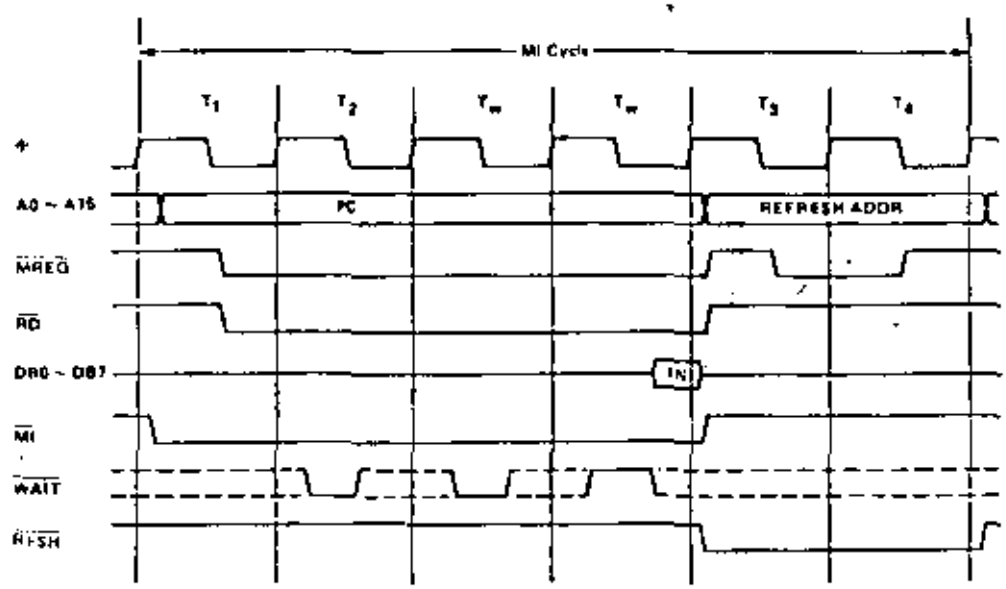
INSTRUCTION FETCH

Figure 4.0-1 shows the timing during an M1 cycle (OP code fetch). Notice that the PC is placed on the address bus at the beginning of the M1 cycle. One half clock time later the $\overline{MREQ}$ signal goes active. At this time the address to the memory has had time to stabilize so that the falling edge of $\overline{MREQ}$ can be used directly as a chip enable clock to dynamic memories. The $\overline{RD}$ line also goes active to indicate that the memory read data should be enabled onto the CPU data bus. The CPU samples the data from the memory on the data bus with the rising edge of the clock of state T3 and this same edge is used by the CPU to turn off the $\overline{RD}$ and $\overline{MRQ}$ signals. Thus the data has already been sampled by the CPU before the $\overline{RD}$ signal becomes inactive. Clock state T3 and T4 of a fetch cycle are used to refresh dynamic memories. (The CPU uses this time to decode and execute the fetched instruction so that no other operation could be performed at this time). During T3 and T4 the lower 7 bits of the address bus contain a memory refresh address and the $\overline{RFSH}$ signal becomes active to indicate that a refresh read of all dynamic memories should be accomplished. Notice that a $\overline{RD}$ signal is not generated during refresh time to prevent data from different memory segments from being gated onto the data bus. The $\overline{MREQ}$ signal during refresh time should be used to perform a refresh read of all memory elements. The refresh signal can not be used by itself since the refresh address is only guaranteed to be stable during $\overline{MREQ}$ time.



INSTRUCTION OP CODE FETCH
FIGURE 4.0-1

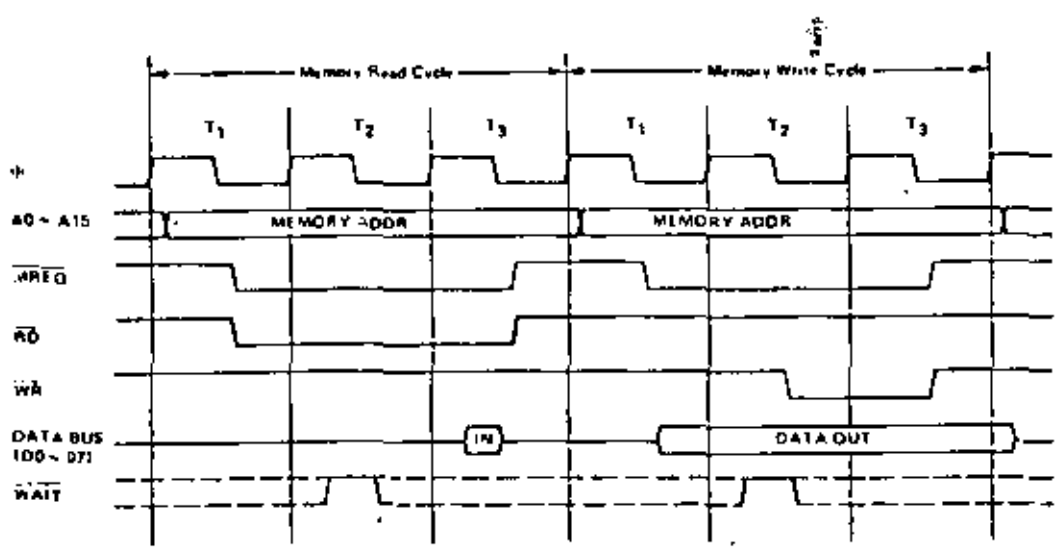
Figure 4.0-1A illustrates how the fetch cycle is delayed if the memory activates the $\overline{WAIT}$ line. During T2 and every subsequent Tw, the CPU samples the $\overline{WAIT}$ line with the falling edge of Φ. If the $\overline{WAIT}$ line is active at this time, another wait state will be entered during the following cycle. Using this technique the read cycle can be lengthened to match the access time of any type of memory device.



INSTRUCTION OP CODE FETCH WITH WAIT STATES
FIGURE 4.0-1A

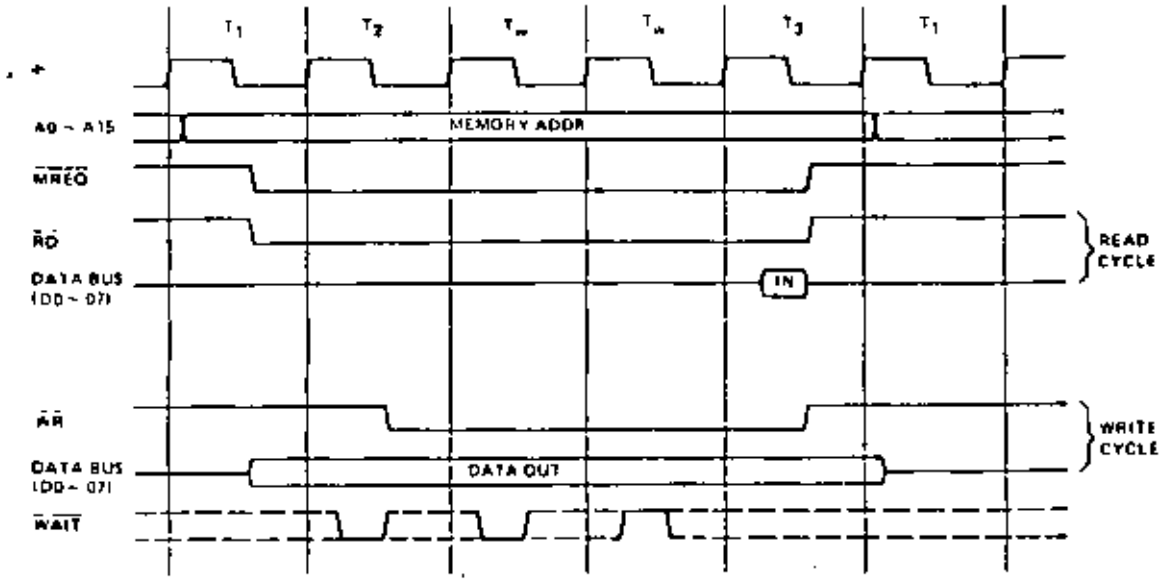
MEMORY READ OR WRITE

Figure 4.0-2 illustrates the timing of memory read or write cycles other than an OP code fetch (M1 cycle). These cycles are generally three clock periods long unless wait states are requested by the memory via the $\overline{WAIT}$ signal. The $\overline{MREQ}$ signal and the $\overline{RD}$ signal are used the same as in the fetch cycle. In the case of a memory write cycle, the $\overline{MREQ}$ also becomes active when the address bus is stable so that it can be used directly as a chip enable for dynamic memories. The $\overline{WR}$ line is active when data on the data bus is stable so that it can be used directly as a R/W pulse to virtually any type of semiconductor memory. Furthermore the $\overline{WR}$ signal goes inactive one half T state before the address and data bus contents are changed so that the overlap requirements for virtually any type of semiconductor memory type will be met.



MEMORY READ OR WRITE CYCLES
FIGURE 4.0-2

Figure 4.0-2A illustrates how a $\overline{WAIT}$ request signal will lengthen any memory read or write operation. This operation is identical to that previously described for a fetch cycle. Notice in this figure that a separate read and a separate write cycle are shown in the same figure although read and write cycles can never occur simultaneously.



MEMORY READ OR WRITE CYCLES WITH WAIT STATES
FIGURE 4.0-2A

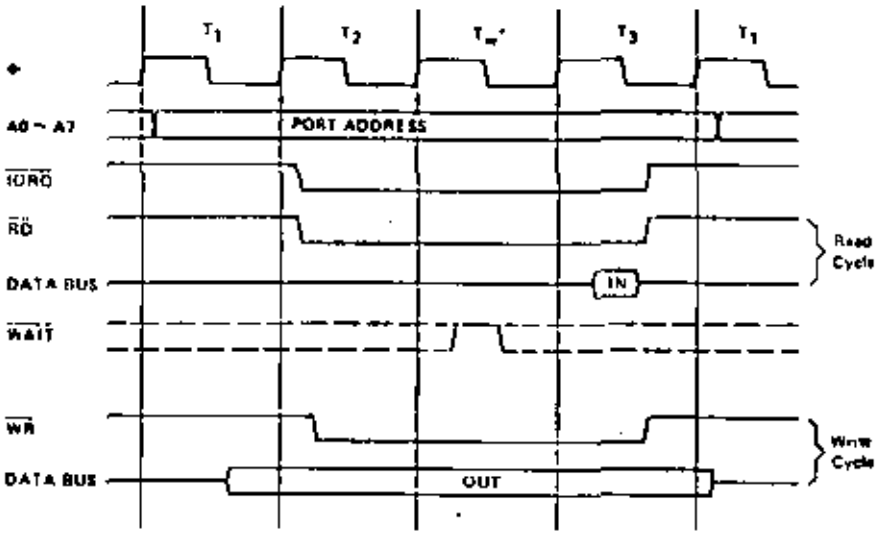
INPUT OR OUTPUT CYCLES

Figure 4.0-3 illustrates an I/O read or I/O write operation. Notice that during I/O operations a single wait state is automatically inserted. The reason for this is that during I/O operations, the time from when the $\overline{IORQ}$ signal goes active until the CPU must sample the $\overline{WAIT}$ line is very short and without this extra state sufficient time does not exist for an I/O port to decode its address and activate the $\overline{WAIT}$ line if a wait is required. Also, without this wait state it is difficult to design MOS I/O devices that can operate at full CPU speed. During this wait state time the $\overline{WAIT}$ request signal is sampled. During a read I/O operation, the $\overline{RD}$ line is used to enable the addressed port onto the data bus just as in the case of a memory read. For I/O write operations, the $\overline{WR}$ line is used as a clock to the I/O port, again with sufficient overlap timing automatically provided so that the rising edge may be used as a data clock.

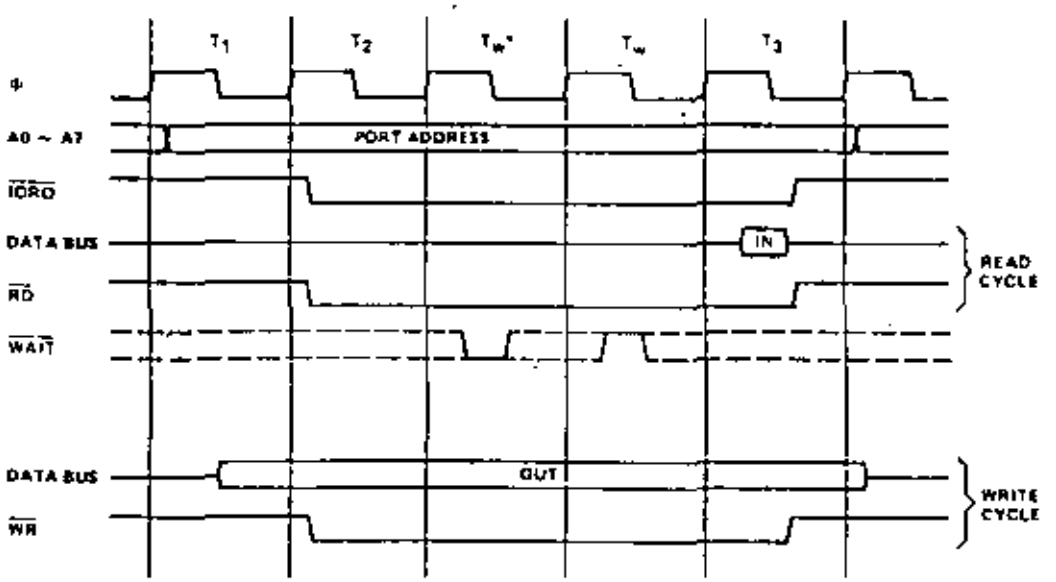
Figure 4.0-3A illustrates how additional wait states may be added with the $\overline{WAIT}$ line. The operation is identical to that previously described.

BUS REQUEST/ACKNOWLEDGE CYCLE

Figure 4.0-4 illustrates the timing for a Bus Request/Acknowledge cycle. The $\overline{BUSRQ}$ signal is sampled by the CPU with the rising edge of the last clock period of any machine cycle. If the $\overline{BUSRQ}$ signal is active, the CPU will set its address, data and tri-state control signals to the high impedance state with the rising edge of the next clock pulse. At that time any external device can control the buses to transfer data between memory and I/O devices. (This is generally known as Direct Memory Access [DMA] using cycle stealing). The maximum time for the CPU to respond to a bus request is the length of a machine cycle and the external controller can maintain control of the bus for as many clock cycles as is desired. Note, however, that if very long DMA cycles are used, and dynamic memories are being used, the external controller must also perform the refresh function. This situation only occurs if very large blocks of data are transferred under DMA control. Also note that during a bus request cycle, the CPU cannot be interrupted by either a $\overline{NMI}$ or an $\overline{INT}$ signal.

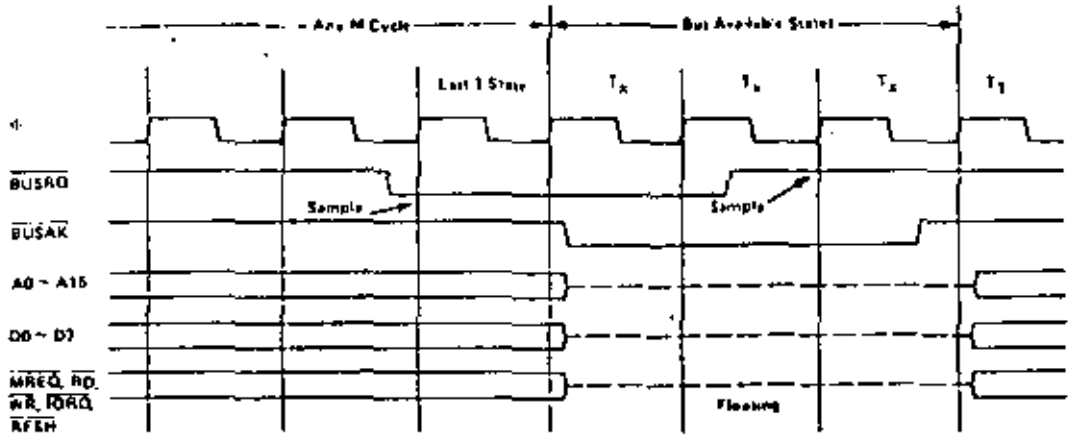


INPUT OR OUTPUT CYCLES
FIGURE 4.0-3



INPUT OR OUTPUT CYCLES WITH WAIT STATES
FIGURE 4.0-3A

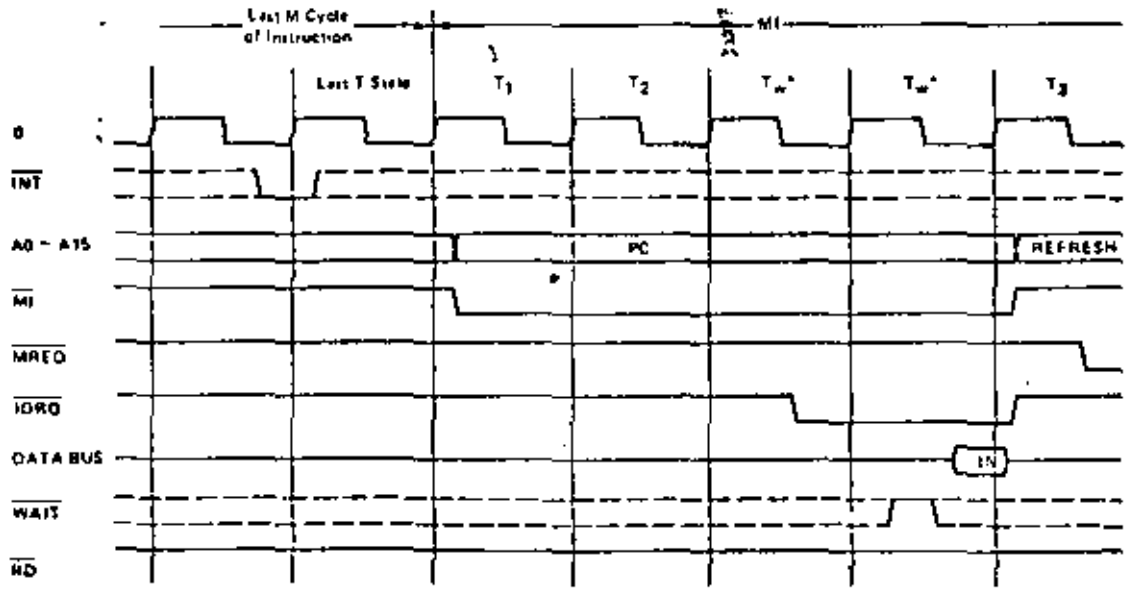
* Automatically inserted WAIT state



BUS REQUEST/ACKNOWLEDGE CYCLE
FIGURE 4.0-4

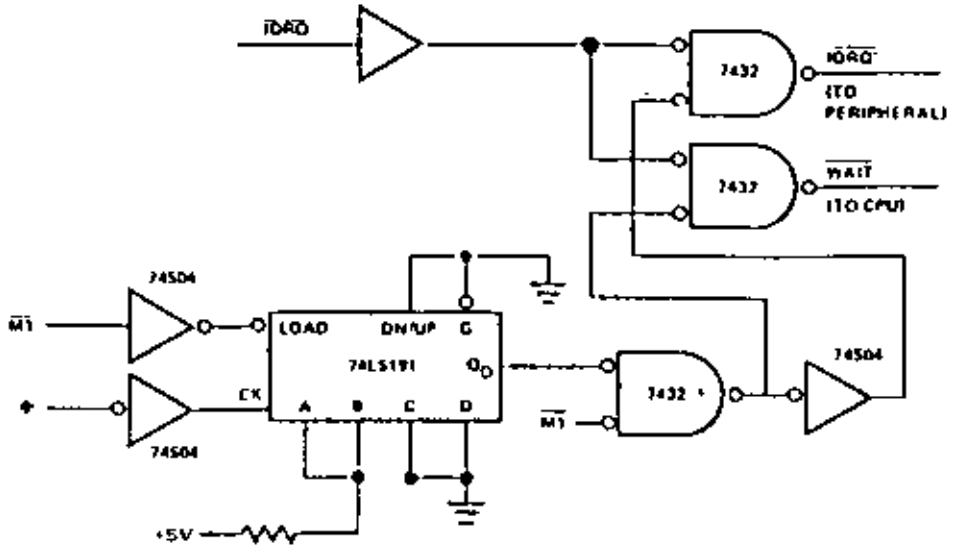
INTERRUPT REQUEST/ACKNOWLEDGE CYCLE

Figure 4.0-5 illustrates the timing associated with an interrupt cycle. The interrupt signal ($\overline{INT}$) is sampled by the CPU with the rising edge of the last clock at the end of any instruction. The signal will not be accepted if the internal CPU software controlled interrupt enable flip-flop is not set or if the $\overline{BUSRQ}$ signal is active. When the signal is accepted a special M1 cycle is generated. During this special M1 cycle the $\overline{IORQ}$ signal becomes active (instead of the normal $\overline{MREQ}$) to indicate that the interrupting device can place an 8-bit vector on the data bus. Notice that two wait states are automatically added to this cycle. These states are added so that a ripple priority interrupt scheme can be easily implemented. The two wait states allow sufficient time for the ripple signals to stabilize and identify which I/O device must insert the response vector. Refer to section 8.0 for details on how the interrupt response vector is utilized by the CPU.

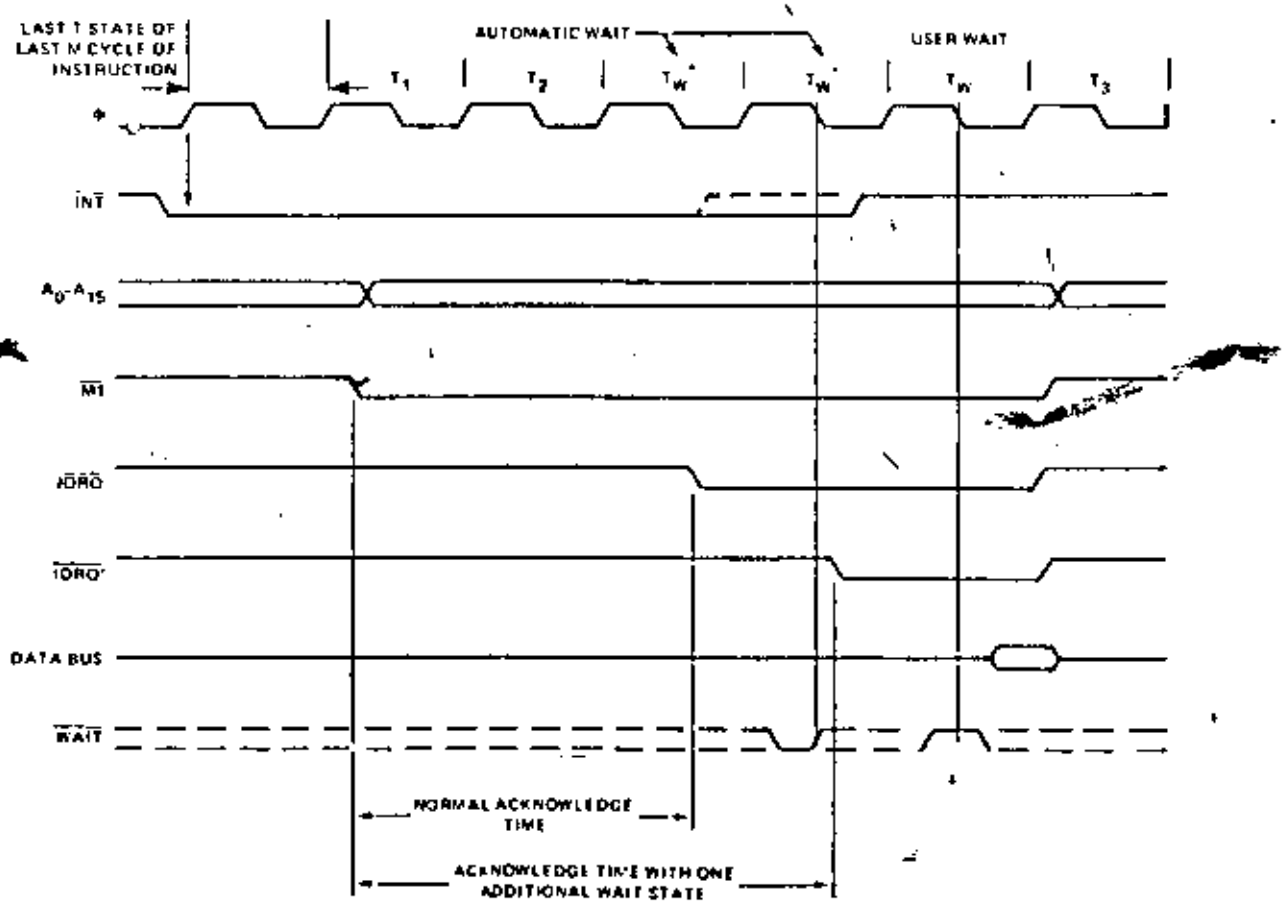


INTERRUPT REQUEST/ACKNOWLEDGE CYCLE
FIGURE 4.0-5

Figures 4.0-5A and 4.0-5B illustrate how a programmable counter can be used to extend interrupt acknowledge time. (Configured as shown to add one wait state)



EXTENDING INTERRUPT ACKNOWLEDGE TIME WITH WAIT STATE
FIGURE 4.0-5A



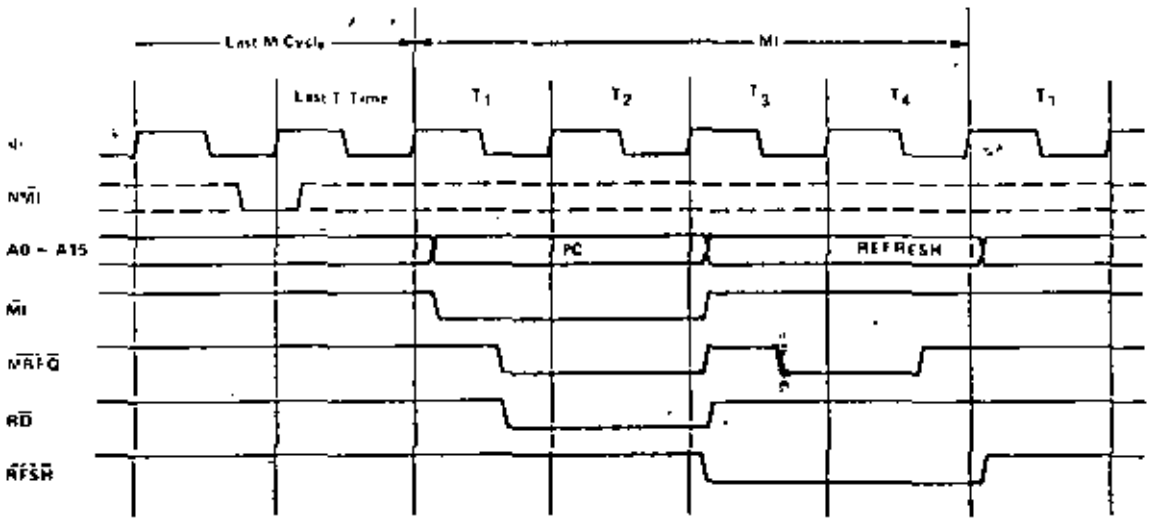
REQUEST/ACKNOWLEDGE CYCLE WITH ONE ADDITIONAL WAIT STATE
FIGURE 4.0-5B

NON MASKABLE INTERRUPT RESPONSE

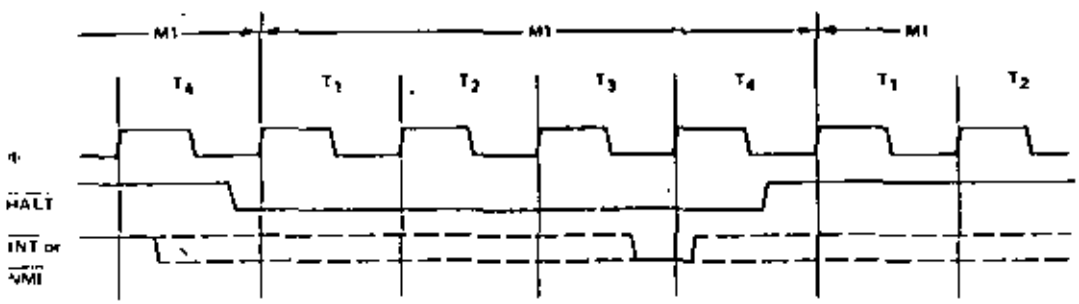
Figure 4.0-6 illustrates the request/acknowledge cycle for the non maskable interrupt. This signal is sampled at the same time as the interrupt line, but this line has priority over the normal interrupt and it can not be disabled under software control. Its usual function is to provide immediate response to important signals such as an impending power failure. The CPU response to a non maskable interrupt is similar to a normal memory read operation. The only difference being that the content of the data bus is ignored while the processor automatically stores the PC in the external stack and jumps to location 0066_H. The service routine for the non maskable interrupt must begin at this location if this interrupt is used.

HALT EXIT

Whenever a software halt instruction is executed the CPU begins executing NOP's until an interrupt is received (either a non maskable or a maskable interrupt while the interrupt flip flop is enabled). The two interrupt lines are sampled with the rising clock edge during each T4 state as shown in figure 4.0-7. If a non maskable interrupt has been received or a maskable interrupt has been received and the interrupt enable flip-flop is set, then the halt state will be exited on the next rising clock edge. The following cycle will then be an interrupt acknowledge cycle corresponding to the type of interrupt that was received. If both are received at this time, then the non maskable one will be acknowledged since it has highest priority. The purpose of executing NOP instructions while in the halt state is to keep the memory refresh signals active. Each cycle in the halt state is a normal M1 (fetch) cycle except that the data received from the memory is ignored and a NOP instruction is forced internally to the CPU. The halt acknowledge signal is active during this time to indicate that the processor is in the halt state.



NON MASKABLE INTERRUPT REQUEST OPERATION
FIGURE 4.0-6



HALT EXIT
FIGURE 4.0-7

5.0 Z-80 CPU INSTRUCTION SET

The Z-80 CPU can execute 158 different instruction types including all 78 of the 8080A CPU. The instructions can be broken down into the following major groups:

- Load and Exchange
- Block Transfer and Search
- Arithmetic and Logical
- Rotate and Shift
- Bit Manipulation (set, reset, test)
- Jump, Call and Return
- Input/Output
- Basic CPU Control

5.1 INTRODUCTION TO INSTRUCTION TYPES

The load instructions move data internally between CPU registers or between CPU registers and external memory. All of these instructions must specify a source location from which the data is to be moved and a destination location. The source location is not altered by a load instruction. Examples of load group instructions include moves between any of the general purpose registers such as move the data to Register B from Register C. This group also includes load immediate to any CPU register or to any external memory location. Other types of load instructions allow transfer between CPU registers and memory locations. The exchange instructions can trade the contents of two registers.

A unique set of block transfer instructions is provided in the Z-80. With a single instruction a block of memory of any size can be moved to any other location in memory. This set of block moves is extremely valuable when large strings of data must be processed. The Z-80 block search instructions are also valuable for this type of processing. With a single instruction, a block of external memory of any desired length can be searched for any 8-bit character. Once the character is found or the end of the block is reached, the instruction automatically terminates. Both the block transfer and the block search instructions can be interrupted during their execution so as to not occupy the CPU for long periods of time.

The arithmetic and logical instructions operate on data stored in the accumulator and other general purpose CPU registers or external memory locations. The results of the operations are placed in the accumulator and the appropriate flags are set according to the result of the operation. An example of an arithmetic operation is adding the accumulator to the contents of an external memory location. The results of the addition are placed in the accumulator. This group also includes 16-bit addition and subtraction between 16-bit CPU registers.

The rotate and shift group allows any register or any memory location to be rotated right or left with or without carry either arithmetic or logical. Also, a digit in the accumulator can be rotated right or left with two digits in any memory location.

The bit manipulation instructions allow any bit in the accumulator, any general purpose register or any external memory location to be set, reset or tested with a single instruction. For example, the most significant bit of register H can be reset. This group is especially useful in control applications and for controlling software flags in general purpose programming.

The jump, call and return instructions are used to transfer between various locations in the user's program. This group uses several different techniques for obtaining the new program counter address from specific external memory locations. A unique type of call is the restart instruction. This instruction actually contains the new address as a part of the 8-bit OP code. This is possible since only 8 separate addresses located in page zero of the external memory may be specified. Program jumps may also be achieved by loading register HL, IX or IY directly into the PC, thus allowing the jump address to be a complex function of the routine being executed.

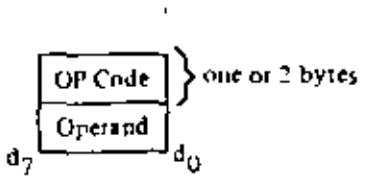
The input/output group of instructions in the Z-80 allow for a wide range of transfers between external memory locations or the general purpose CPU registers, and the external I/O devices. In each case, the port number is provided on the lower 8 bits of the address bus during any I/O transaction. One instruction allows this port number to be specified by the second byte of the instruction while other Z-80 instructions allow it to be specified as the content of the C register. One major advantage of using the C register as a pointer to the I/O device is that it allows different I/O ports to share common software driver routines. This is not possible when the address is part of the OP code if the routines are stored in ROM. Another feature of these input instructions is that they set the flag register automatically so that additional operations are not required to determine the state of the input data (for example its parity). The Z-80 CPU includes single instructions that can move blocks of data (up to 256 bytes) automatically to or from any I/O port directly to any memory location. In conjunction with the dual set of general purpose registers, these instructions provide for fast I/O block transfer rates. The value of this I/O instruction set is demonstrated by the fact that the Z-80 CPU can provide all required floppy disk formatting (i.e., the CPU provides the preamble, address, data and enables the CRC codes) on double density floppy disk drives on an interrupt driven basis.

Finally, the basic CPU control instructions allow various options and modes. This group includes instructions such as setting or resetting the interrupt enable flip flop or setting the mode of interrupt response.

5.2 ADDRESSING MODES

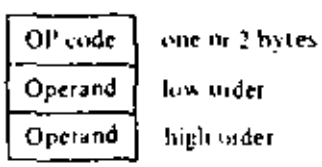
Most of the Z-80 instructions operate on data stored in internal CPU registers, external memory or in the I/O ports. Addressing refers to how the address of this data is generated in each instruction. This section gives a brief summary of the types of addressing used in the Z-80 while subsequent sections detail the type of addressing available for each instruction group.

Immediate. In this mode of addressing the byte following the OP code in memory contains the actual operand.



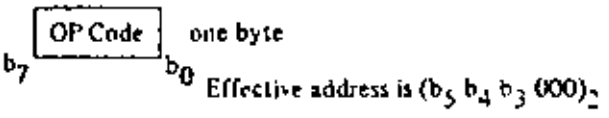
Examples of this type of instruction would be to load the accumulator with a constant, where the constant is the byte immediately following the OP code.

Immediate Extended. This mode is merely an extension of immediate addressing in that the two bytes following the OP codes are the operand.

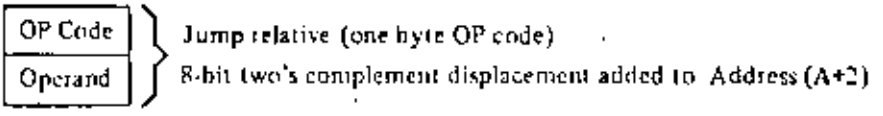


Examples of this type of instruction would be to load the HL register pair (16-bit register) with 16 bits (2 bytes) of data.

Modified Page Zero Addressing. The Z-80 has a special single byte CALL instruction to any of 8 locations in page zero of memory. This instruction (which is referred to as a restart) sets the PC to an effective address in page zero. The value of this instruction is that it allows a single byte to specify a complete 16-bit address where commonly called subroutines are located, thus saving memory space.

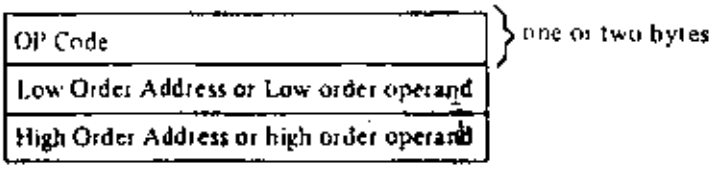


Relative Addressing. Relative addressing uses one byte of data following the OP code to specify a displacement from the existing program to which a program jump can occur. This displacement is a signed two's complement number that is added to the address of the OP code of the following instruction.



The value of relative addressing is that it allows jumps to nearby locations while only requiring two bytes of memory space. For most programs, relative jumps are by far the most prevalent type of jump due to the proximity of related program segments. Thus, these instructions can significantly reduce memory space requirements. The signed displacement can range between +127 and -128 from A + 2. This allows for a total displacement of +129 to -126 from the jump relative OP code address. Another major advantage is that it allows for relocatable code.

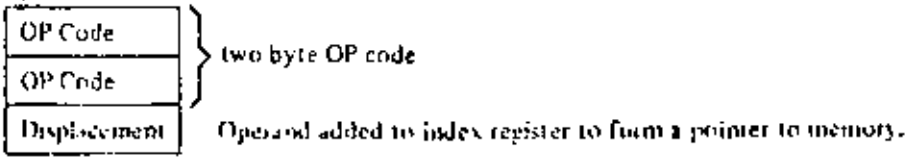
Extended Addressing. Extended Addressing provides for two bytes (16 bits) of address to be included in the instruction. This data can be an address to which a program can jump or it can be an address where an operand is located.



Extended addressing is required for a program to jump from any location in memory to any other location, or load and store data in any memory location.

When extended addressing is used to specify the source or destination address of an operand, the notation (nn) will be used to indicate the content of memory at nn, where nn is the 16-bit address specified in the instruction. This means that the two bytes of address nn are used as a pointer to a memory location. The use of the parentheses always means that the value enclosed within them is used as a pointer to a memory location. For example, (1200) refers to the contents of memory at location 1200.

Indexed Addressing. In this type of addressing, the byte of data following the OP code contains a displacement which is added to one of the two index registers (the OP code specifies which index register is used) to form a pointer to memory. The contents of the index register are not altered by this operation.



An example of an indexed instruction would be to load the contents of the memory location (Index Register + Displacement) into the accumulator. The displacement is a signed two's complement number. Indexed addressing greatly simplifies programs using tables of data since the index register can point to the start of any table. Two index registers are provided since very often operations require two or more tables. Indexed addressing also allows for relocatable code.

The two index registers in the Z-80 are referred to as IX and IY. To indicate indexed addressing the notation:

(IX+d) or (IY+d)

is used. Here d is the displacement specified after the OP code. The parentheses indicate that this value is used as a pointer to external memory.

Register Addressing. Many of the Z-80 OP codes contain bits of information that specify which CPU register is to be used for an operation. An example of register addressing would be to load the data in register B into register C.

Implied Addressing. Implied addressing refers to operations where the OP code automatically implies one or more CPU registers as containing the operands. An example is the set of arithmetic operations where the accumulator is always implied to be the destination of the results.

Register Indirect Addressing. This type of addressing specifies a 16-bit CPU register pair (such as HL) to be used as a pointer to any location in memory. This type of instruction is very powerful and it is used in a wide range of applications.

An example of this type of instruction would be to load the accumulator with the data in the memory location pointed to by the HL register contents. Indexed addressing is actually a form of register indirect addressing except that a displacement is added with indexed addressing. Register indirect addressing allows for very powerful but simple to implement memory accesses. The block move and search commands in the Z-80 are extensions of this type of addressing where automatic register incrementing, decrementing and comparing has been added. The notation for indicating register indirect addressing is to put parentheses around the name of the register that is to be used as the pointer. For example, the symbol

(HL)

specifies that the contents of the HL register are to be used as a pointer to a memory location. Often register indirect addressing is used to specify 16-bit operands. In this case, the register contents point to the lower order portion of the operand while the register contents are automatically incremented to obtain the upper portion of the operand.

Bit Addressing. The Z-80 contains a large number of bit set, reset and test instructions. These instructions allow any memory location or CPU register to be specified for a bit operation through one of three previous addressing modes (register, register indirect and indexed) while three bits in the OP code specify which of the eight bits is to be manipulated.

ADDRESSING MODE COMBINATIONS

Many instructions include more than one operand (such as arithmetic instructions or loads). In these cases, two types of addressing may be employed. For example, load can use immediate addressing to specify the source and register indirect or indexed addressing to specify the destination.

5.3 INSTRUCTION OP CODES

This section describes each of the Z-80 instructions and provides tables listing the OP codes for every instruction. In each of these tables the OP codes in shaded areas are identical to those offered in the 8080A CPU. Also shown is the assembly language mnemonic that is used for each instruction. All instruction OP codes are listed in hexadecimal notation. Single byte OP codes require two hex characters while double byte OP codes require four hex characters. The conversion from hex to binary is repeated here for convenience.

Hex		Binary		Decimal	Hex		Binary		Decimal
0	=	0000	=	0	8	=	1000	=	8
1	=	0001	=	1	9	=	1001	=	9
2	=	0010	=	2	A	=	1010	=	10
3	=	0011	=	3	B	=	1011	=	11
4	=	0100	=	4	C	=	1100	=	12
5	=	0101	=	5	D	=	1101	=	13
6	=	0110	=	6	E	=	1110	=	14
7	=	0111	=	7	F	=	1111	=	15

Z-80 instruction mnemonics consist of an OP code and zero, one or two operands. Instructions in which the operand is implied have no operand. Instructions which have only one logical operand or those in which one operand is invariant (such as the Logical OR instruction) are represented by a one operand mnemonic. Instructions which may have two varying operands are represented by two operand mnemonics.

LOAD AND EXCHANGE

Table 5.3-1 defines the OP code for all of the 8-bit load instructions implemented in the Z-80 CPU. Also shown in this table is the type of addressing used for each instruction. The source of the data is found on the top horizontal row while the destination is specified by the left hand column. For example, load register C from register B uses the OP code 48H. In all of the tables the OP code is specified in hexadecimal notation and the 48H (=0100 1000 binary) code is fetched by the CPU from the external memory during M1 time, decoded and then the register transfer is automatically performed by the CPU.

The assembly language mnemonic for this entire group is LD, followed by the destination followed by the source (LD DEST., SOURCE). Note that several combinations of addressing modes are possible. For example, the source may use register addressing and the destination may be register indirect; such as load the memory location pointed to by register HL with the contents of register D. The OP code for this operation would be 72. The mnemonic for this load instruction would be as follows:

LD (HL), D

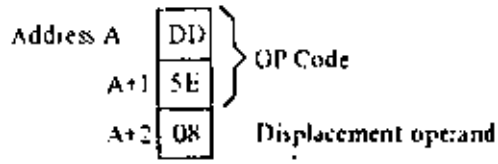
The parentheses around the HL means that the contents of HL are used as a pointer to a memory location. In all Z-80 load instruction mnemonics the destination is always listed first, with the source following. The Z-80 assembly language has been defined for ease of programming. Every instruction is self documenting and programs written in Z-80 language are easy to maintain.

Note in table 5.3-1 that some load OP codes that are available in the Z-80 use two bytes. This is an efficient method of memory utilization since 8, 16, 24 or 32 bit instructions are implemented in the Z-80. Thus often utilized instructions such as arithmetic or logical operations are only 8-bits which results in better memory utilization than is achieved with fixed instruction sizes such as 16-bits.

All load instructions using indexed addressing for either the source or destination location actually use three bytes of memory with the third byte being the displacement d. For example a load register E with the operand pointed to by IX with an offset of +8 would be written:

LD E, (IX + 8)

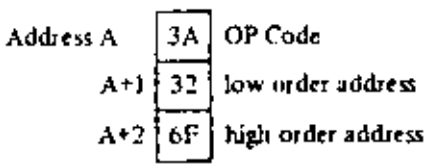
The instruction sequence for this in memory would be:



The two extended addressing instructions are also three byte instructions. For example the instruction to load the accumulator with the operand in memory location 6F32H would be written:

LD A,(6F32H)

and its instruction sequence would be:

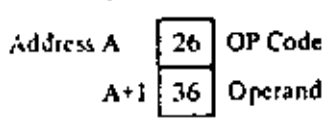


Notice that the low order portion of the address is always the first operand.

The load immediate instructions for the general purpose 8-bit registers are two-byte instructions. The instruction load register H with the value 36H would be written:

LD H, 36H

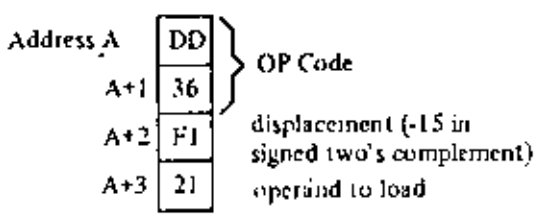
and its sequence would be:



Loading a memory location using indexed addressing for the destination and immediate addressing for the source requires four bytes. For example:

LD (IX – 15), 21H

would appear as:



Notice that with any indexed addressing the displacement always follows directly after the OP code.

Table 5.3-2 specifies the 16-bit load operations. This table is very similar to the previous one. Notice that the extended addressing capability covers all register pairs. Also notice that register indirect operations specifying the stack pointer are the PUSH and POP instructions. The mnemonic for these instructions is "PUSH" and "POP." These differ from other 16-bit loads in that the stack pointer is automatically decremented and incremented as each byte is pushed onto or popped from the stack respectively. For example the instruction:

PUSH AF

is a single byte instruction with the OP code of F5H. When this instruction is executed the following sequence is generated:

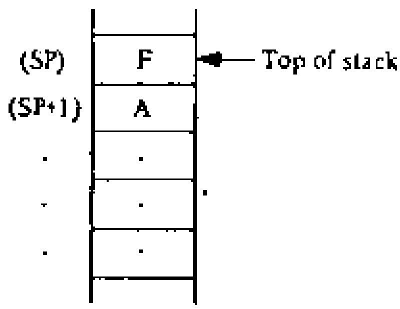
Decrement SP

LD (SP), A

Decrement SP

LD (SP), F

Thus the external stack now appears as follows:



SOURCE

DESTINATION		IMPLIED I	IMPLIED R	REGISTER A	B	C	D	E	H	L	REG INDIRECT (HL)	(BC)	(DE)	INDEXED (IX+d)	(IY+d)	EXT. ADDR (nn)	IMME. n
REGISTER	A	ED 57	ED 5F	7F	78	79	7A	7B	7C	7D	7E	0A	1A	DD 7E d	FD 7E d	3A n n	3E n
	B			47	40	41	42	43	44	45	46			DD 46 d	FD 46 d		06 n
	C			4F	48	49	4A	4B	4C	4D	4E			DD 4E d	FD 4E d		0E n
	D			57	50	51	52	53	54	55	56			DD 56 d	FD 56 d		16 n
	E			5F	58	59	5A	5B	5C	5D	5E			DD 5E d	FD 5E d		1E n
	H			67	60	61	62	63	64	65	66			DD 66 d	FD 66 d		26 n
	L			6F	68	69	6A	6B	6C	6D	6E			DD 6E d	FD 6E d		2E n
REG INDIRECT	(HL)			77	70	71	72	73	74	75							36 n
	(BC)			02													
	(DE)			12													
INDEXED	(IX+d)			DD 77 d	DD 70 d	DD 71 d	DD 72 d	DD 73 d	DD 74 d	DD 75 d							DD 36 d n
	(IY+d)			FD 77 d	FD 70 d	FD 71 d	FD 72 d	FD 73 d	FD 74 d	FD 75 d							FD 36 d n
EXT. ADDR	(nn)			32 n n													
IMPLIED	I			ED 47													
	R			ED 4F													

8 BIT LOAD GROUP

'LD'

TABLE 5.3–1

The POP instruction is the exact reverse of a PUSH. Notice that all PUSH and POP instructions utilize a 16-bit operand and the high order byte is always pushed first and popped last. That is a:

PUSH BC is PUSH B then C
PUSH DE is PUSH D then E
PUSH HL is PUSH H then L
POP HL is POP L then H

The instruction using extended immediate addressing for the source obviously requires 2 bytes of data following the OP code. For example:

LD DE, 0659H

will be:

Address A	11	OP Code
A+1	59	Low order operand to register E
A+2	06	High order operand to register D

In all extended immediate or extended addressing modes, the low order byte always appears first after the OP code.

Table 5.3-3 lists the 16-bit exchange instructions implemented in the Z-80. OP code 08H allows the programmer to switch between the two pairs of accumulator flag registers while D9H allows the programmer to switch between the duplicate set of six general purpose registers. These OP codes are only one byte in length to absolutely minimize the time necessary to perform the exchange so that the duplicate banks can be used to effect very fast interrupt response times.

BLOCK TRANSFER AND SEARCH

Table 5.3-4 lists the extremely powerful block transfer instructions. All of these instructions operate with three registers.

HL points to the source location.
DE points to the destination location.
BC is a byte counter.

After the programmer has initialized these three registers, any of these four instructions may be used. The LDI (Load and Increment) instruction moves one byte from the location pointed to by HL to the location pointed to by DE. Register pairs HL and DE are then automatically incremented and are ready to point to the following locations. The byte counter (register pair BC) is also decremented at this time. This instruction is valuable when blocks of data must be moved but other types of processing are required between each move. The LDIR (Load, increment and repeat) instruction is an extension of the LDI instruction. The same load and increment operation is repeated until the byte counter reaches the count of zero. Thus, this single instruction can move any block of data from one location to any other.

Note that since 16-bit registers are used, the size of the block can be up to 64K bytes (1K = 1024) long and it can be moved from any location in memory to any other location. Furthermore the blocks can be overlapping since there are absolutely no constraints on the data that is used in the three register pairs.

The LDD and LDDR instructions are very similar to the LDI and LDIR. The only difference is that register pairs HL and DE are decremented after every move so that a block transfer starts from the highest address of the designated block rather than the lowest.

SOURCE

DESTINATION		REGISTER							IMM. EXT.	EXT. ADDR.	REG. INDIR.
		AF	BC	DE	HL	SP	IX	IY	nn	(nn)	(SP)
REGISTER	AF										F1
	BC								01 n n	ED 4B n n	C1
	DE								11 n n	ED 5B n n	D1
	HL								21 n n	2A n n	E1
	SP				F9		DD F9	FD F9	31 n n	ED 7B n n	
	IX								DD 21 n n	DD 2A n n	DD E1
	IY								FD 21 n n	FD 2A n n	FD E1
EXT. ADDR.	(nn)		ED 43 n n	ED 53 n n	22 n n	ED 73 n n	DD 22 n n	FD 22 n n			
REG. IND.	(SP)	F5	C5	D5	E5		DD E5	FD E5			

PUSH INSTRUCTIONS

POP INSTRUCTIONS

NOTE: The Push & Pop instructions adjust the SP after every execution

16 BIT LOAD GROUP
'LD'
'PUSH' AND 'POP'
TABLE 5.3–2

		IMPLIED ADDRESSING				
		AF	BC, DE & HL	HL	IX	IY
IMPLIED	AF	08				
	BC, DE & HL		D9			
	DE			EB		
REG. INDIR.	(SP)			E3	DD E3	FD E3

EXCHANGES
'EX' AND 'EXX'
TABLE 5.3–3

			SOURCE: REG. INDIR	SOURCE: (HL)	
DESTINATION: REG INDIR.	(DE)		ED A0		'LDI' – Load (DE)←(HL) Inc HL & DE, Dec BC
			ED B0		'LDIR' – Load (DE)←(HL) Inc HL & DE, Dec BC, Repeat until BC = 0
			ED A8		'LDD' – Load (DE)←(HL) Dec HL & DE, Dec BC
			ED B8		'LDDR' – Load (DE)←(HL) Dec HL & DE, Dec BC, Repeat until BC = 0

Reg HL points to source
Reg DE points to destination
Reg BC is byte counter

BLOCK TRANSFER GROUP
TABLE 5.3–4

Table 5.3-5 specifies the OP codes for the four block search instructions. The first, CPI (compare and increment) compares the data in the accumulator, with the contents of the memory location pointed to by register HL. The result of the compare is stored in one of the flag bits (see section 6.0 for a detailed explanation of the flag operations) and the HL register pair is then incremented and the byte counter (register pair BC) is decremented.

The instruction CPIR is merely an extension of the CPI instruction in which the compare is repeated until either a match is found or the byte counter (register pair BC) becomes zero. Thus, this single instruction can search the entire memory for any 8-bit character.

The CPD (Compare and Decrement) and CPDR (Compare, Decrement and Repeat) are similar instructions, their only difference being that they decrement HL after every compare so that they search the memory in the opposite direction. (The search is started at the highest location in the memory block).

It should be emphasized again that these block transfer and compare instructions are extremely powerful in string manipulation applications.

ARITHMETIC AND LOGICAL

Table 5.3-6 lists all of the 8-bit arithmetic operations that can be performed with the accumulator, also listed are the increment (INC) and decrement (DEC) instructions. In all of these instructions, except INC and DEC, the specified 8-bit operation is performed between the data in the accumulator and the source data specified in the table. The result of the operation is placed in the accumulator with the exception of compare (CP) that leaves the accumulator unaffected. All of these operations affect the flag register as a result of the specified operation. (Section 6.0 provides all of the details on how the flags are affected by any instruction type). INC and DEC instructions specify a register or a memory location as both source and destination of the result. When the source operand is addressed using the index registers the displacement must follow directly. With immediate addressing the actual operand will follow directly. For example the instruction:

AND 07H

would appear as:

Address A	E6	OP Code
A+1	07	Operand

SEARCH LOCATION

REG. INDIR. (HL)	
ED A1	'CPI' Inc HL, Dec BC
ED B1	'CPIR', Inc HL, Dec BC repeat until BC = 0 or find match
ED A9	'CPD' Dec HL & BC
ED B9	'CPDR' Dec HL & BC Repeat until BC = 0 or find match

HL points to location in memory to be compared with accumulator contents
BC is byte counter

BLOCK SEARCH GROUP
TABLE 5.3–5

Assuming that the accumulator contained the value F3H the result of 03H would be placed in the accumulator:

Acc before operation	1111 0011 = F3H
Operand	0000 0111 = 07H
Result to Acc	0000 0011 = 03H

The Add instruction (ADD) performs a binary add between the data in the source location and the data in the accumulator. The subtract (SUB) does a binary subtraction. When the add with carry is specified (ADC) or the subtract with carry (SBC), then the carry flag is also added or subtracted respectively. The flags and decimal adjust instruction (DAA) in the Z-80 (fully described in section 6.0) allow arithmetic operations for:

multiprecision packed BCD numbers

multiprecision signed or unsigned binary numbers

multiprecision two's complement signed numbers

Other instructions in this group are logical and (AND), logical or (OR), exclusive or (XOR) and compare (CP).

There are five general purpose arithmetic instructions that operate on the accumulator or carry flag. These five are listed in table 5.3-7. The decimal adjust instruction can adjust for subtraction as well as addition, thus making BCD arithmetic operations simple. Note that to allow for this operation the flag N is used. This flag is set if the last arithmetic operation was a subtract. The negate accumulator (NEG) instruction forms the two's complement of the number in the accumulator. Finally notice that a reset carry instruction is not included in the Z-80 since this operation can be easily achieved through other instructions such as a logical AND of the accumulator with itself.

Table 5.3-8 lists all of the 16-bit arithmetic operations between 16-bit registers. There are five groups of instructions including add with carry and subtract with carry. ADC and SBC affect all of the flags. These two groups simplify address calculation operations or other 16-bit arithmetic operations.

SOURCE

	REGISTER ADDRESSING							REG. INDIR.	INDEXED		IMMED.
	A	B	C	D	E	H	L	(HL)	(IX+d)	(IY+d)	n
'ADD'	87	80	81	82	83	84	85	86	DD 86 d	FD 86 d	C6 n
ADD w CARRY 'ADC'	8F	88	89	8A	8B	8C	8D	8E	DD 8E d	FD 8E d	CE n
SUBTRACT 'SUB'	97	90	91	92	93	94	95	96	DD 96 d	FD 96 d	D6 n
SUB w CARRY 'SBC'	9F	98	99	9A	9B	9C	9D	9E	DD 9E d	FD 9E d	DE n
'AND'	A7	A0	A1	1A2	A3	A4	A5	A6	DD A6 d	FD A6 d	E6 n
'XOR'	AF	A8	A9	AA	AB	AC	AD	AE	DD AE d	FD AE d	EE n
'OR'	B7	B0	B1	B2	B3	B4	B5	B6	DD B6 d	FD B6 d	F6 n
COMPARE 'CP'	BF	B8	B9	BA	BB	BC	BD	BE	DD BE d	FD BE d	FE n
INCREMENT 'INC'	3C	04	0C	14	1C	24	2C	34	DD 34 d	FD 34 d	
DECREMENT 'DEC'	3D	05	0D	15	1D	25	2D	35	DD 35 d	FD 35 d	

8 BIT ARITHMETIC AND LOGIC
TABLE 5.3–6

Decimal Adjust Acc, 'DAA'	27
Complement Acc, 'CPL'	2F
Negate Acc, 'NEG' (2's complement)	ED 44
Complement Carry Flag, 'CCF'	3F
Set Carry Flag, 'SCF'	37

GENERAL PURPOSE AF OPERATIONS
TABLE 5.3-7

DESTINATION		SOURCE: BC	DE	HL	SP	IX	IY
'ADD'	HL	09	19	29	39		
'ADD'	IX	DD 09	DD 19		DD 39	DD 29	
'ADD'	IY	FD 09	FD 19		FD 39		FD 29
ADD WITH CARRY AND SET FLAGS 'ADC'	HL	ED 4A	ED 5A	ED 6A	ED 7A		
SUB WITH CARRY AND SET FLAGS 'SBC'	HL	ED 42	ED 52	ED 62	ED 72		
INCREMENT 'INC'		03	13	23	33	DD 23	FD 23
DECREMENT 'DEC'		0B	1B	2B	3B	DD 2B	FD 2B

16 BIT ARITHMETIC
TABLE 5.3–8

ROTATE AND SHIFT

A major capability of the Z-80 is its ability to rotate or shift data in the accumulator, any general purpose register, or any memory location. All of the rotate and shift OP codes are shown in table 5.3-9. Also included in the Z-80 are arithmetic and logical shift operations. These operations are useful in an extremely wide range of applications including integer multiplication and division. Two BCD digit rotate instructions (RRD and RLD) allow a digit in the accumulator to be rotated with the two digits in a memory location pointed to by register pair HL. (See figure 5.3-9). These instructions allow for efficient BCD arithmetic.

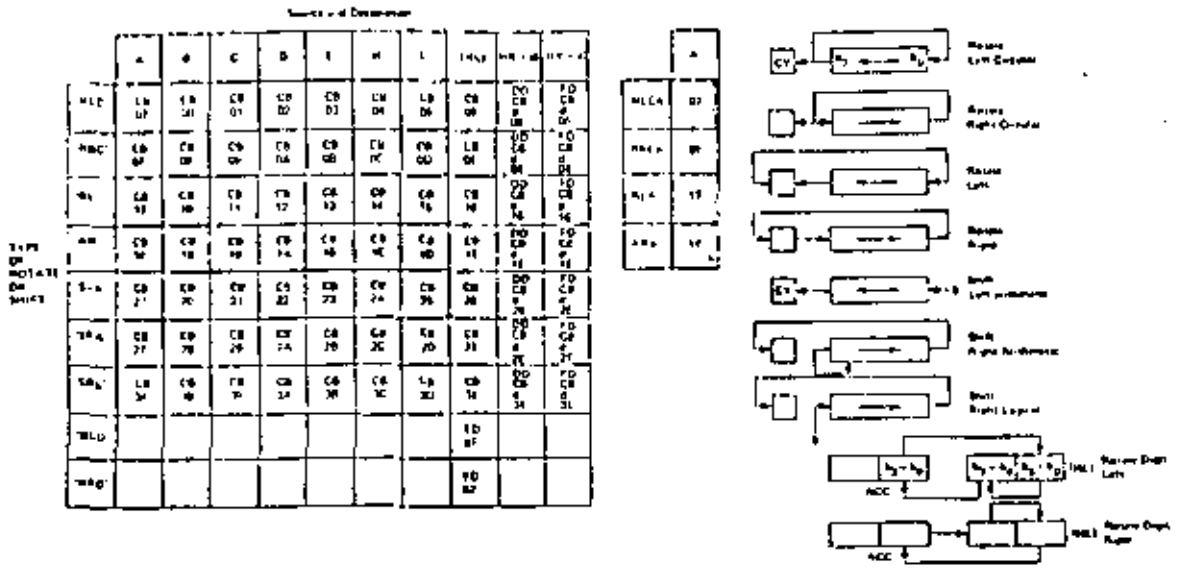
BIT MANIPULATION

The ability to set, reset and test individual bits in a register or memory location is needed in almost every program. These bits may be flags in a general purpose software routine, indications of external control conditions or data packed into memory locations to make memory utilization more efficient.

The Z-80 has the ability to set, reset or test any bit in the accumulator, any general purpose register or any memory location with a single instruction. Table 5.3-10 lists the 240 instructions that are available for this purpose. Register addressing can specify the accumulator or any general purpose register on which the operation is to be performed. Register indirect and indexed addressing are available to operate on external memory locations. Bit test operations set the zero flag (Z) if the tested bit is a zero. (Refer to section 6.0 for further explanation of flag operation).

JUMP, CALL AND RETURN

Figure 5.3-11 lists all of the jump, call and return instructions implemented in the Z-80 CPU. A jump is a branch in a program where the program counter is loaded with the 16-bit value as specified by one of the three available addressing modes (Immediate Extended, Relative or Register Indirect). Notice that the jump group has several different conditions that can be specified to be met before the jump will be made. If these conditions are not met, the program merely continues with the next sequential instruction. The conditions are all dependent on the data in the flag register. (Refer to section 6.0 for details on the flag register). The immediate extended addressing is used to jump to any location in the memory. This instruction requires three bytes (two to specify the 16-bit address) with the low order address byte first followed by the high order address byte.



ROTATES AND SHIFTS
TABLE 5.3-9

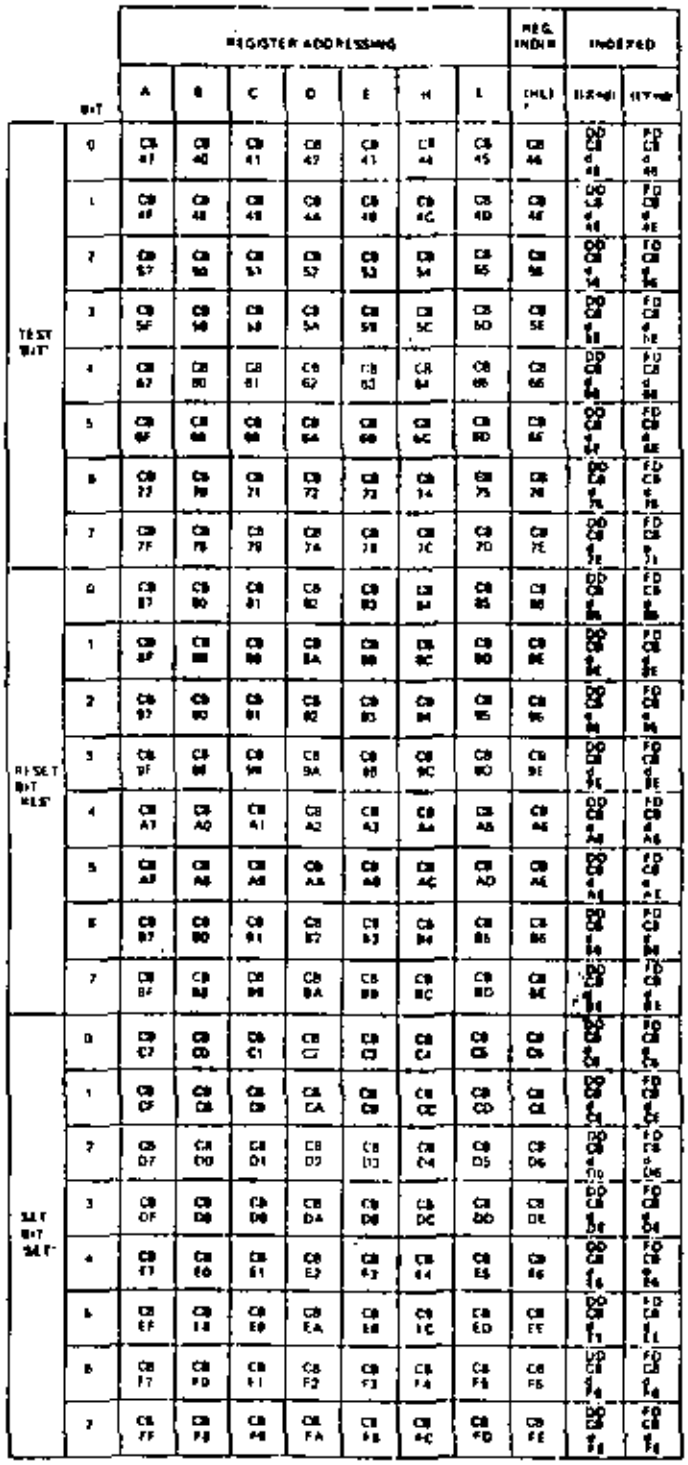
For example an unconditional Jump to memory location 3E32H would be:

Address A	C3	OP Code
A+1	32	Low order address
A+2	3E	High order address

The relative jump instruction uses only two bytes, the second byte is a signed two's complement displacement from the existing PC. This displacement can be in the range of +129 to -126 and is measured from the address of the instruction OP code.

Three types of register indirect jumps are also included. These instructions are implemented by loading the register pair HL or one of the index registers IX or IY directly into the PC. This capability allows for program jumps to be a function of previous calculations.

A call is a special form of a jump where the address of the byte following the call instruction is pushed onto the stack before the jump is made. A return instruction is the reverse of a call because the data on the top of the stack is popped directly into the PC to form a jump address. The call and return instructions allow for simple subroutine and interrupt handling. Two special return instructions have been included in the Z-80 family of components. The return from interrupt instruction (RETI) and the return from non maskable interrupt (RETN) are treated in the CPU as an unconditional return identical to the OP code C9H. The difference is that (RETI) can be used at the end of an interrupt routine and all Z-80 peripheral chips will recognize the execution of this instruction for proper control of nested priority interrupt handling. This instruction coupled with the Z-80 peripheral devices implementation simplifies the normal return from nested interrupt. Without this feature the following software sequence would be necessary to inform the interrupting device that the interrupt routine is completed:



BIT MANIPULATION GROUP
TABLE 5.3-10

Disable Interrupt	— prevent interrupt before routine is exited.
LD A, n OUT n, A	— notify peripheral that service routine is complete
Enable Interrupt	
Return	

This seven byte sequence can be replaced with the one byte EI instruction and the two byte RETI instruction in the Z80. This is important since interrupt service time often must be minimized.

To facilitate program loop control the instruction DJNZ e can be used advantageously. This two byte, relative jump instruction decrements the B register and the jump occurs if the B register has not been decremented to zero. The relative displacement is expressed as a signed two's complement number. A simple example of its use might be:

Address	Instruction	Comments
N, N + 1	LD B, 7	;set B register to count of 7
N + 2 to N + 9	(Perform a sequence of instructions)	;loop to be performed 7 times
N + 10, N + 11	DJNZ –8	;to jump from N + 12 to N + 2
N + 12	(Next Instruction)	

CONDITION

			UN-COND.	CARRY	NON CARRY	ZERO	NON ZERO	PARITY EVEN	PARITY ODD	SIGN NEG	SIGN POS	REG B=0
JUMP 'JP'	IMMED. EXT.	nn	C3 n n	DA n n	D2 n n	CA n n	C2 n n	EA n n	E2 n n	FA n n	F2 n n	
JUMP 'JR'	RELATIVE	PC+e	18 e-2	38 e-2	30 e-2	28 e-2	20 e-2					
JUMP 'JP'	REG. INDIR.	(HL)	E9									
JUMP 'JP'		(IX)	DD E9									
JUMP 'JP'		(IY)	FD E9									
'CALL'	IMMED. EXT.	nn	CD n n	DC n n	D4 n n	CC n n	C4 n n	EC n n	E4 n n	FC n n	F4 n n	
DECREMENT B, JUMP IF NON ZERO 'DJNZ'	RELATIVE	PC+e										10 e-2
RETURN 'RET'	REGISTER INDIR.	(SP) (SP+1)	C9	D8	D0	C8	C0	E8	E0	F8	F0	
RETURN FROM INT 'RETI'	REG. INDIR.	(SP) (SP+1)	ED 4D									
RETURN FROM NON MASKABLE INT 'RETN'	REG. INDIR.	(SP) (SP+1)	ED 45									

NOTE—CERTAIN FLAGS HAVE MORE THAN ONE PURPOSE. REFER TO SECTION 6.0 FOR DETAILS

JUMP, CALL and RETURN GROUP
TABLE 5.3–11

Table 5.3-12 lists the eight OP codes for the restart instruction. This instruction is a single byte call to any of the eight addresses listed. The simple mnemonic for these eight calls is also shown. The value of this instruction is that frequently used routines can be called with this instruction to minimize memory usage.

	OP CODE	
CALL ADDRESS 0000_H	C7	'RST 0'
0008_H	CF	'RST 8'
0010_H	D7	'RST 16'
0018_H	DF	'RST 24'
0020_H	E7	'RST 32'
0028_H	EF	'RST 40'
0030_H	F7	'RST 48'
0038_H	FF	'RST 56'

RESTART GROUP
TABLE 5.3—12

INPUT/OUTPUT

The Z-80 has an extensive set of Input and Output instructions as shown in table 5.3-13 and table 5.3-14. The addressing of the input or output device can be either absolute or register indirect, using the C register. Notice that in the register indirect addressing mode data can be transferred between the I/O devices and any of the internal registers. In addition eight block transfer instructions have been implemented. These instructions are similar to the memory block transfers except that they use register pair HL for a pointer to the memory source (output commands) or destination (input commands) while register B is used as a byte counter. Register C holds the address of the port for which the input or output command is desired. Since register B is eight bits in length, the I/O block transfer command handles up to 256 bytes.

In the instructions IN A, n and OUT n, A the I/O device address n appears in the lower half of the address bus (A_0-A_7) while the accumulator content is transferred in the upper half of the address bus. In all register indirect input output instructions, including block I/O transfers the content of register C is transferred to the lower half of the address bus (device address) while the content of register B is transferred to the upper half of the address bus.

INPUT DESTINATION			SOURCE PORT ADDRESS: IMMED. (n)	SOURCE PORT ADDRESS: REG. INDIR. (C)
INPUT 'IN'	REG ADDRESSING	A	DB n	ED 78
		B		ED 40
		C		ED 48
		D		ED 50
		E		ED 58
		H		ED 60
		L		ED 68
'INI' – INPUT & Inc HL, Dec B	REG. INDIR	(HL)		ED A2
'INIR' – INP, Inc HL, Dec B, REPEAT IF B≠0				ED B2
'IND' – INPUT & Dec HL, Dec B				ED AA
'INDR' – INPUT, Dec HL, Dec B, REPEAT IF B≠0				ED BA

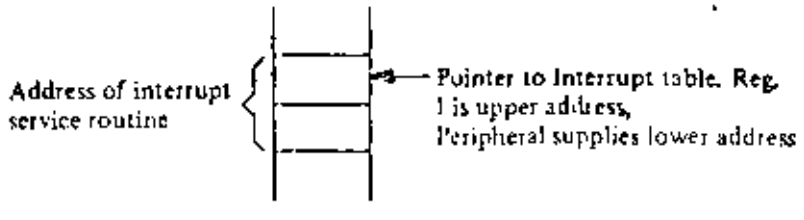
BLOCK INPUT COMMANDS

INPUT GROUP

TABLE 5.3–13

CPU CONTROL GROUP

The final table, table 5.3-15 illustrates the six general purpose CPU control instructions. The NOP is a do-nothing instruction. The HALT instruction suspends CPU operation until a subsequent interrupt is received, while the DI and EI are used to lock out and enable interrupts. The three interrupt mode commands set the CPU into any of the three available interrupt response modes as follows. If mode zero is set the interrupting device can insert any instruction on the data bus and allow the CPU to execute it. Mode 1 is a simplified mode where the CPU automatically executes a restart (RST) to location 0038H so that no external hardware is required. (The old PC content is pushed onto the stack). Mode 2 is the most powerful in that it allows for an indirect call to any location in memory. With this mode the CPU forms a 16-bit memory address where the upper 8-bits are the content of register I and the lower 8-bits are supplied by the interrupting device. This address points to the first of two sequential bytes in a table where the address of the service routine is located. The CPU automatically obtains the starting address and performs a CALL to this address.



SOURCE

			REGISTER							REG. IND.
			A	B	C	D	E	H	L	(HL)
'OUT'	IMMED.	(n)	D3 n							
	REG. IND.	(C)	ED 79	ED 41	ED 49	ED 51	ED 59	ED 61	ED 69	
'OUTI' – OUTPUT Inc HL, Dec b	REG. IND.	(C)								ED A3
'OTIR' – OUTPUT, Inc HL, Dec B, REPEAT IF B≠0	REG. IND.	(C)								ED B3
'OUTD' – OUTPUT Dec HL & B	REG. IND.	(C)								ED AB
'OTDR' – OUTPUT, Dec HL & B, REPEAT IF B≠0	REG. IND.	(C)								ED BB

BLOCK OUTPUT COMMANDS

PORT DESTINATION ADDRESS

OUTPUT GROUP
TABLE 5.3–14

'NOP'	00	
'HALT'	76	
DISABLE INT '(DI)'	F3	
ENABLE INT '(EI)'	FB	
SET INT MODE 0 'IM0'	ED 46	8080A MODE
SET INT MODE 1 'IM1'	ED 56	CALL TO LOCATION 0038_H
SET INT MODE 2 'IM2'	ED 5E	INDIRECT CALL USING REGISTER I AND 8 BITS FROM INTERRUPTING DEVICE AS A POINTER.

MISCELLANEOUS CPU CONTROL
TABLE 5.3–15

6.0 FLAGS

Each of the two Z-80 CPU Flag registers contains six bits of information which are set or reset by various CPU operations. Four of these bits are testable; that is, they are used as conditions for jump, call or return instructions. For example a jump may be desired only if a specific bit in the flag register is set. The four testable flag bits are:

1) Carry Flag (C) – This flag is the carry from the highest order bit of the accumulator. For example, the carry flag will be set during an add instruction where a carry from the highest bit of the accumulator is generated. This flag is also set if a borrow is generated during a subtraction instruction. The shift and rotate instructions also affect this bit.

2) Zero Flag (Z) – This flag is set if the result of the operation loaded a zero into the accumulator. Otherwise it is reset.

3) Sign Flag (S) – This flag is intended to be used with signed numbers and it is set if the result of the operation was negative. Since bit 7 (MSB) represents the sign of the number (A negative number has a 1 in bit 7), this flag stores the state of bit 7 in the accumulator.

4) Parity/Overflow Flag (P/V) – This dual purpose flag indicates the parity of the result in the accumulator when logical operations are performed (such as AND A, B) and it represents overflow when signed two's complement arithmetic operations are performed. The Z-80 overflow flag indicates that the two's complement number in the accumulator is in error since it has exceeded the maximum possible (+127) or is less than the minimum possible (-128) number than can be represented in two's complement notation. For example consider adding:

```
+120 =    0111 1000
+105 =    0110 1001
------------------
  C  =  0  1110 0001  = -95 (wrong) Overflow has occured
```

Here the result is incorrect. Overflow has occurred and yet there is no carry to indicate an error. For this case the overflow flag would be set. Also consider the addition of two negative numbers:

```
 -5 =    1111 1011
-16 =    1111 0000
------------------
  C  =  1  1110 1011  = -21 correct
```

Notice that the answer is correct but the carry is set so that this flag can not be used as an overflow indicator. In this case the overflow would not be set.

For logical operations (AND, OR, XOR) this flag is set if the parity of the result is even and it is reset if it is odd.

There are also two non-testable bits in the flag register. Both of these are used for BCD arithmetic. They are:

1) Half carry (H) – This is the BCD carry or borrow result from the least significant four bits of operation. When using the DAA (Decimal Adjust Instruction) this flag is used to correct the result of a previous packed decimal add or subtract.

2) Subtract Flag (N) – Since the algorithm for correcting BCD operations is different for addition or subtraction, this flag is used to specify what type of instruction was executed last so that the DAA operation will be correct for either addition or subtraction.

The Flag register can be accessed by the programmer and its format is as follows:

S	Z	X	H	X	P/V	N	C

X means flag is indeterminate.

Table 6.0-1 lists how each flag bit is affected by various CPU instructions. In this table a '•' indicates that the instruction does not change the flag, an 'X' means that the flag goes to an indeterminate state, a '0' means that it is reset, a '1' means that it is set and the symbol '‡' indicates that it is set or reset according to the previous discussion. Note that any instruction not appearing in this table does not affect any of the flags.

Table 6.0-1 includes a few special cases that must be described for clarity. Notice that the block search instruction sets the Z flag if the last compare operation indicated a match between the source and the accumulator data. Also, the parity flag is set if the byte counter (register pair BC) is not equal to zero. This same use of the parity flag is made with the block move instructions. Another special case is during block input or output instructions, here the Z flag is used to indicate the state of register B which is used as a byte counter. Notice that when the I/O block transfer is complete, the zero flag will be reset to a zero (i.e. B=0) while in the case of a block move command the parity flag is reset when the operation is complete. A final case is when the refresh or I register is loaded into the accumulator, the interrupt enable flip flop is loaded into the parity flag so that the complete state of the CPU can be saved at any time.

Instruction	C	Z	P/V	S	N	H	Comments
ADD A, s ADC A,s	‡	‡	V	‡	0	‡	8-bit add or add with carry
SUB s, SBC A, s CP s, NEG	‡	‡	V	‡	1	‡	8 bit subtract, subtract with carry, compare and negate accumulator
AND s	0	‡	P	‡	0	1	Logical operations
OR s; XOR s	0	‡	P	‡	0	0	And set's different flags
INC s	•	‡	V	‡	0	‡	8-bit increment
DEC m	•	‡	V	‡	1	‡	8-bit decrement
ADD DD, ss	‡	•	•	•	0	X	16 bit add
ADC HL, ss	‡	‡	V	‡	0	X	16-bit add with carry
SBC HL, ss	‡	‡	V	‡	1	X	16-bit subtract with carry
RLA; RLCA, RRA, RRCA	‡	•	•	•	0	0	Rotate accumulator
RL m, RLC m; RR m, RRC m SLA m, SRA m, SRL m	‡	‡	P	‡	0	0	Rotate and shift location m
RLD, RRD	•	‡	P	‡	0	0	Rotate digit left and right
DAA	‡	‡	P	‡	•	‡	Decimal adjust accumulator
CPL	•	•	•	•	1	1	Complement accumulator
SCF	1	•	•	•	0	0	Set carry
CCF	‡	•	•	•	0	X	Complement carry
IN r, (C)	•	‡	P	‡	0	0	Input register indirect
INI; IND; OUTI; OUTD	•	‡	X	X	1	X	Block input and output
INIR; INDR; OTIR; OTDR	•	1	X	X	1	X	Z = 0 if B ≠ 0 otherwise Z = 1
LDI, LDD	•	X	‡	X	0	0	Block transfer instructions
LDIR, LDDR	•	X	0	X	0	0	P/V = 1 if BC ≠ 0, otherwise P/V = 0
CPI, CPIR, CPD, CPDR	•	‡	‡	‡	1	X	Block search instructions Z = 1 if A = (HL), otherwise Z = 0 P/V = 1 if BC ≠ 0, otherwise P/V = 0
LD A, I; LD A, R	•	‡	IFF	‡	0	0	The content of the interrupt enable flip-flop (IFF) is copied into the P/V flag
BIT b, s	•	‡	X	X	0	1	The state of bit b of location s is copied into the Z flag
NEG	‡	‡	V	‡	1	‡	Negate accumulator

The following notation is used in this table:

Symbol	Operation
C	Carry/link flag. C=1 if the operation produced a carry from the MSB of the operand or result.
Z	Zero flag. Z=1 if the result of the operation is zero.
S	Sign flag. S=1 if the MSB of the result is one.
P/V	Parity or overflow flag. Parity (P) and overflow (V) share the same flag. Logical operations affect this flag with the parity of the result, while arithmetic operations affect this flag with the overflow of the result. If P/V holds parity, P/V=1 if the result of the operation is even, P/V=0 if result is odd. If P/V holds overflow, P/V=1 if the result of the operation produced an overflow.
H	Half-carry flag. H=1 if the add or subtract operation produced a carry into or borrow from into bit 4 of the accumulator.
N	Add/Subtract flag. N=1 if the previous operation was a subtract.
	H and N flags are used in conjunction with the decimal adjust instruction (DAA) to properly correct the result into packed BCD format following addition or subtraction using operands with packed BCD format.
‡	The flag is affected according to the result of the operation.
•	The flag is unchanged by the operation.
0	The flag is reset by the operation.
1	The flag is set by the operation.
X	The flag is a "don't care."
V	P/V flag affected according to the overflow result of the operation.
P	P/V flag affected according to the parity result of the operation.
r	Any one of the CPU registers A, B, C, D, E, H, L.
s	Any 8 bit location for all the addressing modes allowed for the particular instruction.
ss	Any 16 bit location for all the addressing modes allowed for that instruction.
u	Any one of the two index registers IX or IY.
R	Refresh counter.
n	8-bit value in range <0, 255>
nn	16-bit value in range <0, 65535>
m	Any 8-bit location for all the addressing modes allowed for the particular instruction.

SUMMARY OF FLAG OPERATION
TABLE 6.0-1

7.0 SUMMARY OF OP CODES AND EXECUTION TIMES

The following section gives a summary of the Z-80 instructions set. The instructions are logically arranged into groups as shown on tables 7.0-1 through 7.0-11. Each table shows the assembly language mnemonic OP code, the actual OP code, the symbolic operation, the content of the flag register following the execution of each instruction, the number of bytes required for each instruction as well as the number of memory cycles and the total number of T states (external clock periods) required for the fetching and execution of each instruction. Care has been taken to make each table self-explanatory without requiring any cross reference with the text of other tables.

Mnemonic	Symbolic Operation	Flags C	Z	P/V	S	N	H	OP-code 76	543	210	No. of Bytes	No. of M Cycles	No. of T Cycles	Comments	
LD r, r'	r ← r'	•	•	•	•	•	•	01	r	r'	1	1	4	r, r'	Reg.
LD r, n	r ← n	•	•	•	•	•	•	00	r	110	2	2	7	000	B
								←	n	→				001	C
LD r, (HL)	r ← (HL)	•	•	•	•	•	•	01	r	110	1	2	7	010	D
LD r, (IX+d)	r ← (IX+d)	•	•	•	•	•	•	11	011	101	3	5	19	011	E
								01	r	110				100	H
								←	d	→				101	L
LD r, (IY+d)	r ← (IY+d)	•	•	•	•	•	•	11	111	101	3	5	19	111	A
								01	r	110					
								←	d	→					
LD (HL), r	(HL) ← r	•	•	•	•	•	•	01	110	r	1	2	7		
LD (IX+d), r	(IX+d) ← r	•	•	•	•	•	•	11	011	101	3	5	19		
								01	110	r					
								←	d	→					
LD (IY+d), r	(IY+d) ← r	•	•	•	•	•	•	11	111	101	3	5	19		
								01	110	r					
								←	d	→					
LD (HL), n	(HL) ← n	•	•	•	•	•	•	00	110	110	2	3	10		
								←	n	→					
LD (IX+d), n	(IX+d) ← n	•	•	•	•	•	•	11	011	101	4	5	19		
								00	110	110					
								←	d	→					
								←	n	→					
LD (IY+d), n	(IY+d) ← n	•	•	•	•	•	•	11	111	101	4	5	19		
								00	110	110					
								←	d	→					
			(L)					←	n	→					
LD A, (BC)	A ← (BC)	•	•	•	•	•	•	00	001	010	1	2	7		
LD A, (DE)	A ← (DE)	•	•	•	•	•	•	00	011	010	1	2	7		
LD A, (nn)	A ← (nn)	•	•	•	•	•	•	00	111	010	3	4	13		
								←	n	→					
								←	n	→					
LD (BC), A	(BC) ← A	•	•	•	•	•	•	00	000	010	1	2	7		
LD (DE), A	(DE) ← A	•	•	•	•	•	•	00	010	010	1	2	7		
LD (nn), A	(nn) ← A	•	•	•	•	•	•	00	110	010	3	4	13		
								←	n	→					
								←	n	→					
LD A, I	A ← I	•	↕	IFF	↕	0	0	11	101	101	2	2	9		
								01	010	111					
LD A, R	A ← R	•	↕	IFF	↕	0	0	11	101	101	2	2	9		
								01	011	111					
LD I, A	I ← A	•	•	•	•	•	•	11	101	101	2	2	9		
								01	000	111					
LD R, A	R ← A	•	•	•	•	•	•	11	101	101	2	2	9		
								01	001	111					

Notes: r, r' means any of the registers A, B, C, D, E, H, L

IFF the content of the interrupt enable flip-flop (IFF) is copied into the P/V flag

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown

↕ = flag is affected according to the result of the operation

8-BIT LOAD GROUP

TABLE 7.0-3

Mnemonic	Symbolic Operation	C	Z	P/V	S	N	H	Op-Code 76 543 210	No. of Bytes	No. of M Cycles	No. of T States	Comments
LD dd, nn	dd ← nn	•	•	•	•	•	•	00 dd0 001 ← n → ← n →	3	3	10	dd Pair 00 BC 01 DE 10 HL 11 SP
LD IX, nn	IX ← nn	•	•	•	•	•	•	11 011 101 00 100 001 ← n → ← n →	4	4	14	
LD IY, nn	IY ← nn	•	•	•	•	•	•	11 111 101 00 100 001 ← n → ← n →	4	4	14	
LD HL, (nn)	H ← (nn+1) L ← (nn)	•	•	•	•	•	•	00 101 010 ← n → ← n →	3	5	16	
LD dd, (nn)	dd_H ← (nn+1) dd_L ← (nn)	•	•	•	•	•	•	11 101 101 01 dd1 011 ← n → ← n →	4	6	20	
LD IX, (nn)	IX_H ← (nn+1) IX_L ← (nn)	•	•	•	•	•	•	11 011 101 00 101 010 ← n → ← n →	4	6	20	
LD IY, (nn)	IY_H ← (nn+1) IY_L ← (nn)	•	•	•	•	•	•	11 111 101 00 101 010 ← n → ← n →	4	6	20	
LD (nn), HL	(nn+1) ← H (nn) ← L	•	•	•	•	•	•	00 100 010 ← n → ← n →	3	5	16	
LD (nn), dd	(nn+1) ← dd_H (nn) ← dd_L	•	•	•	•	•	•	11 101 101 01 dd0 011 ← n → ← n →	4	6	20	
LD (nn), IX	(nn+1) ← IX_H (nn) ← IX_L	•	•	•	•	•	•	11 011 101 00 100 010 ← n → ← n →	4	6	20	
LD (nn), IY	(nn+1) ← IY_H (nn) ← IY_L	•	•	•	•	•	•	11 111 101 00 100 010 ← n → ← n →	4	6	20	
LD SP, HL	SP ← HL	•	•	•	•	•	•	11 111 001	1	1	6	
LD SP, IX	SP ← IX	•	•	•	•	•	•	11 011 101 11 111 001	2	2	10	
LD SP, IY	SP ← IY	•	•	•	•	•	•	11 111 101 11 111 001	2	2	10	
PUSH qq	(SP-2) ← qq_L (SP-1) ← qq_H	•	•	•	•	•	•	11 qq0 101	1	3	11	qq Pair 00 BC 01 DE 10 HL 11 AF
PUSH IX	(SP-2) ← IX_L (SP-1) ← IX_H	•	•	•	•	•	•	11 011 101 11 100 101	2	4	15	
PUSH IY	(SP-2) ← IY_L (SP-1) ← IY_H	•	•	•	•	•	•	11 111 101 11 100 101	2	4	15	
POP qq	qq_H ← (SP+1) qq_L ← (SP)	•	•	•	•	•	•	11 qq0 001	1	3	10	
POP IX	IX_H ← (SP+1) IX_L ← (SP)	•	•	•	•	•	•	11 011 101 11 100 001	2	4	14	
POP IY	IY_H ← (SP+1) IY_L ← (SP)	•	•	•	•	•	•	11 111 101 11 100 001	2	4	14	

Notes: dd is any of the register pairs BC, DE, HL, SP
qq is any of the register pairs AF, BC, DE, HL
$(PAIR)_H$, $(PAIR)_L$ refer to high order and low order eight bits of the register pair respectively.
E.g. BC_L = C, AF_H = A

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ flag is affected according to the result of the operation.

16 BIT LOAD GROUP

TABLE 7.0-2

Mnemonic	Symbolic Operation	Flags C	Z	P/V	S	N	H	Op-Code 76 543 210	No. of Bytes	No. of M Cycles	No. of T States	Comments
EX DE, HL	DE ↔ HL	•	•	•	•	•	•	11 101 011	1	1	4	
EX AF, AF'	AF ↔ AF'	•	•	•	•	•	•	00 001 000	1	1	4	
EXX	$\begin{pmatrix}BC\\DE\\HL\end{pmatrix}\leftrightarrow\begin{pmatrix}BC'\\DE'\\HL'\end{pmatrix}$	•	•	•	•	•	•	11 011 001	1	1	4	Register bank and auxiliary register bank exchange
EX (SP), HL	H ↔ (SP+1) L ↔ (SP)	•	•	•	•	•	•	11 100 011	1	5	19	
EX (SP), IX	$IX_H \leftrightarrow (SP+1)$ $IX_L \leftrightarrow (SP)$	•	•	•	•	•	•	11 011 101 11 100 011	2	6	23	
EX (SP), IY	$IY_H \leftrightarrow (SP+1)$ $IY_L \leftrightarrow (SP)$	•	•	•	•	•	•	11 111 101 11 100 011	2	6	23	
LDI	(DE) ← (HL) DE ← DE+1 HL ← HL+1 BC ← BC-1	•	•	①	•	0	0	11 101 101 10 100 000	2	4	16	Load (HL) into (DE), increment the pointers and decrement the byte counter (BC)
LDIR	(DE) ← (HL) DE ← DE+1 HL ← HL+1 BC ← BC-1 Repeat until BC = 0	•	•	0	•	0	0	11 101 101 10 110 000	2 2	5 4	21 16	If BC ≠ 0 If BC = 0
LDD	(DE) ← (HL) DE ← DE-1 HL ← HL-1 BC ← BC-1	•	•	①	•	0	0	11 101 101 10 101 000	2	4	16	
LDDR	(DE) ← (HL) DE ← DE-1 HL ← HL-1 BC ← BC-1 Repeat until BC = 0	•	•	0	•	0	0	11 101 101 10 111 000	2 2	5 4	21 16	If BC ≠ 0 If BC = 0
CPI	A - (HL) HL ← HL+1 BC ← BC-1	•	②	①	‡	1	‡	11 101 101 10 100 001	2	4	16	
CPIR	A - (HL) HL ← HL+1 BC ← BC-1 Repeat until A = (HL) or BC = 0	•	②	①	‡	1	‡	11 101 101 10 110 001	2 2	5 4	21 16	If BC ≠ 0 and A ≠ (HL) If BC = 0 or A = (HL)
CPD	A - (HL) HL ← HL-1 BC ← BC-1	•	②	①	‡	1	‡	11 101 101 10 101 001	2	4	16	
CPDR	A - (HL) HL ← HL-1 BC ← BC-1 Repeat until A = (HL) or BC = 0	•	②	①	‡	1	‡	11 101 101 10 111 001	2 2	5 4	21 16	If BC ≠ 0 and A ≠ (HL) If BC = 0 or A = (HL)

Notes: ① P/V flag is 0 if the result of BC-1 = 0, otherwise P/V = 1
② Z flag is 1 if A = (HL), otherwise Z = 0

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown.
‡ = flag is affected according to the result of the operation

EXCHANGE GROUP AND BLOCK TRANSFER AND SEARCH GROUP

TABLE 7.0 3

Mnemonic	Symbolic Operation	Flags C	Z	P/V	S	N	H	Op-Code 76 543 210	No. of Bytes	No. of M Cycles	No. of T States	Comments
ADD A,r	A ← A + r	↕	↕	V	↕	0	↕	10 000 r	1	1	4	r Reg 000 B 001 C 010 D 011 E 100 H 101 L 111 A
ADD A,n	A ← A + n	↕	↕	V	↕	0	↕	11 000 110 ← n →	2	2	7	
ADD A,(HL)	A ← A + (HL)	↕	↕	V	↕	0	↕	10 000 110	1	2	7	
ADD A,(IX+d)	A ← A + (IX+d)	↕	↕	V	↕	0	↕	11 011 101 10 000 110 ← d →	3	5	19	
ADD A,(IY+d)	A ← A+(IY+d)	↕	↕	V	↕	0	↕	11 111 101 10 000 110 ← d →	3	5	19	
ADC A,s	A ← A + s + CY	↕	↕	V	↕	0	↕	001				s is any of r, n, (HL), (IX+d), (IY+d) as shown for ADD instruction The indicated bits replace the 000 in the ADD set above
SUB s	A ← A − s	↕	↕	V	↕	1	↕	010				
SBC A,s	A ← A − s − CY	↕	↕	V	↕	1	↕	011				
AND s	A ← A ∧ s	0	↕	P	↕	0	1	100				
OR s	A ← A ∨ s	0	↕	P	↕	0	0	110				
XOR s	A ← A ⊕ s	0	↕	P	↕	0	0	101				
CP s	A − s	↕	↕	V	↕	1	↕	111				
INC r	r ← r + 1	•	↕	V	↕	0	↕	00 r 100	1	1	4	
INC (HL)	(HL) ← (HL)+1	•	↕	V	↕	0	↕	00 110 100	1	3	11	
INC (IX+d)	(IX+d) ← (IX+d)+1	•	↕	V	↕	0	↕	11 011 101 00 110 100 ← d →	3	6	23	
INC (IY+d)	(IY+d) ← (IY+d) + 1	•	↕	V	↕	0	↕	11 111 101 00 110 100 ← d →	3	6	23	
DEC m	m ← m−1	•	↕	V	↕	1	↕	101				m is any of r, (HL), (IX+d), (IY+d) as shown for INC Same format and states as INC Replace 100 with 101 in OP code

Notes: The V symbol in the P/V flag column indicates that the P/V flag contains the overflow of the result of the operation. Similarly the P symbol indicates parity. V = 1 means overflow, V = 0 means not overflow. P = 1 means parity of the result is even, P = 0 means parity of the result is odd.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown.
↕ = flag is affected according to the result of the operation.

