

Introduction to Microprocessors

DCAP210

Edited by:
Gurwinder Kaur



L OVELY
P ROFESSIONAL
U NIVERSITY



INTRODUCTION TO MICROPROCESSORS

Edited By:
Gurwinder Kaur

Printed by
EXCEL BOOKS PRIVATE LIMITED
A-45, Naraina, Phase-I,
New Delhi-110028
for
Lovely Professional University
Phagwara

SYLLABUS

Introduction to Microprocessors

Objectives: To develop the hardware skills required for the complete understanding of the architecture and programming of the microprocessor used in computing world.

S. No.	Description
1.	Microprocessors, Microcomputers, and Assembly Language: Microprocessors, microprocessor Instruction Set and Computer Languages, Application
2.	Introduction to 8085 Assembly Language Programming: 8085 programming model, Instruction Classification, How to write a simple program?
3.	Microprocessor Architecture and Microcomputer Systems: Microprocessor architecture and its operation, Memory, I/O devices, Example of Microcomputer system
4.	8085 Microprocessor Architecture and Memory Interfacing: 8085 MPU, Memory Interfacing, How does an 8085- Based Single board Microcomputer work?
5.	Interfacing I/O Devices: Basic interfacing Concepts, Interfacing output displays, Interfacing input devices, Memory Mapped I/O
6.	Introduction to 8085 Instructions: Data transfer operations, Arithmetic operations, Logic operations, branch operations
7.	Programming Techniques with Additional Instructions: Programming techniques: Looping, Counting and Indexing, Additional Data transfer instructions
8.	Counters and Time delays: Counter and time delays, Illustrative program: Hexadecimal counter
9.	Stack and Subroutines: Stack, Subroutine, Restart, Conditional call and Return Instruction
10.	Interrupts: 8085 interrupts

CONTENT

Unit 1:	Microprocessors and Microcomputers <i>Gurwinder Kaur, Lovely Professional University</i>	1
Unit 2:	Introduction to Assembly Language <i>Gurwinder Kaur, Lovely Professional University</i>	20
Unit 3:	Assembly Language Programming of 8085 <i>Gurwinder Kaur, Lovely Professional University</i>	34
Unit 4:	Microprocessor Architecture <i>Dinesh Kumar, Lovely Professional University</i>	53
Unit 5:	Microcomputer System <i>Dinesh Kumar, Lovely Professional University</i>	67
Unit 6:	The 8085 Microprocessor Architecture <i>Dinesh Kumar, Lovely Professional University</i>	77
Unit 7:	Memory Interfacing <i>Parminder Kaur, Lovely Professional University</i>	97
Unit 8:	Interfacing I/O Devices <i>Parminder Kaur, Lovely Professional University</i>	110
Unit 9:	Introduction to 8085 Instructions <i>Parminder Kaur, Lovely Professional University</i>	125
Unit 10:	Programming Techniques with Additional Instructions <i>Gurwinder Kaur, Lovely Professional University</i>	137
Unit 11:	Counters and Time Delays <i>Gurwinder Kaur, Lovely Professional University</i>	152
Unit 12:	The Stacks <i>Dinesh Kumar, Lovely Professional University</i>	166
Unit 13:	Subroutines <i>Dinesh Kumar, Lovely Professional University</i>	176
Unit 14:	Interrupts <i>Gurwinder Kaur, Lovely Professional University</i>	185

Unit 1: Microprocessors and Microcomputers

Notes

CONTENTS

Objectives

Introduction

1.1 Microprocessor

1.2 Microcomputer

1.2.1 Automobile Analogy

1.2.2 Intel MCS-4 4-B Chip Set

1.2.3 Intel 8008 Microprocessor

1.2.4 8080 More and No More

1.3 Historical Perspective

1.3.1 Moore's Law

1.4 Microprocessor Instruction Set

1.4.1 Implied Addressing

1.4.2 Register Addressing

1.4.3 Immediate Addressing

1.4.4 Direct Addressing

1.4.5 Register Indirect Addressing

1.4.6 Combined Addressing Modes

1.4.7 Timing Effects of Addressing Modes

1.4.8 Decoding

1.5 Summary

1.6 Keywords

1.7 Self-Assessment Questions

1.8 Review Questions

1.9 Further Reading

Objectives

After studying this unit, you will be able to understand the following:

- Explain about microprocessor
- Describe the important areas of microprocessor
- Explain about microcomputer
- Understand the microprocessor instruction set

Notes

Introduction

A microprocessor, sometimes called a logic chip, is a computer processor on a microchip.

The microprocessor contains all, or most of, the central processing unit (CPU) functions and is the “engine” that goes into motion when you turn your computer on. A microprocessor is designed to perform arithmetic and logic operations that make use of small number-holding areas called registers. Typical microprocessor operations include adding, subtracting, comparing two numbers, and fetching numbers from one area to another. These operations are the result of a set of instructions that are part of the microprocessor design.

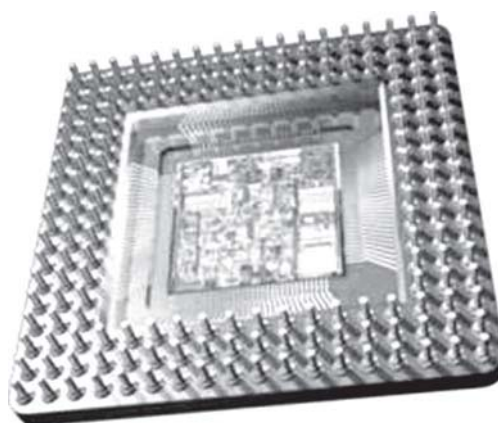
When your computer is turned on, the microprocessor gets the first instruction from the basic input/output system (BIOS) that comes with the computer as part of its memory. After that, either the BIOS, or the operating system that BIOS loads into computer memory, or an application program is “driving” the microprocessor, giving it instructions to perform.

1.1 Microprocessor

Microprocessors are regarded as one of the most important devices in our everyday machines called computers. Before we start, we need to understand what exactly microprocessors are and their appropriate implementations. Microprocessor is an electronic circuit that functions as the central processing unit (CPU) of a computer, providing computational control. Microprocessors are also used in other advanced electronic systems, such as computer printers, automobiles, and jet airliners. Typical microprocessors incorporate arithmetic and logic functional units as well as the associated control logic, instruction processing circuitry, and a portion of the memory hierarchy. Portions of the interface logic for the input/output (I/O) and memory subsystems may also be infused, allowing cheaper overall systems. While many microprocessors have single chip designs, some high-performance designs rely on a few chips to provide multiple functional units and relatively large caches.

When combined with other integrated circuits that provide storage for data and programs, often on a single semiconductor base to form a chip, the microprocessor becomes the heart of a small computer, or microcomputer. Microprocessors are classified by the semiconductor technology of their design (TTL, transistor-transistor logic; CMOS, complementary-metal-oxide semiconductor; or ECL, emitter-coupled logic), by the width of the data format (4-bit, 8-bit, 16-bit, 32-bit, or 64-bit) they process; and by their instruction set (CISC, complex-instruction-set computer, or RISC, reduced instruction set computer: see RISC processor). TTL technology is most commonly used, while CMOS is favoured for portable computers and other battery-powered devices because of its low power consumption. ECL is used where the need for its greater speed offsets the fact that

Figure 1.1: Chips



it consumes the most power. Four-bit devices, while inexpensive, are good only for simple control applications; in general, the wider the data format, the faster and more expensive the device. CISC processors, which have 70 to several hundred instructions, are easier to program than RISC processors, but are slower and more expensive.

Microprocessors have been described in many different ways. They have been compared with the brain and the heart of humans. Their operation has been likened to a switched board, and to the nervous system in an animal. They have often been called microcomputers. The original purpose of the microprocessor was to control memory. That is what they were originally designed to do, and that is what they do today. Specifically, a microprocessor is “a component that implements memory.”

A microprocessor can do any information-processing task that can be expressed, precisely, as a plan. It is totally uncommitted as to what its plan will be. It is a truly general-purpose information processing for microprocessor.



Did u know?

Computer's performance is also influenced by the system bus architecture, memory used, type of the processor and software program being running. Pentium 4 is the fastest type of the Intel's processor that contains 125,000,000 transistors and operates at the speed of 3.6 GHz.

1.2 Microcomputer

The term microcomputer is generally synonymous with personal computer, or a computer that depends on a microprocessor. Microcomputers are designed to be used by individuals, whether in the form of PCs, workstations or notebook computers. A microcomputer contains a CPU on a microchip (the microprocessor), a memory system (typically ROM and RAM), a bus system and I/O ports, typically housed in a motherboard.

Intel's first microcomputer appeared in November 1971:

Intel delivered two different microcomputers five months apart: the MCS-4, emphasizing low cost in November 1971, and the MCS-8, for versatility in April 1972. “The MCS-4 and MCS-8 CPU chip sell in quantity for less than ₹ 1000 each, and are powerful alternatives to random logic”. These two Microcomputer Systems (MCS) were aimed at two very different markets. One would eventually lead to the under ₹ 4500 controller, the other would be the engine for a versatile personal computer (PC). By analogy it was like creating the “motorbike” and the “station wagon” at the same time. The advertised prophecy of “a new era” became fulfilled over the subsequent 20-year period.

1.2.1 Automobile Analogy

Our challenge was how to scale down a general purpose computer to fit on to a chip. Imagine that the only passenger vehicle in existence is an eight-passenger van costing ₹ 2250 000. At first it would be difficult to imagine ₹ 1000 version of this vehicle. The specifications would need to be drastically reduced to meet the price goal. Some ideas to consider:

1. reducing capacity by 75%
2. reducing speed by 90%
3. reducing range by 75%

The golf cart might be the result. However, if golf carts are unknown at the time, it is not easy to envision how to scale down a van.

1.2.2 Intel MCS-4 4-B Chip Set

Although Intel began as a memory chip company, in 1969 we took on a project for Busicom of Japan to design eight custom LSI chips for a desktop calculator. Each custom chip had a specialized

Notes

function—keyboard, printer, display, serial arithmetic, control, etc. With only two designers, Intel didn’t have the manpower to do that many custom chips. We needed to solve their problems with fewer chip designs. Ted Hoff chose a programmed computer solution using only one complex logic chip (CPU) and two memory chips; memory chips are repetitive and easier to design. Intel was a memory chip company, so we found a way to solve our problem using memory chips!

In 1970 Intel designers implemented a 4-b computer on three LSI chips (CPU, ROM, RAM) housed in 16-pin packages. Reducing the data word to 4-b (for a BCD digit) was a compromise between 1-b serial calculator chips and conventional 16-b computers.

Figure 1.2: Transistor



The scaled down 4-b word size made the CPU chip size practical (2200 transistors). We used the 16-pin package, because it was the *only one* available in our company. This limited pin count forced us to time multiplex a 4-b bus. This small bus simplified the printed circuit board (PCB), as it used fewer connections. However, the multiplex logic increased chip area of the specialized ROM/RAM memory chips, which then had to have built-in address registers. Increasing the transistor count to save chip connections was a novel idea. In school we learned to minimize logic, not interconnections! Later, LSI “philosophers” would preach, “logic is free”.

	1. MCS-4 Features:
256 × 8	Read Only Memory (2 kb ROM) with 4-b I/O port
80 × 4	Random Access Memory (320 b RAM) with 4-b output port
4-b	CPU chip with: 16 x 4-b index registers, 45 1 and 2 byte instructions, 4-level Subroutine Address Stack, 12-b Program Counter (4 k addresses).

(A) ROM Chip (4001)

Conventional calculators utilized specialized custom chips for keyboard, display, and printer control. With the MCS-4 all control logic is done in firmware, program stored in ROM. A single ROM chip design is customized (with a mask during chip manufacturing) for a customer’s

Figure 1.3: ROM Chip



particular program. The CPU's 12-b Program Counter addresses up to 16 ROM chips. Simple applications use only one ROM chip; the desktop calculator used four. The same chip mask also configured each ROM port bit as an input or output.

Additionally, the ROM chip had an integrated address register, an output data register, multiplexors, and control and timing logic. The specialized RAM chip had similar resources.

(B) RAM Chip (4002)

Calculators need to hold several 16-digit decimal floating-point numbers. We organized the RAM accordingly, and ended up with a 20-digit word (80 b):

16 digits for the fraction

2 digits for the exponent

2 digits for signs and control

20 digits × 4 b/digit

Figure 1.4: RAM Chip (4002)



The RAM chip stored four 80-b numbers and additionally the chip had an output port. The use of three-transistor dynamic memory cells made the RAM chip feasible. A built-in refresh counter was used to maintain data integrity. Refresh took place during instruction fetch cycles, when the RAM data was not being accessed. Dynamic RAM memory cells were also used inside the CPU for the 64-b index register array and 48-b Program Counter/stack array. Intel expertise in dynamic memory was an enabling factor for the MCS-4!

(C) Input/Output Ports

To conserve chip count and to utilize existing power/clock pins, the 16-pin ROM and RAM chips also had integrated 4-b ports for direct connection of I/O devices. To activate an output, a program selected a particular RAM/ROM chip (using an index register) and sent 4-b of accumulator data from the CPU to the selected output port. In the desk calculator application, the display, keyboard, and printer were connected to these ports. Keyboard scanning, decoding, and debouncing were all done under program control of the I/O ports; all printer and display refresh was done in firmware. A small shift register (4003) was used for output port expansion. External transistors and diodes were used for amplification and isolation.

Figure 1.5: Input/Output Ports



Notes

(D) Microprocessor — CPU Chip (4004)

In the calculator application, each user key stroke caused thousands of CPU instructions to be executed from ROM. We wrote many subroutines which operated on 16-digit numbers stored in RAM.

Figure 1.6: Microprocessor — CPU Chip



As an example, a 10-byte loop for digit serial addition took about 80 μ s/digit (similar speed as IBM 1620 computer sold in 1960 for ₹ 100 000). In this add routine a CPU index register would address each of the 16 digits stored in the RAM memory. The program would bring in one digit at a time into the CPU’s accumulator register to do arithmetic. A Decrement and Jump instruction was used to index to the next RAM location. One major difference, compared to most computers, was the MCS-4’s separate program and data memories. Conventional computers ran programs from RAM (core) memory. However, our application firmware needed to be permanently stored in ROM. A major change was needed for subroutine linkage. Normally, as part of a minicomputer subroutine call instruction execution (PDP-8, HP 2114) the calling program’s return address would be saved at the top of the subroutine in RAM. Since MCS-4 routines were in ROM (can’t write into it) we could not use this method. Instead. We used, a push down stack inside the CPU for saving up to three return addresses. This was not a new idea. Stacks had been used in Burrough’s computers and the IBM 1620, which Ted Hoff had programmed—we used our experience with large-scale computers. Ultimately this limited depth of four levels (which was all we could squeeze on to this small chip) was frustrating for programmers and succeeding generations went to eight or more levels (8008, 4040, 8048). Today’s computers have stacks of many megabytes; but their usage is very similar to their use in the 4004.

