

MCS-85™ User's Manual

Table of Contents

CHAPTER 1	
Part 1: Introduction to the Functions of a Computer	1-1
Part 2: Introduction to MCS-85™	1-6
CHAPTER 2	
Functional Description	2-1
CHAPTER 3	
System Operation and Interfacing	3-1
CHAPTER 4	
The 8080 Central Processor Unit	4-1
CHAPTER 5	
The Instruction Set	5-1
Instruction Set Index	5-19
CHAPTER 6	
DEVICE SPECIFICATIONS	
MCS-85	
8085A/8085A-2 8-Bit Microprocessor	6-1
8155/8156/8155-2/8156-2 RAM/IO/Counter-Timer	6-17
8185/8185-2 RAM	6-32
8355/8355-2 ROM/IO	6-37
8755A/8755A-2 EPROM/IO	6-45
MCS-80	
8080A/8080A-1/8080A-2 8-Bit Microprocessor	6-56
8224 Clock Generator and Driver	6-64
8228/8238 System Controller and Bus Driver	6-68
8801 Clock Generator Crystal	6-72
System Support Components	
8205 High-Speed 1 Out of 8 Binary Decoder	6-74
8212 8-Bit Input/Output Port	6-80
8218/8219 Bi-polar Microcomputer Bus Controllers	6-90
8237/8237-2 High Performance Programmable DMA Controller	6-101
8257/8257-5 Programmable DMA Controller	6-115
8259A/8259A-2/8259A-8 Programmable Interrupt Controller	6-132
8282/8283 Octal Latch	6-150
8286/8287 Octal Bus Transceiver	6-154

Table of Contents (Continued)

Peripherals*

8041A/8641A/8741A Universal Peripheral Interface	6-158
8202 Dynamic RAM Controller	6-159
8251A Programmable Communication Interface	6-160
8253/8253-5 Programmable Interval Timer	6-161
8255A/8255A-5 Programmable Peripheral Interface	6-162
8271 Programmable Floppy Disk Controller	6-163
8272 Single/Double Density Floppy Disk Controller	6-164
8273 Programmable HDLC/SDLC Protocol Controller	6-165
8275 Programmable CRT Controller	6-166
8279/8279-5 Programmable Keyboard/Display Interface	6-167
8291 GPIB Talker/Listener	6-168
8292 GPIB Controller	6-169
8293 GPIB Transceiver	6-170
8294 Data Encryption Unit	6-171
8295 Dot Matrix Printer Controller	6-172

Static RAMs

2114A 1024 × 4 Bit Static RAM	6-173
2141 4096 × 1 Bit Static RAM	6-174
2142 1024 × 4 Bit Static RAM	6-175
2148 1024 × 4 Bit Static RAM	6-176

ROMs/EPROMs

2716 16K (2K × 8) UV Erasable PROM	6-177
2732 32K (4K × 8) UV Erasable PROM	6-178
2758 8K (1K × 8) UV Erasable Low Power PROM	6-182
3604A, 3624A 4K (512 × 8) High-Speed PROM	6-183
3605A, 3625A 4K (1K × 4) PROM	6-184
3628 8K (1K × 8) Bipolar PROM	6-187
3636 16K (2K × 8) Bipolar PROM	6-190

CHAPTER 7

DEVELOPMENT AIDS

ICE-80™ MCS-80™ In-Circuit Emulator	7-1
UPP-103 Universal PROM Programmer	7-7
Intellec® Microcomputer Development System Model 220	7-9
Intellec® Microcomputer Development System Model 230	7-13
ISIS-II Diskette Operating System	7-17
PL/M-80 High Level Programming Language Inteltec® Resident Computer	7-20
ICE-85™ MCS-85™ In-Circuit Emulator	7-23
SDK-85 MCS-85™ System Design Kit	7-27
Fortran-80 8080/8085 ANS Fortran 77 Inteltec® Resident Compiler	7-33
Basic-80 Extended ANS 1978 Basic Inteltec® Resident Interpreter	7-37
ICIS-Cobol™ Software Package	7-40

APPENDIX

Applications of MCS-85™	A1-1
-------------------------------	------

*This section contains partial data sheets. For complete details, refer to the Intel Peripheral Design Handbook.

Chapter 1 Introduction

MOS

CO

MOS

CO



CHAPTER 1

PART 1: INTRODUCTION TO THE FUNCTIONS OF A COMPUTER

This chapter introduces certain basic computer concepts. It provides background information and definitions which will be useful in later chapters of this manual. Those already familiar with computers may skip this material, at their option.

A TYPICAL COMPUTER SYSTEM

A typical digital computer consists of:

- a) A central processor unit (CPU)
- b) A memory
- c) Input/output (I/O) ports

The memory serves as a place to store **Instructions**, the coded pieces of information that direct the activities of the CPU, and **Data**, the coded pieces of information that are processed by the CPU. A group of logically related instructions stored in memory is referred to as a **Program**. The CPU "reads" each instruction from memory in a logically determined sequence, and uses it to initiate processing actions. If the program sequence is coherent and logical, processing the program will produce intelligible and useful results.

The memory is also used to store the data to be manipulated, as well as the instructions that direct that manipulation. The program must be organized such that the CPU does not read a non-instruction word when it expects to see an instruction. The CPU can rapidly access any data stored in memory; but often the memory is not large enough to store the entire data bank required for a particular application. The problem can be resolved by providing the computer with one or more **Input Ports**. The CPU can address these ports and input the data contained there. The addition of input ports enables the computer to receive information from external equipment (such as a paper tape reader or floppy disk) at high rates of speed and in large volumes.