8 BIT ARITHMETIC AND LOGICAL GROUP
TABLE 2.04

Mnemonic	Symbolic Operation	Flags						Op-Code	No. of Bytes	No. of M Cycles	No. of T States	Comments
		C	Z	P/V	S	N	H	76 543 210				
DAA	Converts acc. content into packed BCD following add or subtract with packed BCD operands	↕	↕	P	↕	•	↕	00 100 111	1	1	4	Decimal adjust accumulator
CPL	$A \leftarrow \overline{A}$	•	•	•	•	1	1	00 101 111	1	1	4	Complement accumulator (one's complement)
NEG	$A \leftarrow 0 - A$	↕	↕	V	↕	1	↕	11 101 101 01 000 100	2	2	8	Negate acc. (two's complement)
CCF	$CY \leftarrow \overline{CY}$	↕	•	•	•	0	X	00 111 111	1	1	4	Complement carry flag
SCF	$CY \leftarrow 1$	1	•	•	•	0	0	00 110 111	1	1	4	Set carry flag
NOP	No operation	•	•	•	•	•	•	00 000 000	1	1	4	
HALT	CPU halted	•	•	•	•	•	•	01 110 110	1	1	4	
DI	IFF ← 0	•	•	•	•	•	•	11 110 011	1	1	4	
EI	IFF ← 1	•	•	•	•	•	•	11 111 011	1	1	4	
IM 0	Set interrupt mode 0	•	•	•	•	•	•	11 101 101 01 000 110	2	2	8	
IM 1	Set interrupt mode 1	•	•	•	•	•	•	11 101 101 01 010 110	2	2	8	
IM2	Set interrupt mode 2	•	•	•	•	•	•	11 101 101 01 011 110	2	2	8	

Notes: IFF indicates the interrupt enable flip-flop
CY indicates the carry flip-flop.

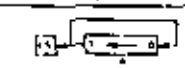
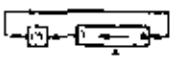
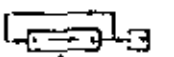
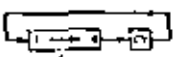
Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ = flag is affected according to the result of the operation.

Mnemonic	Symbolic Operation	Flags C	Z	P/V	S	N	H	Op-Code 76 543 210	No. of Bytes	No. of M Cycles	No. of T States	Comments
ADD HL, ss	HL ← HL + ss	‡	*	*	*	0	X	00 ss1 001	1	3	11	ss Reg. 00 BC 01 DE 10 HL 11 SP
ADC HL, ss	HL ← HL + ss + CY	‡	‡	V	‡	0	X	11 101 101 01 ss1 010	2	4	15	
SBC HL, ss	HL ← HL - ss - CY	‡	‡	V	‡	1	X	11 101 101 01 ss0 010	2	4	15	
ADD IX, pp	IX ← IX + pp	‡	*	*	*	0	X	11 011 101 00 pp1 001	2	4	15	pp Reg. 00 BC 01 DE 10 IX 11 SP
ADD IY, rr	IY ← IY + rr	‡	*	*	*	0	X	11 111 101 00 rr1 001	2	4	15	rr Reg. 00 BC 01 DE 10 IY 11 SP
INC ss	ss ← ss + 1	*	*	*	*	*	*	00 ss0 011	1	1	6	
INC IX	IX ← IX + 1	*	*	*	*	*	*	11 011 101 00 100 011	2	2	10	
INC IY	IY ← IY + 1	*	*	*	*	*	*	11 111 101 00 100 011	2	2	10	
DEC ss	ss ← ss - 1	*	*	*	*	*	*	00 ss1 011	1	1	6	
DEC IX	IX ← IX - 1	*	*	*	*	*	*	11 011 101 00 101 011	2	2	10	
DEC IY	IY ← IY - 1	*	*	*	*	*	*	11 111 101 00 101 011	2	2	10	

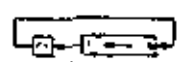
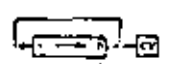
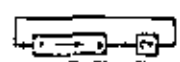
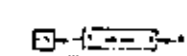
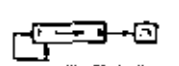
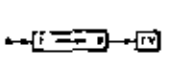
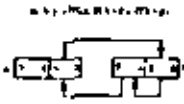
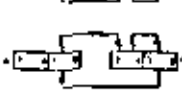
Notes: ss is any of the register pairs BC, DE, HL, SP
pp is any of the register pairs BC, DE, IX, SP
rr is any of the register pairs BC, DE, IY, SP.

Flag Notation: * = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown.
‡ = flag is affected according to the result of the operation.

16-BIT ARITHMETIC GROUP
TABLE 7.0-6

Mnemonic	Symbolic Operation	Flags C	Z	P/V	S	N	H	OpCode 76 543 210	No. of Bytes	No. of M Cycles	No. of T States	Comments
RLCA		‡	•	•	•	0	0	00 000 111	1	1	4	Rotate left circular accumulator
RLA		‡	•	•	•	0	0	00 010 111	1	1	4	Rotate left accumulator
RRCA		‡	•	•	•	0	0	00 001 111	1	1	4	Rotate right circular accumulator
RRA		‡	•	•	•	0	0	00 011 111	1	1	4	Rotate right accumulator
RLC r		‡	‡	P	‡	0	0	11 001 011 00 000 r	2	2	8	Rotate left circular register r
RLC (HL)		‡	‡	P	‡	0	0	11 001 011 00 000 110	2	4	15	
RLC (IX+d)		‡	‡	P	‡	0	0	11 011 101 11 001 011 ← d → 00 000 110	4	6	23	
RLC (IY+d)		‡	‡	P	‡	0	0	11 111 101 11 001 011 ← d → 00 000 110	4	6	23	

r	Reg.
000	B
001	C
010	D
011	E
100	H
101	L
111	A

Mnemonic	Symbolic Operation	Flags C	Z	P/V	S	N	H	OpCode 76 543 210	No. of Bytes	No. of M Cycles	No. of T States	Comments
RL m		‡	‡	P	‡	0	0	010				Instruction format and states are as shown for RLC m. To form new OP-code replace 000 of RLC m with shown code
RRC m		‡	‡	P	‡	0	0	001				
RR m		‡	‡	P	‡	0	0	011				
SLA m		‡	‡	P	‡	0	0	100				
SRA m		‡	‡	P	‡	0	0	101				
SRL m		‡	‡	P	‡	0	0	111				
RLD		•	‡	P	‡	0	0	11 101 101 01 101 111	2	5	18	Rotate digit left and right between the accumulator and location (HL). The content of the upper half of the accumulator is unaffected
RRD		•	‡	P	‡	0	0	11 101 101 01 100 111	2	5	18	

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
‡ = flag is affected according to the result of the operation.

ROTATE AND SHIFT GROUP
TABLE 7.0-7

Mnemonic	Symbolic Operation	Flags C	Z	P/V	S	N	H	Op-Code 76 543 210	No. of Bytes	No. of M Cycles	No. of T States	Comments
BIT b, r	$Z \leftarrow \overline{r_b}$	*	↕	X	X	0	1	11 001 011 01 b r	2	2	8	r: 000 B, 001 C, 010 D, 011 E, 100 H, 101 L, 111 A b: Bit Tested 000 0, 001 1, 010 2, 011 3, 100 4, 101 5, 110 6, 111 7
BIT b, (HL)	$Z \leftarrow \overline{(HL)_b}$	*	↕	X	X	0	1	11 001 011 01 b 110	2	3	12	
BIT b, (IX+d)	$Z \leftarrow \overline{(IX+d)_b}$	*	↕	X	X	0	1	11 011 101 11 001 011 ← d → 01 b 110	4	5	20	
BIT b, (IY+d)	$Z \leftarrow \overline{(IY+d)_b}$	*	↕	X	X	0	1	11 111 101 11 001 011 ← d → 01 b 110	4	5	20	
SET b, r	$r_b \leftarrow 1$	*	*	*	*	*	*	11 001 011 11 b r	2	2	8	
SET b, (HL)	$(HL)_b \leftarrow 1$	*	*	*	*	*	*	11 001 011 11 b 110	2	4	15	
SET b, (IX+d)	$(IX+d)_b \leftarrow 1$	*	*	*	*	*	*	11 011 101 11 001 011 ← d → 11 b 110	4	6	23	
SET b, (IY+d)	$(IY+d)_b \leftarrow 1$	*	*	*	*	*	*	11 111 101 11 001 011 ← d → 11 b 110	4	6	23	
RES b, m	$m_b \leftarrow 0$ m = r, (HL), (IX+d), (IY+d)							10				To form new OP-code replace 11 of SET b,m with 10. Flags and time states for SET instruction

Notes: The notation s_b indicates bit b (0 to 7) of location s.

Flag Notation: * = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown, ↕ = flag is affected according to the result of the operation.

BIT SET, RESET AND TEST GROUP
TABLE 7.0-8

Mnemonic	Symbolic Operation	Flags C	Z	P/V	S	N	H	Op-Code 76 543 210	No. of Bytes	No. of M Cycles	No. of T States	Comments
JP nn	PC ← nn	•	•	•	•	•	•	11 000 011 ← n → ← n →	3	3	10	
JP cc, nn	If condition cc is true PC ← nn, otherwise continue	•	•	•	•	•	•	11 cc 010 ← n → ← n →	3	3	10	cc: 000 NZ non zero; 001 Z zero; 010 NC non carry; 011 C carry; 100 PO parity odd; 101 PE parity even; 110 P sign positive; 111 M sign negative
JR e	PC ← PC + e	•	•	•	•	•	•	00 011 000 ← e-2 →	2	3	12	
JR C, e	If C = 0, continue	•	•	•	•	•	•	00 111 000 ← e-2 →	2	2	7	If condition not met
	If C = 1, PC ← PC + e								2	3	12	If condition is met
JR NC, e	If C = 1, continue	•	•	•	•	•	•	00 110 000 ← e-2 →	2	2	7	If condition not met
	If C = 0, PC ← PC + e								2	3	12	If condition is met
JR Z, e	If Z = 0 continue	•	•	•	•	•	•	00 101 000 ← e-2 →	2	2	7	If condition not met
	If Z = 1, PC ← PC + e								2	3	12	If condition is met
JR NZ, e	If Z = 1, continue	•	•	•	•	•	•	00 100 000 ← e-2 →	2	2	7	If condition not met
	If Z = 0, PC ← PC + e								2	3	12	If condition met
JP (HL)	PC ← HL	•	•	•	•	•	•	11 101 001	1	1	4	
JP (IX)	PC ← IX	•	•	•	•	•	•	11 011 101 11 101 001	2	2	8	
JP (IY)	PC ← IY	•	•	•	•	•	•	11 111 101 11 101 001	2	2	8	
DJNZ, e	B ← B-1 If B = 0, continue	•	•	•	•	•	•	00 010 000 ← e-2 →	2	2	8	If B = 0
	If B ≠ 0, PC ← PC + e								2	3	13	If B ≠ 0

Notes: e represents the extension in the relative addressing mode

e is a signed two's complement number in the range <-126, 129>

e-2 in the op-code provides an effective address of pc + e as PC is incremented by 2 prior to the addition of e.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown, ↕ = flag is affected according to the result of the operation

Mnemonic	Symbolic Operation	Flags C	Z	P/V	S	N	H	Op-Code 76	543	210	No. of Bytes	No. of M Cycles	No. of T States	Comments
CALL nn	(SP-1)←PC_H (SP-2)←PC_L PC←nn	•	•	•	•	•	•	11 ← ←	001 n n	101 → →	3	5	17	
CALL cc, nn	If condition cc is false continue, otherwise same as CALL nn	•	•	•	•	•	•	11 ← ←	cc n n	100 → →	3 3	3 5	10 17	If cc is false If cc is true
RET	PC_L←(SP) PC_H←(SP+1)	•	•	•	•	•	•	11	001	001	1	3	10	
RET cc	If condition cc is false continue, otherwise same as RET	•	•	•	•	•	•	11	cc	000	1 1	1 3	5 11	If cc is false If cc is true
RETI	Return from interrupt	•	•	•	•	•	•	11 01	101 001	101 101	2	4	14	
RETN	Return from non maskable interrupt	•	•	•	•	•	•	11 01	101 000	101 101	2	4	14	
RST p	(SP-1)←PC_H (SP-2)←PC_L PC_H←0 PC_L←P	•	•	•	•	•	•	11	t	111	1	3	11	

cc	Condition	
000	NZ	not zero
001	Z	zero
010	NC	non carry
011	C	carry
100	PO	parity odd
101	PE	parity even
110	P	sign positive
111	M	sign negative

t	P
000	00H
001	08H
010	10H
011	18H
100	20H
101	28H
110	30H
111	38H

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown
‡ = flag is affected according to the result of the operation.

Mnemonic	Symbolic Operation	Flags C	Z	P/V	S	N	H	Op-Code	No. of Bytes	No. of M Cycles	No. of T States	Comments
IN A, (n)	A ← (n)	•	•	•	•	•	•	11 011 011 ← n →	2	3	11	n to $A_0 - A_7$ Acc to $A_8 - A_{15}$
IN r, (C)	r ← (C) if r = 110 only the flags will be affected	•	↕	P	↕	0	•	11 101 101 01 r 000	2	3	12	C to $A_0 - A_7$ B to $A_8 - A_{15}$
INI	(HL) ← (C) B ← B - 1 HL ← HL + 1	X	↕ ①	X	X	1	X	11 101 101 10 100 010	2	4	16	C to $A_0 - A_7$ B to $A_8 - A_{15}$
INIR	(HL) ← (C) B ← B - 1 HL ← HL + 1 Repeat until B = 0	X	1	X	X	1	X	11 101 101 10 110 010	2 2	5 (If B ≠ 0) 4 (If B = 0)	21 16	C to $A_0 - A_7$ B to $A_8 - A_{15}$
IND	(HL) ← (C) B ← B - 1 HL ← HL - 1	X	↕ ①	X	X	1	X	11 101 101 10 101 010	2	4	16	C to $A_0 - A_7$ B to $A_8 - A_{15}$
INDR	(HL) ← (C) B ← B - 1 HL ← HL - 1 Repeat until B = 0	X	1	X	X	1	X	11 101 101 10 111 010	2 2	5 (If B ≠ 0) 4 (If B = 0)	21 16	C to $A_0 - A_7$ B to $A_8 - A_{15}$
OUT (n), A	(n) ← A	•	•	•	•	•	•	11 010 011 ← n →	2	3	11	n to $A_0 - A_7$ Acc to $A_8 - A_{15}$
OUT (C), r	(C) ← r	•	•	•	•	•	•	11 101 101 01 r 001	2	3	12	C to $A_0 - A_7$ B to $A_8 - A_{15}$
OUTI	(C) ← (HL) B ← B - 1 HL ← HL + 1	X	↕ ①	X	X	1	X	11 101 101 10 100 011	2	4	16	C to $A_0 - A_7$ B to $A_8 - A_{15}$
OTIR	(C) ← (HL) B ← B - 1 HL ← HL + 1 Repeat until B = 0	X	1	X	X	1	X	11 101 101 10 110 011	2 2	5 (If B ≠ 0) 4 (If B = 0)	21 16	C to $A_0 - A_7$ B to $A_8 - A_{15}$
OUTD	(C) ← (HL) B ← B - 1 HL ← HL - 1	X	↕ ①	X	X	1	X	11 101 101 10 101 011	2	4	16	C to $A_0 - A_7$ B to $A_8 - A_{15}$
OTDR	(C) ← (HL) B ← B - 1 HL ← HL - 1 Repeat until B = 0	X	1	X	X	1	X	11 101 101 10 111 011	2 2	5 (If B ≠ 0) 4 (If B = 0)	21 16	C to $A_0 - A_7$ B to $A_8 - A_{15}$

Notes ① If the result of B - 1 is zero the Z flag is set, otherwise it is reset.

Flag Notation: • = flag not affected, 0 = flag reset, 1 = flag set, X = flag is unknown,
↕ = flag is affected according to the result of the operation.

8.0 INTERRUPT RESPONSE

The purpose of an interrupt is to allow peripheral devices to suspend CPU operation in an orderly manner and force the CPU to start a peripheral service routine. Usually this service routine is involved with the exchange of data, or status and control information, between the CPU and the peripheral. Once the service routine is completed, the CPU returns to the operation from which it was interrupted.

INTERRUPT ENABLE – DISABLE

The Z80 CPU has two interrupt inputs, a software maskable interrupt and a non maskable interrupt. The non maskable interrupt (NMI) can *not* be disabled by the programmer and it will be accepted whenever a peripheral device requests it. This interrupt is generally reserved for very important functions that must be serviced whenever they occur, such as an impending power failure. The maskable interrupt (INT) can be selectively enabled or disabled by the programmer. This allows the programmer to disable the interrupt during periods where his program has timing constraints that do not allow it to be interrupted. In the Z80 CPU there is an enable flip flop (called IFF) that is set or reset by the programmer using the Enable Interrupt (EI) and Disable Interrupt (DI) instructions. When the IFF is reset, an interrupt can not be accepted by the CPU.

Actually, for purposes that will be subsequently explained, there are two enable flip flops, called IFF_1 and IFF_2.

IFF_1	IFF_2
Actually disables interrupts from being accepted.	Temporary storage location for IFF_1.

The state of IFF_1 is used to actually inhibit interrupts while IFF_2 is used as a temporary storage location for IFF_1. The purpose of storing the IFF_1 will be subsequently explained.

A reset to the CPU will force both IFF_1 and IFF_2 to the reset state so that interrupts are disabled. They can then be enabled by an EI instruction at any time by the programmer. When an EI instruction is executed, any pending interrupt request will not be accepted until after the instruction following EI has been executed. This single instruction delay is necessary for cases when the following instruction is a return instruction and interrupts must not be allowed until the return has been completed. The EI instruction sets both IFF_1 and IFF_2 to the enable state. When an interrupt is accepted by the CPU, both IFF_1 and IFF_2 are automatically reset, inhibiting further interrupts until the programmer wishes to issue a new EI instruction. Note that for all of the previous cases, IFF_1 and IFF_2 are always equal.

The purpose of IFF_2 is to save the status of IFF_1 when a non maskable interrupt occurs. When a non maskable interrupt is accepted, IFF_1 is reset to prevent further interrupts until reenabled by the programmer. Thus, after a non maskable interrupt has been accepted, maskable interrupts are disabled but the previous state of IFF_1 has been saved so that the complete state of the CPU just prior to the non maskable interrupt can be restored at any time. When a Load Register A with Register I (LD A, I) instruction or a Load Register A with Register R (LD A, R) instruction is executed, the state of IFF_2 is copied into the parity flag where it can be tested or stored.

A second method of restoring the status of IFF_1 is thru the execution of a Return From Non Maskable Interrupt (RETN) instruction. Since this instruction indicates that the non maskable interrupt service routine is complete, the contents of IFF_2 are now copied back into IFF_1, so that the status of IFF_1 just prior to the acceptance of the non maskable interrupt will be restored automatically.

Figure 8.0-1 is a summary of the effect of different instructions on the two enable flip flops.

Action	IFF_1	IFF_2	
CPU Reset	0	0	
DI	0	0	
EI	1	1	
LD A, I	•	•	$IFF_2 \rightarrow$ Parity flag
LD A, R	•	•	$IFF_2 \rightarrow$ Parity flag
Accept NMI	0	•	
RETN	IFF_2	•	$IFF_2 \rightarrow IFF_1$

"•" indicates no change

FIGURE 8.0-1
INTERRUPT ENABLE/DISABLE FLIP FLOPS

CPU RESPONSE

Non Maskable

A nonmaskable interrupt will be accepted at all times by the CPU. When this occurs, the CPU ignores the next instruction that it fetches and instead does a restart to location 0066H. Thus, it behaves exactly as if it had received a restart instruction but, it is to a location that is not one of the 8 software restart locations. A restart is merely a call to a specific address in page 0 of memory.

Maskable

The CPU can be programmed to respond to the maskable interrupt in any one of three possible modes.

Mode 0

This mode is identical to the 8080A interrupt response mode. With this mode, the interrupting device can place any instruction on the data bus and the CPU will execute it. Thus, the interrupting device provides the next instruction to be executed instead of the memory. Often this will be a restart instruction since the interrupting device only need supply a single byte instruction. Alternatively, any other instruction such as a 3 byte call to any location in memory could be executed.

The number of clock cycles necessary to execute this instruction is 2 more than the normal number for the instruction. This occurs since the CPU automatically adds 2 wait states to an interrupt response cycle to allow sufficient time to implement an external daisy chain for priority control. Section 5.0 illustrates the detailed timing for an interrupt response. After the application of RESET the CPU will automatically enter interrupt Mode 0.

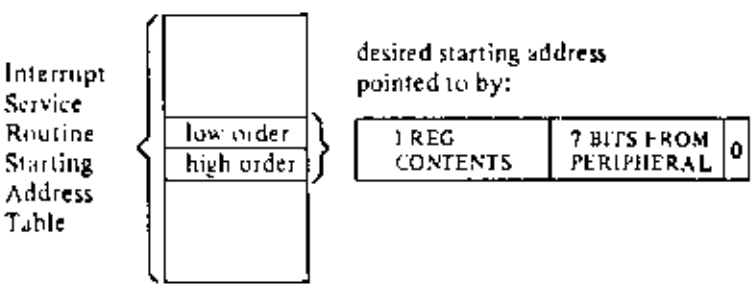
Mode 1

When this mode has been selected by the programmer, the CPU will respond to an interrupt by executing a restart to location 0038H. Thus the response is identical to that for a non maskable interrupt except that the call location is 0038H instead of 0066H. Another difference is that the number of cycles required to complete the restart instruction is 2 more than normal due to the two added wait states.

Mode 2

This mode is the most powerful interrupt response mode. With a single 8 bit byte from the user an indirect call can be made to any memory location.

With this mode the programmer maintains a table of 16 bit starting addresses for every interrupt service routine. This table may be located anywhere in memory. When an interrupt is accepted, a 16 bit pointer must be formed to obtain the desired interrupt service routine starting address from the table. The upper 8 bits of this pointer is formed from the contents of the I register. The I register must have been previously loaded with the desired value by the programmer, i.e. LD I, A. Note that a CPU reset clears the I register so that it is initialized to zero. The lower eight bits of the pointer must be supplied by the interrupting device. Actually, only 7 bits are required from the interrupting device as the least significant bit must be a zero. This is required since the pointer is used to get two adjacent bytes to form a complete 16 bit service routine starting address and the addresses must always start in even locations.



The first byte in the table is the least significant (low order) portion of the address. The programmer must obviously fill this table in with the desired addresses before any interrupts are to be accepted.

Note that this table can be changed at any time by the programmer (if it is stored in Read/Write Memory) to allow different peripherals to be serviced by different service routines.

Once the interrupting devices supplies the lower portion of the pointer, the CPU automatically pushes the program counter onto the stack, obtains the starting address from the table and does a jump to this address. This mode of response requires 19 clock periods to complete (7 to fetch the lower 8 bits from the interrupting device, 6 to save the program counter, and 6 to obtain the jump address.)

Note that the Z80 peripheral devices all include a daisy chain priority interrupt structure that automatically supplies the programmed vector to the CPU during interrupt acknowledge. Refer to the Z80-PIO, Z80-SIO and Z80-CTC manuals for details.

9.0 HARDWARE IMPLEMENTATION EXAMPLES

This chapter is intended to serve as a basic introduction to implementing systems with the Z80-CPU.

MINIMUM SYSTEM

Figure 9.0-1 is a diagram of a very simple Z-80 system. Any Z-80 system must include the following five elements:

1) Five volt power supply
2) Oscillator
3) Memory devices
4) I/O circuits
5) CPU

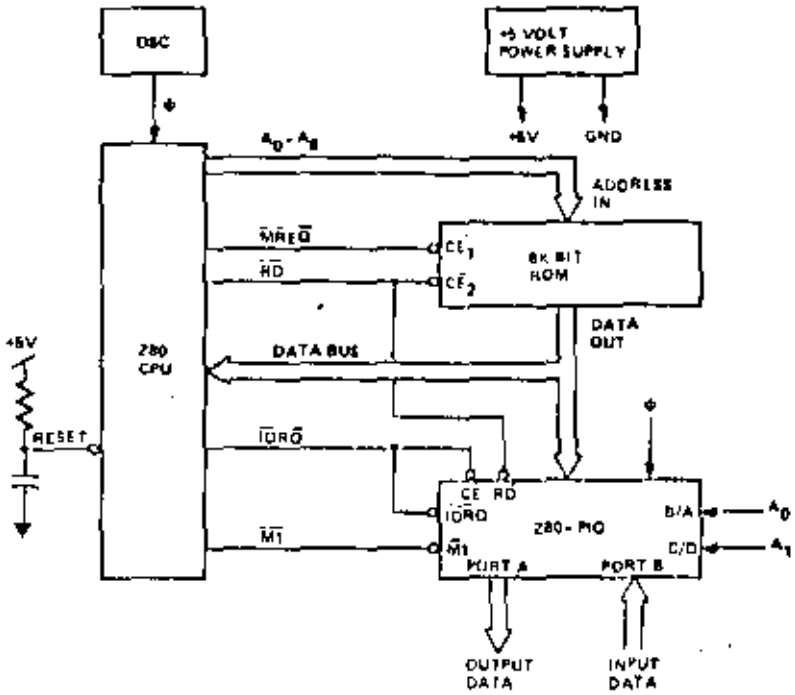


FIGURE 9.0-1
MINIMUM Z80 COMPUTER SYSTEM

Since the Z80-CPU only requires a single 5 volt supply, most small systems can be implemented using only this single supply.

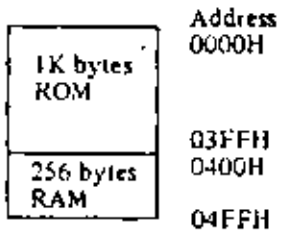
The oscillator can be very simple since the only requirement is that it be a 5 volt square wave. For systems not running at full speed, a simple RC oscillator can be used. When the CPU is operated near the highest possible frequency, a crystal oscillator is generally required because the system timing will not tolerate the drift or jitter that an RC network will generate. A crystal oscillator can be made from inverters and a few discrete components or monolithic circuits are widely available.

The external memory can be any mixture of standard RAM, ROM, or PROM. In this simple example we have shown a single 8K bit ROM (1K bytes) being utilized as the entire memory system. For this example we have assumed that the Z-80 internal register configuration contains sufficient Read/Write storage so that external RAM memory is not required.

Every computer system requires I/O circuits to allow it to interface to the "real world." In this simple example it is assumed that the output is an 8 bit control vector and the input is an 8 bit status word. The input data could be gated onto the data bus using any standard tri-state driver while the output data could be latched with any type of standard TTL latch. For this example we have used a Z80 PIO for the I/O circuit. This single circuit attaches to the data bus as shown and provides the required 16 bits of TTL compatible I/O. (Refer to the Z80 PIO manual for details on the operation of this circuit.) Notice in this example that with only three LSI circuits, a simple oscillator and a single 5 volt power supply, a powerful computer has been implemented.

ADDING RAM

Most computer systems require some amount of external Read/Write memory for data storage and to implement a "stack." Figure 9.0-2 illustrates how 256 bytes of static memory can be added to the previous example. In this example the memory space is assumed to be organized as follows:



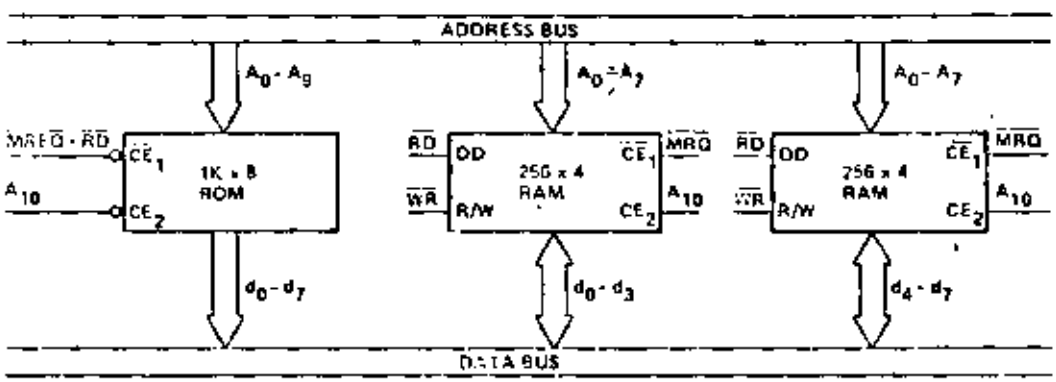


FIGURE 9.0-2
ROM & RAM IMPLEMENTATION EXAMPLE

In this diagram the address space is described in hexidecimal notation. For this example, address bit A_{10} separates the ROM space from the RAM space so that it can be used for the chip select function. For larger amounts of external ROM or RAM, a simple TTL decoder will be required to form the chip selects.

MEMORY SPEED CONTROL

For many applications, it may be desirable to use slow memories to reduce costs. The $\overline{WAIT}$ line on the CPU allows the Z-80 to operate with any speed memory. By referring back to section 4 you will notice that the memory access time requirements are most severe during the M1 cycle instruction fetch. All other memory accesses have an additional one half of a clock cycle to be completed. For this reason it may be desirable in some applications to add one wait state to the M1 cycle so that slower memories can be used. Figure 9.0-3 is an example of a simple circuit that will accomplish this task. This circuit can be changed to add a single wait state to any memory access as shown in Figure 9.0-4.

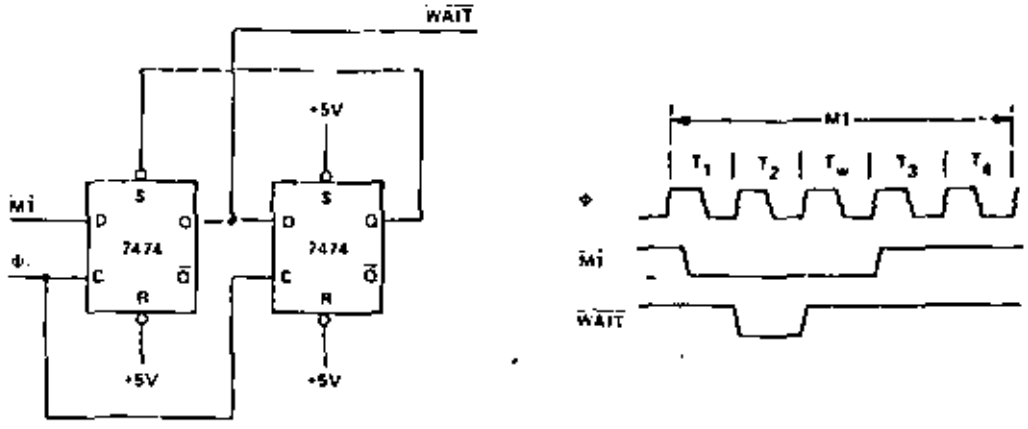


FIGURE 9.0-3
ADDING ONE WAIT STATE TO AN M1 CYCLE

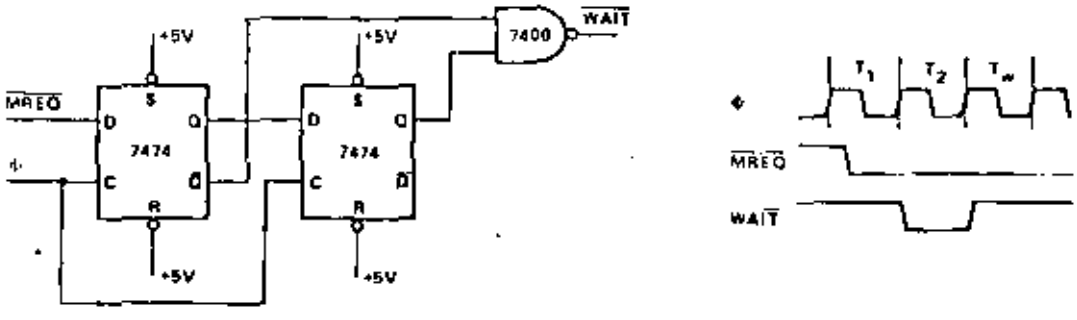


FIGURE 9.0-4
ADDING ONE WAIT STATE TO ANY MEMORY CYCLE

INTERFACING DYNAMIC MEMORIES

This section is intended only to serve as a brief introduction to interfacing dynamic memories. Each individual dynamic RAM has varying specifications that will require minor modifications to the description given here and no attempt will be made in this document to give details for any particular RAM. Separate application notes showing how the Z80-CPU can be interfaced to most popular dynamic RAM's are available from Zilog.

Figure 9.0-5 illustrates the logic necessary to interface 8K bytes of dynamic RAM using 18 pin 4K dynamic memories. This figure assumes that the RAM's are the only memory in the system so that A_{12} is used to select between the two pages of memory. During refresh time, all memories in the system must be read. The CPU provides the proper refresh address on lines A_0 through A_6. To add additional memory to the system it is necessary to only replace the two gates that operate on A_{12} with a decoder that operates on all required address bits. For larger systems, buffering for the address and data bus is also generally required.

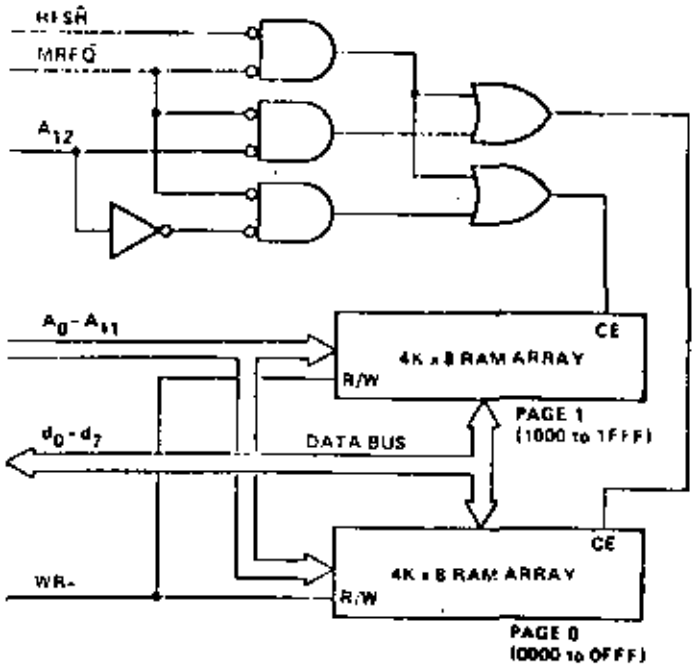


FIGURE 9.0-5
INTERFACING DYNAMIC RAMS

10.0 SOFTWARE IMPLEMENTATION EXAMPLES

10.1 METHODS OF SOFTWARE IMPLEMENTATION

Several different approaches are possible in developing software for the Z-80 (Figure 10.1). First of all, Assembly Language or PL/Z may be used as the source language. These languages may then be translated into machine language on a commercial time sharing facility using a cross-assembler or cross-compiler or, in the case of assembly language, the translation can be accomplished on a Z-80 Development System using a resident assembler. Finally, the resulting machine code can be debugged either on a time-sharing facility using a Z-80 simulator or on a Z-80 Development System which uses a Z80-CPU directly.

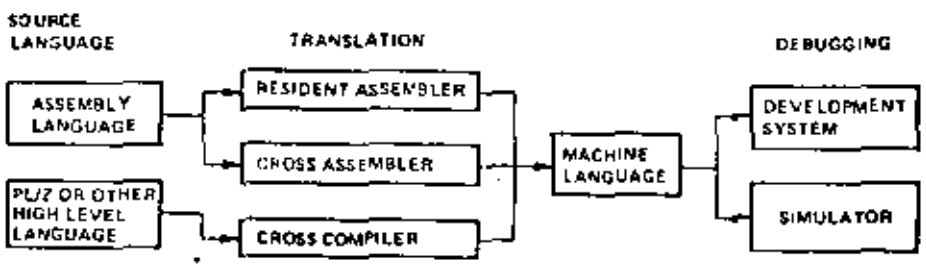


FIGURE 10.1

In selecting a source language, the primary factors to be considered are clarity and ease of programming vs. code efficiency. A high level language such as PL/Z with its machine independent constructs is typically better for formulating and maintaining algorithms, but the resulting machine code is usually somewhat less efficient than what can be written directly in assembly language. These tradeoffs can often be balanced by combining PL/Z and assembly language routines, identifying those portions of a task which must be optimized and writing them as assembly language subroutines.

Deciding whether to use a resident or cross assembler is a matter of availability and short-term vs. long-term expense. While the initial expenditure for a development system is higher than that for a time-sharing terminal, the cost of an individual assembly using a resident assembler is negligible while the same operation on a time-sharing system is relatively expensive and in a short time this cost can equal the total cost of a development system.

Debugging on a development system vs. a simulator is also a matter of availability and expense combined with operational fidelity and flexibility. As with the assembly process, debugging is less expensive on a development system than on a simulator available through time-sharing. In addition, the fidelity of the operating environment is preserved through real-time execution on a Z80-CPU and by connecting the I/O and memory components which will actually be used in the production system. The only advantage to the use of a simulator is the range of criteria which may be selected for such debugging procedures as tracing and setting breakpoints. This flexibility exists because a software simulation can achieve any degree of complexity in its interpretation of machine instructions while development system procedures have hardware limitations such as the capacity of the real-time storage module, the number of breakpoint registers and the pin configuration of the CPU. Despite such hardware limitations, debugging on a development system is typically more productive than on a simulator because of the direct interaction that is possible between the programmer and the authentic execution of his program.

10.2 SOFTWARE FEATURES OFFERED BY THE Z80-CPU

The Z-80 instruction set provides the user with a large and flexible repetoire of operations with which to formulate control of the Z80-CPU.

The primary, auxiliary and index registers can be used to hold the arguments of arithmetic and logical operations, or to form memory addresses, or as fast-access storage for frequently used data.

Information can be moved directly from register to register; from memory to memory; from memory to registers; or from registers to memory. In addition, register contents and register/memory contents can be exchanged without using temporary storage. In particular, the contents of primary and auxilary registers can be completely exchanged by executing only two instructions, EX and EXX. This register exchange procedure can be used to separate the set of working registers between different logical procedures or to expand the set of available registers in a single procedure.

Storage and retrieval of data between pairs of registers and memory can be controlled on a last-in first-out basis through PUSH and POP instructions which utilize a special stack pointer register, SP. This stack register is available both to manipulate data and to automatically store and retrieve addresses for subroutine linkage. When a subroutine is called, for example, the address following the CALL instruction is placed on the top of the push-down stack pointed to by SP. When a subroutine returns to the calling routine, the address on the top of the stack is used to set the program counter for the address of the next instruction. The stack pointer is adjusted automatically to reflect the current "top" stack position during PUSH, POP, CALL and RET instructions. This stack mechanism allows pushdown data stacks and subroutine calls to be nested to any practical depth because the stack area can potentially be as large as memory space.

The sequence of instruction execution can be controlled by six different flags (carry, zero, sign, parity/overflow, add-subtract, half-carry) which reflect the results of arithmetic, logical, shift and compare instructions. After the execution of an instruction which sets a flag, that flag can be used to control a conditional jump or return instruction. These instructions provide logical control following the manipulation of single bit, eight-bit byte (or) sixteen-bit data quantities.

A full set of logical operations, including AND, OR, XOR (exclusive - OR), CPL (NOR) and NEG (two's complement) are available for Boolean operations between the accumulator and 1) all other eight-bit registers, 2) memory locations or 3) immediate operands.

In addition, a full set of arithmetic and logical shifts in both directions are available which operate on the contents of all eight-bit primary registers or directly on any memory location. The carry flag can be included or simply set by these shift instructions to provide both the testing of shift results and to link register/register or register/memory shift operations.

10.3 EXAMPLES OF USE OF SPECIAL Z80 INSTRUCTIONS

A. Let us assume that a string of data in memory starting at location "DATA" is to be moved into another area of memory starting at location "BUFFER" and that the string length is 737 bytes. This operation can be accomplished as follows:

```
LD      HL , DATA       ;START ADDRESS OF DATA STRING
LD      DE , BUFFER     ;START ADDRESS OF TARGET BUFFER
LD      BC , 737        ;LENGTH OF DATA STRING
LDIR                    ;MOVE STRING - TRANSFER MEMORY POINTED TO
                        ;BY HL INTO MEMORY LOCATION POINTED TO BY DE
                        ;INCREMENT HL AND DE, DECREMENT BC
                        ;PROCESS UNTIL BC = 0.
```

11 bytes are required for this operation and each byte of data is moved in 21 clock cycles.

B. Let's assume that a string in memory starting at location "DATA" is to be moved into another area of memory starting at location "BUFFER" until an ASCII $ character (used as string delimiter) is found. Let's also assume that the maximum string length is 132 characters. The operation can be performed as follows:

```
        LD      HL , DATA       ;STARTING ADDRESS OF DATA STRING
        LD      DE , BUFFER     ;STARTING ADDRESS OF TARGET BUFFER
        LD      BC , 132        ;MAXIMUM STRING LENGTH
        LD      A , '$'         ;STRING DELIMITER CODE
LOOP:   CP      (HL)            ;COMPARE MEMORY CONTENTS WITH DELIMITER
        JR      Z , END - $     ;GO TO END IF CHARACTERS EQUAL
        LDI                     ;MOVE CHARACTER (HL) to (DE)
                                ;INCREMENT HL AND DE, DECREMENT BC
        JP      PE , LOOP       ;GO TO "LOOP" IF MORE CHARACTERS
END:                            ;OTHERWISE, FALL THROUGH
                                ;NOTE: P/V FLAG IS USED
                                ;TO INDICATE THAT REGISTER BC WAS
                                ;DECREMENTED TO ZERO.
```

19 bytes are required for this operation.

C. Let us assume that a 16-digit decimal number represented in packed BCD format (two BCD digits/byte) has to be shifted as shown in the Figure 10.2 in order to mechanize BCD multiplication or division. The operation can be accomplished as follows:

```
        LD      HL , DATA       ;ADDRESS OF FIRST BYTE
        LD      B , COUNT       ;SHIFT COUNT
        XOR     A               ;CLEAR ACCUMULATOR
ROTAT:  RLD                     ;ROTATE LEFT LOW ORDER DIGIT IN ACC
                                ;WITH DIGITS IN (HL)
        INC     HL              ;ADVANCE MEMORY POINTER
        DJNZ    ROTAT - $       ;DECREMENT B AND GO TO ROTAT IF
                                ;B IS NOT ZERO, OTHERWISE FALL THROUGH
```

11 bytes are required for this operation.

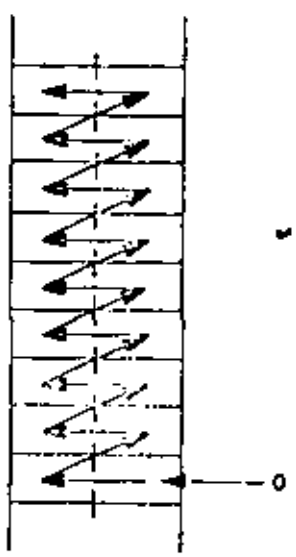


FIGURE 10.2

D. Let us assume that one number is to be subtracted from another and a) that they are both in packed BCD format, b) that they are of equal but varying length, and c) that the result is to be stored in the location of the minuend. The operation can be accomplished as follows:

```
        LD      HL , ARG1       ; ADDRESS OF MINUEND
        LD      DE , ARG2       ; ADDRESS OF SUBTRAHEND
        LD      B , LENGTH      ; LENGTH OF TWO ARGUMENTS
        AND     A               ; CLEAR CARRY FLAG
SUBDEC: LD      A , (DE)        ; SUBTRAHEND TO ACC
        SBC     A , (HL)        ; SUBTRACT (HL) FROM ACC
        DAA                     ; ADJUST RESULT TO DECIMAL CODED VALUE
        LD      (HL) , A        ; STORE RESULT
        INC     HL              ; ADVANCE MEMORY POINTERS
        INC     DE
        DJNZ    SUBDEC - $      ; DECREMENT B AND GO TO "SUBDEC" IF B
                                ; NOT ZERO, OTHERWISE FALL THROUGH
```

17 bytes are required for this operation.

10.4 EXAMPLES OF PROGRAMMING TASKS

A. The following program sorts an array of numbers each in the range (0,255) into ascending order using a standard exchange sorting algorithm.

```
01/22/76     11:14:37                BUBBLE LISTING                    PAGE 1
LOC    OBJ CODE    STMT  SOURCE STATEMENT

                     1   ; *** STANDARD EXCHANGE (BUBBLE) SORT ROUTINE ***
                     2   ;
                     3   ; AT ENTRY:  HL CONTAINS ADDRESS OF DATA
                     4                C CONTAINS NUMBER OF ELEMENTS TO BE SORTED
                     5                (1<C<256)
                     6   ;
                     7   ; AT EXIT: DATA SORTED IN ASCENDING ORDER
                     8   ;
                     9   ; USE OF REGISTERS
                    10   ;
                    11   ; REGISTER   CONTENTS
                    12   ;
                    13   ; A          TEMPORARY STORAGE FOR CALCULATIONS
                    14   ; B          COUNTER FOR DATA ARRAY
                    15   ; C          LENGTH OF DATA ARRAY
                    16   ; D          FIRST ELEMENT IN COMPARISON
                    17   ; E          SECOND ELEMENT IN COMPARISON
                    18   ; H          FLAG TO INDICATE EXCHANGE
                    19   ; L          UNUSED
                    20   ; IX         POINTER INTO DATA ARRAY
                    21   ; IY         UNUSED
                    22   ;
0000   222600       23   SORT:  LD     (DATA), HL     ;SAVE DATA ADDRESS
0003   CB84         24   LOOP:  RES    FLAG, H        ;INITIALIZE EXCHANGE FLAG
0005   41           25          LD     B, C           ;INITIALIZE LENGTH COUNTER
0006   05           26          DEC    B              ;ADJUST FOR TESTING
0007   DD2A2600     27          LD     IX, (DATA)     ;INITIALIZE ARRAY POINTER
000B   DD7E00       28   NEXT:  LD     A, (IX)        ;FIRST ELEMENT IN COMPARISON
000E   57           29          LD     D, A           ;TEMPORARY STORAGE FOR ELEMENT
000F   DD5E01       30          LD     E, (IX+1)      ;SECOND ELEMENT IN COMPARISON
0012   93           31          SUB    E              ;COMPARISON FIRST TO SECOND
0013   3008         32          JR     NC, NOEX-$     ;IF FIRST > SECOND, NO JUMP
0015   DD7300       33          LD     (IX), E        ;EXCHANGE ARRAY ELEMENTS
0018   DD7201       34          LD     (IX+1), D
001B   CBC4         35          SET    FLAG, H        ;RECORD EXCHANGE OCCURRED
001D   DD23         36   NOEX:  INC    IX             ;POINT TO NEXT DATA ELEMENT
001F   10EA         37          DJNZ   NEXT-$         ;COUNT NUMBER OF COMPARISONS
                    38                                ;REPEAT IF MORE DATA PAIRS
0021   CB44         39          BIT    FLAG, H        ;DETERMINE IF EXCHANGE OCCURRED
0023   20DE         40          JR     NZ, LOOP-$     ;CONTINUE IF DATA UNSORTED
0025   C9           41          RET                   ;OTHERWISE, EXIT
                    42   ;
0026                43   FLAG:  EQU    0              ;DESIGNATION OF FLAG BIT
0026                44   DATA:  DEFS   2              ;STORAGE FOR DATA ADDRESS
                    45          END
```

B. The following program multiplies two unsigned 16 bit integers and leaves the result in the HL register pair.

```
01/22/76     11:32:36          MULTIPLY LISTING                    PAGE 1
LOC   OBJ CODE   STMT  SOURCE STATEMENT

0000               1  MULT:;  UNSIGNED SIXTEEN BIT INTEGER MULTIPLY.
                   2  ;       ON ENTRANCE:  MULTIPLIER IN DE.
                   3  ;                     MULTIPLICAND IN HL.
                   4  ;
                   5  ;       ON EXIT:  RESULT IN HL.
                   6  ;
                   7  ;       REGISTER USES:
                   8  ;
                   9  ;
                  10  ;       H     HIGH ORDER PARTIAL RESULT
                  11  ;       L     LOW ORDER PARTIAL RESULT
                  12  ;       D     HIGH ORDER MULTIPLICAND
                  13  ;       E     LOW ORDER MULTIPLICAND
                  14  ;       B     COUNTER FOR NUMBER OF SHIFTS
                  15  ;       C     HIGH ORDER BITS OF MULTIPLIER
                  16  ;       A     LOW ORDER BITS OF MULTIPLIER
                  17  ;
0000  0610        18          LD    B,16;       NUMBER OF BITS- INITIALIZE
0002  4A          19          LD    C,D;        MOVE MULTIPLIER
0003  7B          20          LD    A,E;
0004  EB          21          EX    DE,HL;      MOVE MULTIPLICAND
0005  210000      22          LD    HL,0;       CLEAR PARTIAL RESULT
0008  CB39        23  MLOOP:  SRL   C;          SHIFT MULTIPLIER RIGHT
000A  1F          24          RRA                LEAST SIGNIFICANT BIT IS
                  25  ;                        IN CARRY.
000B  3001        26          JR    NC,NOADD-$; IF NO CARRY, SKIP THE ADD.
000D  19          27          ADD   HL,DE;      ELSE ADD MULTIPLICAND TO
                  28  ;                        PARTIAL RESULT.
000E  EB          29  NOADD:  EX    DE,HL;      SHIFT MULTIPLICAND LEFT
000F  29          30          ADD   HL,HL;      BY MULTIPLYING IT BY TWO.
0010  EB          31          EX    DE,HL;
0011  10F5        32          DJNZ  MLOOP-$;    REPEAT UNTIL NO MORE BITS.
0013  C9          33          RET;
                  34          END;
```

Absolute Maximum Ratings

Temperature Under Bias	Specified operating range
Storage Temperature	−65°C to +150°C
Voltage On Any Pin with Respect to Ground	−0.3V to +7V
Power Dissipation	1.5W

*Comment

Stresses above those listed under "Absolute Maximum Rating" may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or any other condition above those indicated in the operational sections of this specification is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.

Note: For Z80-CPU all AC and DC characteristics are the same for the military grade parts except I_{CC}.

I_{CC} = 200 mA

Z80-CPU D.C. Characteristics

T_A = 0°C to 70°C, V_{CC} = 5V ± 5% unless otherwise specified

Symbol	Parameter	Min	Typ	Max.	Unit	Test Condition
V_{ILC}	Clock Input Low Voltage	-0.3		0.45	V	
V_{IHC}	Clock Input High Voltage	V_{CC} -.6		V_{CC}+.3	V	
V_{IL}	Input Low Voltage	-0.3		0.8	V	
V_{IH}	Input High Voltage	2.0		V_{CC}	V	
V_{OL}	Output Low Voltage			0.4	V	I_{OL} = 1.8mA
V_{OH}	Output High Voltage	2.4			V	I_{OH} = −250μA
I_{CC}	Power Supply Current			150	mA	
I_{LI}	Input Leakage Current			10	μA	V_{IN}=0 to V_{CC}
I_{LOH}	Tri State Output Leakage Current in Float			10	μA	V_{OUT}=2.4 to V_{CC}
I_{LOL}	Tri State Output Leakage Current in Float			-10	μA	V_{OUT}=0.4V
I_{LD}	Data Bus Leakage Current in Input Mode			±10	μA	$0 \leq V_{IN} \leq V_{CC}$

Z80A-CPU D.C. Characteristics

T_A = 0°C to 70°C, V_{CC} = 5V ± 5% unless otherwise specified

Symbol	Parameter	Min	Typ.	Max	Unit	Test Condition
V_{ILC}	Clock Input Low Voltage	-0.3		0.45	V	
V_{IHC}	Clock Input High Voltage	V_{CC} -.6		V_{CC}+.3	V	
V_{IL}	Input Low Voltage	-0.3		0.8	V	
V_{IH}	Input High Voltage	2.0		V_{CC}	V	
V_{OL}	Output Low Voltage			0.4	V	I_{OL} = 1.8mA
V_{OH}	Output High Voltage	2.4			V	I_{OH} = −250μA
I_{CC}	Power Supply Current		90	200	mA	
I_{LI}	Input Leakage Current			10	μA	V_{IN}=0 to V_{CC}
I_{LOH}	Tri State Output Leakage Current in Float			10	μA	V_{OUT}=2.4 to V_{CC}
I_{LOL}	Tri State Output Leakage Current in Float			-10	μA	V_{OUT}=0.4V
I_{LD}	Data Bus Leakage Current in Input Mode			±10	μA	$0 \leq V_{IN} \leq V_{CC}$

Capacitance

T_A = 25°C, f = 1 MHz,
unmeasured pins returned to ground

Symbol	Parameter	Max.	Unit
C_Φ	Clock Capacitance	35	pF
C_{IN}	Input Capacitance	5	pF
C_{OUT}	Output Capacitance	10	pF

Z80-CPU Ordering Information

C = Ceramic
P = Plastic
S = Standard 5V ± 5% 0° to 70°C
E = Extended 5V ± 5% −40° to 85°C
M = Military 5V ± 10% −55° to 125°C

Capacitance

T_A = 25°C, f = 1 MHz,
unmeasured pins returned to ground

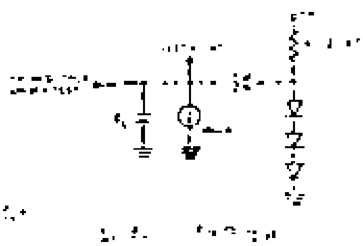
Symbol	Parameter	Max	Unit
C_Φ	Clock Capacitance	35	pF
C_{IN}	Input Capacitance	5	pF
C_{OUT}	Output Capacitance	10	pF

Z80A-CPU Ordering Information

C = Ceramic
P = Plastic
S = Standard 5V ± 5% 0° to 70°C

Signal	Symbol	Parameter	Min	Max	Unit	Test Condition
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	nsec	
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	nsec	C_L = [illegible]
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	nsec	C_L = [illegible]
[illegible]	[illegible]	[illegible]	0		nsec	
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	nsec	C_L = [illegible]
MREQ	[illegible]	MREQ Delay From Falling Edge of Clock, MREQ Low		90	nsec	C_L = 50pF
	[illegible]	MREQ Delay From Rising Edge of Clock, MREQ Low		[illegible]	nsec	
	[illegible]	MREQ Delay From Falling Edge of Clock, MREQ High		[illegible]	nsec	
	[illegible]	MREQ Delay From Rising Edge of Clock, MREQ High		[illegible]	nsec	
[illegible]	[illegible]	[illegible] Delay From Rising Edge of Clock, [illegible] Low		[illegible]	nsec	C_L = 50pF
	[illegible]	[illegible] Delay From Falling Edge of Clock, [illegible] Low		[illegible]	nsec	
	[illegible]	[illegible] Delay From Rising Edge of Clock, [illegible] High		[illegible]	nsec	
	[illegible]	[illegible] Delay From Falling Edge of Clock, [illegible] High		[illegible]	nsec	
[illegible]	[illegible]	WR Delay From Rising Edge of Clock, WR Low		[illegible]	nsec	C_L = 50pF
	[illegible]	WR Delay From Falling Edge of Clock, WR Low		[illegible]	nsec	
	[illegible]	WR Delay From Falling Edge of Clock, WR High		[illegible]	nsec	
	[illegible]	[illegible]	[illegible]		nsec	
M1	[illegible]	M1 Delay From Rising Edge of Clock, M1 Low		[illegible]	nsec	C_L = [illegible]
	[illegible]	M1 Delay From Rising Edge of Clock, M1 High		[illegible]	nsec	
RFSH	$t_{DL}(RF)$	RFSH Delay From Rising Edge of Clock, RFSH Low		[illegible]	nsec	C_L = 50pF
	$t_{DH}(RF)$	RFSH Delay From Rising Edge of Clock, RFSH High		150	nsec	
WAIT	[illegible]	WAIT Setup Time to Falling Edge of Clock	70		nsec	
[illegible]	[illegible]	[illegible]		[illegible]	nsec	C_L = [illegible]
INT	[illegible]	INT Setup Time to Rising Edge of Clock	[illegible]		nsec	
NMI	[illegible]	Pulse Width, NMI Low	[illegible]		nsec	
BUSRQ	[illegible]	BUSRQ Setup Time to Rising Edge of Clock	[illegible]		nsec	
BUSAK	[illegible]	BUSAK Delay From Rising Edge of Clock, BUSAK Low		[illegible]	nsec	C_L = [illegible]
	[illegible]	BUSAK Delay From Falling Edge of Clock, BUSAK High		[illegible]	nsec	
RESET	[illegible]	RESET Setup Time to Rising Edge of Clock	[illegible]		nsec	
	[illegible]	Delay to Float [illegible]		[illegible]	nsec	
	[illegible]	M1 Stable Prior to IORQ [illegible]	[illegible]		nsec	

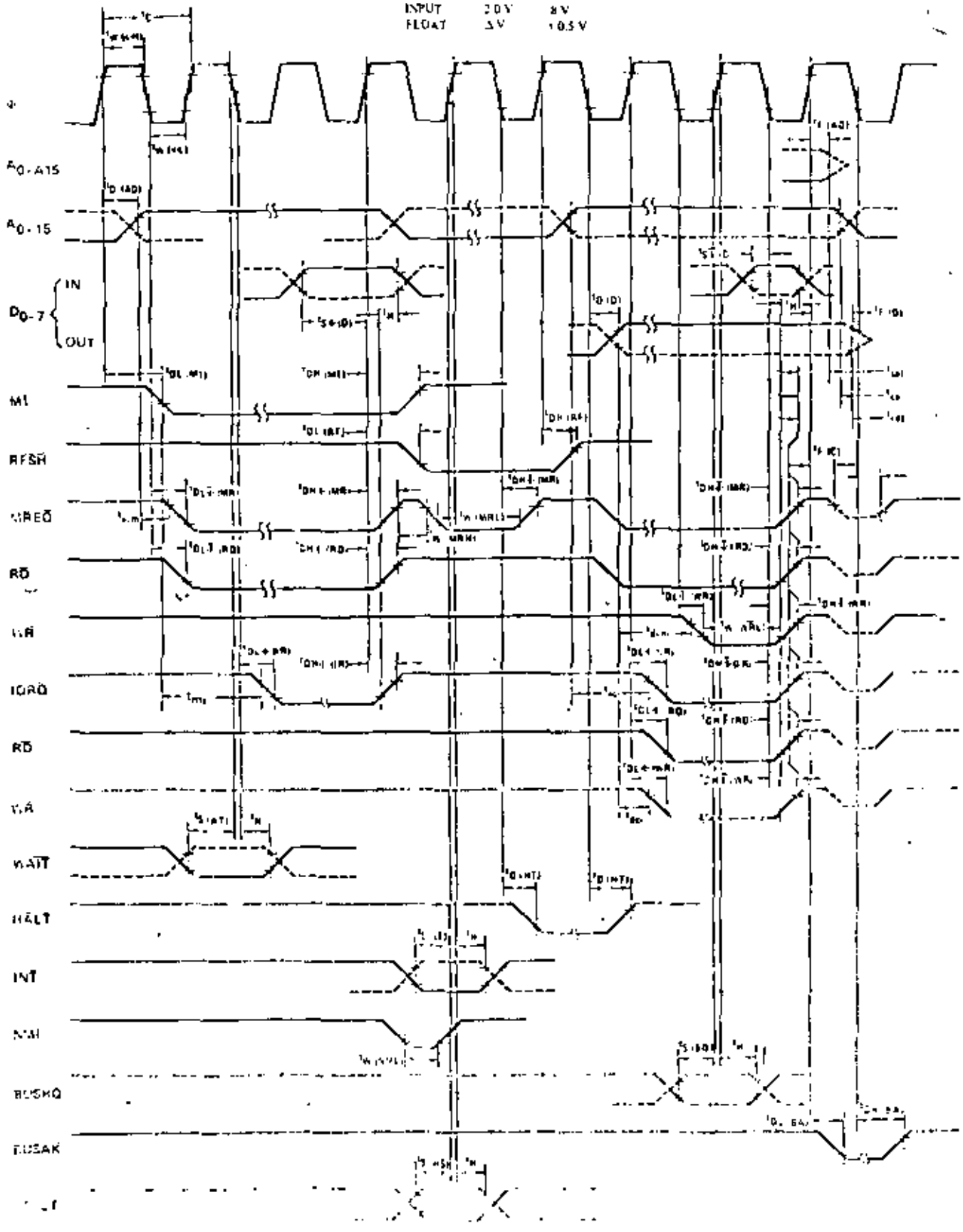
[illegible]



[illegible]

Timing measurements are made at the following voltages, unless otherwise specified:

	"1"	"0"
CLOCK	V_{CC} + 5V	.45V
OUTPUT	2.0V	.8V
INPUT	2.0V	.8V
FLOAT	.5V	+0.5V



T_A = 0°C to 70°C, V_{CC} = +5V ± 5%, Unless Otherwise Noted.

Signal	Symbol	Parameter	Min	Max	Unit	Test Condition	
Φ	t_c	Clock Period	25	[12]	nsec		[12] $t_c = t_{w(\Phi H)} + t_{w(\Phi L)} + t_r + t_f$
	$t_{w(\Phi H)}$	Clock Pulse Width, Clock High	110	[1]	nsec		
	$t_{w(\Phi L)}$	Clock Pulse Width, Clock Low	110	2000	nsec		
	t_r, t_f	Clock Rise and Fall Time		30	nsec		
A_{0-15}	$t_{a(AD)}$	Address Output Delay		110	nsec	C_L = 50pf	
	$t_{a(AM)}$	Delay to Float		90	nsec		
	t_{acm}	Address Stable Prior to $\overline{MREQ}$ (Memory Cycle)	[1]		nsec		[1] $t_{acm} = t_{w(\Phi H)} + t_f - 35$
	t_{aci}	Address Stable Prior to $\overline{IORQ}$, $\overline{RD}$ or $\overline{WR}$ (I/O Cycle)	[2]		nsec		[2] $t_{aci} = t_c - 20$
	t_{ca}	Address Stable From $\overline{RD}$, $\overline{WR}$, $\overline{IORQ}$ or $\overline{MREQ}$	[3]		nsec		[3] $t_{ca} = t_{w(\Phi L)} + t_r - 50$
	t_{caf}	Address Stable From $\overline{RD}$ or $\overline{WR}$ During Float	[4]		nsec		[4] $t_{caf} = t_{w(\Phi L)} + t_r - 45$
D_{0-7}	$t_{d(D)}$	Data Output Delay		150	nsec	C_L = 50pf	[5] $t_{dcm} = t_c - 170$
	$t_{f(D)}$	Delay to Float During Write Cycle		90	nsec		[6] $t_{dci} = t_{w(\Phi L)} + t_f - 170$
	$t_{s(D)}$	Data Setup Time to Rising Edge of Clock During M1 Cycle	75		nsec		[7] $t_{cdf} = t_{w(\Phi L)} + t_r - 70$
	$t_{s(D)}$	Data Setup Time to Falling Edge of Clock During M2 to M5	50		nsec		
	t_{dcm}	Data Stable Prior to $\overline{WR}$ (Memory Cycle)	[5]		nsec		
	t_{dci}	Data Stable Prior to $\overline{WR}$ (I/O Cycle)	[6]		nsec		
	t_{cdf}	Data Stable From $\overline{WR}$	[7]				
	t_H	Any Hold Time for Setup Time		0	nsec		
$\overline{MREQ}$	$t_{d(CR)}$	$\overline{MREQ}$ Delay From Falling Edge of Clock, $\overline{MREQ}$ Low		85	nsec	C_L = 50pf	[8] $t_w(\overline{MRL}) = t_c - 30$
	$t_{d(CR)}$	$\overline{MREQ}$ Delay From Rising Edge of Clock, $\overline{MREQ}$ High		85	nsec		[9] $t_w(\overline{MRH}) = t_{w(\Phi H)} + t_f - 20$
	$t_{d(CF)}$	$\overline{MREQ}$ Delay From Falling Edge of Clock, $\overline{MREQ}$ High		85	nsec		
	$t_{w(MRL)}$	Pulse Width, $\overline{MREQ}$ Low	[8]		nsec		
	$t_{w(MRH)}$	Pulse Width, $\overline{MREQ}$ High	[9]		nsec		
$\overline{IORQ}$	$t_{d(CR)}$	$\overline{IORQ}$ Delay From Rising Edge of Clock, $\overline{IORQ}$ Low		75	nsec	C_L = 50pF	
	$t_{d(CF)}$	$\overline{IORQ}$ Delay From Falling Edge of Clock, $\overline{IORQ}$ Low		85	nsec		
	$t_{d(CR)}$	$\overline{IORQ}$ Delay From Rising Edge of Clock, $\overline{IORQ}$ High		85	nsec		
	$t_{d(CF)}$	$\overline{IORQ}$ Delay From Falling Edge of Clock, $\overline{IORQ}$ High		85	nsec		
$\overline{RD}$	$t_{d(CR)}$	$\overline{RD}$ Delay From Rising Edge of Clock, $\overline{RD}$ Low		95	nsec	C_L = 50pF	
	$t_{d(CF)}$	$\overline{RD}$ Delay From Falling Edge of Clock, $\overline{RD}$ Low		95	nsec		
	$t_{d(CR)}$	$\overline{RD}$ Delay From Rising Edge of Clock, $\overline{RD}$ High		85	nsec		
	$t_{d(CF)}$	$\overline{RD}$ Delay From Falling Edge of Clock, $\overline{RD}$ High		85	nsec		
$\overline{WR}$	$t_{d(CR)}$	$\overline{WR}$ Delay From Rising Edge of Clock, $\overline{WR}$ Low		85	nsec	C_L = 50pf	
	$t_{d(CF)}$	$\overline{WR}$ Delay From Falling Edge of Clock, $\overline{WR}$ Low		95	nsec		
	$t_{d(CF)}$	$\overline{WR}$ Delay From Falling Edge of Clock, $\overline{WR}$ High		80	nsec		
	$t_{w(WR)}$	Pulse Width, $\overline{WR}$ Low	[10]		nsec		[10] $t_{w(\overline{WRL})} = t_c - 30$
$\overline{M1}$	$t_{d(M1)}$	$\overline{M1}$ Delay From Rising Edge of Clock, $\overline{M1}$ Low		100	nsec	C_L = 50pF	
	$t_{d(M1)}$	$\overline{M1}$ Delay From Rising Edge of Clock, $\overline{M1}$ High		130	nsec		
$\overline{RFSH}$	$t_{d(CRF)}$	$\overline{RFSH}$ Delay From Rising Edge of Clock, $\overline{RFSH}$ Low		130	nsec	C_L = 50pF	
	$t_{d(CF)}$	$\overline{RFSH}$ Delay From Rising Edge of Clock, $\overline{RFSH}$ High		120	nsec		
$\overline{WAIT}$	$t_{s(WT)}$	$\overline{WAIT}$ Setup Time to Falling Edge of Clock	70		nsec		
$\overline{HALT}$	$t_{d(HT)}$	$\overline{HALT}$ Delay Time From Falling Edge of Clock		300	nsec	C_L = 50pF	
$\overline{INT}$	$t_{s(IT)}$	$\overline{INT}$ Setup Time to Rising Edge of Clock	80		nsec		
$\overline{NMI}$	$t_{w(NML)}$	Pulse Width, $\overline{NMI}$ Low	80		nsec		
$\overline{BUSRQ}$	$t_{s(BQ)}$	$\overline{BUSRQ}$ Setup Time to Rising Edge of Clock	50		nsec		
$\overline{BUSAK}$	$t_{d(BA)}$	Delay to Float, $\overline{BUSAK}$ Low		120	nsec	C_L = 50pF	
	$t_{d(BA)}$	Delay to Active, $\overline{BUSAK}$ High		140	nsec		
RESET	$t_{s(RS)}$	RESET Setup Time to Rising Edge of Clock	90		nsec		
	$t_{f(C)}$	Delay to Float ($\overline{MREQ}$, $\overline{IORQ}$, $\overline{RD}$ and $\overline{WR}$)		80	nsec		
	t_{mr}	M1 Stable Prior to IORQ (Interrupt Ack.)	[11]		nsec		[11] $t_{mr} = 2t_c + t_{w(\Phi H)} + t_f - 65$

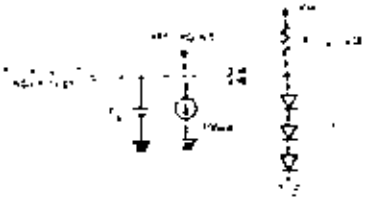
Notes

A. [illegible]

B. All control signals are internally synchronized, so they may be totally asynchronous with respect to the clock.

C. The $\overline{RESET}$ signal must be active for a minimum of 3 clock cycles.

D. Output loads are [illegible]



Zilog Z80-CPU INSTRUCTION SET

ADC HL, ss	Add with Carry Reg. pair ss to HL
ADC A, s	Add with carry operand s to Acc.
ADD A, n	Add value n to Acc.
ADD A, r	Add Reg. r to Acc.
ADD A, (HL)	Add location (HL) to Acc.
ADD A, (IX+d)	Add location (IX+d) to Acc.
ADD A, (IY+d)	Add location (IY+d) to Acc.
ADD HL, ss	Add Reg. pair ss to HL
ADD IX, pp	Add Reg. pair pp to IX
ADD IY, rr	Add Reg. pair rr to IY
AND s	Logical 'AND' of operand s and Acc.
BIT b, (HL)	Test BIT b of location (HL)
BIT b, (IX+d)	Test BIT b of location (IX+d)
BIT b, (IY+d)	Test BIT b of location (IY+d)
BIT b, r	Test BIT b of Reg. r
CALL cc, nn	Call subroutine at location nn if condition cc if true
CALL nn	Unconditional call subroutine at location nn
CCF	Complement carry flag
CP s	Compare operand s with Acc.
CPD	Compare location (HL) and Acc. decrement HL and BC
CPDR	Compare location (HL) and Acc. decrement HL and BC, repeat until BC=0
CPI	Compare location (HL) and Acc. increment HL and decrement BC
CPIR	Compare location (HL) and Acc. increment HL, decrement BC repeat until BC=0
CPL	Complement Acc. (1's complement)
DAA	Decimal adjust Acc.
DEC m	Decrement operand m
[illegible]	[illegible]

DEC IY	Decrement IY
DEC ss	Decrement Reg. pair ss
DI	Disable interrupts
DJNZ e	Decrement B and Jump relative if B≠0
EI	Enable interrupts
EX (SP), HL	Exchange the location (SP) and HL
EX (SP), IX	Exchange the location (SP) and IX
EX (SP), IY	Exchange the location (SP) and IY
EX AF, AF'	Exchange the contents of AF and AF'
EX DE, HL	Exchange the contents of DE and HL
EXX	Exchange the contents of BC, DE, HL with contents of BC', DE', HL' respectively
HALT	HALT (wait for interrupt or reset)
IM 0	Set interrupt mode 0
IM 1	Set interrupt mode 1
IM 2	Set interrupt mode 2
IN A, (n)	Load the Acc. with input fr device n
IN r, (C)	Load the Reg. r with input from device (C)
INC (HL)	Increment location (HL)
INC IX	Increment IX
INC (IX+d)	Increment location (IX+d)
INC IY	Increment IY
INC (IY+d)	Increment location (IY+d)
INC r	Increment Reg. r
INC ss	Increment Reg. pair ss
IND	Load location (HL) with input from port (C), decrement HL and B
INDR	Load location (HL) with input from port (C), decrement HL and decrement B, repeat until B=0
INI	Load location (HL) with input [illegible]

INIR	Load location (HL) with input from port (C), increment HL and decrement B, repeat until B=0
JP (HL)	Unconditional Jump to (HL)
JP (IX)	Unconditional Jump to (IX)
JP (IY)	Unconditional Jump to (IY)
JP cc, nn	Jump to location nn if condition cc is true
JP nn	Unconditional jump to location nn
JP C, e	Jump relative to PC+e if carry=1
JR e	Unconditional Jump relative to PC+e
JP NC, e	Jump relative to PC+e if carry=0
JR NZ, e	Jump relative to PC+e if non zero (Z=0)
JR Z, e	Jump relative to PC+e if zero (Z=1)
LD A, (BC)	Load Acc. with location (BC)
LD A, (DE)	Load Acc. with location (DE)
LD A, I	Load Acc. with I
LD A, (nn)	Load Acc. with location nn
LD A, R	Load Acc. with Reg. R
LD (BC), A	Load location (BC) with Acc.
LD (DE), A	Load location (DE) with Acc.
LD (HL), n	Load location (HL) with value n
LD dd, nn	Load Reg. pair dd with value nn
LD HL, (nn)	Load HL with location (nn)
LD (HL), r	Load location (HL) with Reg. r
LD I, A	Load I with Acc.
LD IX, nn	Load IX with value nn
LD IX, (nn)	Load IX with location (nn)
LD (IX+d), n	Load location (IX+d) with value n
LD (IX+d), r	Load location (IX+d) with Reg. r
LD IY, nn	Load IY with value nn
LD IY, (nn)	Load IY with location (nn)
LD (IY+d), n	Load location (IY+d) with value n
LD (IY+d), r	Load location (IY+d) with Reg. r
LD (nn), A	Load location (nn) with Acc.
LD (nn), dd	Load location (nn) with Reg. pair dd
LD (nn), HL	Load location (nn) with HL
LD (nn), IX	Load location (nn) with IX
LD (nn), IY	Load location (nn) with IY
LD R, A	Load R with Acc.
LD r, (HL)	Load Reg. r with location (HL)
LD r, (IX+d)	Load Reg. r with location (IX+d)
LD r, (IY+d)	Load Reg. r with location (IY+d)
LD r, n	Load Reg. r with value n
LD r, r'	Load Reg. r with Reg. r'
LD SP, HL	Load SP with HL
LD SP, IX	Load SP with IX
LD SP, IY	Load SP with IY
LDD	Load location (DE) with location (HL), decrement DE, HL and BC
LDDR	Load location (DE) with location (HL), decrement DE, HL and BC; repeat until BC=0
LDI	Load location (DE) with location (HL), increment DE, HL, decrement BC
LDIR	Load location (DE) with location (HL), increment DE, HL, decrement BC and repeat until BC=0
NEG	Negate Acc. (2's complement)
NOP	No operation
OR s	Logical 'OR' or operand s and Acc.
OTDR	Load output port (C) with location (HL) decrement HL and B, repeat until B=0
OTIR	Load output port (C) with location (HL), increment HL, decrement B, repeat until B=0
OUT (C), r	Load output port (C) with Reg. r
OUT (n), A	Load output port (n) with Acc.
OUTD	Load output port (C) with location (HL), decrement HL and B
OUTI	Load output port (C) with location (HL), increment HL and decrement B

POP IX	Load IX with top of stack
POP IY	Load IY with top of stack
POP qq	Load Reg. pair qq with top of stack
PUSH IX	Load IX onto stack
PUSH IY	Load IY onto stack
PUSH qq	Load Reg. pair qq onto stack
RES b, m	Reset Bit b of operand m
RET	Return from subroutine
RET cc	Return from subroutine if condition cc is true
RETI	Return from interrupt
RETN	Return from non maskable interrupt
RL m	Rotate left through carry operand m
RLA	Rotate left Acc. through carry
RLC (HL)	Rotate location (HL) left circular
RLC (IX+d)	Rotate location (IX+d) left circular
RLC (IY+d)	Rotate location (IY+d) left circular
RLC r	Rotate Reg. r left circular
RLCA	Rotate left circular Acc.
RLD	Rotate digit left and right between Acc. and location (HL)
RR m	Rotate right through carry operand m
RRA	Rotate right Acc. through carry
RRC m	Rotate operand m right circular
RRCA	Rotate right circular Acc.
RRD	Rotate digit right and left between Acc. and location (HL)
RST p	Restart to location p
SBC A, s	Subtract operand s from Acc. with carry
SBC HL, ss	Subtract Reg. pair ss from HL with carry
SCF	Set carry flag (C=1)
SET b, (HL)	Set Bit b of location (HL)
SET b, (IX+d)	Set Bit b of location (IX+d)
SET b, (IY+d)	Set Bit b of location (IY+d)
SET b, r	Set Bit b of Reg. r
SLA m	Shift operand m left arithmetic
SRA m	Shift operand m right arithmetic
SRL m	Shift operand m right logical
SUB s	Subtract operand s from Acc.
XOR s	Exclusive 'OR' operand s and Acc.

Z80™-PIO
Z80A™-PIO

Technical Manual

TABLE OF CONTENTS

1.0 INTRODUCTION

The Z-80 Parallel I/O (PIO) Circuit is a programmable, two port device which provides a TTL compatible interface between peripheral devices and the Z80-CPU. The CPU can configure the Z80-PIO to interface with a wide range of peripheral devices with no other external logic required. Typical peripheral devices that are fully compatible with the Z80 PIO include most keyboards, paper tape readers and punches, printers, PROM programmers, etc. The Z80-PIO utilizes N channel silicon gate depletion load technology and is packaged in a 40 pin DIP. Major features of the Z80-PIO include:

- Two independent 8 bit bidirectional peripheral interface ports with 'handshake' data transfer control
- Interrupt driven 'handshake' for fast response
- Any one of four distinct modes of operation may be selected for a port including:

  Byte output

  Byte input

  Byte bidirectional bus (Available on Port A only)

  Bit control mode

  All with interrupt controlled handshake
- Daisy chain priority interrupt logic included to provide for automatic interrupt vectoring without external logic
- Eight outputs are capable of driving Darlington transistors
- All inputs and outputs fully TTL compatible
- Single 5 volt supply and single phase clock are required.

One of the unique features of the Z80-PIO that separates it from other interface controllers is that all data transfer between the peripheral device and the CPU is accomplished under total interrupt control. The interrupt logic of the PIO permits full usage of the efficient interrupt capabilities of the Z80-CPU during I/O transfers. All logic necessary to implement a fully nested interrupt structure is included in the PIO so that additional circuits are not required. Another unique feature of the PIO is that it can be programmed to interrupt the CPU on the occurrence of specified status conditions in the peripheral device. For example, the PIO can be programmed to interrupt if any specified peripheral alarm conditions should occur. This interrupt capability reduces the amount of time that the processor must spend in polling peripheral status.

2.0 PIO ARCHITECTURE

A block diagram of the Z80-PIO is shown in Figure 2.0-1. The internal structure of the Z80-PIO consists of a Z80-CPU bus interface, internal control logic, Port A I/O logic, Port B I/O logic, and interrupt control logic. The CPU bus interface logic allows the PIO to interface directly to the Z80-CPU with no other external logic. However, address decoders and/or line buffers may be required for large systems. The internal control logic synchronizes the CPU data bus to the peripheral device interfaces (Port A and Port B). The two I/O ports (A and B) are virtually identical and are used to interface directly to peripheral devices.

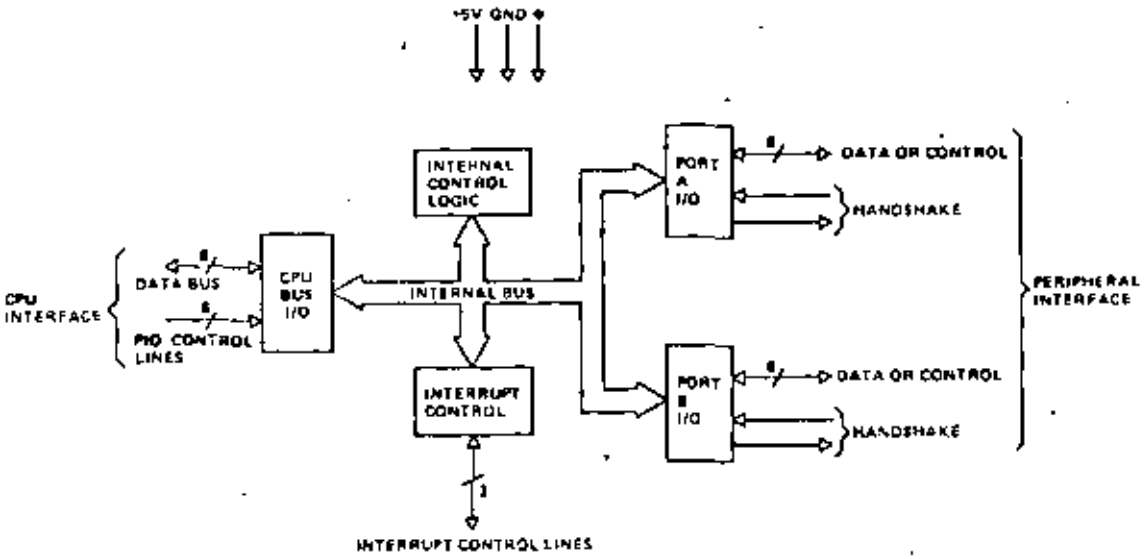


FIGURE 2.0-1
PIO BLOCK DIAGRAM

The Port I/O logic is composed of 6 registers with "handshake" control logic as shown in Figure 2.0-2. The registers include: an 8 bit data input register, an 8 bit data output register, a 2 bit mode control register, an 8 bit mask register, an 8 bit input/output select register, and a 2 bit mask control register.

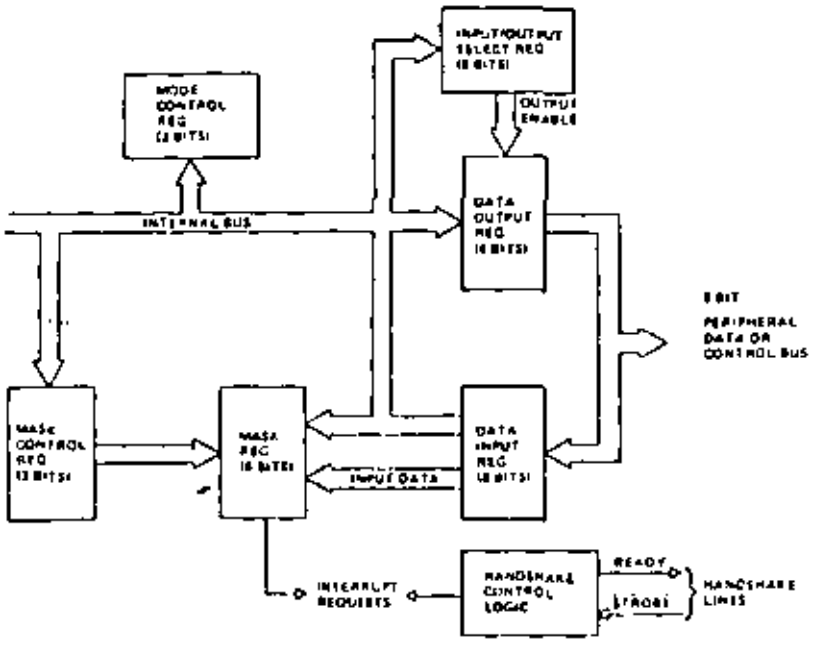


FIGURE 2.0-2
PORT I/O BLOCK DIAGRAM

The 2-bit mode control register is loaded by the CPU to select the desired operating mode (byte output, byte input, byte bidirectional bus, or bit control mode). All data transfer between the peripheral device and the CPU is achieved through the data input and data output registers. Data may be written into the output register by the CPU or read back to the CPU from the input register at any time. The handshake lines associated with each port are used to control the data transfer between the PIO and the peripheral device.

The 8-bit mask register and the 8-bit input/output select register are used only in the bit control mode. In this mode any of the 8 peripheral data or control bus pins can be programmed to be an input or an output as specified by the select register. The mask register is used in this mode in conjunction with a special interrupt feature. This feature allows an interrupt to be generated when any or all of the unmasked pins reach a specified state (either high or low). The 2-bit mask control register specifies the active state desired (high or low) and if the interrupt should be generated when *all* unmasked pins are active (AND condition) or when *any* unmasked pin is active (OR condition). This feature reduces the requirement for CPU status checking of the peripheral by allowing an interrupt to be automatically generated on specific peripheral status conditions. For example, in a system with 3 alarm conditions, an interrupt may be generated if any one occurs or if all three occur.

The interrupt control logic section handles all CPU interrupt protocol for nested priority interrupt structures. The priority of any device is determined by its physical location in a daisy chain configuration. Two lines are provided in each PIO to form this daisy chain. The device closest to the CPU has the highest priority. Within a PIO, Port A interrupts have higher priority than those of Port B. In the byte input, byte output or bidirectional modes, an interrupt can be generated whenever a new byte transfer is requested by the peripheral. In the bit control mode an interrupt can be generated when the peripheral status matches a programmed value. The PIO provides for complete control of nested interrupts. That is, lower priority devices may not interrupt higher priority devices that have not had their interrupt service routine completed by the CPU. Higher priority devices may interrupt the servicing of lower priority devices.

When an interrupt is accepted by the CPU in mode 2, the interrupting device must provide an 8-bit interrupt vector for the CPU. This vector is used to form a pointer to a location in the computer memory where the address of the interrupt service routine is located. The 8-bit vector from the interrupting device forms the least significant 8 bits of the indirect pointer while the I Register in the CPU provides the most significant 8 bits of the pointer. Each port (A and B) has an independent interrupt vector. The least significant bit of the vector is automatically set to a 0 within the PIO since the pointer must point to two adjacent memory locations for a complete 16-bit address.

The PIO decodes the RETI (Return from interrupt) instruction directly from the CPU data bus so that each PIO in the system knows at all times whether it is being serviced by the CPU interrupt service routine without any other communication with the CPU.

3.0 PIN DESCRIPTION

A diagram of the Z80-PIO pin configuration is shown in Figure 3.0-1. This section describes the function of each pin.

D_7-D_0 — Z80-CPU Data Bus (bidirectional, tristate)
This bus is used to transfer all data and commands between the Z80-CPU and the Z80-PIO. D_0 is the least significant bit of the bus.

B/A Sel — Port B or A Select (input, active high)
This pin defines which port will be accessed during a data transfer between the Z80-CPU and the Z80-PIO. A low level on this pin selects Port A while a high level selects Port B. Often Address bit A_0 from the CPU will be used for this selection function.

C/D Sel — Control or Data Select (input, active high)
This pin defines the type of data transfer to be performed between the CPU and the PIO. A high level on this pin during a CPU write to the PIO causes the Z-80 data bus to be interpreted as a *command* for the port selected by the B/A Select line. A low level on this pin means that he Z-80 data bus is being used to transfer data between the CPU and the PIO. Often Address bit A_1 from the CPU will be used for this function.

$\overline{CE}$ — Chip Enable (input, active low)
A low level on this pin enables the PIO to accept command or data inputs from the CPU during a write cycle or to transmit data to the CPU during a read cycle. This signal is generally a decode of four I/O port numbers that encompass port A and B, data and control.

Φ — System Clock (input)
The Z80-PIO uses the standard Z-80 system clock to synchronize certain signals internally. This is a single phase clock.

$\overline{M1}$ — Machine Cycle One Signal from CPU (input, active low)
This signal from the CPU is used as a sync pulse to control several internal PIO operations. When $\overline{M1}$ is active and the $\overline{RD}$ signal is active, the Z80-CPU is fetching an instruction from memory. Conversely, when $\overline{M1}$ is active and $\overline{IORQ}$ is active, the CPU is acknowledging an interrupt. In addition, the $\overline{M1}$ signal has two other functions within the Z80-PIO.

1. $\overline{M1}$ synchronizes the PIO interrupt logic.
2. When $\overline{M1}$ occurs without an active $\overline{RD}$ or $\overline{IORQ}$ signal the PIO logic enters a reset state.

$\overline{IORQ}$ — Input/Output Request from Z80-CPU (input, active low)
The $\overline{IORQ}$ signal is used in conjunction with the B/A Select, C/D Select, $\overline{CE}$, and $\overline{RD}$ signals to transfer commands and data between the Z80-CPU and the Z80-PIO. When $\overline{CE}$, $\overline{RD}$ and $\overline{IORQ}$ are active, the port addressed by B/A will transfer data to the CPU (a read operation). Conversely, when $\overline{CE}$ and $\overline{IORQ}$ are active but $\overline{RD}$ is not active, then the port addressed by B/A will be written into from the CPU with either data or control information as specified by the C/D Select signal. Also, if $\overline{IORQ}$ and $\overline{M1}$ are active simultaneously, the CPU is acknowledging an interrupt and the interrupting port will automatically place its interrupt vector on the CPU data bus if it is the highest priority device requesting an interrupt.

$\overline{RD}$ — Read Cycle Status from the Z80-CPU (input, active low)
If $\overline{RD}$ is active a MEMORY READ or I/O READ operation is in progress. The $\overline{RD}$ signal is used with B/A Select, C/D Select, $\overline{CE}$, and $\overline{IORQ}$ signals to transfer data from the Z80-PIO to the Z80-CPU.

IEI — Interrupt Enable In (input, active high)
This signal is used to form a priority interrupt daisy chain when more than one interrupt driven device is being used. A high level on this pin indicates that no other devices of higher priority are being serviced by a CPU interrupt service routine.

IEO — Interrupt Enable Out (output, active high)
The IEO signal is the other signal required to form a daisy chain priority scheme. It is high only if IEI is high and the CPU is not servicing an interrupt from this PIO. Thus this signal blocks lower priority devices from interrupting while a higher priority device is being serviced by its CPU interrupt service routine.

$\overline{INT}$ — Interrupt Request (output, open drain, active low)
When $\overline{INT}$ is active the Z80-PIO is requesting an interrupt from the Z80-CPU.

$A_0 - A_7$ — Port A Bus (bidirectional, tristate)
This 8 bit bus is used to transfer data and/or status or control information between Port A of the Z80-PIO and a peripheral device. A_0 is the least significant bit of the Port A data bus.

$\overline{A\ STB}$ — Port A Strobe Pulse from Peripheral Device (input, active low)
The meaning of this signal depends on the mode of operation selected for Port A as follows:

1) Output mode: The positive edge of this strobe is issued by the peripheral to acknowledge the receipt of data made available by the PIO.
2) Input mode: The strobe is issued by the peripheral to load data from the peripheral into the Port A input register. Data is loaded into the PIO when this signal is active.
3) Bidirectional mode: When this signal is active, data from the Port A output register is gated onto Port A bidirectional data bus. The positive edge of the strobe acknowledges the receipt of the data.
4) Control mode: The strobe is inhibited internally.

A RDY — Register A Ready (output, active high)
The meaning of this signal depends on the mode of operation selected for Port A as follows:

1) Output mode: This signal goes active to indicate that the Port A output register has been loaded and the peripheral data bus is stable and ready for transfer to the peripheral device.
2) Input mode: This signal is active when the Port A input register is empty and is ready to accept data from the peripheral device.
3) Bidirectional mode: This signal is active when data is available in the Port A output register for transfer to the peripheral device. In this mode data is not placed on the Port A data bus unless $\overline{A\ STB}$ is active.
4) Control mode: This signal is disabled and forced to a low state.