(E) Distributed Logic Architecture

The time division multiplexing of the 4-b bus, the on-chip dynamic RAM memories, and the CPU’s address stack are the highlights of the MCS-4 architecture. However, there is another interesting feature—distributed decoding of instructions. The ROM/RAM chips watched the bus, and locally decoded port instructions, as they were sent from the ROM. This eliminated the need for the CPU to have separate signal lines to the I/O ports, and also saved CPU logic. This is not a feature used in conventional computers.

(F) MCS-4 Applications

The smallest system would contain two chips a CPU and a ROM. A typical calculator had 4 ROM’s and a RAM chip with five I/O ports, (20) wires for connecting peripheral devices. A fully loaded system could have 16 ROM and 16 RAM chips, and obviously a plethora of I/O ports. Typical applications included:

digital scales	taxi meters
gas pumps	traffic light
elevator control	vending machines
medical instruments	

Figure 1.7: MCS-4



Busicom of Japan produced several calculator models using the MCS-4 chip set. Ted Hoff made the original proposal for the MCS-4 and did the feasibility study for the first calculator. Federico Faggin did all of the logic and circuit design and implemented the layout; Busicom's M. Shima wrote most of Busicom's firmware. The Intel patent on the MCS-4 has 17 claims, but the single chip processor is not claimed as an invention.

Intel supported the MCS-4 with a Cross assembler and later with a standalone development system, the Intellec "blue" box.

The MCS-4 evolved into the single chip microcomputers 8048/8051. These chips emphasized small size and low cost. These, along with a variety of other manufacturer's parts, have evolved into the under 1 computer on a chip used in toys, automobiles, and appliances. These chips are very pervasive—almost invisible.

1.2.3 Intel 8008 Microprocessor

Intel made a custom 512-b shift register memory chip for use in (their customer) Data point's low cost bit-serial computer. This 8-b CPU, implemented with TTL MSI, had around 50 data processing instructions.

This custom chip design was never used by Data point, and it became a standard Intel product, which marketing dubbed the 8008 (twice 4004!).

Although the arithmetic unit and registers were twice as large as in the MCS-4, we expected that the control logic could be about the same if we deleted a few Data point defined instructions. Unlike the MCS-4's two-memory address space, the 8008 had one memory address space for program and data. The symmetric and regular instruction set was attractive. However, the only memory addressing was indirect through the High-Low (HL) register pair. Today's computers have huge amounts of memory, and a plethora of memory addressing instructions.

The 8008 CPU had six 8-b general purpose registers (B, C, D, E, H, L) and an 8-b accumulator. The push down program counter stack had 8-levels. Both of these register arrays were implemented with dynamic memory cells and the CPU had built-in "hidden" refresh during instruction fetch cycles, similar to the MCS-4.

We decided that the 8008 would utilize standard memory components (not custom ROM's and RAM's as in the MCS-4). This increased the parts count on a minimum system because separate

Figure 1.8: 8008 Microprocessor



Notes

address registers, multiplexors and I/O latch chips would need to be added to make the system work; in practice about 40 additional small chips were needed. But standard memories were available in high volume at low cost, and in a larger system the extra chip overhead could be tolerated. Using memory chips with different access times requires a synchronizer scheme, and therefore ready/wait signal pins were provided to perform a handshake function. These interface signals are more sophisticated in today's processors, but the 8008 demonstrated the idea.

The availability of Electrically Programmed ROM's (EPROM) was significant in allowing customers to experiment with their software. A product synergy evolved between Intel's memory component business and the microprocessor. Intel had an 18-pin package in volume production for the 1k dynamic RAM chip (1103); this gave two more pins for the 8008 than we had on the MCS-4, but we still had to time-multiplex an 8-b bus. By reducing the Program Counter width to 14-b we saved two package pins. The jump instruction contained a 16-b address, but two of the bits were ignored. The 8008 could have 16k bytes of memory, and at the time, this seemed enormous. (Today, users want 16 meg.)

(A) Little Endian

Some have wondered why the addresses in the 8008 were stored "backward" with the little end first, e.g. the low order byte of a two-byte address is stored in the lower addressed memory location. We (regrettably) specified this ordering as part of the JUMP instruction format in the spirit of compatibility with the Data point 2200. Recall that their original processor was bit serial; the addresses would be stored low to high bit in the machine code (bit-backward). Other computer makers organize the addresses with the "big end" first. The lack of standardization has been a problem in the industry.

(B) Applications

One of the first users of the 8008 was Seiko in Japan for a sophisticated scientific calculator. Other users included business machines and a variety of general purpose computers.

Intel did not apply for a patent on the 8008. Data point contracted with Texas Instruments in 1970 to get a second source for this chip. TI patented their design, but never got into production.

After about one year of experience with programming the 8008 CPU chip, we had a number of requested enhancements from our users. We proposed to build the 8080 as a follow on chip; this chip was very popular and led to the microcomputer revolution and the Personal Computer. It is ironic that Data point ultimately competed in the marketplace with PC products based upon their own, Data point defined, architecture!

1.2.4 8080 More and No More

(A) More

Based upon Intel's success with their new microcomputer product line Faggin convinced Vasdasz in 1972 to fund a project to convert the P-MOS 8008 into the newer N-MOS technology. This technology offered about a 2x speedup without making logic changes. After a short study, it was determined that a new mask set was needed because of the incompatibility of transistor size ratios. Faggin reckoned that since a new mask set was needed, he would fix some of the 8008's shortcomings.

We evolved the 8080 specification to improve performance 10x. We used the greater density to put in more logic (4500 transistors) and do more in parallel; the on chip control logic grew by 50%.

We put the stack in memory, did 16-b operations, and improved memory addressing. Now 40-pin plastic packages were available, and the address bus and data bus could be brought out in parallel. This design also simplified the external circuitry and TTL voltage compatible signals were provided.

Figure 1.9: 8080 Chip



Deleting the on-chip stack saved chip area, but was a net advantage to the user—now the stack had unlimited size. In the 8080, the registers were arranged as pairs of 8 b, to provide 16-b data handling. The three register pairs were designated as: BC, DE, HL. The High/Low register pair was the only way to address memory in the older 8080. This was limiting to programmers, so in the 8080 direct memory addressing instructions were added, as well as several specialized instructions for the HL register pair. One instruction, XTHL, provided for exchanging the top of stack with HL; another instruction, XHLD, swapped the contents of HL with the DE register pair. As these special instructions were not very symmetric, applying only to HL, we optimized their logic implementation. One of Ted Hoff's tricks was the use of an *exchange* flip-flop for DE/HL. This flip-flop designated one of the pairs as HL and the other register pair as DE. Simply toggling this flip-flop affected an apparent exchange! This saved a lot of logic; but by mistake, the reset pin had been connected to this flip-flop. An early 8080 user manual stated: "after reset, the HL/DE register contents may be exchanged" (later the reset connection was cut). The lack of instruction set symmetry was a nuisance to programmers and later CPU's instruction sets were considerably more regular; of course there were more transistors "to bum."

(B) No More

This is why the last 12 instructions were never implemented and why there was room in the instructions set for the 8085 microprocessor's added instructions. The 8080 was very successful in the market. Meanwhile, competition blossomed and a variety of great processors developed including the Motorola 6800 and the MOS Technology 6502. The 8080 CPU chip was patented by Intel and has three claims.

(C) 8085

To meet competition in 1976, Intel decided to develop a more integrated version of the 8080. This chip contained 6500 transistors. The new N-MOS was more TTL compatible and this chip needed few external parts.

There were 12 unused operation codes in the 8080 which provided room to expand the CPU's function. At Intel, a committee studied, argued, and finally decided after many months which instructions to add. Although all of these new codes were utilized by the 8085 designers, by the time this product got to market it was almost obsolete. To reduce compatibility requirements with the 8086 which was in design, 10 of the new 12 instructions were never announced in the data sheet. They have only been an interesting historical anomaly and a lesson about design by committee.

Figure 1.10: 8085 Chip



Notes

(D) 8086

In 1978, Intel's W. Davidow, vice president of the microcomputer group, rushed to staff a 16 b microcomputer development project. It was to have around 30 000 transistors, 12 times more than the 4004.

Figure 1.11: 8086 Chip



This new computer had multiplication and division and a host of other new features. However, it was constrained to be upwardly compatible with the 8080 (and 8008). Accordingly, the designers decided to keep the 16 b basic addresses and to use segment registers to get extended 20 b addresses. Two versions were created—the 8088 had an 8-b data bus for compatibility with 8-b memory systems, and the 16 b 8086. With 1 megabyte of memory addressing, this processor was a serious contender in the computer market place. This chip density required to match the 16-b minicomputers was “arriving” as had been predicted.

The decision by IBM to use the 8088 in a word processor and personal computer created enormous market momentum for Intel. The 186, 286, 386, 486 followed over the next 15 years, with some shadow of 8008 features still apparent. These components would be “truly pervasive”.



Caution

Chip density should be in multiple of 2 in the microprocessor versions.

1.3 Historical Perspective

The promises of high density solid state circuitry were becoming apparent in the 1950's. In 1959, Holland contemplated large-scale computers built with densities of 108 components per cubic foot. The integrated circuit was developed in parallel at both TI and Fairchild; the density of IC's was doubling every year. “Entire subsystems on a chip” were predicted if a high volume standard chip could be defined. For a 1966 forecast of chip complexity, it was then estimated that about 10 k-20 k gates would fit on a chip and that a good portion of a CPU would therefore be on one chip.

1.3.1 Moore's Law

Gordon Moore's empirical relationship is cited in a number of forms, but its essential thesis is that the numbers of transistors which can be manufactured on a single die will double every 18 months. The starting point for this exponential growth curve is usually set at 1959 or 1962, the period during which the first silicon planar transistors were designed and tested.

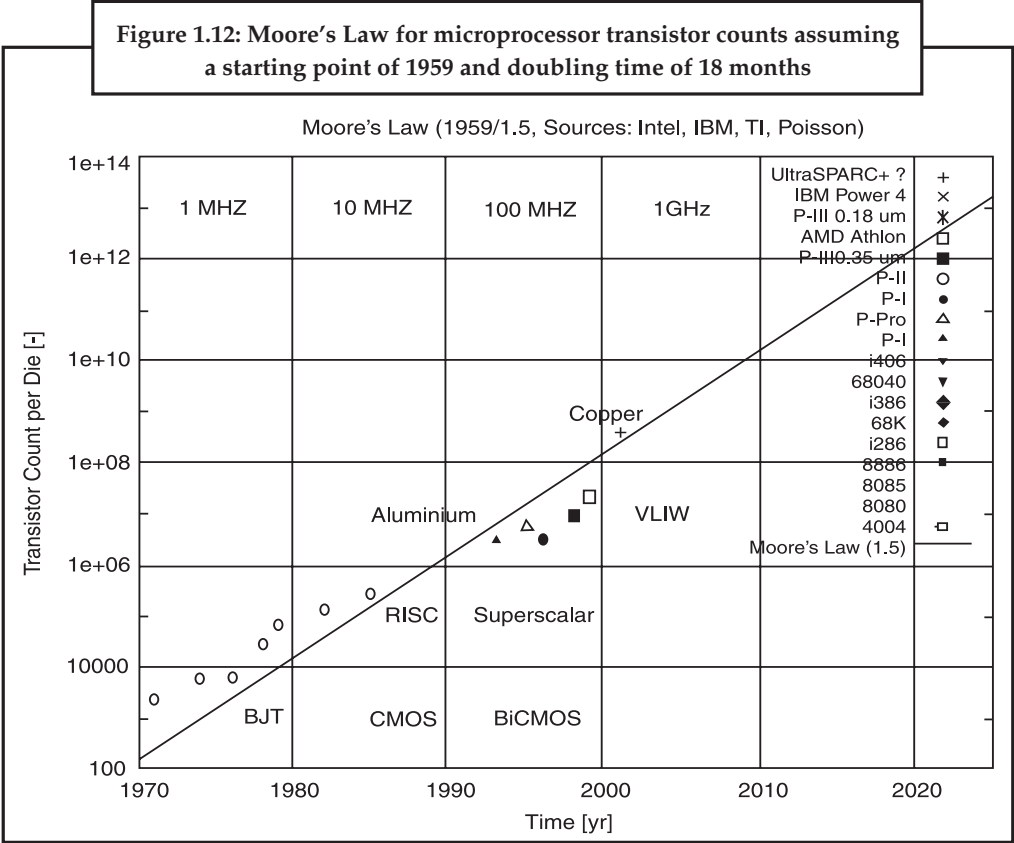
We now have four decades of empirical data to validate Moore's argument. Figure 1.12 depicts a sampling of microprocessor transistor counts, against the basic form of Moore's Law. Clearly the empirical data supports the argument well, even allowing for considerable noise in the dataset.

Similar plots for high density Dynamic Random Access Memory (DRAM) devices yield a very similar correlation between Moore's Law and actual device storage capacity.

Two important questions can be raised. The first is that of “how do we relate the achievable computing performance of systems to Moore's Law?”. The second is the critical question of “how long will Moore's Law hold out?”. Both deserve careful examination.

Notes

The computing performance of a computer system cannot be easily measured, nor can it be related in a simple manner to the number of transistors in the processor itself. Indeed, the only widely used measures of performance are various benchmark programs, which serve to provide essentially ordinal comparisons of relative performance for complete systems running a specific benchmark. This is for good reasons, since the speed with which any machine can solve a given problem depends upon the internal architecture of the system, the internal micro architecture of the processor chip itself, the internal bandwidth of the buses, the speed and size of the main



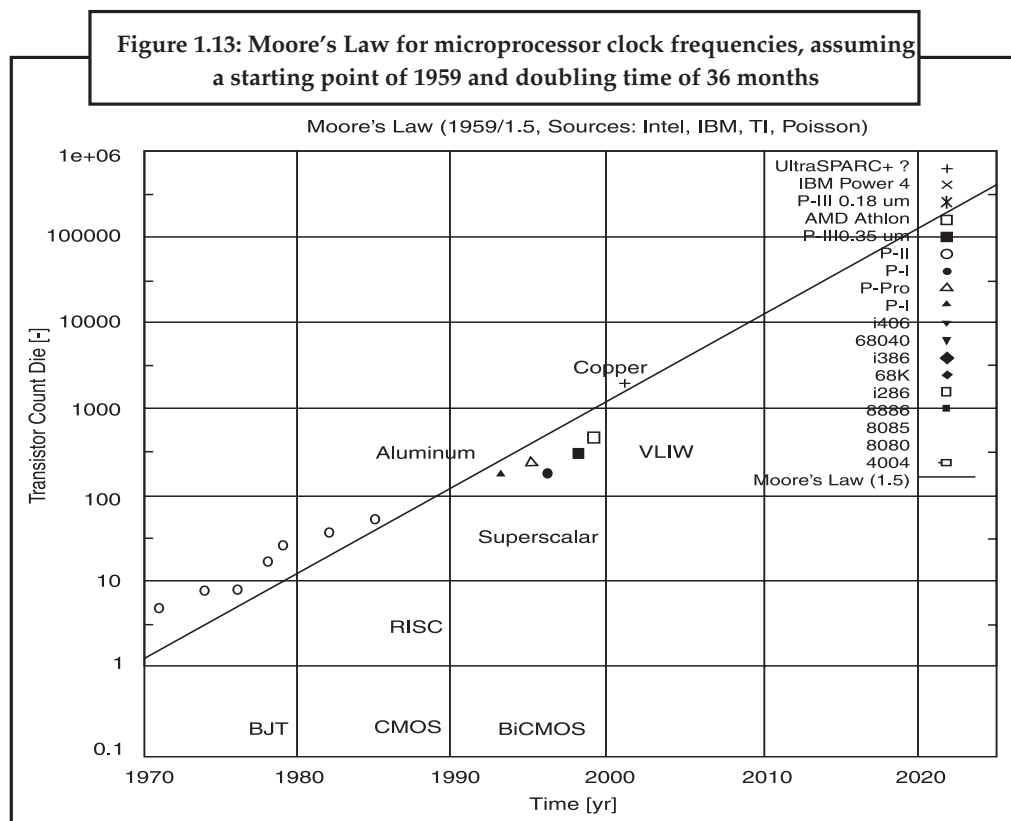
memory, the performance of the disks, the behaviour of the operating system software and the characteristics of the compiler used to generate executable code. The clock speed of the processor chip itself is vital, but in many instances may be less relevant than the aggregated performance effects of other parts of the system.