A computer also requires one or more **Output Ports** that permit the CPU to communicate the result of its processing to the outside world. The output may go to a display, for use by a human operator, to a peripheral device that produces "hard-copy," such as a line-printer, to a

peripheral storage device, such as a floppy disk unit, or the output may constitute process control signals that direct the operations of another system, such as an automated assembly line. Like input ports, output ports are addressable. The input and output ports together permit the processor to communicate with the outside world.

The CPU unifies the system. It controls the functions performed by the other components. The CPU must be able to fetch instructions from memory, decode their binary contents and execute them. It must also be able to reference memory and I/O ports as necessary in the execution of instructions. In addition, the CPU should be able to recognize and respond to certain external control signals, such as INTERRUPT and WAIT requests. The functional units within a CPU that enable it to perform these functions are described below.

THE ARCHITECTURE OF A CPU

A typical central processor unit (CPU) consists of the following interconnected functional units:

- Registers
- Arithmetic/Logic Unit (ALU)
- Control Circuitry

Registers are temporary storage units within the CPU. Some registers, such as the program counter and instruction register, have dedicated uses. Other registers, such as the accumulator, are for more general purpose use.

Accumulator:

The accumulator usually stores one of the operands to be manipulated by the ALU. A typical instruction might direct the ALU to add the contents of some other register to the contents of the accumulator and store the result in the accumulator itself. In general, the accumulator is both a source (operand) and a destination (result) register.

Often a CPU will include a number of additional general purpose registers that can be used to store operands or intermediate data. The availability of general purpose

registers eliminates the need to “shuffle” intermediate results back and forth between memory and the accumulator, thus improving processing speed and efficiency.

Program Counter (Jumps, Subroutines and the Stack):

The instructions that make up a program are stored in the system’s memory. The central processor references the contents of memory, in order to determine what action is appropriate. This means that the processor must know which location contains the next instruction.

Each of the locations in memory is numbered, to distinguish it from all other locations in memory. The number which identifies a memory location is called its **Address**.

The processor maintains a counter which contains the address of the next program instruction. This register is called the **Program Counter**. The processor updates the program counter by adding “1” to the counter each time it fetches an instruction, so that the program counter is always current (pointing to the next instruction).

The programmer therefore stores his instructions in numerically adjacent addresses, so that the lower addresses contain the first instructions to be executed and the higher addresses contain later instructions. The only time the programmer may violate this sequential rule is when an instruction in one section of memory is a **Jump** instruction to another section of memory.

A jump instruction contains the address of the instruction which is to follow it. The next instruction may be stored in any memory location, as long as the programmed jump specifies the correct address. During the execution of a jump instruction, the processor replaces the contents of its program counter with the address embodied in the Jump. Thus, the logical continuity of the program is maintained.

A special kind of program jump occurs when the stored program “Calls” a subroutine. In this kind of jump, the processor is required to “remember” the contents of the program counter at the time that the jump occurs. This enables the processor to resume execution of the main program when it is finished with the last instruction of the subroutine.

A **Subroutine** is a program within a program. Usually it is a general-purpose set of instructions that must be executed repeatedly in the course of a main program. Routines which calculate the square, the sine, or the logarithm of a program variable are good examples of functions often written as subroutines. Other examples might be programs designed for inputting or outputting data to a particular peripheral device.

The processor has a special way of handling subroutines, in order to insure an orderly return to the main program. When the processor receives a Call instruction, it increments the Program Counter and stores the counter’s contents in a reserved memory area known as the **Stack**. The Stack thus saves the address of the instruction to be executed after the subroutine is completed. Then the pro-

cessor loads the address specified in the Call into its Program Counter. The next instruction fetched will therefore be the first step of the subroutine.

The last instruction in any subroutine is a **Return**. Such an instruction need specify no address. When the processor fetches a Return instruction, it simply replaces the current contents of the Program Counter with the address on the top of the stack. This causes the processor to resume execution of the calling program at the point immediately following the original Call Instruction.

Subroutines are often **Nested**; that is, one subroutine will sometimes call a second subroutine. The second may call a third, and so on. This is perfectly acceptable, as long as the processor has enough capacity to store the necessary return addresses, and the logical provision for doing so. In other words, the maximum depth of nesting is determined by the depth of the stack itself. If the stack has space for storing three return addresses, then three levels of subroutines may be accommodated.

Processors have different ways of maintaining stacks. Some have facilities for the storage of return addresses built into the processor itself. Other processors use a reserved area of external memory as the stack and simply maintain a **Pointer** register which contains the address of the most recent stack entry. The external stack allows virtually unlimited subroutine nesting. In addition, if the processor provides instructions that cause the contents of the accumulator and other general purpose registers to be “pushed” onto the stack or “popped” off the stack via the address stored in the stack pointer, multi-level interrupt processing (described later in this chapter) is possible. The status of the processor (i.e., the contents of all the registers) can be saved in the stack when an interrupt is accepted and then restored after the interrupt has been serviced. This ability to save the processor’s status at any given time is possible even if an interrupt service routine, itself, is interrupted.