$B_0 - B_7$ — Port B Bus (bidirectional, tristate)
This 8 bit bus is used to transfer data and/or status or control information between Port B of the PIO and a peripheral device. The Port B data bus is capable of supplying 1.5ma @ 1.5V to drive Darlington transistors. B_0 is the least significant bit of the bus.

$\overline{B\ STB}$ — Port B Strobe Pulse from Peripheral Device (input, active low)
The meaning of this signal is similar to that of $\overline{A\ STB}$ with the following exception:
In the Port A bidirectional mode this signal strobes data from the peripheral device into the Port A input register.

B RDY — Register B Ready (output, active high)
The meaning of this signal is similar to that of A Ready with the following exception:
In the Port A bidirectional mode this signal is high when the Port A input register is empty and ready to accept data from the peripheral device.

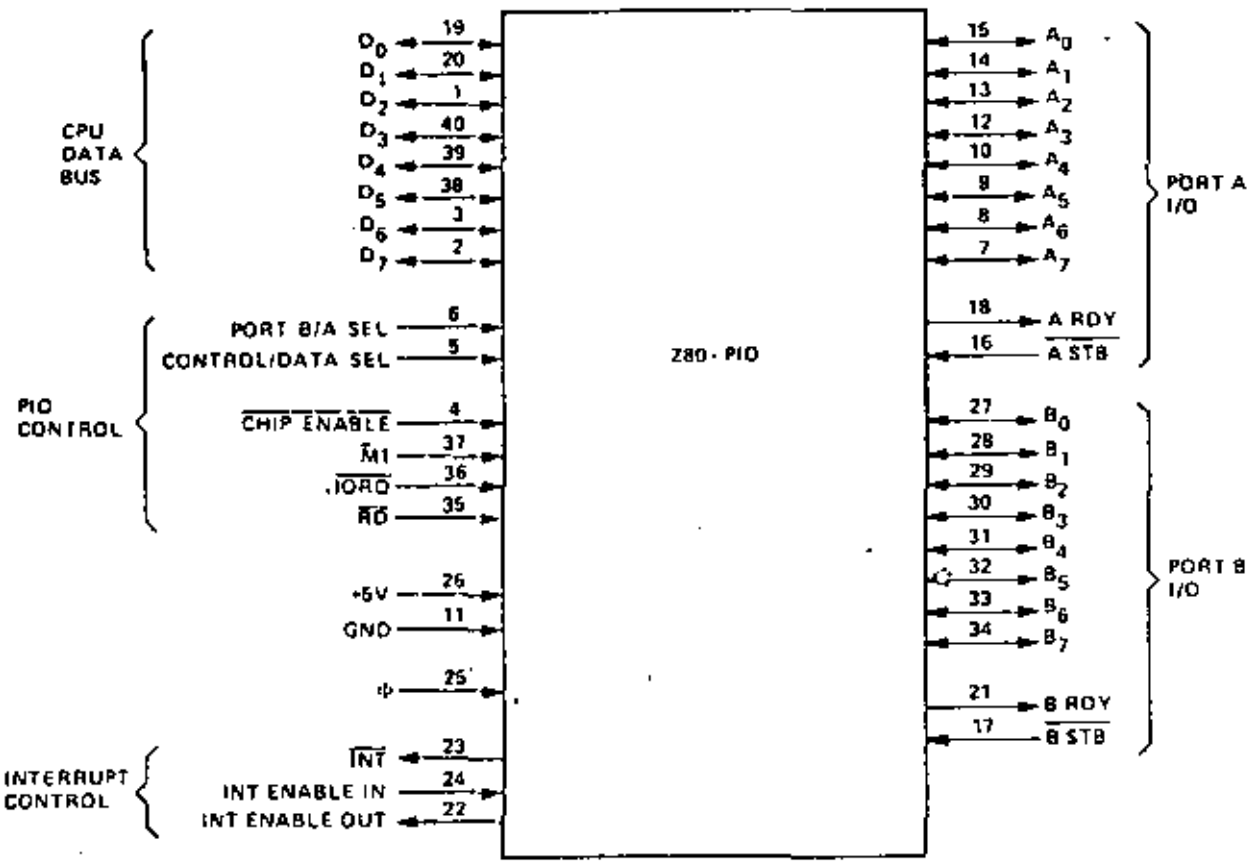


FIGURE 3.0-1
PIO PIN CONFIGURATION

4.0 PROGRAMMING THE PIO

4.1 RESET

The Z80 PIO automatically enters a reset state when power is applied. The reset state performs the following functions:

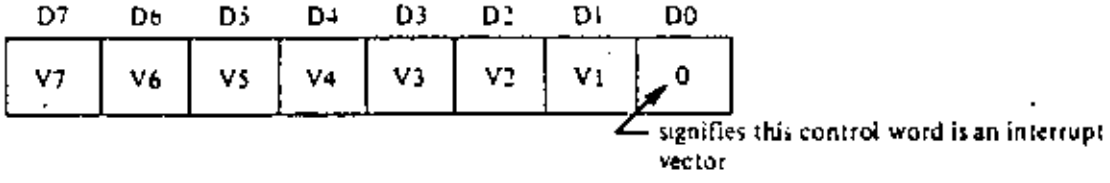
1) Both port mask registers are reset to inhibit all port data bits.
2) Port data bus lines are set to a high impedance state and the Ready "handshake" signals are inactive (low). Mode 1 is automatically selected.
3) The vector address registers are *not* reset.
4) Both port interrupt enable flip flops are reset.
5) Both port output registers are reset.

In addition to the automatic power on reset, the PIO can be reset by applying an $\overline{M1}$ signal without the presence of a $\overline{RD}$ or $\overline{IORQ}$ signal. If no $\overline{RD}$ or $\overline{IORQ}$ is detected during $\overline{M1}$ the PIO will enter the reset state immediately after the $\overline{M1}$ signal goes inactive. The purpose of this reset is to allow a single external gate to generate a reset without a power down sequence. This approach was required due to the 40 pin packaging limitation.

Once the PIO has entered the internal reset state it is held there until the PIO receives a control word from the CPU.

4.2 LOADING THE INTERRUPT VECTOR

The PIO has been designed to operate with the Z80-CPU using the mode 2 interrupt response. This mode requires that an interrupt vector be supplied by the interrupting device. This vector is used by the CPU to form the address for the interrupt service routine of that port. This vector is placed on the Z-80 data bus during an interrupt acknowledge cycle by the highest priority device requesting service at that time. (Refer to the Z80-CPU Technical Manual for details on how an interrupt is serviced by the CPU). The desired interrupt vector is loaded into the PIO by writing a control word to the desired port of the PIO with the following format:

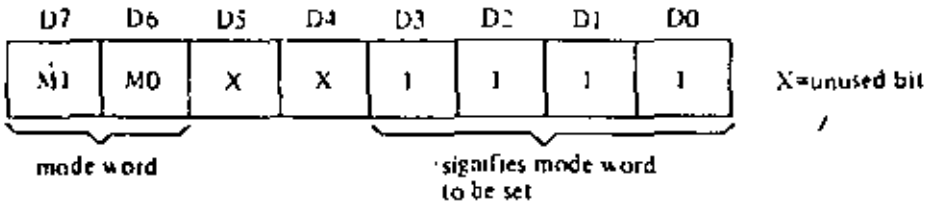


D0 is used in this case as a flag bit which when low causes V7 thru V1 to be loaded into the vector register. At interrupt acknowledge time, the vector of the interrupting port will appear on the Z-80 data bus exactly as shown in the format above.

4.3 SELECTING AN OPERATING MODE

Port A of the PIO may be operated in any of four distinct modes: Mode 0 (output mode), Mode 1 (input mode), Mode 2 (bidirectional mode), and Mode 3 (control mode). Note that the mode numbers have been selected for mnemonic significance; i.e. 0=Out, 1=In, 2=Bidirectional. Port B can operate in any of these modes except Mode 2.

The mode of operation must be established by writing a control word to the PIO in the following format:



Bits D7 and D6 from the binary code for the desired mode according to the following table:

D7	D6	Mode
0	0	0 (output)
0	1	1 (input)
1	0	2 (bidirectional)
1	1	3 (control)

Bits D5 and D4 are ignored. Bits D3-D0 must be set to 1111 to indicate "Set Mode".

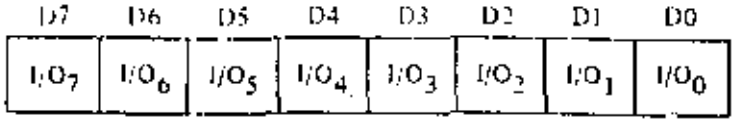
Selecting Mode 0 enables any data written to the port output register by the CPU to be enabled onto the port data bus. The contents of the output register may be changed at any time by the CPU simply by writing a new data word to the port. Also the current contents of the output register may be read back to the Z80-CPU at any time through the execution of an input instruction.

With Mode 0 active, a data write from the CPU causes the Ready handshake line of that port to go high to notify the peripheral that data is available. This signal remains high until a strobe is received from the peripheral. The rising edge of the strobe generates an interrupt (if it has been enabled) and causes the Ready line to go inactive. This very simple handshake is similar to that used in many peripheral devices.

Selecting Mode 1 puts the port into the input mode. To start handshake operation, the CPU merely performs an input read operation from the port. This activates the Ready line to the peripheral to signify that data should be loaded into the empty input register. The peripheral device then strobes data into the port input register using the strobe line. Again, the rising edge of the strobe causes an interrupt request (if it has been enabled) and deactivates the Ready signal. Data may be strobed into the input register regardless of the state of the Ready signal if care is taken to prevent a data overrun condition.

Mode 2 is a bidirectional data transfer mode which uses all four handshake lines. Therefore only Port A may be used for Mode 2 operation. Mode 2 operation uses the Port A handshake signals for output control and the Port B handshake signals for input control. Thus, both A RDY and B RDY may be active simultaneously. The only operational difference between Mode 0 and the output portion of Mode 2 is that data from the Port A output register is allowed on to the port data bus only when A $\overline{STB}$ is active in order to achieve a bidirectional capability.

Mode 3 operation is intended for status and control applications and does not utilize the handshake signals. When Mode 3 is selected, the next control word sent to the PIO must define which of the port data bus lines are to be inputs and which are outputs. The format of the control word is shown below:

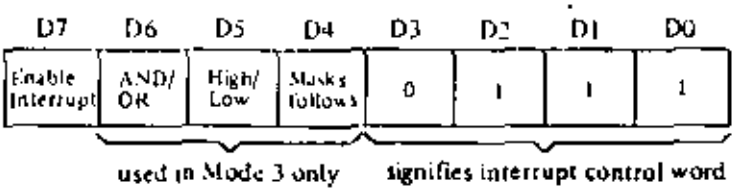


If any bit is set to a one, then the corresponding data bus line will be used as an input. Conversely, if the bit is reset, the line will be used as an output.

During Mode 3 operation the strobe signal is ignored and the Ready line is held low. Data may be written to a port or read from a port by the Z80-CPU at any time during Mode 3 operation. When reading a port, the data returned to the CPU will be composed of input data from port data bus lines assigned as inputs plus port output register data from those lines assigned as outputs.

4.4 SETTING THE INTERRUPT CONTROL WORD

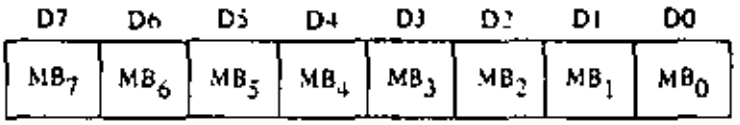
The interrupt control word for each port has the following format:



If bit D7=1 the interrupt enable flip flop of the port is set and the port may generate an interrupt. If bit D7=0 the enable flag is reset and interrupts may not be generated. If an interrupt is pending when the enable flag is set, it will then be enabled onto the CPU interrupt request line. Bits D6, D5, and D4 are used only with Mode 3 operation. However, setting bit D4 of the interrupt control word during any mode of operation will cause any pending interrupt to be reset. These three bits are used to allow for interrupt operation in Mode 3 when any group of the I/O lines go to certain defined states. Bit D6 (AND/OR) defines the logical operation to be performed in port monitoring. If bit D6=1, an AND function is specified and if D6=0, an OR function is specified. For example, if the AND function is specified, all bits must go to a specified state before an interrupt will be generated while the OR function will generate an interrupt if any specified bit goes to the active state.

Bit D5 defines the active polarity of the port data bus line to be monitored. If bit D5=1 the port data lines are monitored for a high state while if D5=0 they will be monitored for a low state.

If bit D4=1 the next control word sent to the PIO must define a mask as follows:



Only those port lines whose mask bit is zero will be monitored for generating an interrupt.

5.0 TIMING

5.1 OUTPUT MODE (MODE 0)

Figure 5.0-1 illustrates the timing associated with Mode 0 operation. An output cycle is always started by the execution of an output instruction by the CPU. A $\overline{WR}$* pulse is generated by the PIO during a CPU I/O write operation and is used to latch the data from the CPU data bus into the addressed port's (A or B) output register. The rising edge of the $\overline{WR}$* pulse then raises the Ready flag after the next falling edge of Φ to indicate that data is available for the peripheral device. In most systems the rising edge of the Ready signal can be used as a latching signal in the peripheral device if desired. The Ready signal will remain active until: (1) a positive edge is received from the strobe line indicating that the peripheral has taken the data, or (2) if already active, Ready will be forced low 1½ Φ cycles after the leading edge of $\overline{IORQ}$ if the port's output register is written into. Ready will return high on the first falling edge of Φ after the trailing edge of $\overline{IORQ}$. This guarantees that Ready is low when port data is changing. The Ready signal will not go inactive until a falling edge occurs on the clock (Φ) line. The purpose of delaying the negative transition of the Ready signal until after a negative clock transition is that it allows for a very simple generation scheme for the strobe pulse. By merely connecting the Ready line to the Strobe line, a strobe with a duration of one clock period will be generated with no other logic required. The positive edge of the strobe pulse automatically generates an $\overline{INT}$ request if the interrupt enable flip flop has been set and this device is the highest priority device requesting an interrupt.

If the PIO is not in a reset state, the output register may be loaded before mode 0 is selected. This allows the port output lines to become active in a user defined state.

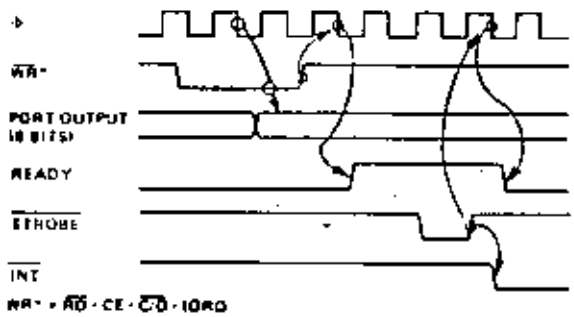


FIGURE 5.0-1
MODE 0 (OUTPUT) TIMING

5.2 INPUT MODE (MODE 1)

Figure 5.0-2 illustrates the timing of an input cycle. The peripheral initiates this cycle using the strobe line after the CPU has performed a data read. A low level on this line loads data into the port input register and the rising edge of the strobe line activates the interrupt request line ($\overline{INT}$) if the interrupt enable is set and this is the highest priority requesting device. The next falling edge of the clock line (Φ) will then reset the Ready line to an inactive state signifying that the input register is full and further loading must be inhibited until the CPU reads the data. The CPU will in the course of its interrupt service routine, read the data from the interrupting port. When this occurs, the positive edge from the CPU $\overline{RD}$ signal will raise the Ready line with the next low going transition of Φ, indicating that new data can be loaded into the PIO. If already active, Ready will be forced low one and one-half Φ periods following the leading edge of $\overline{IORQ}$ during a read of a PIO port. If the user strobes data into the PIO only when Ready is high, the forced state of Ready will prevent input register data from changing while the CPU is reading the PIO. Ready will go high again after the trailing edge of the $\overline{IORQ}$ as previously described.

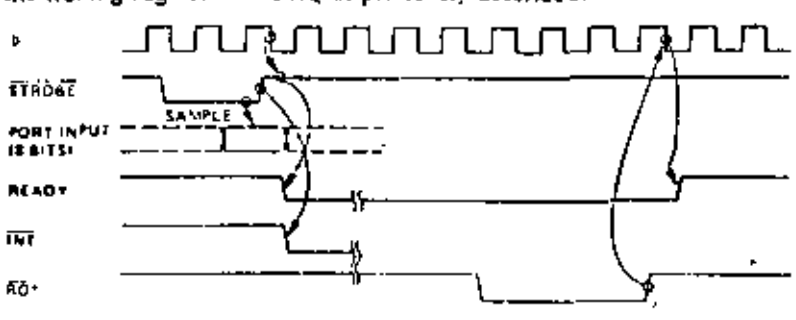


FIGURE 5.0-2
MODE 1 (INPUT) TIMING

5.3 BIDIRECTIONAL MODE (MODE 2)

This mode is merely a combination of Mode 0 and Mode 1 using all four handshake lines. Since it requires all four lines, it is available only on Port A. When this mode is used on Port A, Port B must be set to the Bit Control Mode. The same interrupt vector will be returned for a Mode 3 interrupt on Port B and an input transfer interrupt during Mode 2 operation of Port A. Ambiguity is avoided if Port B is operated in a polled mode and the Port B mask register is set to inhibit all bits.

Figure 5.0-3 illustrates the timing for this mode. It is almost identical to that previously described for Mode 0 and Mode 1 with the Port A handshake lines used for output control and the Port B lines used for input control. The difference between the two modes is that, in Mode 2, data is allowed out onto the bus only when the A strobe is low. The rising edge of this strobe can be used to latch the data into the peripheral since the data will remain stable until after this edge. The input portion of Mode 2 operates identically to Mode 1. Note that both Port A and Port B must have their interrupts enabled to achieve an interrupt driven bidirectional transfer.

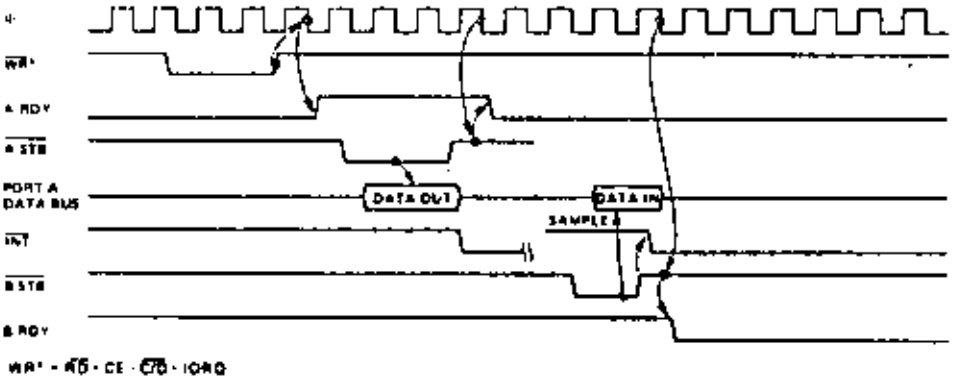


FIGURE 5.0-3
PORT A, MODE 2 (BIDIRECTIONAL) TIMING

The peripheral must not gate data onto a port data bus while $\overline{A\ STB}$ is active. Bus contention is avoided if the peripheral uses $\overline{B\ STB}$ to gate input data onto the bus. The PIO uses the $\overline{B\ STB}$ low level to latch this data. The PIO has been designed with a zero hold time requirement for the data when latching in this mode so that this simple gating structure can be used by the peripheral. That is, the data can be disabled from the bus immediately after the strobe rising edge.

5.4 CONTROL MODE (MODE 3)

The control mode does not utilize the handshake signals and a normal port write or port read can be executed at any time. When writing, the data will be latched into output registers with the same timing as Mode 0. A RDY will be forced low whenever Port A is operated in Mode 3. B RDY will be held low whenever Port B is operated in Mode 3 unless Port A is in Mode 2. In the latter case, the state of B RDY will not be affected.

When reading the PIO, the data returned to the CPU will be composed of output register data from those port data lines assigned as outputs and input register data from those port data lines assigned as inputs. The input register will contain data which was present immediately prior to the falling edge of $\overline{RD}$. See Figure 5.0-4.

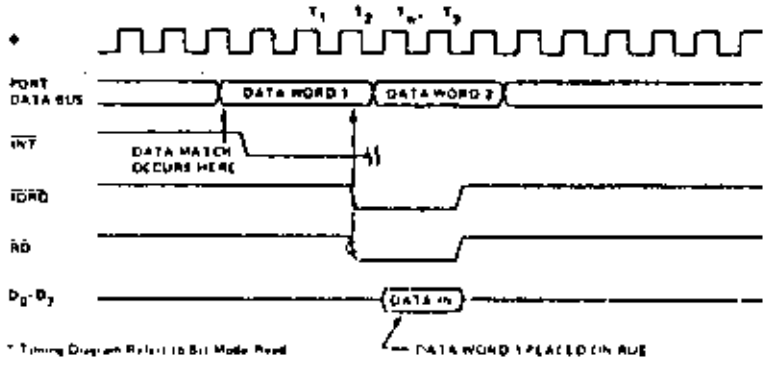


FIGURE 5.0-4

An interrupt will be generated if interrupts from the port are enabled and the data on the port data lines satisfies the logical equation defined by the 8-bit mask and 2-bit mask control registers. Another interrupt will not be generated until a change occurs in the status of the logical equation. A Mode 3 interrupt will be generated only if the result of a Mode 3 logical operation changes from false to true. For example, assume that the Mode 3 logical equation is an "OR" function. An unmasked port data line becomes active and an interrupt is requested. If a second unmasked port data line becomes active concurrently with the first, a new interrupt will not be requested since a change in the result of the Mode 3 logical operation has not occurred.

If the result of a logical operation becomes true immediately prior to or during $\overline{M1}$, an interrupt will be requested after the trailing edge of $\overline{M1}$.

6.0 INTERRUPT SERVICING

Some time after an interrupt is requested by the PIO, the CPU will send out an interrupt acknowledge ($\overline{M1}$ and $\overline{IORQ}$). During this time the interrupt logic of the PIO will determine the highest priority port which is requesting an interrupt. (This is simply the device with its Interrupt Enable Input high and its Interrupt Enable Output low). To insure that the daisy chain enable lines stabilize, devices are inhibited from changing their interrupt request status when $\overline{M1}$ is active. The highest priority device places the contents of its interrupt vector register onto the Z80 data bus during interrupt acknowledge.

Figure 6.0-1 illustrates the timing associated with interrupt requests. During $\overline{M1}$ time, no new interrupt requests can be generated. This gives time for the Int Enable signals to ripple through up to four PIO circuits. The PIO with IEI high and IEO low during $\overline{INTA}$ will place the 8-bit interrupt vector of the appropriate port on the data bus at this time.

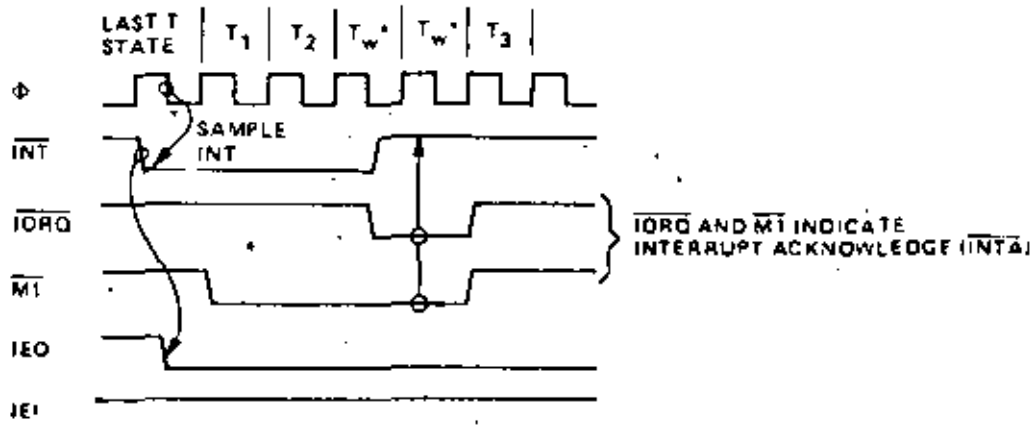


FIGURE 6.0-1
INTERRUPT ACKNOWLEDGE TIMING

If an interrupt requested by the PIO is acknowledged, the requesting port is 'under service'. IEO of this port will remain low until a return from interrupt instruction (RETI) is executed while IEI of the port is high. If an interrupt request is not acknowledged, IEO will be forced high for one $\overline{M1}$ cycle after the PIO decodes the opcode 'ED'. This action guarantees that the two byte RETI instruction is decoded by the proper PIO port. See Figure 6.0-2.

Figure 6.0-3 illustrates a typical nested interrupt sequence that could occur with four ports connected in the daisy chain. In this sequence Port 2A requests and is granted an interrupt. While this port is being serviced, a higher priority port (1B) requests and is granted an interrupt. The service routine for the higher priority port is completed and a RETI instruction is executed to indicate to the port that its routine is complete. At this time the service routine of the lower priority port is completed.

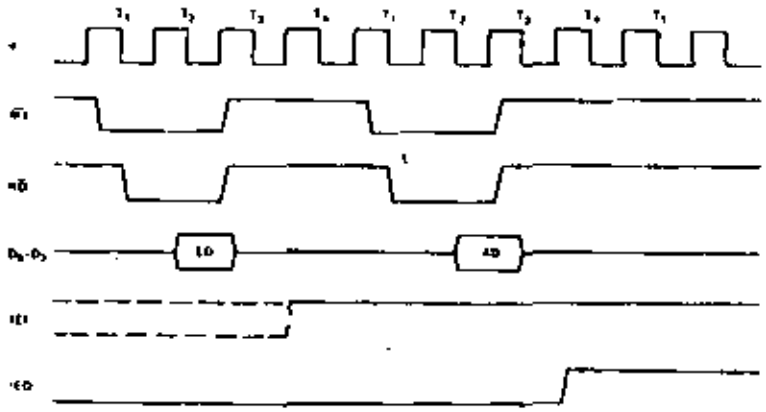
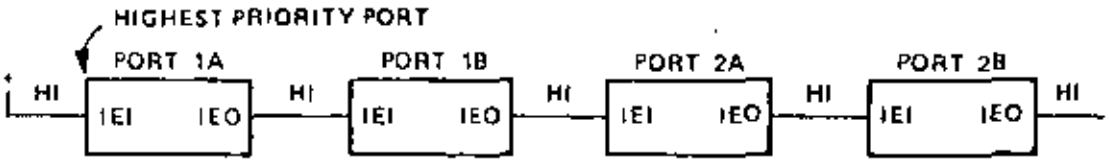
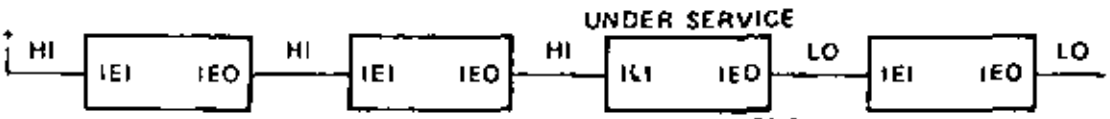


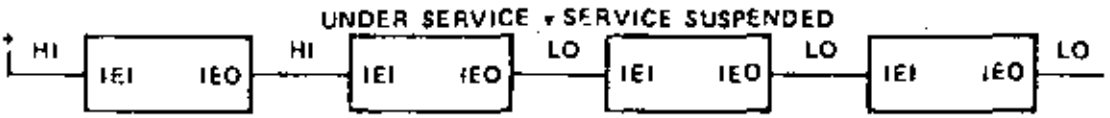
FIGURE 6.0-2
RETURN FROM INTERRUPT CYCLE



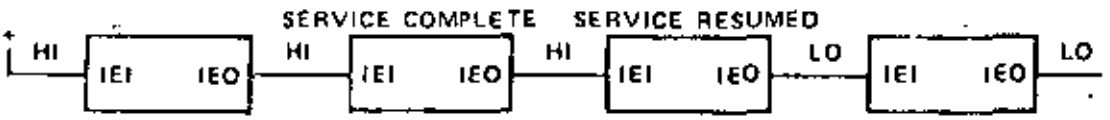
1. PRIORITY INTERRUPT DAISY CHAIN BEFORE ANY INTERRUPT OCCURS.



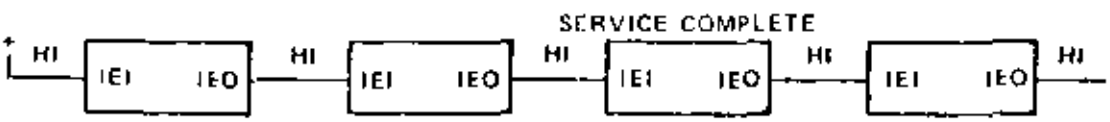
2. PORT 2A REQUESTS AN INTERRUPT AND IS ACKNOWLEDGED.



3. PORT 1B INTERRUPTS, SUSPENDS SERVICING OF PORT 2A.



4. PORT 1B SERVICE ROUTINE COMPLETE, "RETI" ISSUED, PORT 2A SERVICE RESUMED.



5. SECOND "RETI" INSTRUCTION ISSUED ON COMPLETION OF PORT 2A SERVICE ROUTINE.

FIGURE 6.0-3
DAISY CHAIN INTERRUPT SERVICING

7.0 APPLICATIONS

7.1 EXTENDING THE INTERRUPT DAISY CHAIN

Without any external logic, a maximum of four Z80-PIO devices may be daisy chained into a priority interrupt structure. This limitation is required so that the interrupt enable status (IEO) ripples through the entire chain between the beginning of $\overline{M1}$, and the beginning of $\overline{IORQ}$ during an interrupt acknowledge cycle. Since the interrupt enable status cannot change during $\overline{M1}$, the vector address returned to the CPU is assured to be from the highest priority device which requested an interrupt.

If more than four PIO devices must be accommodated, a "look-ahead" structure may be used as shown in Figure 7.0-1. With this technique more than thirty PIO's may be chained together using standard TTL logic.

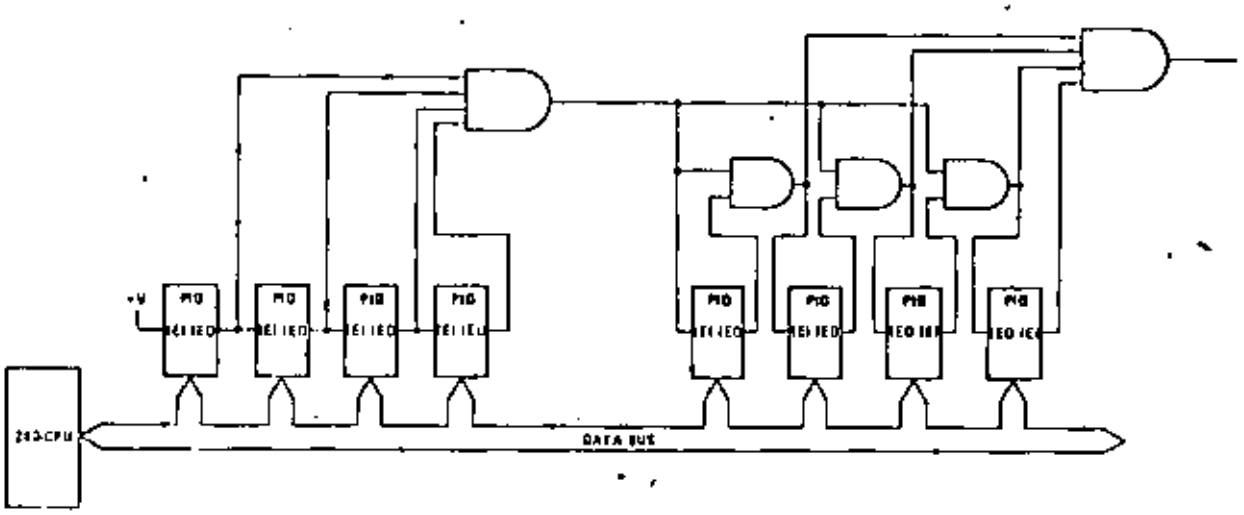


FIGURE 7.0-1
A METHOD OF EXTENDING THE INTERRUPT PRIORITY DAISY CHAIN

7.2 I/O DEVICE INTERFACE

In this example, the Z80-PIO is connected to an I/O terminal device which communicates over an 8 bit parallel bidirectional data bus as illustrated in Figure 7.0-2. Mode 2 operation (bidirectional) is selected by sending the following control word to Port A:

D7	D6	D5	D4	D3	D2	D1	D0
1	0	X	X	1	1	1	1

Mode Control

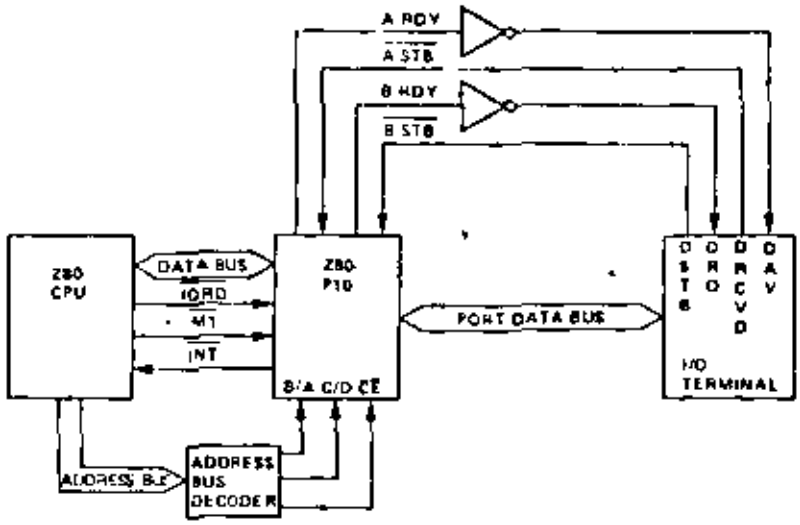


FIGURE 7.0-2

EXAMPLE I/O INTERFACE

Next, the proper interrupt vector is loaded (refer to CPU Manual for details on the operation of the interrupt).

D7	D6	D5	D4	D3	D2	D1	D0
V7	V6	V5	V4	V3	V2	V1	0

Interrupts are then enabled by the rising edge of the first $\overline{M1}$ after the interrupt mode word is set unless that $\overline{M1}$ defines an interrupt acknowledge cycle. If a mask follows the interrupt mode word, interrupts are enabled by the rising edge of the first $\overline{M1}$ following the setting of the mask.

Data can now be transferred between the peripheral and the CPU. The timing for this transfer is as described in Section 5.0.

7.3 CONTROL INTERFACE

A typical control mode application is illustrated in Figure 7.0-3. Suppose an industrial process is to be monitored. The occurrence of any abnormal operating condition is to be reported to a Z80-CPU based control system. The process control and status word has the following format:

D7	D6	D5	D4	D3	D2	D1	D0
Special Test	Turn On Power	Power Failure Alarm	Halt Processing	Temp. Alarm	Turn Heaters On	Pressurize System	Pressure Alarm

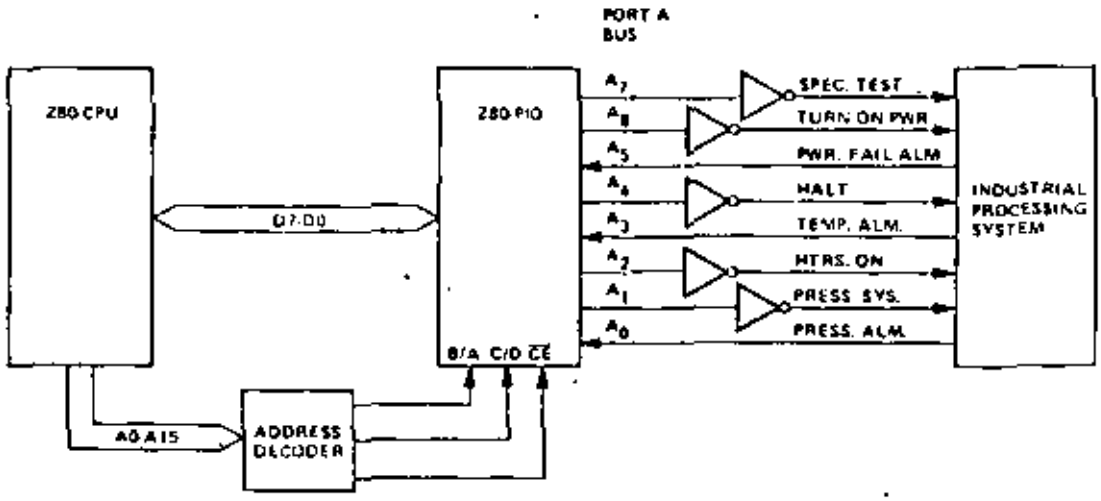


FIGURE 7.0-3
CONTROL MODE APPLICATION

The PIO may be used as follows. First Port A is set for Mode 3 operation by writing the following control word to Port A.

D7	D6	D5	D4	D3	D2	D1	D0
1	1	X	X	1	1	1	1

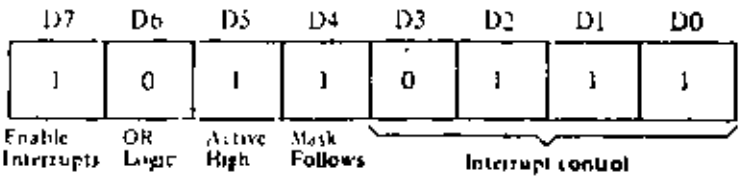
Whenever Mode 3 is selected, the next control word sent to the port must be an I/O select word. In this example we wish to select port data lines A5, A3 and A0 as inputs and so the following control word is written:

D7	D6	D5	D4	D3	D2	D1	D0
0	0	1	0	1	0	0	1

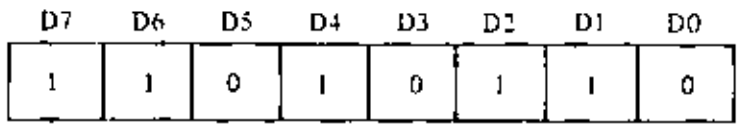
Next the desired interrupt vector must be loaded (refer to the CPU manual for details):

D7	D6	D5	D4	D3	D2	D1	D0
V7	V6	V5	V4	V3	V2	V1	0

An interrupt control word is next sent to the port:

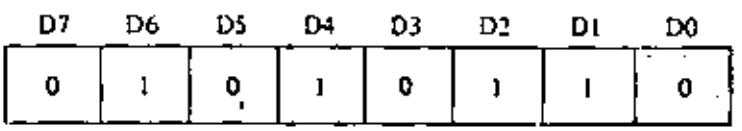


The mask word following the interrupt mode word is:



Selects A5, A3 and A0 to be monitored

Now, if a sensor puts a high level on line A5, A3, or A0, an interrupt request will be generated. The mask word may select any combination of inputs or outputs to cause an interrupt. For example, if the mask word above had been:



then an interrupt request would also occur if bit A7 (Special Test) of the output register was set.

Assume that the following port assignments are to be used:

$E0_H$ = Port A Data
$E1_H$ = Port B Data
$E2_H$ = Port A Control
$E3_H$ = Port B Control

All port numbers are in hexadecimal notation. This particular assignment of port numbers is convenient since A_0 of the address bus can be used as the Port B/A Select and A_1 of the address bus can be used as the Control/Data Select. The Chip Enable would be the decode of CPU address bits A_7 thru A_2 (1110 00). Note that if only a few peripheral devices are being used, a Chip Enable decode may not be required since a higher order address bit could be used directly.

8.0 PROGRAMMING SUMMARY

8.1 LOAD INTERRUPT VECTOR

V7	V6	V5	V4	V3	V2	V1	0

8.2 SET MODE

M1	M0	X	X	1	1	1	1

M_1	M_0	Mode
0	0	Output
0	1	Input
1	0	Bidirectional
1	1	Bit Control

When selecting Mode 3, the next word must set the I/O Register:

I/O_7	I/O_6	I/O_5	I/O_4	I/O_3	I/O_2	I/O_1	I/O_0

I/O = 1 Sets bit to Input
I/O = 0 Sets bit to Output

8.3 SET INTERRUPT CONTROL

Int Enable	AND/ OR	High/ Low	Mask Follows	0	1	1	1

Used in Mode 3 only

If the "mask follows" bit is high, the next control word written to the port must be the mask:

MB_7	MB_6	MB_5	MB_4	MB_3	MB_2	MB_1	MB_0

MB = 0, Monitor bit
MB = 1, Mask bit from being monitored

Also, the interrupt enable flip flop of a port may be set or reset without modifying the rest of the interrupt control word by using the following command:

Int Enable	X	X	X	0	0	1	1

Absolute Maximum Ratings

Temperature Under Bias	Specified operating range
Storage Temperature	−65° C to +150° C
Voltage On Any Pin With Respect To Ground	−0.3 V to +7 V
Power Dissipation	0.6 W

*Comment
Stresses above those listed under "Absolute Maximum Ratings" may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or any other conditions above those indicated in the operational sections of this specification is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.

Note: All AC and DC characteristics remain the same for the military grade parts except I_{CC}.

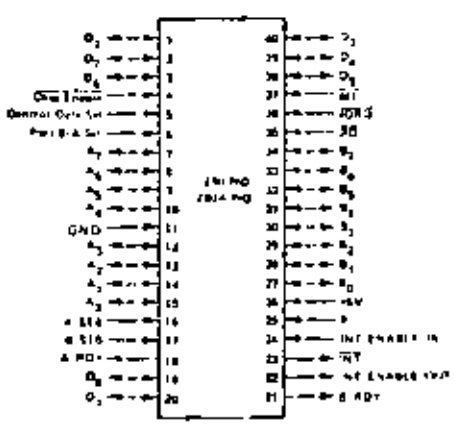
I_{CC} = 130 mA

Z80-PIO and Z80A-PIO D.C. Characteristics

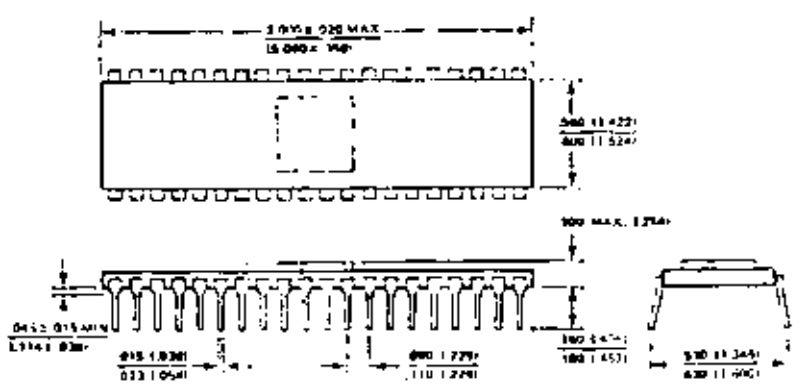
T_A = 0° C to 70° C, V_{CC} = 5 V ± 5% unless otherwise specified

Symbol	Parameter	Min	Max.	Unit	Test Condition
V_{ILC}	Clock Input Low Voltage	−0.3	.45	V	
V_{IHC}	Clock Input High Voltage	V_{CC}−.6	V_{CC}+.3	V	
V_{IL}	Input Low Voltage	−0.3	0.8	V	
V_{IH}	Input High Voltage	2.0	V_{CC}	V	
V_{OL}	Output Low Voltage		0.4	V	I_{OL} = 2.0 mA
V_{OH}	Output High Voltage	2.4		V	I_{OH} = −250 μA
I_{CC}	Power Supply Current		70	mA	
I_{LI}	Input Leakage Current		10	μA	V_{IN} = 0 to V_{CC}
I_{LOH}	Tri State Output Leakage Current in Float		10	μA	V_{OUT} = 2.4 to V_{CC}
I_{LOL}	Tri State Output Leakage Current in Float		−10	μA	V_{OUT} = 0.4 V
I_{LD}	Data Bus Leakage Current in Input Mode		±10	μA	$0 \leq V_{IN} \leq V_{CC}$
I_{OHD}	Darlington Drive Current	−1.5	3.5	mA	V_{OH} = 1.5 V R_{EXT} = 390 Ω Port B Only

Package Configuration



Package Outline



NOTE: Dimensions in parentheses are for metric system (cm).

$T_A = 0°C$ to $70°C$, $V_{CC} = +5V \pm 5\%$, unless otherwise noted

SIGNAL	SYMBOL	PARAMETER	MIN	MAX	UNIT	COMMENTS
Φ	t_c	Clock Period	400	[1]	nsec	
	$t_{w(\Phi H)}$	Clock Pulse Width, Clock High	170	2000	nsec	
	$t_{w(\Phi L)}$	Clock Pulse Width, Clock Low	170	2000	nsec	
	t_r, t_f	Clock Rise and Fall Times		30	nsec	
	t_m	Any Hold Time for Specified Set Up Time	0		nsec	
CS, CE, ETC	$t_{su(CS)}$	Control Signal Set Up Time to Rising Edge of Φ During Read or Write Cycle	280		nsec	
D_0-D_7	$t_{DR(D)}$	Data Output Delay from Falling Edge of $\overline{RD}$		430	nsec	[2]
	$t_{su(D)}$	Data Set Up Time to Rising Edge of Φ During Write or $\overline{M1}$ Cycle	50		nsec	C_L = 50 pF
	$t_{DI(D)}$	Data Output Delay from Falling Edge of $\overline{IORQ}$ During INTA Cycle		340	nsec	[3]
	$t_{f(D)}$	Delay to Floating Bus (Output Buffer Disable Time)		160	nsec	
IEI	$t_{su(IEI)}$	IEI Set Up Time to Falling Edge of $\overline{IORQ}$ During INTA Cycle	140		nsec	
IEO	$t_{dH(IO)}$	IEO Delay Time from Rising Edge of IEI		210	nsec	[5]
	$t_{dL(IO)}$	IEO Delay Time from Falling Edge of IEI		190	nsec	[5], C_L = 50 pF
	$t_{dM(IO)}$	IEO Delay from Falling Edge of $\overline{M1}$ (Interrupt Occurring Just Prior to $\overline{M1}$) See Note A		300	nsec	[5]
$\overline{IORQ}$	$t_{su(IR)}$	$\overline{IORQ}$ Set Up Time to Rising Edge of Φ During Read or Write Cycle	250		nsec	
$\overline{M1}$	$t_{su(M1)}$	$\overline{M1}$ Set Up Time to Rising Edge of Φ During INTA or $\overline{M1}$ Cycle See Note B	210		nsec	
$\overline{RD}$	$t_{su(RD)}$	$\overline{RD}$ Set Up Time to Rising Edge of Φ During Read or $\overline{M1}$ Cycle	240		nsec	
A_0-A_7, B_0-B_7	$t_{S(PD)}$	Port Data Set Up Time to Rising Edge of $\overline{STROBE}$ (Mode 1)	280		nsec	
	$t_{DS(PD)}$	Port Data Output Delay from Falling Edge of $\overline{STROBE}$ (Mode 2)		230	nsec	[5]
	$t_{f(PD)}$	Delay to Floating Port Data Bus from Rising Edge of $\overline{STROBE}$ (Mode 2)		200	nsec	C_L = 50 pF
	$t_{DI(PD)}$	Port Data Stable from Rising Edge of $\overline{IORQ}$ During WR Cycle (Mode 0)		200	nsec	[5]
ASTB, BSTB	$t_{w(ST)}$	Pulse Width, $\overline{STROBE}$	150 [4]		nsec nsec	
INT	$t_{D(IT)}$	INT Delay Time from Rising Edge of $\overline{STROBE}$		490	nsec	
	$t_{D(IT3)}$	INT Delay Time from Data Match During Mode 3 Operation		420	nsec	
ARDY, BRDY	$t_{DH(RY)}$	Ready Response Time from Rising Edge of $\overline{IORQ}$		t_c* 460	nsec	[5] C_L = 50 pF
	$t_{DL(RY)}$	Ready Response Time from Rising Edge of $\overline{STROBE}$		t_c* 400	nsec	[5]

NOTES:

A. $2.5t_c > (N-2)t_{DL(IO)} + t_{DM(IO)} + t_{S(IEI)} + $ TTL Buffer Delay, if any

B. $\overline{M1}$ must be active for a minimum of 2 clock periods to reset the PIO.

[1] $t_c = t_{w(\Phi H)} + t_{w(\Phi L)} + t_r + t_f$

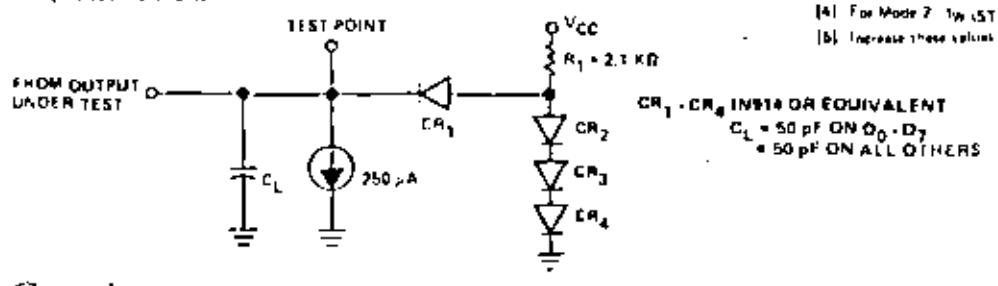
[2] Increase $t_{DR(D)}$ by 10 nsec for each 50 pF increase in loading up to 200 pF max.

[3] Increase $t_{DI(D)}$ by 10 nsec for each 50 pF increase in loading up to 200 pF max.

[4] For Mode 2, $t_{w(ST)} > t_{S(PD)}$

[5] Increase these values by 2 nsec for each 10 pF increase in loading up to 100 pF max.

Output load circuit.



Capacitance

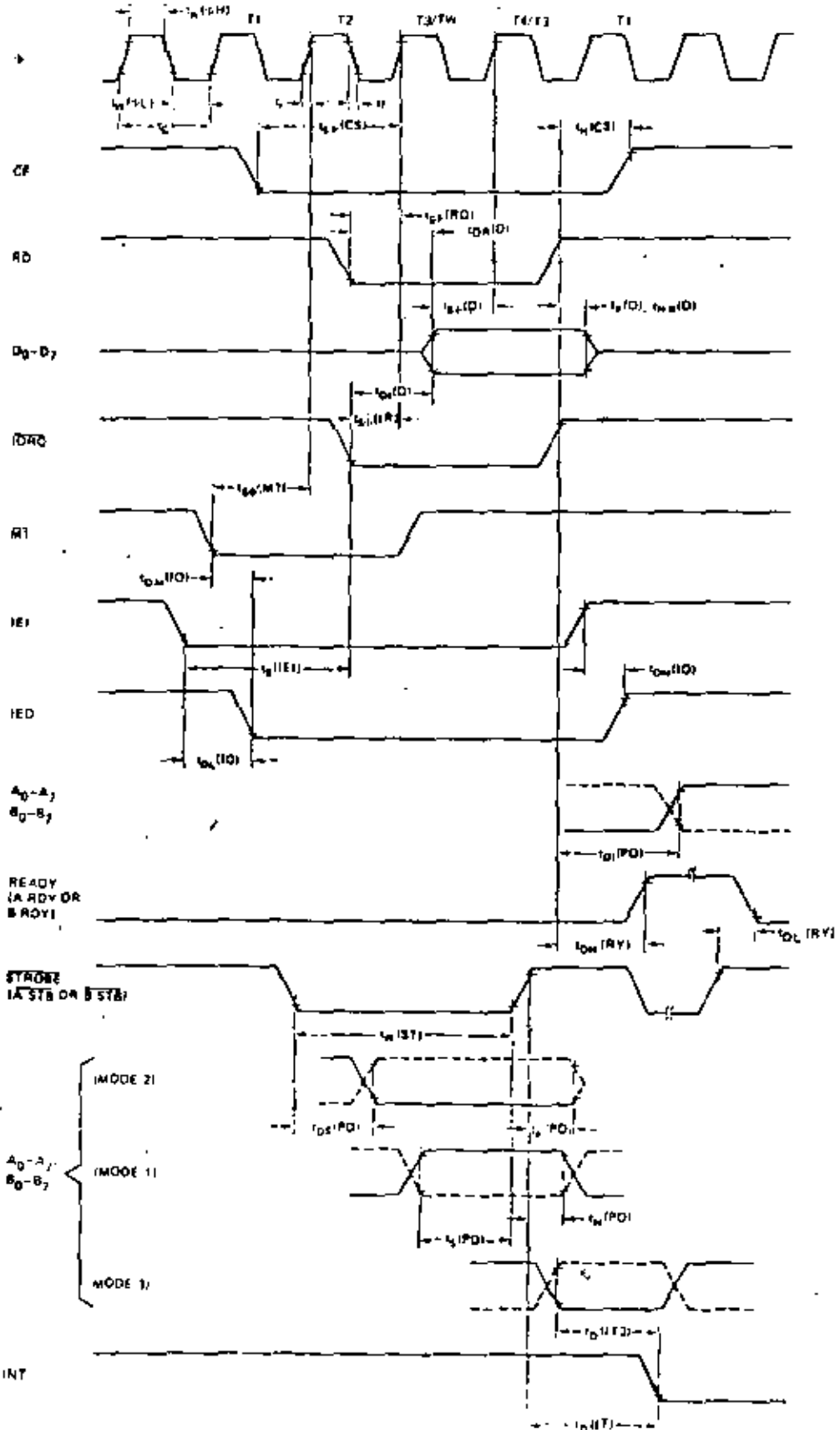
$T_A = 25°C$, f = 1 MHz

Symbol	Parameter	Max.	Unit	Test Condition
C_Φ	Clock Capacitance	10	pF	Unmeasured Pins
C_{IN}	Input Capacitance	5	pF	Returned to Ground
C_{OUT}	Output Capacitance	10	pF	

A.C. Timing Diagram

Timing measurements are made at the following voltages, unless otherwise specified:

	"1"	"0"
CLOCK	Vcc−6	45V
OUTPUT	2.0V	0.8V
INPUT	2.0V	0.8V
FLOAT	ΔV	+0.5V



TA = 0° C to 70° C, Vcc = +5 V ± 5%, unless otherwise noted

SIGNAL	SYMBOL	PARAMETER	MIN	MAX	UNIT	COMMENTS
Φ	t_c	Clock Period	250	[1]	nsec	
	$t_W(\Phi H)$	Clock Pulse Width, Clock High	105	2000	nsec	
	$t_W(\Phi L)$	Clock Pulse Width, Clock Low	105	2000	nsec	
	$t_{r,f}$	Clock Rise and Fall Times		30	nsec	
	t_h	Any Hold Time for Specified Set Up Time	0		nsec	
CS, CE ETC	$t_{S\Phi}(CS)$	Control Signal Set-Up Time to Rising Edge of Φ During Read or Write Cycle	145		nsec	
D_0-D_7	$t_{DR}(D)$	Data Output Delay From Falling Edge of $\overline{RD}$		380	nsec	[2]
	$t_{S\Phi}(D)$	Data Set Up Time to Rising Edge of Φ During Write or $\overline{M1}$ Cycle	50		nsec	C_L = 50 pF
	$t_{DI}(D)$	Data Output Delay from Falling Edge of $\overline{IORQ}$ During INTA Cycle		250	nsec	[3]
	$t_F(D)$	Delay to Floating Bus (Output Buffer Disable Time)		110	nsec	
IEI	$t_S(IEI)$	IEI Set Up Time to Falling Edge of $\overline{IORQ}$ During INTA Cycle	140		nsec	
IEO	$t_{DH}(IO)$	IEO Delay Time from Rising Edge of IEI		160	nsec	[5]
	$t_{DL}(IO)$	IEO Delay Time from Falling Edge of IEI		130	nsec	[5] C_L = 50 pF
	$t_{DM}(IO)$	IEO Delay from Falling Edge of $\overline{M1}$ (Interrupt Occurring Just Prior to $\overline{M1}$). See Note A		190	nsec	[5]
$\overline{IORQ}$	$t_{S\Phi}(IR)$	$\overline{IORQ}$ Set Up Time to Rising Edge of Φ During Read or Write Cycle	115		nsec	
$\overline{M1}$	$t_{S\Phi}(M1)$	$\overline{M1}$ Set Up Time to Rising Edge of Φ During INTA or $\overline{M1}$ Cycle. See Note B	90		nsec	
$\overline{RD}$	$t_{S\Phi}(RD)$	$\overline{RD}$ Set Up Time to Rising Edge of Φ During Read or $\overline{M1}$ Cycle	115		nsec	
A_0-A_7, B_0-B_7	$t_S(PD)$	Port Data Set-Up Time to Rising Edge of $\overline{STROBE}$ (Mode 1)	230		nsec	
	$t_{DS}(PD)$	Port Data Output Delay from Falling Edge of $\overline{STROBE}$ (Mode 2)		210	nsec	[5]
	$t_F(PD)$	Delay to Floating Port Data Bus from Rising Edge of $\overline{STROBE}$ (Mode 2)		180	nsec	C_L = 50 pF
	$t_{DI}(PD)$	Port Data Stable from Rising Edge of $\overline{IORQ}$ During WR Cycle (Mode 0)		180	nsec	[5]
$\overline{ASTB}$, $\overline{BSTB}$	$t_W(ST)$	Pulse Width, $\overline{STROBE}$	150 [4]		nsec nsec	
$\overline{INT}$	$t_D(IT)$	$\overline{INT}$ Delay time from Rising Edge of $\overline{STROBE}$		440	nsec	
	$t_D(IT3)$	$\overline{INT}$ Delay Time from Data Match During Mode 3 Operation		380	nsec	
ARDY, BRDY	$t_{DH}(RY)$	Ready Response Time from Rising Edge of $\overline{IORQ}$		t_c + 410	nsec	[5] C_L = 50 pF
	$t_{DL}(RY)$	Ready Response Time from Rising Edge of $\overline{STROBE}$		t_c + 360	nsec	[5]

NOTES:

A. $2.5\ t_c > (N-2)\ t_{DL}(IO) + t_{DM}(IO) + t_S(IEI)$ + TTL Buffer Delay, if any
B. $\overline{M1}$ must be active for a minimum of 2 clock periods to reset the PIO.

[1] $t_c = t_W(\Phi H) + t_W(\Phi L) + t_r + t_f$
[2] Increase $t_{DR}(D)$ by 10 nsec for each 50 pF increase in loading up to 200 pF max.
[3] Increase $t_{DI}(D)$ by 10 nsec for each 50 pF increase in loading up to 200 pF max.
[4] For Mode 2: $t_W(ST) > t_S(PD)$
[5] Increase these values by 2 nsec for each 10 pF increase in loading up to 100 pF max.

Z-80® SIO
TECHNICAL MANUAL

Z-80 SIO
TECHNICAL MANUAL

Z80-SIO Technical Manual

99

Contents

General Information

The Z80-SIO (Serial Input/Output) is a dual-channel multi-function peripheral component designed to satisfy a wide variety of serial data communications requirements in microcomputer systems. Its basic function is a serial-to-parallel, parallel-to-serial converter/controller, but—within that role—it is configurable by systems software so its "personality" can be optimized for a given serial data communications application.

The Z80-SIO is capable of handling asynchronous and synchronous byte-oriented protocols such as IBM Bisync, and synchronous bit-oriented protocols such as HDLC and IBM SDLC. This versatile device can also be used to support virtually any other serial protocol for applications other than data communications (cassette or floppy disk interfaces, for example).

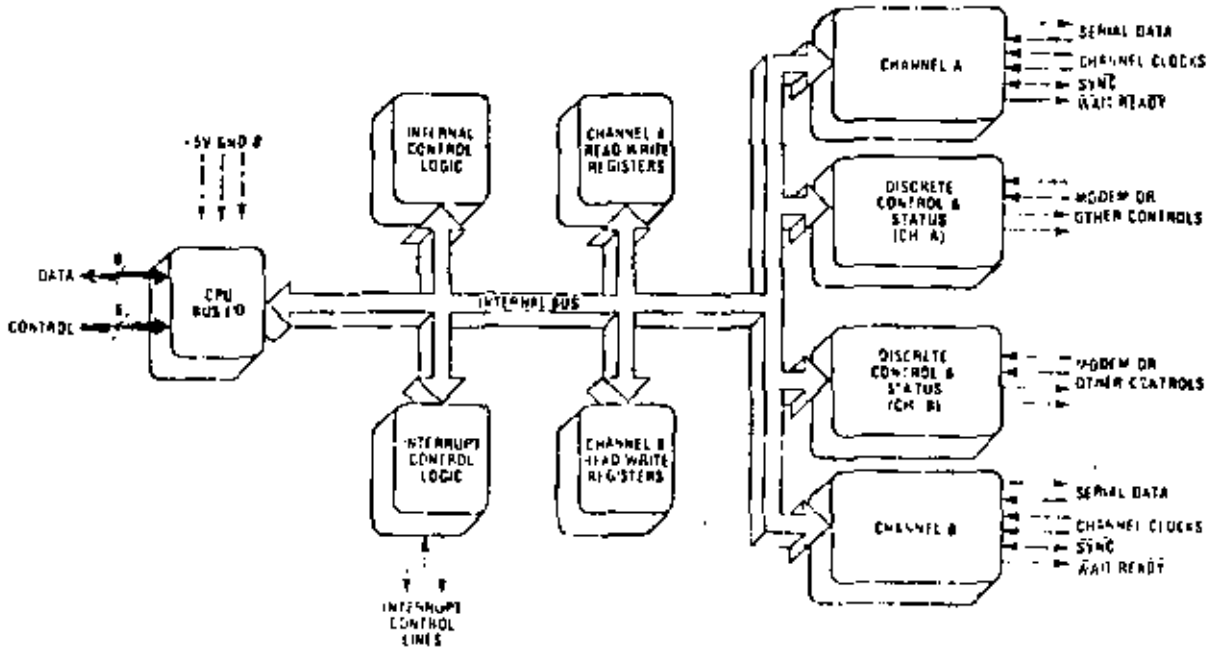
The Z80-SIO can generate and check CRC codes in any synchronous mode and can be programmed to check data integrity in various modes. The device also has facilities for modem controls in both channels. In applications where these controls are not needed, the modem controls can be used for general-purpose I/O.

STRUCTURE

- N-channel silicon-gate depletion-load technology
- 40-pin DIP
- Single 5 V power supply
- Single-phase 5 V clock
- All inputs and outputs TTL compatible

FEATURES

- Two independent full-duplex channels
- Data rates in synchronous or isosynchronous modes:
  - 0-550K bits/second with 2.5 MHz system clock rate
  - 0-880K bits/second with 4.0 MHz system clock rate
- Receiver data registers quadruply buffered; transmitter doubly buffered.
- Asynchronous features:
  - 5, 6, 7 or 8 bits/character
  - 1, 1½ or 2 stop bits
  - Even, odd or no parity
  - ×1, ×16, ×32 and ×64 clock modes
  - Break generation and detection
  - Parity, overrun and framing error detection



Z80-SIO BLOCK DIAGRAM

- Binary synchronous features:
  - Internal or external character synchronization
  - One or two sync characters in separate registers
  - Automatic sync character insertion
  - CRC generation and checking
- HDLC and IBM SDLC features:
  - Abort sequence generation and detection
  - Automatic zero insertion and deletion
  - Automatic flag insertion between messages
  - Address field recognition
  - I-field residue handling
  - Valid receive messages protected from overrun
  - CRC generation and checking
- Separate modem control inputs and outputs for both channels
- CRC-16 or CRC-CCITT block check
- Daisy-chain priority interrupt logic provides automatic interrupt vectoring without external logic
- Modem status can be monitored

Pin Description

D_0-D_7. *System Data Bus* (bidirectional, 3-state). The system data bus transfers data and commands between the CPU and the Z80-SIO. D_0 is the least significant bit.

B/$\overline{A}$. *Channel A Or B Select* (input, High selects Channel B). This input defines which channel is accessed during a data transfer between the CPU and the Z80-SIO. Address bit A_0 from the CPU is often used for the selection function.

C/$\overline{D}$. *Control Or Data Select* (input, High selects Control). This input defines the type of information transfer performed between the CPU and the Z80-SIO. A High at this input during a CPU write to the Z80-SIO causes the information on the data bus to be interpreted as a command for the channel selected by B/$\overline{A}$. A Low at C/$\overline{D}$ means that the information on the data bus is data. Address bit A_1 is often used for this function.

$\overline{CE}$. *Chip Enable* (input, active Low). A Low level at this input enables the Z80-SIO to accept command or data inputs from the CPU during a write cycle, or to transmit data to the CPU during a read cycle.

ϕ. *System Clock* (input). The Z80-SIO uses the standard Z80A System Clock to synchronize internal signals. This is a single-phase clock.

$\overline{M1}$. *Machine Cycle One* (input from Z80-CPU, active Low). When $\overline{M1}$ is active and $\overline{RD}$ is also active, the Z80-CPU is fetching an instruction from memory; when $\overline{M1}$ is active while $\overline{IORQ}$ is active, the Z80-SIO accepts $\overline{M1}$ and $\overline{IORQ}$ as an interrupt acknowledge if the Z80-SIO is the highest priority device that has interrupted the Z80-CPU.

$\overline{IORQ}$. *Input/Output Request* (input from CPU, active Low). $\overline{IORQ}$ is used in conjunction with B/$\overline{A}$, C/$\overline{D}$, $\overline{CE}$ and $\overline{RD}$ to transfer commands and data between the CPU and the Z80-SIO. When $\overline{CE}$, $\overline{RD}$ and $\overline{IORQ}$ are all active,

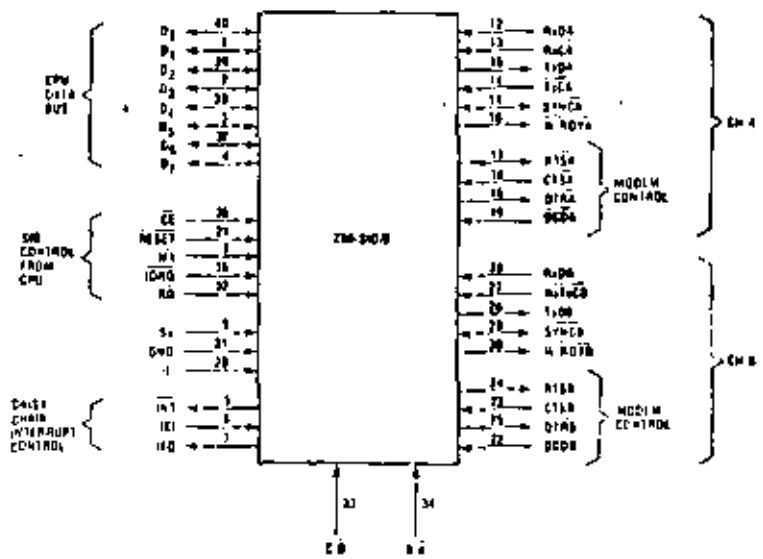


Figure 1. Z80-SIO/0 Pin Configuration

the channel selected by B/$\overline{A}$ transfers data to the CPU (a read operation). When $\overline{CE}$ and $\overline{IORQ}$ are active, but $\overline{RD}$ is inactive, the channel selected by B/$\overline{A}$ is written to by the CPU with either data or control information as specified by C/$\overline{D}$. As mentioned previously, if $\overline{IORQ}$ and $\overline{M1}$ are active simultaneously, the CPU is acknowledging an interrupt and the Z80-SIO automatically places its interrupt vector on the CPU data bus if it is the highest priority device requesting an interrupt.

$\overline{RD}$. *Read Cycle Status.* (input from CPU, active Low). If $\overline{RD}$ is active, a memory or I/O read operation is in progress. $\overline{RD}$ is used with B/$\overline{A}$, $\overline{CE}$ and $\overline{IORQ}$ to transfer data from the Z80-SIO to the CPU.

$\overline{RESET}$, *Reset* (input, active Low). A Low $\overline{RESET}$ disables both receivers and transmitters, forces TxDA and TxDB marking, forces the modem controls High and disables all interrupts. The control registers must be rewritten after the Z80-SIO is reset and before data is transmitted or received.

IEI, *Interrupt Enable In* (input, active High). This signal is used with IEO to form a priority daisy chain when there is more than one interrupt-driven device. A High on this line indicates that no other device of higher priority is being serviced by a CPU interrupt service routine.

IEO. *Interrupt Enable Out* (output, active High). IEO is High only if IEI is High and the CPU is not servicing an interrupt from this Z80-SIO. Thus, this signal blocks lower priority devices from interrupting while a higher priority device is being serviced by its CPU interrupt service routine.

$\overline{INT}$. *Interrupt Request* (output, open drain, active Low). When the Z80-SIO is requesting an interrupt, it pulls $\overline{INT}$ Low.

$\overline{W/RDYA}$, $\overline{W/RDYB}$. *Wait/Ready A, Wait/Ready B* (outputs, open drain when programmed for Wait function, driven High and Low when programmed for Ready function). These dual-purpose outputs may be programmed as Ready lines for a DMA controller or as Wait lines that synchronize the CPU to the Z80-SIO data rate. The reset state is open drain.

$\overline{CTSA}$, $\overline{CTSB}$. *Clear To Send* (inputs, active Low). When programmed as Auto Enables, a Low on these inputs enables the respective transmitter. If not programmed as Auto Enables, these inputs may be programmed as general-purpose inputs. Both inputs are Schmitt-trigger buffered to accommodate slow-risetime inputs. The Z80-SIO detects pulses on these inputs and interrupts the CPU on both logic level transitions. The Schmitt-trigger inputs do not guarantee a specified noise-level margin.

$\overline{DCDA}$, $\overline{DCDB}$. *Data Carrier Detect* (inputs, active Low). These signals are similar to the CTS inputs, except they can be used as receiver enables.

RxDA, RxDB. *Receive Data* (inputs, active High).

TxDA, TxDB. *Transmit Data* (outputs, active High).

$\overline{TxCA}$, $\overline{TxCB}$.* *Transmitter Clocks* (inputs). See the following section on bonding options. The Transmit Clock may be 1, 16, 32 or 64 times the data rate in asynchronous modes. Transmit data is sent on the falling edge of $\overline{TxC}$.

$\overline{RxCA}$, $\overline{RxCB}$.* *Receiver Clocks* (inputs). See the following section on bonding options. The Receive Clock may be 1, 16, 32 or 64 times the data rate in asynchronous modes. Receive data is sampled on the rising edge of $\overline{RxC}$.

*See footnote on next page.

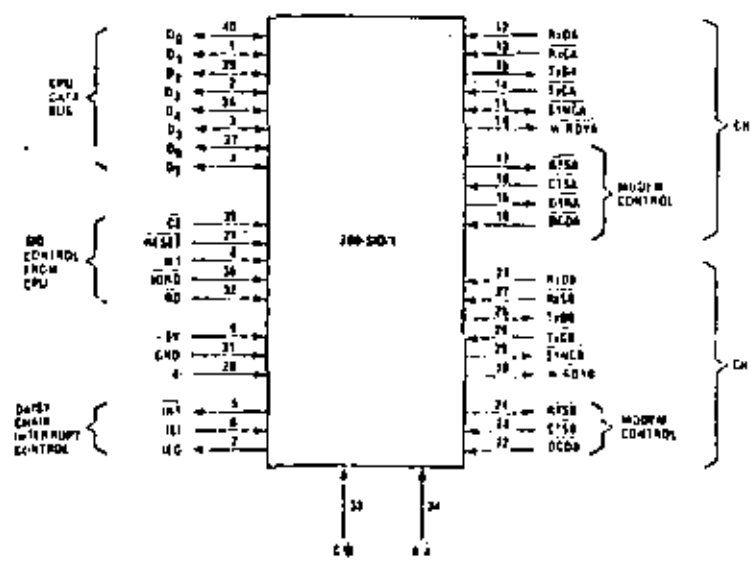


Figure 2. Z80-SIO/1 Pin Configuration

TxCA, TxCB. *Transmitter Clocks* (inputs). See section on bonding options. In asynchronous modes, the transmitter clocks may be 1, 16, 32 or 64 times the data rate. The multiplier for the transmitter and the receiver must be the same. Both the TxC and RxC inputs are Schmitt-trigger buffered for relaxed rise- and fall-time requirements (no noise margin is specified). TxD changes on the falling edge of TxC.

RTSA, RTSB. *Request To Send* (outputs, active Low). When the RTS bit is set, the RTS output goes Low. When the RTS bit is reset in the Asynchronous mode, the output goes High after the transmitter is empty. In Synchronous modes, the RTS pin strictly follows the state of the RTS bit. Both pins can be used as general-purpose outputs.

DTRA, DTRB. *Data Terminal Ready* (outputs, active Low). See note on bonding options. These outputs follow the state programmed into the DTR bit. They can also be programmed as general-purpose outputs.

SYNCA, SYNCB. *Synchronization* (inputs/outputs, active Low). These pins can act either as inputs or outputs. In the Asynchronous Receive mode, they are inputs similar to CTS and DCD. In this mode, the transitions on these lines affect the state of the Sync/Hunt status bits in RR0. In the External Sync mode, these lines also act as inputs. When external synchronization is achieved, SYNC must be driven Low on the second rising edge of RxC after that rising edge of RxC on which the last bit of the sync character was received. In other words, after the sync pattern is detected, the external logic must wait for two full Receive Clock cycles to activate the SYNC input. Once SYNC is forced Low, it is wise to keep it Low until the CPU informs the external sync logic that synchronization has been lost or a new message is about to start. Character assembly begins on the rising edge of RxC that immediately precedes the falling edge of SYNC in the External Sync mode.

In the Internal Synchronization mode (Monosync and Bisync), these pins act as outputs that are active during the part of the receive clock (RxC) cycle in which sync characters are recognized. The sync condition is not latched, so these outputs are active each time a sync pattern is recognized, regardless of character boundaries.

BONDING OPTIONS

The constraints of a 40-pin package make it impossible to bring out the Receive Clock, Transmit Clock, Data Terminal Ready and Sync signals for both channels. Therefore, Channel B must sacrifice a signal or have two signals bonded together. Since user requirements vary, three bondings options are offered:

- Z80-SIO/0 has all four signals, but TxCB and RxCB are bonded together (Fig. 1).
- Z80-SIO/1 sacrifices DTRB and keeps TxCB, RxCB and SYNCB (Fig. 2).
- Z80-SIO/2 sacrifices SYNCB and keeps TxCB, RxCB and DTRB (Fig. 3).

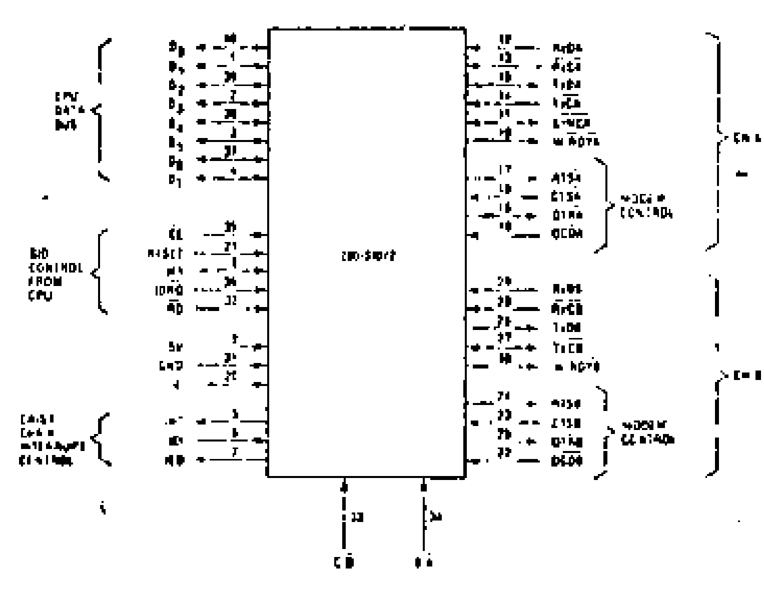


Figure 3. Z80 SIO/2 Pin Configuration

Architecture

The device internal structure includes a Z80-CPU interface, internal control and interrupt logic, and two full-duplex channels. Associated with each channel are read and write registers, and discrete control and status logic that provides the interface to modems or other external devices.

The read and write register group includes five 8-bit control registers, two sync-character registers and two status registers. The interrupt vector is written into an additional 8-bit register (Write Register 2) in Channel B that may be read through Read Register 2 in Channel B. The registers for both channels are designated in the text as follows:

WR0-WR7 — Write Registers 0 through 7
RR0-RR2 — Read Registers 0 through 2

The bit assignment and functional grouping of each register is configured to simplify and organize the programming process. Table 1 illustrates the functions assigned to each read or write register.

WR0	Register pointers, CRC initialize, initialization commands for the various modes, etc.
WR1	Transmit/Receive interrupt and data transfer mode definition.
WR2	Interrupt vector (Channel B only)
WR3	Receive parameters and controls
WR4	Transmit/Receive miscellaneous parameters and modes
WR5	Transmit parameters and controls
WR6	Sync character or SDLC address field
WR7	Sync character or SDLC flag

(a) Write Register Functions

RR0	Transmit/Receive buffer status, interrupt status and external status
RR1	Special Receive Condition status
RR2	Modified interrupt vector (Channel B only)

(b) Read Register Functions

Table 1. Functional Assignments of Read and Write Registers

The logic for both channels provides formats, synchronization and validation for data transferred to and from the channel interface. The modem control inputs Clear to Send (CTS) and Data Carrier Detect (DCD) are monitored by the discrete control logic under program control. All the modem control signals are general purpose in nature and can be used for functions other than modem control.

For automatic interrupt vectoring, the interrupt control logic determines which channel and which device within the channel has the highest priority. Priority is fixed with Channel A assigned a higher priority than Channel B; Receive, Transmit and External/ Status interrupts are prioritized in that order within each channel.

Data Path

The transmit and receive data path for each channel is shown in Figure 4. The receiver has three 8-bit buffer registers in a FIFO arrangement (to provide a 3-byte delay) in addition to the 8-bit receive shift register. This arrangement creates additional time for the CPU to service an interrupt at the beginning of a block of high-speed data. The receive error FIFO stores parity and framing errors and other types of status information for each of the three bytes in the receive data FIFO.

Incoming data is routed through one of several paths depending on the mode and character length. In the Asynchronous mode, serial data is entered in the 3-bit buffer if it has a character length of seven or eight bits, or is entered in the 8-bit receive shift register if it has a length of five or six bits.

In the Synchronous mode, however, the data path is determined by the phase of the receive process currently in operation. A Synchronous Receive operation begins with the receiver in the Hunt phase, during which the receiver searches the incoming data stream for a bit pattern that matches the preprogrammed sync characters (or flags in the SDLC mode). If the device is programmed for Monosync Hunt, a match is made with a single sync character stored in WR7. In Bisync Hunt, a match is made with dual sync characters stored in WR6 and WR7.

In either case the incoming data passes through the receive sync register, and is compared against the programmed sync character in WR6 or WR7. In the Monosync mode, a match between the sync character programmed into WR7 and the character assembled in the receive sync register establishes synchronization.

In the Bisync mode, however, incoming data shifted to the receive shift register while the next eight bits of the message are assembled in the receive sync register. The match between the assembled character in the receive sync registers with the programmed sync character in WR6 and WR7 establishes synchronization. Once synchronization is established, incoming data by-

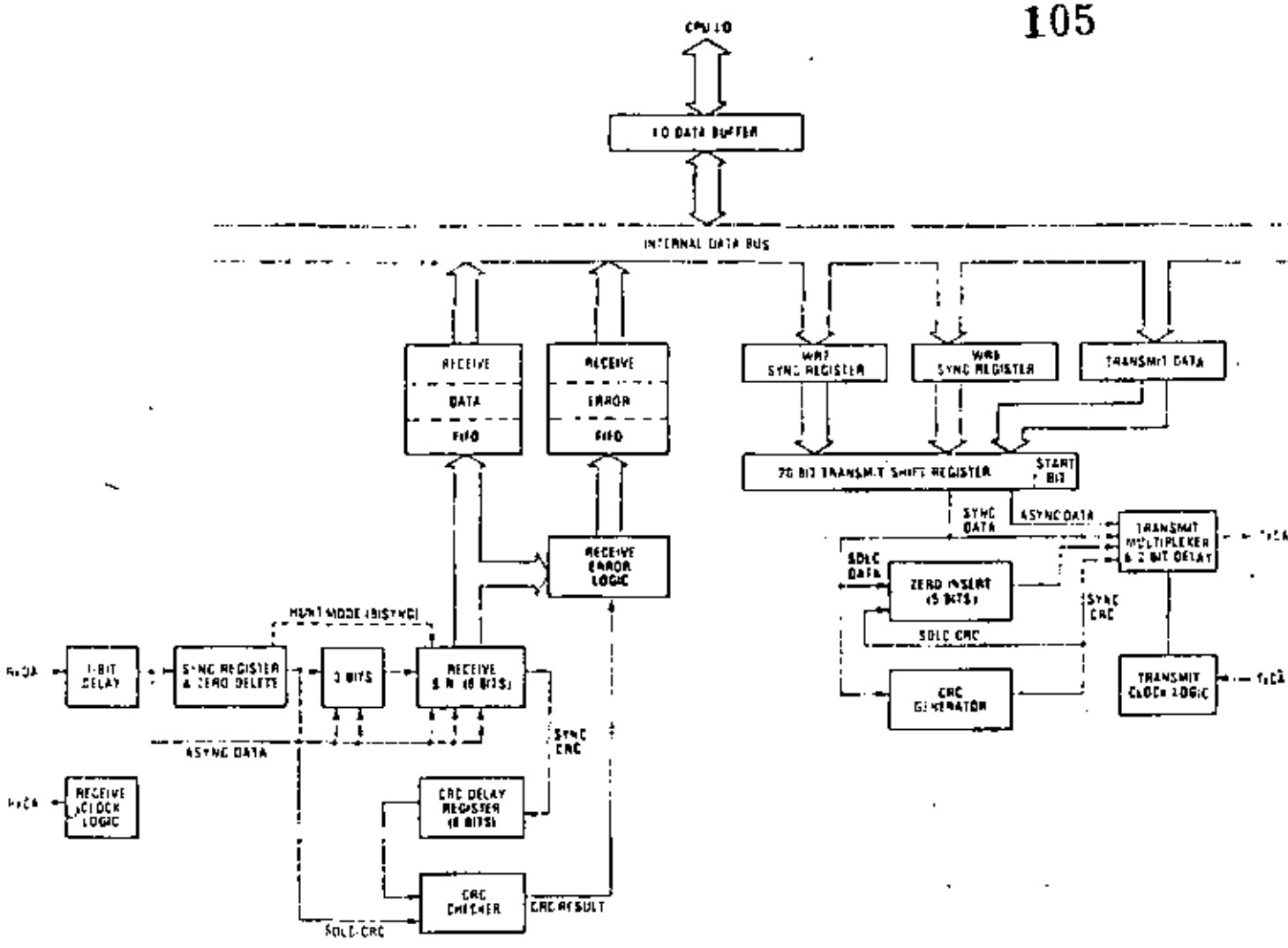


Figure 4. Transmit and Receive Data Path

passes the receive sync register and directly enters the 3-bit buffer.

In the SDLC mode, incoming data first passes through the receive sync register, which continuously monitors the receive data stream and performs zero deletion when indicated. Upon receiving five contiguous 1's, the sixth bit is inspected. If the sixth bit is a 0, it is deleted from the data stream. If the sixth bit is a 1, the seventh bit is inspected. If that bit is a 0, a Flag sequence has been received; if it is a 1, an Abort sequence has been received.

The reformatted data enters the 3-bit buffer and is transferred to the receive shift register. Note that the SDLC receive operation also begins in the Hunt phase, during which the Z80-SIO tries to match the assembled character in the receive shift register with the flag pattern in WR7. Once the first flag character is recognized, all subsequent data is routed through the same path, regardless of character length.

Although the same CRC checker is used for both SDLC and synchronous data, the data path taken for each mode is different. In Bisync protocol, a byte-oriented operation requires that the CPU decide to include the data character in CRC. To allow the CPU ample time to make this decision, the Z80-SIO provides an 8-bit delay for synchronous data. In the SDLC mode, no delay is provided since the Z80-SIO contains logic that determines the bytes on which CRC is calculated.

The transmitter has an 8-bit transmit data register that is loaded from the internal data bus and a 20-bit transmit shift register that can be loaded from WR6, WR7 and the transmit data register. WR6 and WR7 contain sync characters in the Monosync or Bisync modes, or address field (one character long) and flag respectively in the SDLC mode. During Synchronous modes, information contained in WR6 and WR7 is loaded into the transmit shift register at the beginning of the message and, as a time filler, in the middle of the message if a Transmit Underrun condition occurs. In the SDLC mode, the flags are loaded into the transmit shift register at the beginning and end of message.

Asynchronous data in the transmit shift register is formatted with start and stop bits and is shifted out to the transmit multiplexer at the selected clock rate. Synchronous (Monosync or Bisync) data is shifted out to the transmit multiplexer and also to the CRC generator at the ×1 clock rate.

SDLC/HDLC data is shifted out through the zero insertion logic, which is disabled while the flags are being sent. For all other fields (address, control and frame check) a 0 is inserted following five contiguous 1's in the data stream. The CRC generator result for SDLC data is also routed through the zero insertion logic.

Functional Description

The functional capabilities of the Z80-SIO can be described from two different points of view: as a data communications device, it transmits and receives serial data, and meets the requirements of various data communications protocols; as a Z80 family peripheral, it interacts with the Z80-CPU and other Z80 peripheral circuits, and shares their data, address and control busses, as well as being a part of the Z80 interrupt structure. As a peripheral to other microprocessors, the Z80-SIO offers valuable features such as non-vectored interrupts, polling and simple handshake capabilities.

The first part of the following functional description describes the interaction between the CPU and Z80-SIO; the second part introduces its data communications capabilities.

I/O CAPABILITIES

The Z80-SIO offers the choice of Polling, Interrupt (vectored or non-vectored) and Block Transfer modes to transfer data, status and control information to and from the CPU. The Block Transfer mode can be implemented under CPU or DMA control.

Polling. The Polled mode avoids interrupts. Status registers RR0 and RR1 are updated at appropriate times for each function being performed (for example, CRC Error status valid at the end of the message). All the interrupt modes of the Z80-SIO must be disabled to operate the device in a polled environment.

While in its Polling sequence, the CPU examines the status contained in RR0 for each channel; the RR0 status bits serve as an acknowledge to the Poll inquiry. The two RR0 status bits D_0 and D_2 indicate that a receive or transmit data transfer is needed. The status also indicates Error or other special status conditions (see "Z80-SIO Programming"). The Special Receive Condition status contained in RR1 does not have to be read in a Polling sequence because the status bits in RR1 are accompanied by a Receive Character Available status in RR0.

Interrupts. The Z80-SIO offers an elaborate interrupt scheme to provide fast interrupt response in real-time applications. As mentioned earlier, Channel B registers WR2 and RR2 contain the interrupt vector that points to an interrupt service routine in the memory. To serve operations in both channels and to eliminate the necessity of writing a status analysis routine, the Z80-SIO can modify the interrupt vector in RR2 so it points directly to one of eight interrupt service routines. This is done under program control by setting a program bit (WR1, D_2) in Channel B called "Status Affects Vector." When this bit is set, the interrupt vector in WR2 is modified according to the assigned priority of the various interrupting conditions. The table in the Write Register 1 description (Z80-SIO Programming section) shows the modification details.

Transmit interrupts, Receive interrupts and External/Status interrupts are the main sources of interrupts (Figure 5). Each interrupt source is enabled under program control with Channel A having a higher priority than Channel B, and with Receiver, Transmit and External/Status interrupts prioritized in that order within each channel. When the Transmit interrupt is enabled, the CPU is interrupted by the transmit buffer *becoming* empty. (This implies that the transmitter must have had a data character written into it so it can become empty.) When enabled, the receiver can interrupt the CPU in one of three ways:

- Interrupt on first receive character
- Interrupt on all receive characters
- Interrupt on a Special Receive condition

Interrupt On First Character is typically used with the Block Transfer mode. Interrupt On All Receive Characters has the option of modifying the interrupt vector in the event of a parity error. The Special Receive Condition interrupt can occur on a character or message basis (End Of Frame interrupt in SDLC, for example). The Special Receive condition can cause an interrupt only if the Interrupt On First Receive Character or Interrupt On All Receive Characters mode is selected. In Interrupt On First Receive Character, an interrupt can occur from Special Receive conditions (except Parity Error) after the first receive character interrupt (example: Receive Overrun interrupt).

The main function of the External/Status interrupt is to monitor the signal transitions of the $\overline{CTS}$, $\overline{DCD}$ and $\overline{SYNC}$ pins; however, an External/Status interrupt is also caused by a Transmit Underrun condition or by the detection of a Break (Asynchronous mode) or Abort (SDLC mode) sequence in the data stream. The interrupt caused by the Break/Abort sequence has a special feature that allows the Z80-SIO to interrupt when the Break/Abort sequence is detected or terminated. This feature facilitates the proper termination of the current message, correct initialization of the next message, and the accurate timing of the Break/Abort condition in external logic.

CPU/DMA Block Transfer. The Z80-SIO provides a Block Transfer mode to accommodate CPU block transfer functions and DMA controllers (Z80-DMA or other designs). The Block Transfer mode uses the WAIT/READY output in conjunction with the Wait/Ready bits of Write Register 1. The WAIT/READY output can be defined under software control as a WAIT line in the CPU Block Transfer mode or as a READY line in the DMA Block Transfer mode.

To a DMA controller, the Z80-SIO READY output indicates that the Z80-SIO is ready to transfer data to or from memory. To the CPU, the WAIT output indicates that the Z80-SIO is not ready to transfer data, thereby requesting the CPU to extend the I/O cycle. The programming of bits 5, 6 and 7 of Write Register 1 and the logic states of the WAIT/READY line are defined in the Write Register 1 description (Z80-SIO Programming section.)

DATA COMMUNICATIONS CAPABILITIES

In addition to the I/O capabilities previously discussed, the Z80-SIO provides two independent full-duplex channels as well as Asynchronous, Synchronous and SDLC (HDLC) operational modes. These modes facilitate the implementation of commonly used data communications protocols.

The specific features of these modes are described in the following sections. To preserve the independence and completeness of each section, some information common to all modes is repeated.

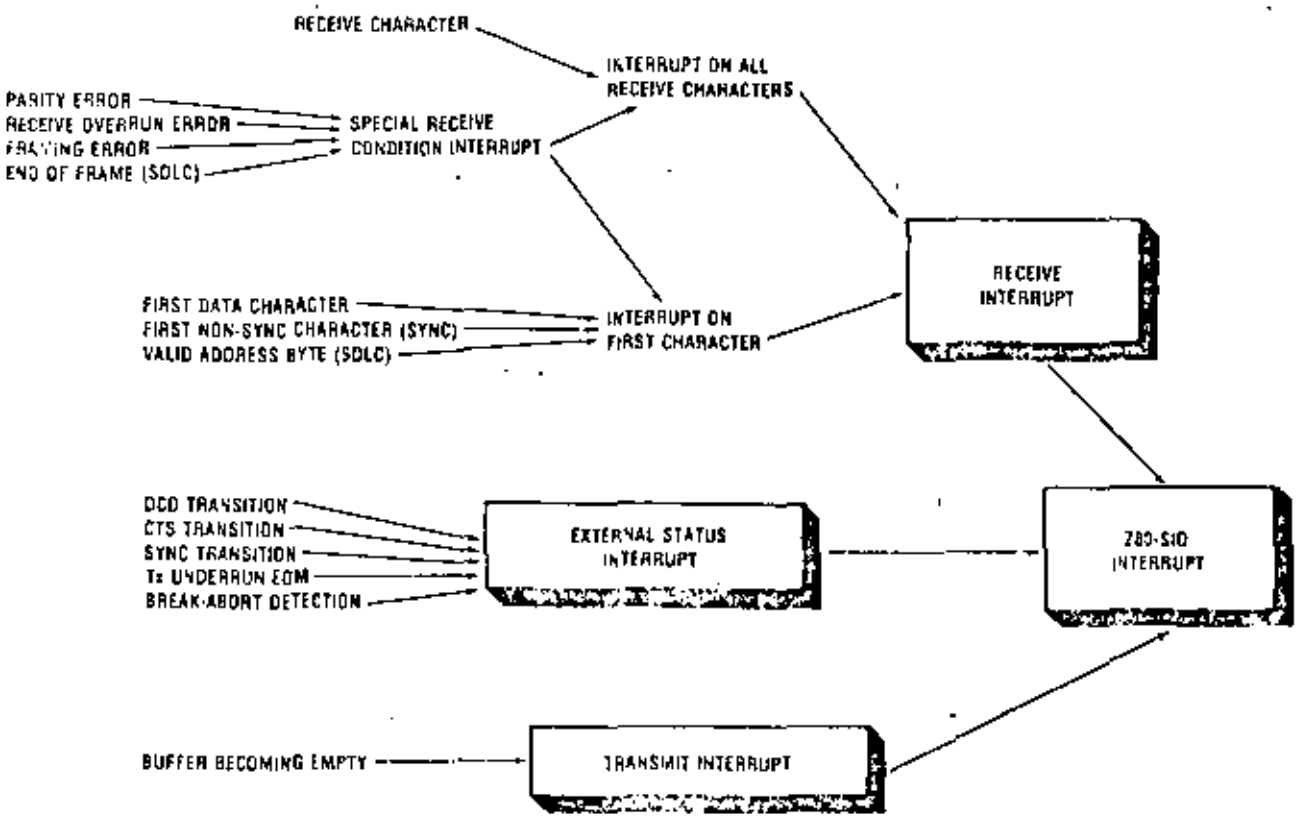


Figure 5. Interrupt Structure

Asynchronous Operation

To receive or transmit data in the Asynchronous mode, the Z80-SIO must be initialized with the following parameters: character length, clock rate, number of stop bits, even or odd parity, interrupt mode, and receiver or transmitter enable. The parameters are loaded into the appropriate write registers by the system program. WR4 parameters must be issued before WR1, WR3 and WR5 parameters or commands.