What can be said is that machines designed with similar internal architectures, using similar operating systems, compilers and running the same compute bound application, will mostly yield benchmark results in the ratios of their respective clock speeds. For instance, a compute bound numerically intensive network simulation written by the author was run on three different generations of Pentium processor, and a mid 1990s SuperSPARC, all running different variants of UNIX, but using the same GCC compiler. The time taken to compute the simulation scaled, within an error of a few percent, with the inverse ratio of clock frequencies. Indeed, careful study of published benchmarks tends to support this argument.

The empirical observation that computing performance is like architecture machines scales approximately with the clock frequency of the chip is useful, insofar as it allows us to relate achievable performance to Moore's Law, with some qualifying caveats. Mead observes that clock speeds scale with the ratio of geometry sizes, as compared to transistor counts which scale with the square of the ratio of geometry sizes. The interpretation of Moore's Law used in Figure 1.13 assumes this square root dependency, and incorporates a scaling factor to adjust the frequency to measured data. The plot in Figure 1.13 shows good agreement with Mead's model.

Notes

The conclusion that we can draw is that we will see a direct performance gain over time, proportional to the square root of the Moore's Law exponential, in machines with a given class of architecture. Since most equipment in operational use today is through prior history locked into specific architectures, such as Intel x86, SPARC, PowerPC, Alpha, MIPS and other, the near term consequence is that we will see performance increase exponentially with time for the life of the architecture.



However, major changes in internal architecture can produce further gains, at an unchanged clock speed. Therefore, the actual performance growth over time has been greater than that conferred by clock speed gains alone. Higher transistor counts allow for more elaborate internal architectures, thereby coupling performance gains to the exponential growth in transistor counts, in a manner which is not easily scaled like clock speeds. This effect was observed in microprocessors with the introduction of pipelining in the 1980s, superscalar processing in the 1990s and will be soon observed again with the introduction of VLIW architectures over the next two years. Since we are also about to observe the introduction of copper metallization during this period, replacing aluminum which has been used since the sixties, we can expect to see a slight excursion above the curve predicted by Moore's Law.

This behaviour relates closely to the second major question, which is that of the anticipated valid lifetime of Moore's Law. Many predictions have been made over the last two decades in relation to its imminent end. However, to date every single obstacle in semiconductor fab processes and packaging has been successfully overcome.

It is not difficult to observe that the limits on the scaling of transistor sizes and thus the achievable transistor counts per die are bounded by quantum physical effects. At some point, carriers will tunnel between structures, rendering transistors unusable and insulators leaky, beyond some point the charge used to define logical states will shrink down to the proverbial single electron.

The bounds for microprocessors are less clear, especially with emerging technologies such as Quantum Dot Transistors (QDT), with sizes of the order of 10 nm, as compared to current MOS technology devices which are at least twenty times larger. It follows that extant and nascent semiconductor component technologies should be capable of supporting further density growth until at least 2010. We cannot accurately predict further process improvements beyond that time, which accords well with “Mead’s Rule” and its projection of 11 years.

However, there is considerable further potential for performance growth in machine architectures. To date most architectural evolution seen in microprocessors has been little more than the reimplementing of architectural ideas used in 1960s and 1970s mainframes, minicomputers and supercomputers, made possible by larger transistor counts. We have seen little architectural innovation in recent decades, and only modest advances in parallel processing techniques.

The achievable performance growth resulting from the adoption of VLIW architectures remains to be seen. While these offer much potential for performance growth through instruction level parallelism, they do not address the problems of parallel computation on multiple processors, and it is unclear at this time how well they will scale up with larger numbers of execution units in processors.

It would be naive to assume that we have already wholly exhausted the potential for architectural improvements in conventional Von-Neumann model stored program machines. Indeed if history teaches us anything, it is that well entrenched technologies can be wiped out very rapidly by new arrivals: the demise of the core memory under the onslaught of the MOS semiconductor memory is a classical case study, as is the GMR read head on the humble disk drive, which rendered older head technologies completely uncompetitive over a two-year period.

It may well be that quantum physical barriers will not be the limiting factor in microprocessor densities and clock speeds, rather the problems of implementing digital logic to run at essentially microwave carrier frequencies will become the primary obstacle to higher clock speeds.

The current trend to integrate increasingly larger portions of the computer system on a single die will continue, alleviating the problem in the medium term; however a 2020 microprocessor running at a 60 GHz clock speed is an unlikely proposition using current design techniques. Vector processing supercomputers built from discrete logic components reached insurmountable barriers of this ilk at hundreds of megahertz, and have become a legacy technology as a result.

Do other alternatives to the monolithic Silicon chip exist? Emerging technologies such as quantum computing and nano-technology both have the potential to further extend performance beyond the obstacles currently looming on the 2010-2020 horizons. Neural computing techniques, Mead argues, have the potential to deliver exponential performance growth with size, rather than speed, in the manner predicted for as yet unrealized highly parallel architectures.

It follows that reaching the speed and density limits of semiconductor technology may mean the end of exponential growth in single chip density and clock speeds, but it is no guarantee that the exponential growth in computer performance will slow down.

What we can predict with a high level of confidence is that Moore’s Law will hold for the coming decade, and exponential growth is very likely to continue beyond that point.



Task

Explain the Concept of Moore’s Law.

1.4 Microprocessor Instruction Set

Instructions can be categorized according to their method of addressing the hardware registers and/or memory.

Notes

1.4.1 Implied Addressing

The addressing mode of certain instructions is implied by the instruction's function. For example, the STC (set carry flag) instruction deals only with the carry flag, the DAA (decimal adjust accumulator) instruction deals with the accumulator.

1.4.2 Register Addressing

Quite a large set of instructions call for register addressing. With these instructions, you must specify one of the registers A through E, H or L as well as the operation code. With these instructions, the accumulator is implied as a second operand. For example, the instruction CMP E may be interpreted as 'compare the contents of the E register with the contents of the accumulator.

Most of the instructions that use register addressing deal with 8-bit values. However, a few of these instructions deal with 16-bit register pairs. For example, the PCHL instruction exchanges the contents of the program counter with the contents of the H and L registers.

1.4.3 Immediate Addressing

Instructions that use immediate addressing have data assembled as a part of the instruction itself. For example, the instruction CPI 'C' may be interpreted as 'compare the contents of the accumulator with the letter C'. When assembled, this instruction has the hexadecimal value FE43. Hexadecimal 43 is the internal representation for the letter C. When this instruction is executed, the processor fetches the first instruction byte and determines that it must fetch one more byte. The processor fetches the next byte into one of its internal registers and then performs the compare operation.

Notice that the names of the immediate instructions indicate that they use immediate data. Thus, the name of an add instruction is ADD; the name of an add immediate instruction is ADI.

All but two of the immediate instructions use the accumulator as an implied operand, as in the CPI instruction shown previously. The MVI (move immediate) instruction can move its immediate data to any of the working registers including the accumulator or to memory. Thus, the instruction MVI D, OFFH moves the hexadecimal value FF to the D register.

The LXI instruction (load register pair immediate) is even more unusual in that its immediate data is a 16-bit value. This instruction is commonly used to load addresses into a register pair. As mentioned previously, your program must initialize the stack pointer; LXI is the instruction most commonly used for this purpose. For example, the instruction LXI SP,30FFH loads the stack pointer with the hexadecimal value 30FF.

1.4.4 Direct Addressing

Jump instructions include a 16-bit address as part of the instruction. For example, the instruction JMP 1000H causes a jump to the hexadecimal address 1000 by replacing the current contents of the program counter with the new value 1000H.

Instructions that include a direct address require three bytes of storage: one for the instruction code, and two for the 16-bit address.

1.4.5 Register Indirect Addressing

Register indirect instructions reference memory via a register pair. Thus, the instruction MOV M, C moves the contents of the C register into the memory address stored in the H and L register pair. The instruction LDAX B loads the accumulator with the byte of data specified by the address in the B and C register pair.

1.4.6 Combined Addressing Modes

Some instructions use a combination of addressing modes. A CALL instruction, for example, combines direct addressing and registers indirect addressing. The direct address in a CALL

instruction specifies the address of the desired subroutine; the register indirect address is the stack pointer. The CALL instruction pushes the current contents of the program counter into the memory location specified by the stack pointer.

1.4.7 Timing Effects of Addressing Modes

Addressing modes affect both the amount of time required for executing an instruction and the amount of memory required for its storage. For example, instructions that use implied or register addressing, execute very quickly since they deal directly with the processor's hardware or with data already present in hardware registers. Most important, however, is that the entire instruction can be fetched with a single memory access. The number of memory accesses required is the single greatest factor in determining execution timing. More memory accesses therefore require more execution time. A CALL instruction, for example, requires five memory accesses: three to access the entire instruction and two more to push the contents of the program counter onto the stack.

The processor can access memory once during each processor cycle. Each cycle comprises a variable number of states. The length of a state depends on the clock frequency specified for your system, and may range from 480 nanoseconds to 2 microseconds. Thus, the timing for a four-state instruction may range from 1.920 microseconds through 8 microseconds. (The 8085 have a maximum clock frequency of 5 MHz and therefore a minimum state length of 200 nanoseconds.)



Task

Create the set of assembly language instruction for the simple microprocessor.

1.4.8 Decoding

The instruction decoder needs to turn each of the opcodes into a set of signals that drive the different components inside the microprocessor. Let's take the ADD instruction as an example and look at what it needs to do:

1. During the first clock cycle, we need to actually load the instruction. Therefore the instruction decoder needs to:
 - activate the tri-state buffer for the program counter
 - activate the RD line
 - activate the data-in tri-state buffer
 - latch the instruction into the instruction register
2. During the second clock cycle, the ADD instruction is decoded. It needs to do very little:
 - set the operation of the ALU to addition
 - latch the output of the ALU into the C register
3. During the third clock cycle, the program counter is incremented (in theory this could be overlapped into the second clock cycle).

Every instruction can be broken down as a set of sequenced operations like these that manipulate the components of the microprocessor in the proper order. Some instructions, like this ADD instruction, might take two or three clock cycles. Others might take five or six clock cycles.



Did u know?

The mnemonics assigned to the instructions are designed to indicate the function of the instruction.

Notes



Case Study

History and Evolution of Microprocessors

The invention of the transistor in 1947 was a significant development in the world of technology. It could perform the function of a large component used in a computer in the early years. Shockley, Brattain and Bardeen are credited with this invention and were awarded the Nobel Prize for the same. Soon it was found that the function of this large component was easily performed by a group of transistors arranged on a single platform. This platform, known as the integrated chip (IC), turned out to be a very crucial achievement and brought along a revolution in the use of computers. A person named Jack Kilby of Texas Instruments was honoured with the Nobel Prize for the invention of IC, which laid the foundation on which microprocessors were developed. At the same time, Robert Noyce of Fairchild made a parallel development in IC technology for which he was awarded the patent.

ICs proved beyond doubt that complex functions could be integrated on a single chip with a highly developed speed and storage capacity. Both Fairchild and Texas Instruments began the manufacture of commercial ICs in 1961. Later, complex developments in the IC led to the addition of more complex functions on a single chip. The stage was set for a single controlling circuit for all the computer functions. Finally, Intel corporation's Ted Hoff and Frederico Fagin were credited with the design of the first microprocessor.

The work on this project began with an order from a Japanese calculator company Busicom to Intel, for building some chips for it. Hoff felt that the design could integrate a number of functions on a single chip making it feasible for providing the required functionality. This led to the design of Intel 4004, the world's first microprocessor. The next in line was the 8-bit 8008 microprocessor. It was developed by Intel in 1972 to perform complex functions in harmony with the 4004.

This was the beginning of a new era in computer applications. The use of mainframes and huge computers was scaled down to a much smaller device that was affordable to many. Earlier, their use was limited to large organizations and universities. With the advent of microprocessors, the use of computers trickled down to the common man. The next processor in line was Intel's 8080 with an 8-bit data bus and a 16-bit address bus. This was amongst the most popular microprocessors of all time.

Very soon, the Motorola corporation developed its own 6800 in competition with the Intel's 8080. Fagin left Intel and formed his own firm Zilog. It launched a new microprocessor Z80 in 1980 that was far superior to the previous two versions. Similarly, a break off from Motorola prompted the design of 6502, a derivative of the 6800. Such attempts continued with some modifications in the base structure.

The use of microprocessors was limited to task-based operations specifically required for company projects such as the automobile sector. The concept of a 'personal computer' was still a distant dream for the world and microprocessors were yet to come into personal use. The 16-bit microprocessors started becoming a commercial sell-out in the 1980s with the first popular one being the TMS9900 of Texas Instruments.

Intel developed the 8086 which still serves as the base model for all latest advancements in the microprocessor family. It was largely a complete processor integrating all the required features in it. 68000 by Motorola was one of the first microprocessors to develop the concept of microcoding in its instruction set. They were further developed to 32 bit architectures. Similarly, many players like Zilog, IBM and Apple were successful in getting their own products in the market. However, Intel had a commanding position in the market right through the microprocessor era.

Contd...

Notes

The 1990s saw a large-scale application of microprocessors in the personal computer applications developed by the newly formed Apple, IBM and Microsoft corporation. It witnessed a revolution in the use of computers, which by then was a household entity.

This growth was complemented by a highly sophisticated development in the commercial use of microprocessors. In 1993, Intel brought out its 'Pentium Processor' which is one of the most popular processors in use till date. It was followed by a series of excellent processors of the Pentium family, leading into the 21st century. The latest one in commercial use is the Pentium Dual Core technology and the Xeon processor. They have opened up a whole new world of diverse applications. Supercomputers have become common, owing to this amazing development in microprocessors.

Questions:

1. Give the inventing year of transistor in the history of microprocessor.
2. Which company developed the first microprocessors?