Instruction Register and Decoder:

Every computer has a **Word Length** that is characteristic of that machine. A computer’s word length is usually determined by the size of its internal storage elements and interconnecting paths (referred to as **Busses**); for example, a computer whose registers and busses can store and transfer 8 bits of information has a characteristic word length of 8-bits and is referred to as an 8-bit parallel processor. An eight-bit parallel processor generally finds it most efficient to deal with eight-bit binary fields, and the memory associated with such a processor is therefore organized to store eight bits in each addressable memory location. Data and instructions are stored in memory as eight-bit binary numbers, or as numbers that are integral multiples of eight bits: 16 bits, 24 bits, and so on. This characteristic eight-bit field is often referred to as a **Byte**.

Each operation that the processor can perform is identified by a unique byte of data known as an **Instruction**

Code or Operation Code. An eight-bit word used as an instruction code can distinguish between 256 alternative actions, more than adequate for most processors.

The processor fetches an instruction in two distinct operations. First, the processor transmits the address in its Program Counter to the memory. Then the memory returns the addressed byte to the processor. The CPU stores this instruction byte in a register known as the **Instruction Register**, and uses it to direct activities during the remainder of the instruction execution.

The mechanism by which the processor translates an instruction code into specific processing actions requires more elaboration than we can here afford. The concept, however, should be intuitively clear to any logic designer. The eight bits stored in the instruction register can be decoded and used to selectively activate one of a number of output lines, in this case up to 256 lines. Each line represents a set of activities associated with execution of a particular instruction code. The enabled line can be combined with selected timing pulses, to develop electrical signals that can then be used to initiate specific actions. This translation of code into action is performed by the **Instruction Decoder** and by the associated control circuitry.

An eight-bit instruction code is often sufficient to specify a particular processing action. There are times, however, when execution of the instruction requires more information than eight bits can convey.

One example of this is when the instruction references a memory location. The basic instruction code identifies the operation to be performed, but cannot specify the object address as well. In a case like this, a two- or three-byte instruction must be used. Successive instruction bytes are stored in sequentially adjacent memory locations, and the processor performs two or three fetches in succession to obtain the full instruction. The first byte retrieved from memory is placed in the processor's instruction register, and subsequent bytes are placed in temporary storage; the processor then proceeds with the execution phase. Such an instruction is referred to as **Variable Length**.

Address Register(s):

A CPU may use a register or register-pair to hold the address of a memory location that is to be accessed for data. If the address register is **Programmable**, (i.e., if there are instructions that allow the programmer to alter the contents of the register) the program can "build" an address in the address register prior to executing a **Memory Reference** instruction (i.e., an instruction that reads data from memory, writes data to memory or operates on data stored in memory).

Arithmetic/Logic Unit (ALU):

All processors contain an arithmetic/logic unit, which is often referred to simply as the **ALU**. The ALU, as its name implies, is that portion of the CPU hardware which

performs the arithmetic and logical operations on the binary data.

The ALU must contain an **Adder** which is capable of combining the contents of two registers in accordance with the logic of binary arithmetic. This provision permits the processor to perform arithmetic manipulations on the data it obtains from memory and from its other inputs.

Using only the basic adder a capable programmer can write routines which will subtract, multiply and divide, giving the machine complete arithmetic capabilities. In practice, however, most ALUs provide other built-in functions, including hardware subtraction, boolean logic operations, and shift capabilities.

The ALU contains **Flag Bits** which specify certain conditions that arise in the course of arithmetic and logical manipulations. Flags typically include **Carry**, **Zero**, **Sign**, and **Parity**. It is possible to program jumps which are conditionally dependent on the status of one or more flags. Thus, for example, the program may be designed to jump to a special routine if the carry bit is set following an addition instruction.

Control Circuitry:

The control circuitry is the primary functional unit within a CPU. Using clock inputs, the control circuitry maintains the proper sequence of events required for any processing task. After an instruction is fetched and decoded, the control circuitry issues the appropriate signals (to units both internal and external to the CPU) for initiating the proper processing action. Often the control circuitry will be capable of responding to external signals, such as an interrupt or wait request. An **Interrupt** request will cause the control circuitry to temporarily interrupt main program execution, jump to a special routine to service the interrupting device, then automatically return to the main program. A **Wait** request is often issued by a memory or I/O element that operates slower than the CPU. The control circuitry will idle the CPU until the memory or I/O port is ready with the data.

COMPUTER OPERATIONS

There are certain operations that are basic to almost any computer. A sound understanding of these basic operations is a necessary prerequisite to examining the specific operations of a particular computer.

Timing:

The activities of the central processor are cyclical. The processor fetches an instruction, performs the operations required, fetches the next instruction, and so on. This orderly sequence of events requires precise timing, and the CPU therefore requires a free running oscillator clock which furnishes the reference for all processor actions. The combined fetch and execution of a single instruction is referred to as an **Instruction Cycle**. The portion of a cycle identified

with a clearly defined activity is called a **State**. And the interval between pulses of the timing oscillator is referred to as a **Clock Period**. As a general rule, one or more clock periods are necessary for the completion of a state, and there are several states in a cycle.

Instruction Fetch:

The first state(s) of any instruction cycle will be dedicated to fetching the next instruction. The CPU issues a read signal and the contents of the program counter are sent to memory, which responds by returning the next instruction word. The first byte of the instruction is placed in the instruction register. If the instruction consists of more than one byte, additional states are required to fetch each byte of the instruction. When the entire instruction is present in the CPU, the program counter is incremented (in preparation for the next instruction fetch) and the instruction is decoded. The operation specified in the instruction will be executed in the remaining states of the instruction cycle. The instruction may call for a memory read or write, an input or output and/or an internal CPU operation, such as a register-to-register transfer or an add-registers operation.