If the data is transmitted over a modem or RS232C interface, the REQUEST TO SEND (RTS) and DATA TERMINAL READY (DTR) outputs must be set along with the Transmit Enable bit. Transmission cannot begin until the Transmit Enable bit is set.

The Auto Enables feature allows the programmer to send the first data character of the message to the Z80-SIO without waiting for CTS. If the Auto Enables bit is set, the Z80-SIO will wait for the CTS pin to go Low before it begins data transmission. CTS, DCD and SYNC are general-purpose I/O lines that may be used for functions other than their labeled purposes. If CTS is used for another purpose, the Auto Enables Bit must be programmed to 0.

Figure 6 illustrates asynchronous message formats; Table 2 shows WR3, WR4 and WR5 with bits set to indicate the applicable modes, parameters and commands in asynchronous modes. WR2 (Channel B only) stores the interrupt vector; WR1 defines the interrupt modes and data transfer modes. WR6 and WR7 are not used in asynchronous modes. Table 3 shows the typical program steps that implement a full-duplex receive/transmit operation in either channel.

Asynchronous Transmit

The Transmit Data output (TxD) is held marking (High) when the transmitter has no data to send. Under program control, the Send Break (WR5, D_4) command can be issued to hold TxD spacing (Low) until the command is cleared.

The Z80-SIO automatically adds the start bit, the programmed parity bit (odd, even or no parity) and the programmed number of stop bits to the data character to be transmitted. When the character length is six or seven bits, the unused bits are automatically ignored by the Z80-SIO. If the character length is five bits or less, refer to the table in the Write Register 5 description (Z80-SIO Programming section) for the data format.

Serial data is shifted from TxD at a rate equal to 1, 1/16th, 1/32nd or 1/64th of the clock rate supplied to the Transmit Clock input (TxC). Serial data is shifted out on the falling edge of (TxC).

If set, the External/Status Interrupt mode monitors the status of DCD, CTS and SYNC throughout the transmission of the message. If these inputs change for a period of time greater than the minimum specified pulse width, the interrupt is generated. In a transmit operation, this feature is used to monitor the modem control signal CTS.

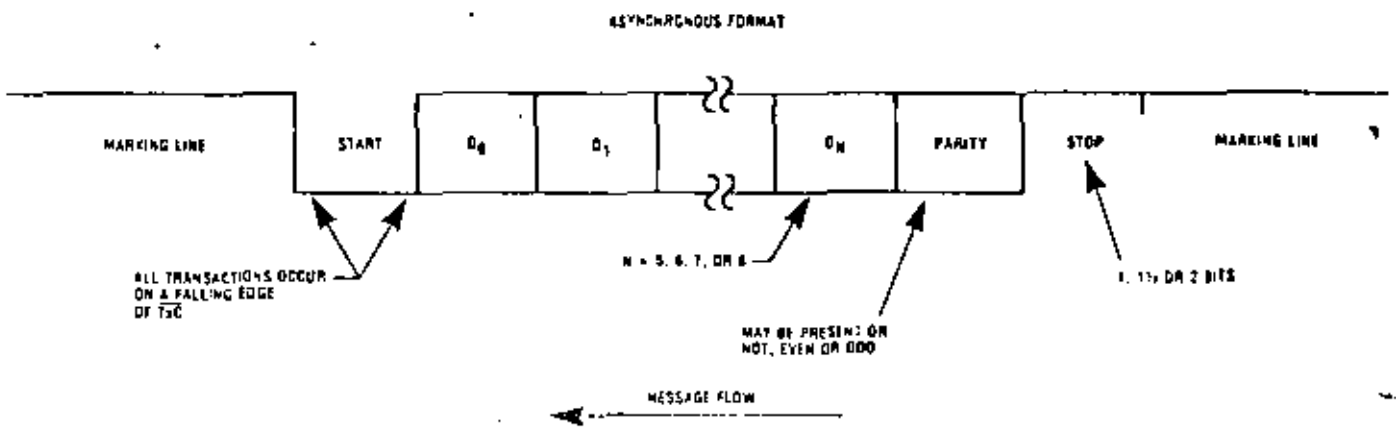


Figure 6. Asynchronous Message Format

An Asynchronous Receive operation begins when the Receive Enable bit is set. If the Auto Enables option is selected, $\overline{DCD}$ must be Low as well. A Low (spacing) condition on the Receive Data input (RxD) indicates a start bit. If this Low persists for at least one-half of a bit time, the start bit is assumed to be valid and the data input is then sampled at mid-bit time until the entire character is assembled. This method of detecting a start bit improves error rejection when noise spikes exist on an otherwise marking line.

If the ×1 clock mode is selected, bit synchronization must be accomplished externally. Receive data is sampled on the rising edge of RxC. The receiver inserts 1's when a character length of other than eight bits is used. If parity is enabled, the parity bit is not stripped from the assembled character for character lengths other than eight bits. For lengths other than eight bits, the receiver assembles a character length of the required number of data bits, plus a parity bit and 1's for any unused bits. For example, the receiver assembles a 5-bit character with the following format: 1 1 P D_4 D_3 D_2 D_1 D_0.

Since the receiver is buffered by three 8-bit registers in addition to the receive shift register, the CPU has enough time to service an interrupt and to accept the data character assembled by the Z80-SIO. The receiver also has three buffers that store error flags for each data character in the receive buffer. These error flags are loaded at the same time as the data characters.

After a character is received, it is checked for the following error conditions:

- When parity is enabled, the Parity Error bit (RR1, D_4) is set whenever the parity bit of the character does not match with the programmed parity. Once this bit is set, it remains set until the Error Reset Command (WR0) is given.
- The Framing Error bit (RR1, D_6) is set if the character is assembled without any stop bits (that is, a Low level detected for a stop bit). Unlike the Parity Error bit, this bit is set (and not latched) only for the character on which it occurred. Detection of framing error adds an additional one-half of a bit time to the character time so the framing error is not interpreted as a new start bit.
- If the CPU fails to read a data character while more than three characters have been received, the Receive Overrun bit (RR1, D_5) is set. When this occurs, the fourth character assembled replaces the third character in the receive buffers. With this arrangement, only the character that has been written over is flagged with the Receive Overrun Error bit. Like Parity Error, this bit can only be reset by the Error Reset command from the CPU. Both the Framing Error and Receive Overrun Error cause an interrupt with the interrupt vector indicating a Special Receive condition (if Status Affects Vector is selected).

Since the Parity Error and Receive Overrun Error flags are latched, the error status that is read reflects an error in the current word in the receive buffer plus any Parity or Overrun Errors received since the last Error Reset command. To keep correspondence between the state of the error buffers and the contents of the receive data buffers, the error status register must be read before the data. This is easily accomplished if vectored

	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
WR3	00 = Rx 5 BITS CHAR 10 = Rx 6 BITS CHAR 01 = Rx 7 BITS CHAR 11 = Rx 8 BITS CHAR		AUTO ENABLES	0	0	0	0	Rx ENABLE
WR4	00 = ×1 CLOCK MODE 01 = ×16 CLOCK MODE 10 = ×32 CLOCK MODE 11 = ×64 CLOCK MODE		0	0	00 = NOT USED 01 = 1 STOP BIT CHAR 10 = 1½ STOP BITS CHAR 11 = 2 STOP BITS CHAR		EVEN $\overline{ODD}$ PARITY	PARITY ENABLE
WR5	DTR	00 = Tx 5 BITS (OR LESS) CHAR 10 = Tx 6 BITS CHAR 01 = Tx 7 BITS CHAR 11 = Tx 8 BITS CHAR		SEND BREAK	Tx ENABLE	0	RTS	0

Table 2. Contents of Write Registers 3, 4 and 5 in Asynchronous Modes

FUNCTION	TYPICAL PROGRAM STEPS		COMMENTS
	REGISTER	*INFORMATION LOADED*	
INITIALIZE	WR0	CHANNEL RESET	Reset SIO
	WR0	POINTER 2	
	WR2	INTERRUPT VECTOR	Channel B only
	WR0	POINTER 4, RESET EXTERNAL STATUS INTERRUPT	
	WR4	ASYNCHRONOUS MODE, PARITY INFORMATION, STOP BITS INFORMATION, CLOCK RATE INFORMATION	Issue parameters
	WR0	POINTER 3	
	WR3	RECEIVE ENABLE, AUTO ENABLES, RECEIVE CHARACTER LENGTH	
	WR0	POINTER 5	
	WR5	REQUEST TO SEND, TRANSMIT ENABLE, TRANSMIT CHARACTER LENGTH, DATA TERMINAL READY	Receive and Transmit both fully initialized. Auto Enables will enable Transmitter if $\overline{CTS}$ is active and Receiver if $\overline{DCD}$ is active.
	WR0	POINTER 1, RESET EXTERNAL STATUS INTERRUPT	
	WR1	TRANSMIT INTERRUPT ENABLE, STATUS AFFECTS VECTOR, INTERRUPT ON ALL RECEIVE CHARACTERS, DISABLE WAIT/READY FUNCTION, EXTERNAL INTERRUPT ENABLE	Transmit Receive interrupt mode selected. External Interrupt monitors the status of the $\overline{CTS}$, $\overline{DCD}$ and $\overline{SYNC}$ inputs and detects the Break sequence. Status Affects Vector in Channel B only.
	TRANSFER FIRST DATA BYTE TO SIO		This data byte must be transferred to SIO before transmit interrupts will occur.
IDLE MODE	EXECUTE HALT INSTRUCTION OR SOME OTHER PROGRAM		Program is waiting for an interrupt from the SIO.
DATA TRANSFER AND ERROR MONITORING	Z80 INTERRUPT ACKNOWLEDGE CYCLE TRANSFERS RR2 TO CPU		When the interrupt occurs, the interrupt vector is modified by: 1. Receive Character Available; 2. Transmit Buffer Empty; 3. External/Status change, and 4. Special Receive condition.
	IF A CHARACTER IS RECEIVED • TRANSFER DATA CHARACTER TO CPU • UPDATE POINTERS AND PARAMETERS • RETURN FROM INTERRUPT		Program control is transferred to one of the eight interrupt service routines
	IF TRANSMITTER BUFFER IS EMPTY • TRANSFER DATA CHARACTER TO SIO • UPDATE POINTERS AND PARAMETERS • RETURN FROM INTERRUPT		
	IF EXTERNAL STATUS CHANGES: • TRANSFER RR0 TO CPU • PERFORM ERROR ROUTINES (INCLUDE BREAK DETECTION) • RETURN FROM INTERRUPT		If used with processors other than the Z80, the modified interrupt vector (RR2) should be returned to the CPU in the Interrupt Acknowledge sequence.
	IF SPECIAL RECEIVE CONDITION OCCURS: • TRANSFER RR1 TO CPU • DO SPECIAL ERROR (E.G. FRAMING ERROR) ROUTINE • RETURN FROM INTERRUPT		
TERMINATION	REDEFINE RECEIVE/TRANSMIT INTERRUPT MODES		When transmit or receive data transfer is complete.
	DISABLE TRANSMIT/RECEIVE MODES		
	UPDATE MODEM CONTROL OUTPUTS (E.G. RTS OFF)		In Transmit, the All Sent status bit indicates transmission is complete.

Table 3. Asynchronous Mode

interrupts are used, because a special interrupt vector is generated for these conditions.

While the External/Status interrupt is enabled, break detection causes an interrupt and the Break Detected status bit (RR0, D_7) is set. The Break Detected interrupt should be handled by issuing the Reset External/Status Interrupt command to the Z80-SIO in response to the first Break Detected interrupt that has a Break status of 1 (RR0, D_7). The Z80-SIO monitors the Receive Data input and waits for the Break sequence to terminate, at which point the Z80-SIO interrupts the CPU with the Break status set to 0. The CPU must again issue the Reset External/Status Interrupt command in its interrupt service routine to reinitialize the break detection logic.

The External/Status interrupt also monitors the status of $\overline{DCD}$. If the $\overline{DCD}$ pin becomes inactive for a period greater than the minimum specified pulse width, an interrupt is generated with the DCD status bit (RR0, D_3) set to 1. Note that the $\overline{DCD}$ input is inverted in the RR0 status register.

If the status is read after the data, the error data for the next word is also included if it has been stacked in the buffer. If operations are performed rapidly enough so the next character is not yet received, the status register remains valid. An exception occurs when the Interrupt On First Character Only mode is selected. A special interrupt in this mode holds the error data and the character itself (even if read from the buffer) until the Error Reset command is issued. This prevents further data from becoming available in the receiver until the Reset command is issued, and allows CPU intervention on the character with the error even if DMA or block transfer techniques are being used.

If Interrupt On Every Character is selected, the interrupt vector is different if there is an error status in RR1. If a Receiver Overrun occurs, the most recent character received is loaded into the buffer; the character preceding it is lost. When the character that has been written over the other characters is read, the Receive Overrun bit is set and the Special Receive Condition vector is returned if Status Affects Vector is enabled.

In a polled environment, the Receive Character Available bit (RR0, D_0) must be monitored so the Z80-CPU can know when to read a character. This bit is automatically reset when the receive buffers are read. To prevent overwriting data in polled operations, the transmit buffer status must be checked before writing into the transmitter. The Transmit Buffer Empty bit is set to 1 whenever the transmit buffer is empty.

Synchronous Operation

Before describing synchronous transmission and reception, the three types of character synchronization—Monosync, Bisync and External Sync—require some explanation. These modes use the ×1 clock for both Transmit and Receive operations. Data is sampled on the rising edge of the Receive Clock input ($\overline{RxC}$). Transmitter data transitions occur on the falling edge of the Transmit Clock input ($\overline{TxC}$).

The differences between Monosync, Bisync and External Sync are in the manner in which initial character synchronization is achieved. The mode of operation must be selected before sync characters are loaded, because the registers are used differently in the various modes. Figure 7 shows the formats for all three of these synchronous modes.

Monosync. In a Receive operation, matching a single sync character (8-bit sync mode) with the programmed sync character stored in WR7 implies character synchronization and enables data transfer.

Bisync. Matching two contiguous sync characters (16-bit sync mode) with the programmed sync characters stored in WR6 and WR7 implies character synchronization. In both the Monosync and Bisync modes, $\overline{SYNC}$ is used as an output, and is active for the part of the receive clock that detects the sync character.

External Sync. In this mode, character synchronization is established externally; $\overline{SYNC}$ is an input that indicates external character synchronization has been achieved. After the sync pattern is detected, the external logic must wait for two full Receive Clock cycles to activate the $\overline{SYNC}$ input. The SYNC input must be held Low until character synchronization is lost. Character assembly begins on the rising edge of $\overline{RxC}$ that precedes the falling edge of $\overline{SYNC}$.

In all cases after a reset, the receiver is in the Hunt phase, during which the Z80-SIO looks for character synchronization. The hunt can begin only when the receiver is enabled, and data transfer can begin only when character synchronization has been achieved. If character synchronization is lost, the Hunt phase can be re-entered by writing a control word with the Enter Hunt Phase bit set (WR3, D_4). In the Transmit mode, the transmitter always sends the programmed number of sync bits (8 or 16). In the Monosync mode, the transmitter transmits from WR6; the receiver compares against WR7.

In the Monosync, Bisync and External Sync modes, assembly of received data continues until the Z80-SIO is reset, or until the receiver is disabled (by command or by $\overline{DCD}$ in the Auto Enables mode), or until the CPU s the Enter Hunt Phase bit.

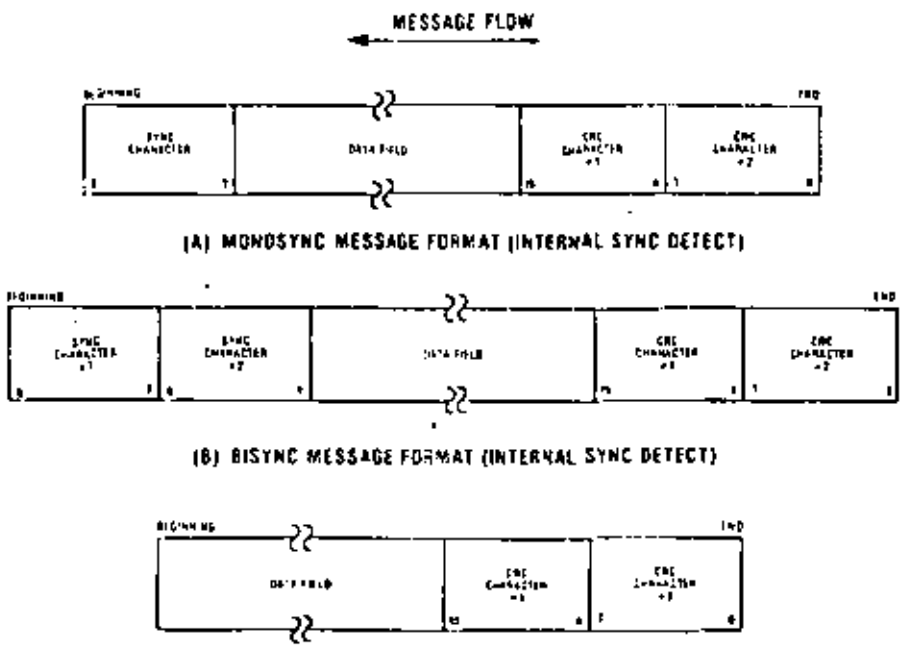


Figure 7. Synchronous Formats

After initial synchronization has been achieved, the operation of the Monosync, Bisync and External Sync modes is quite similar. Any differences are specified in the following text.

Table 4 shows how WR3, WR4 and WR5 are used in synchronous receive and transmit operations. WR0 points to other registers and issues various commands, WR1 defines the interrupt modes, WR2 stores the interrupt vector, and WR6 and WR7 store sync characters. Table 5 illustrates the typical program steps that implement a half-duplex Bisync transmit operation.

Synchronous Transmit

INITIALIZATION

The system program must initialize the transmitter with the following parameters: odd or even parity, ×1 clock mode, 8- or 16-bit sync character(s), CRC polynomial, Transmitter Enables, Request To Send, Data Terminal Ready, interrupt modes and transmit character length. WR4 parameters must be issued before WR1, WR3, WR5, WR6 and WR7 parameters or commands.

One of two polynomials—CRC-16 ($X^{16}+X^{15}+X^2+1$) or SDLC ($X^{16}+X^{12}+X^5+1$)—may be used with synchronous modes. In either case (SDLC mode not selected), the CRC generator and checker are reset to all 1's. In the transmit initialization process, the CRC generator is initialized by setting the Reset Transmit CRC Generator command bits (WR0). Both the transmitter and the receiver use the same polynomial.

Transmit Interrupt Enable or Wait/Ready Enable can be selected to transfer the data. The External/Status interrupt mode is used to monitor the status of the $\overline{\text{CLEAR TO SEND}}$ input as well as the Transmit Underrun/EOM latch. Optionally, the Auto Enables feature can be used to enable the transmitter when $\overline{\text{CTS}}$ is active. The first data transfer to the Z80-SIO can begin when the External/Status interrupt occurs (CTS status bit set) or immediately following the Transmit Enable command (if the Auto Enables modes is set).

Transmit data is held marking after reset or if the transmitter is not enabled. Break may be programmed to generate a spacing line that begins as soon as the Send Break bit is set. With the transmitter fully initialized and enabled, the default condition is continuous transmission of the 8- or 16-bit sync character.

DATA TRANSFER AND STATUS MONITORING

In this phase, there are several combinations of interrupts and Wait/Ready.

Data Transfer Using Interrupts. If the Transmit Interrupt Enable bit (WR1, D_1) is set, an interrupt is generated each time the transmit buffer becomes empty. The interrupt can be satisfied either by writing another character into the transmitter or by resetting the Transmitter Interrupt Pending latch with a Reset Transmitter Pending command (WR0, CMD_5). If the interrupt is satisfied with this command and nothing more is written into the transmitter, there can be no further Transmit Buffer Empty interrupts, because it is the process of the buffer becoming empty that causes the interrupts and the buffer cannot become empty when it is already empty. This situation does cause a Transmit Underrun condition, which is explained in the "Bisync Transmit Underrun" section.

	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
WR3	00 = Rx 5 BITS/CHAR 10 = Rx 6 BITS/CHAR 01 = Rx 7 BITS/CHAR 11 = Rx 8 BITS/CHAR		AUTO ENABLES	ENTER HUNT MODE	Rx CRC ENABLE	0	SYNC CHAR LOAD INHIBIT	RX ENABLE
WR4	0	0	00 = 8-BIT SYNC CHAR 01 = 16-BIT SYNC CHAR 10 = SDLC MODE 11 = EXT SYNC MODE		0	0 SELECTS SYNC MODES	EVEN $\overline{\text{ODD}}$ PARITY	PARITY ENABLE
WR5	DTR	00 = Tx 5 BITS (OR LESS)/CHAR 10 = Tx 6 BITS/CHAR 01 = Tx 7 BITS/CHAR 11 = Tx 8 BITS/CHAR		SEND BREAK	Tx ENABLE	1 SELECTS CRC-16	RTS	Tx CRC ENABLE

Table 4. Contents of Write Registers 3, 4 and 5 in Synchronous Modes

Data Transfer Using WAIT/READY. To the CPU, the activation of WAIT indicates that the Z80-SIO is not ready to accept data and that the CPU must extend the output cycle. To a DMA controller, READY indicates that the transmit buffer is empty and that the Z80-SIO is ready to accept the next data character. If the data character is not loaded into the Z80-SIO by the time the transmit shift register is empty, the Z80-SIO enters the Transmit Underrun condition.

Bisync Transmit Underrun. In Bisync protocol, filler characters are inserted to maintain synchronization when the transmitter has no data to send (Transmit Underrun condition). The Z80-SIO has two programmable options for solving this situation: it can insert sync characters, or it can send the CRC characters generated so far, followed by sync characters.

These options are under the control of the Reset Transmit Underrun/EOM command in WR0. Following a chip or channel reset, the Transmit Underrun/EOM status bit (RR0, D_6) is in a set condition and allows the insertion of sync characters when there is no data to send. CRC is not calculated on the automatically inserted sync characters. When the CPU detects the end of message, a Reset Transmit Underrun/EOM command can be issued. This allows CRC to be sent when the transmitter has no data. In this case, the Z80-SIO sends CRC, followed by sync characters, to terminate the message.

There is no restriction as to when in the message the Transmit Underrun/EOM bit can be reset. If Reset is issued after the first data character has been loaded the 16-bit CRC is sent and followed by sync characters the first time the transmitter has no data to send. Because of the Transmit Underrun condition, an External/Status interrupt is generated whenever the Transmit Underrun/EOM bit becomes set.

In the case of sync insertion, an interrupt is generated only after the first automatically inserted sync character has been loaded. The status indicates the Transmit Underrun/EOM bit and the Transmit Buffer Empty bit are set.

In the case of CRC insertion, the Transmit Underrun/EOM bit is set and the Transmit Buffer Empty bit is reset while CRC is being sent. When CRC has been completely sent, the Transmit Buffer Empty status bit is set and an interrupt is generated to indicate to the CPU that another message can begin (this interrupt occurs because CRC has been sent and sync has been loaded). If no more messages are to be sent, the program can terminate transmission by resetting RTS, and disabling the transmitter (WR5, D_3).

Pad characters may be sent by setting the Z80-SIO to 8 bits/transmit character and writing FF to the transmitter while CRC is being sent. Alternatively, the sync characters can be redefined as pad characters during this time. The following example is included to clarify this point.

The Z80-SIO interrupts with the Transmit Buffer Empty bit set.

The CPU recognizes that the last character (EOM) of the message has already been sent to the Z80-SIO by examining the internal program status.

To force the Z80-SIO to send CRC, the CPU issues the Reset Transmit Underrun/EOM Latch command (WR0) and satisfies the interrupt with the Reset Transmit Interrupt Pending command. (This command prevents the Z80-SIO from requesting more data.) Because of the transmit underrun caused by this command, the Z80-SIO starts sending CRC. The Z80-SIO also causes an External/Status interrupt with the Transmit Underrun/EOM latch set.

The CPU satisfies this interrupt by loading pad characters into the transmit buffer and issuing the Reset External/Status Interrupt command.

With this sequence, CRC is followed by a pad character instead of a sync character. Note that the Z80-SIO will interrupt with a Transmit Buffer Empty interrupt when CRC is completely sent and that the pad character is loaded into the transmit shift register.

From this point on the CPU can send more pad characters or sync characters.

Bisync CRC Generation. Setting the Transmit CRC enable bit (WR5, D_0) initiates CRC accumulation when the program sends the first data character to the Z80-SIO. Although the Z80-SIO automatically transmits up to two sync characters (16-bit sync), it is wise to send a few more sync characters ahead of the message (before enabling Transmit CRC) to ensure synchronization at the receiving end.

The transmit CRC Enable bit can be changed on the fly any time in the message to include or exclude a particular data character from CRC accumulation. The Transmit CRC Enable bit should be in the desired state when the data character is loaded from the transmit data buffer into the transmit shift register. To ensure this bit is in the proper state, the Transmit CRC Enable bit must be issued before sending the data character to the Z80-SIO.

Transmit Transparent Mode. Transparent mode (Bisync protocol) operation is made possible by the ability to change Transmit CRC Enable on the fly and by the additional capability of inserting 16-bit sync characters. Exclusion of DLE characters from CRC calculation can be achieved by disabling CRC calculation immediately preceding the DLE character transfer to the Z80-SIO.

In the case of a Transmit Underrun condition in the Transparent mode, a pair of DLE-SYN characters are sent. The Z80-SIO can be programmed to send the DLE-SYN sequence by loading a DLE character into WR6 and a sync character into WR7.

Transmit Termination. The Z80-SIO is equipped with a special termination feature that maintains data integrity and validity. If the transmitter is disabled while a data or sync character is being sent, that character is sent as usual, but is followed by a marking line rather than CRC or sync characters. When the transmitter is disabled, a

FUNCTION	TYPICAL PROGRAM STEPS		COMMENTS
	REGISTER	*INFORMATION LOADED*	
	WR0	CHANNEL RESET, RESET TRANSMIT CRC GENERATOR	Reset SIO, initialize CRC generator.
	WR0	POINTER 2	
	WR2	INTERRUPT VECTOR	Channel B only
	WR0	POINTER 3	
	WR3	AUTO ENABLES	Transmission begins only after $\overline{CTS}$ is detected.
	WR0	POINTER 4	
	WR4	PARITY INFORMATION, SYNC MODES INFORMATION, ×1 CLOCK MODE	Issue transmit parameters.
	WR0	POINTER 5	
	WR5	SYNC CHARACTER 1	
	WR0	POINTER 7, RESET EXTERNAL STATUS INTERRUPTS	
INITIALIZE	WR7	SYNC CHARACTER 2	
	WR0	POINTER 1, RESET EXTERNAL STATUS INTERRUPTS	
	WR1	STATUS AFFECTS VECTOR, EXTERNAL INTERRUPT ENABLE, TRANSMIT INTERRUPT ENABLE OR WAIT/READY MODE ENABLE	External Interrupt mode monitors the status of $\overline{CTS}$ and $\overline{DCD}$ input pins as well as the status of Tx Underrun EOM latch. Transmit Interrupt Enable interrupts when the Transmit buffer becomes empty; the Wait Ready mode can be used to transfer data using DMA or CPU Block Transfer.
	WR0	POINTER 5	Status Affects Vector (Channel B only).
	WR5	REQUEST TO SEND, TRANSMIT ENABLE, BISYNC CRC, TRANSMIT CHARACTER LENGTH	Transmit CRC Enable should be set when first non-sync data is sent to Z80-SIO
	FIRST SYNC BYTE TO SIO		Need several sync characters in the beginning of message. Transmitter is fully initialized.
IDLE MODE	EXECUTE HALT INSTRUCTION OR SOME OTHER PROGRAM		Waiting for interrupt or Wait/Ready output to transfer data.
DATA TRANSFER AND STATUS MONITORING	*WHEN INTERRUPT (WAIT/READY) OCCURS:* • INCLUDE/EXCLUDE DATA BYTE FROM CRC ACCUMULATION (IN SIO). • TRANSFER DATA BYTE FROM CPU (OR MEMORY) TO SIO. • DETECT AND SET APPROPRIATE FLAGS FOR CONTROL CHARACTERS (IN CPU) • RESET Tx UNDERRUN EOM LATCH (WR0) IF LAST CHARACTER OF MESSAGE IS DETECTED. • UPDATE POINTERS AND PARAMETERS (CPU). • RETURN FROM INTERRUPT.		Interrupt occurs (Wait/Ready becomes active) when first data byte is being sent. Wait mode allows CPU block transfer from memory to SIO; Ready mode allows DMA block transfer from memory to SIO. The DMA chip can be programmed to capture special control characters (by examining only the bits that specify ASCII or EBCDIC control characters), and interrupt CPU.
	IF ERROR CONDITION OR STATUS CHANGE OCCURS. • TRANSFER RR0 TO CPU. • EXECUTE ERROR ROUTINE. • RETURN FROM INTERRUPT.		Tx Underrun/EOM indicates either transmit underrun (sync character being sent) or end of message (CRC-16 being sent).
	REDEFINE INTERRUPT MODES		
TERMINATION	UPDATE MODEM CONTROL OUTPUTS (E.G., TURN OFF RTS)		Program should gracefully terminate message.
	DISABLE TRANSMIT MODE		

Table 5. Bisync Transmit Mode

character in the buffer remains in the buffer. If the transmitter is disabled while CRC is being sent, the 16-bit transmission is completed, but sync is sent instead of CRC.

A programmed break is effective as soon as it is written into the control register; characters in the transmit buffer and shift register are lost.

In all modes, characters are sent with the least significant bits first. This requires right-hand justification of transmitted data if the word length is less than eight bits. If the word length is five bits or less, the special technique described in the Write Register 5 discussion (Z80-SIO Programming section) must be used for the data format. The states of any unused bits in a data character are irrelevant, except when in the Five Bits Or Less mode.

If the External/Status Interrupt Enable bit is set, transmitter conditions such as "starting to send CRC characters," "starting to send sync characters," and $\overline{CTS}$ changing state cause interrupts that have a unique vector if Status Affects Vector is set. This interrupt mode may be used during block transfers.

All interrupts may be disabled for operation in a Polled mode, or to avoid interrupts at inappropriate times during the execution of a program.

Synchronous Receive

INITIALIZATION

The system program initiates the Synchronous Receive operation with the following parameters: odd or even parity, 8- or 16-bit sync characters, ×1 clock mode, CRC polynomial, receive character length, etc. Sync characters must be loaded into registers WR6 and WR7. The receivers can be enabled only after all receive parameters are set. WR4 parameters must be issued before WR1, WR3, WR5, WR6 and WR7 parameters or commands.

After this is done, the receiver is in the Hunt phase. It remains in this phase until character synchronization is achieved. Note that, under program control, all the leading sync characters of the message can be inhibited from loading the receive buffers by setting the Sync Character Load Inhibit bit in WR3.

DATA TRANSFER AND STATUS MONITORING

After character synchronization is achieved, the assembled characters are transferred to the receive data FIFO. The following four interrupt modes are available to transfer the data and its associated status to the CPU.

No Interrupts Enabled. This mode is used for a purely polled operation or for off-line conditions.

Interrupt On First Character Only. This mode is normally used to start a polling loop or a Block Transfer instruction using $\overline{WAIT/READY}$ to synchronize the CPU or the DMA device to the incoming data rate. In this mode the Z80-SIO interrupts on the first character and then after interrupts only if Special Receive conditions are detected. The mode is reinitialized with the Enable Interrupt On Next Receive Character command to allow the next character received to generate an interrupt. Parity errors do not cause interrupts in this mode, but End Of Frame (SDLC mode) and Receive Overrun do.

If External/Status interrupts are enabled, they may interrupt any time $\overline{DCD}$ changes state.

Interrupt On Every Character. Whenever a character enters the receive buffer, an interrupt is generated. Error and Special Receive conditions generate a special vector if Status Affects Vector is selected. Optionally, a Parity Error may be directed not to generate the special interrupt vector.

Special Receive Condition Interrupts. The Special Receive Condition interrupt can occur only if either the Receive Interrupt On First Character Only or Interrupt On Every Receive Character modes is also set. The Special Receive Condition interrupt is caused by the Receive Overrun error condition. Since the Receive Overrun and Parity error status bits are latched, the error status—when read—reflects an error in the current word in the receive buffer in addition to any Parity or Overrun errors received since the last Error Reset command. These status bits can only be reset by the Error reset command.

CRC Error Checking and Termination. A CRC error check on the receive message can be performed on a per character basis under program control. The Receive CRC Enable bit (WR3, D_3) must be set/reset by the program before the next character is transferred from the receive shift register into the receive buffer register. This ensures proper inclusion or exclusion of data characters in the CRC check.

To allow the CPU ample time to enable or disable the CRC check on a particular character, the Z80-SIO calculates CRC eight bit times after the character has been transferred to the receive buffer. If CRC is enabled before the next character is transferred, CRC is calculated on the transferred character. If CRC is disabled before the time of the next transfer, calculation proceeds on the word in progress, but the word just transferred to the buffer is not included. When these requirements are satisfied, the 3-byte receive data buffer is, in effect, unusable in Bisync operation. CRC may be enabled and disabled as many times as necessary for a given calculation.

In the Monosync, Bisync and External Sync modes, the CRC/Framing Error bit (RR1, D_6) contains the comparison result of the CRC checker 16 bit times (eight bits delay and eight shifts for CRC) after the character has been transferred from the receive shift register to the buffer. The result should be zero, indicating an error-

free transmission. (Note that the result is valid only at the end of CRC calculation. If the result is examined before this time, it usually indicates an error.) The comparison is made with each transfer and is valid only as long as the character remains in the receive FIFO.

Following is an example of the CRC checking operation when four characters (A, B, C and D) are received in that order.

Character A loaded into buffer
Character B loaded into buffer

If CRC is disabled before C is in the buffer, CRC is not calculated on B.

Character C loaded into buffer

After C is loaded, the CRC/Framing Error bit shows the result of the comparison through character A.

117

Character D loaded into buffer

After D is in the buffer, the CRC Error bit shows the result of the comparison through character B whether or not B was included in the CRC calculations.

Due to the serial nature of CRC calculation, the Receive Clock ($\overline{RxC}$) must cycle 16 times (8-bit delay plus 8-bit CRC shift) after the second CRC character has been loaded into the receive buffer, or 20 times (the previous 16 plus 3-bit buffer delay and 1-bit input delay) after the last bit is at the RxD input, before CRC calculation is complete. A faster external clock can be gated into the Receive Clock input to supply the required 16 cycles. The Transmit and Receive Data Path diagram (Figure 4) illustrates the various points of delay in the CRC path.

The typical program steps that implement a half-duplex Bisync Receive mode are illustrated in Table 6. The complete set of command and status bit definitions are explained under "Z80-SIO Programming."

FUNCTION	TYPICAL PROGRAM STEPS		COMMENTS
	REGISTER	*INFORMATION LOADED*	
	WR0	CHANNEL RESET, RESET RECEIVE CRC CHECKER	Reset SIO, initialize Receive CRC checker.
	WR0	POINTER 2	
	WR2	INTERRUPT VECTOR	Channel B only
	WR0	POINTER 4	
	WR4	PARITY INFORMATION, SYNC MODES INFORMATION, x1 CLOCK MODE	Issue receive parameters.
	WR0	POINTER 5, RESET EXTERNAL STATUS INTERRUPT	
	WR5	BISYNC CRC-16, DATA TERMINAL READY	
	WR0	POINTER 3	
INITIALIZE	WR3	SYNC CHARACTER LOAD INHIBIT, RECEIVE CRC ENABLE; ENTER HUNT MODE, AUTO ENABLES, RECEIVE CHARACTER LENGTH	Sync character load inhibit strips all the leading sync characters at the beginning of the message. Auto Enables enables the receiver to accept data only after the DCD input is active.
	WR0	POINTER 6	
	WR6	SYNC CHARACTER 1	
	WR0	POINTER 7	
	WR7	SYNC CHARACTER 2	
	WR0	POINTER 1, RESET EXTERNAL STATUS INTERRUPT	
	WR1	STATUS AFFECTS VECTOR, EXTERNAL INTERRUPT ENABLE, RECEIVE INTERRUPT ON FIRST CHARACTER ONLY	In this interrupt mode, only the first non-sync data character is transferred to the CPU. All subsequent data is transferred on a DMA basis; however Special Receive Condition interrupts will interrupt the CPU. Status Affects Vector used in Channel B only.

Table 6. Bisync Receive Mode

FUNCTION	TYPICAL PROGRAM STEPS	COMMENTS
INITIALIZE (CONTINUED)	WR0 POINTER 3, ENABLE INTERRUPT ON NEXT RECEIVE CHARACTER	Resetting this interrupt mode provides simple program loopback entry to next transaction.
	WR3 RECEIVE ENABLE, SYNC CHARACTER LOAD INHIBIT, ENTER HUNT MODE, AUTO ENABLE, RECEIVE WORD LENGTH	WR3 is reissued to enable receiver. Receive CRC Enable must be set after receiving SOH or STX character.
IDLE MODE	EXECUTE HALT INSTRUCTION OR SOME OTHER PROGRAM	Receive mode is fully initialized and the system is waiting for interrupt on first character.
	WHEN INTERRUPT ON FIRST CHARACTER OCCURS, THE CPU DOES THE FOLLOWING: • TRANSFERS DATA BYTE TO CPU • DETECTS AND SETS APPROPRIATE FLAGS FOR CONTROL CHARACTERS (IN CPU) • INCLUDES/EXCLUDES DATA BYTE IN CRC CHECKER • UPDATES POINTERS AND OTHER PARAMETERS • ENABLES WAIT/READY FOR DMA OPERATION • ENABLES DMA CONTROLLER • RETURNS FROM INTERRUPT	During the Hunt mode, the SIO detects two contiguous characters to establish synchronization. The CPU establishes the DMA mode and all subsequent data characters are transferred by the DMA controller. The controller is also programmed to capture special characters (by examining only the bits that specify ASCII or EBCDIC control characters) and interrupt the CPU upon detection. In response the CPU examines the status or control characters and takes appropriate action (e.g. CRC Enable Update).
DATA TRANSFER AND STATUS MONITORING	*WHEN WAIT/READY BECOMES ACTIVE, THE DMA CONTROLLER DOES THE FOLLOWING:* • TRANSFERS DATA BYTE TO MEMORY • INTERRUPTS CPU IF A SPECIAL CHARACTER IS CAPTURED BY THE DMA CONTROLLER • INTERRUPTS THE CPU IF THE LAST CHARACTER OF THE MESSAGE IS DETECTED	
	FOR MESSAGE TERMINATION, THE CPU DOES THE FOLLOWING: • TRANSFERS RR1 TO THE CPU • SETS ACK/NAK REPLY FLAG BASED ON CRC RESULT • UPDATES POINTERS AND PARAMETERS • RETURNS FROM INTERRUPT	The SIO interrupts the CPU for error condition, and the error routine aborts present message, clears the error condition, and repeats the operation.
TERMINATION	REDEFINE INTERRUPT MODES AND SYNC MODES UPDATE MODEM CONTROLS DISABLES RECEIVE MODE	

Table 6. Bisync Receive Mode (Continued)

SDLC (HDLC) Operation

The Z80-SIO is capable of handling both High-level Synchronous Data Link Control (HDLC) and IBM Synchronous Data Link Control (SDLC) protocols. In the following text, only SDLC is referred to because of the high degree of similarity between SDLC and HDLC.

The SDLC mode is considerably different than Synchronous Bisync protocol because it is bit oriented rather than character oriented and, therefore, can naturally handle transparent operation. Bit orientation makes SDLC a flexible protocol in terms of message length and bit patterns. The Z80-SIO has several built-in features to handle variable message length. Detailed information concerning SDLC protocol can be found in literature published on this subject, such as IBM document GA27-3093.

The SDLC message, called the frame (Figure 8), is opened and closed by flags that are similar to the sync characters in Bisync protocol. The Z80-SIO handles the transmission and recognition of the flag characters that mark the beginning and end of the frame. Note that the Z80-SIO can receive shared-zero flags, but cannot transmit them. The 8-bit address field of an SDLC frame contains the secondary station address. The Z80-SIO has an Address Search mode that recognizes the secondary station address so it can accept or reject the frame.

Since the control field of the SDLC frame is transparent to the Z80-SIO, it is simply transferred to the CPU. The Z80-SIO handles the Frame Check sequence in a manner that simplifies the program by incorporating features such as initializing the CRC generator to all 1's, resetting the CRC checker when the opening flag is detected in the Receive mode, and sending the Frame Check/Flag sequence in the Transmit mode. Controller hardware is simplified by automatic zero insertion and deletion logic contained in the Z80-SIO.

Table 7 shows the contents of WR3, WR4 and WR5 during SDLC Receive and Transmit modes. WR0 points to other registers and issues various commands. WR1 defines the interrupt modes; WR2 stores the interrupt vector. WR7 stores the flag character and WR6 the secondary address.

SDLC Transmit

INITIALIZATION

Like Synchronous operation, the SDLC Transmit mode must be initialized with the following parameters: SDLC mode, SDLC polynomial, Request To Send, Data Terminal Ready, transmit character length, transmit interrupt modes (or Wait/Ready function), Transmit Enable, Auto Enables and External/Status interrupt.

Selecting the SDLC mode and the SDLC polynomial enables the Z80-SIO to initialize the CRC Generator to all 1's. This is accomplished by issuing the Reset Transmit CRC Generator command (WR0). Refer to the Synchronous Operation section for more details on the interrupt modes.

After reset, or when the transmitter is not enabled, the Transmit Data output is held marking. Break may be programmed to generate a spacing line. With the transmitter fully initialized and enabled, continuous flags are transmitted on the Transmit Data output.

An abort sequence may be sent by issuing the Send Abort command (WR0, CMD-). This causes at least eight, but less than fourteen, 1's to be sent before the line reverts to continuous flags. It is possible that the Abort sequence (eight 1's) could follow up to five continuous 1 bits (allowed by the zero insertion logic) and thus cause up to thirteen 1's to be sent. Any data being transmitted and any data in the transmit buffer is lost when an abort is issued.

When required, an extra 0 is automatically inserted when there are five contiguous 1's in the data stream. This does not apply to flags or aborts.

DATA TRANSFER AND STATUS MONITORING

There are several combinations of interrupts and the Wait/Ready function in the SDLC mode.

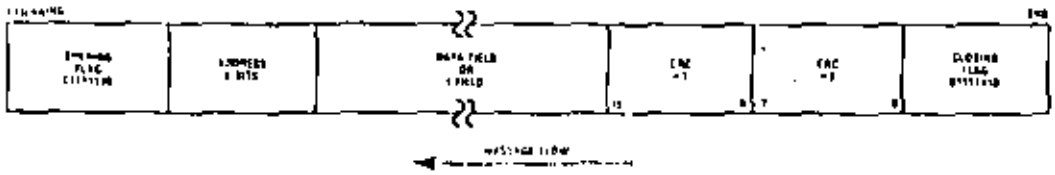


Figure 8. Transmit/Receive SDLC/HDLC Message Format

Data Transfer Using Interrupts. If the Transmit Interrupt Enable bit is set, an interrupt is generated each time the buffer becomes empty. The interrupt may be satisfied either by writing another character into the transmitter or by resetting the Transmit Interrupt Pending latch with a Reset Transmitter Pending command (WR0, CMD5). If the interrupt is satisfied with this command and nothing more is written into the transmitter, there are no further transmitter interrupts. The result is a Transmit Underrun condition. When another character is written and sent out, the transmitter can again become empty and interrupt the CPU. Following the flags in an SDLC operation, the 8-bit address field, control field and information field may be sent to the Z80-SIO using the Transmit Interrupt mode. The Z80-SIO transmits the Frame Check sequence using the Transmit Underrun feature.

When the transmitter is first enabled, it is already empty and obviously cannot then become empty. Therefore, no Transmit Buffer Empty interrupts can occur until after the first data character is written.

When the transmitter is first enabled, it is already empty and cannot then become empty. Therefore, no Transmit Buffer Empty interrupts can occur until after the first data character is written.

Data Transfer Using Wait/Ready. If the Wait/Ready function has been selected, WAIT indicates to the CPU that the Z80-SIO is not ready to accept the data and the CPU must extend the I/O cycle. To a DMA controller, READY indicates that the transmitter buffer is empty and that the Z80-SIO is ready to accept the next character. If the data character is not loaded into the Z80-SIO by the time the transmit shift register is empty, the Z80-SIO enters the Transmit Underrun condition. Address, control and information fields may be transferred to the Z80-SIO with this mode using the Wait/Ready function. The Z80-SIO transmits the Frame Check sequence using the Transmit Underrun feature.

SDLC Transmit Underrun/End Of Message. SDLC-like protocols do not have provisions for fill characters within a message. The Z80-SIO therefore automatically terminates an SDLC frame when the transmit data buffer and output shift register have no more bits to send. does this by first sending the two bytes of CRC and following these with one or more flags. This technique allows very high-speed transmissions under DMA or CPU block I/O control without requiring the CPU to respond quickly to the end of message situation.

The action that the Z80-SIO takes in the underrun situation depends on the state of the Transmit Underrun/EOM command. Following a reset, the Transmit Underrun/EOM status bit is in the set state and prevents the insertion of CRC characters during the time there is no data to send. Consequently, flag characters are sent. The Z80-SIO begins to send the frame as data is written into the transmit buffer. Between the time the first data byte is written and the end of the message, the Reset Transmit Underrun/EOM command must be issued. Thus the Transmit Underrun/EOM status bit is in the reset state at the end of the message (when underrun occurs), which automatically sends the CRC characters. The sending of CRC again sets the Transmit/Underrun/EOM status bit.

Although there is no restriction as to when the Transmit Underrun/EOM bit can be reset within a message, it is usually reset after the first data character (secondary address) is sent to the Z80-SIO. Resetting this bit allows CRC and flags to be sent when there is no data to send which gives additional time to the CPU for recognizing the fault and responding with an abort command. By resetting it early in the message, the entire message has the maximum amount of CPU response time in an unintentional transmit underrun situation.

	BIT 7	BIT 6	BIT 5	BIT 4	BIT 3	BIT 2	BIT 1	BIT 0
WR3	00 = Rx 5 BITS CHAR 10 = Rx 6 BITS CHAR 01 = Rx 7 BITS CHAR 11 = Rx 8 BITS/CHAR		AUTO ENABLES	ENTER HUNT MODE (IF INCOMING DATA NOT NEEDED)	Rx CRC ENABLE	ADDRESS SEARCH MODE	0	Rx ENABLE
WR4	0	0	1 0 SELECTS SDLC MODE		0	0	0	0
WR5	DTR	00 = Tx 5 BITS (OR LESS) CHAR 10 = Tx 6 BITS CHAR 01 = Tx 7 BITS CHAR 11 = Tx 8 BITS CHAR		0	Tx ENABLE	0 SELECTS SDLC CRC	RTS	Tx CRC ENABLE

Table 7. Contents of Write Registers 3, 4 and 5 in SDLC Modes

When the External/Status interrupt is set and while CRC is being sent, the Transmit Underrun/EOM bit is set and the Transmit Buffer Empty bit is reset to indicate that the transmit register is full of CRC data. When CRC has been completely sent, the Transmit Buffer Empty status bit is set and an interrupt is generated to indicate to the CPU that another message can begin. This interrupt occurs because CRC has been sent and the flag has been loaded. If no more messages are to be sent, the program can terminate transmission by resetting $\overline{RTS}$, and disabling the transmitter.

In the SDLC mode, it is good practice to reset the Transmit Underrun/EOM status bit immediately after the first character is sent to the Z80-SIO. When the Transmit Underrun is detected, this ensures that the transmission time is filled by CRC characters, giving the CPU enough time to issue the Send Abort command. This also stops the flags from going on the line prematurely and eliminates the possibility of the receiver accepting the frame as valid data. The situation can happen because it is possible that—at the receiving end—the data pattern immediately preceding the automatic flag insertion could match the CRC checker, giving a false CRC check result. The External/Status interrupt is generated whenever the Transmit Underrun/EOM bit is set because of the Transmit Underrun condition.

The transmit underrun logic provides additional protection against premature flag insertion if the proper response is given to the Z80-SIO by the CPU interrupt service routine. The following example is given to clarify this point:

The Z80-SIO raises an interrupt with the Transmit Buffer Empty status bit set.

The CPU does not respond in time and causes a Transmit Underrun condition.

The Z80-SIO starts sending CRC characters (two bytes).

The CPU eventually satisfies the Transmit Buffer Empty interrupt with a data character that follows the CRC character being transmitted.

The Z80-SIO sets the External/Status interrupt with the Transmit Underrun/EOM status bit set.

The CPU recognizes the Transmit Underrun/EOM status and determines from its internal program status that the interrupt is not for "end of message".

The CPU immediately issues a Send Abort Command (WR0) to the Z80-SIO.

The Z80-SIO sends the Abort sequence by destroying whatever data (CRC, data or flag) is being sent.

This sequence illustrates that the CPU has a protection of 22 minimum and 30 maximum transmit clock cycles.

SDLC CRC Generation. The CRC generator must be reset to all 1's at the beginning of each frame before CRC accumulation can begin. Actual accumulation begins when the program sends the address field (eight bits) to the Z80-SIO. Although the Z80-SIO automatically transmits one flag character following the Transmit Enable, it may be wise to send a few more flag characters ahead of the message to ensure character synchronization at the receiving end. This can be done by externally timing out after enabling the transmitter and before loading the first character.

The Transmit CRC Enable (WR5, D_0) should be enabled prior to sending the address field. In the SDLC mode all the characters between the opening and closing flags are included in CRC accumulation, and the CRC generated in the Z80-SIO transmitter is inverted before it is sent on the line.

Transmit Termination. If the transmitter is disabled while a character is being sent, that character (data or flag) is sent in the normal fashion, but is followed by a marking line rather than CRC or flag characters.

A character in the buffer when the transmitter is disabled remains in the buffer; however, a programmed Abort sequence is effective as soon as it is written into the control register. Characters being transmitted, if any, are lost. In the case of CRC, the 16-bit transmission is completed if the transmitter is disabled; however, flags are sent in place of CRC.

In all modes, characters are sent with the least-significant bits first. This requires right-hand justification of data to be transmitted if the word length is less than eight bits. If the word length is five bits or less, the special technique described in the Write Register 5 section ("Z80-SIO Programming" chapter; "Write Registers" section) must be used.

Since the number of bits/character can be changed on the fly, the data field can be filled with any number of bits. When used in conjunction with the Receiver Residue codes, the Z80-SIO can receive a message that has a variable I-field and retransmit it exactly as received with no previous information about the character structure of the I-field (if any). A change in the number of bits does not affect the character in the process of being shifted out. Characters are sent with the number of bits programmed at the time that the character is loaded from the transmit buffer to the transmitter.

If the External/Status Interrupt Enable is set, transmitter conditions such as "starting to send CRC characters," "starting to send flag characters," and $\overline{CTS}$ changing state cause interrupts that have a unique vector if Status Affects Vector is set. All interrupts can be disabled for operation in a polled mode.

Table 8 shows the typical program steps that imple-

Z80™-CTC
Z80A™-CTC

Technical Manual

TABLE OF CONTENTS

1.0 INTRODUCTION

The Z80-Counter Timer Circuit (CTC) is a programmable component with four independent channels that provide counting and timing functions for microcomputer systems based on the Z80-CPU. The CPU can configure the CTC channels to operate under various modes and conditions as required to interface with a wide range of devices. In most applications, little or no external logic is required. The Z80-CTC utilizes N-channel silicon gate depletion load technology and is packaged in a 28-pin DIP. The Z80-CTC requires only a single 5 volt supply and a one-phase 5 volt clock. Major features of the Z80-CTC include:

- All inputs and outputs fully TTL compatible.
- Each channel may be selected to operate in either Counter Mode or Timer Mode.
- Used in either mode, a CPU-readable Down Counter indicates number of counts-to-go until zero.
- A Time Constant Register can automatically reload the Down Counter at Count Zero in Counter and Timer Mode.
- Selectable positive or negative trigger initiates time operation in Timer Mode. The same input is monitored for event counts in Counter Mode.
- Three channels have Zero Count/Timeout outputs capable of driving Darlington transistors.
- Interrupts may be programmed to occur on the zero count condition in any channel.
- Daisy chain priority interrupt logic included to provide for automatic interrupt vectoring without external logic.

2.0 CTC ARCHITECTURE

2.1 OVERVIEW

A block diagram of the Z80-CTC is shown in figure 2.0-1. The internal structure of the Z80-CTC consists of a Z80-CPU bus interface, Internal Control Logic, four sets of Counter/Timer Channel Logic, and Interrupt Control Logic. The four independent counter/timer channels are identified by sequential numbers from 0 to 3. The CTC has the capability of generating a unique interrupt vector for each separate channel (for automatic vectoring to an interrupt service routine). The 4 channels can be connected into four contiguous slots in the standard Z80 priority chain with channel number 0 having the highest priority. The CPU bus interface logic allows the CTC device to interface directly to the CPU with no other external logic. However, port address decoders and/or line buffers may be required for large systems.

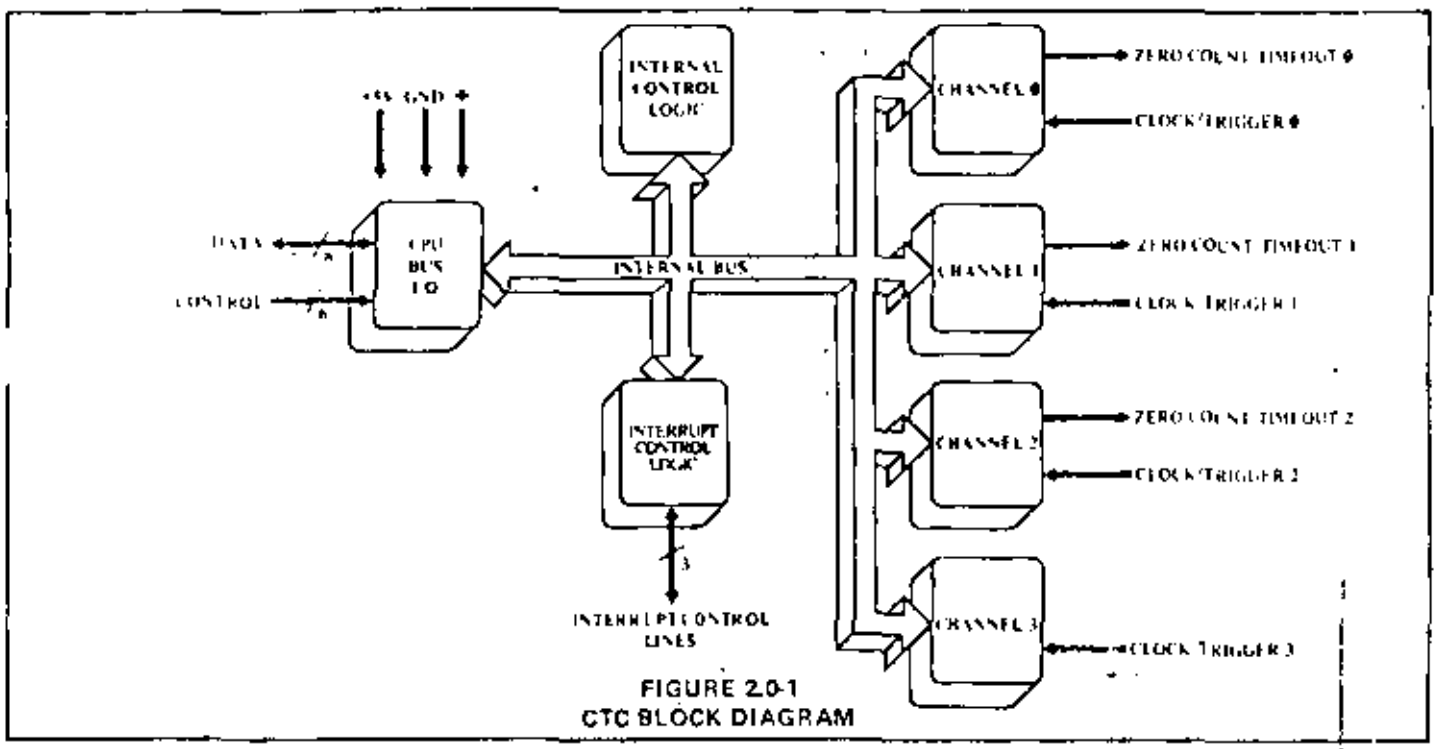


FIGURE 2.0-1
CTC BLOCK DIAGRAM

2.2 STRUCTURE OF CHANNEL LOGIC

The structure of one of the four sets of Counter/Timer Channel Logic is shown in figure 2.0-2. This logic is composed of 2 registers, 2 counters and control logic. The registers are an 8-bit Time Constant Register and an 8-bit Channel Control Register. The counters are an 8-bit CPU-readable Down Counter and an 8-bit Prescaler.

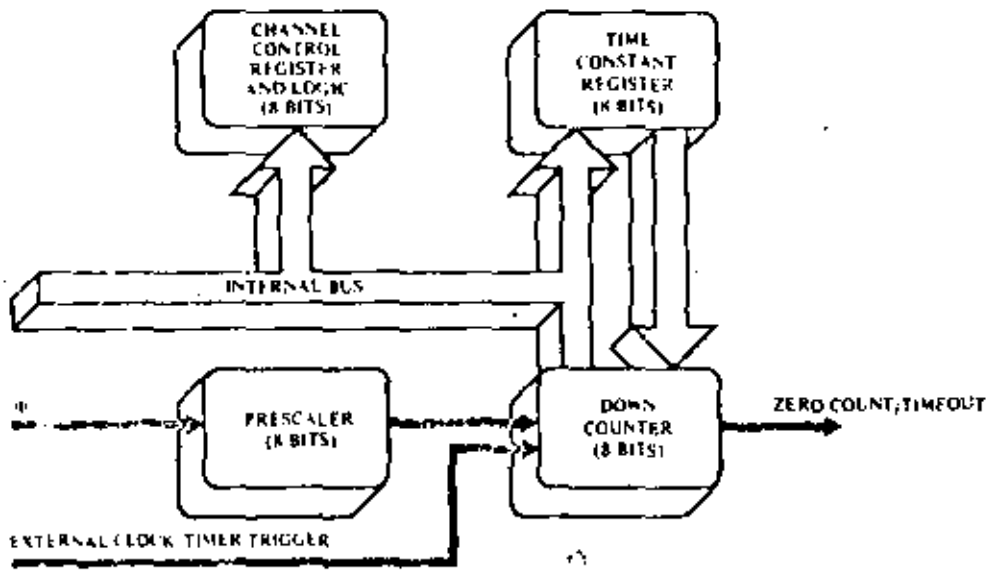


FIGURE 2.0-2
CHANNEL BLOCK DIAGRAM

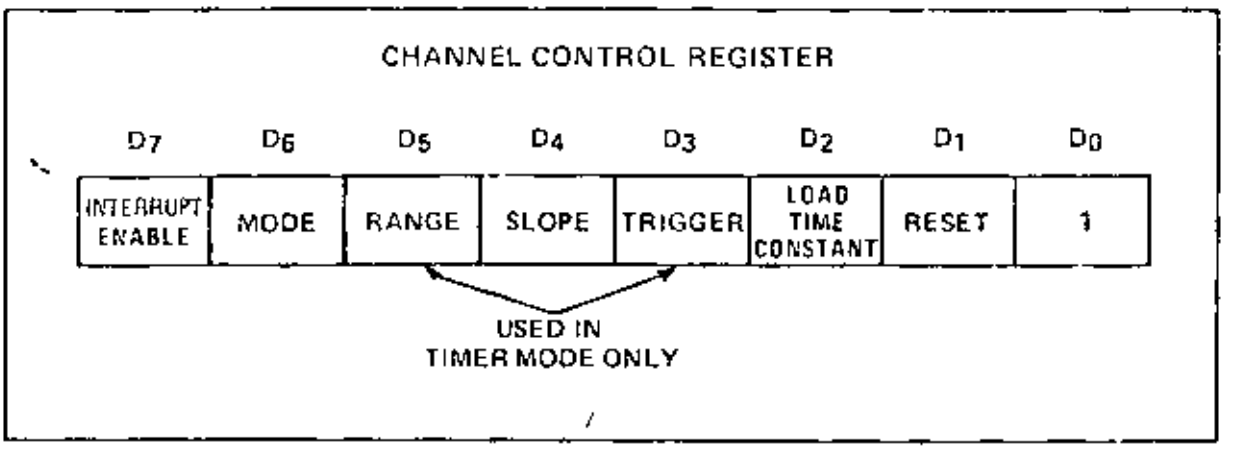
2.2.1 THE CHANNEL CONTROL REGISTER AND LOGIC

The Channel Control Register (8-bit) and Logic is written to by the CPU to select the modes and parameters of the channel. Within the entire CTC device there are four such registers, corresponding to the four Counter/Timer Channels. Which of the four is being written to depends on the encoding of two channel select input pins, CS0 and CS1 (usually attached to A0 and A1 of the CPU address bus). This is illustrated in the truth table below:

	CS1	CS0
Ch 0	0	0
Ch 1	0	1
Ch 2	1	0
Ch 3	1	1

2.2.1 CONTINUED

In the control word written to program each Channel Control Register, bit 0 is always set, and the other 7 bits are programmed to select alternatives on the channel's operating modes and parameters, as shown in the diagram below. (For a more complete discussion see section 4.0: "CTC Operating Modes" and section 5.0: "CTC Programming.")



2.2.2 THE PRESCALER

Used in the Timer Mode only, the Prescaler is an 8-bit device which can be programmed by the CPU via the Channel Control Register to divide its input, the System Clock (φ), by 16 or 256. The output of the Prescaler is then fed as an input to clock the Down Counter, which initially and every time it clocks down to zero, is reloaded automatically with the contents of the Time Constant Register. In effect this again divides the System Clock by an additional factor of the time constant. Every time the Down Counter counts down to zero, its output, Zero Count/Timeout (ZC/TO), is pulsed high.

2.2.3 THE TIME CONSTANT REGISTER

The Time Constant Register is an 8-bit register, used in both Counter Mode and Timer Mode, programmed by the CPU just after the Channel Control Word with an integer time constant value of 1 through 256. This register loads the programmed value into the Down Counter when the CTC is first initialized and reloads the same value into the Down Counter automatically whenever it counts down thereafter to zero. If a new time constant is loaded into the Time Constant Register while a channel is counting or timing, the present down count will be completed before the new time constant is loaded into the Down Counter. (For details of how a time constant is written to a CTC channel, see section 5.0: "CTC Programming.")

2.2.4 THE DOWN COUNTER

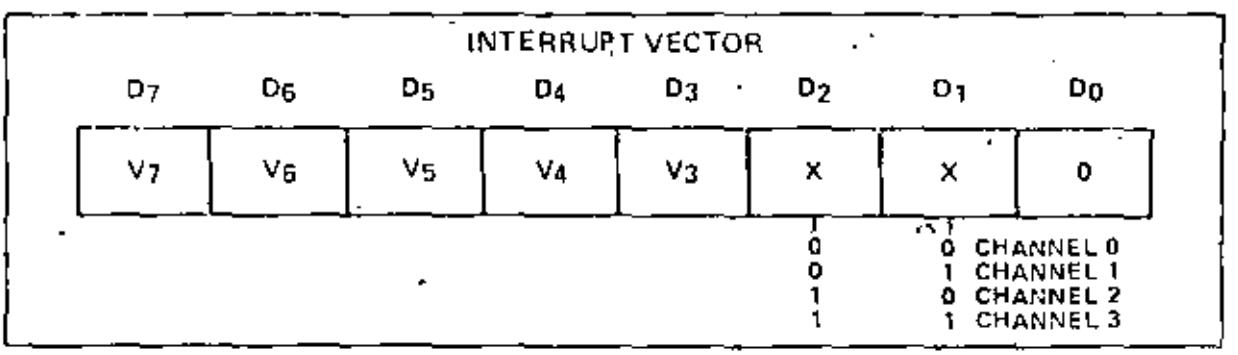
The Down Counter is an 8-bit register, used in both Counter Mode and Timer Mode, loaded initially, and later when it counts down to zero, by the Time Constant Register. The Down Counter is decremented by each external clock edge in the Counter Mode, or in the Timer Mode, by the clock output of the Prescaler. At any time, by performing a simple I/O Read at the port address assigned to the selected CTC channel, the CPU can access the contents of this register and obtain the number of counts-to-zero. Any CTC channel may be programmed to generate an interrupt request sequence each time the zero count is reached.

In channels 0, 1, and 2, when the zero count condition is reached, a signal pulse appears at the corresponding ZC/TO pin. Due to package pin limitations, however, channel 3 does not have this pin and so may be used only in applications where this output pulse is not required.

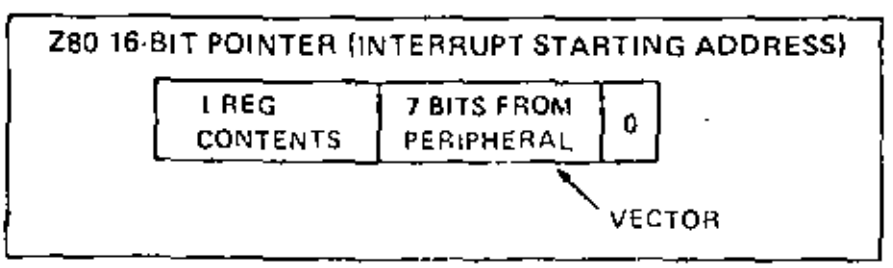
2.3 INTERRUPT CONTROL LOGIC

The Interrupt Control Logic insures that the CTC acts in accordance with Z80 system interrupt protocol for nested priority interrupting and return from interrupt. The priority of any system device is determined by its physical location in a daisy chain configuration. Two signal lines (IEI and IEO) are provided in CTC devices to form this system daisy chain. The device closest to the CPU has the highest priority; within the CTC, interrupt priority is predetermined by channel number, with channel 0 having highest priority down to channel 3 which has the lowest priority. The purpose of a CTC-generated interrupt, as with any other peripheral device, is to force the CPU to execute an interrupt service routine. According to Z80 system interrupt protocol, lower priority devices or channels may not interrupt higher priority devices or channels that have already interrupted and have not had their interrupt service routines completed. However, high priority devices or channels may interrupt the servicing of lower priority devices or channels.

A CTC channel may be programmed to request an interrupt every time its Down Counter reaches a zero count of zero. (To utilize this feature requires that the CPU be programmed for interrupt mode 2.) Some time after the interrupt request, the CPU will send out an interrupt acknowledge, and the CTC's Interrupt Control Logic will determine the highest-priority channel which is requesting an interrupt within the CTC device. Then if the CTC's IEI input is active, indicating that it has priority within the system daisy chain, it will place an 8-bit Interrupt Vector on the system data bus. The high-order 5 bits of this vector will have been written to the CTC earlier as part of the CTC initial programming process; the next two bits will be provided by the CTC's Interrupt Control Logic as a binary code corresponding to the highest-priority channel requesting an interrupt; finally the low-order bit of the vector will always be zero according to a convention described below.



This interrupt vector is used to form a pointer to a location in memory where the address of the interrupt service routine is stored in a table. The vector represents the least significant 8 bits, while the CPU reads the contents of the I register to provide the most significant 8-bits of the 16-bit pointer. The address in memory pointed to will contain the low-order byte, and the next highest address will contain the high-order byte of an address which in turn contains the first opcode of the interrupt service routine. Thus in mode 2, a single 8-bit vector stored in an interrupting CTC can result in an indirect call to any memory location.



There is a Z80 system convention that all addresses in the interrupt service routine table should have their low-order byte in an even location in memory, and their high-order byte in the next highest location in memory, which will always be odd so that the least significant bit of any interrupt vector will always be even. Hence the least significant bit of any interrupt vector will always be zero.

The RETI instruction is used at the end of any interrupt service routine to initialize the daisy chain enable line IEO for proper control of nested priority interrupt handling. The CTC monitors the system data bus and decodes this instruction when it occurs. Thus the CTC channel control logic will know when the CPU has completed servicing an interrupt, without any further communication with the CPU being necessary.

3.0 CTC PIN DESCRIPTION

A diagram of the Z80-CTC pin configuration is shown in figure 3.0-1. This section describes the function of each pin.

D7 - D0

Z80-CPU Data Bus (bi-directional, tri-state)

This bus is used to transfer all data and command words between the Z80-CPU and the Z80-CTC. There are 8 bits on this bus, of which D0 is the least significant.

CS1 – CS0

Channel Select (input, active high)

These pins form a 2-bit binary address code for selecting one of the four independent CTC channels for an I/O Write or Read. (See truth table below.)

	CS1	CS0
Ch 0	0	0
Ch 1	0	1
Ch 2	1	0
Ch 3	1	1

$\overline{CE}$

Chip Enable (input, active low)

A low level on this pin enables the CTC to accept control words, Interrupt Vectors, or time constant data words from the Z80 Data Bus during an I/O Write cycle, or to transmit the contents of the Down Counter to the CPU during an I/O Read cycle. In most applications this signal is decoded from the 8 least significant bits of the address bus for any of the four I/O port addresses that are mapped to the four Counter/Timer Channels.

Clock (Φ)

System Clock (input)

This single-phase clock is used by the CTC to synchronize certain signals internally.

$\overline{M1}$

Machine Cycle One Signal from CPU (input, active low)

When $\overline{M1}$ is active and the $\overline{RD}$ signal is active, the CPU is fetching an instruction from memory. When $\overline{M1}$ is active and the $\overline{IORQ}$ signal is active, the CPU is acknowledging an interrupt, alerting the CTC to place an Interrupt Vector on the Z80 Data Bus if it has daisy chain priority and one of its channels has requested an interrupt.

$\overline{IORQ}$

Input/Output Request from CPU (input, active low)

The $\overline{IORQ}$ signal is used in conjunction with the $\overline{CE}$ and $\overline{RD}$ signals to transfer data and Channel Control Words between the Z80-CPU and the CTC. During a CTC Write Cycle, $\overline{IORQ}$ and $\overline{CE}$ must be true and $\overline{RD}$ false. The CTC does not receive a specific write signal, instead generating its own internally from the inverse of a valid $\overline{RD}$ signal. In a CTC Read Cycle, $\overline{IORQ}$, $\overline{CE}$ and $\overline{RD}$ must be active to place the contents of the Down Counter on the Z80 Data Bus. If $\overline{IORQ}$ and $\overline{M1}$ are both true, the CPU is acknowledging an interrupt request, and the highest-priority interrupting channel will place its Interrupt Vector on the Z80 Data Bus.

$\overline{RD}$

Read Cycle Status from the CPU (input, active low)

The $\overline{RD}$ signal is used in conjunction with the $\overline{IORQ}$ and $\overline{CE}$ signals to transfer data and Channel Control Words between the Z80-CPU and the CTC. During a CTC Write Cycle, $\overline{IORQ}$ and $\overline{CE}$ must be true and $\overline{RD}$ false. The CTC does not receive a specific write signal, instead generating its own internally from the inverse of a valid $\overline{RD}$ signal. In a CTC Read Cycle, $\overline{IORQ}$, $\overline{CE}$ and $\overline{RD}$ must be active to place the contents of the Down Counter on the Z80 Data Bus.

IEI

Interrupt Enable In (input, active high)

This signal is used to help form a system-wide interrupt daisy chain which establishes priorities when more than one peripheral device in the system has interrupting capability. A high level on this pin indicates that no other interrupting devices of higher priority in the daisy chain are being serviced by the Z80-CPU.

IEO

Interrupt Enable Out (output, active high)

The IEO signal, in conjunction with IEI, is used to form a system-wide interrupt priority daisy chain. IEO is high only if IEI is high and the CPU is not servicing an interrupt from any CTC channel. Thus this signal blocks lower priority devices from interrupting while a higher priority interrupting device is being serviced by the CPU.

$\overline{INT}$

Interrupt Request (output, open drain, active low)

This signal goes true when any CTC channel which has been programmed to enable interrupts has a zero count condition in its Down Counter.

$\overline{RESET}$

Reset (input, active low)

This signal stops all channels from counting and resets channel interrupt enable bits in all control registers, thereby disabling CTC-generated interrupts. The ZC/TO and $\overline{INT}$ outputs go to their inactive states, IEO reflects IEI, and the CTC's data bus output drivers go to the high impedance state.

CLK/TRG3 – CLK/TRG0

External Clock/Timer Trigger (input, user-selectable active high or low)

There are four CLK/TRG pins, corresponding to the four independent CTC channels. In the Counter Mode, every active edge on this pin decrements the Down Counter. In the Timer Mode, an active edge on this pin initiates the timing function. The user may select the active edge to be either rising or falling.

ZC/TO2 – ZC/TO0

Zero Count/Timeout (output, active high)

There are three ZC/TO pins, corresponding to CTC channels 2 through 0. (Due to package pin limitations channel 3 has no ZC/TO pin.) In either Counter Mode or Timer Mode, when the Down Counter decrements to zero an active high going pulse appears at this pin.

3.0 CTC PIN DESCRIPTION

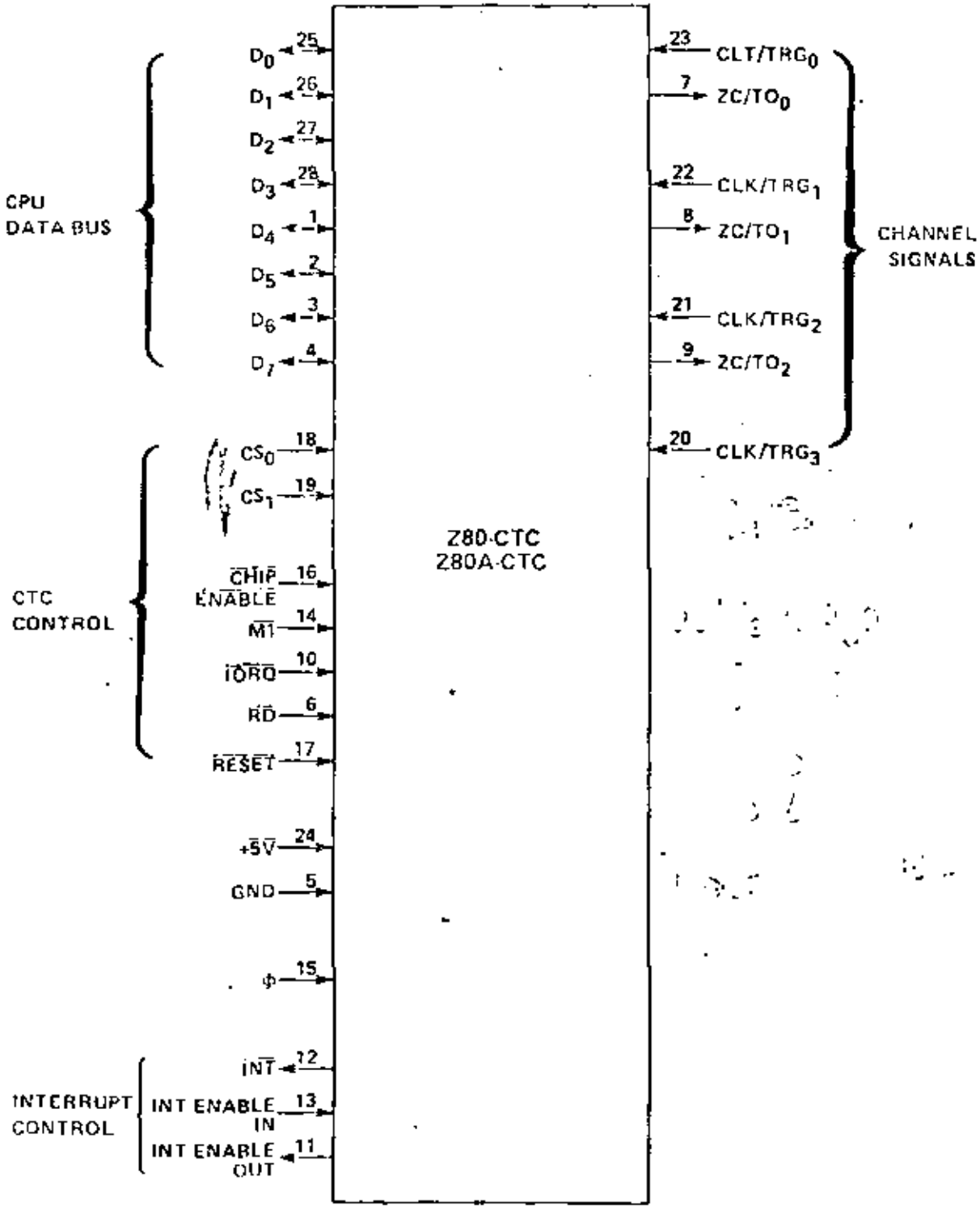


FIGURE 3.0-1
CTC PIN CONFIGURATION

4.0 CTC OPERATING MODES

At power on the Z80-CTC state is undefined. Asserting $\overline{RESET}$ puts the CTC in a known state. Before any channel can begin counting or timing, a Channel Control Word and a time constant data word must be written to the appropriate registers of that channel. Further, if any channel has been programmed to enable interrupts, an Interrupt Vector word must be written to the CTC's Interrupt Control Logic. (For further details, refer to section 5.0: "CTC Programming.") When the CPU has written all of these words to the CTC all active channels will be programmed for immediate operation in either the Counter Mode or the Timer Mode.

4.1 CTC COUNTER MODE

In this mode the CTC counts edges of the CLK/TRG input. The Counter Mode is programmed for a channel when its Channel Control Word is written with bit 6 set. The Channel's External Clock (CLK/TRG) input is monitored for a series of triggering edges; after each, in synchronization with the next rising edge of Φ (the System Clock), the Down Counter (which was initialized with the time constant data word at the start of any sequence of down-counting) is decremented. Although there is no set-up time requirement between the triggering edge of the External Clock and the rising edge of Φ, (Clock), the Down Counter will not be decremented until the following Φ pulse. (See the parameter ts(CK) in section 8.3: "A.C. Characteristics.") A channel's External Clock input is pre-programmed by bit 4 of the Channel Control Word to trigger the decrementing sequence with either a high or a low going edge.

In any of Channels 0, 1, or 2, when the Down Counter is successively decremented from the original time constant until finally it reaches zero, the Zero Count (ZC/TO) output pin for that channel will be pulsed active (high). (However, due to package pin limitations, channel 3 does not have this pin and so may only be used in applications where this output pulse is not required.) Further, if the channel has been so pre-programmed by bit 7 of the Channel Control Word, an interrupt request sequence will be generated. (For more details, see section 7.0: "CTC Interrupt Servicing.")

As the above sequence is proceeding, the zero count condition also results in the automatic reload of the Down Counter with the original time constant data word in the Time Constant Register. There is no interruption in the sequence of continued down-counting. If the Time Constant Register is written to with a new time constant data word while the Down Counter is decrementing, the present count will be completed before the new time constant will be loaded into the Down Counter.

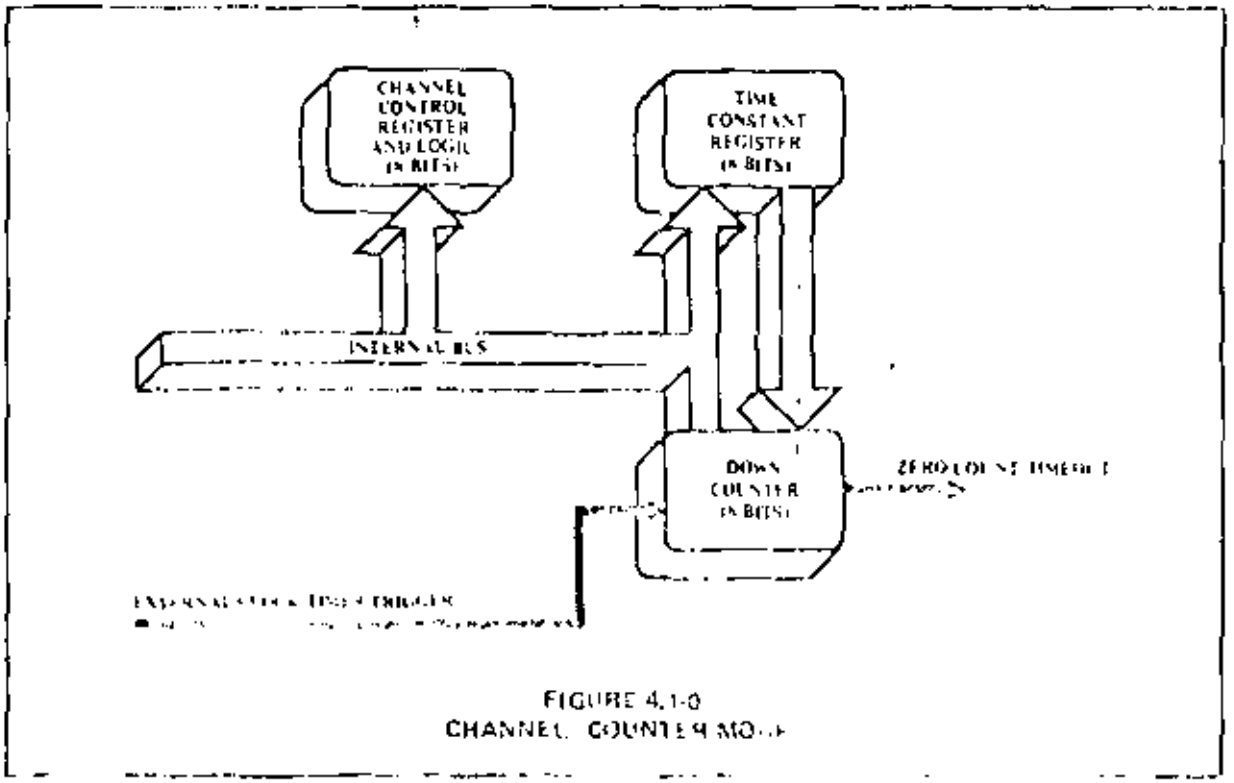


FIGURE 4.1-0
CHANNEL COUNTER MODE

4.2 CTC TIMER MODE

In the Timer Mode, the CTC generates timing intervals that are an integer value of the system clock period. The Timer Mode is programmed for a channel when its Channel Control Word is written with bit 6 reset. The channel then may be used to measure intervals of time based on the System Clock period. The System Clock is fed through two successive counters, the Prescaler and the Down Counter. Depending on the pre-programmed bit 5 in the Channel Control Word, the Prescaler divides the System Clock by a factor of either 16 or 256. The output of the Prescaler is then used as a clock to decrement the Down Counter, which may be pre-programmed with any time constant integer between 1 and 256. As in the Counter Mode, the time constant is automatically reloaded into the Down Counter at each zero-count condition, and counting continues. Also at zero-count, the channel's Time Out (ZC/TO) output (which is the output of the Down Counter) is pulsed, resulting in a uniform pulse train of precise period given by the product

$$t_c * P * TC$$

where t_c is the System Clock period, P is the Prescaler factor of 16 or 256 and TC is the pre-programmed time constant.

Bit 3 of the Channel Control Word is pre-programmed to select whether timing will be automatically initiated, or whether it will be initiated with a triggering edge at the channel's Timer Trigger (CLK/TRG) input. If bit 3 is reset the timer automatically begins operation at the start of the CPU cycle following the I/O Write machine cycle that loads the time constant data word to the channel. If bit 3 is set the timer begins operation on the second succeeding rising edge of Φ after the Timer Trigger edge following the loading of the time constant data word. If no time constant data word is to follow then the timer begins operation on the second succeeding rising edge of Φ after the Timer Trigger edge following the control word write cycle. Bit 4 of the Channel Control Word is pre-programmed to select whether the Timer Trigger will be sensitive to a rising or falling edge. Although there is no set-up requirement between the active edge of the Timer Trigger and the next rising edge of Φ. If the Timer Trigger edge occurs closer than a specified minimum set-up time to the rising edge of Φ, the Down Counter will not begin decrementing until the following rising edge of Φ. (See the parameter ts(TR) in section 8.3: "A.C. Characteristics".)

If bit 7 in the Channel Control Word is set, the zero-count condition in the Down Counter, besides causing a pulse at the channel's Time Out pin, will be used to initiate an interrupt request sequence. (For more details, see section 7.0: "CTC Interrupt Servicing".)

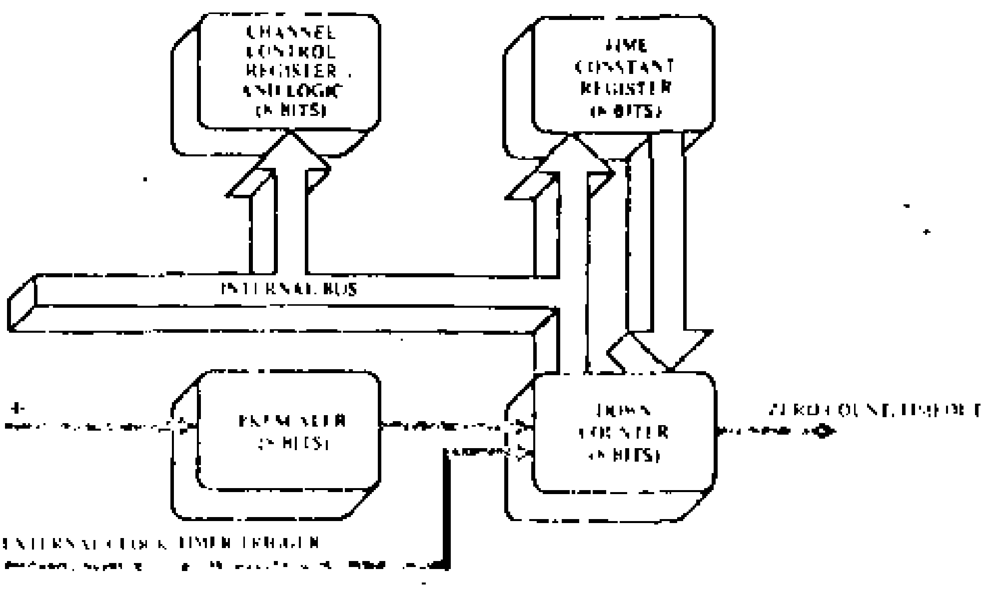


FIGURE 4.2.0
CHANNEL – TIMER MODE

5.0 CTC PROGRAMMING

Before a Z80 CTC channel can begin counting or timing operations, a Channel Control Word and a Time Constant data word must be written to it by the CPU. These words will be stored in the Channel Control Register and the Time Constant Register of that channel. In addition, if any of the four channels have been programmed with bit 7 of their Channel Control Words to enable interrupts, an Interrupt Vector must be written to the appropriate register in the CTC. Due to automatic features in the Interrupt Control Logic, one preprogrammed Interrupt Vector suffices for all four channels.

5.1 LOADING THE CHANNEL CONTROL REGISTER

To load a Channel Control Word, the CPU performs a normal I/O Write sequence to the port address corresponding to the desired CTC channel. Two CTC input pins, namely CS0 and CS1, are used to form a 2-bit binary address to select one of four channels within the device. (For a truth table, see section 2.2.1: "The Channel Control Register and Logic".) In many system architectures, these two input pins are connected to CPU Address Bus lines A0 and A1, respectively, so that the four channels in a CTC device will occupy contiguous I/O port addresses. A word written to a CTC channel will be interpreted as a Channel Control Word, and loaded into the Channel Control Register, its bit 0 is a logic 1. The other seven bits of this word select operating modes and conditions as indicated in the diagram below. Following the diagram the meaning of each bit will be discussed in detail.

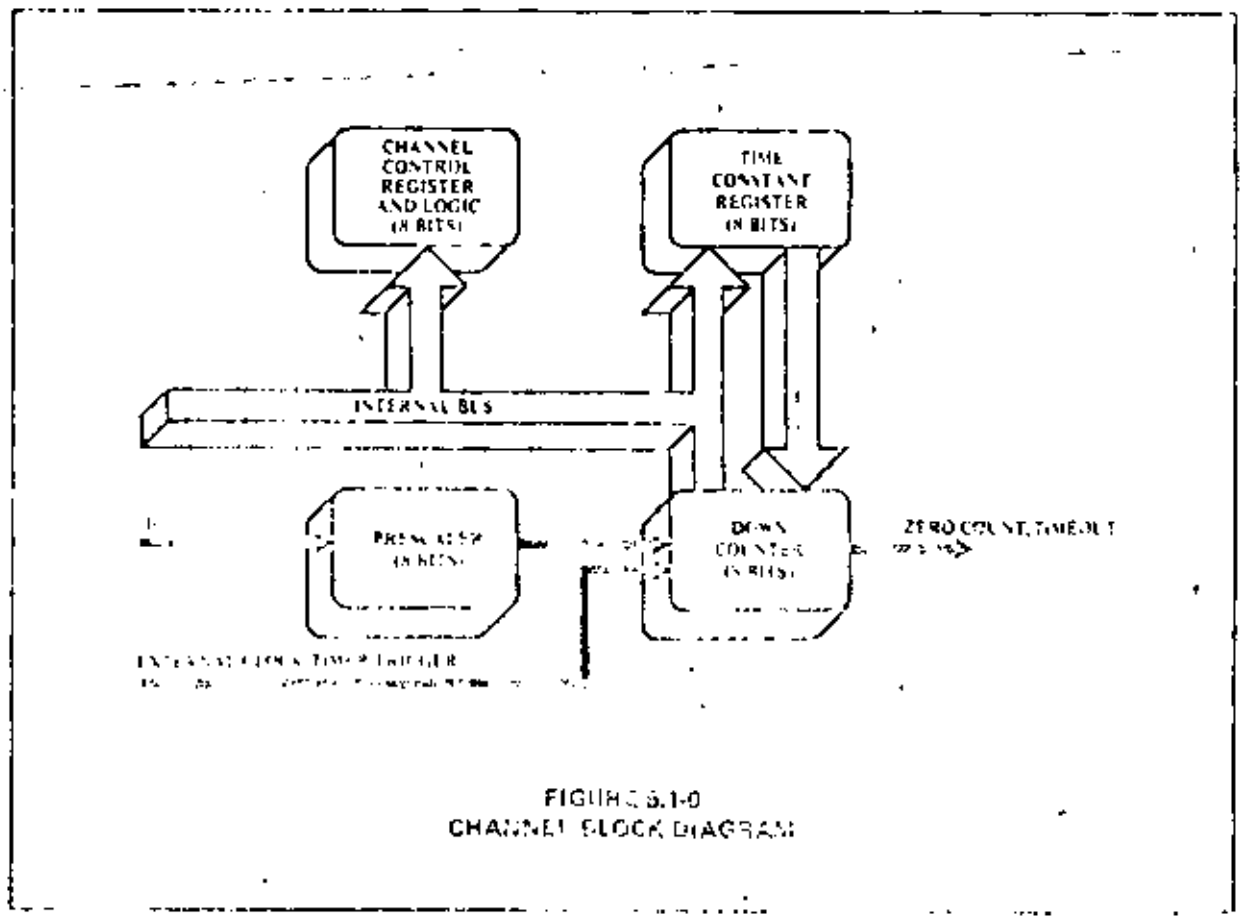
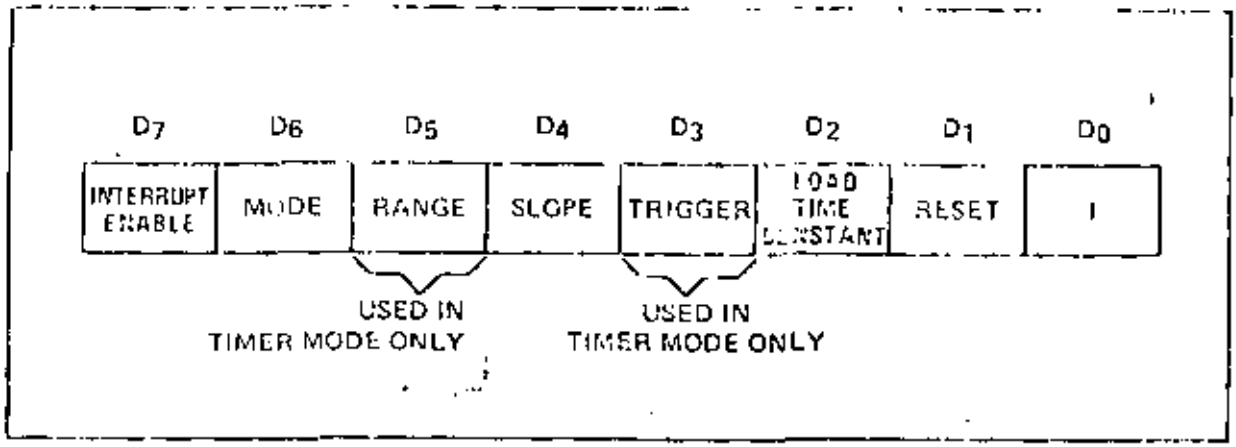


FIGURE 5.1-0
CHANNEL BLOCK DIAGRAM

5.1 LOADING THE CHANNEL CONTROL REGISTER (CONT'D)



Bit 7 = 1

The channel is enabled to generate an interrupt request sequence every time the Down Counter reaches a zero-count condition. To set this bit to 1 in any of the four Channel Control Registers necessitates that an Interrupt Vector also be written to the CTC before operation begins. Channel interrupts may be programmed in either Counter Mode or Timer Mode. If an updated Channel Control Word is written to a channel already in operation, with bit 7 set, the interrupt enable selection will not be retroactive to a preceding zero-count condition.