1.5 Summary

- A microprocessor is designed to perform arithmetic and logic operations that make use of small number-holding areas called *registers*.
- Microprocessor is a silicon chip that contains a CPU.
- Bandwidth is the number of bits processed in a single instruction.
- The speed of the microprocessor is measured in MHz or GHz.
- Pentium 4 is the fastest type of the Intel's processor that contains 125,000,000 transistors and operates at the speed of 3.6 GHz.
- Microcomputers are designed to be used by individuals, whether in the form of PCs, workstations or notebook computers.
- Simple applications use only one ROM chip; the desktop calculator used four.
- One of the first users of the 8008 was Seiko in Japan for a sophisticated scientific calculator.

1.6 Keywords

Bandwidth: The number of bits processed in a single instruction.

BIOS: The basic input/output system that comes with the computer as part of its memory.

EPROM: Electrically programmed ROM's was significant in allowing customers to experiment with their software.

Instruction set: The set of instructions that the microprocessor can execute.

The gate to pin ratio: Optimization consists of maximizing the number of gates inside compared to the number of pins outside.



Lab Exercise

1. How do instructions run in the ROM? Explain.
2. Give some basic code of assembly language.

Notes

1.7 Self-Assessment Questions

1. Most of the instructions that use register addressing deal with 8-bit values.
(a) 4-bit (b) 16-bit
(c) 8-bit (d) None of these
2. An can translate the words into their bit patterns very easily.
(a) assembler (b) transistors
(c) processor (d) microprocessor
3. The first microprocessor was (early 1970's) used in calculators.
(a) Intel 4040 (b) Intel 4400
(c) Intel 8080 (d) Intel 4004
4. A microprocessor is a computer processor on a
(a) minichip (b) microchip
(c) minicomputer (d) None of these
5. When your computer is turned on, the gets the first instruction from the basic input/output system.
(a) microprocessor (b) microcomputer
(c) microchip (d) None of these
6. Our program tells the CPU to perform what is called an
(a) segment register (b) processor
(c) interrupt (d) None of these

1.8 Review Questions

1. Define microprocessor in brief and also discuss its important areas.
2. Explain about microcomputers and its functions.
3. Explain in brief the fields in which microprocessors have brought changes.
4. Discuss in brief about Intel 8008 Microprocessor.
5. Explain microprocessor instruction set in brief.
6. What are the methods of addressing of Instructions?
7. Describe the instructions that affect the Stack and/or Stack Pointer.
8. What is assembly language? Discuss with the help of example.
9. Define decoding and its need.
10. What are the different types of chip? Discuss each of them.

Answers for Self-Assessment Questions

1. (c) 2. (a) 3. (d) 4. (b) 5. (a) 6. (c)

1.9 Further Reading

Notes



Books

Microprocessor 8085 Lab Manual, by: G.T. Swamy



Online link

<http://www.webopedia.com/TERM/M/microprocessor.html>.

<http://searchcio-midmarkessst.techtarget.com/definition/microprocessor>.

Unit 2: Introduction to Assembly Language

CONTENTS

Objectives

Introduction

2.1 Assembly Language

2.1.1 Running the Program

2.1.2 Opcodes and Operands

2.1.3 Labels

2.1.4 Comments

2.1.5 Pseudo-ops (Assembler Directives)

2.2 The Assembly Process

2.3 Assembly Language Statements

2.4 Computer Languages

2.4.1 Application Software Classification

2.5 Summary

2.6 Keywords

2.7 Self-Assessment Questions

2.8 Review Questions

2.9 Further Reading

Objectives

After studying this unit, you will be able to understand the following:

- Explain Assembly Language
- Describe about Assembler Directives
- Discuss about Assessment Directives
- Explain the Assembly Process and Its Types
- Describe about the Assembly Language Statements
- Describe about Computer Languages
- Application Software Classification

Introduction

This is a brief introduction to assembly language. Assembly language is the most basic programming language available for any processor. With assembly language, a programmer works only with operations implemented directly on the physical CPU. Assembly language lacks high-level conveniences such as variables and functions, and it is not portable between various families of processors. Nevertheless, assembly language is the most powerful computer programming language available, and it gives programmers the insight required to write effective code in high-level languages. Learning assembly language is well worth the time and effort of every serious programmer.

Before we can explore the process of writing computer programs, we have to go back to the basics and learn exactly what a computer is and how it works. Every computer, no matter how simple or complex, has at its heart exactly two things: a CPU and some memory. Together, these two things are what make it possible for your computer to run programs.

On the most basic level, a computer program is nothing more than a collection of numbers stored in memory. Different numbers tell the CPU to do different things. The CPU reads the numbers one at a time, decodes them, and does what the numbers say. For example, if the CPU reads the number 64 as part of a program, it will add 1 to the number stored in a special location called AX. If the CPU reads the number 146, it will swap the number stored in AX with the number stored in another location called BX. By combining many simple operations such as these into a program, a programmer can make the computer perform many incredible things.

As an example, here are the numbers of a simple computer program: 184, 0, 184, 142, 216, 198, 6, 158, 15, 36, 205, 32.

2.1 Assembly Language

Although the numbers of the above program make perfect sense to a computer, they are about as clear as mud to a human. Who would have guessed that they put a dollar sign on the screen? Clearly, entering numbers by hand is a lousy way to write a program.

It doesn't have to be this way, though. A long time ago, someone came up with the idea that computer programs could be written using words instead of numbers. A special program called an *assembler* would then take the programmer's words and convert them to numbers that the computer could understand. This new method, called writing a program in *assembly language*, saved programmers thousands of hours, since they no longer had to look up hard-to-remember numbers in the backs of programming books, but could use simple words instead.

The program above, written in assembly language, looks like this:

```
MOV     AX,      47104
MOV     DS,      AX
MOV     [3998],  36
        INT      32
```

When an assembler reads this sample program, it converts each line of code into one CPU-level instruction. This program uses two types of instructions, MOV and INT. On Intel processors, the MOV instruction moves data around, while the INT instruction transfers processor control to the device drivers or operating system.

The program still isn't quite clear, but it is much easier to understand than it was before. The first instruction, MOV AX, 47104, tells the computer to copy the number 47104 into the location AX. The next instruction, MOV DS, AX, tells the computer to copy the number in AX into the location DS. The next instruction, MOV [3998], 36 tells the computer to put the number 36 into memory location 3998. Finally, INT 32 exits the program by returning to the operating system.

Before we go on, we would like to explain just how this program works. Inside the CPU are a number of locations, called registers, which can store a number. Some registers, such as AX, are general purpose, and don't do anything special. Other registers, such as DS, control the way the CPU works. DS just happens to be a *segment register*, and is used to pick which area of memory the CPU can write to. In our program, we put the number 47104 into DS, which tells the CPU to access the memory on the video card. The next thing our program does is to put the number 36 into location 3998 of the video card's memory. Since 36 is the code for the dollar sign, and 3998 is the memory location of the bottom right-hand corner of the screen, a dollar sign shows up on the screen a few microseconds later. Finally, our program tells the CPU to perform what is called an

Notes

interrupt. An interrupt is used to stop one program and execute another in its place. In our case, we want interrupt 32, which ends our program and goes back to MS-DOS, or whatever other program was used to start our program.

2.1.1 Running the Program

Let's go ahead and run this program. First, be sure to print these instructions out, since you will need to refer to them as we go on. Next, click on your start menu, and run the program called MS-DOS Prompt. A black screen with white text should appear. We are now in MS-DOS, the way computers used to be 20 years ago. MS-DOS was before the days of the mouse, so you must type commands on the keyboard to make the computer do things.

First, we want you to type the word *debug*, and press enter. The cursor should move down a line, and you should see the Debug prompt, which is a simple dash. We are now in a program called *Debug*. Debug is a powerful utility that lets you directly access the registers and memory of your computer for various purposes. In our case, we want to enter our program into memory and run it, so we'll use Debug's *a* command, for assemble. Go ahead and type *a100* now. The cursor will move down another line, and you will see something like *1073:0100*. This is the memory location we are going to enter assembly language instructions at. The first number is the segment, and the second number is the memory location within the segment. Your Debug program will probably pick a different segment for your program than mine did, so don't worry if it's different. Another thing to note is that Debug only understands hexadecimal numbers, which are a sort of computer shorthand. Hexadecimal numbers sometimes contain letters as well as well as digits.

Let's go ahead and enter our program now. Type each of the instructions below into Debug exactly as they appear, and press enter after each one. When you finish entering the last instruction, press enter twice to tell Debug that we are done entering instructions.

```
mov ax,B800
mov ds,ax
mov byte[0F9E],24
int 20
```

As you can see, we have converted all the numbers into hexadecimal, and have made a few other changes so Debug can understand what's going on. If you make a mistake while entering the above program, press enter twice, type *a100*, and start entering instructions again at the beginning of the program.

Once you have entered the program, you can go ahead and run it. Simply type *g* for *go* and press enter when you are ready to start the program.

Let's get back to Windows now. Go ahead and type *q* to get out of Debug. Now, type *exit* to get out of MS-DOS. You should now be back in Windows.

2.1.2 Opcodes and Operands

Two of the parts (OPCODE and OPERANDS) are **mandatory**. An instruction must have an OPCODE (the thing the instruction is to do), and the appropriate number of operands (the things it is supposed to do it to). The OPCODE is a symbolic name for the opcode of the corresponding LC-3b instruction. The idea is that it is easier to remember an operation by the symbolic name *ADD*, *AND*, or *LDW* than by the four-bit quantity *0001*, *0101*, or *0110*. The number of operands depends on the operation being performed. For example, the *ADD* instruction (line 0E) requires three operands (two sources to obtain the numbers to be added, and one destination to designate where the result is to be placed). All three operands must be explicitly identified in the instruction.

AGAIN ADD R3, R3, R2**Notes**

The operands to be added are obtained from register 2 and from register 3. The result is to be placed in register 3. We represent each of the registers 0 through 7 as R0, R2 - R7.

The LEA instruction (line 06) requires two operands (the memory location whose address is to be read) and the destination register which is to contain that address after the instruction completes execution. We will see momentarily that memory locations will be given symbolic addresses called labels. In this case, the location whose address is to be read is given the label NUMBER. The destination into which that address is to be loaded is register 2.

LEA R2, NUMBER

As we discussed in class, operands can be obtained from registers, from memory, or they may be literal (i.e. immediate) values in the instruction. In the case of register operands, the registers are explicitly represented (such as R2 and R3 in line 0C). In the case of memory operands, the symbolic name of the memory location is explicitly represented (such as NUMBER in line 06 and SIX in line 08). In the case of immediate operands, the actual value is explicitly represented (such as the value 0 in line 0A).

AND R3, R3; Clear R3

It will contain the product.

A literal value must contain a symbol identifying the representation base of the number. We use for decimal, x for hexadecimal, and b for binary. Sometimes there is no ambiguity, such as in the case 3F0A, which is a hex number. Nonetheless, we write it as x3F0A. Sometimes there is ambiguity, such as in the case 1000. X1000 represents the decimal number 4096, b1000 represents the decimal number 8, and 1000 represents the decimal number 1000.



Did u know?

1. Inside the CPU are a number of locations, called registers, which can store a number.
2. We use for decimal, x for hexadecimal, and b for binary.

2.1.3 Labels

Labels are symbolic names which are used to identify memory locations that are referred to explicitly in the program. In LC-3b assembly language, a label consists of from one to 20 alphanumeric characters (i.e. a capital or lower case letter of the alphabet, or a decimal digit), starting with a letter of the alphabet. Now, Under21, R2D2, and C3PO are all examples of possible LC-3b assembly language labels.

There are two reasons for explicitly referring to a memory location:

1. The location contains the target of a branch instruction
2. The location contains a value that is loaded or stored.

The location AGAIN is specifically referenced by the branch instruction in line 10.

BRP AGAIN

If the result of ADD R, R1 is positive (as evidenced by the P condition code being set), then the program branches to the location explicitly referenced as AGAIN to perform iteration.

The location number is specifically referenced by the LEA instruction in line 06. The value stored in the memory location explicitly referenced as number is loaded into R2. If a location in the program is not explicitly referenced, then there is no need to give it a label.

Notes



Caution

END does not stop the program during execution. In fact, .END does not even exist at the time of execution. It is simply a delimiter; it marks the end of the source program.


2.1.4 Comments

Comments are messages intended only for human consumption. They have no effect on the translation process and indeed are not acted on by the LC-3b Assembler. They are identified in the program by semicolons. A semicolon signifies that the rest of the line is a comment and is to be ignored by the assembler. If the semicolon is the first non-blank character on the line, the entire line is ignored. If the semicolon follows the operands of an instruction, then only the comment is ignored by the assembler.

The purpose of comments is to make the program more comprehensible to the human reader. They help explain a nonintuitive aspect of an instruction or a set of instructions. In line 0A, the comment “Clear R3; it will contain the product” lets the reader know that the instruction on line 0A is initializing R3 prior to accumulating the product of the two numbers. While the purpose of line 0A may be obvious to the programmer today, it may not be the case two years from now, after the programmer has written an additional 30,000 lines of code and cannot remember why he/she wrote AND R, R3, 0. It may also be the case that two years from now, the programmer no longer works for the company and the company needs to modify the program in response to a product update. If the task is assigned to someone who has never seen the code before, comments go a long way to helping comprehension.

It is important to make comments that provide additional insight and not just restate the obvious. There are two reasons for this. First, comments that restate the obvious are a waste of everyone’s time. Second, they tend to obscure the comments that say something important because they add clutter to the program. For example, in line 0F, the comment “Decrement R1” would be a bad idea. It would provide no additional insight to the instruction, and it would add clutter to the page.

Another purpose of comments, and also the judicious use of extra blank spaces to a line, is to make the visual presentation of a program easier to understand. So, for example, comments are used to separate pieces of the program from each other to make the program more readable. That is, lines of code that work together to compute a single result are placed on successive lines, while pieces of a program that produce separate results are separated from each other. For example, note that lines 0E through 10 are separated from the rest of the code by lines 0D and 11. There is nothing on lines 0D and 11 other than the semicolons. Extra spaces that are ignored by the assembler provide an opportunity to align elements of a program for easier readability. For example, all the opcodes start in the same column on the page.



Task Prepare a program and use some of the instructions.

2.1.5 Pseudo-ops (Assembler Directives)

The LC-3b assembler is a program that takes as input a string of characters representing a computer program written in LC-3b assembly language, and translates it into a program in the ISA of the LC-3b. Pseudo-ops are helpful to the assembler in performing that task. Actually, a more formal name for a pseudo-op is assembler directive. They are called pseudo-ops because they do not refer to operations that will be performed by the program during execution. Rather, the pseudo-op is strictly a message to the assembler to help the assembler in the assembly process. Once the assembler handles the message, the pseudo-op is discarded. The LC-3b assembler contains five pseudo-ops:

.ORIG, .FILL, .BLKW, .STRINGZ, and .END. All are easily recognizable by the dot as their first character.

.ORIG

.ORIG tells the assembler where in memory to place the LC-3b program. In line 05, .ORIG x3050 says, start with location x3050. As a result, the LEA R2, NUMBER instruction will be put in location x3050.