Memory Read:

An instruction **fetch** is merely a special memory read operation that brings the instruction to the CPU's instruction register. The instruction fetched may then call for data to be read from memory into the CPU. The CPU again issues a read signal and sends the proper memory address; memory responds by returning the requested word. The data received is placed in the accumulator or one of the other general purpose registers (not the instruction register).

Memory Write:

A memory write operation is similar to a read except for the direction of data flow. The CPU issues a write signal, sends the proper memory address, then sends the data word to be written into the addressed memory location.

Wait (memory synchronization):

As previously stated, the activities of the processor are timed by a master clock oscillator. The clock period determines the timing of all processing activity.

The speed of the processing cycle, however, is limited by the memory's **Access Time**. Once the processor has sent a read address to memory, it cannot proceed until the memory has had time to respond. Most memories are capable of responding much faster than the processing cycle requires. A few, however, cannot supply the addressed byte within the minimum time established by the processor's clock.

Therefore a processor should contain a synchronization provision, which permits the memory to request a **Wait state**. When the memory receives a read or write enable signal, it places a request signal on the processor's **READY** line, causing the CPU to idle temporarily. After the memory has

had time to respond, it frees the processor's **READY** line, and the instruction cycle proceeds.

Input/Output:

Input and Output operations are similar to memory read and write operations with the exception that a peripheral I/O device is addressed instead of a memory location. The CPU issues the appropriate input or output control signal, sends the proper device address and either receives the data being input or sends the data to be output.

Data can be input/output in either parallel or serial form. All data within a digital computer is represented in binary coded form. A binary data word consists of a group of bits; each bit is either a one or a zero. **Parallel** I/O consists of transferring all bits in the word at the same time, one bit per line. **Serial** I/O consists of transferring one bit at a time on a single line. Naturally serial I/O is much slower, but it requires considerably less hardware than does parallel I/O.

Interrupts:

Interrupt provisions are included on many central processors, as a means of improving the processor's efficiency. Consider the case of a computer that is processing a large volume of data, portions of which are to be output to a printer. The CPU can output a byte of data within a single machine cycle but it may take the printer the equivalent of many machine cycles to actually print the character specified by the data byte. The CPU could then remain idle waiting until the printer can accept the next data byte. If an interrupt capability is implemented on the computer, the CPU can output a data byte then return to data processing. When the printer is ready to accept the next data byte, it can request an interrupt. When the CPU acknowledges the interrupt, it suspends main program execution and automatically branches to a routine that will output the next data byte. After the byte is output, the CPU continues with main program execution. Note that this is, in principle, quite similar to a subroutine call, except that the jump is initiated externally rather than by the program.

More complex interrupt structures are possible, in which several interrupting devices share the same processor but have different priority levels. Interruptive processing is an important feature that enables maximum utilization of a processor's capacity for high system throughput.

Hold:

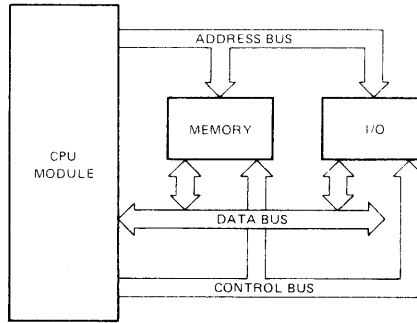
Another important feature that improves the throughput of a processor is the **Hold**. The hold provision enables **Direct Memory Access (DMA)** operations.

In ordinary input and output operations, the processor itself supervises the entire data transfer. Information to be placed in memory is transferred from the input device to the processor, and then from the processor to the designated memory location. In similar fashion, information that goes

from memory to output devices goes by way of the processor.

Some peripheral devices, however, are capable of transferring information to and from memory much faster than the processor itself can accomplish the transfer. If any appreciable quantity of data must be transferred to or from such a device, then **system throughput** will be increased by

having the device accomplish the transfer directly. The processor must temporarily suspend its operation during such a transfer, to prevent conflicts that would arise if processor and peripheral device attempted to access memory simultaneously. It is for this reason that a **hold** provision is included on some processors.



PART 2: INTRODUCTION TO MCS-85™

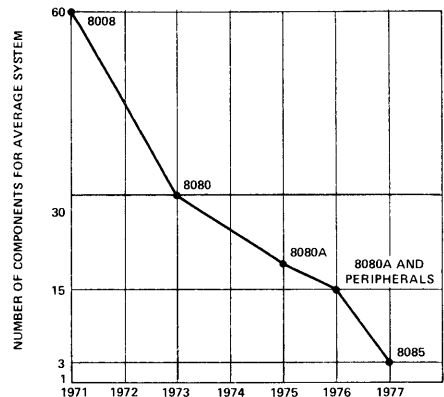
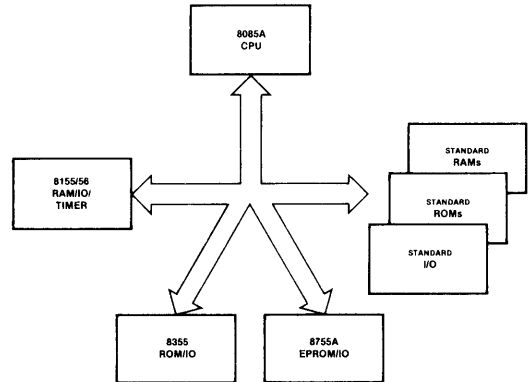
THE MCS-85™ MICROCOMPUTER SYSTEM