Bit 7 = 0

Channel interrupts disabled.

Bit 6 = 1

Counter Mode selected. The Down Counter is decremented by each triggering edge of the External Clock (CLK/TRG) input. The Prescaler is not used.

Bit 6 = 0

Timer Mode selected. The Prescaler is clocked by the System Clock Φ, and the output of the Prescaler in turn clocks the Down Counter. The output of the Down Counter (the channel's ZC/TO output) is a uniform square pulse train of period given by the product:

$$t_c * P * TC$$

where t_c is the period of System Clock Φ; P is the Prescaler factor of 16 or 256, and TC is the time constant data word.

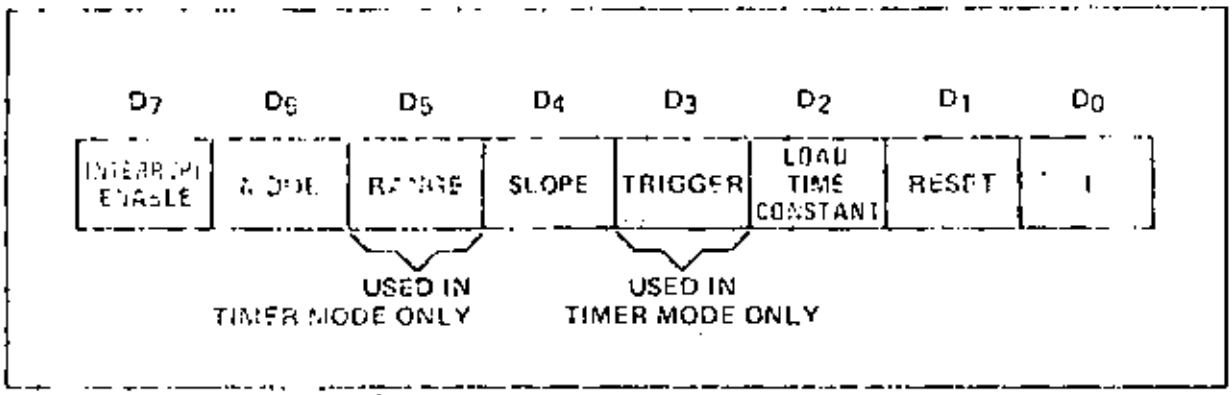
Bit 5 = 1

(Defined for Timer Mode only.) Prescaler factor is 256.

Bit 5 = 0

(Defined for Timer Mode only.) Prescaler factor is 16.

5.1 LOADING THE CHANNEL CONTROL REGISTER (CONT'D)



Bit 4 = 1

TIMER MODE -- positive edge trigger starts timer operation.

COUNTER MODE -- positive edge decrements the down counter.

Bit 4 = 0

TIMER MODE -- negative edge trigger starts timer operation.

COUNTER MODE -- negative edge decrements the down counter.

Bit 3 = 1

Timer Mode Only, -- External trigger is valid for starting timer operation after rising edge of T_2 of the machine cycle following the one that loads the time constant. The Prescaler is decremented 2 clock cycles later if the setup time is met, otherwise 3 clock cycles.

Bit 3 = 0

Timer Mode Only -- Timer begins operation on the rising edge of T_2 of the machine cycle following the one that loads the time constant.

Bit 2 = 1

The time constant data word for the Time Constant Register will be the next word written to this channel. If an updated Channel Control Word and time constant data word are written to a channel while it is already in operation, the Down Counter will continue decrementing to zero before the new time constant is loaded into it.

Bit 2 = 0

No time constant data word for the Time Constant Register should be expected to follow. To program bit 2 to this state implies that this Channel Control Word is intended to update the status of a channel already in operation, since a channel will not operate without a correctly programmed data word in the Time Constant Register, or a 1 as set bit 2 in this Channel Control Word provides the only way of writing to the Time Constant Register.

Bit 1 = 1

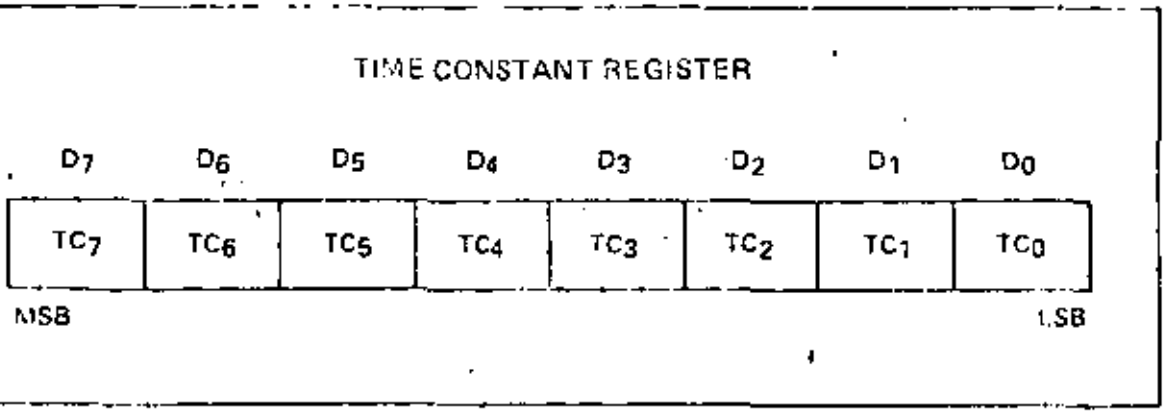
Resets channel stops down counter timing. This is not a stored condition. Upon writing into this bit a reset pulse occurs in current channel operation, however, none of the bits in the channel control register are changed. If bit 2 = 1 and bit 1 = 1 the channel will resume operation upon loading a time constant.

Bit 1 = 0

Channel continues current operation.

5.2 LOADING THE TIME CONSTANT REGISTER

A channel may not begin operation in either Timer Mode or Counter Mode until a time constant data word is written into the Time Constant Register by the CPU. This data word will be operated on the next I/O Write to this channel following the I/C Write of the Channel Control Word, provided that bit 2 of the Channel Control Word is set. The time constant data word may be any integer value in the range 1-256. If all eight bits in this word are zero, it is interpreted as 256. If a time constant data word is loaded to a channel already in operation, the Down Counter will continue decrementing to zero before the new time constant is loaded from the Time Constant Register to the Down Counter.



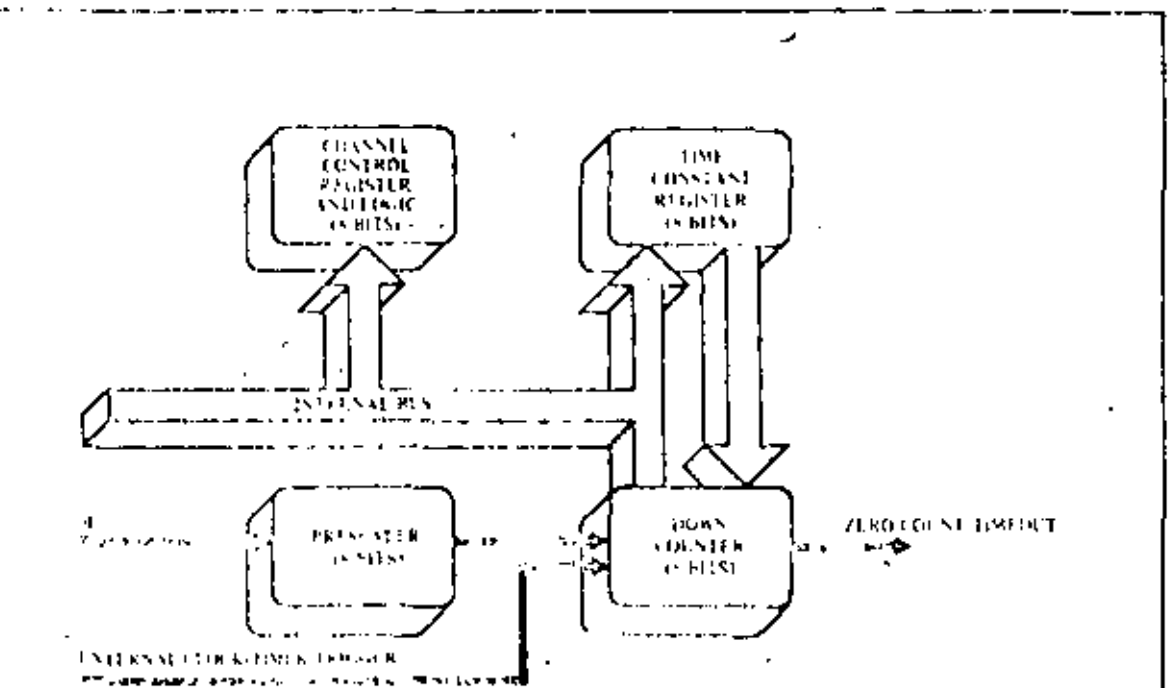
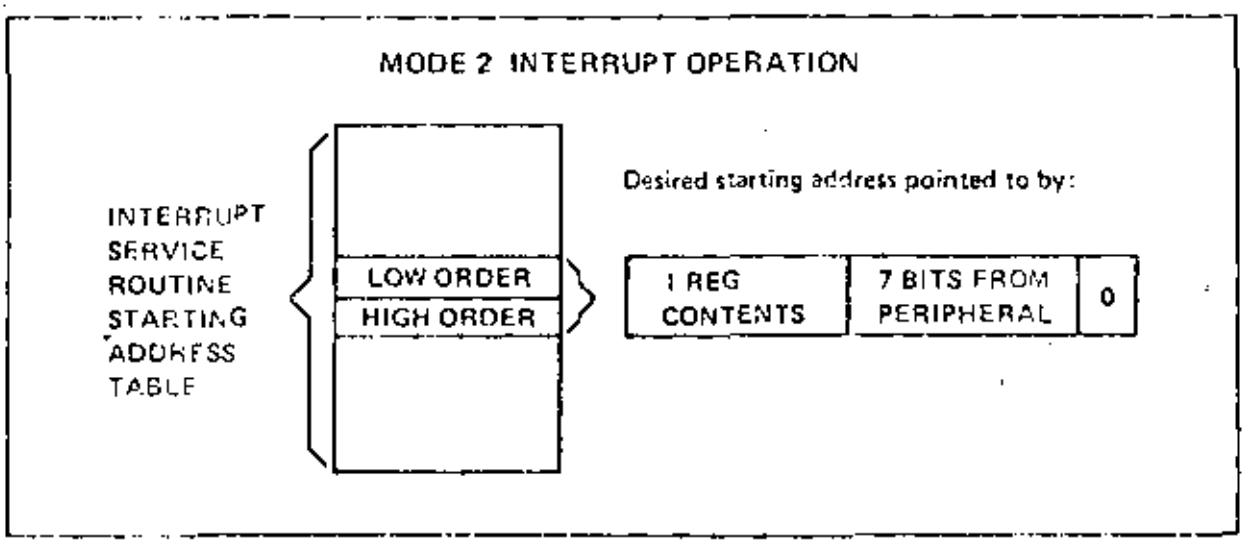


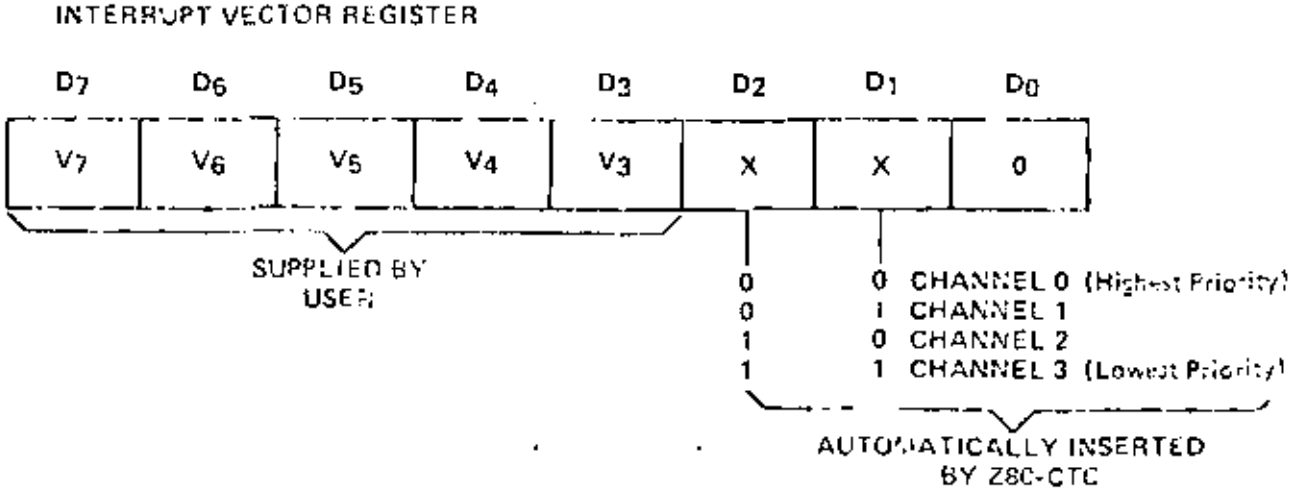
FIGURE 5.2-0
CHANNEL BLOCK DIAGRAM

5.3 LOADING THE INTERRUPT VECTOR REGISTER

The Z80-CTC has been designed to operate with the Z80-CPU programmed for mode 2 interrupt response. Under the requirements of this mode, when a CTC channel requests an interrupt and is acknowledged, a 16-bit pointer must be formed to obtain a corresponding interrupt service routine starting address from a table in memory. The upper 8 bits of this pointer are provided by the CPU's I register, and the lower 8 bits of the pointer are provided by the CTC in the form of an Interrupt Vector unique to the particular channel that requested the interrupt. (For further details see section 7.0: "CTC Interrupt Servicing".)



The high order 5 bits of this Interrupt Vector must be written to the CTC in advance as part of the initial programming sequence. To do so, the CPU must write to the I/O port address corresponding to the CTC channel 0, just as it would if a Channel Control Word were being written to that channel, except that bit 0 of the word being written must contain a 0. (As explained above in section 5.1, if bit 0 of a word written to a channel were set to 1, the word would be interpreted as a Channel Control Word, so a 0 in bit 0 signals the CTC to load the incoming word into the Interrupt Vector Register.) Bits 1 and 2, however, are not used when loading this vector. At the time when the interrupting channel must place the Interrupt Vector on the Z80 Data Bus, the Interrupt Control Logic of the CTC automatically supplies a binary code in bits 1 and 2 identifying which of the four CTC channels is to be serviced.



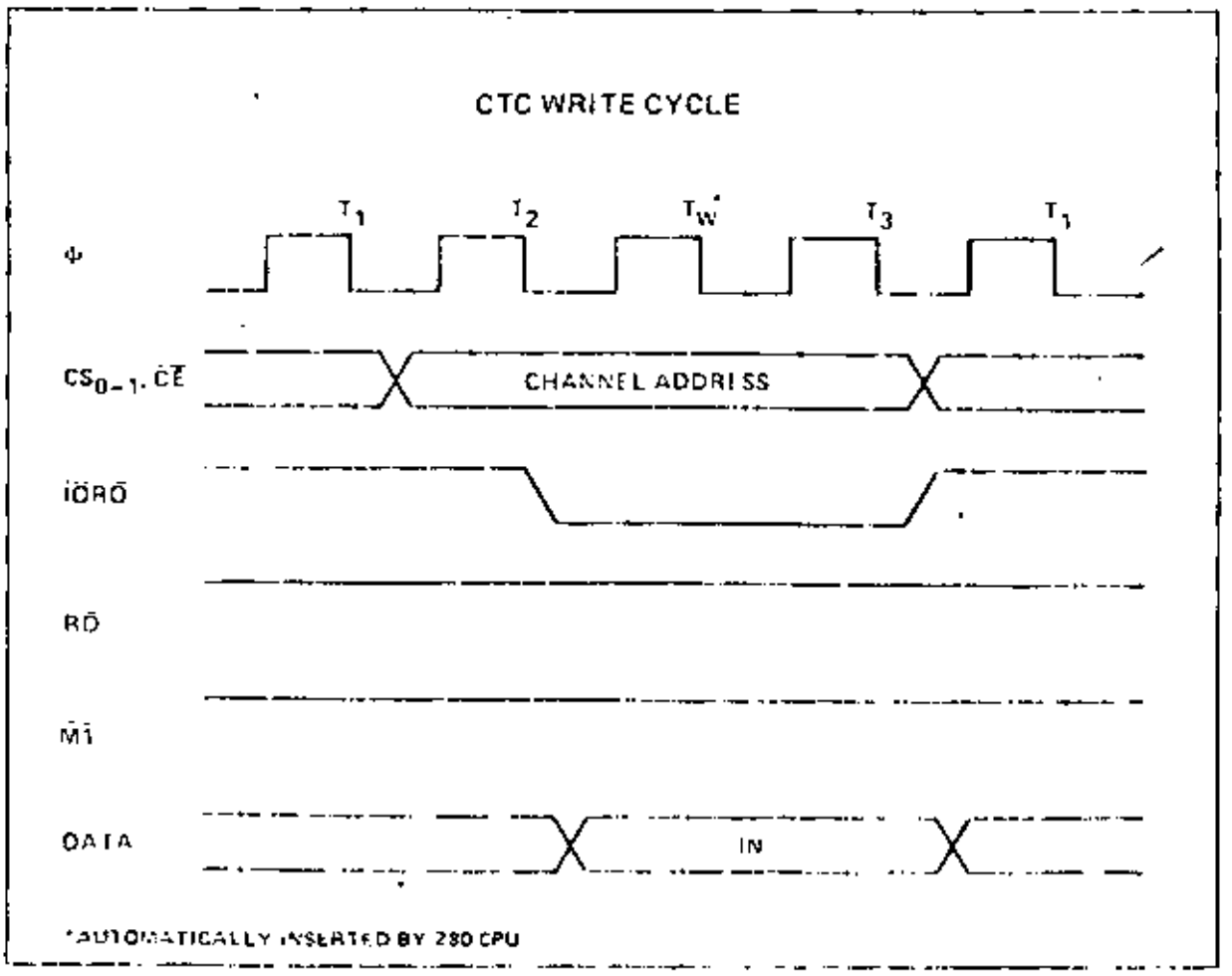
6.0 CTC TIMING

This section illustrates the timing relationships of the relevant CTC pins for the following types of operation: writing a word to the CTC, reading a word from the CTC, counting, and timing. Elsewhere in this manual may be found timing diagrams relating to interrupt servicing (section 7.0) and an A.C. Timing Diagram which quantitatively specifies the timing relationships (section 8.4).

6.1 CTC WRITE CYCLE

Figure 6.0-1 illustrates the timing associated with the CTC Write Cycle. This sequence is applicable to loading either a Channel Control Word, an Interrupt Vector, or a time constant data word.

In the sequence shown, during clock cycle T_1, the Z80-CPU prepares for the Write Cycle with a false (high) signal at CTC input pin $\overline{RD}$ (Read). Since the CTC has no separate Write signal input, it generates its own internally from the false $\overline{RD}$ input. Later, during clock cycle T_2, the Z80-CPU initiates the Write Cycle with true (low) signals at CTC input pins $\overline{IORQ}$ (I/O Request) and $\overline{CE}$ (Chip Enable). (Note: $\overline{M1}$ must be false to distinguish the cycle from an interrupt acknowledge.) Also at this time a 2-bit binary code appears at CTC inputs CS1 and CS0 (Channel Select 1 and 0), specifying which of the four CTC channels is being written to, and the word being written appears on the Z80 Data Bus. Now everything is ready for the word to be latched into the appropriate CTC internal register in synchronization with the rising edge beginning clock cycle T_3. No additional wait states are allowed.



6.2 CTC READ CYCLE

Figure 6.0-2 illustrates the timing associated with the CTC Read Cycle. This sequence is used any time the CPU reads the current contents of the Down Counter. During clock cycle T_1, the Z80-CPU initiates the Read Cycle with true signals at input pins RD (Read), IORQ (I/O Request), and CE (Chip Enable). Also at this time a 2-bit binary code appears at CTC inputs CS1 and CS0 (Channel Select 1 and 0), specifying which of the four CTC channels is being read from. (Note: M1 must be false to distinguish the cycle from an interrupt acknowledge.) On the rising edge of the cycle T_2 the valid contents of the Down Counter as of the rising edge of cycle T_2 will be available on the Z80 Data Bus. No additional wait states are allowed.

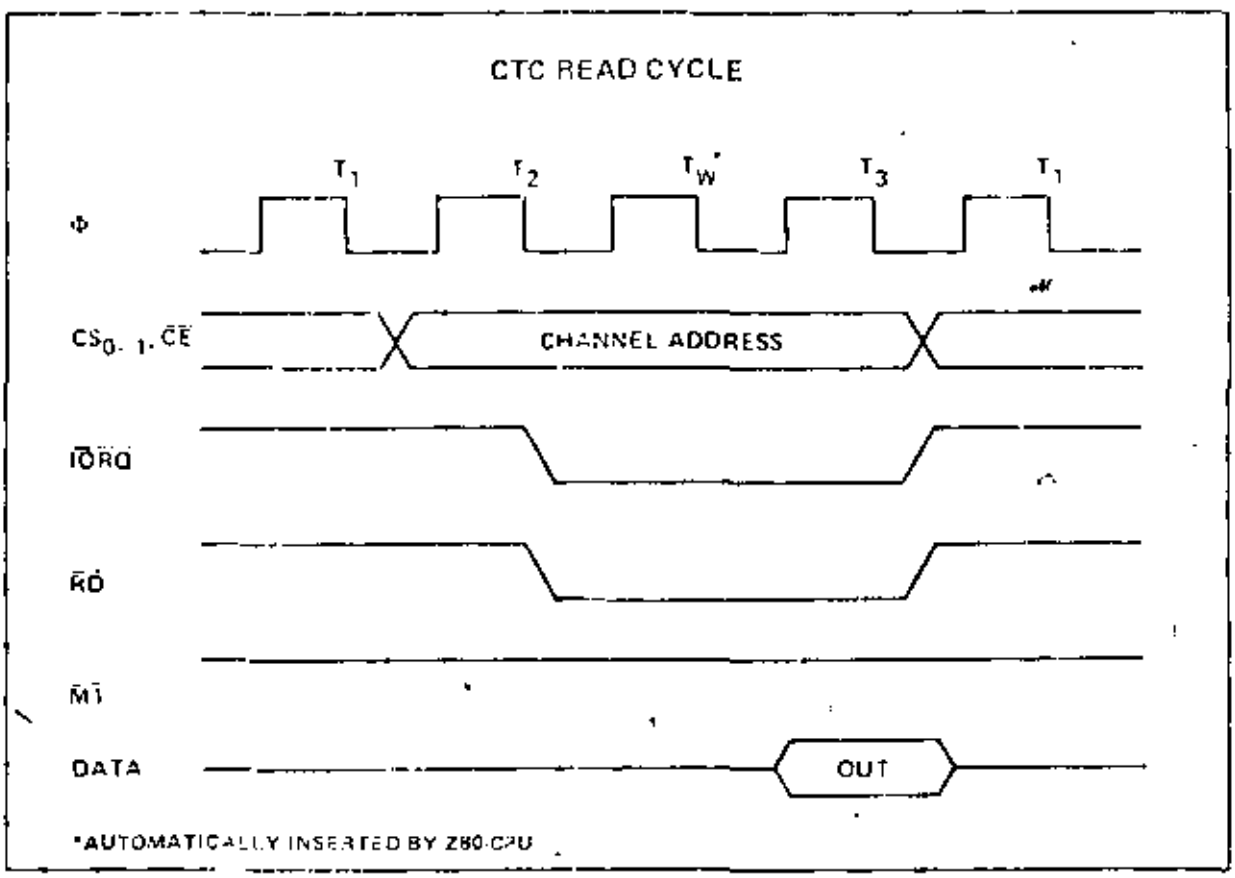
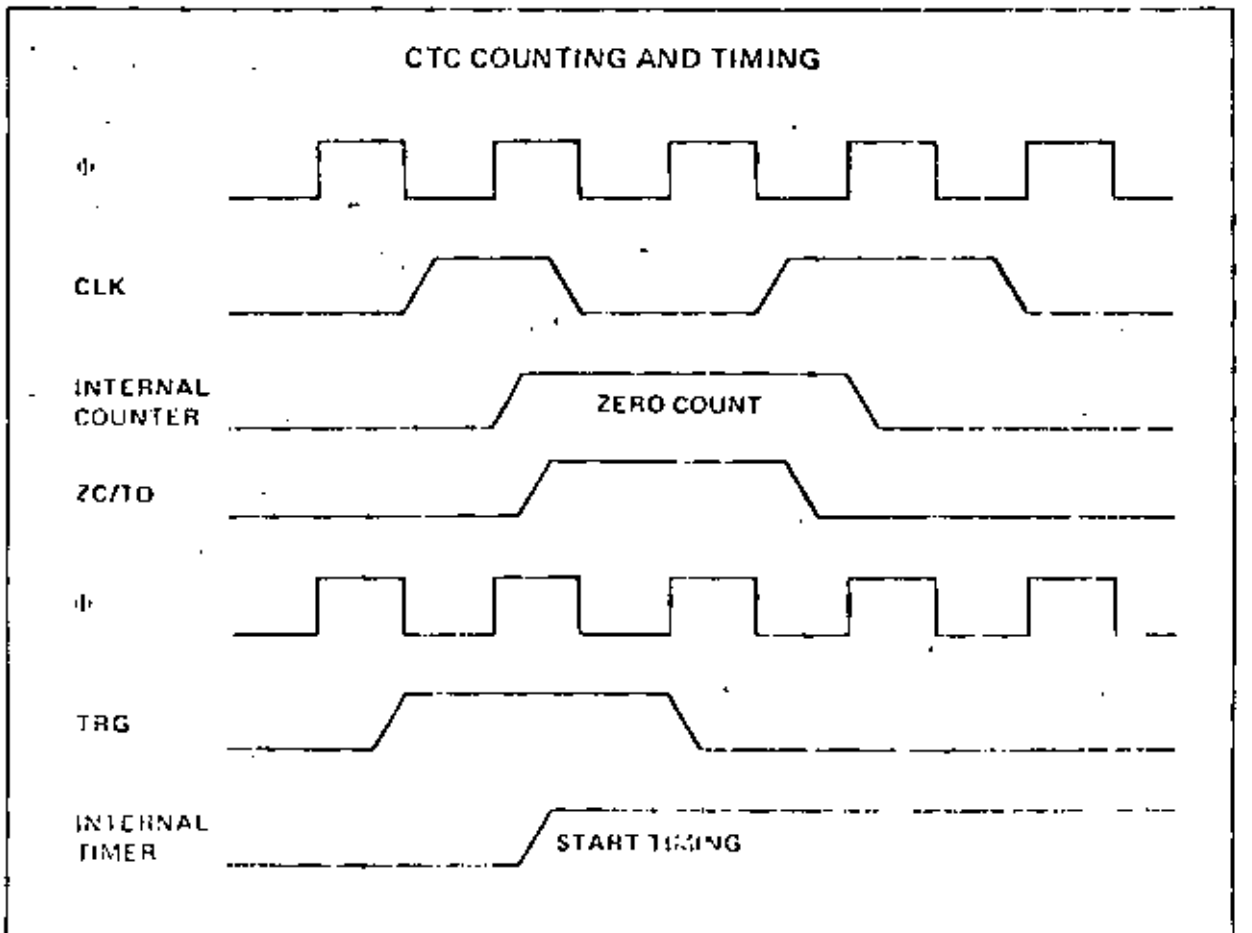


Figure 6.0-3 illustrates the timing diagram for the CTC Counting and Timing Modes.

In the Counter Mode, the edge (rising edge is active in this example) from the external hardware connected to pin CLK/TRG decrements the Down Counter in synchronization with the System Clock Φ. As specified in the A.C. Characteristics (Section 9.1) this CLK/TRG pulse must have a minimum width and the minimum period must not be less than twice the system clock period. Although there is no set-up time requirement between the active edge of the CLK/TRG and the rising edge of Φ if the CLK/TRG edge occurs closer than a specified minimum time, the decrement of the Down Counter will be delayed one cycle of Φ. Immediately after the decrement of the Down Counter, 1 to 0, the ZC/TO output is pulsed true.

In the Timer Mode, a pulse trigger (user-selectable as either active high or active low) at the CLK/TRG pin enables timing function on the second succeeding rising edge of Φ. As in the Counter Mode, the triggering pulse is detected asynchronously and must have a minimum width. The timing function is initiated in synchronization with Φ, and a minimum set-up time is required between the active edge of the CLK/TRG and the next rising edge of Φ. If the CLK/TRG edge occurs closer than this, the initiation of the timer function will be delayed one cycle of Φ.



7.0 CTC INTERRUPT SERVICING

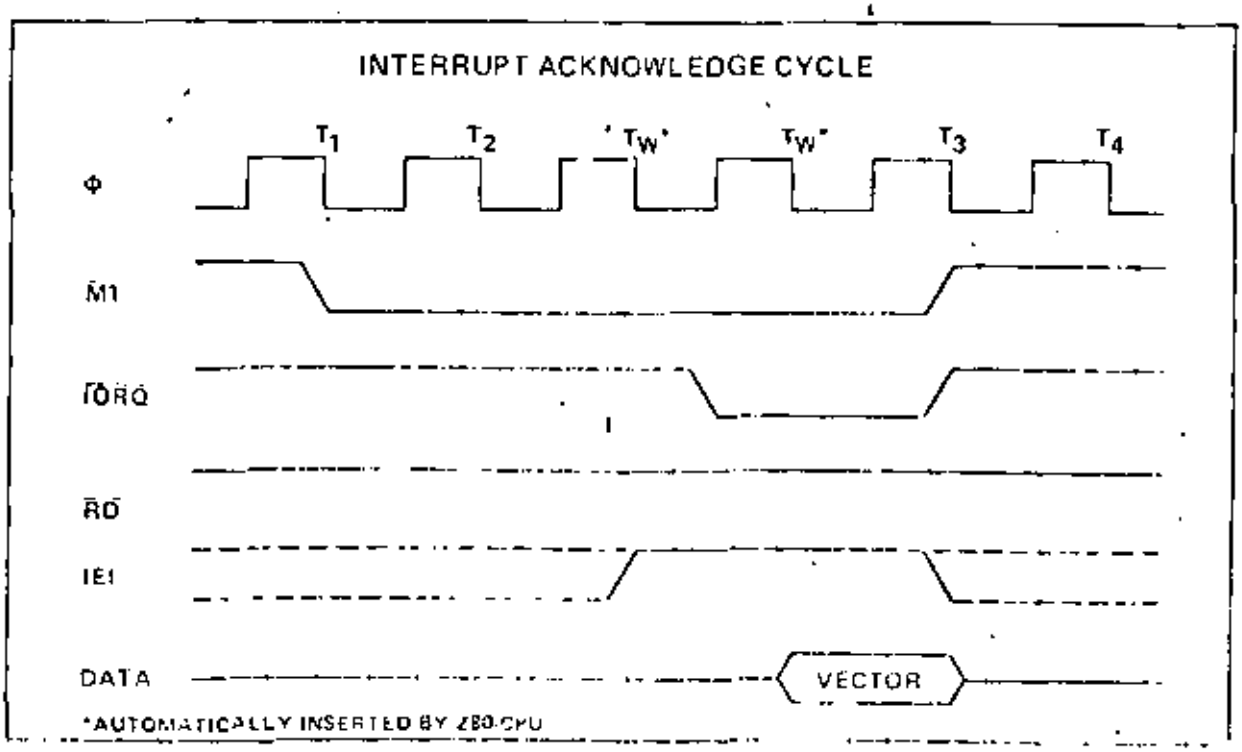
Each CTC channel may be individually programmed to request an interrupt every time its Down Counter reaches a count of zero. The purpose of a CTC-generated interrupt, as for any other peripheral device, is to force the CPU to execute an interrupt service routine. To utilize this feature the Z80-CPU must be programmed for mode 2 interrupt response. Under the requirements of this mode, when a CTC channel requests an interrupt and is acknowledged, a 16-bit pointer must be formed to obtain a corresponding interrupt service routine starting address from a table in memory. The lower 8 bits of the pointer are provided by the CTC in the form of an Interrupt Vector unique to the particular channel that requested the interrupt. (For further details, refer to chapter 5.0 of the Z80-CPU Technical Manual.)

The CTC's Interrupt Control Logic insures that it acts in accordance with Z80 system interrupt protocol for nested priority interrupt and proper return from interrupt. The priority of any system device is determined by its physical location in a daisy chain configuration. Two signal lines (IEI and IEO) are provided in the CTC and all Z80 peripheral devices to form the system daisy chain. The device closest to the CPU has the highest priority; within the CTC, interrupt priority is predetermined by channel number, with channel 0 having highest priority. According to Z80 system interrupt protocol, low priority devices or channels may not interrupt higher priority devices or channels that have already interrupted and not had their interrupt service routines completed. However, high priority devices or channels may interrupt the servicing of lower priority devices or channels. (For further details, see section 2.3: "Interrupt Control Logic".)

Sections 7.1 and 7.2 below describe the nominal timing relationships of the relevant CTC pins for the Interrupt Acknowledge Cycle and the Return from Interrupt Cycle. Section 7.3 below discusses a typical example of daisy chain interrupt servicing.

7.1 INTERRUPT ACKNOWLEDGE CYCLE

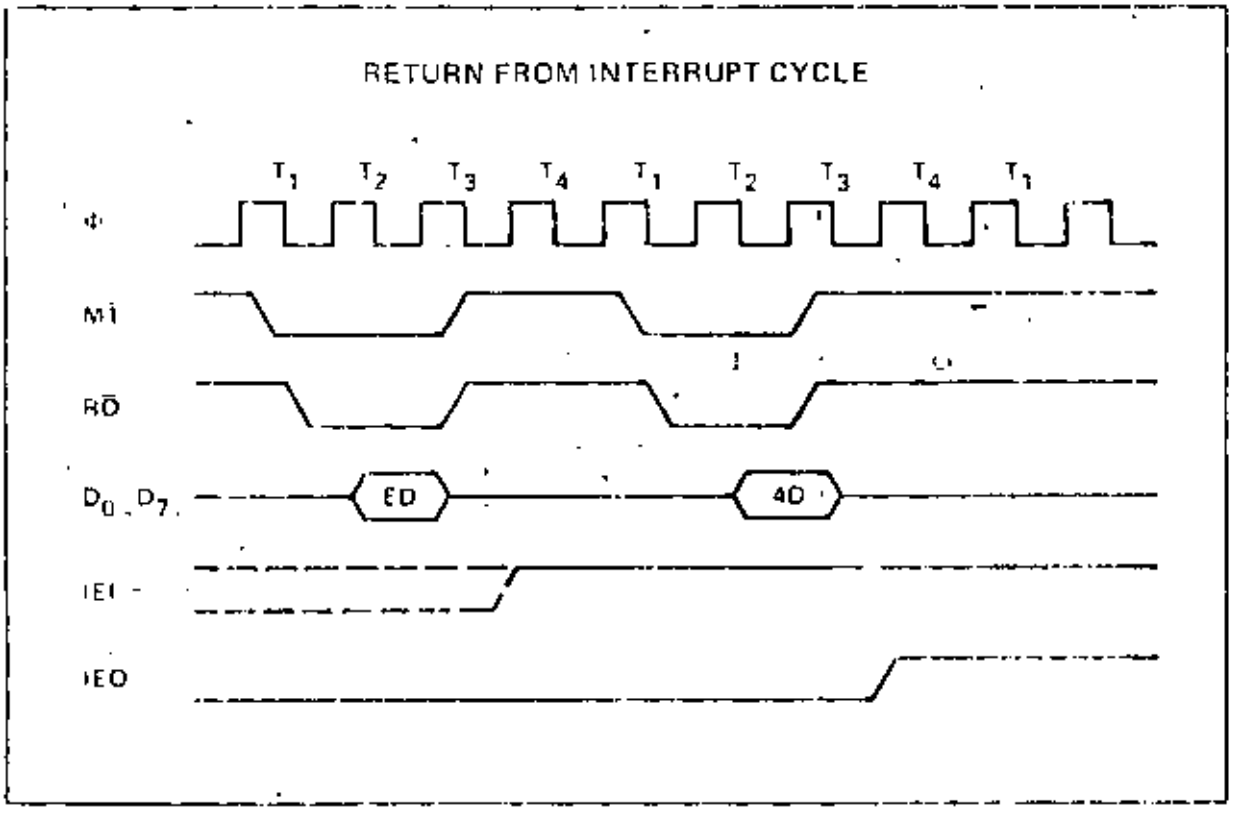
Figure 7.0-1 illustrates the timing associated with the Interrupt Acknowledge Cycle. Some time after an interrupt is requested by the CTC, the CPU will send out an interrupt acknowledge ($\overline{M1}$ and $\overline{IORQ}$). To insure that the daisy chain enable lines stabilize, channels are inhibited from changing their interrupt request status when $\overline{M1}$ is active. $\overline{M1}$ is active about two clock cycles earlier than $\overline{IORQ}$, and $\overline{RD}$ is false to distinguish the cycle from an instruction fetch. During this time the interrupt logic of the CTC will determine the highest priority channel requesting an interrupt. If the CTC Interrupt Enable Input (IEI) is active, then the highest priority interrupting channel within the CTC places its Interrupt Vector onto the Data Bus when $\overline{IORQ}$ goes active. Two wait states (T_W*) are automatically inserted at this time to allow the daisy chain to stabilize. Additional wait states may be added.



7.2 RETURN FROM INTERRUPT CYCLE

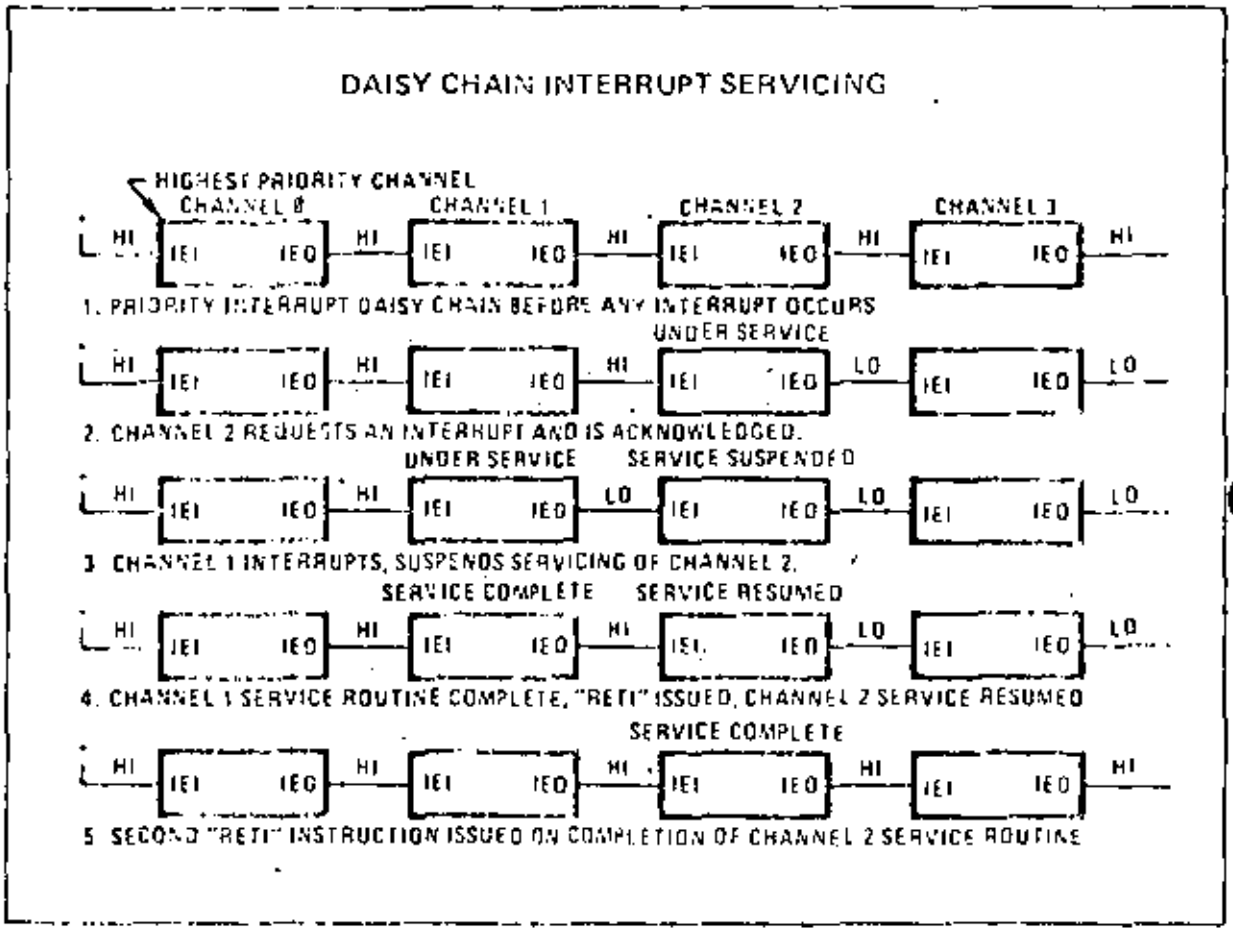
Figure 7.0 2 illustrates the timing associated with the RETI Instruction. This instruction is used at the end of an interrupt service routine to initialize the daisy chain enable lines for proper control of nested priority interrupt handling. The CTC decodes the two-byte RETI code internally and determines whether it is intended for a channel being serviced.

When several Z80 peripheral chips are in the daisy chain IEI will become active on the chip currently under service when an ED H opcode is decoded. If the following opcode is 4DH, the peripheral being serviced will be re-initialized and its IEO will become active. Additional wait states are allowed.



7.3 DAISY CHAIN INTERRUPT SERVICING

Figure 7.0-3 illustrates a typical nested interrupt sequence which may occur in the CTC. In this example, channel 2 interrupts and is granted service. While this channel is being serviced, higher priority channel 1 interrupts and is granted service. The service routine for the higher priority channel is completed, and a RETI instruction (see section 7.2 for further details) is executed to signal the channel that its routine is complete. At this time, the service routine of the lower priority channel 2 is resumed and completed.



8.0 ABSOLUTE MAXIMUM RATINGS

Temperature Under Bias	0° C to +70° C
Storage Temperature	-65° C to +150° C
Voltage On Any Pin With Respect To Ground	-0.3 V to +7 V
Power Dissipation	0.8 W

*Comment
Stresses above those listed under "Absolute Maximum Rating" may cause permanent damage to the device. This is a stress rating only and functional operation of the device at these or any other condition above those indicated in the operational sections of this specification is not implied. Exposure to absolute maximum rating conditions for extended periods may affect device reliability.

8.1 D.C. CHARACTERISTICS

T_A = 0° C to 70° C, V_{CC} = 5V ± 5% unless otherwise specified

Z80-CTC

Symbol	Parameter	Min	Max	Unit	Test Condition
V_{ILC}	Clock Input Low Voltage	-0.3	.45	V	
V_{IHC}	Clock Input High Voltage [1]	V_{CC} - .6	V_{CC} + .3	V	
V_{IL}	Input Low Voltage	-0.3	0.8	V	
V_{IH}	Input High Voltage	2.0	V_{CC}	V	
V_{OL}	Output Low Voltage		0.4	V	I_{OL} = 2 mA
V_{OH}	Output High Voltage	2.4		V	I_{OH} = -250 µA
I_{CC}	Power Supply Current		120	mA	T_C = 400 nsec
I_{LI}	Input Leakage Current		10	µA	V_{IN} = 0 to V_{CC}
I_{LOH}	Tri-State Output Leakage Current in Float		10	µA	V_{OUT} = 2.4 to V_{CC}
I_{LOL}	Tri-State Output Leakage Current in Float		-10	µA	V_{OUT} = 0.4V
I_{OHD}	Darlington Drive Current	-1.5		mA	V_{OH} = 1.5V R_{EXT} = 390Ω

Z80A-CTC

Symbol	Parameter	Min	Max	Unit	Test Condition
V_{ILC}	Clock Input Low Voltage	-0.3	.45	V	
V_{IHC}	Clock Input High Voltage [1]	V_{CC} - .6	V_{CC} + .3	V	
V_{IL}	Input Low Voltage	-0.3	0.8	V	
V_{IH}	Input High Voltage	2.0	V_{CC}	V	
V_{OL}	Output Low Voltage		0.4	V	I_{OL} = 2 mA
V_{OH}	Output High Voltage	2.4		V	I_{OH} = -250 µA
I_{CC}	Power Supply Current		120	mA	T_C = 250 nsec
I_{LI}	Input Leakage Current		10	µA	V_{IN} = 0 to V_{CC}
I_{LOH}	Tri-State Output Leakage Current in Float		10	µA	V_{OUT} = 2.4 to V_{CC}
I_{LOL}	Tri-State Output Leakage Current in Float		-10	µA	V_{OUT} = 0.4V
I_{OHD}	Darlington Drive Current	-1.5		mA	V_{OH} = 1.5V R_{EXT} = 390Ω

8.2 CAPACITANCE

T_A = 25° C, f = 1 MHz

Symbol	Parameter	Max.	Unit	Test Condition
C_ϕ	Clock Capacitance	20	pF	Unmeasured Pins Returned to Ground
C_{IN}	Input Capacitance	5	pF	
C_{OUT}	Output Capacitance	10	pF	

T_A = 0° C to 70° C, V_{CC} = +5 V ± 5%, unless otherwise noted

Signal	Symbol	Parameter	Min	Max	Unit	Comments
Φ	t_C	Clock Period	400	[1]	ns	
	$t_{W(\Phi H)}$	Clock Pulse Width, Clock High	170	2000	ns	
	$t_{W(\Phi L)}$	Clock Pulse Width, Clock Low	170	2000	ns	
	t_r, t_f	Clock Rise and Fall Times		30	ns	
	t_H	Any Hold Time for Specified Setup Time	0		ns	
CS, $\overline{CE}$, etc.	$t_{S(CS)}$	Control Signal Setup Time to Rising Edge of Φ During Read or Write Cycle	160		ns	
D_0–D_7	$t_{DR(D)}$	Data Output Delay from Rising Edge of $\overline{RD}$ During Read Cycle		480	ns	[2]
	$t_{S(D)}$	Data Setup Time to Rising Edge of Φ During Write or MT Cycle	60		ns	
	$t_{DI(D)}$	Data Output Delay from Falling Edge of IOHQ During INTA Cycle		340	ns	[2]
	$t_{F(D)}$	Delay to Floating Bus (Output Buffer Disable Time)		230	ns	
IEI	$t_{S(IEI)}$	IEI Setup Time to Falling Edge of $\overline{IORQ}$ During INTA Cycle	200		ns	
IEO	$t_{DH(IO)}$	IEO Delay Time from Rising Edge of IEI		220	ns	[3]
	$t_{DL(IO)}$	IEO Delay Time from Falling Edge of IEI		190	ns	[3]
	$t_{DM(IO)}$	IEO Delay from Falling Edge of $\overline{M1}$ (Interrupt Occurring just Prior to $\overline{M1}$)		300	ns	[3]
$\overline{IORQ}$	$t_{S(IR)}$	$\overline{IORQ}$ Setup Time to Rising Edge of Φ During Read or Write Cycle	250		ns	
$\overline{M1}$	$t_{S(M1)}$	$\overline{M1}$ Setup Time to Rising Edge of Φ During INTA or MI Cycle	210		ns	
$\overline{RD}$	$t_{S(RD)}$	$\overline{RD}$ Setup Time to Rising Edge of Φ During Read or MI Cycle	240		ns	
$\overline{INT}$	$t_{DC(IT)}$	$\overline{INT}$ Delay Time from Rising Edge of CLK/TRG		$2t_{C(H)}$ + 200		Counter Mode
	$t_{D(IT)}$	$\overline{INT}$ Delay Time from Rising Edge of Φ		$t_{C(H)}$ + 200		Timer Mode
CLK/TRG_{0-3}	$t_{C(CK)}$	Clock Period	$2t_{C(H)}$			Counter Mode
	t_r, t_f	Clock and Trigger Rise and Fall Times		50		
	$t_{S(CK)}$	Clock Setup Time to Rising Edge of Φ for Immediate Count	210			Counter Mode
	$t_{S(TR)}$	Trigger Setup Time to Rising Edge of Φ for Enabling of Prescaler on Following Rising Edge of Φ	210			Timer Mode
	$t_{W(CTH)}$	Clock and Trigger High Pulse Width	200			Counter and Timer Modes
	$t_{W(CTL)}$	Clock and Trigger Low Pulse Width	200			Counter and Timer Modes
ZC/TO_{0-2}	$t_{DH(ZC)}$	ZC/TO Delay Time from Rising Edge of Φ, ZC/TO High		190		Counter and Timer Modes
	$t_{DL(ZC)}$	ZC/TO Delay Time from Falling Edge of Φ, ZC/TO Low		190		Counter and Timer Modes

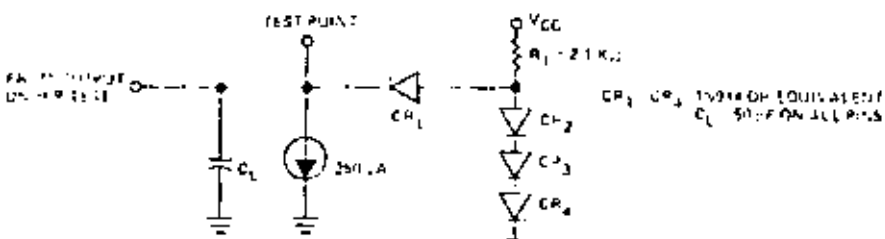
Notes: [1] $t_C + t_{W(\Phi H)} + t_{W(\Phi L)} + t_r + t_f$

[2] Increase delay by 10 nsec for each 50 pF increase in loading, 200 pF maximum for data lines and 100 pF for control lines

[3] Increase delay by 2 nsec for each 10 pF increase in loading, 100 pF maximum

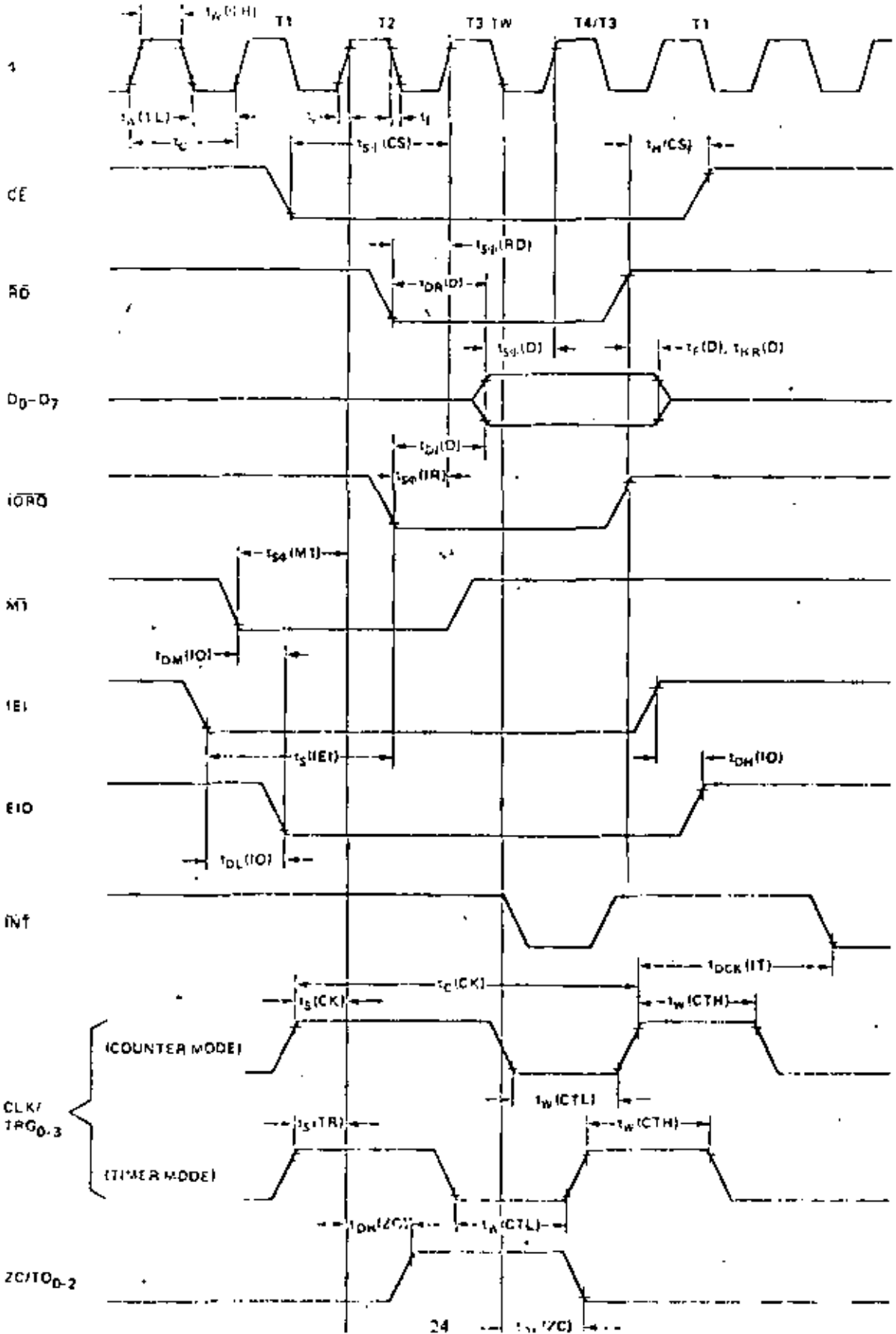
[4] $\overline{RESET}$ must be active for a minimum of 3 clock cycles

OUTPUT LOAD CIRCUIT



	"1"	"0"
CLOCK	V_{CC} - 6V	.45V
OUTPUT	2.0V	.8V
INPUT	2.0V	.8V
FLOAT	ΔV	±0.5V

Timing measurements are made at the following voltages unless otherwise specified.



$T_A = 0^\circ C$ to $70^\circ C$, $V_{CC} = +5 V \pm 5\%$, unless otherwise noted

Signal	Symbol	Parameter	Min	Max	Unit	Comments
Φ	t_C	Clock Period	250	[1]	ns	
	$t_{W(PWH)}$	Clock Pulse Width, Clock High	105	2000	ns	
	$t_{W(PWL)}$	Clock Pulse Width, Clock Low	105	2000	ns	
	t_r, t_f	Clock Rise and Fall Times		30	ns	
	t_H	Any Hold Time for Specified Setup Time	0		ns	
CS, $\overline{CE}$, etc.	$t_{S\Phi(CS)}$	Control Signal Setup Time to Rising Edge of Φ During Read or Write Cycle	60		ns	
D_0–D_7	$t_{DR(D)}$	Data Output Delay from Falling Edge of $\overline{RD}$ During Read Cycle		380	ns	[2]
	$t_{S(D)}$	Data Setup Time to Rising Edge of Φ During Write or M1 Cycle	50		ns	
	$t_{DI(D)}$	Data Output Delay from Falling Edge of IORQ During INTA Cycle		160	ns	[2]
	$t_{F(D)}$	Delay to Floating Bus (Output Buffer Disable Time)		110	ns	
IEI	$t_{S(IEI)}$	IEI Setup Time to Falling Edge of $\overline{IORQ}$ During INTA Cycle	140		ns	
IEO	$t_{DH(IO)}$	IEO Delay Time from Rising Edge of IEI		160	ns	[3]
	$t_{DL(IO)}$	IEO Delay Time from Falling Edge of IEI		130	ns	[3]
	$t_{DX(IO)}$	IEO Delay from Falling Edge of $\overline{M1}$ (Interrupt Occurring just Prior to $\overline{M1}$)		190	ns	[3]
$\overline{IORQ}$	$t_{S\Phi(IR)}$	$\overline{IORQ}$ Setup Time to Rising Edge of Φ During Read or Write Cycle	115		ns	
$\overline{M1}$	$t_{S\Phi(M1)}$	$\overline{M1}$ Setup Time to Rising Edge of Φ During INTA or M1 Cycle	90		ns	
$\overline{RD}$	$t_{S\Phi(RD)}$	$\overline{RD}$ Setup Time to Rising Edge of Φ During Read or M1 Cycle	115		ns	
$\overline{INT}$	$t_{DCK(IT)}$	$\overline{INT}$ Delay Time from Rising Edge of CLK/TRG		$2t_C(\Phi) + 140$		Counter Mode
	$t_{D\Phi(IT)}$	$\overline{INT}$ Delay Time from Rising Edge of Φ		$t_C(\Phi) + 140$		Timer Mode
CLK/TRG_{0-3}	$t_{C(C)}$	Clock Period	$2t_C(\Phi)$			Counter Mode
	t_r, t_f	Clock and Trigger Rise and Fall Times		30		
	$t_{S(CK)}$	Clock Setup Time to Rising Edge of Φ for Immediate Count	130			Counter Mode
	$t_{S(TR)}$	Trigger Setup Time to Rising Edge of Φ for enabling of Prescaler on Following Rising Edge of Φ	130			Timer Mode
	$t_{W(CTH)}$	Clock and Trigger High Pulse Width	120			Counter and Timer Modes
	$t_{W(CTL)}$	Clock and Trigger Low Pulse Width	120			Counter and Timer Modes
ZC/TO_{0-2}	$t_{DH(ZC)}$	ZC/TO Delay Time from Rising Edge of Φ, ZC/TO High		120		Counter and Timer Modes
	$t_{DL(ZC)}$	ZC/TO Delay Time from Rising Edge of Φ, ZC/TO Low		120		Counter and Timer Modes

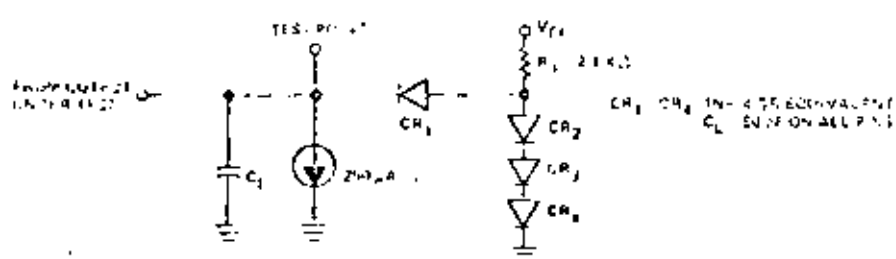
Notes: [1] $t_C = t_{W(PWH)} + t_{W(PWL)} + t_r + t_f$

[2] Increase delay by 10 nsec for each 50 pF increase in loading. 200 pF maximum for data lines and 100 pF for control lines.

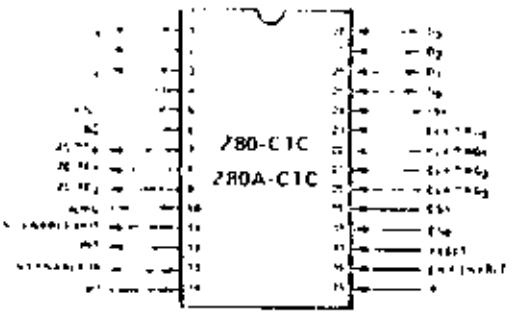
[3] Increase delay by 2 nsec for each 10 pF increase in loading. 100 pF maximum.

[4] $\overline{RESET}$ must be active for a minimum of 3 clock cycles.

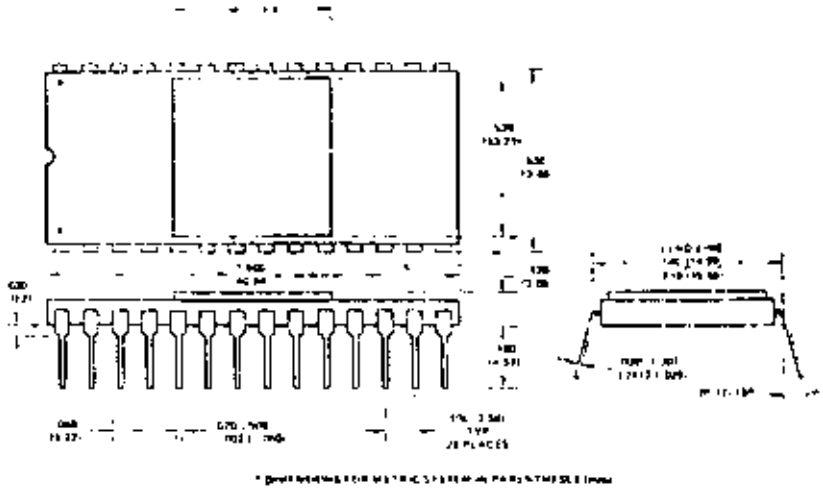
OUTPUT LOAD CIRCUIT



8.6 PACKAGE CONFIGURATION



PACKAGE OUTLINE



Ordering Information

C Ceramic
P Plastic
S Standard 5V±5%, 0° to 70°C
E Extended 5V±5%, -40° to 85°C
M Military 5V±10%, -55° to 125°C

Example:

Z80-CTC CS (Ceramic -Standard Range)
Z80A-CTC PS (Plastic -Standard Range, 4 MHz)

ZILOG Z80 MICROCOMPUTER SYSTEM COMPONENT FAMILY

• Z80, Z80A-CPU	CENTRAL PROCESSOR UNIT
• Z80, Z80A-PIO	PARALLEL I/O
• Z80, Z80A-CTC	COUNTER/TIMER CIRCUIT
• Z80, Z80A-DMA	DIRECT MEMORY ACCESS
• Z80, Z80A-SIO	SERIAL I/O
• Z6104	4K x 1 STATIC RAM
• Z6116	16K x 1 DYNAMIC RAM

centro de educación continua
división de estudios superiores
facultad de ingeniería, unam

INTRODUCCION A LOS MICROPROCESADORES Y SUS APLICACIONES.

T ERMINOLOGIA

AGOSTO, 1979.

GENERAL TERMS and DEFINTIONS

~~Access Time~~ (Read and/or Write)	The time interval between the instant data is inserted into or requested from memory and the instant that data transfer is completed (may be the longest path.)
Accumulator	A temporary storage register(s) that (stores) sums and other arithmetic and logical operations of an arithmetic logic unit (ALU.)
Address	A character or group of bits that identifies a particular location in memory, or other locations of data sources and destinations.
Addressing Modes	The methods of specifying the location(s) of data or program segments in memory or other locations.
Arithmetic and Logic Unit (ALU)	That part of a CPU that performs arithmetic, logical and related operations. Along with memory and control, an essential microprocessor element.
Architecture	The organizational structure of a computing system. Refers mainly to physical makeup of the microcomputer (Section 2) or microprocessor (Section 10) parts in this volume.
Assembler	A computer program used to translate a symbolic-language statement to a machine-language statement on a one-for-one basis.
Assembly Language	A programming language utilizing symbolic representation. The symbolic representation, sometimes called mnemonics, suggests the instruction function and is translatable by the assembler into machine language.
Asynchronous Operation	A switching network operation with no common timing source. Circuit operation is such that the completion of one event initiates the next.
Baud	A measure of serial data transmission flow in communications applications. Baud rate generally defines signal bits per second transmitted, but may also include character framing bits.
Benchmark Problem	A frequently used (sample) problem employed to compare and evaluate computers (microcomputers.) Permits comparison of the number of instructions, memory words, and operation cycles required to solve the same problem.
Binary Coded Decimal (BCD)	A number coding system in which each decimal digit is represented by a 4-bit binary word. The decimal number 13 becomes the coded-number 0001 0011 in BCD using an 8-4-2-1 binary code.
Bit	The abbreviation of "binary digit" and the single characters in a binary number (1 or 0).

GENERAL TERMS and DEFINITIONS

A decision-making capability that permits a processor to select one from a number of alternative sets of instructions depending on the data being processed.

A circuit employed to minimize the effects of a following circuit on the preceding circuit.

One or more conductors used for transmitting signals or power from one or more sources to one or more destinations.

A pre-determined binary element string (number of consecutive bits) operated on as an entity. A byte is usually but not necessarily 8-bits.

it The unit of a computing system that includes circuits controlling the interpretation and execution of instructions.

A generator of periodic signals used to synchronize circuit operations.

A computer program used to translate a high-level language program (e.g.,) FORTRAN) into a computer oriented (assembly or machine) language program.

A group of program conditions such as carry, borrow, overflow, etc. that are particularly relevant to instruction execution.

A bus carrying the signals that regulate system operation within and without the computer.

A sequence of instructions that directs the central processing unit (CPU) in the various operations it performs.

A computer program used to translate symbolic-language programs assembled on one computer into machine-language programs that operate on another computer.

A method of interrupt priority in which the interrupt bus is searched serially.

A bus used to communicate data internally and externally to and from the CPU, memory, and peripheral devices.

A register holding the memory address of the operand used by an instruction. The data pointer "points" to the memory location of the (data) operand.

GENERAL TERMS and DEFINITIONS

Data Register	Any register that holds data.
Debug (Program)	A computer program designed to aid in detecting, tracing and eliminating errors in microcomputer (or other computer) programs while they are running. Allows replacing, adding, or revising instructions into the main operating program.
Decrement	A program instruction that decreases the contents of a storage location.
Dedicated Microprocessor	A microprocessor that has been programmed for a specific, single application.
Diagnostic Program	A computer program designed to check the operation of various hardware and software parts of the microcomputer sytem. Typically written for each functional area; e.g., CPU diagnostics for CPU checks, Memory diagnostics for Memory checks, etc. . .
Direct Addressing	An addressing mode in which the address of the instruction or operand is completely specified in the instruction without reference to a base register or index register.
Direct Memory Access (DMA)	A method of inserting input/output data into storage or obtaining input/output data from storage directly, without involving the usual flow of data through the processor registers.
Editor (Program)	A computer program designed to allow manipulation of source program text material.
Emulate	To imitate one system with another, the latter being microprogrammable and equipped with a special microprogram, so that the imitating system executes the same programs and achieves the same results as the imitated system. (See Simulate.)
Execution Time	The time, normally expressed in clock cycles, required to carry out an instruction.
Fetch	The reading out of an instruction from main memory and the insertion of the instruction into working memory.
Firmware	Programming instructions stored in a read-only-memory (ROM.)
Flag Bit	An information bit that indicates the occurrence of special conditions such as — overflow, carry, interrupt
Flow Chart	A graphical representation of a problem and the operations to be accomplished to solve the problem.
FORTRAN	A high-level programming language used to facilitate the expression of computer programs in arithmetic terms and formulae. Short for "Formula Translator."
Handshaking	A colloquial term that describes the method used by a modem (or other asynchronous devices) to establish a communication link for eventual data transmission.

GENERAL TERMS and DEFINITIONS

A problem-oriented programming language where a single functional statement may translate into a series of instructions in machine language (a low-level language.) FORTRAN, COBOL, and BASIC are common high-level languages.

A program instruction that increases the contents of a storage location.

An addressing mode in which the operand is located in the instruction itself, or the memory location immediately following the instruction.

A register that provides a programming flexibility in that the information it contains can be used to modify memory address by addition or subtraction.

An addressing mode in which the address portion of an instruction is modified by an index-register during instruction execution. A means of changing an instruction address on the basis of external commands.

An addressing mode that specifies a memory location containing the address of data and not the data itself.

In a programming language, an expression that defines a computer operation and identifies its operands.

The time required in fetching an instruction from memory and executing it.

The measure of the memory space required to store an instruction.

The total list of instructions that can be executed by a given microprocessor.

A program that fetches and immediately executes instructions written in a higher level language. (See Compiler and Assembler.)

An external signal that temporarily suspends the normal program operation in order to permit processing of a high-priority operation. Multiple interrupt capability requires establishment of an interrupt priority system.

General term applied to equipment and/or data involved in connecting the central processing unit (CPU) with the outside world. The control electronics necessary to tie the computer to various external application areas.

A connection to a central processing unit (CPU) wired or programmed to connect data between the CPU and external devices, i.e., keyboard, display, card reader, etc. May be an input, output, or bidirectional port.

GENERAL TERMS and DEFINTIONS

Jump	An unconditional departure from the normal instruction sequence to a different place in the program.
Loader (Program)	A computer program designed to read (enter) a program from an input device into working storage. (Random Access or Read/Write Memory — RAM.)
Look Ahead	A central processing unit (CPU) feature that allows the computer to mask (ignore) an interrupt request until the following instruction has been executed.
Loop	A sequence of instructions in which the final instruction causes repetition of the sequence a number of times until a termination condition as detected by a branch instruction is reached.
Machine Language	The numeric form of specifying every bit of every instruction in a program. The final binary program code that a computer uses directly.
Machine Cycle	The basic central processing unit (CPU) cycle in which; an address may be sent to memory and one word (data or instruction) may be read or written; or in which a fetched instruction may be executed One machine cycle is made up of several clock cycles.
Macro Instruction	A symbolic source assembly language statement that combines any group of frequently used commands (instructions). The macro instruction will be expanded by the assembler into several machine-language instructions.
Memory Address Register	The register in the central processing unit (CPU) that contains the address of the storage (memory) location being accessed.
Microcomputer	A computer system whose central processing unit (CPU) is a microprocessor. A basic microcomputer includes a microprocessor, memory, and input/output facility.
Microinstruction	An element in a microprogram.
Microprocessor	A single integrated circuit chip, or set of chips, capable of performing the functions of a computer central processor.
Microprogram	The computer instructions stored in a read-only-memory (ROM) that implement the basic instruction set of the computer.
Microprogrammable Computer	A computer in which the microprogram (microinstructions) in the read-only-memory (ROM) can be changed, thus changing the computer's instruction set.

GENERAL TERMS and DEFINITIONS

Symbolic names or abbreviations for instructions, registers, memory locations, etc., suggesting the definition or function thereof.

Devices or equipment employed to interface data processing equipment with communication lines.

Simultaneous execution (by use of multiple CPU's operating with a common memory) of multiple programs by one computer.

Calling a subroutine from, or enclosing a program loop within, a larger routine or loop. For a subroutine the nesting level is the number of times the subroutine is nested.

The output from a compiler or assembler which is itself executable machine code or is suitable for processing to produce executable machine code.

The field of the instruction that represents the specific operation to be performed.

The data with which a mathematical or other operation is performed.

A computer program that controls the overall operation of a computer system, including such things as memory allocation, input/output distribution, interrupt processing, job scheduling, etc.

A bit in the condition code register that indicates if the previous operation in the program caused an arithmetic overflow, i.e., caused a quantity to be generated that exceeded the capacity of the results register.

A natural grouping of memory locations, e.g., 2^8=256 consecutive bytes in an 8-bit microcomputer may typically constitute a "page" of memory.

The processing of all the bits of a word (byte) simultaneously by transmitting on separate channel or bus lines.

Auxiliary equipment (in the outside world) used to enter data in and receive data from a (microl-computer.

Temporary suspension of a microcomputer program to permit execution of a program or part of a program of higher priority.

Central processor unit (CPU) registers that contain memory addresses. Sometimes called Data Pointers or Program Counters.

A method used to identify the source of an interrupt request. In polling, the interrupt request search is done serially.

Port (I/O Port)	Terminals that provide electrical access between the (microlcomputer and the outside world.
Program Counter	A register in the central processing unit (CPU) whose job is to step the (microlcomputer through the (microlprogram. This register holds (saves) the address of the instruction under execution in the event of program interrupt or branching activity.
Programmable Logic Array (PLA)	An array of logic elements whose interconnections can be programmed (usually mask programmable, some field programmable) to perform a specific logical function.
Programmable Read-Only Memory (PROM)	A memory array manufactured with a constant logical pattern (all ones or zeros) and that can be written into (programmed) by the user. EAROM's (electrically alterable ROM's) can be electrically erased and reprogrammed. EPROMS can be erased and reprogrammed using ultraviolet light and electrical signals.
Push Down Stack	See "Stack."
Random Access Memory (RAM)	A memory device that has read and write capability and that provides direct access to stored information, regardless of location. Read or write time is independent of data location
Read	To acquire and/or to interpret data from a memory device.
Register	Small scale memory capable of holding a fixed amount of data, such as a word. Used mainly as temporary storage, and accessible to the central processing unit (CPU.) The number of registers in a microprocessor is directly related to the program efficiency.
Relative Addressing	An addressing mode in which the address is determined by adding an offset address to a base address stored in some register. Permits the computer to relocate blocks of data by changing only one number.
Read Only-Memory (ROM)	A memory device generally used to store a computer's control programs (firmware), and not alterable by normal operating methods. Exceptions are PROM's, EPROM's and EAROM's, which are ROM's whose contents may be altered in the field by special means.
Scratch Pad Memory	Read/Write (RAM) memory devices or registers that are used to store temporarily intermediate results (data), or memory addresses (pointers.)
Simulate	To imitate one system with another using a program written in assembly or high-level language so that the imitating system accepts the same data, executes the same programs, and accomplishe the same results as the system being imitated. (See Emulate.)
Slice	A partition of a microprocessor that enables several identical units to be paralleled or cascaded and augmented by control logic to realize the CPU.
Software	A collection of computer programs, procedures, rules and documentation used or associated with the operation of the computer.

GENERAL TERMS and DEFINITIONS

A computer program in a high-level or assembly language.

A sequence of registers or memory storage locations accessible in a last-in-first-out (LIFO) basis.

The counter or register that addresses the current stack location.

A computer system that is complete within itself and does not require connection to another computer system to operate.

A group of instructions (subprogram) that may be reached from more than one place in a main program with a branch back to the source point when the subprogram task is completed.

Circuit operation using a common timing source (clock) to time circuit or data transfer operations.

See "Branch Instruction"

A device used in communications applications to interface (connect) a word parallel data terminal (controller) to a bit serial communication network.

An interrupt system in which each interrupt can be immediately serviced without having to determine which interrupt has occurred by polling.

In a given (micro)computer, the most common storage entity for instructions and data.

To record data.

centro de educación continua
división de estudios superiores
facultad de ingeniería, unam

INTRODUCCION A LOS MICROPROCESADORES Y SUS APLICACIONES

ARTICULOS 'SPECTRUM' IEEE

AGOSTO, 1979

Increased capabilities, architectural compatibility, and clearly defined interfaces were the chief architectural goals of Zilog's new Z8000 microprocessor family. Here is an account of how those goals were met for two members of that family—the Z8000 CPU and the MMU.

The Z8000 family is a new set of microprocessor components (CPU, CPU support chips, peripherals, and memories) which supports the Z8000 architecture. The account of how architectural goals were selected and achieved for two key members of this family—the Z8000 CPU and the memory management unit—illustrates how much of a challenge microprocessor architecture represents to the semiconductor industry. MOS technology shows enormous potential, but it is still difficult to use because of limitations on pin count, power dissipation, speed, and complexity.[1]

Since this discussion is restricted to technical issues, we will not allude to the many additional factors (marketing considerations, human considerations, self-imposed restrictions, etc.) which make architecture such a fascinating and difficult discipline. Furthermore, no attempt has been made to exhaustively describe the Z8000 architecture and components. Interested readers should consult the specific manuals for a more complete description.[2,3]

The goals of the Z8000 architecture: increased capabilities, architectural compatibility, increased clarity

The primary reason for introducing a new system architecture is to significantly improve the control and processing capabilities of microprocessors while maintaining their price/performance advantages. Technical advances have permitted the implementation of substantially increased processor power, but the most significant motivation for a new component family is generality. Only through such a family could we provide for architecturally compatible growth over a wide range of processing power requirements.

Our approach was a staged system architecture which attempts to provide new components, enhanced features, and new functions, while protecting the user's investment in hardware and software. The Z8000 family supports a single unified architecture for all small, medium, and high-end user applications which are implemented using a mix of components within the same family.

The goals of the Z8000 architecture can be grouped into three categories: increased capabilities, architectural compatibility over a wide range of processing powers, and increased clarity. In all these cases the resulting architectural features apply either to the basic architecture (that seen by an applications programmer) or to system architecture (that seen by a system designer or an operating system programmer).

Increased capabilities. All existing 8-bit microprocessors and many 16-bit minicomputers suffer from having a small address space. So, one of our goals was to provide access to a large address space (8M bytes). A second goal was to provide more resources in terms of registers (16 general-purpose 16-bit registers), in terms of data types (from bits to 32 bits), and in terms of additional instructions compared to existing microprocessors (multiply and divide, multiple register saving instructions, specialized instructions for compiler support etc.).

To facilitate complex applications it was important to support multiprogramming with good hardware support of task switching, interrupts, traps, and two execution modes. Operating systems also required a good hardware protection system.

Finally, we wanted to increase overall system performance. This resulted in the choice of an implementation using a 16-bit-wide data path to memory.

0018-9162-79/0820-0010$00.75 © 1979 IEEE

Architectural compatibility. One of the important lessons learned from previous computer system designs is that the design of a new family architecture is a rare occurrence. One way to apply this lesson is to design a unified architecture compatible over a wide range of processing powers. If we anticipate user growth from small to large systems within a family architecture, then such an approach can significantly increase its life.

The two versions of the Z8000 (a 40-pin unsegmented and a 48-pin segmented version) are designed to achieve this goal, but many other features contribute indirectly to the family compatibility. For small aplications an unsegmented Z8000 with one or more 64K-byte address spaces can be used. For medium applications, a segmented Z8000 and one memory management unit allows direct access to 4M bytes of address space. For large applications a segmented Z8000 and multiple pairs of MMUs allow the use of several 8M-byte address spaces.

Since the segmented Z8000 can run in an unsegmented mode, both systems are compatible. Finally, to achieve even larger processing power through hardware replication, the architecture provides basic mechanisms for both multiprocessing and distributed processing.

Clarity. Clarity in an architecture is a measure of how well key interfaces are defined and specified. This is an elusive but important goal in a family where new and unforeseen components will be added during the life of its architecture.

We felt bus protocols were so important that we developed an independent specification for the Z-bus along with the individual device manuals.

Clarity in terms of the basic architecture means regularity and extendability of the instruction set, as well as the general and simple handling of the operating system interfaces. Clarity in terms of the system architecture means a well-defined method of communication between the various components. The key link between these components is the Z-bus, which is a shared system bus. In the section on communication with other devices, we describe some of the various types of bus protocols. At Zilog we felt this was so important that we developed an independent specification for the Z-bus along with the individual device manuals.[4]

Comparison with other system architectures

We are convinced that the differences between microprocessor system architecture and large computer system architecture are not sufficient to require a different design approach, although they certainly influence the details of design compromises. The last section of this paper deals with implementation tradeoffs and illustrates some particular compromises. (In a few places we mix implementation considerations with descriptions of architectural tradeoffs. Despite the importance of separating an architecture from its implementation, we found that this separation is often absent during the actual creation of a new architecture.)

Two differences between conventional computer systems and microprocessor systems have the greatest impact: price structure and component boundary differences. For high-end LSI systems, it makes sense to have one unified architecture, but unlike their computer family counterparts (IBM 360/370, PDP-11) different implementations cannot be justified on a price/performance basis. Speed and performance are mainly dependent on the state of technology, and therefore, for a given application, a user will waste the speed willingly since another slower implementation would cost the same. This does not exclude different versions of one implementation, which reflect only different test and production criteria such as package type, functional temperature range, and even speed range.

Most computer systems have both external and internal interfaces. External interfaces which define system boundaries are often standardized (e.g., the IBM channel interface or the DEC unibus). The internal interfaces of most mini or large computer systems are essentially hidden. In contrast, the component boundaries of a microprocessor-based system represent actual interfaces, and most users must be familiar with them as well as with external interfaces. Because the component interfaces are more visible and often must be more general, the microprocessor-oriented system bus emerges as a key standardization link to allow a wider mix of components and designs.

The basic architecture

Address space considerations. It is advantageous to have more than one address space, with each address space as large as possible. In the Z8000, memory references and I/O references are viewed as references to different address spaces. The I/O space is discussed in the section below on communication with other devices. Memory references may be instructions or data and stack accesses, with each type of access possible in either system or normal modes. The Z8000 distinguishes between each of these reference possibilities by using different combinations of its status lines. Separating the various address spaces can be used to increase the total number of addressable bytes and to achieve protection. The size of each address space depends on the versions of the Z8000 used. The 40-pin package version allows each address space to be at most 64K bytes, the 48-pin package version allows each address space to be at most 8000K bytes.

The 40-pin version is intended for systems, often used as dedicated systems, where the program and data spaces are small. In this case, relocation is not usually important. Using the different address spaces, one has a simple way to address in practice up to 4 x 64K bytes (with a maximum of 6 x 64K bytes). Some simple protection is achieved by separating these spaces in hardware.

The 48-pin version with one or more MMUs is intended for the medium to large applications where relocation and better memory protection are important.[3] In these cases, status information can also be used to separate between address spaces by using multiple MMUs. But it is also essential to achieve the detailed memory protection required. (It is possible to use the 48-pin version without an MMU.) For these high-end applications, the address spaces are so large that one is unlikely to exhaust them. Experience with large computers shows that 8M bytes is probably adequate. The current implementation of the Z8000 uses 8M-byte address spaces, but the architecture provides for 31-bit address (2147M bytes).

In both versions, the Z8000 allows direct access to each address space. Direct access means that the addresses used in instructions or registers have as many bits as the address space size requires. In other schemes the effective address is a combination of a shorter field in the instruction and other extension bits often found in an implied register. Despite the shorter address fields, we believe this "indirect access" does not save bytes, because extra instructions must be used to load and save the implied registers, which are typically in short supply.

Registers. The Z8000 is primarily a memory-to-register architecture. This characteristic does not entirely exclude other organizations, and mechanisms exist in the Z8000 to support them. For example, memory-to-memory operations are supported for strings, whereas stack operations are supported for procedure and process changes. This choice provides upward compatibility with the Z80. A register architecture also results in good performance, since register accesses are made at a greater speed than memory accesses in the current implementation.

Experience with register-oriented machines seems to confirm that four general-purpose registers are not enough and that a "proper" number is between eight and 32.[5] The Z8000 supports bytes, words (16-bit), and long words (32-bit), and a few instructions even use quadruple-word (64-bit) data elements. If we choose 16, 16-bit registers allow eight 32-bit registers as well as four 64-bit registers (Figure 1). Since addresses are 32 bits, the necessity of at least eight 32-bit registers was obvious. The impact of the 4-bit register field on the instruction format depends also on the number of address modes and operands. Sixteen registers allowed a reasonable tradeoff, whereas 32 registers would have resulted in too few one-word instructions.

With one minor restriction any register can be used by any instruction as an accumulator, source operand, index, or memory pointer. This regularity of the structure is so important that it is worthwhile to sacrifice any possible encoding improvements in instruction formats which could result from dedicating registers to special functions. Encoding improvements based on instruction frequency, so that frequent instructions use one word, are more effective in saving space without having a negative effect on the architecture.

Why not have specialized registers? The difficulty lies in the fact that the restrictions caused by dedication are inconsistent with one another.

Most applications dedicate the available registers to specific functions. For example, most high-level languages require a stack pointer and a stack frame pointer. Then why not, one might argue, have specialized registers? The difficulty lies in the fact that the restrictions caused by dedication are inconsistent with one another. If the architecture supplies only general-purpose registers, the user is free to dedicate them to specific usages for his application without restrictions. This is important in the context of microprocessors where user applications are not well known and where high-level languages are still used infrequently.

For example, the Z8000 allows software stacks to be implemented with any register. There are also two hardware supported stacks, but the registers used are still general-purpose and can participate in any operation. There is no allocated stack frame pointer, since any register can be used by means of the proper combination of addressing modes. The savings realized by register specialization are unattractive when the given function can still be performed simply. The loss that would result from restricting the applications would be too great. In contrast, significant savings result from excluding R0 from use as an index or memory pointer. This exclusion allows one to distinguish between the indexed and direct addressing modes which use the same combination of the instruction address mode field. The price is small, since R0 still can be an acumulator or source register and 15 others accumulator, index, and/or memory pointers are available. In this case the restriction made sense.

Another decision to be made about registers is their size. Since the architecture handles multiple data types we must have multiple data register sizes, which can hold each data type. The solution of the problem is implemented in the architecture by pairing registers, two 1-byte registers make a word register, two word registers make a long word register, etc.

Data types. Users would like to have as many directly implemented data types as possible. A data type is supported when it has a hardware representa-

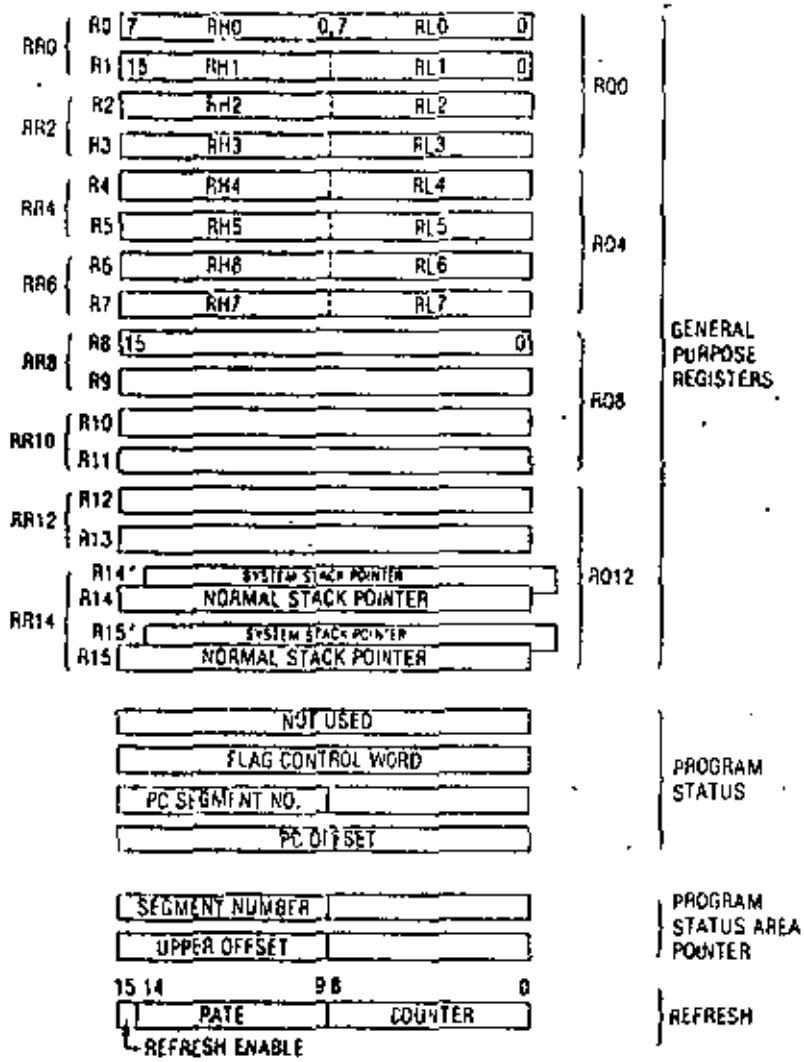


Figure 1. CPU registers (segmented version).

tion and instructions which directly apply to it. New data types can always be simulated in terms of basic data types, but hardware support provides faster and more convenient operations. At the same time, a proliferation of fully supported data types complicates the architecture and the implementations.

The Z8000 supports several primitive types in the architecture and provides expansion mechanisms. The basic data types are obviously the ones expected to be used most frequently. The extended data types are built using existing data types and manipulated using existing instructions.

The basic data type is the byte, which is also the basic addressable element. All other data types are referenced using their first byte address and their length in bytes. The architecture also supports the following data types: bytes (8 bits), words (16 bits), long words (32 bits), bytes, and word strings. In addition, bits are fully supported and addressed by number within a byte or word. BCD digits are supported and represented as two 4-bit digits in 1 byte. One consequence of this data type organization is that byte, word, and long-word registers are needed to support them. The Z8000 even provides quadruple register—another extension—used in long-word manipulation.

Other data types are supported by using one of the preceding data types; for example, addresses are manipulated as long words, and each element (segment number or offset) can be manipulated as a byte or a word. Instructions are one to five words strings, the program status is four words, etc.

As the family grows, support for new data types will be added. The architecture will need to support them in its registers or in memory if they do not fit in registers (as strings are implemented today). But most important, the architecture will have to support the addition of new instructions to its repertoire.

Instructions. In designing an instruction format the architect must decide how to allocate a limited number of bits to the opcode field, address mode field, and other operand subfields. Instruction usage statistics are the best source of data to influence decisions about instruction set format.[1, 6, 7] Behind their usage lies a strong technical position: we do not

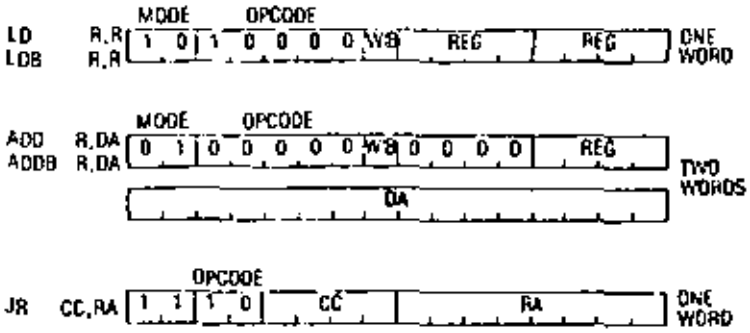


Figure 2. Examples of instruction formats (nonsegmented version).

believe that any one of the various instruction set structures--register oriented, memory oriented, stack oriented, symmetrical, or asymmetrical, etc.—are always better when used exclusively. Thus the task of the architect is to decide what his most important goals are, and for each of them adapt the best features of the various structures so that on the average, and for his set of goals, an optimum solution can be found. We do not believe that the optimum will be very sharp; it will be more like a range of applications for which the resulting composite structure works well. We decided to use a register structure for compatibility, multiple word instructions for speed, memory-to-memory instructions for strings, stack structure for process control and procedure support, "short" instruction for byte density improvement, etc.

Instruction format consideration. The Z8000 has over 110 distinct instruction types; several instruction formats are illustrated in Figure 2. The opcode field specifies the type of instruction (for example, ADD and LD). The mode field indicates the addressing modes (for example, Register (R), Direct Address (DA). The data element type (W/B) and register designator fields complete the basic instruction fields. Long word instructions use a different opcode value from their byte or word counterpart. Frequent instructions are encoded in a single word, and less frequent instructions which use more than two operands use two words. There are often additional fields for special elements such as immediate values or condition code descriptors (CC). Instructions can designate one, two, or three operands explicitly. The instruction TRANSLATE AND TEST is the only one with four operands and is also the only one with an implied register operand.

Several restraints can guide the proper choice of an instruction format. A large number of opcodes (used or reserved) is very important: having a given instruction implemented in hardware saves bytes and improves speed. But one usually needs to concentrate more on the completeness of the operations available on a particular data type rather than on adding more and more esoteric instructions which, if used frequently, will not significantly affect performance. Great care must be given to the problem of expanding the instruction set so, for example, new data types can be added.