.FILL

.FILL tells the assembler to set aside the next location in the program and initialize it with the value of the operand. In line 15, the ninth location in the resultant LC-3b program is initialized to the value x0006.

.BLKW

.BLKW tells the assembler to set aside some number of sequential memory locations (i.e. a Block of Words) in the program. The actual number is the operand of the .BLKW pseudo-op. In line 11, the pseudo-op instructs the assembler to set aside one location in memory (and also to label it NUMBER, incidentally).

The pseudo-op .BLKW is particularly useful when the actual value of the operand is not yet known. For example, one might want to set aside a location in memory for storing a character input from a keyboard. It will not be until the program is run that we will know the identity of that keystroke.

The argument is a sequence of n characters, inside double quotation marks. The first n 1 bytes of memory are initialized with the ASCII codes of the corresponding characters in the string, followed by x00. A final byte x00 is added if necessary to end the string on a word boundary. The n 1st character (x00) provides a convenient sentinel for processing the string of ASCII codes.

For example, the code fragment

```
.ORIG x3010
HELLO .STRINGZ "Hello, World!"
```


would result in the assembler initializing locations x3010 through x301D to the following values:

```
x3010: x48
x3011: x65
x3012: x6C
x3013: x6C
x3014: x6F
x3015: x2C
x3016: x20
x3017: x57
x3018: x6F
x3019: x72
x301A: x6C
x301B: x64
x301C: x21
x301D: x00
```


Notes

.END

.END tells the assembler where the program ends. Any characters that come after .END will not be utilized by the assembler.



Notes .END does not stop the program during execution. In fact, .END does not even exist at the time of execution. It is simply a delimiter it marks the end of the source program.



Caution Sometimes there is no ambiguity, such as in the case 3F0A, which is a hex number, then, we write it as x3F0A.

2.2 The Assembly Process

Before an LC-3b assembly language program can be executed, it must first be translated into a machine language program, that is, one in which each instruction is in the LC-3b ISA. It is the job of the LC-3b assembler to perform that translation.

A Two-Pass Process

In this section, we will see how the assembler goes through the process of translating an assembly language program into a machine language program. You remember that there is in general a one-to-one correspondence between instructions in an assembly language program and instructions in the final machine language program. We could attempt to perform this translation in one pass through the assembly language program, the assembler discards lines 01 to 09, since they contain only comments. Comments are strictly for human consumption; they have no bearing on the translation process. The assembler then moves on to line 0A. Line 0A is a pseudo-op; it tells the assembler that the machine language program is to start a location x3000. The assembler then moves on to line 0B, which it can easily translate into LC-3b machine code. At this point, we have

X3000: 0101010010100000

The LC-3b assembler moves on to translate the next instruction (line 0C). Unfortunately, it is unable to do so, since it does not know the meaning of the symbolic address, PTR. At this point the assembler is stuck, and the assembly process fails.

To prevent the above problem from occurring, the assembly process is done in two complete passes (from beginning to .END) through the entire assembly language program. The objective of the first pass is to identify the actual binary addresses corresponding to the symbolic names (or labels). This set of correspondences is known as the *symbol table*. In pass one, we construct the symbol table. In pass two, we translate the individual assembly language instructions into their corresponding machine language instructions.

Thus, when the assembler examines line 0C for the purpose of translating LEA R3, PTR during the second pass, it already knows the correspondence between PTR and x3028 (from the first pass). Thus it can easily translate line 0C to x3002: 1110011000010011

The problem of not knowing the 16-bit address corresponding to PTR no longer exists.

The First Pass: Creating the Symbol Table

For our purposes, the symbol table is simply a correspondence of symbolic names with their 16-bit memory addresses. We obtain these correspondences by passing through the assembly language program once, noting which instruction is assigned to which address, and identifying each label with the address of its assigned entry. Recall that we provide labels in those cases

where we have to refer to a location, either because it is the target of a branch instruction or because it contains data that must be loaded or stored. Consequently, if we have not made any programming mistakes, and if we identify all the labels, we will have identified all the symbolic addresses used in the program.

The above paragraph assumes that our entire program exists between our .ORIG and .END pseudo-ops:

The assembler examines each instruction in sequence, and increments the LC once for each assembly language instruction. If the instruction examined contains a label, a symbol table entry is made for that label, specifying the current contents of LC as its address. The first pass terminates when the .END instruction is encountered. The first instruction that has a label is at line 13. Since it is the sixth instruction in the program and the LC at that point contains x300A, a symbol table entry is constructed thus:

Symbol	Address
TEST	x300A

The second instruction that has a label is at line 20. At this point, the LC has been incremented to x3018. Thus a symbol table entry is constructed, as follows:

Symbol	Address
GETCHAR	x3018

At the conclusion of the first pass, the symbol table has the following entries:

Symbol	Address
TEST	x300A
GETCHAR	x3018
OUTPUT	x301E
ASCII	x3028
PTR	x302A

The Second Pass: Generating the Machine Language Program

The second pass consists of going through the assembly language program a second time, line by line, this time with the help of the symbol table. At each line, the assembly language instruction is translated into an LC-3b machine language instruction.

Starting again at the top, the assembler again discards lines 01 through 09 because they contain only comments. Line 0A is the .ORIG pseudo-op, which the assembler uses to initialize LC to x3000. The assembler moves on to line 0B, and produces the machine language instruction 0101010010100000. Then the assembler moves on to line 0C.

This time, when the assembler gets to line 0C, it can completely assemble the instruction since it knows that PTR corresponds to x302A. The instruction is LEA, which has an opcode encoding of 1110. The Destination register (DR) is R3, that is, 011.

PCoffset is computed as follows: We know that PTR is the label for address x302A, and that the incremented PC is LC+2, in this case x3004. Since PTR (x302A) must be the sum of the incremented PC (x3004) and twice the sign-extended PCoffset (since the offset is in words and memory is byte-addressable), PCoffset must be x0013 by putting Address Binary.



In order to use the LEA instruction, it is necessary that the source of the load, in this case the address whose label is PTR, is not more than +512 or -510 memory locations from the LEA instruction itself.

Notes

If the address of PTR had been greater than LC+2 +510 or less than LC+2 -512, then the offset would not fit in bits [8:0] of the instruction. In such a case, an assembly error would have occurred, preventing the assembly process from completing successfully. Fortunately, PTR is close enough to the LEA instruction, so the instruction assembled correctly.

The second pass continues. At each step, the LC is incremented and the location specified by LC is assigned the translated LC-3b instruction or, in the case of .FILL, the value specified. When the second pass encounters the .END instruction, assembly terminates.

That process was, on a good day, merely tedious. Fortunately, you do not have to do it for a living; the LC-3b assembler does that. And, since you now know LC154 3b assembly language, there is no need to program in machine language. Now we can write our programs symbolically in LC-3b assembly language and invoke the LC- 3b assembler to create the machine language versions that can execute on an LC-3b computer.

2.3 Assembly Language Statements

Assembly language statements in a source file use the following format:

```
{Label} {Mnemonic {Operand}} {Comment}
```

Each entity above is a field. The four fields above are the label field, the mnemonic field, the operand field, and the comment field.

The label field is (usually) an optional field containing a symbolic label for the current statement. Labels are used in assembly language, just as in HLLs, to mark lines as the targets of GOTOs (jumps). You can also specify variable names, procedure names, and other entities using symbolic labels. Most of the time the label field is optional, meaning a label need be present only if you want a label on that particular line. Some mnemonics, however, require a label, others do not allow one. In general, you should always begin your labels in column one (this makes your programs easier to read).

A mnemonic is an instruction name (e.g. move, add, etc.). The word mnemonic means memory aid. Move is much easier to remember than the binary equivalent of the mov instruction! The braces denote that this item is optional. Note, however, that you cannot have an operand without a mnemonic.

The mnemonic field contains an assembler instruction. Instructions are divided into three classes: 80x86 machine instructions, assembler directives, and pseudo-opcodes.

Assembler directives are special instructions that provide information to the assembler but do not generate any code. Examples include the segment directive, equal, assume, and end. These mnemonics are not valid 80x86 instructions. They are messages to the assembler, nothing else.

A pseudo-opcodes is a message to the assembler, just like an assembler directive, however a pseudo-opcodes will emit object code bytes. Examples of pseudo-opcodes include byte, word, dword, qword, and bytes. These instructions emit the bytes of data specified by their operands but they are not true 80X86 machine instructions.

The operand field contains the operands, or parameters, for the instruction specified in the mnemonic field. Operands never appear on lines by themselves. The type and number of operands (zero, one, two, or more) depend entirely on the specific instruction.

The comment field allows you to annotate each line of source code in your program. Note that the comment field always begins with a semicolon. When the assembler is processing a line of text, it completely ignores everything on the source line following a semicolon.

Each assembly language statement appears on its own line in the source file. You cannot have multiple assembly language statements on a single line. On the other hand, since all the fields in an assembly language statement are optional, blank lines are fine.

Notes

You can use blank lines anywhere in your source file. Blank lines are useful for spacing out certain sections of code, making them easier to read.

The Microsoft Macro Assembler is a free form assembler. The various fields of an assembly language statement may appear in any column (as long as they appear in the proper order). Any number of spaces or tabs can separate the various fields in the statement. To the assembler, the following two code sequences are identical:

```
mov ax, 0
mov bx, ax
add ax, dx
mov cx, ax
mov ax, 0
mov bx, ax
add ax, dx
mov cx, ax
```

1. Unless, of course, the semicolon appears inside a string constant: Directives and Pseudo

The first code sequence is much easier to read than the Pseudo code, with respect to readability, the judicious use of spacing within your program can make all the difference in the world.

Placing the labels in column one, the mnemonics in column 17 (two tabstops), the operand field in column 25 (the third tabstop), and the comments out around column 41 or 49 (five or six tabstops) produces the best looking listings. Assembly language programs are hard enough to read as it is. Formatting your listings to help make them easier to read will make them much easier to maintain.

You may have a comment on the line by itself. In such a case, place the semicolon in column one and use the entire line for the comment, examples:

; The following section of code positions the cursor to the upper

; left hand position on the screen:

```
mov X, 0
mov Y, 0
```

; Now clear from the current cursor position to the end of the

; screen to clear the video display:

; etc.

2.4 Computer Languages

The term computer language includes a wide variety of languages used to communicate with computers. It is broader than the more commonly-used term programming language. Programming languages are a subset of computer languages. For example, HTML is a markup language and a computer language, but it is not traditionally considered a programming language. Machine code is a computer language. It can technically be used for programming, and has been (e.g. the original boots trapper for Altair BASIC), though most would not consider it a programming language.

Computer languages can be divided into two groups: high-level languages and low-level languages. High-level languages are designed to be easier to use, more abstract, and more portable than low-level languages. Syntactically correct programs in some languages are then compiled to

Notes

low-level language and executed by the computer. Most modern software is written in a high-level language, compiled into object code, and then translated into machine instructions.

Computer languages could also be grouped based on other criteria. Another distinction could be made between human-readable and non-human-readable languages. Human-readable languages are designed to be used directly by humans to communicate with the computer. Non-human-readable languages, though they can often be partially understandable, are designed to be more compact and easily processed, sacrificing readability to meet these ends.

2.4.1 Application Software Classification

Application software falls into two general categories: horizontal applications and vertical applications. Horizontal Applications are the most popular and widespread in departments or companies. Vertical Applications are niche products, designed for a particular type of business or division in a company.

There are many types of application software:

- *An application suite* consists of multiple applications bundled together. They usually have related functions, features and user interfaces, and may be able to interact with each other, e.g. open each other's files. Business applications often come in suites, e.g. Microsoft Office, OpenOffice.org and Work, which bundle together a word processor, a spreadsheet, etc.; but suites exist for other purposes, e.g. graphics or music.
- *Enterprise software* addresses the needs of organization processes and data flow, often in a large distributed environment (Examples include financial systems, customer relationship management (CRM), systems and supply-chain management software). Note that Departmental Software is a sub-type of Enterprise Software with a focus on smaller organizations or groups within a large organization (Examples include Travel Expense Management and IT Helpdesk).
- *Enterprise infrastructure software* provides common capabilities needed to support enterprise software systems. (Examples include databases, email servers, and systems for managing networks and security.)
- *Information worker software* addresses the needs of individuals to create and manage information, often for individual projects within a department, in contrast to enterprise management. Examples include time management, resource management, documentation tools, analytical, and collaborative. Word processors, spreadsheets, email and blog clients, personal information system, and individual media editors may aid in multiple information worker tasks.
- *Content access software* is software used primarily to access content without editing, but may include software that allows for content editing. Such software addresses the needs of individuals and groups to consume digital entertainment and published digital content (Examples include Media Players, Web Browsers, Help browsers and Games).
- *Educational software* is related to content access software, but has the content and/or features adapted for use in by educators or students. For example, it may deliver evaluations (tests), track progress through material, or include collaborative capabilities.
- *Simulation software* is computer software for simulation of physical or abstract systems for research, training or entertainment purposes.
- *Media development software* addresses the needs of individuals who generate print and electronic media for others to consume, most often in a commercial or educational setting. This includes Graphic Art software, Desktop Publishing software, Multimedia Development software, HTML editors, Digital Animation editors, Digital Audio and Video composition, and many others.

Notes

- *Mobile applications* ("Mobile apps") run on hand-held devices such as smart phones, tablet computers, portable media players, personal digital assistants and enterprise digital assistants: see mobile application development.
- *Product engineering software* is used in developing hardware and software products. This includes computer aided design (CAD), computer aided engineering (CAE), computer language editing and compiling tools, Integrated Development Environments, and Application Programmer Interfaces.
- A command-line interface is one in which you type in commands to make the computer do something. You have to know the commands and what they do, and type them correctly. DOS and UNIX are examples of command-driven interfaces.
- A graphical user interface (GUI) is one in which you select command choices from various menus, buttons and icons using a mouse. It is a user-friendly interface. Microsoft Windows and Mac OS are both graphical user interfaces.
- A third party server side application that the user may choose to install in his or her account on a social media site or other Web 2.0 website, for example a Facebook app.

Applications can also be classified by computing platform.



Case Study

Quality in an Assembly Process

Situation

The valve assembly process of an automotive supplier had a rolled throughput yield of 75%.

Challenge

A low yield meant loss of production, high labour cost, and the inability of the company to deliver product on time to the customer.

Impact

The ability to resolve the issues of yield would improve the competitive position of the company by reducing material and labour costs.

Simons-White Provided

Based on extensive statistical and Pareto analysis, it was decided to target the leading causes of failures with the valve assembly. The solutions resulted in adding a cleaning operation prior to the assembly process to ensure cleanliness of components and reduce resistance caused by friction in the valve.

A Design of Experiment was developed to identify the factors that contributed to leakage in the assembly. The results of the experiment enabled the team working on the project to establish settings for the component.

Results

Overall the yield improved to 93%, which delivered an average annual savings of ₹ 14100000.