The basic philosophy behind the MCS-85 microcomputer system is one of logical, evolutionary advance in technology without the waste of discarding existing investments in hardware and software. The MCS-85 provides the existing 8080 user with an increase in performance, a decrease in the component count, operation from a single 5-Volt power supply, and still preserves 100% of his existing software investment. For the new microcomputer user, the MCS-85 represents the refinement of the most popular microcomputer in the industry, the Intel 8080, along with a wealth of supporting software, documentation and peripheral components to speed the cycle from prototype to production. The same development tools that Intel has produced to support the 8080 microcomputer system can be used for the MCS-85, and additional add-on features are available to optimize system development for MCS-85.

This section of the MCS-80/85 User's Manual will briefly detail the basic differences between the MCS-85 and MCS-80 families. It will illustrate both the hardware and software compatibilities and also reveal some of the engineering trade-offs that were met during the design of the MCS-85. More detailed discussion of the MCS-85 bus operation and component specifications are available in Chapters: 2, 3, 4, and 5. The information provided in Chapter 1 will be helpful in understanding the basic concepts and philosophies behind the MCS-85.

EVOLUTION

In December 1971, Intel introduced the first general purpose, 8-bit microprocessor, the 8008. It was implemented in P-channel MOS technology and was packaged in a single 18 pin, dual in-line package (DIP). The 8008 used standard semiconductor ROM and RAM and, for the most part, TTL components for I/O and general interface. It immediately found applications in byte-oriented end products such as terminals and computer peripherals where its instruction execution (20 micro-seconds), general

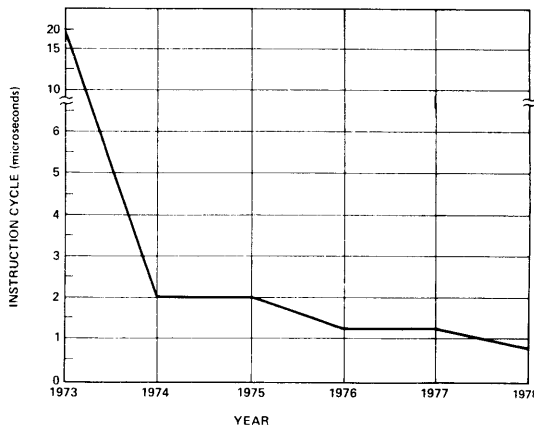


8-BIT SMALL SYSTEM COMPONENT COUNT 1971 - 1977

purpose organization and instruction set matched the requirements of these products. Recognizing that hardware was but a small part in the overall system picture, Intel developed both hardware and software tools for the design engineer so that the transition from prototype to production would be as simple and fast as possible. The commitment of providing a total systems approach with the 8008 microcomputer system was actually the basis for the sophisticated, comprehensive development tools that Intel has available today.

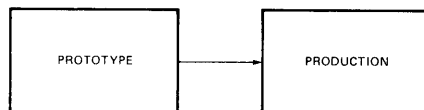
THE 8080A MICROPROCESSOR

With the advent of high-production N-channel RAM memories and 40 pin DIP packaging, Intel designed the 8080A microprocessor. It was designed to be software compatible with the 8008 so that the existing users of the 8008 could preserve their investment in software and at the same time provide dramatically increased performance (2 micro-second instruction execution), while reducing the amount of components necessary to implement a system. Additions were made to the basic instruction set to take advantage of this increased performance and large system-type features were included on-chip such as DMA, 16-bit addressing and external stack memory so that the total spectrum of application could be significantly increased. The 8080 was first sampled in December 1973. Since that time it has become the standard of the industry and is accepted as the primary building block for more microcomputer based applications than all other microcomputer systems combined.



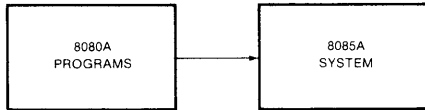
A TOTAL SYSTEMS COMMITMENT

The Intel® 8080A Microcomputer System encompasses a total systems commitment to the user to fully support his needs both in developing prototype systems and reliable, high volume production. From complex MOS/LSI peripheral components to resident high level systems language (PL/M) the Intel® 8080 Microcomputer System provides the most comprehensive, effective solution to today's system problems.



SOFTWARE COMPATIBILITY

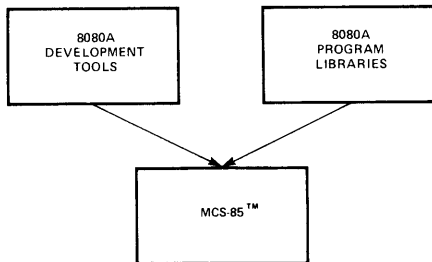
As with any computer system the cost of software development far outweighs that of hardware. A microcomputer-based system is traditionally a very cost-sensitive application and the development of software is one of the key areas where success or failure of the cost objectives is vital.



The 8085A CPU is 100% software compatible with the Intel® 8080A CPU. The compatibility is at the object or "machine code" level so that existing programs written for 8080A execution will run on the 8085A as is. The value of this becomes even more evident to the user who has mask programmed ROMs and wishes to update his system without the need for new masks.