Addressing modes. The Z8000 has eight addressing modes: *register* (R), *indirect register* (IR), *direct address* (DA), *indexed* (X), *immediate* (IM), *base address* (BA), *base indexed* (BX), and *relative address* (RA). Several other addressing modes are implied by specific instructions such as autoincrement or autodecrement.

Although a very large number of addressing modes is beneficial, usage statistics demonstrate that not all combinations of operands, address modes, and operators are meaningful.[5] The five basic addressing modes of R, IR, DA, X, and IM are the most frequently used and apply to most instructions with more than one address mode. For two-operand instructions, statistics show that most of the time the destination is a register. Other cases of addressing mode combinations and less basic addressing modes are associated with special instructions. Thus, the frequent combination of autodecrement for the destination operand with the five basic address modes for the source operand is provided by the PUSH instruction. The combination of autoincrement addressing modes for both source and destination operands is one of the block move instructions. In essence, the address mode field space has been traded for opcode field space. This allows more instructions and combinations while staying within a one-word format.

The price for this tradeoff is the infrequent occurrence of pairs or triples of instructions simulating a missing addressing mode. This situation occurs in most instruction sets in any case.

Code density. Because current microprocessors are restricted to primitive pipeline structures, their speed is largely dependent on the number of executed instruction words. Therefore, code density is not only important because of program size reduction but also because of speed improvement. One would like to encode in the smallest number of bits the most frequent instructions. The basic instruction size increment was chosen to be a word for reasons dealing with alignment, speed penalties, and hardware complexity. Thus the most frequent one and two-operand instructions take one word in their register or register-to-register forms. Less frequent instructions or instructions which use more than two operands use at least two words.

The Z8000 goes even further by selecting several special instructions as "short" instructions which take only one word, when normally they would take two words. These instructions, such as LOAD BYTE REGISTER IMMEDIATE and LOAD WORD REGISTER IMMEDIATE (for small immediate values), CALL RELATIVE, and JUMP RELATIVE, are so frequent statistically that they deserve such special treatment.

A one word JUMP RELATIVE and DECREMENT AND JUMP ON NON-ZERO also have a very significant impact on speed. The short offset mechanism used by addresses (and described below) is also designed to allow one-word addresses. Compared to previous microprocessors, the largest reduction in size and increase in speed results from the Z8000's consistent

and regular structure of the architecture and from its more powerful instruction set—which allows fewer instructions to accomplish a given task.

High-level language support. For microprocessor users, the transition from assembly language to high-level languages will allow greater freedom from architectural dependency and will improve ease of programming.[8] It is easy and tempting to adapt a computer architecture to execute a particular high-level language efficiently.[9] Most programming languages act as a filter and can be supported by a subset of available hardware with greater efficiency.[10] But efficiency for one particular high-level language is likely to lead to inefficiency for unrelated languages. The Z8000 will be used in a wide variety of applications, and we know that a large number of users will still be using assembly languages. Since the Z8000 is a general-purpose microprocessor, language support has been provided only through the inclusion of features designed to minimize typical compilation and code-generation problems. Among these is the regularity of the Z8000 addressing modes and data types. The addressing structure provided by segmentation should support procedures that result from structured programming. Access to parameters and local variables on the procedure stack is supported by index with short offset address mode as well as base address and base indexed address modes. In addition, address arithmetic is aided by the INCREMENT BY 1 TO 16 and DECREMENT BY 1 TO 16 instructions.

Testing of data, logical evaluation, initialization, and comparison of data are made possible by the instructions TEST, TEST CONDITION CODES, LOAD IMMEDIATE INTO MEMORY, and COMPARE IMMEDIATE WITH MEMORY. Compilers and assemblers manipulate character strings frequently, and the instructions TRANSLATE, TRANSLATE AND TEST, BLOCK COMPARE, and COMPARE STRING all result in dramatic speed improvements over software simulations of these important tasks, especially for certain types of languages. In addition, any register can be used as a stack pointer by the PUSH and POP instructions.

Segmentation. In order to provide for convenient code generation and data access, addresses must also be easy to manipulate. Architectures with direct access to memory typically use a linear address space, so that address arithmetic may be used on the entire address. In this case, addresses are manipulated as one of the data types of the same size. This removes the need to distinguish an address as a new data type. In contrast, the Z8000 has a non-linear address space. Addresses are made of two parts: a 7-bit segment number and a 16-bit offset. Only the offset participates in address arithmetic. The segment number is essentially a pointer to a part of the total address space, which can vary in size from 0 to 64K bytes. The hardware representation of a segmented address is a long word or a register pair (Figure 3), which allows the easy manipulation of each part of the address.

The segmented addresses are one of the key mechanisms used to support both large and small

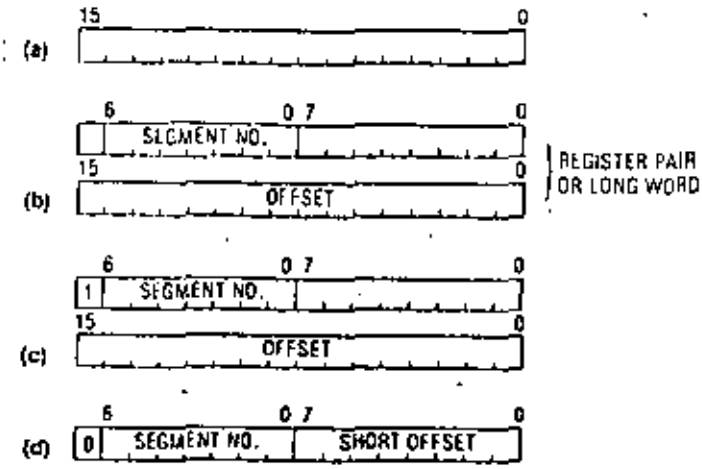


Figure 3. Hardware representation of segmented addresses. Any nonsegmented address is one word, whether it is in a register, memory, or an instruction (Figure 3a). Segmented addresses are always two words in a register or memory (Figure 3b); however, instructions can have one of two forms. The usual case (long offset) requires two words (Figure 3c); however, there is also a short offset form that uses only one word (Figure 3d).

memory systems efficiently. The two versions of the Z8000 implementation, the 40-pin unsegmented and the 48-pin segmented, allow the maintenance of the architectural compatibility and ease the growth between these two application groups. The segmented address space guarantees that each 64K-byte address space of the 40-pin version becomes one of the segments of the 48-pin version. Each 40-pin version's 16-bit address becomes an offset within the segment, and a mode exists in the 48-pin package version in which 40-pin version code can be executed. Furthermore, compatibility with any current 8-bit microprocessor such as the Z80 is easy, and a new microcomputer such as the Z8 can address external data in a shared segment with the Z8000.

The hardware performance of the Z8000 is also improved by address segmentation. Since a segment number does not participate in arithmetic, it can be put on the bus before the result of an address computation is available. This feature allows the use of MMUs with essentially no impact on memory access time by allowing it to function in parallel with the CPU. Indexing operations are also faster because only a 16-bit addition must be performed. Because of the distinction between the segment number and its offset, one can use shorter addresses without software constraints. Short addresses can use a short offset (fewer than 256 bytes) and thereby reduce program size (Figure 3).

Finally, it is very easy to associate with each of the 128 segments of the address space the protection and dynamic relocation features desirable for larger systems. Relocation allows a user to write his application using logical addresses independent of any physical addresses. Relocation is essential, for example, in a disk-based general data processing system with several users. Relocation is not essential for dedicated applications with code typically residing in

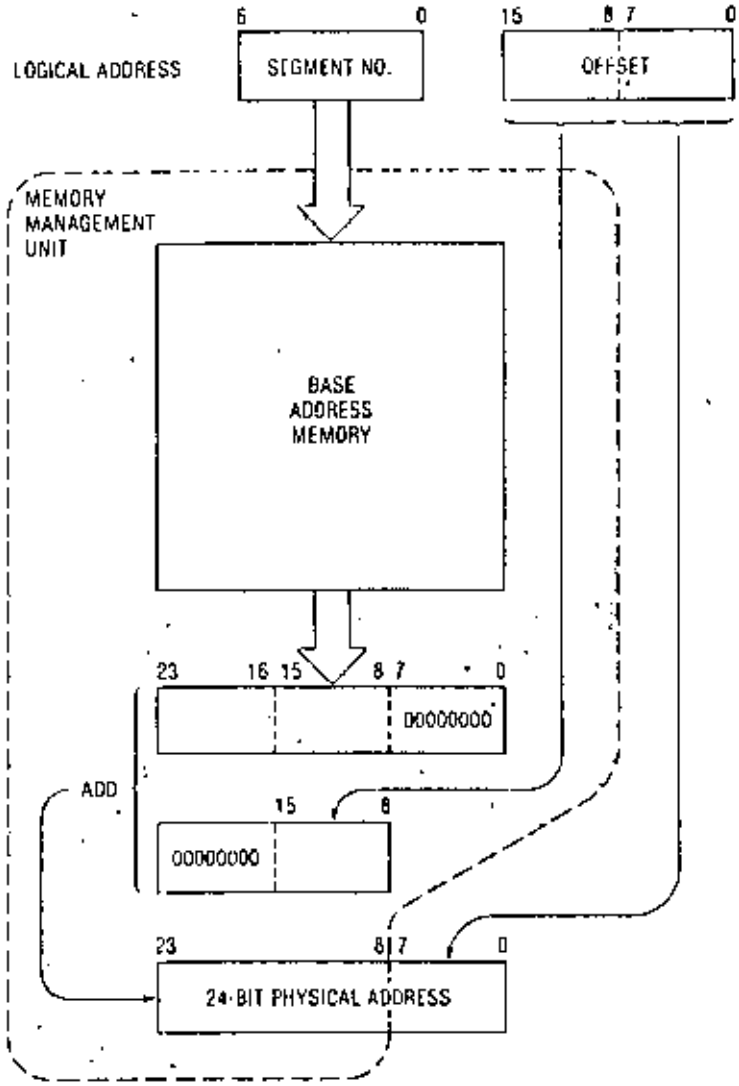


Figure 4. Logical to physical address translation.

ROM. Users whose total memory needs are small are also unlikely to need relocation.

In summary, the choice of a segmented address space has provided—at low cost and with few practical limitations—a powerful solution to the problem of user growth, relocation, and protection as well as virtual memory implementation. We believe that a linear address space could have achieved these results but at a considerably higher price.

The system architecture

Protection facilities. The Z8000 protection facilities can be divided into system protection features and memory protection features. Experience with large computers has demonstrated the advantages of having at least two execution modes with different access rights to hardware facilities. The Z8000 provides the system and normal modes for this purpose. A simple protection system results from the presence of these two modes and their associated stacks. A special class of "privileged" instructions is defined, which deals with I/O, interrupts, traps, and mode changes. Programs in normal mode which attempt to execute a privileged instruction will cause a trap and a change to system mode. The switch from user to system mode can also be caused by the system call instruction. These mechanisms enforce protection and help in designing reliable and efficient operating systems with clean user interfaces. Several other traps are provided to achieve a consistent system: segmentation trap, privileged instruction trap, and undefined instruction trap.

A desirable memory protection scheme is one for which protection information (read only, read write, execute only, system only, size of data or code, etc.) is easily associated with the data and code structures of a given application. It is also one for which a large number of different types of protection information can be verified.

The relocation and memory protection mechanisms described above are provided by an external device: the memory management unit.[2] To provide relocation and protection features directly on the Z8000 would have demanded too much simplification. The external MMU has the further advantage of providing for easier growth by the addition of components. The Z8000 40-pin package does not have to carry the burden of the unused advanced relocation and protection features, although some form of protection can be achieved by hardware separation of the different address spaces. With multiple MMUs, the 48-pin package user can control the relocation and protection complexity desired in his application.

The memory management unit. The MMU performs three functions: (1) address translation of logical address to physical address using dynamic relocation, (2) memory protection, and (3) segment management. The addresses manipulated by the programmer, used by the instructions, and output by the Z8000 are called logical addresses. The MMU uses these logical addresses, composed of a 7-bit segment number and 16-bit offset, and transforms them into a 24-bit physical address (Figure 4). A 24-bit origin or base is logically associated with each segment. To form a 24-bit physical address, the 16 bit offset is added to the base for the given segment. In effect, with the help of one memory management device, the Z8000 can address 8M bytes directly within a 16M-byte physical memory space. The reasons for the choice of a large physical address space include an expectation that large systems will want to use extra bits for complex resource management purposes.

Each segment is given a number of attributes when it is initially entered into the MMU. When a memory reference is made, the protection mechanism checks these attributes against the status information from the CPU. If a mismatch occurs, a trap is generated which interrupts the CPU. The CPU can then check the MMU status registers to determine the cause of the trap. Segment attributes include segment size and type (read only, system only, execute only, in-

valid DMA, invalid CPU, etc.) Other segment protection features include a write warning zone useful for stack operations.

When a memory protection violation is detected, a write inhibit line guarantees that memory will not be incorrectly changed. The invalid DMA and CPU bits indicate that the entry cannot be used by the DMA or CPU respectively, because either the segment number is illegal or the segment entry is not loaded. This fast feature, in conjunction with the segment history information (segment "changed" and segment "referenced" bits) and the segmentation trap mechanism, allows the implementation of a virtual segmented memory system.

The MMU comes in a 48-pin package (Figure 5). The chip inputs are the segment number, the upper 8 bits of the offset, and status information from the CPU. The outputs from the segment chip are the upper 16 bits of the 24-bit physical address and the segmentation trap line. Since the memory management device processes only the upper 8 bits of the offset, the lower 8 bits go directly to memory. This is equivalent to having zeros in the 8 lower bits of the 24-bit origin. Thus, the memory management device only needs to store the upper 16 bits of each base address. Segment limit protection is done in the memory management device, and thus segments can be protected in increments of 256 bytes.

Each MMU stores 64 segment entries that consist of the segment base address, its attributes, size, and status. A pair of MMUs support the 128 segments available in an address space. Additional MMUs can be used to accommodate multiple translation tables. Using the status information provided with each reference, pairs of MMUs can be enabled dynamically.

The memory management device functions constantly while memory references are made, but its translation and protection tables are loaded and unloaded as an I/O peripheral. To achieve this, the memory management device has chip select, address strobe, data strobe, and read/write lines. The Z8000 special byte I/O instructions that use the upper byte of the data bus can load or unload the memory management device.

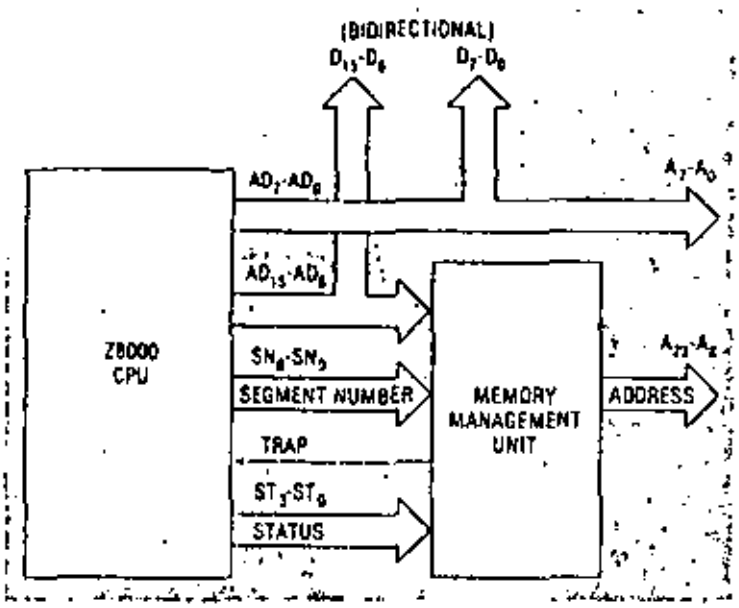


Figure 5. Memory management device with Z8000 CPU.

Mode switching: interrupt and trap handling. From small users in dedicated process control applications to large users in general-purpose data processing applications, asynchronous events such as interrupts and synchronous events like traps must be handled. When these events occur, the state of any currently executing program must be saved during what is generally called a task switch or process switch. The users benefit from the availability of multiple interrupts and traps. They also benefit from a fast, easy, and uniform handling of process switching.

Peripherals using interrupts have widely varying constraints on interrupt processing time. To solve this problem, peripherals with the same characteristics are often associated with one of several interrupts. A priority enforced among the several interrupts allows the required processing time to be guaranteed. Enabling or disabling the various interrupts is the mechanism used to enforce this processing priority.

In the Z8000, we felt that three levels of interupts were sufficient. A *non-maskable interrupt* represents a catastrophic event which requires special handling to preserve system integrity. In addition there are two maskable interrupts: *non-vectored interrupts* and *vectored interrupts*, which correspond to a fixed mapping of interrupt processing routines and to a variable mapping of interrupt processing routines depending on the vector presented by the peripheral to the Z8000.

Both interrupts and traps result in similar process switches. Information related to the old process (its program status) is saved on a special system stack with a code describing the reason for the switch. This allows recursive task switches to occur while leaving the normal stack undisturbed by system information. The state of the new process (its new program status) is loaded from a special area in memory—the program status area—designated by a pointer resident in the CPU (see Figure 6).

The use of the stack and of a pointer to the program status area are specific choices made to allow architectural compatibility if new interrupts or traps are added to the architecture. The choice of the two modes of execution has a strong impact on the design of clean user interfaces. Experience has shown that in large systems the normal mode instruction set and the user interfaces together constitute the most important element in achieving architectural compatibility.

Communication with other devices: the Z-bus. The Z-bus is the shared bus which links all the components of the Z8000 family.[4] The variety and performance requirements of the components are so different that in fact the Z-bus is composed of five buses:

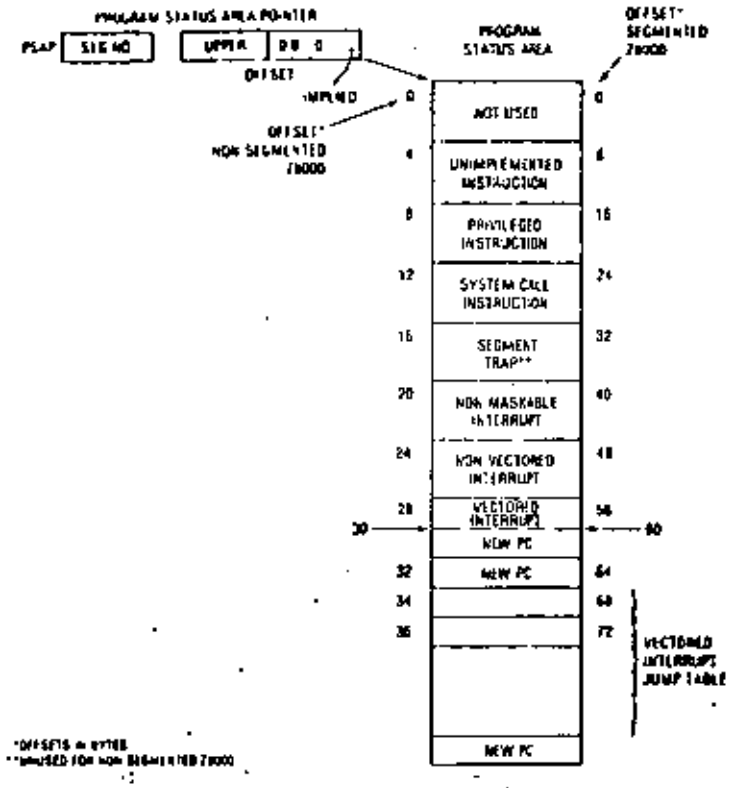


Figure 6. Program status area.

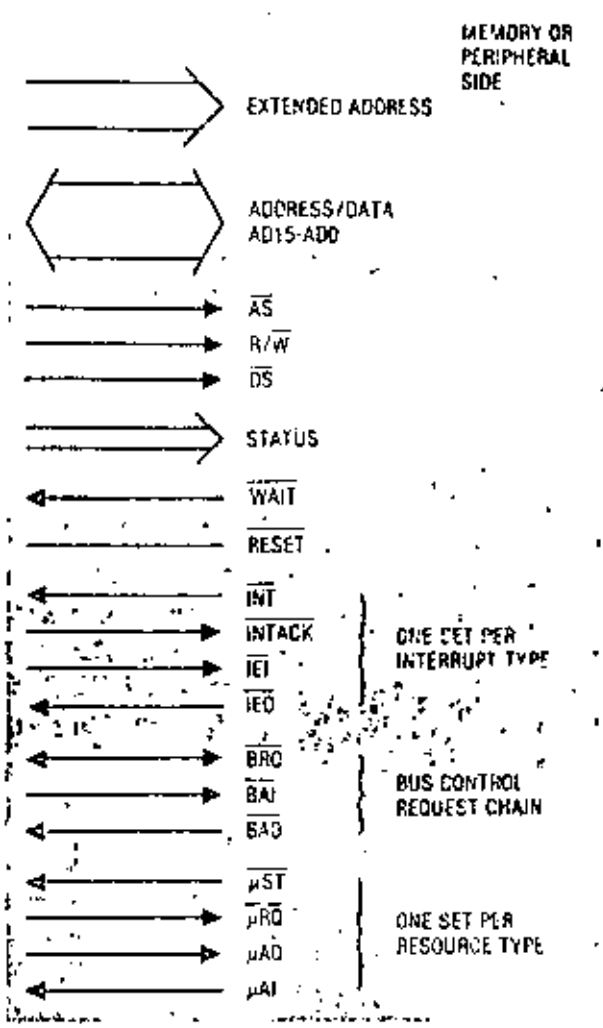


Figure 7. Z-bus signals.

a memory bus, an I/O bus, an interrupt bus, and two resource request buses (Figure 7).

The Z-bus is called a "shared" bus because several components can use it. A bus user is a CPU or a peripheral which can usually generate one or more bus transactions such as memory data request or an I/O request. Identical bus transactions cannot take place at the same time, but serialization mechanisms allow sequential use of the Z-bus. Architecturally, the buses can be grouped into two structures. The I/O structure uses the I/O bus and the interrupt bus. The memory structure uses the memory bus with or without address extensions. Both structures can use the resource request bus and the mastership request bus.

Each bus consists of a set of signals and the protocols which preside over the various types of transactions. Part of each protocol is the timing relationship between relevant signals. The Z8000 CPU provides most of these timing relations. The advantage of such a choice is the significant reduction in the number of components required to build such a system. One consequence is that bus transactions cannot be aborted or delayed freely since some devices, especially memory, have specific timing constraints. The most important consideration for the Z-bus is the need to interface to multiplexed address and data lines of the Z8000 CPU which must fit in 40- and 48-pin packages. The Z-bus maintains these multiplexed address and data lines. Very little speed could be gained by demultiplexing these lines for memory references since memories are themselves multiplexed. The most important advantage of a multiplexed Z-bus is the direct addressability of peripheral internal registers. This feature allows the construction of complex peripherals which maintain a simple program interface.

The Z-bus is known as a transparent or asynchronous bus. Z8000 components do not require that their clocks be synchronized with the CPU clock. The signals used by each transaction provide all the necessary timing. This concept is important: it allows, for example, I/O references to be independent of the speed and clock frequencies required by other Z-bus transactions.

I/O bus versus memory bus: The I/O and memory buses are the most important. The Z8000 family architecture distinguishes between memory and I/O spaces and thus requires specific I/O instructions. This architectural separation allows better protection and has a nicer potential for extension. The I/O and memory buses use a 16-bit address/data bus, which allows 16-bit I/O addresses and 8- or 16-bit data elements. Memory addresses are 16 bits for the 40-pin package or extended to 23 bits using the segmented version. Thus, the memory bus is in fact a logical address bus. The increased speed requirements of future microprocessors is likely to be achieved by tailoring memory and I/O references to their

respective characteristic reference patterns and by using simultaneous I/O and memory referencing. These future possibilities require an architectural separation today. Memory-mapped I/O is still possible, but we feel the loss of protection and potential expandability are too severe to justify memory-mapped I/O by itself.

Both the I/O and memory buses need address, data, and control signals. One important implementation decision was to overlap the signals used by the memory and I/O buses on the same Z8000 CPU pins, with the obvious exception of the status signals used to distinguish between the two types of bus requests. For the current Z8000 implementation the resulting reduction in number of pins is significant. In contrast the impossibility of doing concurrent memory and I/O referencing is not very significant since their speeds are essentially the same.

In addition, memories and peripherals both benefit from the availability of early status information defining the bus transaction type (I/O versus memory, read versus write) ahead of the actual transaction so that bidirectional drivers and other hardware elements can be enabled before the reference. The status lines of the Z8000 CPU provide this type of early status.

The I/O structure. Since many peripherals are connected with one CPU, the I/O bus is shared and serialization must be provided. One solution involves using a master/slave protocol. The CPU is a master which can initiate an I/O transation at any time. The peripherals are slaves which participate in a transaction only when requested by the master. In order to find out if a peripheral needs to be serviced the master can poll each in turn. The Z-bus also provides a faster way of getting the attention of a master: an interrupt bus. In contrast, with the I/O transaction data bus, each peripheral sharing the interrupt bus may "try" to use it simultaneously. The interrupt bus uses an interrupt line, interrupt acknowledge line, and two more lines used to form a daisy chain. The daisy chain is an implementation of a distributed arbitration policy between the requests. Priority of processing is determined by the position in the daisy chain, and peripherals can be preempted. Interrupt vectors are used to determine the identity of the peripherals requesting service via an interrupt.

Other buses. The two resource request buses are used to request the control of the Z-bus from the CPU and to request control of any generalized resource.

The Z8000 CPU or any Z-bus compatible CPU does not need to request the bus to access it as a master, and is, therefore, the default master. Other devices do request bus mastership, but they must go through a non-preemptive distributed arbitration using another daisy chain. The CPU always relinquishes the bus at the end of its current bus transaction.

The resource request chain is a generalization of that concept in which each resource requestor has equal importance and can use the resource in a non-preemptive manner. This mechanism in the Z8000 CPU permits one to implement in software the kind of exclusion and serialization mechanisms needed for multiple distributed systems with critical resource sharing.

Multiprocessing. In the context of today's large mainframe systems characterized by multiple processes sharing one processor, one is tempted to design distributed processing systems with many low-cost microprocessors running dedicated processes. Such an approach distributes intelligence towards the peripherals, results in modularization, and permits easier development and growth. Unfortunately, in the past, the problem with such an approach has been software and not hardware. Thus one cannot be expected to provide detailed solutions in hardware to a software problem that has not been solved yet. However, some basic mechanisms have been provided to allow the sharing of address spaces: large segmented address spaces and the external MMU make this possible, and a resource request bus is provided which in conjunction with software provides the exclusion and serialization control of shared critical resources. These mechanisms and new peripherals like the Z-FIO have been designed to allow easy asynchronous communication between different CPUs.

Implementation tradeoffs

The key family decision: producibility. Confronted with the problem of designing a new LSI-based system architecture, we could have ignored package size considerations by accepting packages with 64 or more pins, or we could have ignored mass production technology constraints by using die sizes larger than 260 mils square. Such solutions are often justified in the implementation of an existing computer system. The component boundaries, package limitations, and technological limitations are secondary to achieving the goal of exact membership in the computer family. But if one were to design a new system architecture with the same lack of constraints, the individual component would not be price-competitive—only the total system would be. A new system architecture based on this approach could only be used to design yet another traditional computer.

The Z8000 family provides basic, general-purpose blocks out of which a system solution to most problems can be implemented.

The Z8000 family market is intended to be much broader, and each component of the family must be economically viable. The staged introduction of components which are economically viable by themselves allows us to serve the market from very small configurations to very large configurations by using more components, in any combination. Not only do we believe that this approach does not restrict

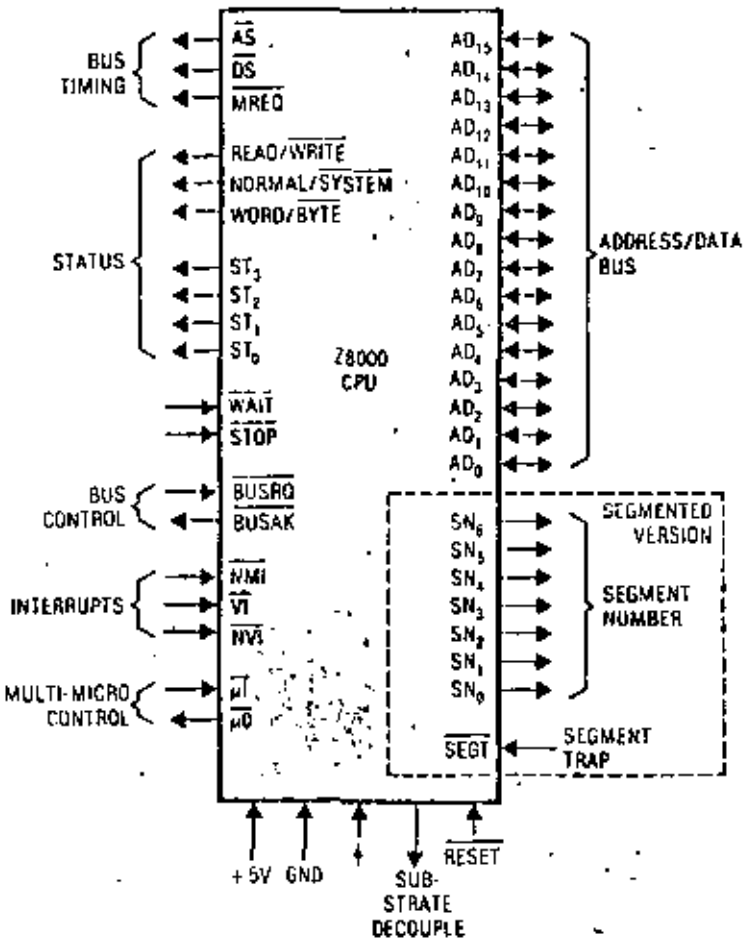


Figure 8. Z8000 pin functions.

system architectural possibilities, but we also believe that the family will be more effective because it will grow with its customer.

The Z8000 family does not always attempt to provide specific architectural solutions, often implemented in hardware, to all system architecture problems. Instead, it provides basic, general-purpose blocks out of which a system solution to most problems can be implemented. The multi-microprocessor and distributed system capabilities of the Z8000 family illustrate the use of open-ended mechanisms to solve a variety of architectural problems, while the memory management of address space illustrates a specific problem supported by a specific solution—the MMU. However, other solutions more appropriate to a particular problem can be used and an advance in the state of the art might be mapped into a new device for the family.

This vision of the family often results in components more powerful and complex than an application may require. The user should not take this as a cause for alarm, but rather as the reason his applications growth will be easier.

Basic CPU implementation decisions. The Z8000 currently uses a 16-bit data bus (Figure 8), an internal register array of 16-bit registers, and a 16-bit parallel ALU. These implementation decisions, which were guided by the technological and practical considerations, have a strong impact on performance.

To achieve good performance with the instruction format and data type envisioned for the Z8000, only a 16-bit bus seems adequate; a 32-bit bus would have necessitated using an unacceptable 56-pin or larger package. Optimal performace is obtained with this chosen bus width if the size of the frequently used register-to-register operations becomes one word. The choice of ALU and internal register width is a tradeoff between speed of the most frequent operations and the chip area needed to implement a wider ALU or data path inside the CPU.

None of these implementation decisions should limit the architecture. Instructions are from one to five words long, and data types and addresses are not limited to 16 bits. For example, 32-bit words are one of the main data types of the machines, and addresses occupy two words. The *address* mechanism illustrates the strong distinction between an architecture and its implementation. The architectural address representation uses a 32-bit word of which 8 bits are reserved and 1 is a short format/long format descriptor. Thus, the Z8000 architecture provides up to 31-bit addresses, but only 23 are currently implemented and 23 pins of the current package are allocated to addresses.

MMU tradeoffs. The MMU and its relation to the Z8000 CPU illustrate tradeoffs that a microprocessor architect and designer team must make to ensure component manufacturability.

To achieve the goals of good architectural compatibility for high-end systems, it was necessary to include the protection and relocation mechanisms described above. But if all desired features were implemented as a one-chip CPU/MMU combination, it would have been too large and, therefore, uneconomical. And if a reduced set of features were implemented, it would have been architecturally too primitive. Thus, the choice was made to maintain all features and use two chips. This new organization has several significant advantages, such as a capability for multiple MMUs, and allows the access of a DMA device to the MMU.

Given the choice of an external MMU, the next set of decisions concerns package size and circuit speed. Having each relocated segment start on a word boundary would have required a 64-pin package and a very fast 24-bit adder (in fact, a 16-bit adder and 8 bits of carry propagation). In contrast, the decision to start segments on 256-byte boundaries allows the use of a 48-pin package, a fast 8-bit adder, and 8 bits of carry propagation. The latter solution is technically superior and places practically no restriction on the architecture. Segment granularity can be viewed as an implementation restriction and not as an architectural restriction.

Making the 8 low-order bits of the offset go directly to memory also significantly reduces memory access time. Since dynamic memories use these bits first, most of the MMU relocation time is hidden during a

normal memory access. The availability of segment numbers earlier than the associated offset bits reinforces this advantage and allows the MMU to result in essentially no memory access speed reduction. Each MMU entry also requires 8 bits less for base and segment size value. This is important: it is desirable to pack as many entries as possible per MMU. With 64 entries a 2K-bit memory is needed, which is technologically difficult in view of the amount of logic surrounding this memory and the complexity of its organization.

The fact than an MMU is only connected to the upper byte of the data bus requires the use of special I/O instructions for its loading and obliges us to replace the possible use of an automatic demand loading of entries by explicit instruction loading. To compensate for the time penalty associated with the loading of potentially unused entries, multiple MMUs are used. They not only allow the implementation of 128 entries, but pairs of MMUs can be automatically enabled by the system and normal mode pins effecting a full environment switch at electronic speed.

We feel this example illustrates one important design approach: to compromise as little as possible on advanced architectural features but to accept compromises which result in implementation ease in order to achieve economical components.

Conclusion

The architectural sophistication of the new 16-bit microprocessors is rapidly approaching the level of the minicomputer and large computer. Problems such as component families, large address spaces, bus standards, I/O structures, software investments, and architectural compatibility are being directly addressed. Some of the solutions to these problems are known, and therefore the transition from 8-bit microprocessors was relatively easy. But the challenges ahead—networks, distributed processing, new applications—are much harder. The impact of microprocessors is already enormous, but we feel they will achieve the often-predicted computer revolution only after these new problems are solved. ■

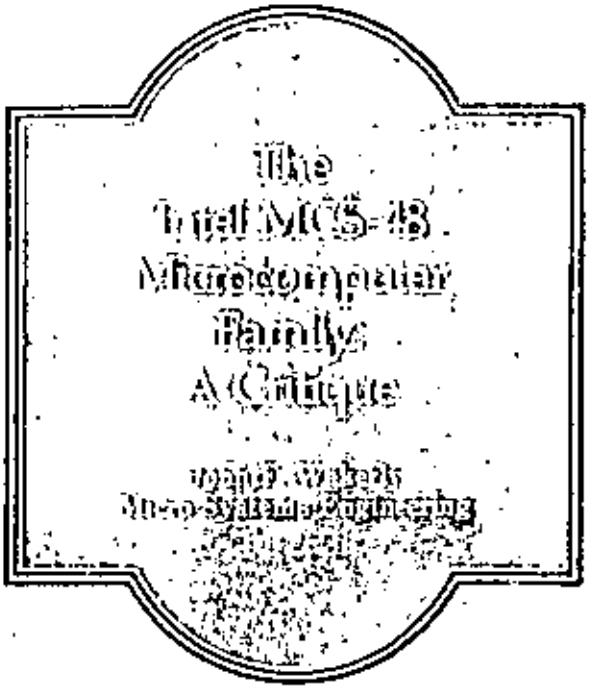
Acknowledgements

The Z8000 family would not exist without the very talented and dedicated designers who contributed to and implemented the ideas described in this paper: Masatoshi Shima for the Z8000, Hiroshi Yonezawa for the MMU, and Ross Freeman for the peripheral devices. Judy Extrin made invaluable contributions to the architecture of the Z8000 and Z8. Many discussions with Charlie Bass, Leonard Shustek, and Forest Baskett have greatly influenced the Z8000. Leonard's instruction set measurements were especially valuable. Dennis Allison, Steve Meyer, Bruce Hunt, and many others must be thanked for their comments on early drafts of this paper.

References

1. B. L. Peuto and L. J. Shustek, "Current Issues in the Architecture of Microprocessors," *Computer*, Vol. 10, No. 2, Feb. 1977, pp. 20-25.
2. Zilog, *Z8000 Technical Manual*, Zilog, Inc., 1979.
3. Zilog, *MMU Technical Manual*, Zilog, Inc., 1979.
4. Zilog, *Z-Bus Specification*, Zilog, Inc., 1979.
5. A. Lunde, "Empirical Evaluation of Some Features of Instruction Set Processor Architectures," *CACM*, Vol. 20, No. 3, Mar. 1977, pp. 143-152.
6. L. J. Shustek, *Analysis and Performance of Computer Instruction Sets*, PhD Dissertation, Dept. of Computer Science, Stanford University, Stanford, Calif., Jan. 1978.
7. B. L. Peuto and L. J. Shustek, "An Instruction Set Timing Model of CPU Performance," *Proc. Fourth Annual Symposium on Computer Architecture*, Mar. 23-25, 1977, pp. 165-178.
8. C. Bass, "PL/Z: A Family of System Programming Languages for Microprocessors," *Computer*, Vol. 11, No. 3, Mar. 1978, pp. 34-39.
9. A. S. Tannenbaum, "Implications of Structured Programming for Machine Architecture," *CACM*, Vol. 21, No. 3, Mar. 1978, pp. 237-246.
10. N. G. Alexander and D. B. Wortman, "Static and Dynamic Characteristics of XPL Programs," *Computer*, Vol. 8, No. 11, Nov. 1975, pp. 41-46.

Bernard L. Peuto is one of the guest editors for this special section; his biography appears with the Introduction on p. 9.



The Intel MCS-48 Microcomputer Family: A Critique

A system designer and teacher, who has made liberal use of microcomputers in his own work and whose students have designed 8048 processors, reviews the capabilities and limitations of the MCS-48 family of microcomputers.

The Intel MCS-48 family of single-chip microcomputers contains at least nine different microcomputer chips having a common instruction set but different amounts of on-chip read-only memory, read/write memory, and input/output (see Table 1). According to Intel, the MCS-48 family was originally aimed primarily at the "4-bit market"—users of Intel's 4040 and other low-cost microcontrollers. Recent entries into the family (the 8021, 8022, 8041, and 8741) are increasingly specialized for low-end microcontroller applications. The MCS-48 family has met this market very well.

The MCS-48 family was also aimed at a second market—applications that require an expandable, single-chip, general-purpose microcomputer. As shown in Table 2, several expansion chips are available to provide an MCS-48 computer with up to 4K bytes of program ROM, 256 or more bytes of external RWM, and as many I/O bits as a designer would ever need. In addition, the external I/O bus of the MCS-48 family allows easy interfacing of standard 8050/8085-compatible peripheral chips. Nevertheless, the architecture of the MCS-48 family makes it difficult to use in many general-purpose applications, where a more capable 8-bit architecture is required.

Table 1.
MCS-48 microcomputers.

PART #	PACKAGE SIZE (pins)	ON-CHIP PROGRAM MEMORY (bytes)	ON-CHIP DATA MEMORY (bytes)	I/O (lines)
8048	40	1K ROM*	64**	27
8748	40	1K EPROM*	64**	27
8035	40	none*	64**	27
8049	40	2K ROM*	128**	27
8039	40	none*	128**	27
8021	28	1K ROM	64	21
8022	40	2K ROM	64	23 plus 2 8-bit A/D conv.
8041	40	1K ROM	64	18 plus master sys. intf.
8741	40	1K EPROM	64	18 plus master sys. intf.

* Expandable to 4K with external chips
** Plus 256 bytes or more of external data memory with external chips

Table 2.
MCS-48 expander chips.

PART #	PACKAGE SIZE (pins)	ON-CHIP PROGRAM MEMORY (bytes)	ON-CHIP DATA MEMORY (bytes)	I/O (lines)
8355	40	2K ROM	none	16
8755	40	2K EPROM	none	16
8155/56	40	none	256	22 plus timer/counter
8243	24	none	none	16

Basic architecture

Figure 1 shows the basic structure of an MCS-48 microcomputer chip. (Table 1 gives the facilities available for each of the microcomputers in the MCS-48 family that had been announced by late 1978.) The first member of the family was introduced in late 1976—the 8048 with 1K bytes of on-chip ROM, 64 bytes of RWM, timer/counter, and 27 I/O bits. A detailed description of the entire family can be found in the user's manual published by Intel.[1]

The MCS-48 is a single-accumulator architecture. Program memory and data memory are logically and physically separated (thus, the MCS-48 is not a von

Neumann machine!). The maximum program address space (including external ROM) supported by the architecture is 4K bytes. There are a maximum of 256 bytes of on-chip (internal) data memory, of which 128 bytes are implemented in the current family leader, the 8049. In addition to internal data memory, the MCS-48 directly supports 256 bytes of external data memory.

Most MCS-48 family members have 27 I/O pins, arranged as three 8-bit ports, two test inputs, and an interrupt input. Additional pins are provided for such functions as power-on reset, single-stepping, and memory and I/O expansion strobes. One 8-bit port and part of a second are used to form a multiplexed address and data bus for I/O and memory expansion.

The MCS-48 has a single-level interrupt system (only one interrupt in service at a time) and accepts interrupts from two sources—its internal timer/counter and an external interrupt input pin. Interrupt calls and returns automatically push and pop the program counter and certain internal status flags using a stack in the internal data memory.

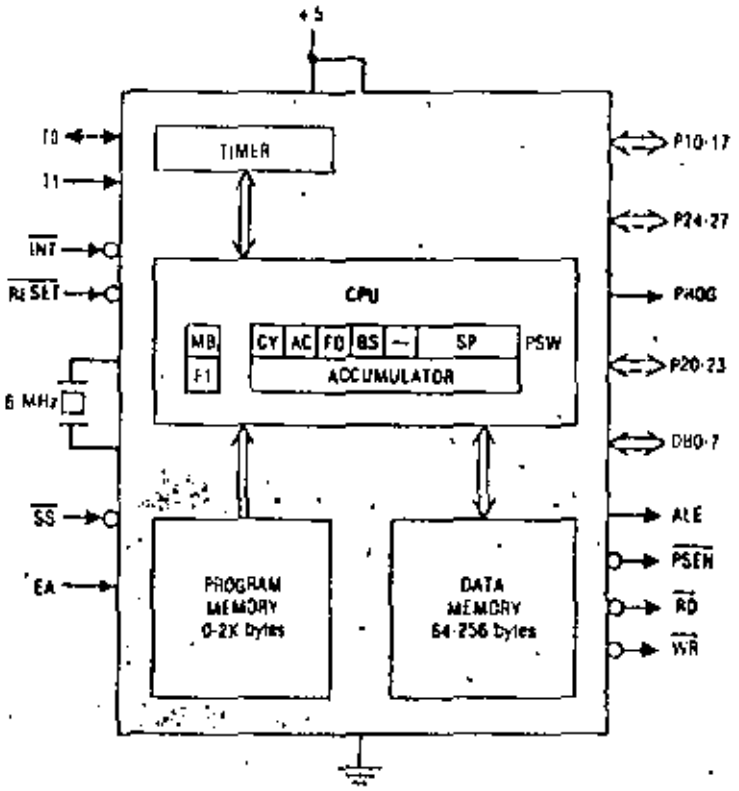


Figure 1. Basic structure of a typical member of Intel's MCS-48 family of microcomputers.

Program store and program control

The MCS-48 architecture supports a maximum of 4K bytes of program store, configured as shown in Figure 2. However, a close look at program-store organization shows that the MCS-48 was originally designed as a 2K-byte machine, with the second 2K-byte capability added as a clumsy afterthought. This creates two problems with the addressing mechanism.

First, the program counter is really only 11 bits and thus addresses instructions only within a 2K-byte bank of program store. Jump and subroutine call instructions likewise specify an 11-bit address. The problem, then, is how to provide a 12th address bit.

Intel's solution is as follows. Provide an internal flag, MB, that can be set and cleared by two instructions (SEL MB1 and SEL MB0, respectively). Whenever a jump or subroutine call is executed, take the 11 low-order PC bits from the instruction, and load the high-order bit from MB. On subroutine calls and returns, push and pop the entire 12-bit address.

There are some problems with this solution. First, in a general sequence of jumps and calls in a 4K system, we don't always know where we came from, and therefore we don't know the current value of MB. So in general, a SEL MBi instruction must precede every jump or call. Naturally the programmer can sometimes avoid this instruction on a case-by-case basis, but this is another thing to worry about.

Having solved the first problem, we think we understand the addressing mechanism until we write our first interrupt routine. Then we wake up in the middle of the night thinking, "Whoops! MB can't be read as part of the processor state PSW. But MB must be set to a new value in order to do jumps within the interrupt routine. How can the old value be restored on return?" We lie awake a few hours dreaming up possible solutions—don't use calls or jumps in in-

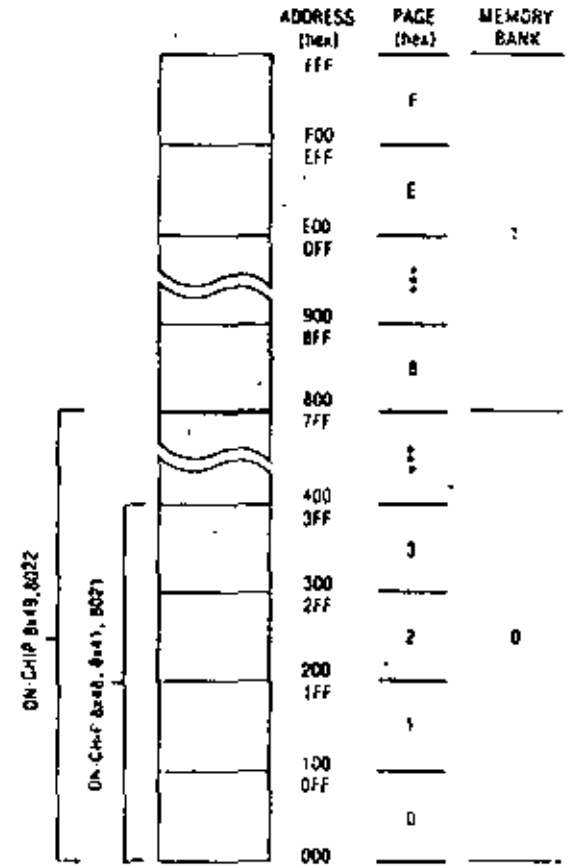


Figure 2. MCS-48 program memory.

How to choose a microcomputer

There are at least six factors to consider in choosing a microcomputer (or microcomputer family) for a product application.

Capability. The μC must have enough ROM, RWM, I/O capability, and speed to satisfy the requirements of the application, plus a design margin. ROM, RWM, and I/O capability can be determined from the manufacturer's literature, while speed is best determined from benchmark programs tailored for the given application.

With some μCs, the amount of ROM, RWM, and I/O can be increased by using extra chips. This expandability helps if the job is initially underestimated or if the marketing department changes the requirements. If the application pushes the absolute memory limits of the μC, it becomes more difficult (and expensive) to develop the programs.

Extensibility. The designer must consider whether improved versions of the μC will be offered. A μC-based product designed in 1979, for example, might be redesigned in 1981 to reduce cost or to add features. It would then be desirable to eliminate extra ROM, RWM, and I/O chips (or avoid having to add them or pick a different μC) by using a new version of the original μC with the extra capability built in. Of course, many products do not undergo this evolution; but if product evolution is expected, the architectural limits of the selected μC should be examined in light of potential application requirements. One can expect lower hardware and software development costs and, probably, lower manufacturing costs if the enhanced product uses an upgraded version of the original μC rather than a completely new one.

Cost. In most areas of the private sector, minimizing cost is a goal, and minimizing μC cost usually means minimizing the number of IC packages. Cost is what prevents the designer from picking a Cray-1 in response to the first factor above, or an IBM 370 in response to the second factor.

If the job is well defined and no product enhancement is anticipated, it is relatively easy to find a minimum-cost μC that will do the job. Otherwise, there are many more tradeoffs to be considered. A simpler μC architecture usually implies a smaller IC die and lower chip cost, but it may also require more chips to support it later. (For example, a μC without a WAIT/READY line may be more difficult to interface to some types of peripherals or memory.) An expandable μC will facilitate later product evolution (if the product is successful), but may increase initial product cost because of instruction efficiency, memory size, I/O pins, or speed sacrificed by the chip designers to make expansion or enhancement possible.

Availability. Many manufacturing organizations require a second source for all components, both to ensure that parts will be available, even if some disaster befalls one source, and to enjoy the normal benefits of competition in a free market.

The designer of a new product is often tempted to select between a μC with one or two sources and one with *no* sources (yet—"We'll have samples in three months"). It is risky to commit to any part unless your purchasing department can order (and receive) 100 pieces from a distributor's shelf. Manufacturers *have* been known to slip schedules and even cancel parts.

On the other hand, marketing and cost factors can motivate the selection of a not-yet-available or very new μC. The new μC can give the product a competitive edge in features or performance. Although the new μC may be in short supply and costly initially, it may be cheaper in the long run because it allows a more efficient design with fewer IC packages.

Expected product lifetime should also be compared with the expected lifetime of the μC. Even if it is inexpensive currently, a μC that has been around for a few years may be a bad choice: production quantities may fall and prices rise in a few more years as newer chips are phased into new designs. Of course, this doesn't apply if your company alone is ordering 100,000 pieces per year.

Support tools. Hardware and software support tools are essential for timely development of a μC-based product. The support tools of a newly introduced μC cannot be expected to be as extensive or reliable as those of an established μC family. This encourages the use of an established μC if quick development is needed, or an extensible μC family with reusable tools if product evolution is expected.

Most single-chip μCs are programmed in assembly language, and a good macroassembler is a must. Most manufacturers supply software tools that run on their own development systems. However, if there are more than one or two programmers on the project, the need for good text editors, simulators, and documentation facilities makes it desirable to run all software support tools on a large central computing facility. The appropriate "cross assemblers" and simulators may or may not be offered by the chip manufacturer.

During product development it is obviously necessary to test and change programs running on the product hardware. Since most single-chip μCs ultimately use mask-programmable ROM to store their programs, another means is needed to store and change programs during development without making new masks. Some μCs have pin-compatible versions with on-chip EPROM instead of ROM that allows reuse of the μC chip with different programs. Many have provisions for using external EPROM chips instead of the on-chip ROM. If production quantities are low, or if software changes are expected after product introduction, EPROM versions may be essential.

Besides EPROM facilities, the main support tool provided by the chip manufacturer is the in-circuit emulator, which stores the software program in the RWM of a development system and emulates the μC through a cable and plug inserted in place of the μC in the product. An emulator is a useful tool for debugging both hardware and software. However, with new μCs, it may not be available as soon as the μC chips are, and even if it is, it may still have bugs.

Specific technical factors. Many specific technical factors can be examined in determining whether a μC will do the job at hand—power consumption, speed, TTL compatibility, package size, instruction set. However, once it is determined that the μC can do the job, the other factors above tend to equal or outweigh the technical "niceness" of the μC chip architecture.

terrupt routines; determine the value of MB experimentally by doing a jump to a fixed location and seeing whether it winds up in MB0 or MB1; keep a software copy of MB, updating it (with interrupts disabled, of course) every time we do a SEL MBi; make all the code fit in 2K, as we expected to do at the start of the project; and so on. The next morning we read the fine print to discover that the MCS-48 forces the most significant bit of the program counter to 0 during all interrupt routines. We should put all of our interrupt code in the bottom 2K of memory and not touch MB; in fact, we should forget that MB exists!

The requirement to put interrupt code in the bottom 2K makes the MCS-48 very difficult to use as a 4K machine in a real-time application. Not only must the basic interrupt service routine be in the bottom 2K but also any utility routine that might be called by it—that is, any code executed before an interrupt return instruction is executed. This could be well over half the code in an interrupt-driven environment.

But the main problem we find with MCS-48 program store, after writing half of our applications programs, is that the address space is just too small. With only two chips (and soon with just one, I'm sure) we can fill the entire 4K-byte address space of the MCS-48 with code for our original application, new features, diagnostics, and—of course—patches.

Conditional jumps specify an 8-bit target address in the current page; it would be far more useful to have a signed offset from the current address.

The annual halving of the cost of IC memory implies that every year we will need another address bit for the maximum-size application program (since most evolving products tend to use the decreased memory cost to increase features, not to reduce product cost). Clearly, then, a 4K limit is too low for any new architecture, even a single-chip microcomputer.

Besides the 2K memory banks, program store is also divided into 256-byte pages. Conditional jumps specify an 8-bit target address in the current page. It would be far more useful to have a signed offset from the current address; this would increase the likelihood of being able to use the short jump address, since most branch targets are within 128 bytes of the branch instruction.[2] More importantly, it would eliminate the partitioning problem created when many procedures must be packed into the memory space and split across page boundaries.

The only indirect jump instruction also uses an 8-bit target address in the current page. Very strangely, this instruction uses an 8-bit value in the accumulator not as the target address, but as a pointer to a program-store byte in the current page that contains the target address. So the page containing the indirect jump instruction must also contain all of the routines to be jumped to, as well as a silly little table that contains their starting addresses in the page. This not only wastes space and time, but, worse, makes it impossible to dynamically compute a target address after assembly time, since the jump table is in ROM. In my recent experience, three experienced programmers have coded MCS-48 indirect jumps improperly, believing "The manual must have a typo—the contents of the accumulator must be the target address itself." In any case, instructions supporting indirect jumps and calls anywhere in the program store (12-bit address) would be far more useful.

Arithmetic and logical operations

The MCS-48 contains a single accumulator in which arithmetic and logical operations take place. Unary operations on the accumulator are as follows:

increment,
decrement,
clear,
one's complement,
decimal adjust,
swap nibbles,
rotate left,
rotate left with carry,
rotate right, and
rotate right with carry.

Binary operations combine the accumulator and an operand specified by one of the addressing modes described in the next section. The binary operations are:

add,
add with carry,
AND,
OR, and
exclusive OR.

There are also "data-move" operations that load or store the accumulator.

The main difficulty with MCS-48 operations is not the operations themselves but the lack of condition codes for testing their results. Only the accumulator can be tested for zero or negative, and an overflow bit is not provided, making comparisons of signed two's-complement numbers very frustrating.

Operands

Most data moves and binary operations use an on-chip read/write internal data memory (see Figure 3) accessible by two addressing modes: REGISTER and INTERNAL REGISTER INDIRECT. The three other addressing modes are EXTERNAL REGISTER INDIRECT, IMMEDIATE, and ACCUMULATOR INDIRECT.

In REGISTER mode an operand is contained in a register specified by a 3-bit field in the instruction. A flag bit BS, set by a SEL RBi instruction, specifies one of two 8-byte register banks, corresponding to internal data memory locations 0-7 if BS is 0 and 24-31 if BS is 1. The specified register may be loaded with an immediate value, moved to or from the accumulator, combined with the accumulator by arithmetic or logical operations, incremented, decremented, or used as a loop counter.

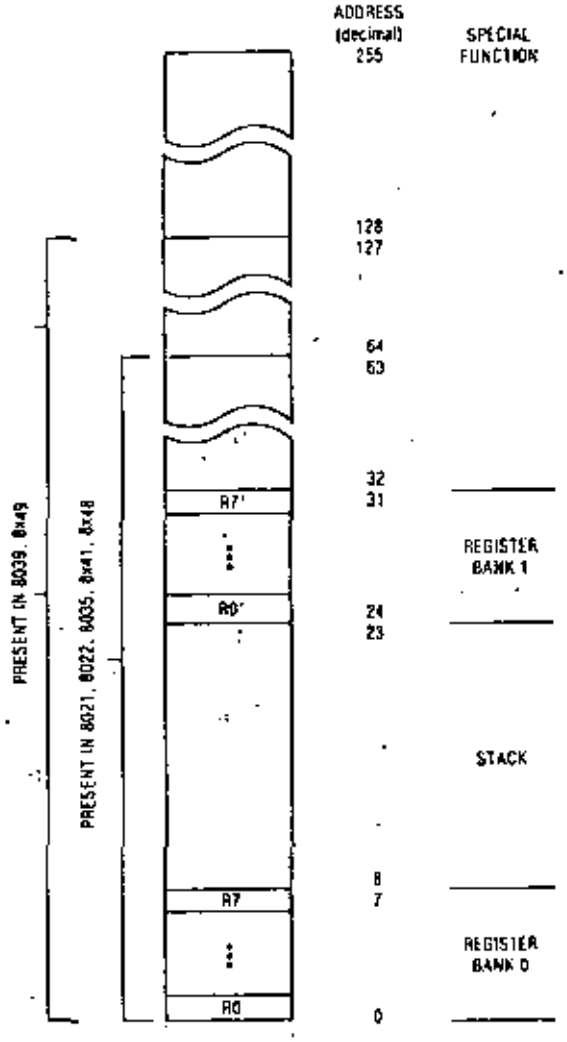


Figure 3. MCS-48 internal data memory.

INTERNAL REGISTER INDIRECT allows either R0 or R1 in the current register bank to be used as an 8-bit pointer to internal data memory. The addressed byte may be loaded with an immediate value, moved to or from the accumulator, combined with the accumulator, or incremented (but for some obscure reason not decremented, even though the necessary "hole" exists in the instruction set).

Locations 8-23 of the internal data memory are reserved for a return address stack (8 entries, 2 bytes per entry). These locations are written by interrupts and subroutine calls and read by interrupt and subroutine return instructions. The stack is too small, making it hard to write procedural code, which is important in larger programs (2K-4K bytes). The programmer must constantly worry about calling sequences and generally enable interrupts only at the top level of the program to avoid overflowing the stack.

There are no instructions to directly push or pop a byte. However, the stack can be rather inconveniently written or read by extracting the stack-pointer field from the PSW, building the appropriate address, and using INTERNAL REGISTER INDIRECT mode.

The architecture also supports up to 256 bytes of external data memory (which resides on a separate chip), accessed by EXTERNAL REGISTER INDIRECT mode. Either R0 or R1 in the current register bank may be used as an 8-bit pointer to external data memory; the addressed byte may be copied into the accumulator or written from the accumulator.

Since pointers are contained in 8-bit registers, the maximum amount of directly accessible data memory supported by the MCS-48 architecture is 256 bytes internal plus 256 bytes external. However, bank switching via I/O bits can be used to address any desired amount of additional external data memory.

The modes for reading operands from program store are rather limited. In IMMEDIATE mode an operand is contained in the byte following the instruction; immediate operands can either be loaded into or combined with the accumulator or be loaded into internal data memory with REGISTER or INTERNAL REGISTER INDIRECT modes.

In ACCUMULATOR INDIRECT mode the accumulator is used as an 8-bit pointer to an operand in either the current page or page 3 of program store; only one type of operation uses this mode, and it loads the accumulator with the specified operand.

A number of instructions specify some "special" operands implicitly, such as the program status word, I/O ports, timer/counter, carry bit, and two 1-bit flags, F0 and F1.

The MCS-48 addressing modes are simple, but they provide most of the facilities a program needs. Still, there are some deficiencies. The most serious problem is the way in which operands in program store are addressed. Since program store only in the current page and in page 3 can be read through a pointer, either lookup tables must all be located in page 3 or the code that reads each table must be in the same page as the table. This is inconvenient if more than one 256-byte translation table is needed. It also makes it difficult to do a ROM checksum self-test routine—a checksum subroutine would have to be placed in every page of program store (and since there is no indirect subroutine call, the main checksum program would have to contain a separate call instruction to each page's checksum routine).

For most programs, the method of indirectly addressing data memory through R0 and R1 is acceptable, but, for some data-structure manipulations, one wishes for one or two more registers that could be used as pointers.

The 256-byte limit on directly addressable internal data memory is too low. The 8049 already contains 128 bytes of RWM, and Intel should soon be able to provide the full 256 bytes of RWM on one chip. The architecture cannot make straightforward use of technology improvements for more RWM once this limit is reached.

Input/output and interrupts

Most MCS-48 microcomputers have three 8-bit I/O ports, as shown in Figure 1. Two of the ports (1 and 2)

are "quasi-bidirectional," an interfacing arrangement shown in Figure 4. This type of I/O port was first introduced in the Fairchild F8.[3] In this arrangement, each I/O pin is both an open-drain output and an input pin with a high-impedance pullup to the logic 1 level. When a pin is used for output, the corresponding input buffer is unused except, possibly, for checking the output value. When a pin is used for input, the corresponding output bit must be set to logic 1 so that the I/O device drives only the high-impedance pullup. This can be contrasted with a tristate I/O port, which provides both active pullup and active pulldown in output mode and high impedance in input mode. Electrically, tristate I/O is more desirable, but it requires extra control bits to set the I/O direction for each port or bit. Intel has improved the quasi-bidirectional design by briefly providing active rather than passive pullup whenever a 1 is written to the port, which speeds up 0-to-1 transitions.

What quasi-bidirectional I/O means to the programmer is that input data on the port is logically ANDed with the current output. Ports 1 and 2 are set to all 1's at system reset, and the programmer must leave bits intended for inputs set at output value 1 at all times. The third port (bus) has conventional tristate outputs and can be used for eight strobed inputs, for eight strobed outputs, or for adding external program or data memory.

Four operations on the ports are available:

read input value into accumulator (IN),
load output latch from accumulator (OUTL),
logical AND output latch with immediate mask (ANL), and
logical OR output latch with immediate mask (ORL).

The logical operations allow a program to set or clear any bit or group of bits in one instruction. However, since the mask is an immediate value in program store, the bits to be set or cleared must be known at assembly time. Otherwise, a copy of the output value must be kept in data memory, combined with the mask by logical operations on the accumulator, and loaded into the port. (In general, the quasi-bidirectional interface prevents simply reading the port to get the old value of the output latch.)

A novel "expander-port" arrangement allows four external 4-bit I/O ports to be added to an MCS-48 using a five-wire interface. Again, four operations on the ports are available:

read input value into accumulator,
load output latch from accumulator,
logical AND output latch with accumulator, and
logical OR output latch with accumulator.

Only the low-order four bits of the accumulator are used in these operations. For these ports, dynamic selection of mask bits is possible because the mask is in the accumulator. On the other hand, dynamic selection takes more overhead because the accumulator must be loaded with the mask (and then possibly restored to its old value).

Both the on chip and expander I/O port instructions contain the port number as an immediate value

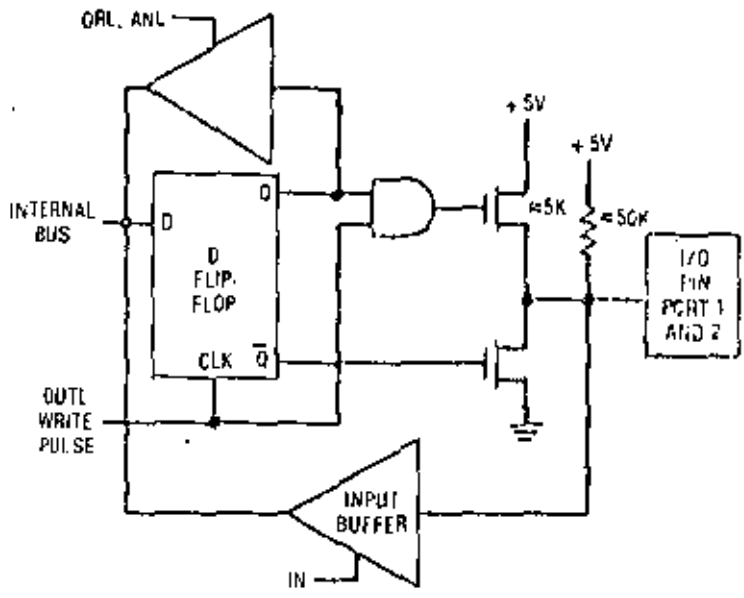


Figure 4. "Quasi-bidirectional" I/O port.

in the instruction; it is not possible to specify the port number dynamically in a register. This makes it impossible to write reusable I/O handlers for identical devices on different ports of the MCS-48 or of the same expander chip. The same problem exists in the MCS-48's older brother, the 8080. However, it is less serious in the MCS-48 for two reasons. First, the MCS-48 is intended for smaller applications less likely to employ many copies of the same I/O device. Second, the available I/O expansion modes do allow dynamic device selection when each device uses a separate I/O chip.

The processor architecture directly supports only 256 bytes of external data memory and four external 4-bit I/O ports. However, the amount of external data memory and I/O can be increased to any practical amount using on-chip I/O-port bits to implement program-controlled bank switching.

In addition to the I/O ports, an MCS-48 has three additional input pins that can be tested by conditional jump instructions. All are multipurpose pins—T0, which can be set up as a clock output under program control; T1, which can be used as the input to the on-chip timer/counter; and the external interrupt input.

The MCS-48 accepts interrupts from two sources—a level-sensitive input pin and an on-chip timer/counter. When an interrupt is serviced, the 12-bit PC and four status bits (carry, half carry, flag 0, register bank select) are pushed onto the internal stack. Depending on the source, a jump to either location 3 or location 7 is taken. The interrupt system is single-level; interrupt service routines cannot be interrupted. An interrupt return instruction restores the PC and status bits and allows further interrupts to be serviced.

At the time of this writing, the T1 interrupt input is generally useless for counting or timing asynchronous external events, because the current chip

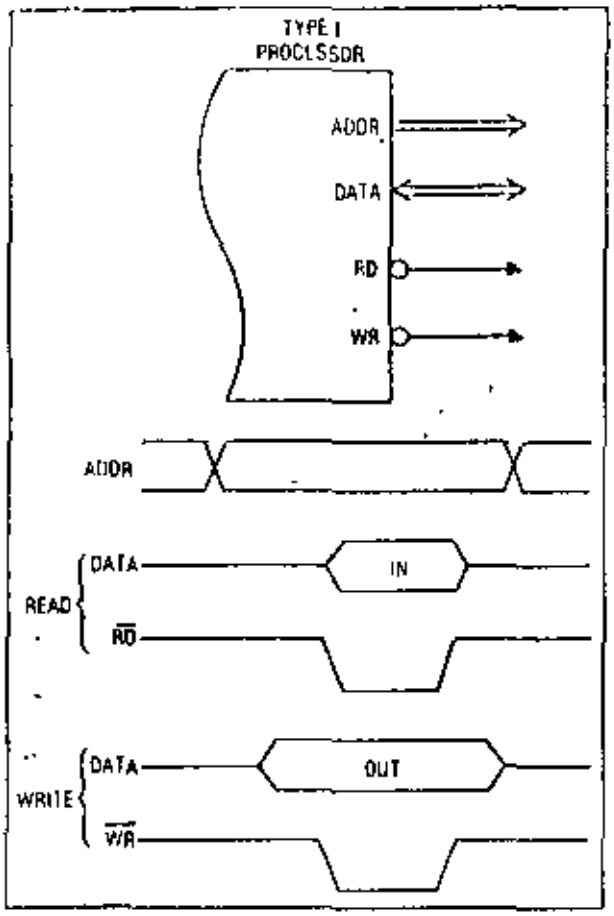


Figure 5. Type I bus control signals.

features a poorly designed synchronizer that sometimes misses input edges and hence skips counts. This is the second time in six months that I have seen an LSI chip whose designers were apparently unaware of problems in synchronizer design (the other was the Z80-SIO, which Zilog has since fixed). I would suggest that chip designers read some of the papers on the subject[4-6] and that academics warn their students of the increasing likelihood of synchronization problems in modern system design.

Ease of programming

Compared to some of the older 4-bit and 8-bit microprocessors, the MCS-48 is a nice machine to program, but it leaves much to be desired compared with an M6801, a Z8, or even an 8085. The single-accumulator architecture, the lack of index registers, and the absence of even a direct data memory addressing mode means that the programmer must constantly be moving things back and forth between the accumulator, the two "pointer" registers R0 and R1, and the rest of the data memory (and keeping track of them!). One may write macros to ease the burden somewhat, at the expense of more inefficient code in the cramped address space. For example, one can write a macro to simulate a direct data-memory-addressing mode:

```
LDA     MACRO MEMADDR
        MOV R0,#MEMADDR
        MOV A,@R0
        ENDM
```

A tale of two buses (or, different strobes for different 'phobes)

A microprocessor memory and I/O bus has many identifying characteristics—data and address word length, multiplexed or nonmultiplexed address and status, separate or memory-mapped I/O, and others.[3] It is interesting to look at two popular read/write clocking arrangements.

Let us call the first technique a Type I (or 1) interface, used by the Intel 8085, 8086, and MCS-48 families. As shown in Figure 5, there are two mutually exclusive control pulses, $\overline{RD}$ and $\overline{WR}$, that indicate a read or write operation.

We'll call the second technique Type Z (or 2), used by the Zilog Z8, Z80, Z8000, and also by the Motorola 6800 family. (Perhaps it should be Type M because the M6800 came first, but Z looks more like 2.) It is also used in principle by MCS-48 expander ports. As shown in Figure 6, there is a single control pulse, $\overline{REQ}$, and a level signal R/$\overline{W}$ that indicates which type of operation is to take place. The timing of R/$\overline{W}$ is similar to that of an address signal.

Figure 7 shows how to use a Type Z processor with a Type I peripheral chip. The decoding shown in the figure can be easily implemented with one-half of a TTL 74LS139 dual 2-to-4 decoder (this even leaves an extra control input for distinguishing between memory and I/O if desired). Assuming that processor and peripheral speeds are comparable, there should be no problem in satisfying the timing requirement of either the processor or the peripheral chip.

Figure 8 shows an attempt to use a Type I processor with a Type Z peripheral chip. The logical AND of $\overline{RD}$ and $\overline{WR}$ from the processor nicely produces $\overline{REQ}$ for the Type Z peripheral. $\overline{RD}$ has the correct logic value to serve as R/$\overline{W}$, but its timing is a problem. The Type Z peripheral expects R/$\overline{W}$ timing to be similar in character to an address signal, that is, it should be valid long before the $\overline{REQ}$ pulse appears. The only way we could ensure this would be to artificially delay $\overline{REQ}$ long enough for $\overline{RD}$ to satisfy the setup time of R/$\overline{W}$. Unfortunately, such a delay (unless highly asymmetric) would also delay the trailing edge of REQ until long after R/$\overline{W}$ ($\overline{RD}$) had gone away—again a problem.

The cleanest way to use a Type I processor with Type Z peripheral chips is to use an address line as R/$\overline{W}$. For example, the least significant bit of the I/O port address could be reserved as R/$\overline{W}$. Hardware decoding of actual port numbers would use the higher-order bits; then software would have to ensure that writes always used odd port addresses, and reads used even.

The conclusion is that it is simple to connect Type I peripherals to Type Z processors, but that the reverse can be difficult. Is this just the way it turned out or was there method to this madness?

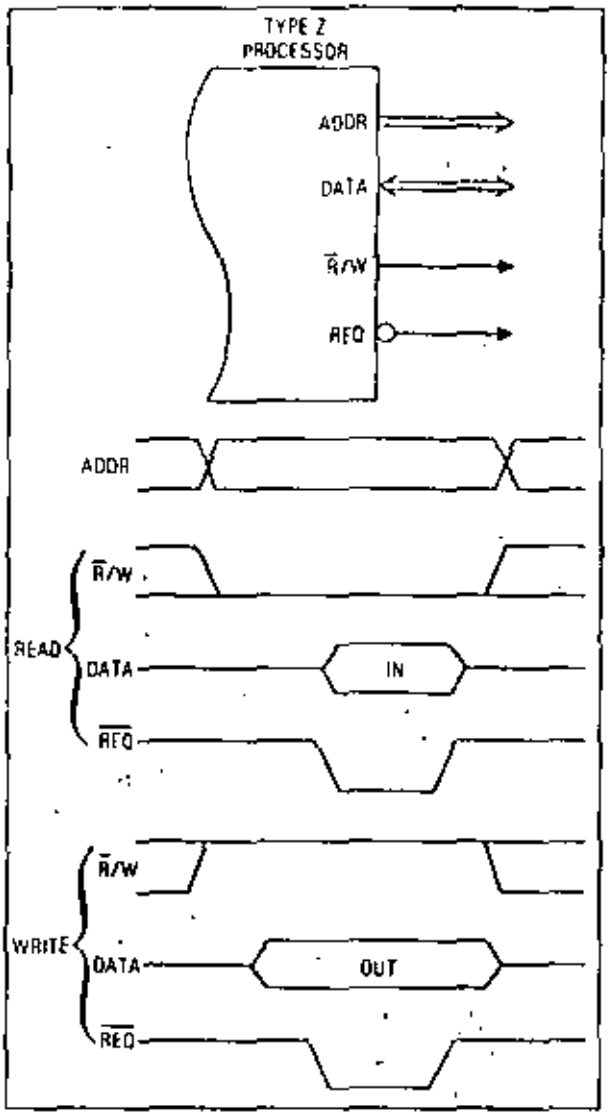


Figure 6. Type Z bus control signals.

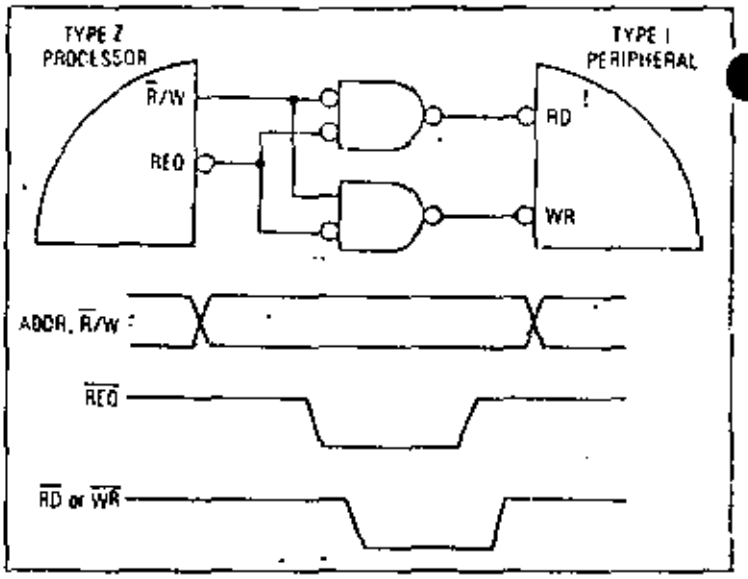


Figure 7. Acceptable Z-to-I interface.

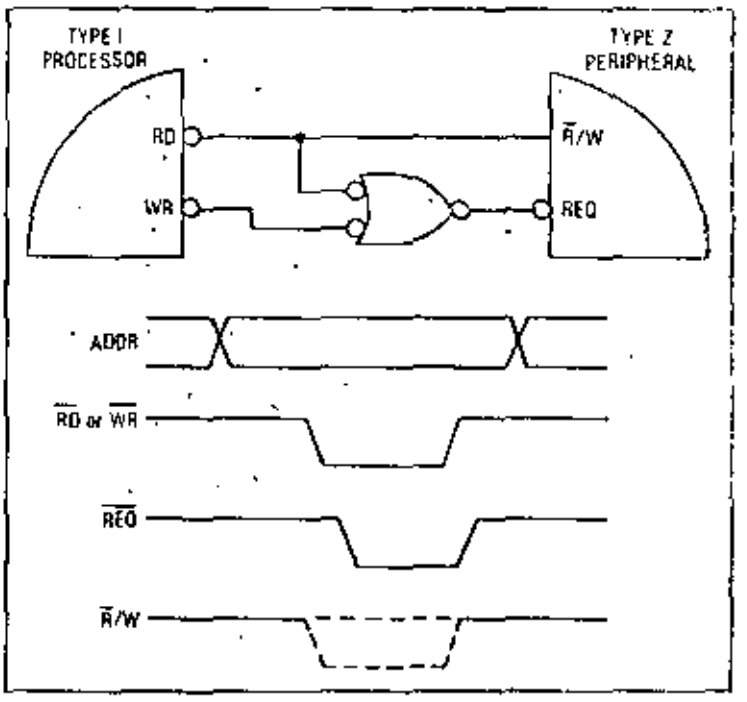


Figure 8. Unacceptable I-to-Z interface.

To use such macros, however, the programmer must give up some registers for use by the macros. In the above example, macros would use R0, leaving only R1 available to the program as a pointer variable. (Buffer copying and other routines requiring two or more pointers get to be a problem.)

The lack of symmetry in the instruction set also creates programming headaches. For example, why are there INC R0, DEC R0, and INC @R0 instructions, but not DEC @R0? Or, why can we conditionally jump on C, Z, T0, and T1 conditions true or false, but on F0, F1, TF, and accumulator bits only true? Except for accumulator bits false, the proper "holes" exist in the instruction set; in fact, it probably took more logic to turn the instructions off than to let them work. I have been told that these "unimportant" instructions leave room for future enhancements, but any worthwhile architectural enhancements would require changes more sweeping than a few special-purpose opcodes.

While nice programs can be written for the MCS-48, they take more effort than those written for a "general-purpose" architecture. Programming difficulty and expense also increase as we bump against the memory limits of the 8048. If we are writing one short (< 1K-byte) program and shipping 50,000 units with it, the programming expense is quite justifiable. If we are writing 10 different 2K- to 4K-byte programs, each of which will be shipped with 5000 units, we might be better off selecting a cleaner (albeit more expensive) machine.

Some electrical characteristics

The MCS-48 uses Intel's reliable *n*-channel silicon-gate MOS process. Several second sources for the

family have been announced—AMD, NEC, Signetics, Siemens, Intersil, RCA. Both Intersil and RCA have announced plans for CMOS versions of MCS-48 chips.

The MCS-48 chips use a single +5 volt supply and have logic input and output levels that are fully TTL compatible. As with all MOS microprocessors, the output drive is limited—typically four or five low-power Schottky (LSTTL) unit loads.

MCS-48 chips contain an oscillator to generate the processor clock (nominally 6 MHz) from an external crystal or RC circuit. It is also possible to connect an external clock directly to the oscillator input. The output of the oscillator feeds a divide-by-three counter whose output (nominally 2 MHz) controls the internal states of the processor. Since the divide-by-three counter cannot be synchronized externally, processor I/O cannot be referenced very well to the 6-MHz clock for tricky interfaces, nor can processors be run in lock-step from a common clock in a triplicated microcomputer (author's pet project).

The MCS-48 family satisfied the usual Intel strategy of being the first in the marketplace with an imperfect but useful product.

The MCS-48 chips have an active-low reset input pin connected to an internal high-impedance pullup resistor and Schmitt trigger. Thus, power-on reset can be accomplished by an external 1 mF capacitor. It is a little more difficult to add a logic-controlled reset (for example, by a watchdog timer), since open-collector or discrete transistor drive is required. And unless the driver circuit is sophisticated, a logic-commanded reset will disable the processor for a long time, due to the time constant of the power-on reset circuit—about 200 msec. In any case, reset destroys the state of the processor. It would be nice to have a nonmaskable interrupt (as in the 8085) that could be used for applications such as watchdog timers.

Development tools

Intel supports two major development tools for the MCS-48 family—a cross assembler and an in-circuit emulator, ICE, both of which run on an Intel MDS microcomputer development system. Unfortunately, Intel does not support any MCS-48 assemblers or simulators that run on a large computing system, a necessity for any large development project. However, they can be obtained from independent software houses and consultants such as Microtec.

Intel MDS software for the MCS-48 lacks the consistency one expects from a good set of software tools. For example, there are at least three very different syntaxes that an engineer or programmer might use to change the value of a memory location in the MDS, depending on whether the monitor, ICE, or PROM programmer is being used. The monitor has a nice syntax that allows us to open a location, change it, and continue to the next location with a small number of keystrokes. In ICE, to read and change four locations, we must type (machine type underlined):

```
* CBYTE 144 TO 147
0144H=01H 3AH 0FFH 59H
* CBYTE 144=11,27,FF,8A
```

To do the same thing in the PROM programmer, the programmer types:

```
* DISPLAY FROM 144 TO 147
0090 01 3A FF 59
* CHANGE 144=11,27,FF,8A
```

In either syntax, one wastes keystrokes, and it's easy to lose track of the address in a long string. This isn't too terrible until we discover that ICE has interpreted and printed addresses and data in hex, while the PROM programmer has assumed inputs in decimal and printed outputs in hex (except for input FF, which the PROM programmer rejects because it looks like an assembler label!).

Another constant annoyance is that the MDS accepts only the RUB character for deleting characters (echoing the deleted character); backspace is not supported. It would have been easy enough to support both erase characters, making both teleprinter and CRT users happy.

Conclusion

The MCS-48 microcomputer family was a reasonable contribution to the state of the art when it was introduced in 1976, in spite of its flaws. It achieved its design goals and satisfied the usual Intel strategy of being the first in the marketplace with an imperfect but useful product.

The MCS-48 is an acceptable choice for applications with initial estimated requirements of less than 1 K bytes of program store, 64 bytes of data memory, and only one or two different programs to be developed. Designers whose applications require more memory or different programs in different chips should seek a more general-purpose architecture, such as the Z8 or 6801.

I have spent much of this article complaining that the MCS-48 is not a general-purpose 8-bit microcomputer. This may seem unfair, since some people at Intel claim the MCS-48 was never intended to go much beyond the old 4-bit market. So why criticize it on that basis? First, to help designers who might otherwise be tempted to use it in a larger application (Intel sales engineers frankly recommend their three-chip 8085 system in such cases). Second, to help designers who have already selected the MCS-48 for a larger application. Third, because discussion of general system requirements should benefit future system designers and chip designers alike. Finally, because Intel does not now offer a clean architecture suitable for the more general-purpose, single-chip, expandable microcomputer market, and I think they should. ■

Acknowledgments

I appreciate the comments of my colleagues during the preparation of this article, especially those from Prem Jain, Paul Frantz, Ed Yarwood, and Dave Stearns. The comments of the reviewers were also very helpful. Thanks go to Dennis Allison for suggesting this article over a year ago, and to Bernard Peuto and Len Shustek for making me finish it. Of course, special thanks go to Intel for bringing us the MCS-48.

Some of this material was prepared while I was a lecturer at the Digital Systems Laboratory at Stanford University. Other material is adapted from my textbooks, *Microcomputers, Vol. 1: Architecture and Programming* and *Microcomputers, Vol. 2: System Hardware Design*, to be published by John Wiley & Sons in late 1979. All opinions expressed in this article are my own.

References

1. Intel Corporation, *MCS-48 Family of Single Chip Microcomputers User's Manual*, Santa Clara, Calif., July 1978.
2. L. J. Shustek, "Analysis and Performance of Computer Instruction Sets," PhD dissertation, Stanford University, Jan. 1978, available from University Microfilms, Ann Arbor, Mich.
3. J. F. Wakerly, "Microcomputer Input/Output Architecture," *Computer*, Vol. 11, No. 2, Feb. 1977, pp. 25-33.
4. T. J. Chaney, S. M. Ornstein, and W. M. Littlefield, "Beware the Synchronizer," presented at COMPCON-72, IEEE Comput. Soc. Conf., San Francisco, Calif., Sept. 12-14, 1972.
5. T. J. Chaney and C. E. Molnar, "Anomalous Behavior of Synchronizer and Arbiter Circuits," *IEEE-TC* (Corresp.), Vol. C-22, No. 4, Apr. 1973, pp. 421-422.
6. M. Pechoucek, "Anomalous Response Times of Input Synchronizers," *IEEE-TC*, Vol. C-25, No. 2, Feb. 1976, pp. 133-139.

MICROSYSTEMS

The first implementation of a new microprocessor architecture promises to narrow the gap between the power of very small and very large computers.

A Microprocessor Architecture for a Changing World: The Motorola 68000

Edward Stritter
Tom Gunter
Motorola Semiconductor

Microprocessor technology is entering a new and especially challenging era. While technology constraints have not completely disappeared, we are nearly to the point where the limiting factor in microprocessor design is not how much function can be included, but how imaginative and creative the designer can be.[1] As a result, several companies have introduced new-generation microprocessors. We describe how one of them, the Motorola 6800, responds to these unique conditions.

Motivations for a new microprocessor architecture

Previous generations of microprocessors were limited by the available technology. Brooks, in an overview article,[2] discusses how the technology constraints and the perceived microprocessor market motivated early microprocessor architecture. Microprocessors were limited in number of registers, data-path width, and instruction-set power primarily because technology could not support more features on a single chip. Other limitations of microprocessors, such as having too small an address space[3] and awkwardness of address computation,[4] may be attributed as much to prevailing perceptions of the potential market as to technology constraints.[5] Whatever the former sources of restraint, however, we are now in a period of technical innovation and spirited competition.

Technological advances. The basic microprocessor technology, MOS, has been steadily advanced in the last few years. The most noticeable improvement has been circuit density (Figure 1), which translates directly into the amount of capability that can be put on a single-chip microprocessor. Whereas earlier microprocessors contained from 5000 to 10,000 transistors per chip, current processors have from 25,000 to 70,000 transistors, which is less than an order of magnitude away from the number in many of the largest maxi-computers. Circuit density is not the only technology advance that has been made: corresponding improvements have been achieved in circuit speed and power dissipation.

Advances in technology have been more evolutionary than revolutionary. The major advance, increased circuit density, is the result of gradual improvements in processing techniques that permit smaller circuit dimensions. Density improvements are expected to continue, since they depend not on overcoming fundamental limitations but only on further evolutionary improvement of existing processes. New microprocessor architectures must be devised to take advantage of this future advancement.

Market demands. The demand for microprocessors in applications not foreseen just a few years ago is providing new opportunities for microprocessor manufacturers. Just as the original microprocessor designers could not predict the many uses that would be found for their devices, today's designers cannot hope to envision more than a few of the eventual applications of new microprocessors. The implication for the designer is that new designs must be flexible and general if they are to be useful in a large number of potential applications.

High software costs. The problem of software costs is even worse in microprocessor applications than it is with computers generally. Decreasing memory costs,

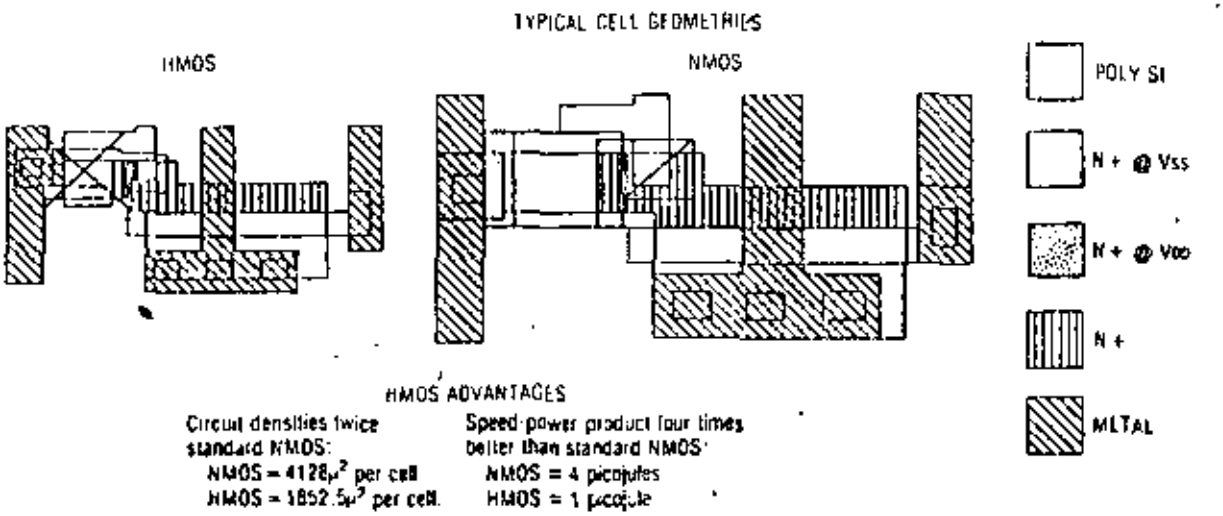


Figure 1. Comparisons of HMOS and NMOS technologies. The HMOS technology used for the MC68000 results in significant improvements to circuit densities and speed-power products.

increasing processor functionality, and more complex applications are combining to increase the size and complexity of microprocessor programs. Software costs of $100,000 or more are clearly incompatible with hardware costs of hundreds of dollars. This cost disparity may be unimportant in large-volume applications, where software costs can be amortized over thousands of hardware units, but it often precludes the use of microprocessors in applications characterized by complex programs but low volume. To help reduce the high cost of software, microprocessor designers must make a strong commitment to supporting high-level languages and disciplined programming practices.

High design costs. The cost of designing and implementing a new device with tens of thousands of transistors is high. Computer design aids are indispensible, but they are also expensive. Designers must attack this design-cost problem in several ways. First, straightforward designs, using regular structures, are easier to implement, test, and correct, and are therefore less expensive than exotic designs. Second, each new architecture must be planned to last for as long as possible and must be easy to expand in the future. Manufacturers can no longer afford to produce new architectures every few years. Experience with trying to extend and improve the original 8-bit microprocessor architectures demonstrates the need for planned expansion. Designers must be careful to include as few limitations to future expansion as possible. The most common mistakes in the past have been limiting address size and not providing unused operation codes for future new instructions.

Design goals for the 68000. Motorola's 68000 microprocessor architecture has been designed to meet the requirements outlined above. (The MC68000's characteristics are summarized in Table 1.)

Architectural family. The 68000 design specifies a computer architecture of which a number of different versions or "implementations" will be produced.[6] The first version, the MC68000, implements only the subset of the complete 68000 architecture allowed by current technology constraints.

Flexibility and usefulness. The 68000 design ensures that the processor is easy to program. As much as possible, there are no unnatural limitations, artifacts, special cases, or other awkward features in the architecture.

Marketability. The 68000 is a general-purpose architecture, reflecting the increasing market acceptance of general-purpose microprocessors for diverse applications.

Expandability. The 68000 design specifies several features, such as floating-point and string operations, that are not implemented in the first version but have been specified now to guarantee future consistency. In addition, unused space has been left in the architecture to accommodate new features that future advances in technology will make possible.

Support of high-level languages. The 68000 architecture contains features for implementing high-level languages, and Motorola is commited to supplying software support for program development in well-known high-level languages.

Table 1.
Motorola MC68000 characteristics.

INTEGER SIZES	8, 16, and 32 bits
ADDRESSING CAPABILITY	16,777,216 bytes
INPUT/OUTPUT	memory-mapped
TECHNOLOGY	HMOS
INTERNAL CYCLE	250 nsec
MEMORY ACCESS	500 nsec
RELATIVE PERFORMANCE	10 to 75 times 8800
PINS	64
POWER	+5V

Resources. The 68000 design provides an address space of 2^{32} bytes (limited to 2^{24} bytes in the initial implementation). Memory is byte addressable, with individual-bit addressing provided for bit-manipulation instructions. Memory may be accessed in units of 1, 8, 16, or 32 bits. CPU resources include sixteen 32-bit registers, a 32-bit program counter (24 bits in the initial implementation), and a 16-bit status register.

The registers (Figure 2) are divided into two classes. The eight data registers are used primarily for data manipulation; they may be operand sources or destinations for all operations but are used in addressing only as index registers. The eight remaining (address) registers are used primarily for addressing. The stack pointer is one of the address registers. The program counter and status word are separate registers.

Addressing. Memory is logically addressed in 8-bit bytes, 16-bit words, or 32-bit long words. The current implementation requires that word and long-word data be word aligned. Bits are individually addressable in the bit-manipulation instructions.

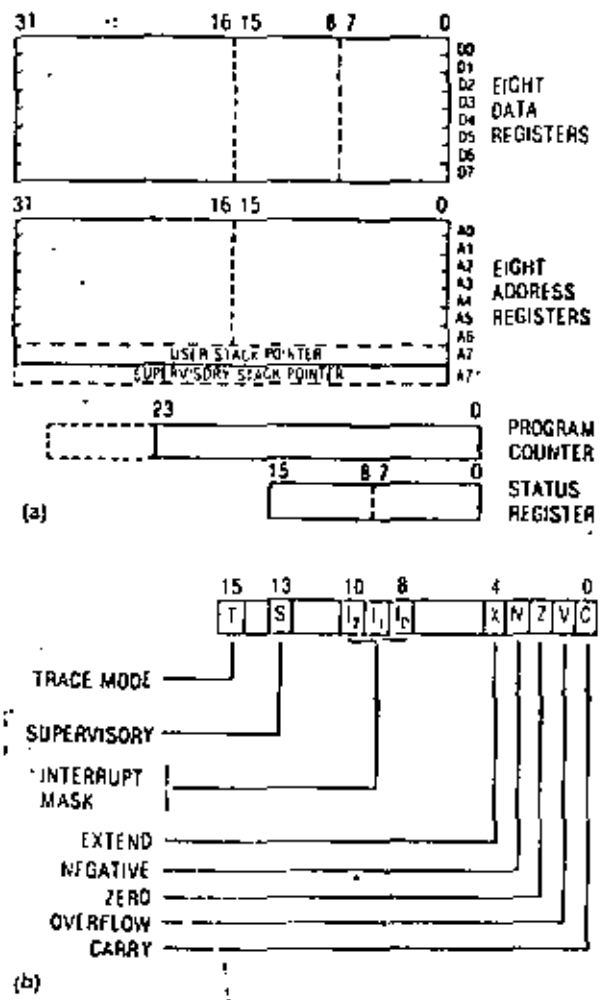


Figure 2. MC68000 programming model (a) and internal structure of status register (b).