Questions:

1. Explain the basic quality of assembly process.
2. Which type quality provided to Simons-White in assembly language?

2.5 Summary

- Every computer has at its heart exactly two things: a CPU and some memory.
- A computer program is nothing more than a collection of numbers stored in memory.
- Different numbers tell the CPU to do different things.
- Inside the CPU are a number of locations, called registers, which can store a number.
- An interrupt is used to stop one program and execute another in its place.
- Debug is a powerful utility that lets you directly access the registers and memory of your computer for various purposes.

2.6 Keywords

An assembler is a special program that takes the programmer's words and converts them to numbers that the computer could understand.

An interrupt is used to stop one program and execute another in its place.

Comments are messages intended only for human consumption. They have no effect on the translation process and indeed are not acted on by the LC-3b Assembler.

Labels: Memory locations will be given symbolic addresses called *labels*.

OPCODE: It is a symbolic name for the opcode of the corresponding LC-3b instruction.



Lab Exercise

1. Give the address of following symbols in assembly:
TEST, GETCHAR
2. Differentiate between Opcodes and Operands.

2.7 Self-Assessment Questions

1. A computer program is a collection of stored in memory.
(a) data (b) program
(c) numbers (d) alphabets
2. The CPU reads the numbers one at a time, them, and does what the numbers say.
(a) codes (b) decodes
(c) find (d) elaborates
3. is a powerful utility that lets you directly access the registers and memory of your computer for various purposes.
(a) Debug (b) Assemble
(c) Cursor (d) None of these
4. What is a symbolic name for the opcode of the corresponding LC-3b instruction?
(a) OPERANDS (b) OPCODE
(c) LDW (d) None of these

5. are symbolic names which are used to identify memory locations that are referred to explicitly in the program.
- (a) Labels (b) Alphanumeric
(c) LC-3b (d) None of these
6. The purpose of is to make the program more comprehensible to the human reader.
- (a) messages (b) opcodes
(c) comments (d) None of these

Notes

2.8 Review Questions

1. Give brief introduction to assembly language.
2. Explain the process of running the program.
3. Explain its importance of assembly language.
4. What are OPCODE and OPERANDS? Explain their importance.
5. What are Labels and what are their uses?
6. What are pseudo-ops? Explain types of pseudo-ops.
7. What do you understand by mnemonic? Also mention its importance.
8. Define the term computer language and discuss its groups.
9. Define Application software and its types.
10. Differentiate between High-level languages and Low-level languages.

Answers for Self-Assessment Questions

1. (c) 2. (b) 3. (a) 4. (b) 5. (a) 6. (c)

2.9 Further Reading



Books

Assembly Language by Kip R. Irvine

Online link

<http://www.laynetworks.com/assembly%20tutorials.htm>

Unit 3: Assembly Language Programming of 8085

CONTENTS

Objectives

Introduction

3.1 The 8085 Programming Model

3.1.1 Registers

3.1.2 Accumulator

3.1.3 Flags

3.1.4 Program Counter (PC)

3.1.5 Stack Pointer (SP)

3.2 The 8085 Addressing Modes

3.3 Instruction Set Classification

3.3.1 Data Transfer (Copy) Operations

3.3.2 Arithmetic Operations

3.3.3 Logical Operations

3.3.4 Branching Operations

3.3.5 Call, Return, and Restart

3.3.6 Machine Control Operations

3.4 Instruction Format

3.4.1 Instruction Word Size

3.4.2 One-Byte Instructions

3.4.3 Two-Byte Instructions

3.4.4 Three-Byte Instructions

3.4.5 Opcode Format

3.5 Sample Programs

3.6 Summary

3.7 Keywords

3.8 Self-Assessment Questions

3.9 Review Questions

3.10 Further Reading

Objectives

After studying this unit, you will be able to understand the following:

- Explain 8085 Programming Model
- Understand the 8085 Addressing Modes

- Describe Instruction Set Classification
- Discuss Instruction Format
- Understand Programming

Notes

Introduction

Intel 8085 microprocessor is the next generation of Intel 8080 CPU family. In addition to being faster than the 8080, the 8085 had the following enhancements:

- Intel 8085 had single 5 Volt power supply.
- Clock oscillator and system controller were integrated on the chip.
- The CPU included serial I/O port.
- Two new instructions were added to 8085 instruction set.

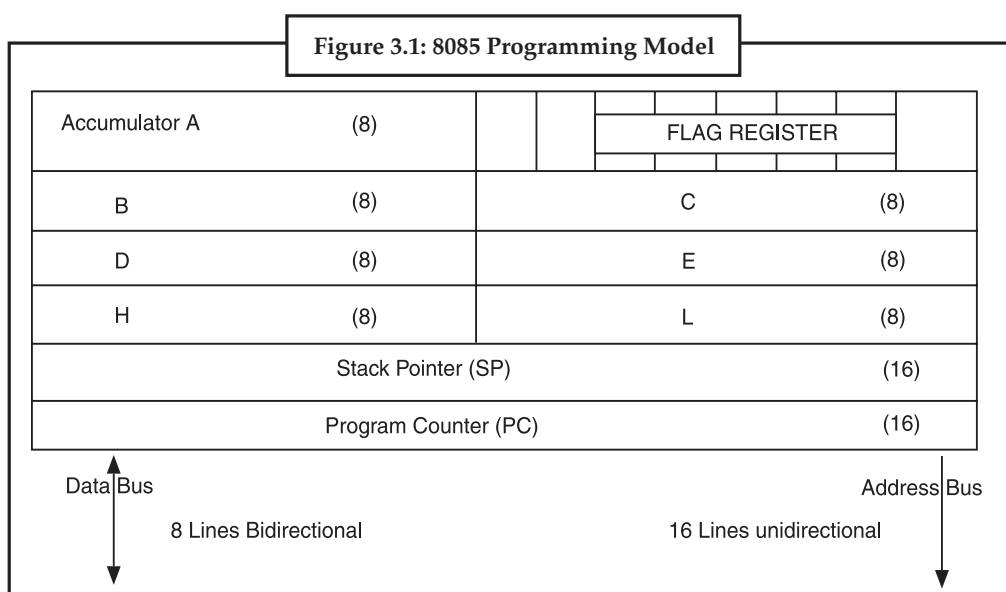
The CPU also included a few undocumented instructions. These instructions were supposed to be a part of the CPU instruction set, but at the last moment they were left undocumented because they were not compatible with forthcoming Intel 8086.

Unlike the other programming languages catalogued here, assembly language is not a single language, but rather a group of languages. Each processor family (and sometimes individual processors within a processor family) has its own assembly language.

In contrast to high level languages, data structures and program structures in assembly language are created by directly implementing them on the underlying hardware. So, instead of cataloguing the data structures and program structures that can be built (in assembly language you can build any structures you so desire, including new structures nobody else has ever created), we will compare and contrast the hardware capabilities of various processor families.

3.1 The 8085 Programming Model


The 8085 programming model includes six registers, one accumulator, and one flag register, as shown in Figure 3.1. In addition, it has two 16-bit registers: the stack pointer and the program counter. They are described briefly as follows:



Notes

3.1.1 Registers

The 8085 has six general-purpose registers to store 8-bit data; these are identified as B, C, D, E, H, and L as shown in the figure. They can be combined as register pairs — BC, DE, and HL – to perform some 16-bit operations. The programmer can use these registers to store or copy data into the registers by using data copy instructions.



Did u know?

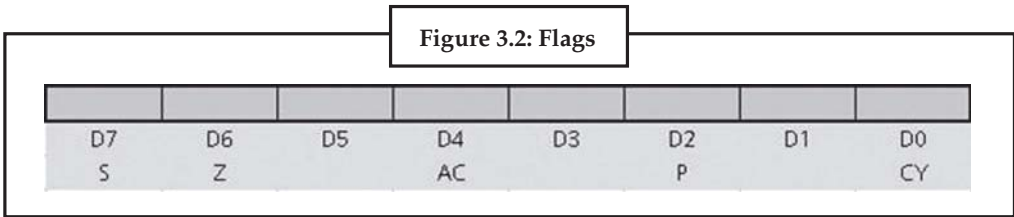
Any registers including memory can be incremented and decremented.

3.1.2 Accumulator

The accumulator is an 8-bit register that is a part of arithmetic/logic unit (ALU). This register is used to store 8-bit data and to perform arithmetic and logical operations. The result of an operation is stored in the accumulator. The accumulator is also identified as register A.

3.1.3 Flags

The ALU includes five flip-flops, which are set or reset after an operation according to data conditions of the result in the accumulator and other registers. They are called Zero (Z), Carry (CY), Sign (S), Parity (P), and Auxiliary Carry (AC) flags; their bit positions in the flag register are shown in the Figure 3.2 below. The most commonly used flags are Zero, Carry, and Sign. The microprocessor uses these flags to test data conditions.



For example, after an addition of two numbers, if the sum in the accumulator is larger than eight bits, the flip-flop uses to indicate a carry called the Carry flag (CY) is set to one. When an arithmetic operation results in zero, the flip-flop called the Zero (Z) flag is set to one. The first Figure shows an 8-bit register, called the flag register, adjacent to the accumulator. However, it is not used as a register; five bit positions out of eight are used to store the outputs of the five flip-flops. The flags are stored in the 8-bit register so that the programmer can examine these flags (data conditions) by accessing the register through an instruction.

These flags have critical importance in the decision-making process of the microprocessor. The conditions (set or reset) of the flags are tested through the software instructions. For example, the instruction JC (Jump on Carry) is implemented to change the sequence of a program when CY flag is set. The thorough understanding of flag is essential in writing assembly language programs.



Caution

The flags are affected according to the results.

3.1.4 Program Counter (PC)

This 16-bit register deals with sequencing the execution of instructions. This register is a memory pointer. Memory locations have 16-bit addresses, and that is why this is a 16-bit register.

The microprocessor uses this register to sequence the execution of the instructions. The function of the program counter is to point to the memory address from which the next byte is to be fetched. When a byte (machine code) is being fetched, the program counter is incremented by one to point to the next memory location.

3.1.5 Stack Pointer (SP)

Notes

The stack pointer is also a 16-bit register used as a memory pointer. It points to a memory location in R/W memory, called the stack. The beginning of the stack is defined by loading 16-bit address in the stack pointer.

This programming model will be used in subsequent tutorials to examine how these registers are affected after the execution of an instruction.

3.2 The 8085 Addressing Modes

The instructions MOV B, A or MVI A, 82H are to copy data from a source into a destination. In these instructions the source can be a register, an input port, or an 8-bit number (00H to FFH). Similarly, a destination can be a register or an output port. The sources and destination are operands. The various formats for specifying operands are called the ADDRESSING MODES. For 8085, they are:

1. Immediate addressing.
2. Register addressing.
3. Direct addressing.
4. Indirect addressing.

Immediate Addressing

Data is present in the instruction. Load the immediate data to the destination provided. Example: MVI R, data

Register Addressing

Data is provided through the registers. Example: MOV Rd, Rs

Direct Addressing

Used to accept data from outside devices to store in the accumulator or send the data stored in the accumulator to the outside device. Accept the data from the port 00H and store them into the accumulator or send the data from the accumulator to the port 01H. Example: IN 00H or OUT 01H

Indirect Addressing

This means that the Effective Address is calculated by the processor. And the content of the address (and the one following) is used to form a second address. The second address is where the data is stored. Note that this requires several memory accesses; two accesses to retrieve the 16-bit address and a further access (or accesses) to retrieve the data which is to be loaded into the register.



Task

Write a program to subtract two 16-bit numbers using 8085.

3.3 Instruction Set Classification

An instruction is a binary pattern designed inside a microprocessor to perform a specific function. The entire group of instructions, called the instruction set, determines what functions the microprocessor can perform. Instruction set determines what functions the microprocessor can perform. These instructions can be classified into the following five functional categories:

Notes

- 1. Data transfer (copy) operations
- 2. Arithmetic operations
- 3. Logical operations
- 4. Branching operations
- 5. Machine-control operations.

3.3.1 Data Transfer (Copy) Operations

This is one of the primary functions of the microprocessor. This group of instructions copies data from a location called a source to another location called a destination, without modifying the contents of the source. The source and destination may be any register, I/O or memory location. In technical manuals, the term data transfer is used for this copying function. However, the term transfer is misleading; it creates the impression that the contents of the source are destroyed when, in fact, the contents are retained without any modification. The various types of data transfer (copy) are listed below together with examples of each type:

Table 3.1: List of Data Transfer Operations

Types	Examples
1. Between Registers.	1. Copy the contents of the register B into register D.
2. Specific data byte to a register or a memory location.	2. Load registers B with the data byte 32H.
3. Between a memory location and a register.	3. From a memory location 2000H to register B.
4. Between an I/O device and the accumulator.	4. From an input keyboard to the accumulator.

Table 3.2: List of Data Transfer Operations

Opcode	Operand	Description
MOV	R _d , R _s	Move (copies a data byte) (a) It is 1-byte instruction (b) Copies data from source register R _s to destination register R _d .
MVI	R, 8-bit	Move Immediate (load a data byte directly) (a) It is a 2-byte instruction (b) Loads the 8-bits data of the second into the register specified.
OUT	8-bit port address	Output to Port (Send a data byte to output device) (a) It is a 2-byte instruction (b) Copies the contents of the Accumulation (A) to the output port specified in second byte.

Contd...

IN	8-bit port address	Input from Port (Read a data byte from a input device) (a) It is a 2-byte instruction. (b) Reads data from the input port specified in the second byte and loads into Accumulator.
HLT		Halt (a) It is 1-byte instruction. (b) Processor stops executing and enters wait state. (c) The address bus and data bus are placed in high impedance state. No register contents are effected.
NOP		No Operation (a) It is a 1-byte instruction. (b) No operation is performed. (c) Used to increase processing time or substitute in place of an instruction.

Notes

The symbols R, R_d and R_s represent any of the 8085 registers as A, B, D, E, H and L.

The two machine control operations: HLT and NOP are necessary to execute the program. NOP is generally used when an error occurs in a program and an instruction needs to be eliminated. It is more convenient to substitute NOP than to reassemble the whole program. When HLT opcode is used then address bus and data bus are placed in the high-impedance state or tri-state. No registers contents are affected by HLT.

As Table 3.2 states that the OUT and IN opcodes are 2-byte instructions the second byte is not defined. The question arises: what is the second byte in the instructions IN and OUT? The answer is the second byte is the I/O port address. Each I/O port is identified by a unique number. The second byte: 8-bits, i.e. 256 (2⁸) combination. Thus 256 input ports and 256 output ports N addresses from 00H to FFH can be connected to the system.

Key Points about Data Transfer (Copy) Operations:

1. Registers are used to load data directly or to save data bytes.
2. The destination register is modified whereas the source register is not affected.
3. Data copy instructions do not affect any of the status flags. This is because it involves two different registers or one register and one data byte. As only the Accumulator register can change or alter the status flag, all other registers B, C, D, E, H and L does not affect the flag register. Flags are affected by the operations in the ALU; therefore operations that take place outside the ALU do not affect flags.
4. The 8085 transfer's data from an input port to the Accumulator (IN) and from the Accumulator to an output port (OUT).
5. The OUT instruction cannot send data from any other register except the Accumulator.
6. The IN instruction can only load data into Accumulator.