PROGRAMMER TRAINING

A cost which is often forgotten is that of programmer training. A new, or modified instruction set, would require programmers to relearn another set of mnemonics and greatly affect the productivity during development. The 100% compatibility of the 8085A CPU assures that no re-training effort will be required.



For the new microcomputer user, the software compatibility between the 8085A and the 8080A means that all of the software development tools that are available for the 8080A and all software libraries for 8080A will operate with the new design and thus save immeasurable cost in development and debug.

The 8085A CPU does however add two instructions to initialize and maintain hardware features of the 8085A. Two of the unused opcodes of the 8080A instruction set were designated for the addition so that 100% compatibility could be maintained.

HARDWARE COMPATIBILITY

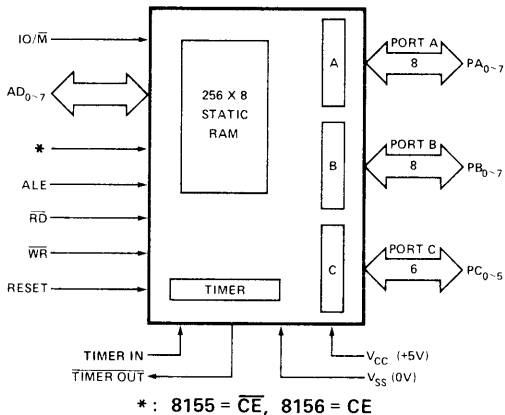
The integration of auxiliary 8080A functions, such as clock generation, system control and interrupt prioritization, dramatically reduces the amount of components necessary for most systems. In addition, the MCS-85 operates off a single +5 Volt power supply to further simplify hardware development and debug. A close examination of the AC/DC specifications of the MCS-85 systems components shows that each is specified to supply a minimum of 400µA of source current and a full TTL load of sink current so that a very substantial system can be constructed without the need for extra TTL buffers or drivers. Input and output voltage levels are also specified so that a minimum of 350mV noise margin is provided for reliable, high-performance operation.

PC BOARD CONSIDERATIONS

The 8085A CPU and the 8080A are not pin-compatible due to the reduction in power supplies and the addition of integrated auxiliary features. However, the pinouts of the MCS-85 system components were carefully assigned to minimize PC board area and thus yield a smooth, efficient layout. For new designs this incompatibility of pinouts presents no problems and for upgrades of existing designs the reduction of components and board area will far offset the incompatibility.

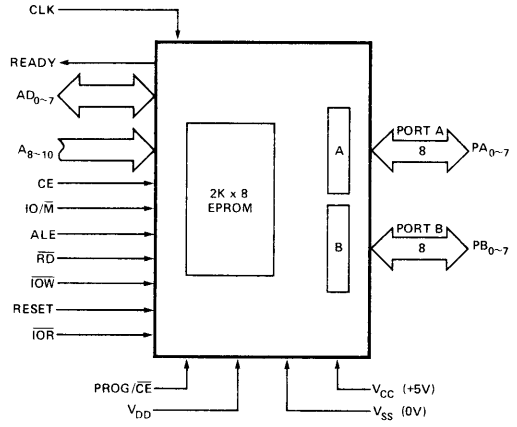
MCS-85™ SPECIAL PERIPHERAL COMPONENTS

The MCS-85 was designed to minimize the amount of components required for most systems. Intel designed several new peripheral components that combine memory, I/O and timer functions to fulfill this requirement. These new peripheral devices directly interface to the multiplexed MCS-85 bus structure and provide new levels in system integration for today's designer.



8155/8156 RAM, I/O and Timer

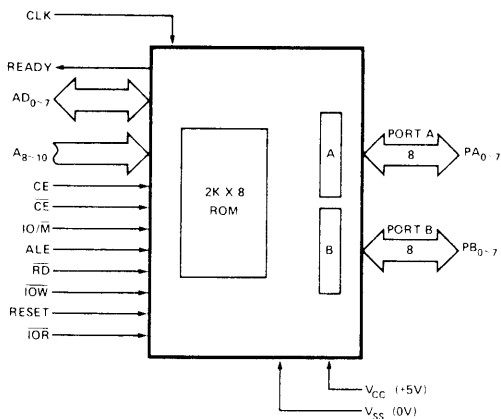
- 256 bytes RAM
- Two 8-bit ports
- One 6-bit port (programmable)
- One 14-bit programmable interval timer
- Single + 5 Volt supply operation
- 40 pin DIP plastic or cerdip package



8755A EPROM and I/O

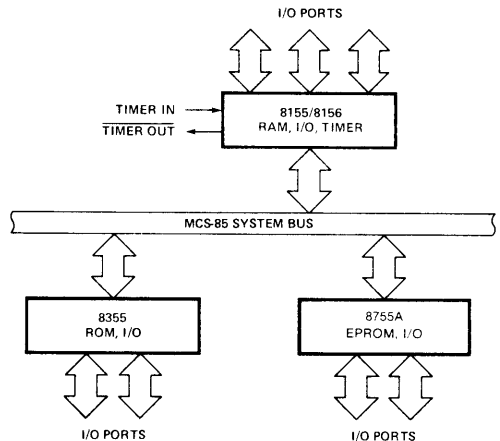
- Socket compatible with 8355
- 2K bytes EPROM
- Two 8-bit ports (direction programmable)
- Single + 5 Volt supply read operation
- U.V. Erasable
- 40 pin DIP package

One of the most important advances made with the MCS-85 is the socket-compatibility of the 8355 and 8755A components. This allows the systems designer to develop and debug in erasable PROM and then, when satisfied, switch over to mask-programmed ROM 8355 with no performance degradation or board layout. It also allows quick prototype production for market impact without going to a compromise solution.