Table 2.
MC68000 addressing modes.

REGISTER DIRECT ADDRESSING:	
Data register direct	EA = Dn
address register direct	EA = An
status register direct	EA = SR
REGISTER DEFERRED ADDRESSING:	
register deferred	EA = (An)
register deferred post-increment	EA = (An); An <— An + N
register deferred pre-decrement	An <— An – N; EA = (An)
base relative	EA = (An) + d16
indexed	EA = (An) + (Xn) + d8
PROGRAM COUNTER RELATIVE:	
relative with offset	EA = (PC) + d16
relative indexed	EA = (PC) + (Xn) + d8
short PC relative branch	EA = (PC) + d8
long PC relative branch	EA = (PC) + d16
ABSOLUTE ADDRESSING:	
absolute short	EA = (next instruction word)
absolute long	EA = (next two instruction words)
IMMEDIATE DATA ADDRESSING:	
immediate	DATA = next instruction word(s)
quick immediate	DATA = sub-field of instruction (4 bits)

DEFINITIONS:
- EA = effective address
- An = address register
- Dn = data register
- Xn = address or data register used as index register
- SR = status register
- PC = program counter
- d8 = 8-bit displacement
- d16 = 16-bit displacement
- N = 1 for byte, 2 for word, and 4 for long word operands
- ( ) = contents of
- <— = replaces

The architecture specifies an optimal memory-management scheme that implements and enforces variable-length segmentation of the address space with access rights specifiable for individual segments. The processor can be used with or without memory management.

Address calculations (Table 2) are specified by 6-bit fields of the instruction. The addressing specification is orthogonal to the operation specification of the instruction; that is, any addressing mode can be used in any instruction that uses addressing.

Addresses are 32-bit quantities (24 bits in the current implementation). The architecture efficiently supports small systems (those with fewer than 2^{16} addressable bytes) by allowing 16-bit address quantities to be specified, moved, or calculated in almost every addressing situation. For example, an absolute address carried in an instruction can use 16 or 32 bits, or an index calculation can use 16 bits (sign extended to 24 bits) or 32 bits of a register as input. This feature allows the architecture to support very large addresses without penalizing the efficiency of programs that require only small addresses. The address size (16 or 32 bits) is individually specified for each use, so that large and small addresses can be intermixed arbitrarily in a program.

A variety of addressing modes are available:

Register direct. The data or address register contains the operand.

Address register deferred. The operand address is in the specified address register.

Address register deferred post-increment. The operand address is in the specified address register. After the operand is accessed, the address in the register is incremented by the operand size (1, 2, or 4).

Address register deferred pre-decrement. The operand address is in the specified address register. Before the operand is accessed, the address register is decremented by the operand size.

Base relative. The operand address is the contents of the specified address register plus a 16-bit signed displacement in the instruction.

Program counter relative. The operand address is the current program counter value plus a 16-bit signed displacement in the instruction.

Indexed. The operand address is the contents of the specified address register plus the contents of an additional (data or address) register specified plus an 8-bit signed displacement in the instruction.

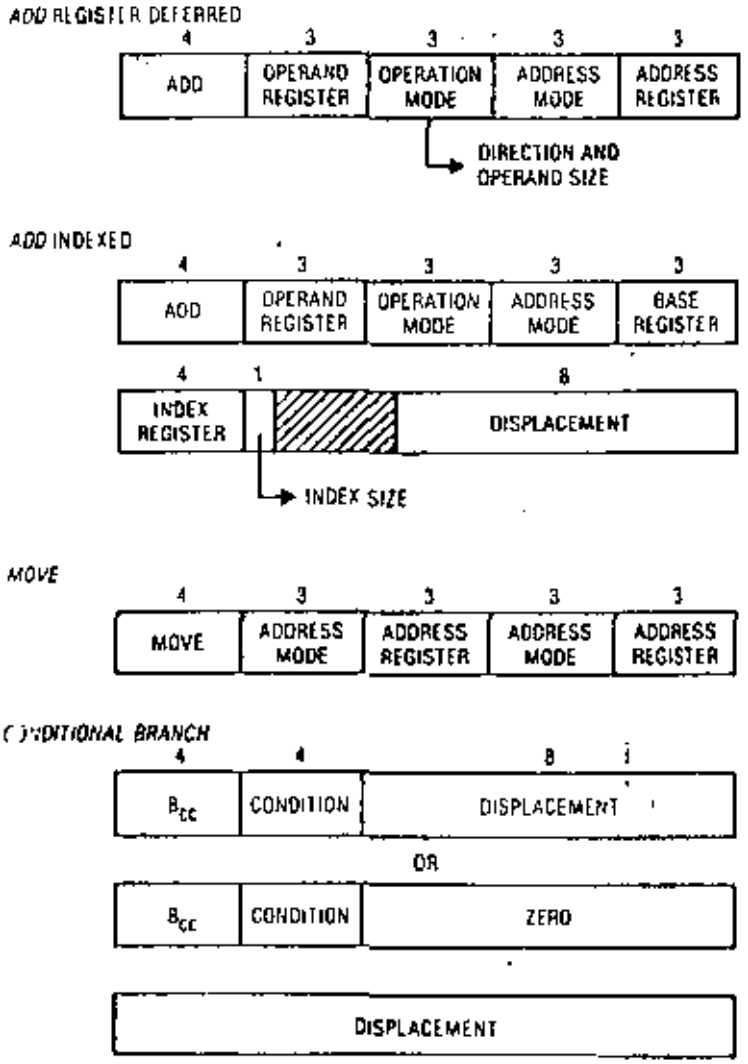


Figure 3. Typical MC68000 instruction formats.

Program counter indexed. The operand address is the current value of the program counter, plus the contents of the specified data or address register, plus an 8-bit signed displacement in the instruction.

Absolute. The operand address is in the instruction.

Immediate. The operand is in the instruction.

Bit addressing. A complete set of bit-manipulation instructions (SET, CLEAR, CHANGE, and TEST) is provided. For these instructions, an individual memory word is addressed using one of the above addressing modes. The individual bit to be manipulated is addressed by its bit number in that word. The bit specification is contained in the instruction or previously calculated in a data register. This mechanism allows bits to be addressed simply, without requiring the use of logical instructions and masks. For registers, all 32 bits are individually addressable.

In all cases, the addresses specified by the program can span the entire address space. No arbitrary segment sizes are imposed, and no separate segment numbers need be manipulated.

Address, like integer, is a fully supported data type. A complete set of address-manipulation operations (MOVE, COMPARE, INCREMENT, DECREMENT, ADD TO, SUBTRACT FROM) is implemented on the address registers. In addition, the LOAD EFFECTIVE ADDRESS instruction performs an arbitrary calculation and puts the result into a specified address register. This provides the programmer, in a single instruction, with the ability to precalculate addresses using any of the processor's addressing modes.

Because there are eight address registers, fewer memory accesses are required for loading and storing temporary address values, and addresses rarely need to be recalculated in different parts of the program. These features minimize the program time spent manipulating addresses, a common bottleneck in existing microprocessors. They also establish a degree of address-size independence;[7] the address-specification fields in instructions are most often only 6 bits, regardless of the fact that a large (32-bit) address is actually being specified.

Data manipulation. The 68000 supports a number of data types and supplies a complete set of operations for each type (Table 3 and Figure 3). In general, the addressing mode is independent of the data type. Also, in cases where it makes sense (integers, logicals, and addresses), the size of the operand may be specified independently of the operation. Operand sources may be either registers or addressed memory locations. The result may be stored either in the register or in the specified memory location. This class of "register-to-memory" operations reduces the number of register stores required to save results. Most operations can be specified to work memory-to-register, register-to-register, register-to-memory, immediate-to-register, or immediate-to-memory. The move instruction is more flexible, being a full two-address instruction. It can specify memory-to-memory move operations as well as the options listed above.

The 68000 data types and the operations that support them are:

Integer. The operations are ADD, SUBTRACT, MULTIPLY, DIVIDE, NEGATE, COMPARE, and ARITHMETIC SHIFT. Integers may be 1, 2, or 4 bytes. Shifts are multiple-bit shifts, either left or right, with shift count specified in the instruction or previously calculated in a data register, and indicate overflow as appropriate.

Multiprecision integer. ADD WITH EXTEND, SUBTRACT WITH EXTEND, NEGATE WITH EXTEND, UNSIGNED MULTIPLY, and UNSIGNED DIVIDE are the primitives supplied for easily implementing multiprecision integer arithmetic. Operands may be 1, 2, or 4 bytes, except for multiply and divide, which operate only on 2-byte quantities.

Logical. The operations are AND, OR, EXCLUSIVE OR, COMPLEMENT, COMPARE, SHIFT, and ROTATE (which allow multiple-position shifts and rotates, left or right, with or without extend bit). Logicals may be 1, 2, or 4 bytes.

Boolean. AND, OR, EXCLUSIVE OR, COMPLEMENT, IMPLICATION, and SET ACCORDING TO CONDITION CODES are provided. (SET ACCORDING TO CONDITION CODES is used to retrieve the logical value of any of the conditional tests that are available to the CONDITIONAL BRANCH instruction.) Boolean data are one-byte quantities.

Bit. The operations are SET, CLEAR, CHANGE, and TEST. Bits are individually addressable.

Decimal. ADD, SUBTRACT, NEGATE, and COMPARE are decimal operations. The decimal (BCD) instructions work on operands in memory (memory-to-memory) two digits (one byte) at a time. Combined with a looping instruction, the decimal instructions implement variable-length memory-to-memory decimal operations.

Character. Character instructions, MOVE and COMPARE, work on operands in memory (memory-to-memory).

Address. Address operations include INCREMENT (by 1, 2, or 4), DECREMENT (by 1, 2, or 4), ADD INTEGER, SUBTRACT INTEGER, COMPARE, and LOAD EFFECTIVE ADDRESS.

Real. Floating-point ADD, SUBTRACT, MULTIPLY, and DIVIDE are specified but not implemented in the first version.

String. STRING MOVE, STRING SEARCH, and TRANSLATE are specified but not implemented in the first version.

Program control. Program-control instructions include CONDITIONAL BRANCH (program counter relative), JUMP, JUMP TO SUBROUTINE, RETURN FROM SUBROUTINE, and RETURN FROM INTERRUPT, all of which are traditional instructions. Sixteen separate operating-system calls are specifiable with the TRAP instruction. Conditional traps, looping, and subroutine control are discussed below. The STOP instruction halts the processor, the RESET instruction reinitializes the system environment, and the MOVE instruction can manipulate the processor status word.

Privilege states. The 68000 processor can operate in user or supervisor state. In supervisor state, the entire instruction set is available. Indication of the current state is given to the external world so that, for instance, address translation can be inhibited when the processor is in supervisor state. In user state, certain instructions, such as STOP, RESET, and those that modify the status word, are not allowed; they cause a

Table 3.
MC6800 instruction set.

MNEMONIC	DESCRIPTION
ABCD	Add decimal with extend
ADD	Add
ADDX	Add with extend
AND	Logical and
ASL	Arithmetic shift left
ASR	Arithmetic shift right
BCC	Branch conditionally
BCHG	Bit test and change
BCLR	Bit test and clear
BRA	Branch always
BSET	Bit test and set
BSR	Branch to subroutine
BTST	Bit test
CHK	Check register against bounds
CLR	Clear operand
CMP	Arithmetic compare
DCNT	Decrement and branch non-zero
DIVS	Signed divide
DIVU	Unsigned divide
EOR	Exclusive or
EXG	Exchange registers
EXT	Signed extend
JMP	Jump
JSR	Jump to subroutine
LDM	Load multiple registers
LDQ	Load register quick
LEA	Load effective address
LINK	Link stack
LSL	Logical shift left
LSR	Logical shift right
MOVE	Move
MULS	Signed multiply
MULU	Unsigned multiply
NBCD	Negate decimal with extend
NEG	Two's complement
NEGX	Two's complement with extend
NOP	No operation
NOT	One's complement
OR	Logical or
PEA	Push effective address
RESET	Reset external devices
ROTL	Rotate left without extend
ROTR	Rotate right without extend
ROTXL	Rotate left with extend
ROTXR	Rotate right with extend
RTR	Return and restore
RTS	Return from subroutine
SBCD	Subtract decimal from extend
SCC	Set conditionally
STM	Store multiple registers
STOP	Stop
SUB	Subtract
SUBX	Subtract with extend
SWAP	Swap data register halves
TAS	Test and set operand
TRAP	Trap
TRAPV	Trap on overflow
TST	Test
UNLK	Unlink stack

SAMPLE PROGRAM:

```
PROGRAM EXAMPLE;
VAR PARAM1, PARAM2: INTEGER;
PROCEDURE PROC (X: INTEGER; VAR Y: INTEGER);
  VAR A, B: INTEGER;
  BEGIN
  <procedure body>
  END;
BEGIN
  PROC (PARAM1, PARAM2)
END.
```

PROGRAM BODY:

```
MOVE     PARAM1 TO -SP@        "push first parameter"
PEA      PARAM2                "push address of 2nd parameter"
JSR      PROC                  "call the procedure"
ADD      #6 TO SP              "pop parameters from the stack"
```

PROCEDURE BODY:

```
LINK     FP, 4                 "link and allocate three local
                                variables"
MOVEM    <registerlist> TO -SP@ "push some register contents"
<procedure body>
MOVEM    <registerlist> FROM SP@+ "restore registers"
UNLK     FP                    "restore stack"
RETURN                         "return to calling procedure"
```

Figure 4. Sample Pascal program and equivalent 68000 code.

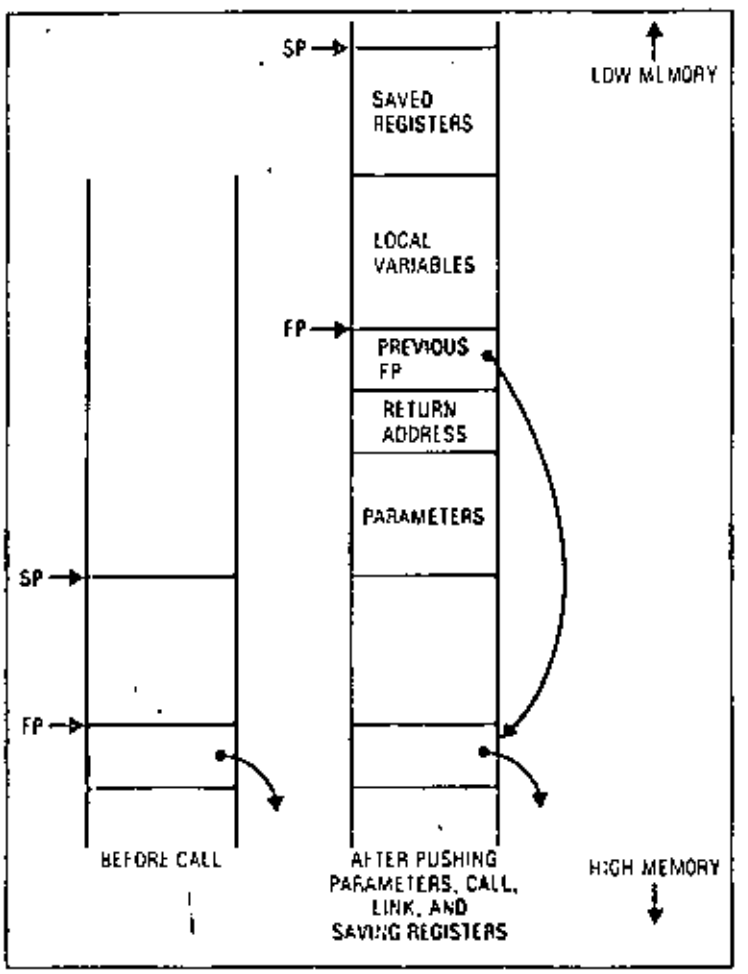


Figure 5. Stack activity on procedure call.

trap to supervisor state (by stacking the current program and status word and loading a new context from a preassigned trap vector). Illegal instructions, unimplemented instructions, interrupts, and traps (operating as system calls) all cause the processor to trap and switch to supervisor state. To ensure proper operation when returning to supervisor state, regardless of user-state activity, there are two stack-pointer registers—one active in user state, one in supervisor state. The user stack-pointer contents are available, by special instructions, to the supervisor-state program.

The 68000's user/system state distinction will allow a small operating-system kernel to provide fully protected virtual address spaces to any number of independent tasks or users.

Trapping on illegal and unimplemented instructions allows the operating system to provide a more functional virtual machine to user-state tasks. For instance, the operating system can transparently provide software implementation of any currently unimplemented instructions (such as floating-point or string manipulation) executed by a user-state task.

High-level-language support. A recent paper by Allison[4] suggests ways in which microprocessor architecture should be designed to support high-level languages. This method, followed by the 68000 designers, is to "examine . . . the runtime representation required for the class of languages to be implemented" and to "provide adequate instructions . . . to support the required runtime representation" and transformations on that representation "without extensive in-line computation."

The 68000 design supports high-level languages, at both compilation time and execution time, with a clean, consistent instruction set; with hardware implementation of commonly used functions (multiply, divide, and address calculation); and with a set of special-purpose instructions designed to manipulate the runtime environment of a high-level-language program. The language constructs aided by these special-purpose instructions include array accessing, limited-precision arithmetic, looping, Boolean-expression evaluation, and procedure calls.

Array accessing. The CHK instruction compares a previously calculated array index (in a data register) against zero and a limit value addressed by the instruction. A trap occurs if the index is out of bounds for that array. This replaces a common sequence of instructions (at least four) with a single instruction.

Limited-precision arithmetic. The TRAPV instruction causes a trap if the preceding operation resulted in overflow. This allows efficient overflow testing to encourage proper checking of arithmetic results.

Looping. A restricted form of the FOR-loop construct is implemented in a single instruction that decrements a count and branches backward if the result is nonzero.

Boolean-expression evaluation. The Scc instructions assign a true or false value to a Boolean variable on the same conditions that are us-

ed by the CONDITIONAL BRANCH instructions. These instructions help implement Boolean-expression evaluation by avoiding extra conditional branches, especially in the case (as with Pascal) where "short-circuited" evaluation may be undesirable because of possible side effects.

Procedure calls. The 68000 uses a stack—pointed to by one of the address registers, called the stack pointer—to build the nested environments of called procedures. Three instructions (plus an additional one for each parameter) implement a high-level-language procedure call (Figure 4). The entire call mechanism uses only the stack and is completely reentrant (Figure 5). These instructions are described in more detail below.

Push parameter values or addresses onto the stack. The MOVE instruction pushes a value onto the stack, and the PUSH EFFECTIVE ADDRESS (see LOAD EFFECTIVE ADDRESS explained earlier) pushes the result of an arbitrary address calculation onto the stack for call by reference.

Call procedure. The JUMP TO SUBROUTINE instruction pushes the return address on the stack and jumps to the procedure entry point.

Establish new local environment. The LINK instruction does all of the following: saves the old contents of the frame pointer (an arbitrary address register) on the stack, points the frame pointer to the new top of stack, and subtracts the number of bytes of local storage required by the procedure from the stack pointer. This establishes local storage for the called procedure and a frame pointer (address register) for index addressing of local variables and parameters.

Save an arbitrary subset of the registers on the stack. The MOVE MULTIPLE REGISTERS instruction saves an arbitrary subset of the registers on the stack (or anywhere in memory) in a single instruction. The registers to be saved are indicated by setting the corresponding bits in a 16-bit field of the instruction.

A set of at most four instructions reverses the process for procedure return:

Reload saved registers. The MOVE MULTIPLE REGISTERS instruction is used here also.

Reestablish previous environment. The UNLINK instruction undoes the work of the LINK instruction.

Return from procedure. The RETURN instruction pops the return address from the stack and returns to the calling procedure.

Pop parameters from the stack. The ADD IMMEDIATE instruction used on the stack pointer pops any number of values off the stack.

The 68000 system architecture

A computer architecture specifies interactions between the processor and its environment by defining such things as interrupt structure, memory segmentation, bus interfaces, and input/output structure. The 68000 system architecture is designed to be as flexible as possible. For instance, I/O device registers are addressed as memory locations (memory-mapped I/O), as on other Motorola microprocessors. Memory-mapped I/O gives the programmer the flexibility and power of the entire instruction set for manipulating device control and data registers. Since no additional instructions are required for I/O, the processor is simpler, and the instruction set is easier to remember. The I/O space is protected by the same memory-management facilities that are used to protect critical areas of memory.

The 68000 bus structure is also designed for simplicity, speed, and flexibility. The address and data lines are separate; no multiplexing is needed. This avoids the need for any separate devices for demultiplexing, ensuring maximum performance for systems in which speed is important. The bus is asynchronous; transfers on the bus are controlled by accompanying handshake signals, so that no assumptions need be made about timing or system synchrony. The use of handshake signals allows devices and memories with large variations in response time to be used on the same processor bus. The processor waits an arbitrary amount of time until the accessed device or memory signals that the transfer is occurring.

A simple bus request/grant protocol is implemented on-chip so that processors and direct-memory-access devices can cooperatively share the system bus with no extra arbitration logic. Also, the chip has a bus-fault input pin that causes instruction execution to be terminated at any point and a trap to be taken if an illegal or faulty memory access is made. This facilitates memory protection.

The 68000 interrupt structure is like that of most minicomputers. Eight priority levels are implemented. Interrupts are vectored so that software has full control over the placement and execution of interrupt-handling routines. The current priority level of the processor is kept in its status word. Interrupts at or below the current priority are inhibited. Interrupts at higher levels may occur, so interrupt handling may be nested. When an enabled interrupt occurs, the processor sends an acknowledge signal. The interrupting device responds with a vector number. The vector number is used by the processor

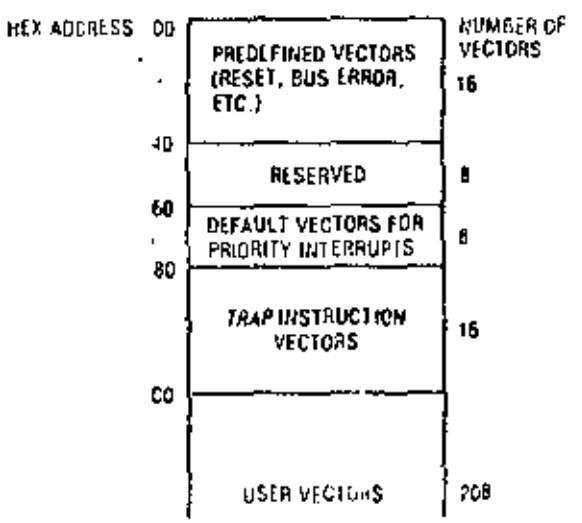
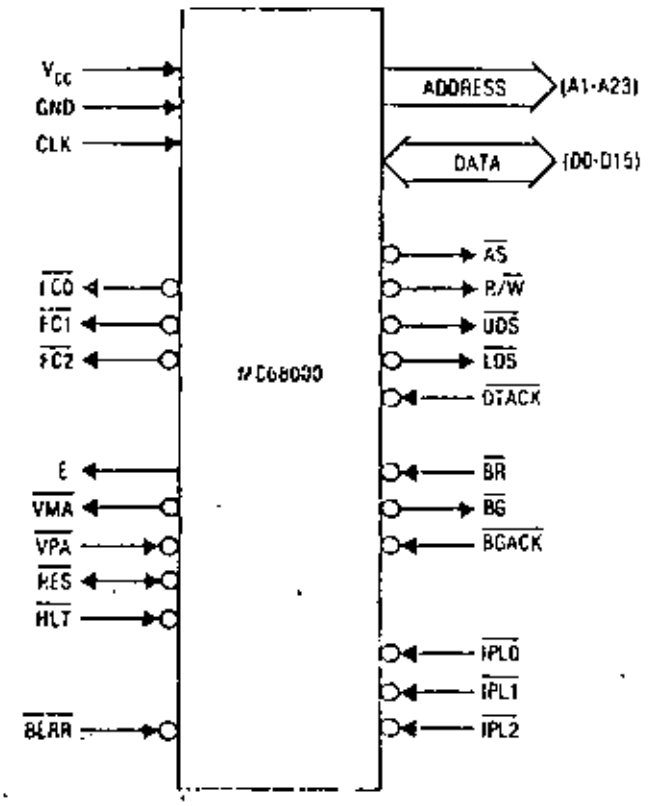


Figure 6. Trap and interrupt vector allocation.



A1-A23	Address leads	23 bit address bus; capable of addressing 16,777,216 bytes in conjunction with UDS and LDS.
D0-D15	Data leads	16 bit data bus, transfers 8 or 16 bits of information
AS	Address strobe	Indicates valid address and provides a bus lock for indivisible operations.
R/W	Read/write	Defines bus operation as read or write and controls external bus buffers.
UDS, LDS	Data strobes	Identifies the byte(s) to be operated on according to R/W and AS.
DTACK	Data transfer acknowledge	Allows the bus cycle to synchronize with slow devices or memories
BR	Bus request	Input to the processor from a device requesting the bus.
BG	Bus grant	Output from the processor granting bus arbitration.
BGACK	Bus grant acknowledge	Confirmation signal from BG indicating a valid selection from the arbitration process
IPL0, IPL1, IPL2	Interrupt priority level	Provides the priority level of the interrupting function to the processor
FC0, FC1, FC2	Function code	Provides external devices with information about the current bus cycle and processor state
CLK	Clock	Master TTL input clock to the processor.
RES	Reset	Provides reset (initialization) signal to the processor and peripheral devices
HLT	Halt	Stops the processor and allows single stepping.
BERR	Bus error	Provides termination of a bus cycle if no response or an invalid response is received
E	Enable	Enable clock for 6800 systems
VPA	Valid peripheral address	Identifies addressed area as a 6800 compatible area
VMA	Valid memory address	Indicates to 6800 family devices that a valid address is on the bus.
V_{CC}	+5 volt (2 pins)	—
GND	Ground (2 pins)	—

to index into a table of interrupt vectors in low memory to find the appropriate entry point to the interrupt handler; there are 256 such vectors (Figure 6). Individual devices on the same priority level can be distinguished by different vector numbers, so no device polling is required. Software traps and exception conditions in the processor also transfer through the vector table; in these cases the vector numbers are assigned by the processor. The vector table is in main memory and therefore can be manipulated by the operating system as necessary. The processor implements a set of default vectors (one for each priority level) so that existing peripheral devices, not equipped to respond with vector numbers, can be used.

68000 systems can be configured with a processor directly connected to memory; the addresses generated by the program are then for the physical memory. This will suffice for many applications. More complex applications, especially those with multiple tasks or even multiple users, will require more sophisticated memory management. A separate single-chip device will be available to provide memory segmentation, address translation, and memory protection.

68000 design and implementation

The single-chip MC68000 microprocessor (Figure 7) is a partial implementation of the 68000 architecture. It implements as large a subset of the complete architecture as current technologies will allow. The relevant technological constraints are limitations on the number of pins and on circuit density. Addresses are limited to 24 bits by present-day packaging technology, which restricts the number of pins per package to 64. Similarly the data path to memory is only 16 bits wide. This is not an architectural limitation, but it does require that two memory accesses be made for each 32-bit datum.

Circuit density limits the number of instructions that can be implemented. One-eighth of the operation-code map is currently unimplemented. Some of this space is allocated in the architecture—for example, for floating-point and string operations. Some of the free space is currently unspecified and will be allocated for future architectural enhancement. All unimplemented instructions cause traps, so that software emulation is possible.

Future implementations of the architecture may expand upwards or shrink downwards in performance and functional capability. Technological advances will soon allow the full architecture to be implemented. As circuit densities improve further, new versions will be faster and smaller (and thus less expensive) and will consume less power. Increased circuit density will also allow the inclusion of on-chip memory and sophisticated speed-up techniques.

Today's state of the art in MOS LSI technology permits approximately one transistor per square mil

Figure 7. MC68000 pin identifications and definitions. The microprocessor is housed in a 64 pin package that allows the use of separate (non-multiplexed) address and data buses.

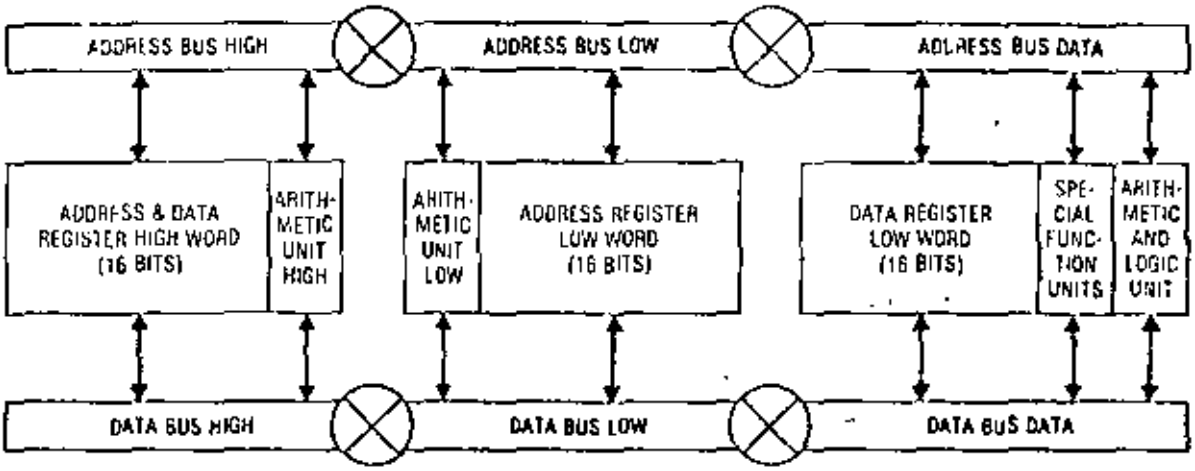


Figure 8. MC68000 execution unit configuration.

of circuit area and permits logic gates to be designed with a speed-to-power product of one picojoule. An advanced high-density *n*-channel silicon-gate MOS technology was selected for the design of the 68000. This technology supports three-micron device geometries and provides the designer with multiple MOS transistor threshold voltages. The technology allows the circuit designer to develop high-performance logic gates using minimum-size devices and to develop internal buffer circuits requiring little power.

The execution unit is a dual-bus structure that performs both address and data processing (Figure 8). The two buses are 16 bits wide, and each can be dynamically reconfigured into three independent sections as required by the microcode. Three independent arithmetic units are available to perform these calculations; also, special logic functions are provided to execute long shifts, priority encoding, and bit manipulation. Each of these units is connected to two internal buses and receives both input operands simultaneously from the registers. Each bus contains both the true and the complement logic values so that differential circuit design can be used for higher-speed operation. The execution unit directly interfaces to the external bus logic and buffers, but its operation is independent of the external timing requirements of the bus.

The control of the 68000 is implemented by microcode. The actual structure of the microprogrammed control structure is discussed in detail in another paper.[8] The microcontrol is implemented as a two-level structure. The first level contains sequences of microinstructions with short "vertical" format and complex branching capabilities. Microinstructions contain the addresses of nanoinstructions, wide "horizontal" control words, stored in the second level. The nanoinstructions directly control the execution unit. The use of microcode is motivated by the high design cost of new VLSI chips. The microcode's regularity of structure compared to combinatorial logic significantly decreases the design complexity. Microcode also permits some engineering decisions—for instance, details of specific instructions—to be delayed. In other words, once the micromachine architecture is determined, hardware implementation (circuit design) and firmware implementation (microprogramming) can be done in parallel.

Conclusion

The Motorola 68000 architecture combines advanced technology improvements with a better understanding of the architectural needs of microprocessor users and microprocessor applications. The 68000 is a step into an area previously occupied

only by high-end minicomputers. It is a 32-bit architecture that supports many data types and data sizes. The advantages of the 68000 include a flexible addressing mechanism, a simple and effective instruction set that can be used to easily build complex operations, a multilevel vectored interrupt structure, and a fast, asynchronous, nonmultiplexed bus architecture. The 68000 architecture describes a family of microprocessors designed for the expanding high-end microcomputer market. ■

References

1. J. R. Rattner, "Microprocessor Architecture—Where Do We Go From Here," *COMPCON Spring 1977 Digest of Papers*, pp. 223-224.
2. F. P. Brooks, "An Overview of Microcomputer Architecture and Software," *Proc. EUROMICRO 1976*, North Holland, pp. 1-6.
3. C. G. Bell and W. D. Strecker, "Computer Structures: What Have We Learned From the PDP-11?," *Proc. 3rd Symposium on Computer Architecture*, 1976, pp. 1-14.
4. L. A. Levanthal and W. C. Walsh, "Addressing Considerations in Microprocessor Design," *COMPCON Spring 1977 Digest of Papers*, pp. 225-229.
5. B. L. Peuto and L. J. Shustek, "Current Issues in the Architecture of Microprocessors," *Computer*, Vol. 10, No. 2, Feb. 1977, pp. 20-25.
6. S. A. Ward, "Toward the Renaissance Computer Architecture," *MIDCON 1977 Preprints*, pp. 1-6.
7. P. E. Stanley, "Address Size Independence in a 16-bit Minicomputer," *Proc. 5th Symposium on Computer Architecture*, 1978, pp. 152-157.
8. D. R. Allison, "A Design Philosophy for Microcomputer Architectures," *Computer*, Vol. 10, No. 2, Feb. 1977, pp. 35-41.
9. E. P. Stritter and H. L. Tredennick, "Microprogrammed Implementation of a Single Chip Microprocessor," *Proc. 11th Annual Microprogramming Workshop*, Nov. 1978, pp. 8-16.

centro de educación continua
división de estudios superiores
facultad de ingeniería, unam

INTRODUCCION A LOS MICROPROCESADORES Y SUS APLICACIONES

CAPITULO IV: MODOS DE DIRECCIONAMIENTO

ING. LUIS G. CORDERO BORBOA

AGOSTO 1979.

IV.- MODOS DE DIRECCIONAMIENTO

1.- ESQUEMAS DE DIRECCIONAMIENTO.

La unidad central de proceso (CPU) en las computadoras debe realizar las siguientes funciones:

- Obtener y traer de memoria primaria al CPU la siguiente instrucción a ejecutar.
- Entender los operandos, esto es, definir la localización de los operandos necesarios para ejecutar la instrucción y traerlos al CPU.
- Ejecutar la instrucción.

Para llevar a cabo las funciones anteriores el CPU debe contar con la siguiente información:

- El código de operación de la instrucción a ejecutar.
- Las direcciones de los operandos y la del resultado.
- La dirección de la siguiente instrucción a ejecutar.

Existen diferentes soluciones que satisfacen los requerimientos anteriores, los cuales determinan la arquitectura de los procesadores que las utilizan.

Se supondrán operaciones aritméticas en las que se tienen dos operandos y un resultado ya que son las que proporcionan el caso más general.

a) Máquinas de "3+1" direcciones

El formato de instrucción en este esquema de direcciona-- miento contiene todos los elementos necesitados por el CPU

para realizar sus funciones.

Un posible formato de instrucción se muestra en la figura IV.1

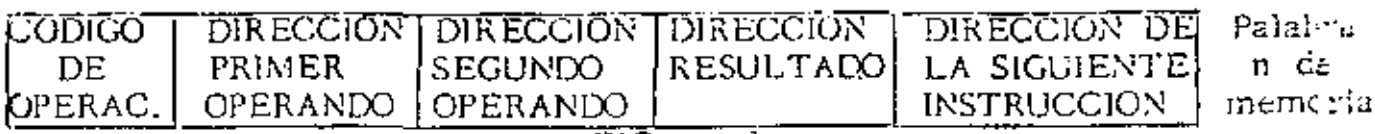


FIG. IV.1

En este caso se tienen cinco campos en el formato de instrucción: Uno para el código de operación que sirve para indicar el tipo de operación a realizar (suma, resta, multiplicación, etc.), tres campos para las direcciones de los operandos y resultado de las operaciones, un campo para indicar la dirección de la siguiente instrucción a ejecutar.

Las instrucciones para ésta máquina podrían ser escritas en forma simbólica en la siguiente forma: ADD A, B, C, D donde ADD representa el código de operación suma y A, B, C y D son nombres simbólicos asignados a localidades de memoria.

Suponiendo que existen las instrucciones suma (ADD), substracción (SUB) y multiplicación (MUL), entonces una posible traducción de la expresión A=(B*C)-(D*E) en FORTRAN a lenguaje simbólico en la máquina de 3+1 direcciones sería:

L1: MUL B, C, T1, L3

L3: MUL D, E, T2, L7

L7: SUB T2, T1, A, L8

L8: Siguiente instrucción

donde T1 y T2 representan localidades temporales usadas para guardar resultados aritméticos intermedios.

Las conclusiones más importantes en este esquema son:
Los programas no necesitan estar almacenados en memoria en forma secuencial ya que el campo de dirección de la siguiente instrucción permite conocer donde fueron almacenados.

Debido a que cada instrucción contiene en forma explícita tres direcciones, no es necesario tener en el CPU hardware para guardar los resultados de las operaciones.

b) Máquinas de "3" direcciones

Considerando que los programas se escriben secuencialmente y que por consiguiente es muy lógico almacenarlos en este mismo orden, se llega a un nuevo esquema de direccionamiento en el cual se sustituyen todos los campos de dirección de la siguiente instrucción por un solo registro dentro del procesador que lleva en forma secuencial y automáticamente la dirección de la siguiente instrucción a ejecutar. Un posible formato de instrucción se muestra en la fig. IV.2 .

Dirección de la sig. inst.	Registro en el procesador	Código de operac.	Dirección primer operando	Dirección segundo operando	Dirección resultado	Palabra n de memoria

FIG. IV.2

Utilizando este esquema de direccionamiento la expresión A=(B*C)-(D*E) en FORTRAN, quedaría expresada como:

```
MUL   B, C, T1
MUL   D, E, T2
SUB   T2, T1, A
Siguiente instrucción
```

Donde se ha suprimido la dirección de la siguiente instrucción ya que ésta es llevada en forma secuencial y automática por un registro del procesador conocido como contador del programa (PC).

Con el esquema de 3 direcciones se logra aprovechar la memoria en forma más eficiente y reducir la longitud de palabra lo que redunda directamente en los costos de la misma.

c) Máquinas de "2" direcciones.

En las operaciones aritméticas no siempre es necesario guardar el resultado en una localidad de memoria y preservar los operandos, por lo que se puede pensar en utilizar uno de ellos para---- guardar el resultado una vez que la operación se ha efectuado. Las consideraciones anteriores llevan a presentar un posible formato de instrucción en esta máquina, mostrado en la figura IV.3

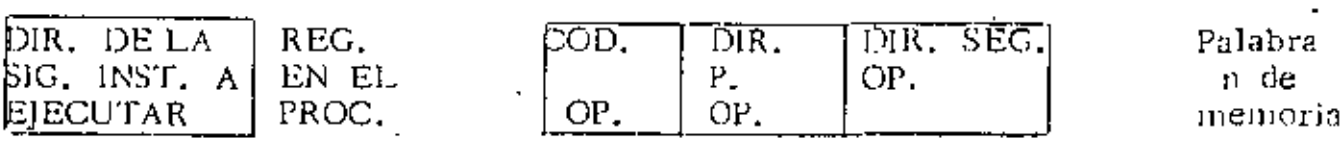


FIG. IV.3

En este esquema se usará la dirección del segundo operando como la dirección del resultado una vez que la operación se haya efectuado, por lo que el segundo operando será destruído. Así pues la expresión A=(B*C)-(D*E) en FORTRAN, quedaría:

```
MUL   B, C
MUL   D, E
SUB   E, C
ADD   C, A
```

La eliminación del campo de dirección del resultado permite reducir la longitud de la palabra de memoria y los costos de la misma, lo que permite usar este esquema en máquinas medianas y chicas.

d) Máquinas de "1" dirección

Este esquema de direccionamiento permite eliminar de todas las instrucciones el campo de dirección de uno de los operando y sustituirlo por un registro dentro del procesador, el cual contendrá a uno de los operandos. A este registro se le conoce como acumulador.

El formato de instrucción para la máquina de 1 dirección se muestra en la figura IV.4

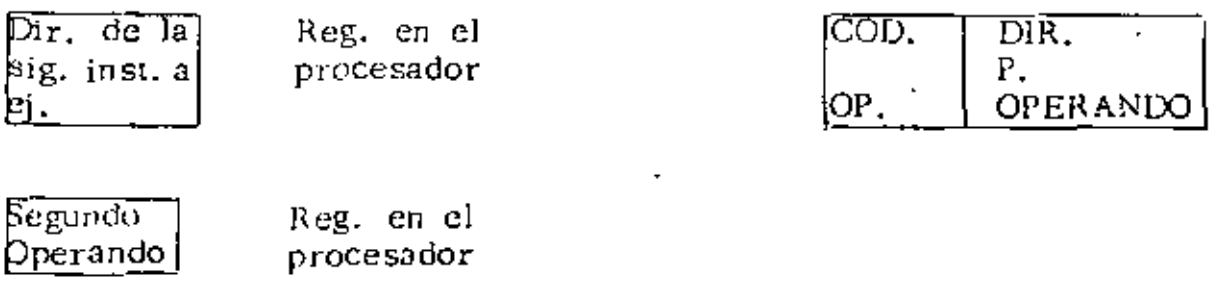


FIG. IV.4

Lo anterior implica la creación de instrucciones que permitan cargar el acumulador con el segundo operando (LAC) y depositar el contenido del acumulador en memoria (DAC).

Es importante hacer notar que todas las operaciones se llevan a cabo implícitamente contra el acumulador y que éste contendrá el resultado de la operación efectuada. La expresión A=(B*C)-(D*E) en FORTRAN, podría traducirse a:

```
LAC   D
MUL   E
DAC   T1
LAC   B
MUL   C
SUB   T1
DAC   A
```

Este esquema de direccionamiento ha sido ampliamente implementado en una gran mayoría de las minicomputadoras, como por ejemplo: PDP-8, PDP-15, IBM-1130, IBM-7090 y CDC 3600.

e) Máquinas de "0" direcciones

Este esquema de direccionamiento solo utiliza el campo de código de operación, por lo que es necesario contar con algún mecanismo que implícitamente permita conocer los operandos.

El mecanismo anterior se implementa usando una pila ó stack, el cual se puede pensar como un conjunto de localidades contiguas de

memoria accesadas usando una disciplina UEPS (últimas entradas, primeras salidas). De lo anterior se concluye que en cada momento se tendrá disponible el elemento que se encuentre en el tope del stack.

El formato de instrucción para éste esquema de direccionamiento se encuentra en la figura IV.5

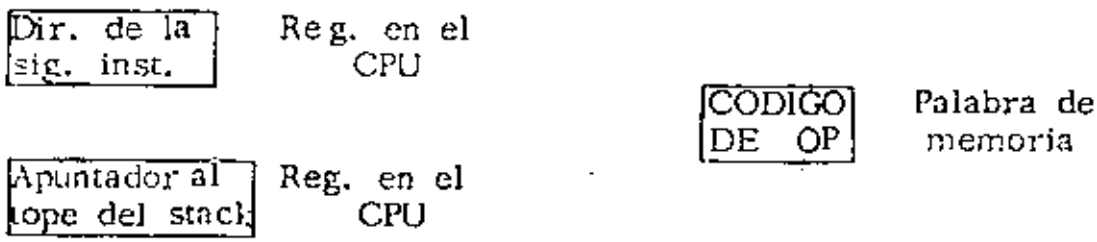
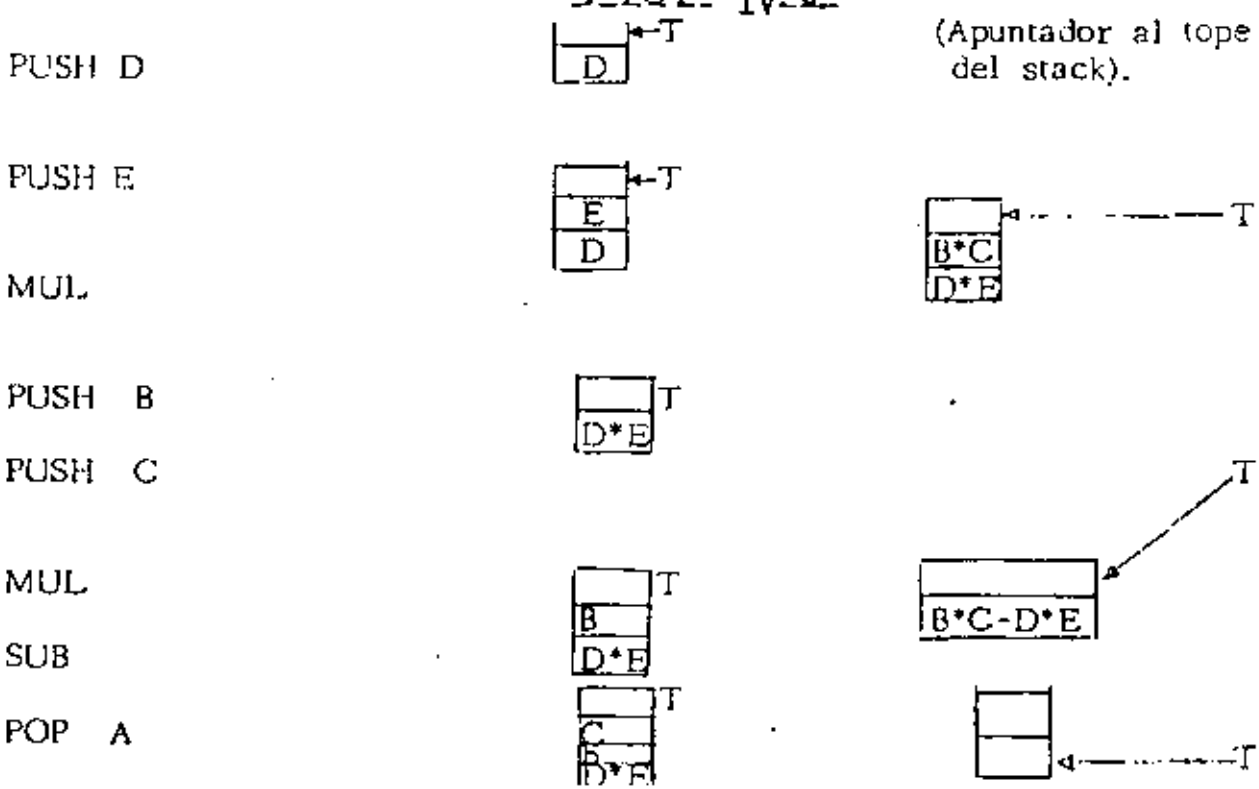


FIG. IV.5

Es necesario contar con instrucciones que permitan meter elementos de memoria al stack (PUSH) y sacar elementos del stack a memoria (POP).

La expresión A=(B*C)-(D*E) en FORTRAN, podría expresarse como:

FIG. IV.6



En la fig. IV.6 se ilustra el estado del stack después de cada una de las inst. anteriores.

Se puede concluir que el conjunto de instrucciones de la máquina no está formado solamente por instrucciones de cero direcciones ya que también se requieren instrucciones de una dirección para meter y sacar elementos al stack.

Se requiere un registro en el procesador que apunte al tope del stack y se elimine el acumulador ya que el resultado de las operaciones -- también quedará en el stack.

2.- METODOS DE DIRECCIONAMIENTO

En las máquinas de una sola dirección el formato de las instrucciones que hace referencia a memoria consta de dos campos: el campo de código de operación y el campo de dirección del operando. Si suponemos que el campo de dirección consta de n bits, entonces la máxima capacidad de memoria direccionable será 2^n localidades. Lo anterior puede resultar bastante drástico en el caso de las minicomputadoras ya que, por lo general tienen palabras de 12 ó 16 bits y si se asignan cuatro de ellos al campo de código de operación solo se pueden direccionar $2^8 = 256$ localidades de memoria en el caso de palabras de 12 bits ó $2^{12} = 4096$ localidades de memoria en el caso de palabras de 16 bits, lo cual resulta insuficiente para la gran mayoría de las aplicaciones.

Lo anterior ha ocasionado diferentes modos de direccionamiento, en los cuales el campo de dirección sirve para calcular la dirección efectiva del operando, logrando una mayor capacidad de memoria direccionable.

a) Inmediato

En este caso el operando puede estar contenido directamente en el campo de dirección ó en la localidad de memoria siguiente a la instrucción.

Será necesario dedicar un bit de la palabra para saber como se debe interpretar la instrucción.

b) Directo

Existe direccionamiento directo cuando el campo de dirección de la instrucción contiene la dirección del operando ó cuando éste campo combinado con algún registro ó palabra de memoria generan la dirección del operando.

b.1) Usando página cero

Uno de los esquemas más comunes de organización de memoria, divide ésta en n páginas de longitud fija, donde n dependerá del tamaño de la memoria y del tamaño de las páginas.

Las máquinas que usan estos esquemas generalmente usan la página cero con propósitos especiales, como son: manejo de interrupciones, traps, localidades autoincrementables, etc.

La forma de indicar si el contenido del campo de dirección se refiere a la página cero, es usando un bit para este propósito, p. ej. si este bit es cero el campo de dirección apunta a una localidad en la página cero.

b.2) Usando página actual

Si el bit de página está en uno, se asume que el campo de dirección apunta a una localidad en la página en la que se encuentra la instrucción. A esta página se le conoce como

página actual.

La dirección del operando se determina sumando los bits de orden superior del PC al campo de dirección de la instrucción.

b.3) Relativo al PC

En este modo de direccionamiento el contenido del campo de dirección de la instrucción, interpretado como un entero con signo, se suma al PC para obtener la dirección del operando.

b.4) Relativo a un registro índice

El contenido del campo de dirección de la instrucción, interpretado como un entero con signo, se suma al contenido de un registro índice para obtener la dirección del operando. En caso de existir más de un registro índice es preciso asignar los bits necesarios para su identificación.

c) Indirecto

En el direccionamiento indirecto el campo de dirección de la instrucción contiene un apuntador a la dirección del operando ó este campo combinado con algún registro ó palabra de memoria genera un apuntador a la dirección del operando.

Mediante un bit en la instrucción se puede saber si el direccionamiento usado es directo ó indirecto.

c.1) Usando página cero

El campo de dirección de la instrucción apunta a una localidad en la página cero. A su vez ésta localidad contiene la dirección del operando.

c.2) Usando página actual

El campo de dirección de la instrucción apunta a una localidad en la página actual. Esta localidad contiene la dirección del operando.

c.3) Relativo al PC

El contenido del campo de dirección de la instrucción, interpretado como un entero con signo, se suma al PC para obtener la dirección del apuntador al operando.

c.4) Relativo a un registro índice

El contenido del campo de dirección de la instrucción, interpretado como un entero con signo, se suma al contenido de un registro índice para obtener la dirección del apuntador al operando.

La combinación de todos los métodos de direccionamiento anteriores con registros de propósito general, permiten lograr modos de direccionamiento bastante poderosos. Cuando se usan los registros de propósito general, el campo de dirección de la instrucción especifica que registro se usa y como se interpreta la información que contiene.

3.- D I R E C C I O N A M I E N T O E N Z - 80

El microprocesador Z-80 es una máquina de una dirección en la que los diferentes modos de direccionamiento son usados por grupos de instrucciones y no se aplican de una forma general a todo el conjunto de instrucciones.

a) Implícito

En este modo de direccionamiento el operando no se define en forma explícita ya que el formato de instrucción es fijo y en los códigos de operación se especifica implícitamente sobre que registros del procesador actúan las instrucciones, por lo que el usuario no puede alterarlo de ninguna manera.

Los grupos de instrucciones, que utilizan este modo de direccionamiento son: carga de 8 bits; carga de 16 bits; intercambio, transferencia de bloques y búsqueda; aritméticas de propósito general y control del CPU.

Ejemplos 1.

b) Inmediato

El operando se encuentra en la localidad de memoria siguiente a la instrucción y se considera que forma parte de la misma. Los valores de los operandos inmediatos en ningun caso podrán exceder la capacidad de representación de un byte. Este modo de direccionamiento se utiliza cuando se desean realizar operaciones con valores constantes.

Los grupos de instrucciones que utilizan este modo de direccionamiento son: carga de 8 bits; aritméticas y lógicas de 8 bits y entrada/salida.

Ejemplos 2.

c) Inmediato extendido

El operando se encuentra en los dos bytes (16 bits) siguientes al código de operación de la instrucción. El primer byte contiguo al código de operación es el menos significativo y el siguiente es el más significativo.

Este modo de direccionamiento es usado por algunas instrucciones de carga de 16 bits.

Ejemplos 3.

d) Registro

El formato de instrucción contiene un campo de dirección de operando donde se especifica cual de los registros del CPU será utilizado como operando.

Los grupos de instrucciones que utilizan este modo de direccionamiento son: carga de 8 bits; carga de 16 bits; aritméticas y lógicas de 8 bits; aritméticas y lógicas de 16 bits; rotaciones y desplazamientos; encendido y apagado de bits; entrada/salida.

Ejemplos 4.

e) Registro indirecto

En este modo de direccionamiento un par de registros (16 bits) contiene la dirección de memoria en la que se encuentra el operando.

Es utilizado por los grupos de instrucciones de carga de 8 bits; intercambio, transferencia de bloques y búsqueda; rotaciones y desplazamientos; prendido y apagado de bits; saltos, llamadas y regreso de subrutinas; entrada/salida.

Ejemplos 5.

f) Extendido

La dirección del operando está contenida dentro del campo de operando de la instrucción. El campo de dirección tiene una longitud de 16 bits por lo que la máxima capacidad de memoria direccionable es de 64 K bytes.

Este modo de direccionamiento es utilizado por los grupos de instrucciones de carga de 8 bits; carga de 16 bits; saltos, llamadas y regreso de subrutinas.

Ejemplos 6.

g) Modificado de página cero

En este modo de direccionamiento el campo de dirección del operando se refiere a una localidad de memoria dentro de la página cero. Este campo de dirección consta de 3 bits y para su correcta interpretación se multiplica por 08H, obteniéndose de esta forma la referencia a las localidades deseadas.

Este modo de direccionamiento se utiliza exclusivamente por la instrucción RST.

Ejemplos 7.

h) Relativo

La dirección del operando se determina sumando al contador del programa el contenido del byte siguiente al código de operación de la instrucción.

El desplazamiento anterior se interpretará como un número en complemento a dos, con lo que se logra un rango de direccionamiento de -126 a +129 localidades relativas al contador del programa.

Este modo de direccionamiento es usado por el grupo de instrucciones de salto, llamada y regreso de subrutinas.

Ejemplos 8.

i) Indexado

La dirección del operando se determina sumando al registro de índice especificado el contenido del byte de desplazamiento.

El desplazamiento se interpreta como una cantidad en complemento a dos, con lo que se logra un rango de direccionamiento de -128 a +127 localidades relativas al registro de índice.

Los grupos de instrucciones que utilizan este modo de direccionamiento son: carga de 8 bits; aritméticas y lógicas de 8 bits; rotaciones y desplazamientos; encendido y apagado de bits; saltos, llamada y regreso de subrutinas.

Ejemplos 9.

j) Bit

Este modo de direccionamiento permite prender o apagar un bit dentro de un operando seleccionado, usando los modos antes descritos.

Ejemplos 10.

ING. LUIS G. CORDERO BORBOA
AGOSTO-79

E J E M P L O S

Se asumirá que todos los ejemplos siguientes utilizan el sistema de numeración hexadecimal.

Ejemplos 1.

```
                ; MODOS DE DIRECCIONAMIENTO DEL MICROPROCESADOR Z-80
                ; PROGRAMA CARGADO EN CASSETE CON EL NOMBRE DE -
                ; "CEC"
                ;
                ;
                ; DIRECCIONAMIENTO IMPLICITO.
                ;
0000 ED5F               LD      A,R
                ;
                ; CARGA EN EL REGISTRO A EL CONTENIDO DEL REGISTRO
                ; DE REFRESCAMIENTO R.
                ;
                ;
0002 2F                 CPL
                ;
                ; REALIZA EL COMPLEMENTO LOGICO DEL CONTENIDO DEL
                ; ACUMULADOR Y LO DEJA EN EL MISMO REGISTRO
                ;
                ;
0003 DD23               INC     IX
                ;
                ; EL CONTENIDO DEL REGISTRO DE INDICE IX SE IN--
                ; CREMENTA EN UNO
                ;
                ;
```

Ejemplos 2.

```
                ;
                ; DIRECCIONAMIENTO INMEDIATO.
                ;
0005 C634               ADD     A,34H
                ;
                ; SUMA AL CONTENIDO DEL REGISTRO ACUMULADOR A, EL
                ; DATO 34H Y DEJA EL RESULTADO EN EL MISMO RE--
                ; GISTRO.
                ;
                ;
0007 E610               AND     10H
                ;
                ; REALIZA LA OPERACION LOGICA AND ENTRE EL CONTE--
                ; NIDO DEL REGISTRO A Y EL DATO 10H, DEJANDO EL -
                ; RESULTADO EN EL MISMO REGISTRO.
                ;
                ;
```

Ejemplos 3.

```
                ;
                ; DIRECCIOINAMIENTO  INMEDIATO  EXTENDIDO
                ;
0009 FD213020           LD      IY,2030H
                ;
                ; CARGA  EN  EL  REGISTRO  DE  INDICE  IY EL DATO 2030H
                ;
                ;
0000 213F12             LD      HL,123FH
                ;
                ; CARGA  EL  REGISTRO  PAR  HL  CON  EL  DATO  123FH
                ;
                ;
```

Ejemplos 4.

```
                ;
                ; DIRECCIONAMIENTO  DE  REGISTRO
                ;
0010 4F                 LD      C,A
                ;
                ; CARGA  EL  REGISTRO  C  CON  EL  CONTENIDO  DEL  REGIS-
                ; TRO  A.
                ;
                ;
0011 80                 ADD     A,B
                ;
                ; SUMA  AL  CONTENIDO  DEL  REGISTRO  A  EL  CONTENIDO  -
                ; DEL  REGISTRO  B  Y  DEJA  EL  RESULTADO  EN  EL  REGIS-
                ; TRO  A.
                ;
                ;
0012 ED52               SBC     HL,DE
                ;
                ; SUBSTRAE  DEL  CONTENIDO  DEL  REGISTRO  HL,  EL  CONTE-
                ; NIDO  DE  LOS  REGISTROS  DE  Y  ACARREO  CY,  DEJANDO
                ; EL  RESULTADO  EN  EL  REGISTRO  HL.
                ;
                ;
```

Ejemplos 5.

```
                    ;
                    ; DIRECCIONAMIENTO DE REGISTRO INDIRECTO.
                    ;
0014 0A                     LD      A,(BC)
                    ;
                    ; CARGA EL REGISTRO A CON EL CONTENIDO DE LA LO-
                    ; CALIDAD DE MEMORIA APUNTADA POR EL REGISTRO PAR
                    ; BC.
                    ;
                    ;
0015 34                     INC     (HL)
                    ;
                    ; INCREMENTA EN UNO EL CONTENIDO DE LA LOCALIDAD
                    ; DE MEMORIA APUNTADA POR EL REGISTRO PAR HL.
                    ;
                    ;
0016 12                     LD      (DE),A
                    ;
                    ; DEPOSITA EL CONTENIDO DEL ACUMULADOR EN LA LOCA-
                    ; LIDAD DE MEMORIA APUNTADA POR EL REGISTRO PAR DE.
                    ;
                    ;
```

Ejemplos 6.

```
                    ;
                    ; DIRECCIONAMIENTO EXTENDIDO
                    ;
0017 3A2010                 LD      A,(1020H)
                    ;
                    ; CARGA EL ACUMULADOR CON EL CONTENIDO DE LA LOCA-
                    ; LIDAD DE MEMORIA 1020H.
                    ;
                    ;
001A FD220400               LD      (0004H),IY
                    ;
                    ; DEPOSITA EL CONTENIDO DEL REGISTRO DE INDICE EN
                    ; LAS LOCALIDADES DE MEMORIA 0004H (BYTE BAJO) Y
                    ; 0005H (BYTE ALTO).
                    ;
                    ;
```

Ejemplos 7.

```
                ;
                ; DIRECCIONAMIENTO  MODIFICADO  DE  PAGINA  CERO
                ;
001E CF                 RST     08H
                ;
                ; EFECTUA  UN  SALTO  INCONDICIONAL  A  LA  LOCALIDAD  DE
                ; MEMORIA  08H  DESPUES  DE  HABER  GUARDADO  EN  EL  --
                ; STACK  EL  CONTENIDO  DEL  CONTADOR  DEL  PROGRAMA
                ;
                ;
```

Ejemplos 8.

```
                ;
                ; DIRECCIONAMIENTO  RELATIVO
                ;
001F 2804               JR      Z,25H
                ;
                ; SI  LA  BANDERA  Z=1,  AL  CONTADOR  DEL  PROGRAMA SE LE
                ; SUMA  EL  VALOR  04H  CON  LO QUE  SE EFECTUARA  UN SAL-
                ; TO  A  LA  LOCALIDAD  DE  MEMORIA  25H.
                ; SI  LA  BANDERA  Z=0  SE  CONTINUARA  EJECUTANDO LA SI-
                ; GUIENTE  INSTRUCCION  DEL  PROGRAMA.
                ;
                ;
0021 30F4               JR      NC,17H
                ;
                ;SI  LA  BANDERA  C=0,  AL  CONTADOR  DEL  PROGRAMA  SE LE
                ; SUMA  EL VALOR  F4H  CON LO QUE  SE  EFECTUARA  UN SAL-
                ; TO  A  LA  LOCALIDAD  DE  MEMORIA 17H
                ; SI  LA  BANDERA  C=1  SE  CONTINUARA  EJECUTANDO  LA  -
                ; SIGUIENTE  INSTRUCCION  DEL  PROGRAMA
                ;
                ;L
```

Ejemplos 9.

```
                ;
                ; DIRECCIONAMIENTO  INDEXADO
                ;
0023 FD364313           LD      (IY+43H),13H
                ;
                ; EL  DESPLAZAMIENTO  43H  SE  SUMA  AL CONTENIDO DEL PE-
                ; GISTRO  IY  PARA DETERMINAR  LA  DIRECCION  EFECTIVA A
                ; DONDE  SE  DEPOSITARA  EL  DATO  13H
                ;
                ;
0027 DD8621             ADD     A,(IX+21H)
                ;
                ; EL  DESPLAZAMIENTO  21H  SE  SUMA  AL  CONTENIDO  DEL
                ; REGISTRO  IX  PARA  DETERMINAR  LA  DIRECCION  DEL  O-
                ; PERANDO  QUE SERA  SUMADO  AL  REGISTRO  A.  EL  RESUL-
                ; TADO  QUEDA  EN  EL  REGISTRO  A
                ;
                ;
002A DD3407             INC     (IX+07H)
                ;
                ; EL  DESPLAZAMIENTO  07H  SE  SUMA  AL  CONTENIDO  DEL
                ; REGISTRO  IX  PARA  DETERMINAR  LA  DIRECCION  DE  LA
                ; LOCALIDAD  DE  MEMORIA  CUYO  CONTENIDO  SE  INCREMEN-
                ; TA  EN  UNO.
                ;
                ;
```

Ejemplos 10.

```
                ;
                ; DIRECCIONAMIENTO  DE  BIT.
                ;
002D CBC7               SET     0000H,A
                ;
                ; ENCIENDE  EL  BIT  0  DEL  REGISTRO  A
                ;
                ;
002F CBAE               RES     05H,(HL)
                ;
                ; APAGA  EL  BIT  5  DE  LA  LOCALIDAD  DE  MEMORIA  DI-
                ; RECCIONADA  POR  EL  REGISTRO  HL.
```

centro de educación continua
división de estudios superiores
facultad de ingeniería, unam

INTRODUCCION A LOS MICROPROCESADORES Y SUS APLICACIONES

MANUALES 280

AGOSTO, 1979.

	TYPICAL PROGRAM STEPS		COMMENTS
	REGISTER	OPERATION CODED	
INITIALIZE	WR0	CHANNEL RESET	Reset SIO.
	WR0	POINTER 2	
	WR2	INTERRUPT VECTOR	Channel B only
	WR0	POINTER 3	
	WR3	AUTO ENABLES	Transmitter sends data only after $\overline{CTS}$ is detected
	WR0	POINTER 4, RESET EXTERNAL STATUS INTERRUPTS	
	WR4	PARITY INFORMATION, SDLC MODE, X1 CLOCK MODE	
	WR0	POINTER 1, RESET EXTERNAL STATUS INTERRUPTS	
	WR1	EXTERNAL INTERRUPT ENABLE, STATUS AFFECTS VECTOR, TRANSMIT INTERRUPT ENABLE OR WAIT READY MODE ENABLE	The External Interrupt mode monitors the status of the $\overline{CTS}$ and $\overline{DCD}$ inputs, as well as the status of Tx Underrun EOM latch. Transmit Interrupt interrupts when the Transmit buffer becomes empty; the Wait Ready mode can be used to transfer data on a DMA or Block Transfer basis. The first interrupt occurs when $\overline{CTS}$ becomes active, at which point flags are transmitted by the Z80 SIO. The first data byte (address field) can be loaded in the Z80-SIO after this interrupt. Flags cannot be sent to the Z80-SIO as data. Status Affects Vector used in Channel B only.
	WR0	POINTER 5	
	WR5	TRANSMIT CRC ENABLE, REQUEST TO SEND, SDLC-CRC, TRANSMIT ENABLE, TRANSMIT WORD LENGTH, DATA TERMINAL READY	SDLC-CRC mode must be defined before initializing transmit crc generator.
	WR0	RESET TRANSMIT CRC GENERATOR	Initialize CRC generator to all 1's.
IDLE MODE	EXECUTE HALT INSTRUCTION OR SOME OTHER PROGRAM		Waiting for Interrupt or Wait/Ready output to transfer data.
DATA TRANSFER AND STATUS MONITORING	*WHEN INTERRUPT (WAIT/READY) OCCURS, THE CPU DOES THE FOLLOWING:* • CHANGES TRANSMIT WORD LENGTH (IF NECESSARY) • TRANSFERS DATA BYTE FROM CPU (MEMORY) TO SIO • RESETS Tx UNDERRUN EOM LATCH (WR0)		Flags are transmitted by the SIO as soon as Transmit Enable is set and $\overline{CTS}$ becomes active. The $\overline{CTS}$ status change is the first interrupt that occurs and is followed by transmit buffer empty for subsequent transfers.
	IF LAST CHARACTER OF THE I-FIELD IS SENT, THE SIO DOES THE FOLLOWING: • SENDS CRC • SENDS CLOSING FLAG • INTERRUPTS CPU WITH BUFFER EMPTY STATUS		Word length can be changed "on the fly" for variable I-field length. The data byte can contain address, control, or I-field information (never a flag). It is a good practice to reset Tx Underrun EOM latch in the beginning of the message to avoid a false end-of-frame detection at the receiving end. This ensures that, when underrun occurs, CRC is transmitted and underrun interrupt (Tx Underrun EOM latch active) occurs. Note that "Send Abort" can be issued to the SIO in response to any interrupting condition to abort the transmission.
	CPU DOES THE FOLLOWING • ISSUES RESET Tx INTERRUPT PENDING COMMAND TO THE Z80-SIO • UPDATES NS COUNT • REPEATS THE PROCESS FOR NEXT MESSAGE, ETC.		
	IF THE VECTOR INDICATES AN ERROR, THE CPU DOES THE FOLLOWING • SENDS ABORT • EXECUTES ERROR ROUTINE • UPDATES PARAMETERS, MODES, ETC. • RETURNS FROM INTERRUPT		
TERMINATION	REDEFINE INTERRUPT MODES		Terminate gracefully.
	UPDATE MODEM CONTROL OUTPUTS		
	DISABLE TRANSMIT MODE		

Table 8. SDLC Transmit Mode

SDLC Receive

INITIALIZATION

The SDLC Receive mode is initialized by the system with the following parameters: SDLC mode, ×1 clock mode, SDLC polynomial, receive word length, etc. The flag characters must also be loaded in WR7 and the secondary address field loaded in WR6. The receiver is enabled only after all the receive parameters have been set. After all this has been done, the receiver is in the Hunt phase and remains in this phase until the first flag is received. While in the SDLC mode, the receiver never re-enters the Hunt phase, unless specifically instructed to do so by the program. The WR4 parameters must be issued prior to the WR1, WR3, WR5, WR6 and WR7 parameters.

Under program control, the receiver can enter the Address Search mode. If the Address Search bit (WR3, D2) is set, a character following the flag (first non-flag character) is compared against the programmed address in WR6 and the hardwired global address (11111111). If the SDLC frame address field matches either address, data transfer begins.

Since the Z80-SIO is capable of matching only one address character, extended address field recognition must be done by the CPU. In this case, the Z80-SIO simply transfers the additional address bytes to the CPU as if they were data characters. If the CPU determines that the frame does not have the correct address field, it can set the Hunt bit, and the Z80-SIO suspends reception and searches for a new message headed by a flag. Since the control field of the frame is transparent to the Z80-SIO, it is transferred to the CPU as a data character. Extra zeros inserted in the data stream are automatically deleted; flags are not transferred to the CPU.

DATA TRANSFER AND STATUS MONITORING

After receipt of a valid flag, the assembled characters are transferred to the receive data FIFO. The following four interrupt modes are available to transfer this data and its associated status.

No Interrupts Enabled. This mode is used for purely polled operations or for off-line conditions.

Interrupt On First Character Only. This mode is normally used to start a software polling loop or a Block Transfer instruction using $\overline{\text{WAIT/READY}}$ to synchronize the CPU or DMA device to the incoming data rate. In this mode, the Z80-SIO interrupts on the first character and thereafter only interrupts if Special Receive conditions are detected. The mode is reinitialized with the Enable Interrupt On Next Receive Character Command.

The first character received after this command is issued causes an interrupt. If External/Status interrupts are enabled, they may interrupt any time the $\overline{\text{DCD}}$ input changes state. Special Receive conditions such as End Of Frame and Receiver Overrun also cause interrupts. The End Of Frame interrupt can be used to exit the Block Transfer mode.

Interrupt On Every Character. An interrupt is generated whenever the receive FIFO contains a character. Error and Special Receive conditions generate a special vector if Status Affects Vector is selected.

Special Receive Condition Interrupts. The Special Receive Condition interrupt is not, as such, a separate interrupt mode. Before the Special Receive condition can cause an interrupt, either Interrupt On First Receive Character Only or Interrupt On Every Character must be selected. The Special Receive Condition interrupt is caused by a Receive Overrun or End Of Frame detection. Since the Receive Overrun status bit is latched, the error status read reflects an error in the current word in the receive buffer in addition to any errors received since the last Error Reset command. The Receive Overrun status bit can only be reset by the Error Reset command. The End Of Frame status bit indicates that a valid ending flag has been received and that the CRC Error and Residue codes are also valid.

Character length may be changed on the fly. If the address and control bytes are processed as 8-bit characters, the receiver may be switched to a shorter character length during the time that the first information character is being assembled. This change must be made fa[illegible] enough so it is effective before the number of bits spec[illegible] fied for the character length have been assembled. For example, if the change is to be from the 8-bit control field to a 7-bit information field, the change must be made *before* the first seven bits of the I-field are assembled.

SDLC Receive CRC Checking. Control of the receive CRC checker is automatic. It is reset by the leading flag and CRC is calculated up to the final flag. The byte that has the End Of Frame bit set is the byte that contains the result of the CRC check. If the CRC/Framing Error bit is not set, the CRC indicates a valid message. A special check sequence is used for the SDLC check because the transmitted CRC check is inverted. The final check must be 0001110100001111. The 2-byte CRC check characters must be read by the CPU and discarded because the Z80-SIO, while using them for CRC checking, treats them as ordinary data.

SDLC Receive Termination. If enabled, a special vector is generated when the closing flag is received. This signals that the byte with the End Of Frame bit set has been received. In addition to the results of the CRC check, RR1 has three bits of Residue code valid at this time. For those cases in which the number of bits in the I-field is not an integral multiple of the character length used, these bits indicate the boundary between the C check bits and the I-field bits. For a detailed description of the meaning of these bits, see the description of the residue codes in RR1 under "Z80-SIO Programming."

Any frame can be prematurely aborted by an Abort sequence. Aborts are detected if seven or more 1's occur

and cause an External/Status interrupt (if enabled) with the Break/Abort bit in RR0 set. After the Reset External/Status interrupts command has been issued a second interrupt occurs when the continuous 1's condition has been cleared. This can be used to distinguish between the Abort and Idle line conditions.

Unlike the synchronous mode, CRC calculation in SDLC does not have an 8-bit delay since all the characters are included in CRC calculation. When the second CRC character is loaded into the receive buffer, CRC calculation is complete.

Table 9 shows the typical steps required to implement a half-duplex SDLC receive mode. The complete set of command and status bit definitions is found in the next section.

FUNCTION	TYPICAL PROGRAM STEPS		COMMENTS
	REGISTER	*INFORMATION LOADED*	
	WR0	CHANNEL 2	Reset SIO
	WR0	POINTER 2	
	WR2	INTERRUPT VECTOR	Channel B only
	WR0	POINTER 4	
	WR4	PARITY INFORMATION, SYNC MODE, SDLC MODE, x1 CLOCK MODE	
	WR0	POINTER 5, RESET EXTERNAL/STATUS INTERRUPTS	
	WR5	SDLC-CRC, DATA TERMINAL READY	
	WR0	POINTER 3	
	WR3	RECEIVE CRC ENABLE, ENTER HUNT MODE, AUTO ENABLES, RECEIVE CHARACTER LENGTH, ADDRESS SEARCH MODE	'Auto Enables' enables the receiver to accept data only after $\overline{DCD}$ becomes active. Address Search Mode enables SIO to match the message address with the programmed address or the global address.
	WR0	POINTER 6	
INITIALIZE	WR6	SECONDARY ADDRESS FIELD	This address is matched against the message address in an SDLC poll operation.
	WR0	POINTER 7	
	WR7	SDLC FLAG 01111110	This flag detects the start and end of frame in an SDLC operation.
	WR0	POINTER 1, RESET EXTERNAL/STATUS INTERRUPTS	In this interrupt mode, only the Address Field (1 character only) is transferred to the CPU. All subsequent fields (Control, Information, etc.) are transferred on a DMA basis. Status Affects Vector in Channel B only.
	WR1	STATUS AFFECTS VECTOR, EXTERNAL INTERRUPT ENABLE, RECEIVE INTERRUPT ON FIRST CHARACTER ONLY.	
	WR0	POINTER 3, ENABLE INTERRUPT ON NEXT RECEIVE CHARACTER	Used to provide simple loop-back entry point for next transaction.
	WR3	RECEIVE ENABLE, RECEIVE CRC ENABLE, ENTER HUNT MODE, AUTO ENABLES, RECEIVER CHARACTER LENGTH, ADDRESS SEARCH MODE	WR3 reissued to enable receiver.
IDLE MODE	EXECUTE HALT INSTRUCTION OR SOME OTHER PROGRAM		SDLC Receive Mode is fully initialized and SIO is waiting for the opening flag followed by a matching address field to interrupt the CPU.

Table 9. SDLC Receive Mode

FUNCTION	TYPICAL PROGRAM STEPS	COMMENTS
	WHEN INTERRUPT ON FIRST CHARACTER OCCURS, THE CPU DOES THE FOLLOWING: • TRANSFERS DATA BYTE (ADDRESS BYTE) TO CPU • DETECTS AND SETS APPROPRIATE FLAG FOR EXTENDED ADDRESS FIELD • UPDATES POINTERS AND PARAMETERS • ENABLES DMA CONTROLLER • ENABLES WAIT/READY FUNCTION IN SIO • RETURNS FROM INTERRUPT	During the Hunt phase, the SIO interrupts when the programmed address ma the message address. The CPU e lishes the DMA mode and all subsequent data characters are transferred by the DMA controller to memory.
	WHEN THE READY OUTPUT BECOMES ACTIVE, THE DMA CONTROLLER DOES THE FOLLOWING • TRANSFERS THE DATA BYTE TO MEMORY • UPDATES THE POINTERS	During the DMA operation, the SIO monitors the $\overline{DCD}$ input and the Abort sequence in the data stream to interrupt the CPU with External Status error. The Special Receive condition interrupt is caused by Receive Overrun error.
DATA TRANSFER AND STATUS MONITORING	*WHEN END OF FRAME INTERRUPT OCCURS, THE CPU DOES THE FOLLOWING:* • EXITS DMA MODE (DISABLES WAIT/READY) • TRANSFERS RR1 TO THE CPU • CHECKS THE CRC ERROR BIT STATUS AND RESIDUE CODES • UPDATES NR COUNT • ISSUES ERROR RESET COMMAND TO SIO	Detection of End of Frame (Flag) causes interrupt and deactivates the Wait/Ready function. Residue codes indicate the bit structure of the last two bytes of the message, which were transferred to memory under DMA. Error Reset is issued to clear the special condition.
	WHEN "ABORT SEQUENCE DETECTED" INTERRUPT OCCURS, THE CPU DOES THE FOLLOWING: • TRANSFERS RR0 TO THE CPU • EXITS DMA MODE • ISSUES THE RESET EXTERNAL STATUS INTERRUPT COMMAND TO THE SIO • ENTERS THE IDLE MODE	Abort sequence is detected when seven or more 1's are found in the data stream CPU is waiting for Abort Sequence to terminate. Termination clears the 'Break/Abort' status bit and causes interrupt.
	WHEN THE SECOND ABORT SEQUENCE INTERRUPT OCCURS, THE CPU DOES THE FOLLOWING • ISSUES THE RESET EXTERNAL STATUS INTERRUPT COMMAND TO THE SIO	At this point, the program proceed terminate this message.
TERMINATION	REDEFINE INTERRUPT MODES, SYNC MODE AND SDLC MODES DISABLE RECEIVE MODE	

Table 9. SDLC Receive Mode (Continued)

To program the Z80-SIO, the system program first issues a series of commands that initialize the basic mode of operation and then other commands that qualify conditions within the selected mode. For example, the Asynchronous mode, character length, clock rate, number of stop bits, even or odd parity are first set, then the interrupt mode and, finally, receiver or transmitter enable. The WR4 parameters must be issued before any other parameters are issued in the initialization routine.

Both channels contain command registers that must be programmed via the system program prior to operation. The Channel Select input (B/$\overline{A}$) and the Control/Data input (C/$\overline{D}$) are the command structure addressing controls, and are normally controlled by the CPU address bus. Figure 14 illustrates the timing relationships for programming the write registers, and transferring data and status.

C/$\overline{D}$	B/$\overline{A}$	Function
0	0	Channel A Data
0	1	Channel B Data
1	0	Channel A Commands/Status
1	1	Channel B Commands/Status

Write Registers

The Z80-SIO contains eight registers (WR0-WR7) in each channel that are programmed separately by the system program to configure the functional personality of the channels. With the exception of WR0, programming the write registers requires two bytes. The first byte contains three bits (D_0-D_2) that point to the selected register; the second byte is the actual control word that is written into the register to configure the Z80-SIO.

Note that the programmer has complete freedom, after pointing to the selected register, of either reading to test the read register or writing to initialize the write register. By designing software to initialize the Z80-SIO in a modular and structured fashion, the programmer can use powerful block I/O instructions.

WR0 is a special case in that all the basic commands (CMD_0-CMD_2) can be accessed with a single byte. Reset (internal or external) initializes the pointer bits D_0-D_2 to point to WR0.

The basic commands (CMD_0-CMD_2) and the CRC controls (CRC_0, CRC_1) are contained in the first byte of any write register access. This maintains maximum flexibility and system control. Each channel contains the following control registers. These registers are addressed as commands (not data).

WRITE REGISTER 0

WR0 is the command register; however, it is also used for CRC reset codes and to point to the other registers.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
CRC Reset Code 1	CRC Reset Code 0	CMD 2	CMD 1	CMD 0	PTR 2	PTR 1	PTR 0

Pointer Bits (D_0-D_2). Bits D_0-D_2 are pointer bits that determine which other write register the next byte is to be written into or which read register the next byte is to be read from. The first byte written into each channel after a reset (either by a Reset command or by the external reset input) goes into WR0. Following a read or write to any register (except WR0), the pointer will point to WR0.

Command Bits (D_3-D_5). Three bits, D_3-D_5, are encoded to issue the seven basic Z80-SIO commands.

Command	CMD_2	CMD_1	CMD_0	
0	0	0	0	Null Command (no effect)
1	0	0	1	Send Abort (SDLC Mode)
2	0	1	0	Reset External/Status Interrupts
3	0	1	1	Channel Reset
4	1	0	0	Enable Interrupt on next Rx Character
5	1	0	1	Reset Transmitter Interrupt Pending
6	1	1	0	Error Reset (latches)
7	1	1	1	Return from Interrupt (Channel A)

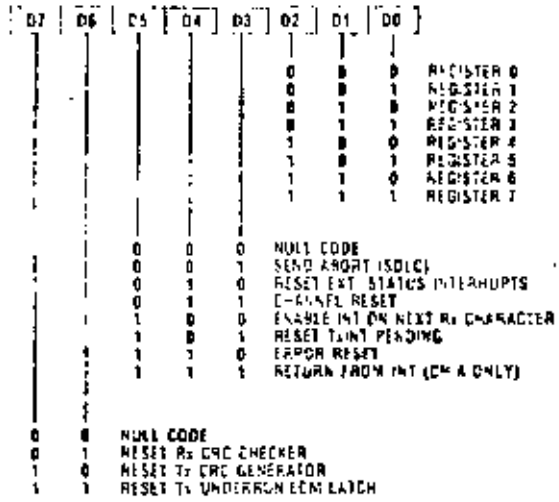
Command 0 (Null). The Null command has no effect. Its normal use is to cause the Z80-SIO to do nothing while the pointers are set for the following byte.

Command 1 (Send Abort). This command is used only with the SDLC mode to generate a sequence of eight to thirteen 1's.

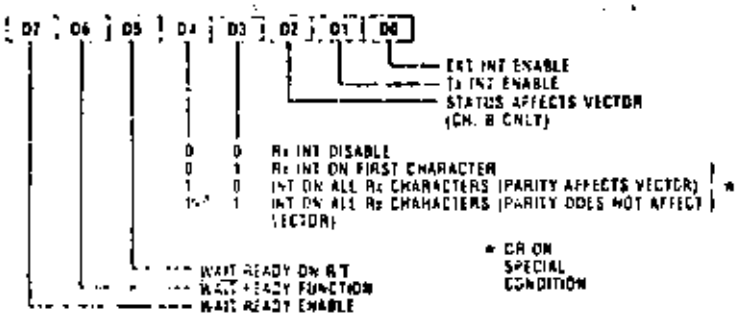
Command 2 (Reset External/Status Interrupts). After an External/Status interrupt (a change on a modem line or a break condition, for example), the status bits of RR0 are latched. This command re-enables them and allows interrupts to occur again. Latching the status bits captures short pulses until the CPU has time to read the change.

Command 3 (Channel Reset). This command performs the same function as an External Reset, but only on a single channel. Channel A Reset also resets the interrupt prioritization logic. All control registers for the channel must be rewritten after a Channel Reset command.

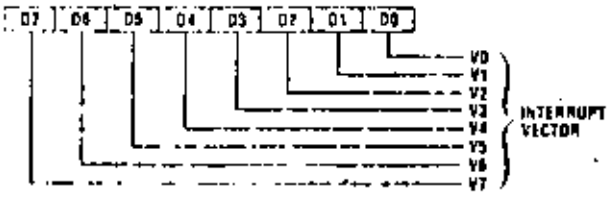
WRITE REGISTER 0



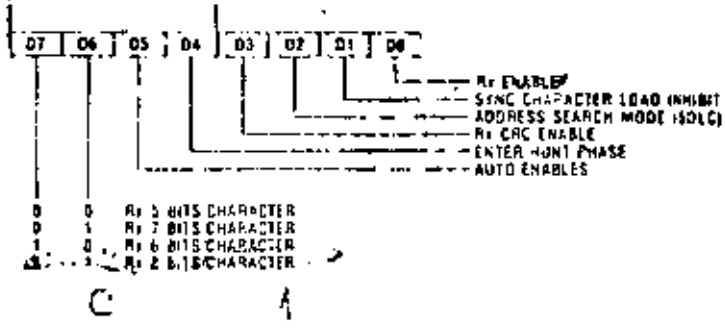
WRITE REGISTER 1



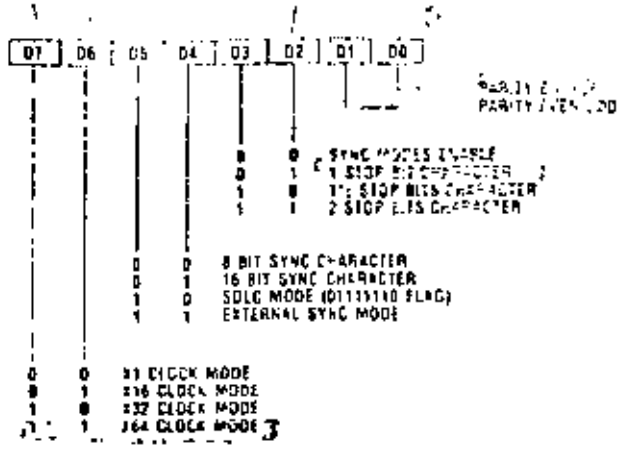
WRITE REGISTER 2 (CHANNEL B ONLY)



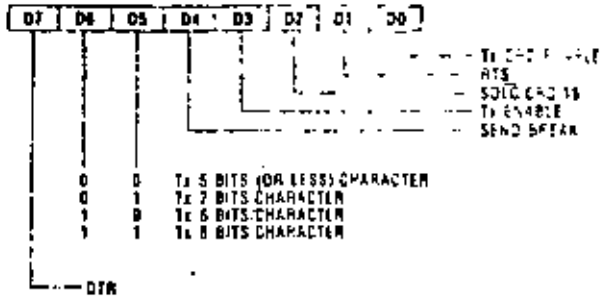
WRITER REGISTER 3



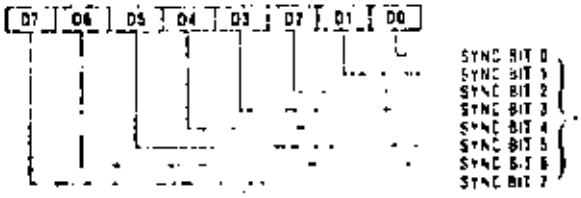
WRITE REGISTER 4



WRITE REGISTER 5

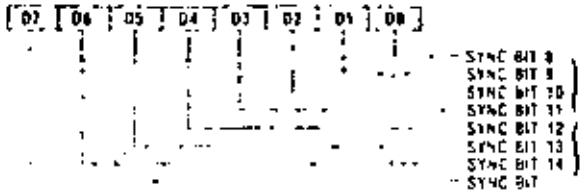


WRITE REGISTER 6



*ALSO SDLC ADDRESS FIELD

WRITE REGISTER 7



*FOR SDLC IT MUST BE PROGRAMMED TO "01111110" FOR FLAG RECOGNITION

Figure 9. Write Register Bit Functions

After a Channel Reset, four extra system clock cycles should be allowed for Z80-SIO reset time before any additional commands or controls are written into that channel. This can normally be the time used by the CPU to fetch the next op code.

Command 4 (Enable Interrupt On Next Receive Character). If the Interrupt On First Receive Character mode is selected, this command reactivates that mode after each complete message is received to prepare the Z80-SIO for the next message.

Command 5 (Reset Transmitter Interrupt Pending). The transmitter interrupts when the transmit buffer becomes empty if the Transmit Interrupt Enable mode is selected. In those cases where there are no more characters to be sent (at the end of message, for example), issuing this command prevents further transmitter interrupts until after the next character has been loaded into the transmit buffer or until CRC has been completely sent.

Command 6 (Error Reset). This command resets the error latches. Parity and Overrun errors are latched in RR1 until they are reset with this command. With this scheme, parity errors occurring in block transfers can be examined at the end of the block.

Command 7 (Return From Interrupt). This command must be issued in Channel A and is interpreted by the Z80-SIO in exactly the same way it would interpret an RETI command on the data bus. It resets the interrupt-under-service latch of the highest-priority internal device under service and thus allows lower priority devices to interrupt via the daisy chain. This command allows use of the internal daisy chain even in systems with no external daisy chain or RETI command.

CRC Reset Codes 0 and 1 (D_6 and D_7). Together, these bits select one of the three following reset commands:

CRC Reset Code 1	CRC Reset Code 0	
0	0	Null Code (no affect)
0	1	Reset Receive CRC Checker
1	0	Reset Transmit CRC Generator
1	1	Reset Tx Underrun/End Of Message latch

The Reset Transmit CRC Generator command normally initializes the CRC generator to all 0's. If the SDLC mode is selected, this command initializes the CRC generator to all 1's. The Receive CRC checker is also initialized to all 1's for the SDLC mode.

WRITE REGISTER 1

WR1 contains the control bits for the various interrupt and Wait/Ready modes.

D_7	D_6	D_5	D_4
Wait/Ready Enable	Wait Or Ready Function	Wait/Ready On Receive/ Transmit	Receive Interrupt Mode 1

D_3	D_2	D_1	D_0
Receive Interrupt Mode 0	Status Affects Vector	Transmit Interrupt Enable	External Interrupts Enable

External/Status Interrupt Enable (D_0). The External Status Interrupt Enable allows interrupts to occur as a result of transitions on the $\overline{DCD}$, $\overline{CTS}$ or $\overline{SYNC}$ inputs, as a result of a Break/Abort detection and termination, or at the beginning of CRC or sync character transmission when the Transmit Underrun/EOM latch becomes set.

Transmitter Interrupt Enable (D_1). If enabled, interrupts occur whenever the transmitter buffer becomes empty.

Status Affects Vector (D_2). This bit is active in Channel B only. If this bit is not set, the fixed vector programmed in WR2 is returned from an interrupt acknowledge sequence. If this bit is set, the vector returned from an interrupt acknowledge is variable according to the following interrupt conditions:

	V_3	V_2	V_1	
Ch B	0	0	0	Ch B Transmit Buffer Empty
	0	0	1	Ch B External/Status Change
	0	1	0	Ch B Receive Character Available
	0	1	1	Ch B Special Receive Condition*
Ch A	1	0	0	Ch A Transmit Buffer Empty
	1	0	1	Ch A External/Status Change
	1	1	0	Ch A Receive Character Available
	1	1	1	Ch A Special Receive Condition*

*Special Receive Conditions: Parity Error, Rx Overrun Error, Framing Error, End Of Frame (SDLC).

Receive Interrupt Modes 0 and 1 (D_3 and D_4). Together these two bits specify the various character-available conditions. In Receive Interrupt modes 1, 2 and 3, a Special Receive Condition can cause an interrupt and modify the interrupt vector.

D_4 Receive Interrupt Mode 1	D_3 Receive Interrupt Mode 0	
0	0	0. Receive Interrupts Disabled
0	1	1. Receive Interrupt On First Character Only
1	0	2. Interrupt On All Receive Characters—parity error is a Special Receive condition
1	1	3. Interrupt On All Receive Characters—parity error is not a Special Receive condition

Wait/Ready Function Selection (D_5-D_7). The Wait and Ready functions are selected by controlling D_5, D_6, and D_7. Wait/Ready function is enabled by setting Wait/Ready Enable (WR1, D_7) to 1. The Ready function is selected by setting D_6 (Wait/Ready function) to 1. If this bit is 1, the $\overline{WAIT/READY}$ output switches from High to Low when the Z80 SIO is ready to transfer data. The Wait function is selected by setting D_6 to 0. If this bit is

...e WAIT/READY output is in the open drain state and ...s Low when active.

...oth the Wait and Ready functions can be used in ...t the Transmit or Receive modes, but not both ...ltaneously. If D_5 (Wait/Ready on Receive/Trans... is set to 1, the Wait/Ready function responds to the ...dition of the receive buffer (empty or full). If D_5 is ...0, the Wait/Ready function responds to the condi... of the transmit buffer (empty or full).