In 8085 processor, data transfer instructions do not affect any flag.

Notes

3.3.2 Arithmetic Operations

These instructions perform arithmetic operations such as addition, subtraction, increment, and decrement.

Addition: Any 8-bit number or the contents of a register or the contents of a memory location can be added to the contents of the accumulator and the sum is stored in the accumulator. No two other 8-bit registers can be added directly (e.g. the contents of register B cannot be added directly to the contents of the register C). The instruction DAD is an exception; it adds 16-bit data directly in register pairs.

Subtraction: Any 8-bit number or the contents of a register or the contents of a memory location can be subtracted from the contents of the accumulator and the results stored in the accumulator. The subtraction is performed in 2's complement, and the results if negative, are expressed in 2's complement. No two other registers can be subtracted directly.

Increment/Decrement: The 8-bit contents of a register or a memory location can be incremented or decrement by 1. Similarly, the 16-bit contents of a register pair (such as BC) can be incremented or decrement by 1. These increment and decrement operations differ from addition and subtraction in an important way; i.e. they can be performed in any one of the registers or in a memory location.

The ADD, ADI, SUB, SUI instructions have following properties:

- (a) Implicitly assumes that the Accumulator is one of the operand and so the second byte is not passed in 2-byte instruction.
- (b) The result is stored in the Accumulator.
- (c) The contents of the operand register is not affected.
- (d) The result can alter all the flags.

R represents any of registers A, B, C, D, E, H, and L.

The INR, DCR instructions have following properties:

- (a) Both the instructions are 1-byte instruction.
- (b) Both affect the contents of the specified register.
- (c) Only CY flag is unaffected, all other flags are altered. This is because these instructions also involve other registers which cannot directly affect the status flag register. The status of the flags is determined by the result of an operation; in most instances, the result is in the accumulator. In increment (INR) operation, results can be in register other than the accumulator.

The ADD, ADI instructions perform addition of register or 8-bit with the contents of the Accumulator. If the sum is larger than 8-bits, it sets the carry flag.

Key Points about Arithmetic Operation:

1. In all arithmetic operations, one operand is the contents of the Accumulator.
2. The result of these operations is also stored in the Accumulator. This replaces the previous contents of Accumulator with current one.
3. The contents of source register are not changed.
4. If the sum is larger than 8-bit then the carry flag (CY) is set.
5. The subtraction is performed by 2's complement method.
6. If the result of subtraction is a negative number then the carry flag is set and the result is represented in the 2's complement form.
7. Carry flag is affected both in addition and subtraction. For subtraction CY is called Borrow flag.
8. Sign flag (S) is ignored, when unsigned arithmetic operations are done.

Notes

Table 3.3: Opcode Operation

Opcode	Operand	Description
ADD	R	Add (a) It is a 1-byte instruction (b) Adds the contents of register R to the contents of Accumulator
ADI	8-bit	Add Immediate (a) It is a 2-byte instruction (b) Adds the 8-bits (second byte) to the contents of the Accumulator
SUB	R	Subtract (a) It is a 1-byte instruction (b) Subtracts the contents of register R from the contents of the Accumulator
SUI	8-bit	Subtract Immediate (a) It is a 2-byte instruction (b) Subtract the 8-bit (second byte) from the contents of the Accumulator.
INR	R	Increment (a) It is a 1-byte instruction (b) Increases the contents of R by 1
DCR	R	Decrement (a) It is a 1-byte instruction (b) Decreases the contents of R by 1

9. The INR (increment) and DCR (decrement) uses only one of the registers. These do not affect CY, even the result is larger than 8-bit.

3.3.3 Logical Operations

As 8085 microprocessor is a programmable logic chip, it can also perform hard-wired logic with its instruction set. The logical operations performed by includes AND, OR, XOR (Exclusive OR) and NOT (Complement). The logical instructions also assume that the Accumulator is one of the operands and also the place where the result is stored. These operations also do not affect the contents of source register. Except the CMA instruction, all other affect Z, P, and S flags according to the result. The logic operations performed are listed in Table 3.4. These instructions perform various logical operations with the contents of the accumulator.

AND, OR Exclusive OR: Any 8-bit number, or the contents of a register, or of a memory location can be logically AND, ORed, or Exclusive-ORed with the contents of the accumulator. The results are stored in the accumulator.

Rotate: Each bit in the accumulator can be shifted either left or right to the next position.

Compare: Any 8-bit number or the contents of a register, or a memory location can be compared for equality, greater than or less than, with the contents of the accumulator.

Complement: The contents of the accumulator can be complemented. All 0s are replaced by 1s and all 1s are replaced by 0s.

Notes

Table 3.4: List of Logical Instructions

Opcode	Operand	Description
ADD	R	Add
ANA	R	Logical ADD with Accumulator (a) It is a 1-byte instruction (b) Logically AND the contents of Register R with the contents of Accumulator.
ANI	8-bit	AND Immediate with Accumulator (a) It is d 2-byte instruction (b) Logically AND the 8-bit with the contents of Accumulator.
ORA	R	Logical OR with Accumulator (a) It is a 1-byte instruction (b) Logically OR the contents of Register R with the contents of Accumulator.
ORI	8-bit	OR Immediate with Accumulator (a) It is a 2-byte instruction. (b) Logical OR the 8-bit with the contents of the Accumulator.
XRA	R	Logical Exclusive—OR with Accumulator (a) It is a 1-byte instruction. (b) Exclusive—ORs the contents of Register R with the contents of Accumulator.
XRI	8-bit	Exclusive—OR Immediate with Accumulator (a) It is a 2-byte instruction. (b) Exclusive—OR the 8-bit with the contents of the Accumulator.
CMA		Complement Accumulator (a) It is a 1-byte instruction. (b) Complements the contents of the Accumulator. (c) No flags are affected.

R represents any of registers *A, B, C, D, E, H* and *L*. In ANA, XRA, XRI and ANI instruction, CY is reset and AC is set. When logical operation instructions are executed bitwise operations are performed between the contents of Accumulator and contents of Register R or 8-bit data value.



Task Write a program by using INR and DCR opcode instruction.

Key Points about Logical Operations:

- 1. Logical operations are performed with the contents of the Accumulator and the result is also stored in the Accumulator.
- 2. The contents of Accumulator are replaced with current one when a logical operation is performed.
- 3. The contents of the source Register R is not altered during logical operations.
- 4. Logical operations cannot be performed directly using the contents of two registers.

- 5. Logical operations simulate eight 2-input gates or inverters.
- 6. The individual bits in the Accumulator are set or reset when logical operations are performed.
- 7. The Sign (S), Parity (P) and Zero (Z) flags are altered and Carry flag (CY) is reset when logical operations are performed.
- 8. NOT operation (CMA) does not affect any flags.

3.3.4 Branching Operations

The computer is based on the concept of flexibility and versatility. The results depend on computer’s ability to transfer control or branch of the operation and instructions in a program. This is not in a sequential order. This can be achieved through Branch Operation. They allow the microprocessor to change the sequence of program either unconditionally or under certain test conditions. Branch instructions instruct the microprocessor to break the sequential execution manner and go to a different memory location and continue executing machine codes in sequential manner from that new location. The address of the new memory location is either specified explicitly or supplied by the microprocessor or any hardware device. These instructions act on the program counter and it is possible to execute a block of instructions many times.

The jump instructions specify the memory location explicitly. This is a 3-byte instruction in which the first byte is opcode and 2nd and 3rd bytes are 16-bit memory address. Jump instructions can be of two categories:

- (1) Unconditional Jump
- (2) Conditional Jump

During the source of writing the program for a particular problem, if the program needs to jump to an instruction which appears in the later part of the program then instead of giving the address, a label or name can be given in the jump instruction. Later when that part of the program is written, the same label can be referred at the statement to which the execution has to jump.

3.3.4.1 Unconditional Jump Instructions

A jump instruction is used to break the normal sequential execution and branch to a different part of the program. This can be achieved by loading the address of the next out-of-sequence instruction into the program counter. This forces the processor to fetch the contents of this new location as its next instruction. The new address is usually specified in the jump instruction. When the jump instruction is executed, no flag is affected. There is only one unconditional jump instruction. The unconditional jump instruction enables the programmer to set up continuous loop, i.e. endless loop. The unconditional jump instruction is listed in Table 3.5.

Table 3.5: List of Unconditional Jump Instructions

Opcode	Operand	Description
JMP	16-bit	Jump (a) It is a 3-byte instruction. (b) Second and third bytes specify the 16-bit memory address. (c) The second byte specifies the low-order and the third byte specifies the high-order memory address.

Notes

3.3.4.2 Conditional Jump Instructions

Conditional jumps allow the processor to make decisions on certain conditions which are indicated by the flags. After every logic and arithmetic operation, flags are set or reset to reflect the status of the result. The conditional jump instructions check the flag status and make decisions about the sequence of the program. The 8085 has five flags, out of which auxiliary carry flag (AC) is used internally. The other four flags are used by the jump instructions. They are:

- (a) Carry flag (CY)
- (b) Zero flag (Z)
- (c) Sign flag (S)
- (d) Parity flag (P).

Two jump instructions are associated with each flag. The sequence of a program can be changed either – because the condition is present or the condition is absent. All conditional jump instructions are 3-byte instructions. The first byte is the opcode, the second byte specifies the low-order memory address and the third byte specifies the high-order memory address. The high-order memory address denotes page number and low-order memory address denotes the line number. The conditional jump instructions are stated in Table 3.6.

Table 3.6: List of Conditional Jump Instructions

Opcode	Operand	Description	Remarks
JC	16-bit	Jump on Carry	If result generates carryand CY = 1.
JNC	16-bit	Jump on No Carry	When CY = 0
JZ	16-bit	Jump on Zero	If result is zero and Z = 1
JNZ	16-bit	Jump on No Zero	When Z = 0
JP	16-bit	Jump on Plus	If (MSB) D ₇ = 0 and S = 0
JM	16-bit	Jump on Minus	If (MSB) D ₇ = 1 and S = 1
JPE	16-bit	Jump on Even Parity	When P = 1
JPO	16-bit	Jump on Odd Parity	When P = 0



Did u know?

Conditional jumps are an important aspect of the decision-making process in the programming. These instructions test for a certain condition (e.g. Zero or Carry flag) and alter the program sequence when the condition is met. In addition, the instruction set includes an instruction called unconditional jump.

3.3.5 Call, Return, and Restart

These instructions change the sequence of a program either by calling a subroutine or returning from a subroutine. The conditional Call and Return instructions also can test condition flags.

3.3.6 Machine Control Operations

These instructions control machine functions such as Halt, Interrupt, or do nothing.

The microprocessor operations related to data manipulation can be summarized in four functions:

1. Copying data
2. Performing arithmetic operations

3. Performing logical operations
4. Testing for a given condition and alerting the program sequence

Some important aspects of the instruction set are noted below:

1. In data transfer, the contents of the source are not destroyed; only the contents of the destination are changed. The data copy instructions do not affect the flags.
2. Arithmetic and Logical operations are performed with the contents of the accumulator, and the results are stored in the accumulator (with some expectations). The flags are affected according to the results.
3. Any register including the memory can be used for increment and decrement.
4. A program sequence can be changed either conditionally or by testing for a given data condition.



Did u know?

In copy inst the data in the source is not changed only the data in the destination Arithmetic and logic operations are performed in the accumulator and the results are stored in accumulator

3.4 Instruction Format

An **instruction** is a command to the microprocessor to perform a given task on a specified data. Each instruction has two parts: one is task to be performed, called the **operation code** (opcode), and the second is the data to be operated on, called the **operand**. The operand (or data) can be specified in various ways. It may include 8-bit (or 16-bit) data, an internal register, a memory location, or 8-bit (or 16-bit) address. In some instructions, the operand is implicit.

3.4.1 Instruction Word Size

The entire group of instructions, called the *instruction set*, determines functions the microprocessor can perform. The 8085 instruction set is classified into the following three groups according to word size:

1. One-word or 1-byte instructions
2. Two-word or 2-byte instructions
3. Three-word or 3-byte instructions

In the 8085, “byte” and “word” are synonymous because it is an 8-bit microprocessor. However, instructions are commonly referred to in terms of bytes rather than words.

3.4.2 One-Byte Instructions

A 1-byte instruction includes the opcode and operand in the same byte. Operand(s) are internal register and are coded into the instruction.

For example:

Task	Opcode	Operand	Binary Code	Hex Code
Copy the contents of the accumulator in the register C.	MOV	C,A	0100 1111	4FH
Add the contents of register B to the contents of the accumulator.	ADD	B	1000 0000	80H
Invert (compliment) each bit in the accumulator.	CMA		0010 1111	2FH

Notes

These instructions are 1-byte instructions performing three different tasks. In the first instruction, both operand registers are specified. In the second instruction, the operand B is specified and the accumulator is assumed. Similarly, in the third instruction, the accumulator is assumed to be the implicit operand. These instructions are stored in 8- bit binary format in memory; each requires one memory location.

```
MOV rd, rs
```

rd ← rs copies contents of rs into rd.

Coded as 01 ddd sss where ddd is a code for one of the 7 general registers which is the destination of the data, sss is the code of the source register.



Example: MOV A,B

Coded as 01111000 = 78H = 170 octal (octal was used extensively in instruction design of such processors).

```
ADD r
```

```
A ← A + r
```

3.4.3 Two-Byte Instructions

In a two-byte instruction, the first byte specifies the operation code and the second byte specifies the operand. Source operand is a data byte immediately following the opcode. For example:

Task	Opcode	Operand	Binary Code	Hex Code	
Load an 8-bit data byte in the accumulator.	MVI	A, Data	0011 1110	3E	First Byte
			DATA	Data	Second Byte

Assume that the data byte is 32H. The assembly language instruction is written as

Mnemonics	Hex Code
MVI A, 32H	3E 32H

The instruction would require two memory locations to store in memory.

```
MVI r,data
```

```
r ← data
```



Example: MVI A, 30H coded as 3EH 30H as two contiguous bytes. This is an example of

immediate addressing.

```
ADI data
```

```
A ← A + data
```

```
OUT port
```

where port is an 8-bit device address. (Port) ← A. Since the byte is not the data but points directly to where it is located this is called direct addressing.

3.4.4 Three-Byte Instructions

In a three-byte instruction, the first byte specifies the opcode, and the following two bytes specify the 16-bit address. Note that the second byte is the low-order address and the third byte is the high-order address.

```
opcode + data byte + data byte
```

For example:

Notes

Task	Opcode	Operand	Binary Code	Hex Code	
Transfer the program sequence to the memory location 2085H.	JMP	2085H	1100 0011	C3	First byte
			100001010	85	Second byte
			010 0000	20	Third byte

This instruction would require three memory locations to store in memory.

Three-byte instructions — opcode + data byte + data byte

LXI rp, data16

rp is one of the pairs of registers BC, DE, HL used as 16-bit registers. The two data bytes are 16-bit data in L H order of significance.

rp <- data16



Example: LXI H,0520H coded as 21H 20H 50H in three bytes. This is also immediate

addressing.