8355 ROM and I/O

- 2K bytes ROM
- Two 8-bit ports (direction programmable)
- Single + 5 Volt supply operation
- 40 pin DIP plastic or cerdip package



SYSTEM EXPANSION

Each of these peripheral components has features that allow a small to medium system to be constructed without the addition of buffers and decoders to maintain the lowest possible component count.

INTRODUCTION TO MCS-85™

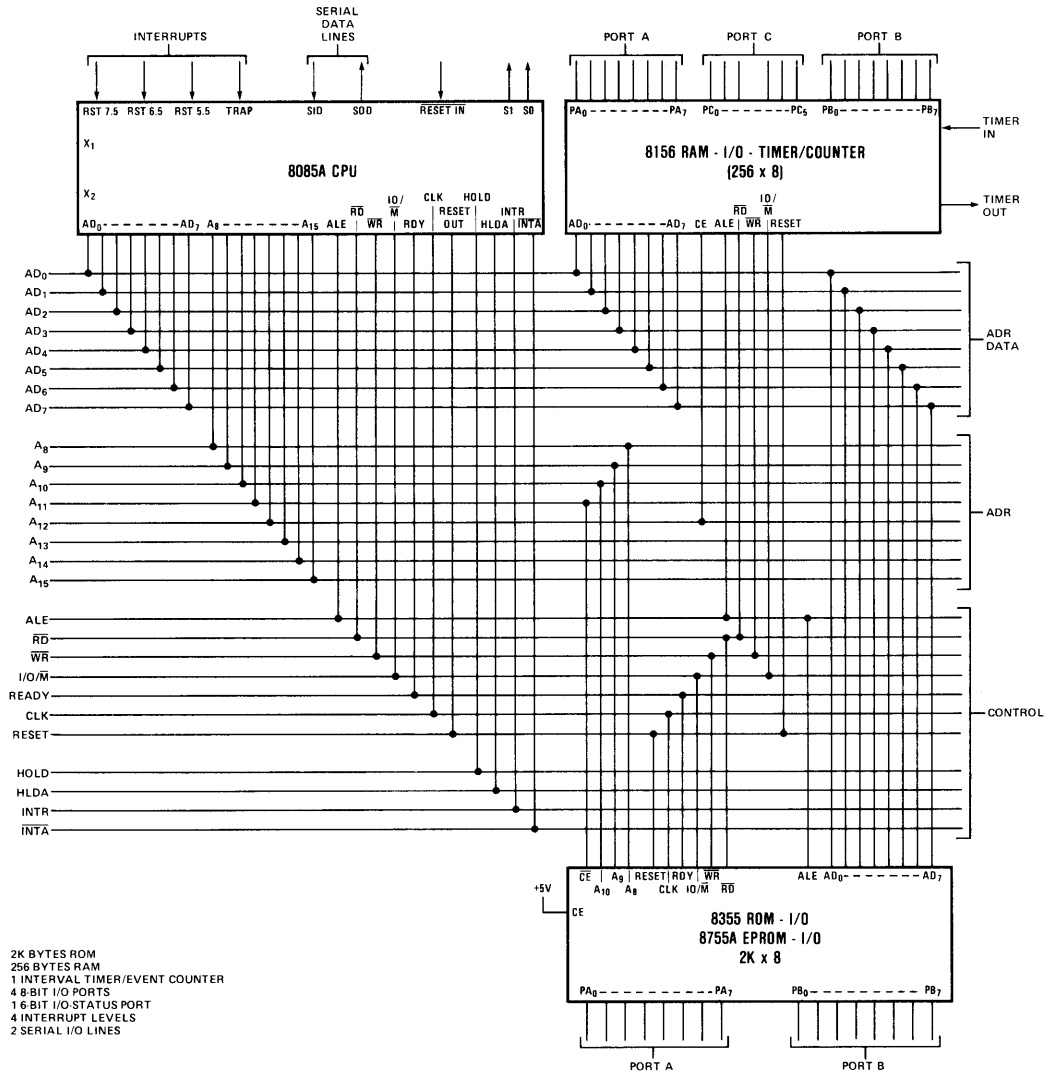
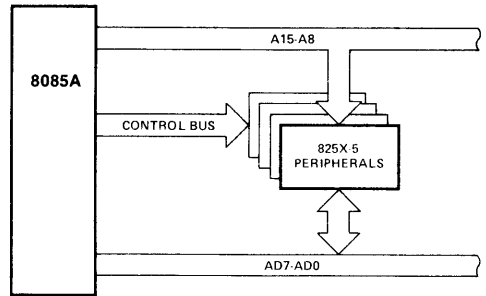