...e logic states of the WAIT/READY output when ac... or inactive depend on the combination of modes ...ted. Following is a summary of these combinations:

If $D_7 = 0$	
And $D_6 = 1$	And $D_6 = 0$
READY is High	WAIT is floating

If $D_7 = 1$	
And $D_5 = 0$	And $D_5 = 1$
...Y Is High when transmit buffer is full.	READY Is High when receive buffer is empty.
Is Low when transmit buffer is full and an SIO data port is selected.	WAIT Is Low when receive buffer is empty and an SIO data port is selected.
...Y Is Low when transmit buffer is empty.	READY Is Low when receive buffer is full.
Is floating when transmit buffer is empty.	WAIT Is floating when receive buffer is full.

...he WAIT output High-to-Low transition occurs with ...delay time $t_{DIC(WR)}$ after the I/O request. The Low-...igh transition occurs with the delay $t_{DH\phi(WR)}$ from ...falling edge of ϕ. The READY output High-to-Low ...sition occurs with the delay $t_{DI\phi(WR)}$ from the rising ...e of ϕ. The READY output Low-to-High transition ...urs with the delay $t_{DIC(WR)}$ after IORQ falls.

...he Ready function can occur any time the Z80-SIO ...t selected. When the READY output becomes active ...), the DMA controller issues IORQ and the corre...nding B/A and C/D inputs to the Z80-SIO to transfer ... The READY output becomes inactive as soon as ...D and CS become active. Since the Ready function ... occur internally in the Z80-SIO whether it is ad...sed or not, the READY output becomes inactive ...n any CPU data or command transfer takes place. ... does not cause problems because the DMA con...ler is not enabled when the CPU transfer takes place.

...he Wait function—on the other hand—is active ... if the CPU attempts to read Z80-SIO data that has ...yet been received, which occurs frequently when ...k transfer instructions are used. The Wait function ...also become active (under program control) if the ... tries to write data while the transmit buffer is still ... The fact that the WAIT output for either channel ...become active when the opposite channel is ad...sed (because the Z80-SIO is addressed) does not ...t operation of software loops or block move in...

WRITE REGISTER 2

WR2 is the interrupt vector register; it exists in Channel B only. V_4–V_7 and V_0 are always returned exactly as written; V_1–V_3 are returned as written if the Status Affects Vector (WR1, D_2) control bit is 0. If this bit is 1, they are modified as explained in the previous section.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
V_7	V_6	V_5	V_4	V_3	V_2	V_1	V_0

WRITE REGISTER 3

WR3 contains receiver logic control bits and parameters.

D_7	D_6	D_5	D_4
Receiver Bits/ Char 1	Receiver Bits/ Char 0	Auto Enables	Enter Hunt Phase

D_3	D_2	D_1	D_0
Receiver CRC Enable	Address Search Mode	Sync Char Load Inhibit	Receiver Enable

Receiver Enable (D_0). A 1 programmed into this bit allows receive operations to begin. This bit should be set only after all other receive parameters are set and the receiver is completely initialized.

Sync Character Load Inhibit (D_1). Sync characters preceding the message (leading sync characters) are not loaded into the receive buffers if this option is selected. Because CRC calculations are not stopped by sync character stripping, this feature should be enabled only at the beginning of the message.

Address Search Mode (D_2). If SDLC is selected, setting this mode causes messages with addresses not matching the programmed address in WR6 or the global (11111111) address to be rejected. In other words, no receive interrupts can occur in the Address Search mode unless there is an address match.

Receiver CRC Enable (D_3). If this bit is set, CRC calculation starts (or restarts) at the beginning of the last character transferred from the receive shift register to the buffer stack, regardless of the number of characters in the stack. See "SDLC Receive CRC Checking" (SDLC Receive section) and "CRC Error Checking" (Synchronous Receive section) for details regarding when this bit should be set.

Enter Hunt Phase (D_4). The Z80-SIO automatically enters the Hunt phase after a reset; however, it can be re-entered if character synchronization is lost for any reason (Synchronous mode) or if the contents of an incoming message are not needed (SDLC mode). The Hunt phase is re-entered by writing a 1 into bit D_4. This sets

After a Channel Reset, four extra system clock cycles should be allowed for Z80-SIO reset time before any additional commands or controls are written into that channel. This can normally be the time used by the CPU to fetch the next op code.

Command 4 (Enable Interrupt On Next Receive Character). If the Interrupt On First Receive Character mode is selected, this command reactivates that mode after each complete message is received to prepare the Z80-SIO for the next message.

Command 5 (Reset Transmitter Interrupt Pending). The transmitter interrupts when the transmit buffer becomes empty if the Transmit Interrupt Enable mode is selected. In those cases where there are no more characters to be sent (at the end of message, for example), issuing this command prevents further transmitter interrupts until after the next character has been loaded into the transmit buffer or until CRC has been completely sent.

Command 6 (Error Reset). This command resets the error latches. Parity and Overrun errors are latched in RR1 until they are reset with this command. With this scheme, parity errors occurring in block transfers can be examined at the end of the block.

Command 7 (Return From Interrupt). This command must be issued in Channel A and is interpreted by the Z80-SIO in exactly the same way it would interpret an RETI command on the data bus. It resets the interrupt-under-service latch of the highest-priority internal device under service and thus allows lower priority devices to interrupt via the daisy chain. This command allows use of the internal daisy chain even in systems with no external daisy chain or RETI command.

CRC Reset Codes 0 and 1 (D_6 and D_7). Together, these bits select one of the three following reset commands:

CRC Reset Code 1	CRC Reset Code 0	
0	0	Null Code (no affect)
0	1	Reset Receive CRC Checker
1	0	Reset Transmit CRC Generator
1	1	Reset Tx Underrun/End Of Message latch

The Reset Transmit CRC Generator command normally initializes the CRC generator to all 0's. If the SDLC mode is selected, this command initializes the CRC generator to all 1's. The Receive CRC checker is also initialized to all 1's for the SDLC mode.

WRITE REGISTER 1

WR1 contains the control bits for the various interrupt and Wait/Ready modes.

D_7	D_6	D_5	D_4
Wait/Ready Enable	Wait Or Ready Function	Wait/Ready On Receive/ Transmit	Receive Interrupt Mode 1

D_3	D_2	D_1	D_0
Receive Interrupt Mode 0	Status Affects Vector	Transmit Interrupt Enable	External Interrupts Enable

External/Status Interrupt Enable (D_0). The External/Status Interrupt Enable allows interrupts to occur as a result of transitions on the $\overline{DCD}$, $\overline{CTS}$ or $\overline{SYNC}$ inputs, as a result of a Break/Abort detection and termination, or at the beginning of CRC or sync character transmission when the Transmit Underrun/EOM latch becomes set.

Transmitter Interrupt Enable (D_1). If enabled, interrupts occur whenever the transmitter buffer becomes empty.

Status Affects Vector (D_2). This bit is active in Channel B only. If this bit is not set, the fixed vector programmed in WR2 is returned from an interrupt acknowledge sequence. If this bit is set, the vector returned from an interrupt acknowledge is variable according to the following interrupt conditions:

	V_3	V_2	V_1	
Ch B	0	0	0	Ch B Transmit Buffer Empty
	0	0	1	Ch B External/Status Change
	0	1	0	Ch B Receive Character Available
	0	1	1	Ch B Special Receive Condition*
Ch A	1	0	0	Ch A Transmit Buffer Empty
	1	0	1	Ch A External/Status Change
	1	1	D_{λ}	Ch A Receive Character Available
	1	1	1	Ch A Special Receive Condition*

*Special Receive Conditions: Parity Error, Rx Overrun Error, Framing Error, End Of Frame (SDLC).

Receive Interrupt Modes 0 and 1 (D_3 and D_4). Together these two bits specify the various character-available conditions. In Receive Interrupt modes 1, 2 and 3, a Special Receive Condition can cause an interrupt and modify the interrupt vector.

D_4 Receive Interrupt Mode 1	D_3 Receive Interrupt Mode 0	
0	0	0. Receive Interrupts Disabled
0	1	1. Receive Interrupt On First Character Only
1	0	2. Interrupt On All Receive Characters—parity error is a Special Receive condition
1	1	3. Interrupt On All Receive Characters—parity error is not a Special Receive condition

Wait/Ready Function Selection (D_5-D_7). The Wait and Ready functions are selected by controlling D_5, D_6, and D_7. Wait/Ready function is enabled by setting Wait/Ready Enable (WR1, D_7) to 1. The Ready function is selected by setting D_6 (Wait/Ready function) to 1. If this bit is 1, the $\overline{WAIT/READY}$ output switches from High to Low when the Z80-SIO is ready to transfer data. The Wait function is selected by setting D_6 to 0. If this bit is

0, the WAIT/READY output is in the open drain state and goes Low when active.

Both the Wait and Ready functions can be used in either the Transmit or Receive modes, but not both simultaneously. If D_5 (Wait/Ready on Receive/Transmit) is set to 1, the Wait/Ready function responds to the condition of the receive buffer (empty or full). If D_5 is set to 0, the Wait/Ready function responds to the condition of the transmit buffer (empty or full).

The logic states of the WAIT/READY output when active or inactive depend on the combination of modes selected. Following is a summary of these combinations:

If $D_7 = 0$

And $D_6 = 1$	And $D_6 = 0$
READY is High	WAIT is floating

If $D_7 = 1$

And $D_6 = 0$		And $D_6 = 1$	
READY	Is High when transmit buffer is full.	READY	Is High when receive buffer is empty.
WAIT	Is Low when transmit buffer is full and an SIO data port is selected.	WAIT	Is Low when receive buffer is empty and an SIO data port is selected.
READY	Is Low when transmit buffer is empty.	READY	Is Low when receive buffer is full.
WAIT	Is floating when transmit buffer is empty.	WAIT	Is floating when receive buffer is full.

The WAIT output High-to-Low transition occurs with the delay time $t_{DI(WR)}$ after the I/O request. The Low-to-High transition occurs with the delay $t_{DH(WR)}$ from the falling edge of ϕ. The READY output High-to-Low transition occurs with the delay $t_{DL(WR)}$ from the rising edge of ϕ. The READY output Low-to-High transition occurs with the delay $t_{DI(WR)}$ after IORQ falls.

The Ready function can occur any time the Z80-SIO is not selected. When the READY output becomes active (Low), the DMA controller issues IORQ and the corresponding B/A and C/D inputs to the Z80-SIO to transfer data. The READY output becomes inactive as soon as IORQ and CS become active. Since the Ready function can occur internally in the Z80-SIO whether it is addressed or not, the READY output becomes inactive when any CPU data or command transfer takes place. This does not cause problems because the DMA controller is not enabled when the CPU transfer takes place.

The Wait function—on the other hand—is active only if the CPU attempts to read Z80-SIO data that has not yet been received, which occurs frequently when block transfer instructions are used. The Wait function can also become active (under program control) if the CPU tries to write data while the transmit buffer is still full. The fact that the WAIT output for either channel can become active when the opposite channel is addressed (because the Z80-SIO is addressed) does not affect operation of software loops or block move instructions.

WRITE REGISTER 2

WR2 is the interrupt vector register; it exists in Channel B only. V_4–V_7 and V_0 are always returned exactly written; V_1–V_3 are returned as written if the Status Affects Vector (WR1, D_2) control bit is 0. If this bit is 1, they are modified as explained in the previous section.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
V_7	V_6	V_5	V_4	V_3	V_2	V_1	V_0

WRITE REGISTER 3

WR3 contains receiver logic control bits and parameters.

D_7	D_6	D_5	D_4
Receiver Bits/ Char 1	Receiver Bits/ Char 0	Auto Enables	Enter Hunt Phase

D_3	D_2	D_1	D_0
Receiver CRC Enable	Address Search Mode	Sync Char Load Inhibit	Receiver Enable

Receiver Enable (D_0). A 1 programmed into this bit allows receive operations to begin. This bit should be set only after all other receive parameters are set and the receiver is completely initialized.

Sync Character Load Inhibit (D_1). Sync characters preceding the message (leading sync characters) are not loaded into the receive buffers if this option is selected. Because CRC calculations are not stopped by sync character stripping, this feature should be enabled only at the beginning of the message.

Address Search Mode (D_2). If SDLC is selected, setting this mode causes messages with addresses not matching the programmed address in WR6 or the global (11111111) address to be rejected. In other words, no receive interrupts can occur in the Address Search mode unless there is an address match.

Receiver CRC Enable (D_3). If this bit is set, CRC calculation starts (or restarts) at the beginning of the last character transferred from the receive shift register to the buffer stack, regardless of the number of characters in the stack. See "SDLC Receive CRC Checking" (SDLC Receive section) and "CRC Error Checking" (Synchronous Receive section) for details regarding when this bit should be set.

Enter Hunt Phase (D_4). The Z80-SIO automatically enters the Hunt phase after a reset; however, it can be re-entered if character synchronization is lost for any reason (Synchronous mode) or if the contents of an incoming message are not needed (SDLC mode). The Hunt phase is re-entered by writing a 1 into bit D_4. This sets the Sync/Hunt bit (D_4) in RR0.

Auto Enables (D_5). If this mode is selected, $\overline{DCD}$ and $\overline{CTS}$ become the receiver and transmitter enables, respectively. If this bit is not set, $\overline{DCD}$ and $\overline{CTS}$ are simply inputs to their corresponding status bits in RR0.

Receiver Bits/Characters 1 and 0 (D_7 and D_6). Together, these bits determine the number of serial receive bits assembled to form a character. Both bits may be changed during the time that a character is being assembled, but they must be changed before the number of bits currently programmed is reached.

D_7	D_6	Bits/Character
0	0	5
0	1	7
1	0	6
1	1	8

WRITE REGISTER 4

WR4 contains the control bits that affect both the receiver and transmitter. In the transmit and receive initialization routine, these bits should be set before issuing WR1, WR3, WR5, WR6, and WR7.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
Clock Rate 1	Clock Rate 0	Sync Modes 1	Sync Modes 0	Stop Bits 1	Stop Bits 0	Parity Even/Odd	Parity

Parity (D_0). If this bit is set, an additional bit position (in addition to those specified in the bits/character control) is added to transmitted data and is expected in receive data. In the Receive mode, the parity bit received is transferred to the CPU as part of the character, unless 8 bits/character is selected.

Parity Even/$\overline{\text{Odd}}$ (D_1). If parity is specified, this bit determines whether it is sent and checked as even or odd (1 = even).

Stop Bits 0 and 1 (D_2 and D_3). These bits determine the number of stop bits added to each asynchronous character sent. The receiver always checks for one stop bit. A special mode (00) signifies that a synchronous mode is to be selected.

D_3 Stop Bits 1	D_2 Stop Bits 0	
0	0	Sync modes
0	1	1 stop bit per character
1	0	1½ stop bits per character
1	1	2 stop bits per character

Sync Modes 0 and 1 (D_4 and D_5). These bits select the various options for character synchronization:

Sync Mode 1	Sync Mode 0	
0	0	8-bit programmed sync
0	1	16-bit programmed sync
1	0	SDLC mode (01111110 flag pattern)
1	1	External Sync mode

Clock Rate 0 and 1 (D_6 and D_7). These bits specify the multiplier between the clock ($\overline{TxC}$ and $\overline{RxC}$) and data rates. For synchronous modes, the × 1 clock rate must be specified. Any rate may be specified for asynchronous modes; however, the same rate must be used for both the receiver and transmitter. The system clock in all modes must be at least 4.5 times the data rate. If the × 1 clock rate is selected, bit synchronization must be accomplished externally.

Clock Rate 1	Clock Rate 0	
0	0	Data Rate × 1 = Clock Rate
0	1	Data Rate × 16 = Clock Rate
1	0	Data Rate × 32 = Clock Rate
1	1	Data Rate × 64 = Clock Rate

WRITE REGISTER 5

WR5 contains control bits that affect the operation of transmitter, with the exception of D_3, which affects the transmitter and receiver.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
DTR	Tx Bits/ Char 1	Tx Bits/ Char 0	Send Break	Tx Enable	CRC-16/ $\overline{\text{SDLC}}$	RTS	Tx CRC Enable

Transmit CRC Enable (D_0). This bit determines if CRC is calculated on a particular transmit character. If it is set at the time the character is loaded from the transmit buffer into the transmit shift register, CRC is calculated on the character. CRC is not automatically sent unless this bit is set when the Transmit Underrun condition exists.

Request To Send (D_1). This is the control bit for the $\overline{RTS}$ pin. When the $\overline{RTS}$ bit is set, the $\overline{RTS}$ pin goes Low; when reset, $\overline{RTS}$ goes High. In the Asynchronous mode, $\overline{RTS}$ goes High only after all the bits of the character are transmitted and the transmitter buffer is empty. In Synchronous modes, the pin directly follows the state of the bit.

CRC-16/$\overline{\text{SDLC}}$ (D_2). This bit selects the CRC polynomial used by both the transmitter and receiver. When set, the CRC-16 polynomial ($X^{16} + X^{15} + X^2 + 1$) is used; when reset the SDLC polynomial ($X^{16} + X^{12} + X^5 + 1$) is used. If the SDLC mode is selected, the CRC generator and checker are preset to all 1's and a special check sequence is used. The SDLC CRC polynomial must be selected when the SDLC mode is selected. If the SDLC mode is not selected, the CRC generator and checker are preset to all 0's (for both polynomials).

Transmit Enable (D_3). Data is not transmitted until this bit is set, and the Transmit Data output is held marking. Data or sync characters in the process of being transmitted are completely sent if this bit is reset after transmission has started. If the transmitter is disabled during the transmission of a CRC character, sync or flag characters are sent instead of CRC.

Send Break (D_4). When set, this bit immediately forces the Transmit Data output to the spacing condition, regardless of any data being transmitted. When reset, TxD returns to marking.

Transmit Bits/Character 0 and 1 (D_5 and D_6). Together, D_6 and D_5 control the number of bits in each byte transferred to the transmit buffer.

D_6 Transmit Bits/ Character 1	D_5 Transmit Bits/ Character 0	Bits/Character
0	0	Five or less
0	1	7
1	0	6
1	1	8

Bits to be sent must be right justified, least-significant bits first. The Five Or Less mode allows transmission of one to five bits per character; however, the CPU should format the data character as shown in the following table.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0	
1	1	1	1	0	0	0	D	Sends one data bit
1	1	1	0	0	0	D	D	Sends two data bits
1	1	0	0	0	D	D	D	Sends three data bits
1	0	0	0	D	D	D	D	Sends four data bits
0	0	0	D	D	D	D	D	Sends five data bits

Data Terminal Ready (D_7). This is the control bit for the $\overline{DTR}$ pin. When set, $\overline{DTR}$ is active (Low); when reset, $\overline{DTR}$ is inactive (High).

WRITE REGISTER 6

This register is programmed to contain the transmit sync character in the Monosync mode, the first eight bits of a 16-bit sync character in the Bisync mode, or a transmit sync character in the External Sync mode. In the SDLC mode, it is programmed to contain the secondary address field used to compare against the address field of the SDLC frame.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
Sync 7	Sync 6	Sync 5	Sync 4	Sync 3	Sync 2	Sync 1	Sync 0

WRITE REGISTER 7

This register is programmed to contain the receive sync character in the Monosync mode, a second byte (last eight bits) of a 16-bit sync character in the Bisync mode, or a flag character (01111110) in the SDLC mode. WR7 is not used in the External Sync mode.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
Sync 15	Sync 14	Sync 13	Sync 12	Sync 11	Sync 10	Sync 9	Sync 8

Read Registers 10

The Z80-SIO contains three registers, RR0 thru RR2 (Figure 10), that can be read to obtain the status information for each channel (except for RR2—Channel B only). The status information includes error conditions, interrupt vector and standard communications-interface signals.

To read the contents of a selected read register other than RR0, the system program must first write the pointer byte to WR0 in exactly the same way as a write register operation. Then, by executing an input instruction, the contents of the addressed read register can be read by the CPU.

The status bits of RR0 and RR1 are carefully grouped to simplify status monitoring. For example, when the interrupt vector indicates that a Special Receive Condition interrupt has occurred, all the appropriate error bits can be read from a single register (RR1).

READ REGISTER 0

This register contains the status of the receive and transmit buffers; the $\overline{DCD}$, $\overline{CTS}$ and $\overline{SYNC}$ inputs; the Transmit Underrun/EOM latch; and the Break/Abort latch.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
Break/ Abort	Transmit Underrun/ EOM	CTS	Sync/ Hunt	DCD	Transmit Buffer Empty	Interrupt Pending (Ch. A only)	Receive Character Available

Receive Character Available (D_0). This bit is set when at least one character is available in the receive buffer; it is reset when the receive FIFO is completely empty.

Interrupt Pending (D_1). Any interrupting condition in the Z80-SIO causes this bit to be set; however, it is readable only in Channel A. This bit is mainly used in applications that do not have vectored interrupts available. During the interrupt service routine in these applications, this bit indicates if any interrupt conditions are present in the Z80-SIO. This eliminates the need for analyzing all the bits of RR0 in both Channels A and B. Bit D_1 is reset when all the interrupting conditions are satisfied. This bit is always 0 in Channel B.

Transmit Buffer Empty (D_2). This bit is set whenever the transmit buffer becomes empty, except when a CRC character is being sent in a synchronous or SDLC mode. The bit is reset when a character is loaded into the transmit buffer. This bit is in the set condition after a reset.

Data Carrier Detect (D_3). The DCD bit shows the state of the $\overline{DCD}$ input at the time of the last change of any of the five External/Status bits (DCD, CTS, Sync/Hunt, Break/Abort or Transmit Underrun/EOM). Any transition of the $\overline{DCD}$ input causes the DCD bit to be latched

and causes an External/Status interrupt. To read the current state of the DCD bit, this bit must be read immediately following a Reset External/Status Interrupt command.

Sync/Hunt (D_4). Since this bit is controlled differently in the Asynchronous, Synchronous and SDLC modes, its operation is somewhat more complex than that of the other bits and therefore requires more explanation.

In asynchronous modes, the operation of this bit is similar to the DCD status bit, except that Sync/Hunt shows the state of the $\overline{SYNC}$ input. Any High-to-Low transition on the $\overline{SYNC}$ pin sets this bit and causes an External/Status interrupt (if enabled). The Reset External/Status Interrupt command is issued to clear the interrupt. A Low-to-High transition clears this bit and sets the External/Status interrupt. When the External/Status interrupt is set by the change in state of any other input or condition, this bit shows the inverted state of the $\overline{SYNC}$ pin at the time of the change. This bit must be read immediately following a Reset External/Status Interrupt command to read the current state of the $\overline{SYNC}$ input.

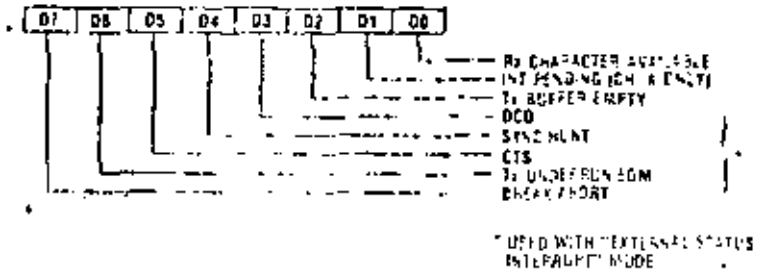
In the External Sync mode, the Sync/Hunt bit operates in a fashion similar to the Asynchronous mode, except the Enter Hunt Mode control bit enables the external sync detection logic. When the External Sync Mode and Enter Hunt Mode bits are set (for example, when the receiver is enabled following a reset), the $\overline{SYNC}$ input must be held High by the external logic until external character synchronization is achieved. A High at the $\overline{SYNC}$ input holds the Sync/Hunt status bit in the reset condition.

When external synchronization is achieved, $\overline{SYNC}$ must be driven Low on the second rising edge of $\overline{RxC}$ after that rising edge of $\overline{RxC}$ on which the last bit of the sync character was received. In other words, after the sync pattern is detected, the external logic must wait for two full Receive Clock cycles to activate the $\overline{SYNC}$ input. Once $\overline{SYNC}$ is forced Low, it is a good practice to keep it Low until the CPU informs the external sync logic that synchronization has been lost or a new message is about to start. Refer to Figure 18 for timing details. The High-to-Low transition of the $\overline{SYNC}$ input sets the Sync/Hunt bit, which—in turn—sets the External/Status interrupt. The CPU must clear the interrupt by issuing the Reset External/Status Interrupt command.

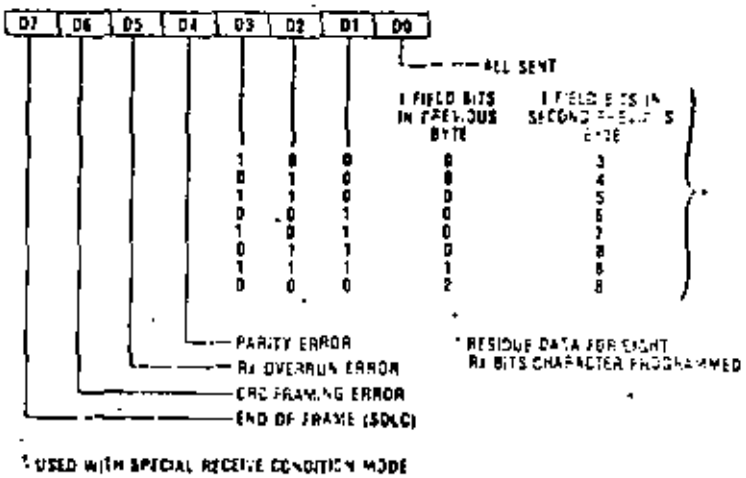
When the $\overline{SYNC}$ input goes High again, another External/Status interrupt is generated that must also be cleared. The Enter Hunt Mode control bit is set whenever character synchronization is lost or the end of message is detected. In this case, the Z80-SIO again looks for a High-to-Low transition on the $\overline{SYNC}$ input and the operation repeats as explained previously. This implies the CPU should also inform the external logic that character synchronization has been lost and that the Z80-SIO is waiting for $\overline{SYNC}$ to become active.

In the Monosync and Bisync Receive modes, the Sync/Hunt status bit is initially set to 1 by the Enter Hunt Mode bit. The Sync/Hunt bit is reset when the Z80-SIO establishes character synchronization. The

READ REGISTER 0



READ REGISTER 1†



READ REGISTER 2

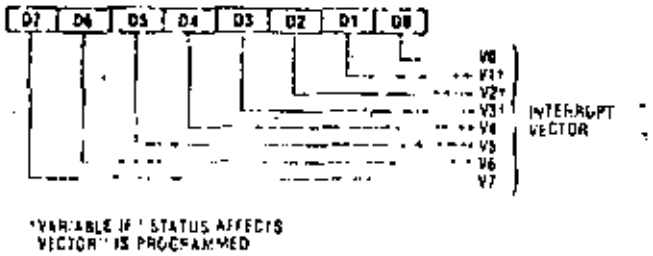


Figure 10. Read Register Bit Functions

High-to-Low transition of the Sync/Hunt bit causes an External/Status interrupt that must be cleared by the CPU issuing the Reset External/Status Interrupt command. This enables the Z80-SIO to detect the next transition of other External/Status bits.

When the CPU detects the end of message or that character synchronization is lost, it sets the Enter Hunt Mode control bit, which—in turn—sets the Sync/Hunt bit to 1. The Low-to-High transition of the Sync/Hunt bit sets the External/Status interrupt, which must also be cleared by the Reset External/Status Interrupt command. Note that the $\overline{\text{SYNC}}$ pin acts as an output in this mode and goes Low every time a sync pattern is detected in the data stream.

In the SDLC mode, the Sync/Hunt bit is initially set by the Enter Hunt mode bit, or when the receiver is disabled. In any case, it is reset to 0 when the opening flag of the first frame is detected by the Z80-SIO. The External/Status interrupt is also generated, and should be handled as discussed previously.

Unlike the Monosync and Bisync modes, once the Sync/Hunt bit is reset in the SDLC mode, it does not need to be set when the end of message is detected. The Z80-SIO automatically maintains synchronization. The only way the Sync/Hunt bit can be set again is by the Enter Hunt Mode bit, or by disabling the receiver.

Clear To Send (D_5). This bit is similar to the DCD bit, except that it shows the inverted state of the $\overline{\text{CTS}}$ pin.

Transmit Underrun/End Of Message (D_6). This bit is in a set condition following a reset (internal or external). The only command that can reset this bit is the Reset Transmit Underrun/EOM Latch command (WR0, D_6 and D_7). When the Transmit Underrun condition occurs, this bit is set; its becoming set causes the External/Status interrupt, which must be reset by issuing the Reset External/Status Interrupt command bits (WR0). This status bit plays an important role in conjunction with other control bits in controlling a transmit operation. Refer to "Bisync Transmit Underrun" and "SDLC Transmit Underrun" for additional details.

Break/Abort (D_7). In the Asynchronous Receive mode, this bit is set when a Break sequence (null character plus framing error) is detected in the data stream. The External/Status interrupt, if enabled, is set when Break is detected. The interrupt service routine must issue the Reset External/Status Interrupt command (WR0, CMD_2) to the break detection logic so the Break sequence termination can be recognized.

The Break/Abort bit is reset when the termination of the Break sequence is detected in the incoming data stream. The termination of the Break sequence also causes the External/Status interrupt to be set. The Reset External/Status Interrupt command must be issued to enable the break detection logic to look for the next Break sequence. A single extraneous null character is present in the receiver after the termination of a break; it should be read and discarded.

In the SDLC Receive mode, this status bit is set by the detection of an Abort sequence (seven or more 1's). The External/Status interrupt is handled the same way as in the case of a Break. The Break/Abort bit is not used in the Synchronous Receive mode.

READ REGISTER 1

This register contains the Special Receive condition status bits and Residue codes for the I-field in the SDLC Receive Mode.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
End Of Frame (SDLC)	CRC/ Framing Error	Receiver Overrun Error	Parity Error	Residue Code 2	Residue Code 1	Residue Code 0	All Sent

All Sent (D_0). In asynchronous modes, this bit is set when all the characters have completely cleared the transmitter. Transitions of this bit do not cause interrupts. It is always set in synchronous modes.

Residue Codes 0, 1 and 2 (D_1–D_3). In those cases of the SDLC receive mode where the I-field is not an integral multiple of the character length, these three bits indicate the length of the I-field. These codes are meaningful only for the transfer in which the End Of Frame bit is set (SDLC). For a receive character length of eight bits per character, the codes signify the following:

Residue Code 2	Residue Code 1	Residue Code 0	I-Field Bits In Previous Byte	I-Field Bits In Second Previous Byte
1	0	0	0	3
0	1	0	0	4
1	1	0	0	5
0	0	1	0	6
1	0	1	0	7
0	1	1	0	8
1	1	1	1	8
0	0	0	2	8

I-Field bits are right-justified in all cases.

If a receive character length different from eight bits is used for the I-field, a table similar to the previous one may be constructed for each different character length. For no residue (that is, the last character boundary coincides with the boundary of the I-field and CRC field), the Residue codes are:

Bits per Character	Residue Code 2	Residue Code 1	Residue Code 0
8 Bits per Character	0	1	1
7 Bits per Character	0	0	0
6 Bits per Character	0	1	0
5 Bits per Character	0	0	1

Parity Error (D_4). When parity is enabled, this bit is set for those characters whose parity does not match the programmed sense (even/odd). The bit is latched, so once an error occurs, it remains set until the Error Reset command (WR0) is given.

Receive Overrun Error (D_5). This bit indicates that more than three characters have been received without a read from the CPU. Only the character that has been written over is flagged with this error, but when this character is read, the error condition is latched until reset by the Error Reset command. If Status Affects Vector is enabled, the character that has been overrun interrupts with a Special Receive Condition vector.

CRC/Framing Error (D_6). If a Framing Error occurs (asynchronous modes), this bit is set (and not latched) for the receive character in which the Framing Error occurred. Detection of a Framing Error adds an additional one-half of a bit time to the character time so the Framing Error is not interpreted as a new start bit. In synchronous and SDLC modes, this bit indicates the result of comparing the CRC checker to the appropriate check value. This bit is reset by issuing an Error Reset command. The bit is not latched, so it is always updated when the next character is received. When used for CRC error and status in synchronous modes, it is usually set since most bit combinations result in a non-zero CRC except for a correctly completed message.

End Of Frame (D_7). This bit is used only with the SDLC mode and indicates that a valid ending flag has been received and that the CRC Error and Residue codes are also valid. This bit can be reset by issuing the Error Reset command. It is also updated by the first character of the following frame.

READ REGISTER 2 (Ch. B Only)

This register contains the interrupt vector written into WR2 if the Status Affects Vector control bit is not set. If the control bit is set, it contains the modified vector shown in the Status Affects Vector paragraph of the Write Register 1 section. When this register is read, the vector returned is modified by the highest priority interrupting condition at the time of the read. If no interrupts are pending, the vector is modified with $V_3=0$, $V_2=1$ and $V_1=1$. This register may be read only through Channel B.

D_7	D_6	D_5	D_4	D_3	D_2	D_1	D_0
V_7	V_6	V_5	V_4	V_3	V_2	V_1	V_0
				Variable if Status Affects Vector is enabled			

The flexibility and versatility of the Z80-SIO make it useful for numerous applications, a few of which are included here. These examples show several applications that combine the Z80-SIO with other members of the Z80 family.

Figure 11 shows simple processor-to-processor communication over a direct line. Both remote processors in this system can communicate to the Z80-CPU with different protocols and data rates. Depending on the complexity of the application, other Z80 peripheral circuits (Z80-CTC, for example) may be required. The unused channel of the Z80-SIO can be used to control other peripherals or they can be connected to other remote processors.

Figure 12 illustrates how both channels of a single Z80-SIO are used with modems that have primary and secondary, or reverse channel options. Alternatively, two modems without these options can be connected to the Z80-SIO. A suitable baud-rate generator (Z80-CTC) must be used for asynchronous modems.

Figure 13 shows the Z80-SIO in a data concentrator, a relatively complex application that uses two Z80-SIOs to perform a variety of functions. The data concentrator can be used to collect data from many terminals over low-speed lines and transmit it over a single high-speed line after editing and reformatting.

The Z80-DMA controller circuit is used with Z80-SIO #2 to transmit the reformatted data at high speed with the required protocol. The high-speed modem provides the transmit clock for this channel. The Z80-CTC counter-timer circuit supplies the transmit and receive clocks for the low-speed lines and is also used as a time-out counter for various functions.

Z80-SIO #1 controls local or remote terminals. A single intelligent terminal is shown within the dashed lines. The terminal employs a Z80-SIO to communicate to the data concentrator on one channel while providing the interface to a line printer over its second channel. The intelligent terminal shown could be designed to operate interactively with the operator.

Depending on the software and hardware capabilities built into this system, the data concentrator can employ store-and-forward or hold-and-forward methods for regulating information traffic between slow terminals and the high-speed remote processor. If the high-speed channel is provided with a dial-out option, the channel can be connected to a number of remote processors over a switched line.

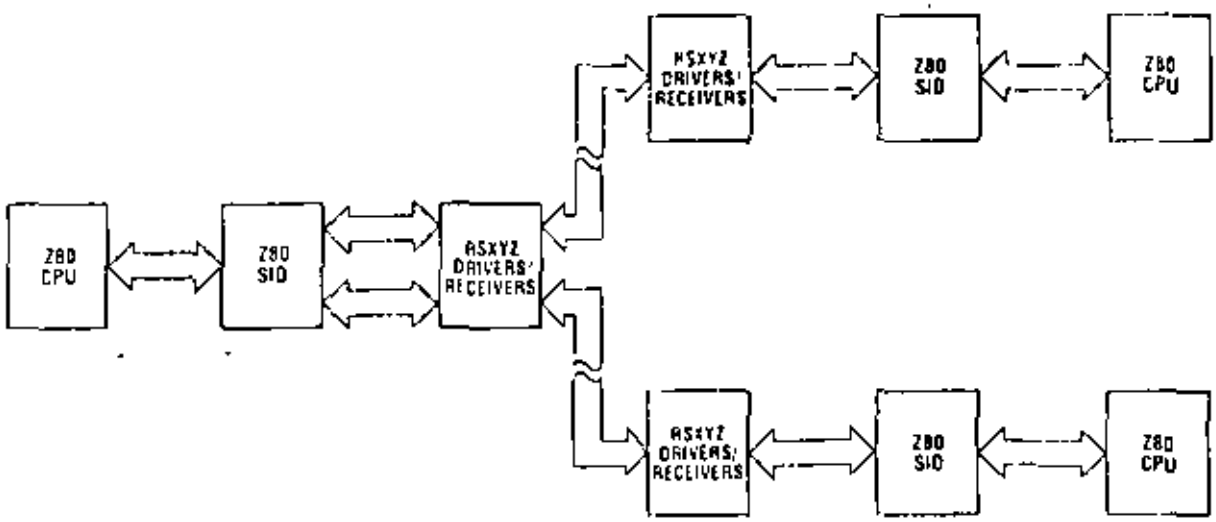


Figure 11. Synchronous/Asynchronous Processor-to-Processor Communication (Direct Wire to Two Remote Locations)

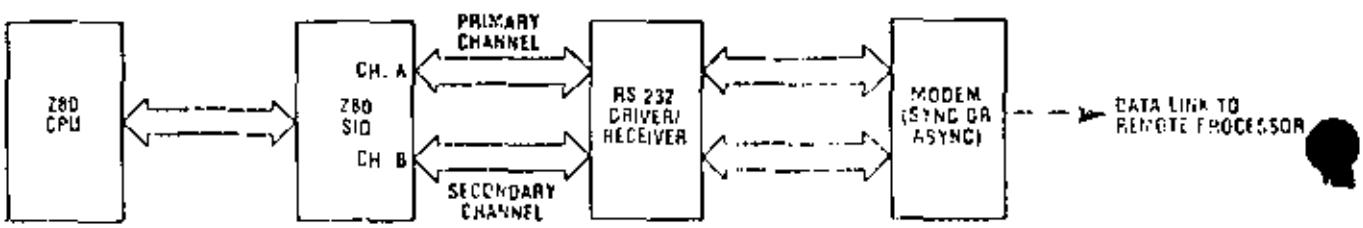


Figure 12. Synchronous/Asynchronous Processor-to-Processor Communication (Using Telephone Line)

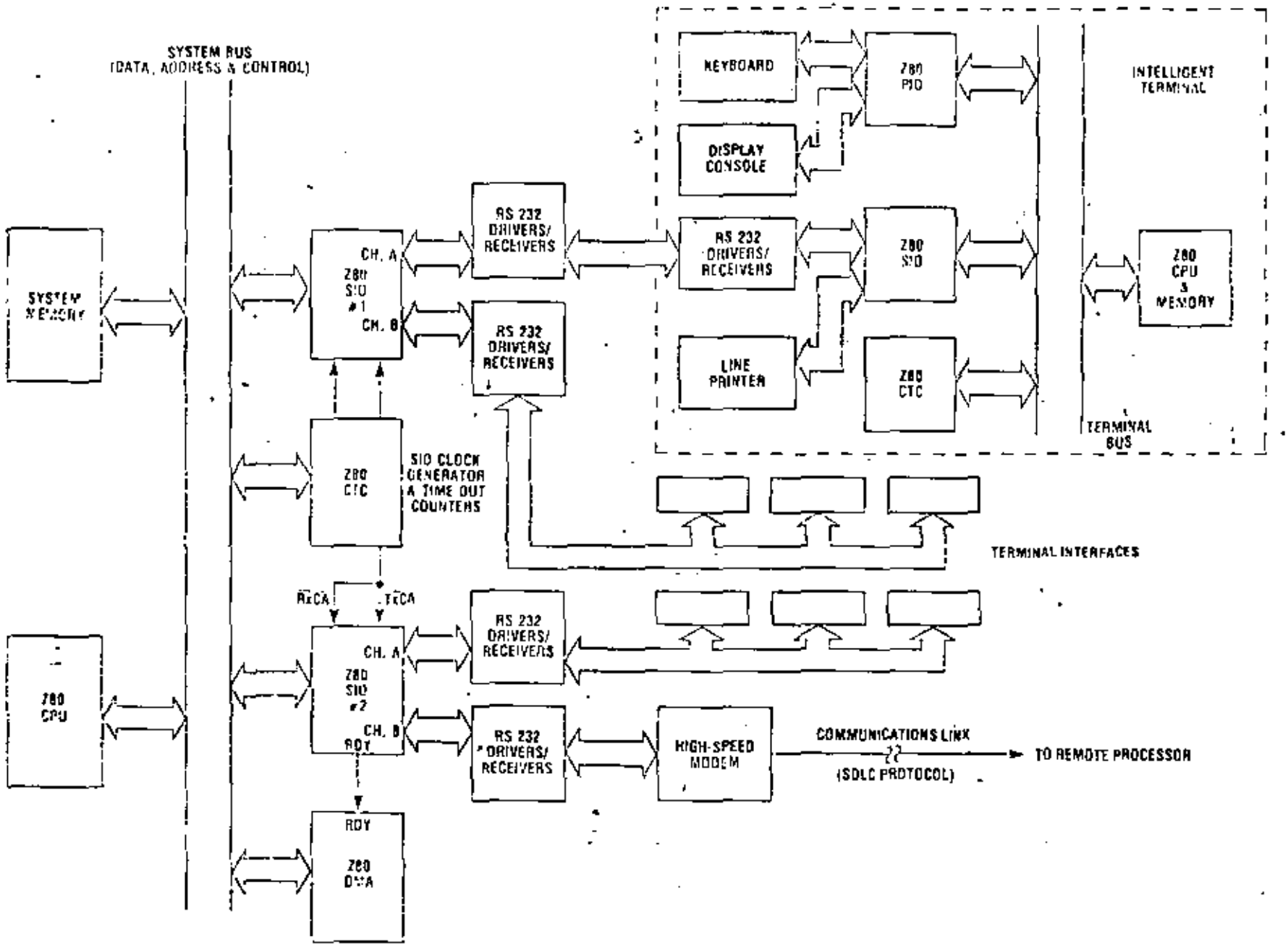


Figure 17 Concentrator

READ CYCLE

The timing signals generated by a Z80-CPU input instruction to read a Data or Status byte from the Z80-SIO are illustrated in Figure 14a.

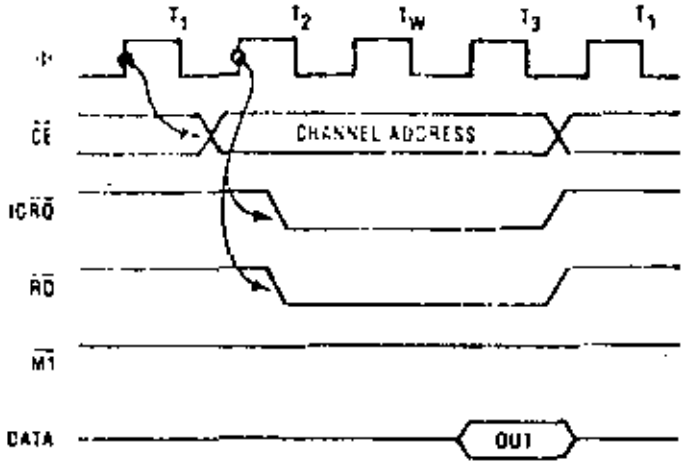


Figure 14a. Read Cycle

WRITE CYCLE

Figure 14b illustrates the timing and data signals generated by a Z80-CPU output instruction to write a Data or Control byte into the Z80-SIO.

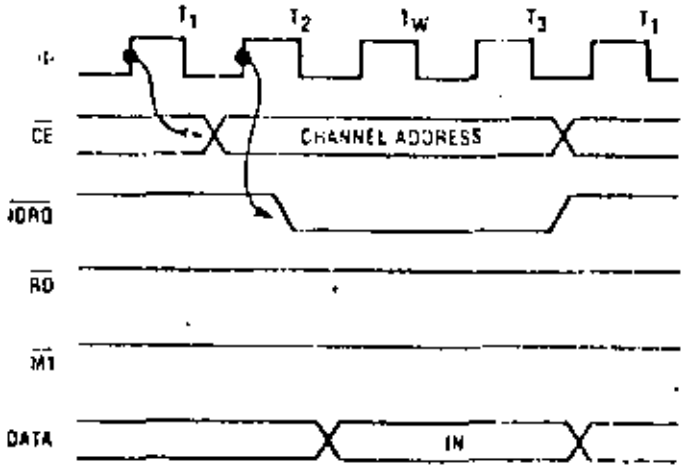


Figure 14b. Write Cycle

INTERRUPT ACKNOWLEDGE CYCLE

After receiving an Interrupt Request signal ($\overline{INT}$ pulled Low), the Z80-CPU sends an Interrupt Acknowledge signal ($\overline{M1}$ and $\overline{IORQ}$ both Low). The daisy-chained interrupt circuits determine the highest priority interrupt requestor. The IEI of the highest priority peripheral is terminated High. For any peripheral that has no interrupt pending or under service, IEO = IEI. Any peripheral that does have an interrupt pending or under service forces its IEO Low.

To insure stable conditions in the daisy chain, all interrupt status signals are prevented from changing while $\overline{M1}$ is Low. When $\overline{IORQ}$ is Low, the highest priority interrupt requestor (the one with IEI High) places its interrupt vector on the data bus and sets its internal interrupt-under-service latch.

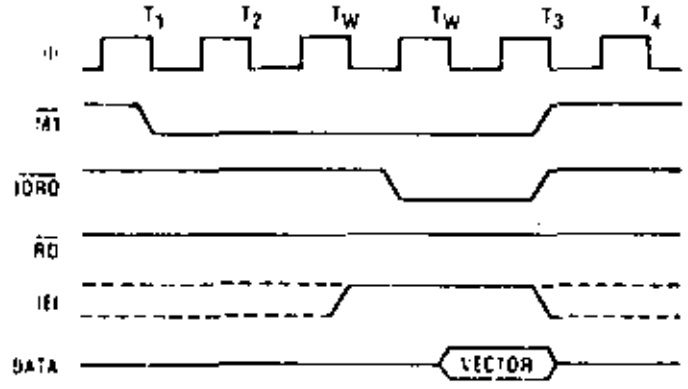


Figure 14c. Interrupt Acknowledge Cycle

RETURN FROM INTERRUPT CYCLE

Normally, the Z80-CPU issues a RETI (RETurn from Interrupt) instruction at the end of an interrupt service routine. RETI is a 2-byte opcode (ED-4D) that resets the interrupt-under-service latch to terminate the interrupt that has just been processed. This is accomplished by manipulating the daisy chain in the following way.

The normal daisy chain operation can be used to detect a pending interrupt; however, it cannot distinguish between an interrupt under service and a pending unacknowledged interrupt of a higher priority. Whenever "ED" is decoded, the daisy chain is modified by forcing High the IEO of any interrupt that has not yet been acknowledged. Thus the daisy chain identifies the device presently under service as the only one with an IEI High and an IEO Low. If the next opcode byte is "4D," the interrupt-under-service latch is reset.

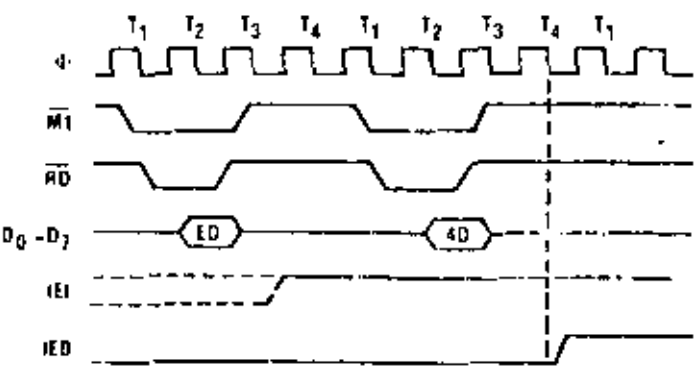


Figure 14d. Return from Interrupt Cycle

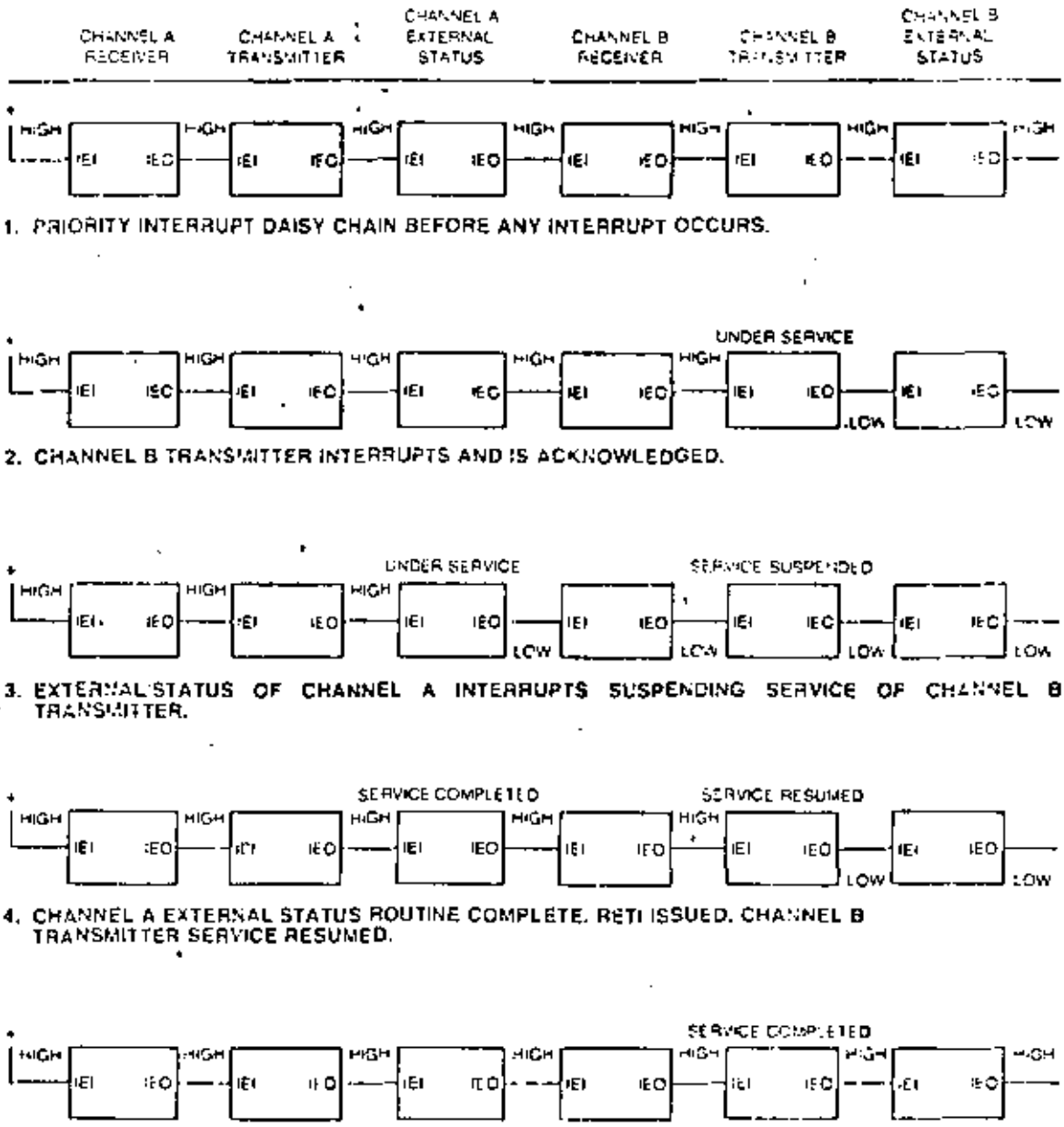
The ripple time of the interrupt daisy chain (both the High-to-Low and the Low-to-High transitions) limits the number of devices that can be placed in the daisy chain. Ripple time can be improved with carry-look-ahead, or by extending the interrupt acknowledge cycle. For further information about techniques for increasing the number of daisy-chained devices, refer to Zilog Application Note 03-0041-01 (*The Z80 Family Program Interrupt Structure*).

DAISY CHAIN INTERRUPT NESTING

Figure 15 illustrates the daisy chain configuration of interrupt circuits and their behavior with nested interrupts (an interrupt that is interrupted by another with a higher priority).

Each box in the illustration could be a separate external Z80 peripheral circuit with a user-defined order of interrupt priorities. However, a similar daisy chain structure also exists inside the Z80-SIO, which has six interrupt levels with a fixed order of priorities.

The case illustrated occurs when the transmitter of Channel B interrupts and is granted service. While this interrupt is being serviced, it is interrupted by a higher priority interrupt from Channel A. The second interrupt is serviced and—upon completion of a RETI instruction is executed or a RETI command is written into the Z80-SIO, resetting the interrupt-under-service latch of the Channel A interrupt. At this time, the service routine for Channel B is resumed. When it is completed, another RETI instruction is executed to complete the interrupt service.



1. PRIORITY INTERRUPT DAISY CHAIN BEFORE ANY INTERRUPT OCCURS.

2. CHANNEL B TRANSMITTER INTERRUPTS AND IS ACKNOWLEDGED.

3. EXTERNAL/STATUS OF CHANNEL A INTERRUPTS SUSPENDING SERVICE OF CHANNEL B TRANSMITTER.

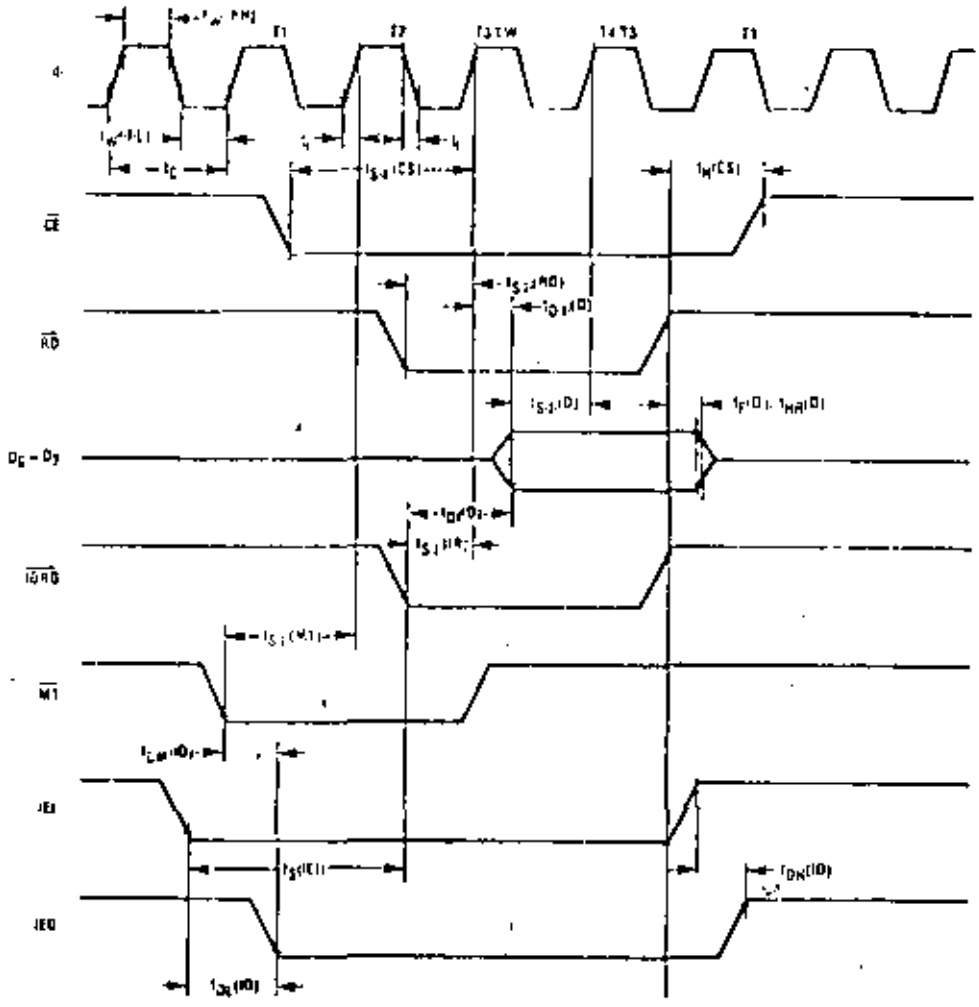
4. CHANNEL A EXTERNAL STATUS ROUTINE COMPLETE. RETI ISSUED. CHANNEL B TRANSMITTER SERVICE RESUMED.

5. CHANNEL B TRANSMITTER SERVICE ROUTINE COMPLETE. SECOND RETI ISSUED.

Figure 15. Typical Interrupt Sequence

AC Characteristics

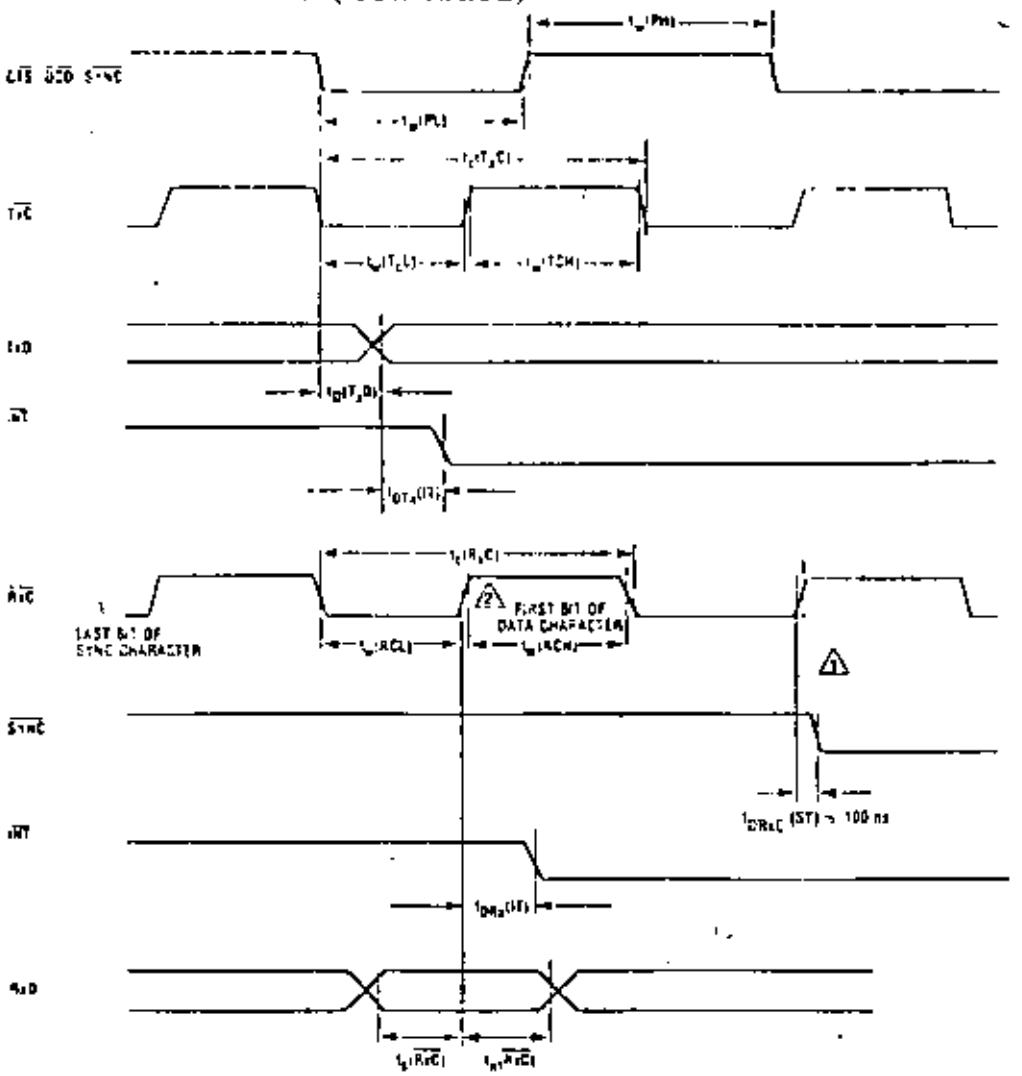
$T_A = 0°C$ to $70°C$, $V_{CC} = +5V, \pm 5\%$



Signal	Symbol	Parameter	Z80-SIO Min	Z80-SIO Max	Z80A SIO Min	Z80A SIO Max	Unit
ϕ	$t_{c}(\phi)$	Clock Period	400	4000	250	4000	ns
	$t_{w}(\phi H)$	Clock Pulse Width, clock HIGH	170	2000	105	2000	ns
	$t_{w}(\phi L)$	Clock Pulse Width, clock LOW	170	2000	105	2000	ns
	t_r, t_f	Clock Rise and Fall Times	0	30	0	30	ns
	t_s	Any Unspecified Read Time for setup times specified below	0		0		ns
$\overline{CE}$, $\overline{B/A}$, $\overline{C/D}$, $\overline{IORQ}$, $\overline{RD}$	$t_{s}(CS)$	Control Signal Setup Time to rising edge of ϕ during Read or Write Cycle	160		145		ns
D_0-D_7	$t_{d}(D)$	Data Output Delay from rising edge of ϕ during Read Cycle		240		220	ns
	$t_{s}(D)$	Data Setup Time to rising edge of ϕ during Write or M1 Cycle	50		50		ns
	$t_{d}(D)$	Data Output Delay from falling edge of $\overline{IORQ}$ during INTA Cycle		340		150	ns
	$t_{z}(D)$	Delay to Floating Bus (output buffer disable time)		230		110	ns
IEI	$t_{s}(IEI)$	IEI Setup Time to falling edge of $\overline{IORQ}$ during INTA Cycle	200		140		ns
IEO	$t_{d}(IO)$	IEO Delay Time from rising edge of IEI (after ED decode)		150		100	ns
	$t_{d}(IO)$	IEO Delay Time from falling edge of IEI		150		100	ns
	$t_{rs}(IO)$	IEO Delay Time from falling edge of $\overline{M1}$ (interrupt occurring just prior to M1)		300		190	ns
M1	$t_{s}(M1)$	$\overline{M1}$ Setup Time to rising edge of ϕ during INTA or M1 Cycle	210		90		ns
$\overline{RD}$	$t_{s}(RD)$	$\overline{RD}$ Setup Time to rising edge of ϕ during Read or M1 Cycle	240		115		ns

If WAIT from the SIO is to be used, $\overline{CE}$, $\overline{IORQ}$, $C/\overline{D}$ and $\overline{M1}$ must be valid for as long as the Wait condition is to persist.

AC Characteristics (Continued)



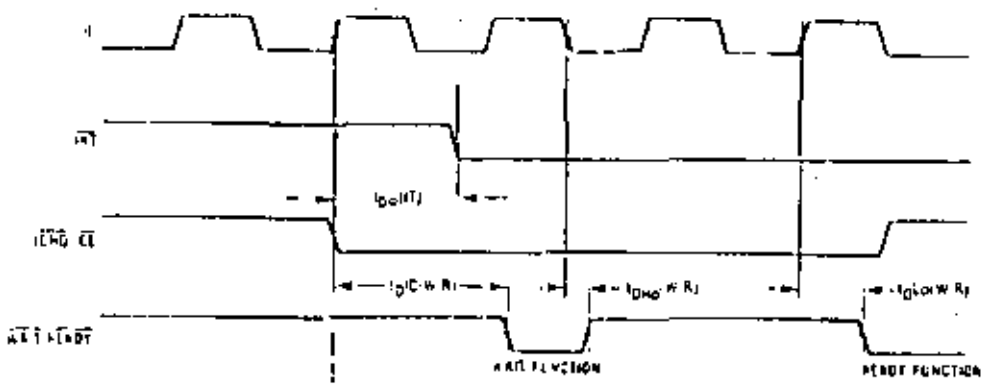
NOTES:

1. The SYNC input must be driven low on the rising edge of RxC delayed two complete clock cycles from the last bit of the sync character.
2. Data character assembly begins on the next Receive Clock cycle after the last bit of the sync character is received.

Signal	Symbol	Parameter	Z80-SIO Min	Z80-SIO Max	Z80A SIO Min	Z80A SIO Max	Unit
INT	$t_{dx}(IT)$	INT Delay Time from rising edge of RxC	10	13	10	13	φ periods
	$t_{dm}(IT)$	INT Delay Time from transition of latest Data Bit	5	9	5	9	φ periods
CTSA CTSB DCDA DCDB SYNCA SYNCB	$t_w(PH)$	Minimum HIGH Pulse Width for latching state into register and generating interrupt	200		200		ns
	$t_w(PL)$	Minimum LOW Pulse Width for latching state into register and generating interrupt	200		200		ns
SYNCA SYNCB	$t_{dx}(SY)$	Sync Pulse Delay Time from rising edge of RxC. Output Modes	4	7	4	7	φ periods
	$t_{dxe}(SY)$	Sync Pulse Delay Time from rising edge of RxC External Sync Mode		100		100	ns
TxCA TxCB	$t_c(TxC)$	Transmit Clock Period	400	*	400	*	ns
	$t_w(TCH)$	Transmit Clock Pulse Width, clock HIGH	180	*	180	*	ns
	$t_w(TCL)$	Transmit Clock Pulse Width, clock LOW	180	*	180	*	ns
TxDA TxDB*	$t_d(TxD)$	TxD Output Delay from falling Edge of TxC (x1 Clock Mode)		400		300	ns
RxCA RxCB	$t_c(RxC)$	Receive Clock Period	400	*	400	*	ns
	$t_w(RCH)$	Receive Clock Pulse Width, clock HIGH	180	*	180	*	ns
	$t_w(RCL)$	Receive Clock Pulse Width, clock LOW	180	*	180	*	ns
RxDA RxDB*	$t_s(RxC)$	Setup Time to rising edge of RxC, x1 mode	0		0		ns
	$t_h(RxC)$	Hold Time from rising edge of RxC, x1 mode	140		140		ns

*In all modes, the system clock (φ) rate must be at least 4.5 times the maximum data rate. RESET must be active a minimum of one complete φ cycle.

AC Characteristics (Continued)



Signal	Symbol	Parameter	Z80 SIO Min	Z80 SIO Max	Z80A SIO Min	Z80A SIO Max	Unit
INT	$t_{pd(INT)}$	INT Delay Time from rising edge of ϕ		200		200	ns
	$t_{pIC(W-RI)}$	Wait/READY Delay Time from $\overline{IORQ}$ or $\overline{CE}$ in Wait Mode		180		130	ns
	$t_{pDRQ(W-RI)}$	Wait/READY Delay Time from rising edge of ϕ Wait/READY HIGH Wait Mode		150		130	ns
WAIT/READY	$t_{pRc(W-RI)}$	Wait/READY Delay Time from rising edge of $\overline{RxC}$ Dual Bit Ready Mode	10	13	10	13	ϕ periods
	$t_{pTx(W-RI)}$	Wait/READY Delay Time from Interval Transmit Dual Bit Ready Mode	5	9	5	9	ϕ periods
	$t_{pLo(W-RI)}$	Wait/READY Delay Time from rising edge of ϕ Wait/READY LOW Ready Mode		120		120	ns

DC Characteristics

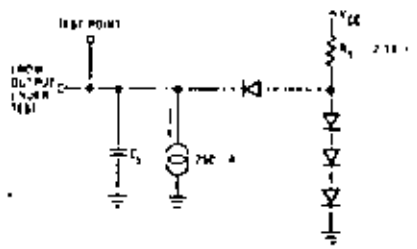
$T_A = 0°C$ to $70°C$, $V_{CC} = +5V, \pm 5\%$

Symbol	Parameter	Min.	Max.	Unit	Test Condition
V_{ILC}	Clock Input Low Voltage	−0.3	+0.45	V	
V_{IHC}	Clock Input High Voltage	$V_{CC}-0.6$	+5.5	V	
V_{IL}	Input Low Voltage	−0.3	+0.8	V	
V_{IH}	Input High Voltage	+2.0	+5.5	V	
V_{OL}	Output Low Voltage		+0.4	V	$I_{OL} = 2.0$ mA
V_{OH}	Output High Voltage	+2.4		V	$I_{OH} = -250\ \mu A$
I_{LI}	Input Leakage Current	−10	+10	μA	$0 \le V_{IN} \le V_{CC}$
I_{LD}	3-State Output/Data Bus Input Leakage Current	−10	+10	μA	$0 \le V_{IN} \le V_{CC}$
I_{LSY}	$\overline{SYNC}$ Pin Leakage Current	−40	−10	μA	
I_{CC}	Power Supply Current		100	mA	

Capacitance

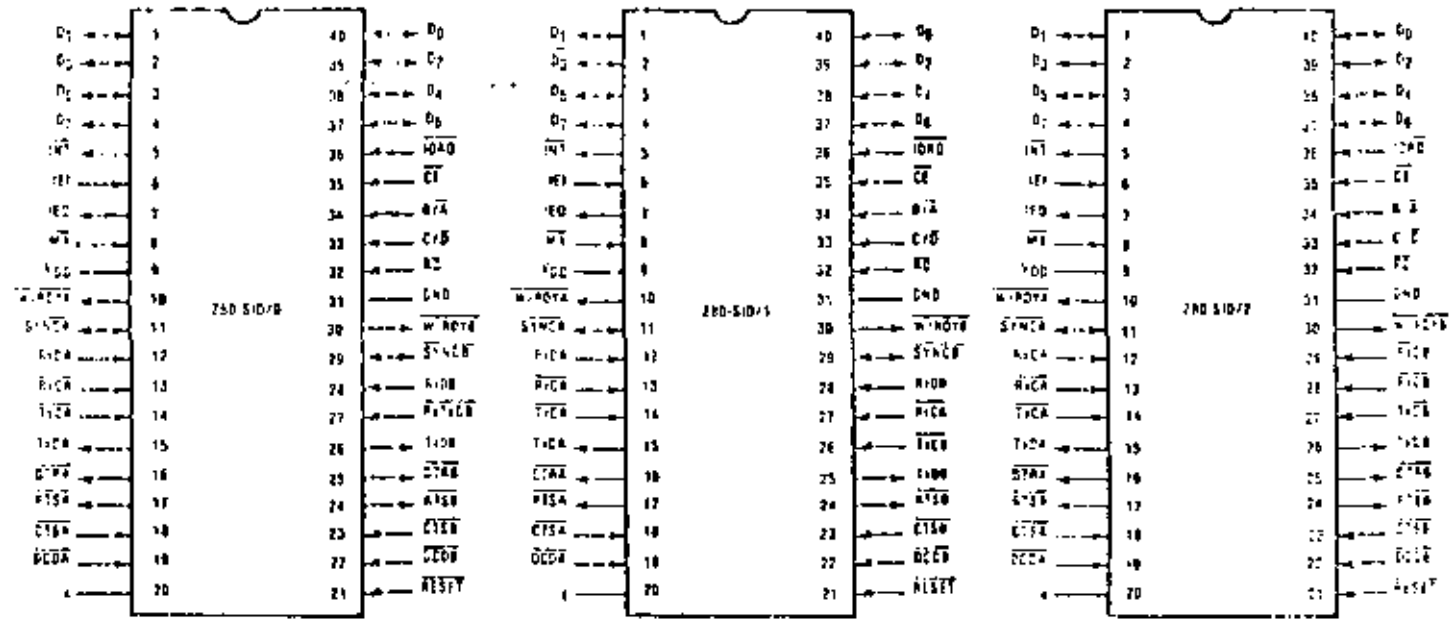
$T_A = 25°C$, $f = 1$ MHz

Symbol	Parameter	Min.	Max.	Unit	Test Condition
C_ϕ	Clock Capacitance		40	pF	Unmeasured pins returned to ground
C_{IN}	Input Capacitance		5	pF	
C_{OUT}	Output Capacitance		10	pF	

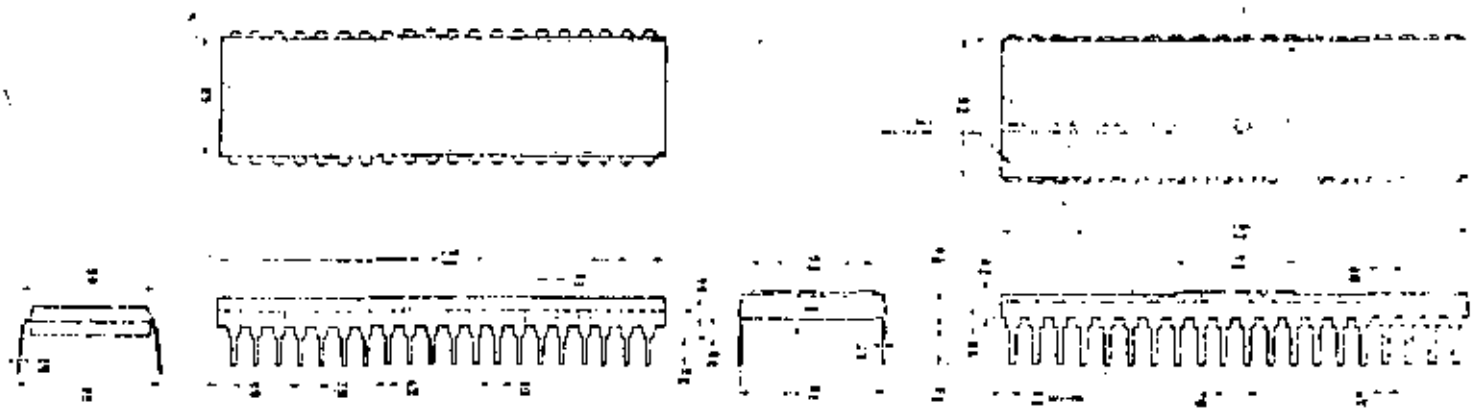


$C_L = 50$ pF. Increase delay by 10 ns for each 50 pF increase in C_L, up to 200 pF maximum.

Package Configurations



Package Outlines



40-Pin Plastic

40-Pin Ceramic

Ordering Information

C — Ceramic
P — Plastic
S — Standard 5V ±5%, 0° to 70°C
E — Extended 5V ±5%, −40° to 85°C
M — Military 5V ±10%, −55° to 125°C
/0 — Type 0 Bonding
/1 — Type 1 Bonding
/2 — Type 2 Bonding

Example:

Z80-SIO/1 CS (Ceramic—Standard Range—Type 1 Bonding)

Z80-SIO/0 PS (Plastic — Standard Range — Type 0 Bonding)

READER'S COMMENTS

Your feedback about this document is important to us: only in this way can we ascertain your needs and fulfill them in the future. Please take the time to fill out this questionnaire and return it to us. This information will be helpful to us, and, in time, to the future users of Zilog systems. Thank you.

Your Name: _____

Company Name: _____

Address: _____

Title of this document: _____

What software products do you have? _____

_____

_____

What is your hardware configuration (including memory size)? _____

_____

_____

Does this publication meet your needs? ☐Yes ☐No

If not, why not? _____

_____

_____

How do you use this publication? (Check all that apply)

- ☐ As an introduction to the subject?
- ☐ As a reference manual?
- ☐ As an instructor or student?

How do you find the material?

	Excellent	Good	Poor
Technicality	☐	☐	☐
Organization	☐	☐	☐
Completeness	☐	☐	☐

What would have improved the material? _____

_____

_____

Other comments, suggestions or corrections: _____

_____

_____

If you found any mistakes in this document, please let us know what and where they were:

_____

_____

First Class

Permit No. 475
Cupertino
California
95014

Business Reply Mail

No Postage Necessary if Mailed in the United States

Postage Will Be Paid By

Zilog
Software Department Librarian
10460 Bubb Road
Cupertino, California 95014

DIRECTORIO DE ASISTENTES AL CURSO: INTRODUCCION A LOS MICROPROCESADORES Y SUS APLICACIONES (DEL 10 DE AGOSTO AL 8 DE SEPTIEMBRE DE 1979)

	NOMBRE Y DIRECCION	EMPRESA Y DIRECCION
1.	ARMANDO AGUIRRE GARCIA Carmen No. 52-A Col. Nativitas México 13, D.F. Tel.	S. A. R. H. Sierra Gorda No. 23 Col. Lomas Tecamachalco México 10, D.F. Tel.540-00-52
2.	ING. VICTOR MANUEL ALVAREZ A. B. De Sahagun No. 27-D Col. Guerrero México 3, D.F. Tel. 535-27-89	S. C. T. Av. Xola y Universidad Ed. "H" 5o. Piso Col. Narvarte México 12, D.F. Tel. 530-64-58
3.	VICTOR GABRIEL BARBOSA GARCIA Martín Castrejón No. 72 Col. Felicitas del Río Morelia, Mich. Tel.	ESCUELA DE INGENIERIA ELECTRICA U.M.S.N.H. Ciudad Universitaria
4.	ING. JAVIER CABRERA VAZQUEZ Gildardo Aviles No. 119 Col. Rafael Lucio Xalapa, Ver. Tel. 7 66 33	FACULTAD DE INGENIERIA UNIVERSIDAD VERACRUZANA Calle de la Pergola S/N Col. Lomas del Estadio Xalapa, Ver. Tel. 7 66 33
5.	HECTOR CALVARIO MARTINEZ Rivera No. 83 Col. Aguilas México 20, D.F. Tel. 593-03-58	CENTRO DE CALCULO FAC. ING. Ciudad Universitaria
6.	ADRIAN CARVAJAL VIZCARRA Medellin No. 385-4 Col. Del Valle México 7, D.F. Tel. 543-23-02 584-77-60	S. A. H. O. P. Reforma No. 77-10 Piso

DIRECTORIO DE ASISTENTES AL CURSO: INTRODUCCION A LOS MICROPROCESADORES Y SUS APLICACIONES (DEL 10 DE AGOSTO AL 8 DE SEPTIEMBRE DE 1979)

	NOMBRE Y DIRECCION	EMPRESA Y DIRECCION
7.	ING. JOSE MANUEL CASTELLANOS ALVAREZ División del Norte Andador 6-3 Col. Villa Coapa México 22, D.F. Tel. 671-01-27	COMISION FEDERAL DE ELECTRICIDAD San Rafael Sta. Cecilia No. 211 Col. Tlalnepantla Edo. de México Tel.
8.	JAIME ARAU CHAVARRIA Tantoyuca No. 38 Xalapa, Ver. Tel. 7 69 40 Ext. 101	COMISION FEDERAL DE ELECTRICIDAD Xalapa, Ver.
9.	ING. ERNESTO CISNEROS PAEZ Av. Veracruz No. 184 Col. Cuajimalpa México 18, D.F. Tel. 2-04-83	FISHER GOVERNOR Hacienda de la Guaracha No. 127 Col. Bosques de Echegaray Edo. de Méx. Tel. 3-73-06-33
10.	ARAMANDO CORONA VAZQUEZ Herrera Tejada No. 87 Col. Federal Jalapa, Ver. Tel. 7 68 91	COMISION FEDERAL DE ELECTRICIDAD Allende No. 155 Col. Jalapa Jalapa, Ver. Tel. 7 22 39
11.	JESUS DAVID CRUZ CERVANTES Guadalcanal No. 12 Col. Euzkadi México 15, D.F. Tel. 355-28-37	S. A. R. H. Gomez Farías No. 2-1er. Piso Col. San Rafael México, D.F. Tel. 535-41-20
12.	SERGIO M. ESPINOSA DE LOS MONETROS G. Extremadura No. 158-1 Col. Mixcoac México 19, D.F. Tel. 563-56-61	S. A. R. H. Gómez Farías No. 2-3o. Piso Col. Tabacalera México 4, D.F. Tel. 535-67-28 Y 29

DIRECTORIO DE ASISTENTES AL CURSO: INTRODUCCION A LOS MICROPORCESADORES Y SUS APLICACIONES (DEL 10 DE AGOSTO AL 8 DE SEPTIEMBRE DE 1979)

	NOMBRE Y DIRECCION	EMPRESA Y DIRECCION
13.	ING. RICARDO ESPRIELLA GODINEZ Calle "Z" No. 26-12 Col. Fovissste México 21, D.F. Tel. 677-96-61	FACULTAD DE INGENIERIA UNAM Ciudad Universitaria Tel. 559-52-15 Ext. 3748
14.	RODOLFO A. FLORES MARQUEZ	COMISION FEDERAL DE ELECTRICIDAD
15.	LUIS H. FRANCO CARDENAS Av. UniversidadNo. 465-4 Col. Del Valle México 16, D.F. Tel. 523-49-36	B A N R U R A L Campeche No. 290-7o. Piso Col. Roma México 11, D.F. Tel.
16.	UBALDO GALLEGOS SANDOVAL Anaxagoras No. 17-304 Col. Narvarte México 12, D.F. Tel.	S. A. R. H. Gómez Farias No. 2-2o. Piso Col. Tabacalera México 4, D.F. Tel. 546-67-86
17.	ING. ALFREDO GAMEZ LOPEZ Betancourt No. 67 Xalapa, Ver. Tel. 7-72-34	FACULTAD DE INGENIERIA UNIVERSIDAD VERACRUZANA Calle de la Pergola S/N Losmas Del Estadio Xalapa, Ver.
18.	ING. JAIME A. GARCIA CASTAÑEDA Cerro Gordo No. 150 Col. Campestre Churubusco México 21, D.F. Tel. 549-15-70	S. A. R. H. Sierra Gorda No. 23 Col. Lomas de Tecamachalco México 10, D.F. Tel. 540-00-52

DIRECTORIO DE ASISTENTES AL CURSO: INTRODUCCION A LOS MICROPROCESADORES Y SUS APLICACIONES (DEL 10 DE AGOSTO AL 8 DE SEPTIEMBRE DE 1979)

	NOMBRE Y DIRECCION	EMPRESA Y DIRECCION
19.	ING. LUIS GARCIA REYES Martín Castrejón No. 35 Col. Felicitas del Río Morelia, Mich. Tel. 2-69-25	ESCUELA DE INGENIERIA ELECTRICA Ciudad Universitaria Morelia, Mich. Tel. 2-77-76
20	ING. ARTURO GUILLERMO GOMEZ PEREZ Cerro de Macuiltepec No. 196 Col. Campestre Churubusco México 21, D.F. Tel. 544-62-58	HUGIN DE MEXICO, S.A. Bahía de Todos Santos 149 Col. Verónica Anzures México 17, D.F. Tel. 545-66-15 Ext. 26
21.	JOSE ANTONIO GONZALEZ CALCANEO Gómez Farias No. 2-4o. Piso Col. San Rafael México 4, D.F. Tel. 546-67-78	S.A.R.H. Gómez Farias No2-4o. Piso
22.	ALFREDO GRANADOS SALAZAR Abundio Martinez No. 249 Col. Vallejo México 14, D.F. Tel.	INSTITUTO MEXICANO DEL PETROLEO Av. De los Cien Metros No. 152 Col. Ind. Vallejo México 14, D.F. Tel. 567-54-76
23.	ING. LUIS GUERRA AVILA Allende No. 217 Col. Clavería México 16, D.F. Tel.	P E M E X Marina Nal. No. 329 Col. Verónica Anzures México 17, D.F. Tel. 545-74-60 Ext. 3365
24.	LUIS HERMAN ARIZCORRETA Río Neva NO. 32 Col. Cuauhtémoc México 5, D.F. Tel. 535-08-07	E S I M E CULHUACAN Av. Santa Ana No. 1000 Col. Sn. Fco. Culhuacan México 13, D.F. Tel. 581-85-90

DIRECTORIO DE ASISTENTES AL CURSO: INTRODUCCION A LOS MICROPROCESADORES Y SUS APLICACIONES (DEL 10 DE AGOSTO AL 8 DE SEPTIEMBRE DE 1979)

	NOMBRE Y DIRECCION	EMPRESA Y DIRECCION
25.	ING. FELIPE LOZA N. Amates No. 7 J. Sn. Mateo Col. Colina Naucalpan Edo. de México Tel. 373-53-14	P E M E X Marina Nacional No. 329
26.	LAURA MARTIN MUNGUIA Codorníz No. 25 Col. Las Alamedas Edo. de Méx. Tel. 2 17-63	FISHER GOVERNOR DE MEXICO,S.A. Hacienda de la Guaracha No. 127 Col. Bosques de Echegaray Naucalpan Tel. 3 73-06-33
27.	ING. JOSE MA. MENDOZA GUTIERREZ	FACULTAD DE INGENIERIA UNAM
28.	ING. CARLOS MORENO PEREZ Zaragoza No. 234 Dpto. 31 Col. Guerrero México 3, D.F. Tel. 526-26-91	INSTITUTO NACIONAL DE INVESTI GACIONES NUCLEARES ININ Benjamín Franklin No. 161 Col. Tacubaya México 18, D.F. Tel. 271-31-46 271-31-85
29.	ALFONSO MUÑOZ KUREJWOWKI Isabel la Catolica No. 457 Col. Algarín México 8, D.F. Tel. 530-77-41	S. A. R. H. Gómez Farías No. 2-1o. Piso Col. Tabacalera México 4, D.F. Tel. 535-41-20
30.	RUBEN EUGENIO MUÑOZ VARGAS Av. Palomas No. 58 Col. Mayorazgos del Bosque Atizapan Edo. de Méx. Tel.	CIA. DE LUZ Y FUERZA DEL CENTRO Av. Playa Pie de la Cuesta 273 Col. Sn. Andrés Tetepilco México 13, D.F. Tel. 539-96-10

DIRECTORIO DE ASISTENTES AL CURSO: INTRODUCCION A LOS MICROPROCESADORES Y SUS APLICACIONES (DEL 10 DE AGOSTO AL 8 DE SEPTIEMBRE DE 1979)

	NOMBRE Y DIRECCION	EMPRESA Y DIRECCION
31.	MARCO A. MURRAY LASSO Rembrandt No. 53 Col. Mixcoac México 19, D.F. Tel. 563-37-40	DEPFI, UNAM Ciudad Universitaria Tel. 548-65-00
32.	ING. JOSE GASTON MOYOLA DEL RIO Tulipán No. 137 Col. Cd. Jardín Tel. 544-03-32	S. A. H. O. P. Culiacán No. 132 Col. Roma Sur México 7, D.F. Tel. 564-94-39
33.	ING. JESUS NUÑEZ VALADEZ Central de talaberteros No. 112 Col. Emilio Carranza México 2, D.F. Tel. 529-36-68	ENEP ARAGON Av. Central y Calle S/N Sn. Juan de Aragón Netzahualcoyotl Tel.
34.	RAUL E. OCHOA ESCOBAR Búfalo No. 82 Col. Del Valle México 12, D.F. Tel. 524-12-99	FACULTAD DE INGENIERIA UNAM Ciudad Universitaria Tel. 550-00-40
35.	ING. GUSTAVO G. ORIGEL COUTIÑO Gemelos No. 119 Col. Prado Chur. México 13, D.F. Tel. 582-32-71	FACULTAD DE INGENIERIA UNAM Ciudad Universitaria Tel.
36.	LUIS OROZCO RAMIREZ Tlaxcaltecas No. 70 Col. La Raza México 15, D.F. Tel. 597-54-31	INSTITUTO MEXICANO DEL PETROLEO Av. de los 100 Metros No. 152 Col. Lindavista México 14, D.F. Tel. 567-54-76

DIRECTORIO DE ASISTENTES AL CURSO: INTRODUCCION A LOS MICROPROCESADORES Y SUS APLICACIONES (DEL 10 DE AGOSTO AL 8 DE SEPTIEMBRE DE 1979)

	NOMBRE Y DIRECCION	EMPRESA Y DIRECCION
37.	ING. GMO. JOSE LUIS PERALTA SOLORIO Cali No. 323-A Col. Valle Dorado Tlalnepantla Edo. de Méx. Tel. 379-34-16	COMISION FEDERAL DE ELECTRICIDAD Av. Toluca y Don Manuelito Col. Olivar de los Padres México 20, D.F. Tel. 595-54-00
38.	ING. J. CRUZ ROBLES GALVEZ Hda. Sn. Diego de los Padres No. 55 Col. Frac. Sta. Elena Sn. Mateo Atenco Tel. 570-21-22 Ext. 136	INSTITUTO NACIONAL DE INVESTI GACIONES NUCLEARES Benjamín Franklin No. 161 Col. Tacubaya México 18, D.F. Tel. 271-31-85
39.	ROSA ELVIA SALAZAR PEREZ Lerdo No. 304 Edif. I. Zaragoza A-104 Col. Guerrero México 3, D.F. Tel. 583-92-36	CIA. DE LUZ Y FUERZA DEL CENTRO Melchor Ocampo No. 171 Col. Anáhuac México 17, D.F. Tel. 535-75-25
40.	ING. EDMUNDO SALINAS BOHORQUEZ Carlos Pereira No. 10-B Col. Viaducto Piedad México 13, D.F. Tel. 538-07-28	INSTITUTO NACIONAL DE INVESTI GACIONES NUCLEARES Benjamín Franklin No. 161 Col. Tacubaya México 18, D.F. Tel. 271-31-46 271-31-85
41.	ING. EDUARDO SANCHEZ MARTINEZ Cerrada Casa Hogerr No.14 Col. Lomas Hidalgo Morelia, Mich. Tel. 289-63	UNIVERSIDAD MICHOACANA ESCUELA DE INGENIERIA ELECTRICA Ciudad Universitaria Morelia, Mich. Tel. 2 76-77
42.	ING. HOMERO SANCHEZ ZORRILLA Acacias No. 9 Dpto. "F" Col. Sta. Ma. La Rivera México 4, D.F. Tel. 547-39-51	S. A. R. H. Ignacio Ramirez No. 20-4o. Piso Col. San Rafael México 4, D.F. Tel. 566-36-25

DIRECTORIO DE ASISTENTES AL CURSO: INTRODUCCION A LOS MICROPROCESADORES Y SUS APLICACIONES (DEL 10 DE AGOSTO AL 8 DE SEPTIEMBRE DE 1979)

	NOMBRE Y DIRECCION	EMPRESA Y DIRECCION
43.	ING. ALEJANDRO SERRANO FELIX Toribio Medina No. 108 Col. Algarín México 8, D.F. Tel. 530-53-03	FACULTAD DE INGENIERIA UNAM. Ciudad Universitaria
44.	PEDRO DE JESUS TOLEDO ECHEGARAY Insurgentes Sur No. 4411 Edif. 21-401 Col. Tlalcoligia México 22, D.F. Tel. 573-51-81	S. A. H. O. P. Reforma No. 77-10 Piso México, D.F. Tel. 591-18-69
45.	ING. FRANCISCO TORRES Presa No. 85 Col. San Jerónimo México 20, D.F. Tel. 595-39-15	COMISION FEDERAL DE ELECTRICIDAD Av. Sn Rafael Sta. Cecilia No. 211 Col. San Rafael Tlalnepantla Edo. de Méx. Tel. 565-36-18
46.	ING. ROBERTO VALENZUELA CEPEDA Av. del Convento No. 13 Col. Churubusco México 21, D.F. Tel. 549-29-36	S. A. R. H. Ignacio Ramírez No. 20 Col. Tabacalera México 4, D.F. Tel. 566-38-48
47.	ING. ALFONSO VAZQUEZ MUÑOZ Melchor Ocampo No. 386 Col. Cuauhtémoc México 5, D.F. Tel. 514-08-33	COMISION FEDERAL DE ELECTRICIDAD Rodano 14 Mezanínne Col. Cuauhtémoc México 5, D.F. Tel. 553-71-33 Ext. 2257
48.	ING. ANGEL YAÑEZ HERNANDEZ Valle del Mexquital No. 168 Col. Valle de Aragón Edo. de Méx. Tel. 566-38-48	S. A. R. H. Ignacio Ramírez No. 20-4o. Piso Col. San Rafael México 4, D.F. Tel. 566-38-48

DIRECTORIO DE ASISTENTES AL CURSO: INTRODUCCION A LOS MICROPROCESADORES Y SUS APLICACIONES (DEL 10 DE AGOSTO AL 8 DE SEPTIEMBRE DE 1979)

	NOMBRE Y DIRECCION	EMPRESA Y DIRECCION
49.	ENRIQUE ZARATE LEYVA Reforma No. 69-9o. Piso Col. Centro México 1, D.F. Tel.	S. A. R. H. Reforma No. 69-9o. Piso Col. Centro México 1, D.F. Tel. 591-09-84
50.	ROBERTO ZENDEJAS MORALES Calz. de la Viga No. 1416 Depto. E-108 Col. Sifón México 8, D.F. Tel.	S. A. R. H. Plaza de la República No. 31-6o. Col. Centro México 1, D.F. Tel. 592-41-82
51.	ING. JESUS IBARRA VELAZQUEZ Gral. Gpe. Victoria No. 150 Col. G. A. Madero México 14, D.F. Tel. 577-26-27	S. A. R. H. Plaza de la República esq. Ponciano Arriaga 6o. Piso México 1, D.F. Tel. 546-50-96