Figure 1-1. MCS-85™ Basic System

**INTERFACING TO MCS-80/85™
PROGRAMMABLE PERIPHERAL
COMPONENTS**

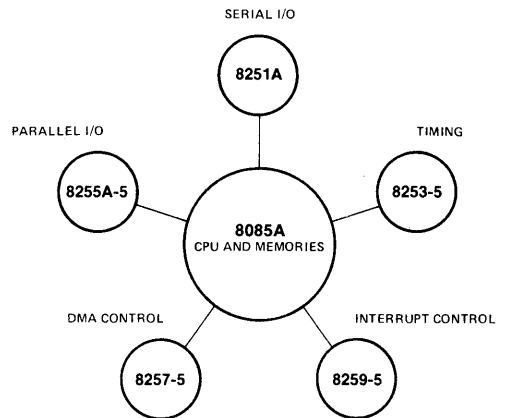
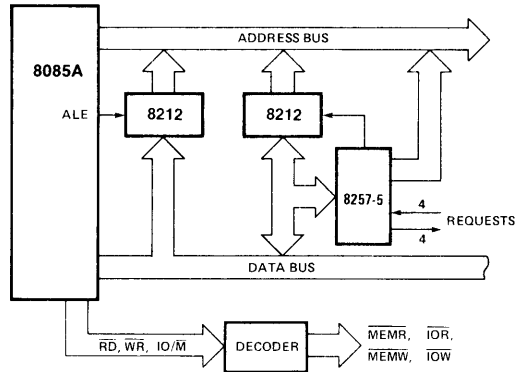
The MCS-85 shares with the MCS-80 a wide range of peripheral components that solve system problems and provide the designer with a great deal of flexibility in his I/O, Interrupt and DMA structures. The MCS-85 is directly compatible with these peripherals, and, with the exception of the 8257-5 DMA controller, needs no additional circuitry for their interface in a minimum system. The 8257-5 DMA controller uses an 8212 latch and some gating to support the multiplexed bus of MCS-85.



PROGRAMMABLE PERIPHERALS

The list of programmable peripherals for use with the 8085A includes:

- 8251A Programmable Communications Interface
- 8253-5 Programmable Interval Timer
- 8255A-5 Programmable Peripheral Interface
- 8257-5 Programmable DMA Controller
- 8259-5 Programmable Interrupt Controller
- 8271 Diskette Controller
- 8273 Synchronous Data Link Controller
- 8275 CRT Controller
- 8278 Keyboard/Display Controller
- 8279 Keyboard/Display Controller



The MCS-80/85 peripheral compatibility assures the designer that all new peripheral components from Intel will interface to the MCS-85 bus structure to further expand the application spectrum of MCS-85.

INTERFACING TO STANDARD MEMORY

The MCS-85 was designed to support the full range of system configurations from small 3 chip applications to large memory and I/O applications. The 8085A CPU issues advanced READ/WRITE status signals (S0, S1, and IO/M) so that, in the case of large systems, these signals could be used to simplify bus arbitration logic and dynamic RAM refresh circuitry.

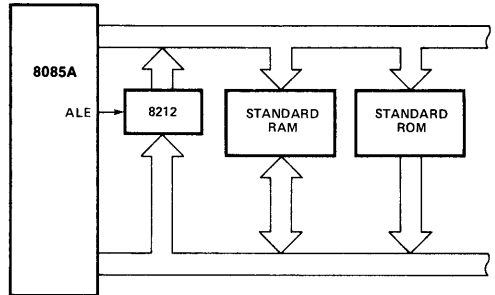
In large, memory-intensive systems, standard memory devices may provide a more cost-effective solution than do the special 8155 and 8355 devices, especially where few I/O lines are required.

DEMULTIPLEXING THE BUS

In order to interface standard memory components such as Intel® 2114, 2142, 2716, 2316E, 2104A and 2117 the MCS-85 bus must be "demultiplexed". This is accomplished by connecting an Intel® 8212 latch to the data bus and strobing the latch with the ALE signal from the 8085A CPU. The ALE signal is issued to indicate that the multiplexed bus contains the lower 8-bits of the address. The 8212 latches this information so that a full 16-bit address is available to interface standard memory components.

USE OF 8212

Large, memory intensive systems are usually multi-card implementations and require some form of TTL buffering to provide necessary current and voltage levels. Frequently, 8212s are used for this purpose. The 8212 has the advantage of being able to latch and demultiplex the address bus and provide extra address drive capability at the same time.



SYSTEM PERFORMANCE

The true benchmark of any microcomputer-based system is the amount of tasks that can be performed by the system in a given period of time. Increasing speed of CPU instruction execution has been the common approach to increasing system throughput but this puts a greater strain on the memory access requirement and bus operation than is usually practical for most applications. A much more desirable method would be to distribute the task-load to peripheral devices.

DISTRIBUTED PROCESSING

The concept of distributed task processing is not new to the computer designer, but until recently little if any task distribution was available to the microcomputer user. The use of the new programmable MCS-80/85 peripherals can relieve the central processor of many of the bookkeeping I/O and timing tasks that would otherwise have to be handled by system software.

INSTRUCTION CYCLE/ACCESS TIME

The basic instruction cycle of the 8085A is 1.3 microseconds, the same speed as the 8080A-1. A close look at the MCS-85 bus operation shows that the access requirement for this speed is only 575 nanoseconds. The MCS-80 access requirements for this speed would be under 300 nanoseconds. This illustrates the efficiency and improved timing margins of the MCS-85 bus structure. The new 8085A-2, a high-speed selected version of the 8085A with a .8 microsecond instruction cycle, provides a 60% performance improvement over the standard 8085A.

CONCLUSIONS: THROUGHPUT/COST

When a total system throughput/cost analysis is taken, the MCS-85 system with its advanced processor will yield the most cost-effective, reliable and producible system.