LDA addr

A <- (addr) Addr is a 16-bit address in L H order. Example: LDA 2134H coded as 3AH 34H 21H. This is also an example of direct addressing.

3.4.5 Opcode Format

In the design of the 8085 microprocessor chip, all operations, registers and status flags are identified with a specific code. First understand how an instruction is designed into the microprocessor, which will be useful in reading a user’s manual. In user manual, the operation codes are specified in binary format and 8-bits are divided into various groups. All internal registers are identified as follows:

Table 3.7: Internal Registers and Their Codes

Code	Register pairs
00	BC
01	DE
10	HL
11	AF or SP

Code	Registers
000	B
001	C
010	D
011	E
100	H
101	L
111	A
110	Reserved for memory related operation

Notes

Some of the opcodes are given below:

- 1. Add the contents of a register to the Accumulator. The opcode is 10000 SSS (5 bit opcode – 3 bits are reserve for a register). If the register is B then the steps are as follows:

ADD	10000
to Register B	000
to Accumulator A	Implicit
Binary Instruction	<u>1000</u> <u>0000</u>
	8 0 H

Then the Assembly language presentation of this is as follows:

Opcode	Operand	Hexcode
ADD	B	80H

- 2. Move/copy the contents of the source register (R_s) to the destination register (R_d). If the R_s is Register A and R_d is Register D and 2-bit opcode or MOV is 01 then the binary value of this instruction is

MOVE the contents	01
To register D	010
From register A	111
Binary Instruction	<u>0101</u> <u>0111</u>
	5 7 H

Then the Assembly language presentation of this is as follows:

Opcode	Operand	Hexcode
MOV	D, A	57H

3.5 Sample Programs

An assembly program to add two numbers

Program

```
MVI D, 8BH
MVI C, 6FH
MOV A, C
1100 0011
1000 0101
0010 0000
ADD D
OUT PORT1
HLT
```

An assembly program to multiply a number by 8 Program

Notes

```

MVI A, 30H
RRC
RRC
RRC
OUT PORT1
HLT

```

An assembly program to find greatest between two numbers Program

```

MVI B, 30H
MVI C, 40H
MOV A, B
CMP C
JZ EQU
JC GRT
OUT PORT1
HLT
EQU: MVI A, 01H
OUT PORT1
HLT
GRT: MOV A, C
OUT PORT1
HLT

```



Case Study

Fetching an Instruction

Let us assume that we are trying the instruction at memory location 2005. That means that the program counter is now set to that value.

The following is the sequence of operations:

- The program counter places the address value on the address bus and the controller issues an RD signal.
- The memory's address decoder gets the value and determines which memory location is being accessed.
- The value in the memory location is placed on the data bus.
- The value on the data bus is read into the instruction decoder inside the microprocessor.
- After decoding the instruction, the control unit issues the proper control signals to perform the operation.

Questions:

1. Summarise the timing signals.
2. Explain Demultiplexing AD7-AD0.

Notes

3.6 Summary

- The microprocessor uses this register to sequence the execution of the instructions.
- An instruction is a binary pattern designed inside a microprocessor to perform a specific function. The entire group of instructions, called the instruction set
- The ADD, ADI instructions perform addition of register or 8-bit with the contents of the Accumulator. If the sum is larger than 8-bits, it sets the carry flag
- 8085 microprocessor is a programmable logic chip; it can also perform hardwired logic with its instruction set. The logical operations performed by includes AND, OR, XOR (Exclusive OR) and NOT (Complement).
- The jump instructions specify the memory location explicitly. This is a 3-byte instruction.

3.7 Keywords

- **Accumulator:** An accumulator is a register in which intermediate arithmetic and logic results are stored.
- **HLT:** HLT (*halt*) is an assembly language instruction which halts the CPU until the next external interrupt is fired.
- **Instruction:** An instruction is a command to the microprocessor to perform a given task on a specified data.
- **NOP or NOOP** (short for No Operation or No Operation Performed) is an assembly language instruction, sequence of programming language statements, or computer protocol command that effectively does nothing at all.
- **PC: Program counter** is a processor register that indicates where the computer is in its instruction sequence.



Lab Exercise

1. Give explanation of 8085 Microprocessor Kit.
2. Write a program to multiply 16-bit number with 8-bit number using 8085.

3.8 Self-Assessment Questions

1. The most commonly used flags are:

(a) bit	(b) carry
(c) register	(d) buffer
2. The accumulator is also identified as

(a) Flag	(b) buffer
(c) register	(d) none of these
3. What is one of the following instructions that is used to copy data from a source into a destination?

(a) MOV B, A	(b) COPY B, A
(c) Carry B, A	(d) all of these

4. is generally used when an error occurs in a program and an instruction needs to be eliminated.
- (a) HLT (b) OUT
(c) MVI (d) NOP
5. Which of the following instructions can only load data into Accumulator
- (a) IN (b) OUT
(c) LOAD (d) MVI
6. The instructions specify the memory location explicitly.
- (a) NOP (b) jump
(c) go to (d) continue

3.9 Review Questions

1. Discuss the 8085 Programming Model.
2. Explain in detail the branching operations.
3. With example explain different groups of instruction sets.
4. Write an 8085 assembly language program to perform 32-bit binary addition.
5. What will be the value in ACC, for the given 8085 program below?

```
MVI C, 7F
MVI B, 3E
MOV A, B
RLC
RLC
ANI 7F
HLT
```

6. Write an 8085 assembly language program to fill a block of 1000_H bytes starting at address 9000_H with characters 2B_H.
7. Indicate the logic levels on each control or status pin as logic '0' or logic '1' for 8085 microprocessor with the given data below:

Machine cycle	RD	WR	IO/M	S ₀	S ₁
Memory write					
Memory read					
I/O read					
I/O write					
O/P code fetch					

8. Explain the different data format.
9. Describe the instruction format in detail.
10. With suitable examples, explain the addressing modes of 8085.

Notes

Answers for Self-Assessment Questions

- | | | | |
|--------|--------|--------|--------|
| 1. (b) | 2. (c) | 3. (a) | 4. (d) |
| 5. (a) | 6. (b) | | |

3.10 Further Reading



Books

8080A/8085 Assembly Language Programming, Lance A. Leventhal.



Online link

<http://books.google.com/books?>

Unit 4: Microprocessor Architecture

Notes

CONTENTS

Objectives

Introduction

4.1 Microprocessor

- 4.1.1 The Address Bus
- 4.1.2 The Data Bus
- 4.1.3 The Control Bus
- 4.1.4 X64 vs X86

4.2 Architecture of Microprocessor

- 4.2.1 Arithmetic Logic Unit
- 4.2.2 Accumulator
- 4.2.3 Program Counter (PC)
- 4.2.4 Address, Data and Status Registers and Stack Pointer
- 4.2.5 Control Unit
- 4.2.6 CISC and RISC
- 4.2.7 Manufacturing
- 4.2.8 Multi-core Architecture
- 4.2.9 Von Neumann Architecture
- 4.2.10 Data Path
- 4.2.11 Harvard Architecture
- 4.2.12 Dual-Core vs Quad-Core Architecture

4.3 Microprocessor Operations

- 4.3.1 Input and Output
- 4.3.2 Arithmetic Logic Unit
- 4.3.3 Memory
- 4.3.4 Control Unit
- 4.3.5 Information Exchange

4.4 Microprocessor Memory

- 4.4.1 Actions of the Memory Unit
- 4.4.2 Memory and Addressing

4.5 Summary

4.6 Keywords

4.7 Self-Assessment Questions

4.8 Review Questions

4.9 Further Reading

Notes

Objectives

After studying this unit, you will be able to understand the following:

- Overview of Microprocessor
- Explain the Architecture of Microprocessor
- Define Microprocessor Operations
- Define Microprocessor Memory

Introduction

Research in microprocessor architecture investigates ways to increase the speed at which the microprocessor executes programs. All approaches have in common the goal of exposing and exploiting parallelism hidden within programs. A program consists of a long sequence of instructions. The microprocessor maintains the illusion of executing one instruction at a time, but under the covers it attempts to overlap the execution of hundreds of instructions at a time. Overlapping instructions is challenging due to interactions among them (data and control dependencies). A prevailing theme, *speculation*, encompasses a wide range of approaches for overcoming the performance-debilitating effects of instruction interactions. They include branch prediction and speculation for expanding the parallelism scope of the microprocessor to hundreds or thousands of instructions, dynamic scheduling for extracting instructions that may execute in parallel and overlapping their execution with long-latency memory accesses, caching and prefacing to collapse the latency of memory accesses, and value prediction and speculation for parallelizing the execution of data-dependent instructions, to mention a few.

Within this speculation framework, there is room for exposing and exploiting different styles of parallelism. *Instruction-level parallelism* (ILP) pertains to concurrency among individual instructions. Such fine-grained parallelism is the most flexible but not necessarily the most efficient. *Data-level parallelism* (DLP) pertains to performing the same operation on many data elements at once. This style of fine-grained parallelism is very efficient, but only applies when such regularity exists in the application. *Thread-level parallelism* (TLP) involves identifying large tasks within the program, each comprised of many instructions, that are conjectured to be independent or semi-independent and whose parallel execution may be attempted speculatively. Such coarse-grained parallelism is well-suited to emerging multi-core microprocessors (multiple processing cores on a single chip). With the advent of multi-core microprocessors, robust mixtures of ILP, DLP, and TLP are likely.

Microprocessor architecture research has always been shaped by underlying technology trends, making it a rapidly changing and vigorous field. As technology advances, previously discarded approaches are revisited with dramatic commercial success (e.g. superscalar processing became possible with ten-million transistor integration). By the same token, technology limitations cause a rethinking of the status quo (e.g. deeper pipelining seems unsustainable due to increasing power consumption).

4.1 Microprocessor

The microprocessor can be programmed to perform functions on given data by writing specific instructions into its memory. The microprocessor reads one instruction at a time, matches it with its instruction set, and performs the data manipulation specified. The result is either stored back into memory or displayed on an output device.

The 8085 uses three separate buses to perform its operations:

- The address bus
- The data bus
- The control bus

4.1.1 The Address Bus

16 bits wide (A0 A1...A15): Therefore, the 8085 can access locations with numbers from 0 to 65,536. Or, the 8085 can access a total of 64K addresses.

“Unidirectional”: Information flows out of the microprocessor and into the memory or peripherals.

When the 8085 wants to access a peripheral or a memory location, it places the 16-bit address on the address bus and then sends the appropriate control signals.

4.1.2 The Data Bus

8 bits wide (D0 D1...D7) & “Bi-directional”, Information flows both ways between the microprocessor and memory or I/O. The 8085 uses the data bus to transfer the binary information.

Since the data bus has 8-bits only, the 8085 can manipulate data 8 bits at a time only.

4.1.3 The Control Bus

There is no real control bus. Instead, the control bus is made up of a number of single bit control signals.



Did u know?

There are eight datelines', D0 through D7, in the 8085 microprocessor. They are shared, or multiplexed with the eight low order address lines, A0 through A7, and are called AD0 through AD7 on the...

Accessing Information in Memory:

For the microprocessor to access (Read or Write) information in memory (RAM or ROM), it needs to do the following:

- Select the right memory chip (using part of the address bus).
- Identify the memory location (using the rest of the address bus).
- Access the data (using the data bus).

Microprocessors function as the “brain” of a computer system. As technology has progressed, microprocessors have become faster, smaller and capable of doing more work per clock cycle.

However, when trying to choose a microprocessor, it is difficult to understand what the designations mean. Should you choose an x64 processor or an x86 processor? Is a dual-core fast enough, or do you need a quad-core processor? Understanding the differences in microprocessor architecture will aid in the decision-making process.



Did u know?

The latest technological changes in 32-and 64-bit microprocessors.

4.1.4 X64 vs X86

A microprocessor may be listed for sale as an “x86 processor” or an “x64 processor.” What is the difference, though?

An “x86” processor is a microprocessor that is capable of processing information in 32-bit pieces (called “instructions”). Each “bit” is a piece of information that the computer uses to transmit information, run computations and perform other such processes. A processor using x86 architecture is considered to be a successor technology to the original microprocessors used in

Notes

the IBM PC. Since the original processor used in an IBM PC was based upon the Intel 8086 microprocessor, successive microprocessors using the same set of instructions to run have been named similarly—the 80286, 80386 and 80486, for example.

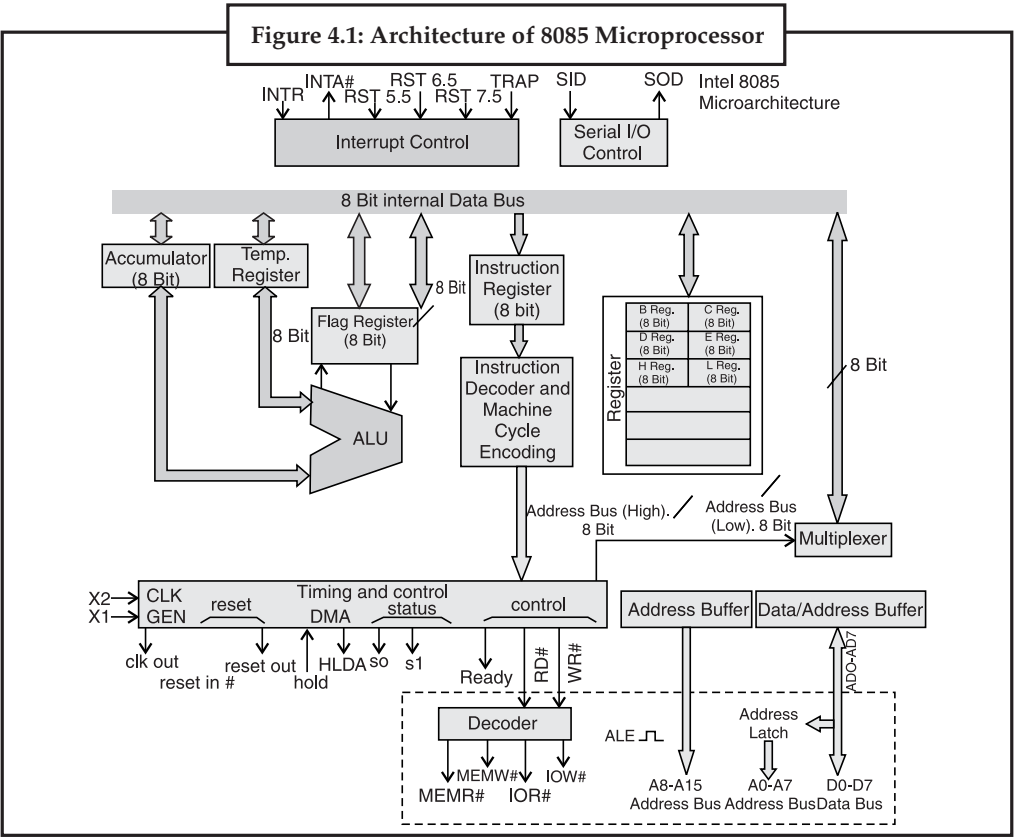
A microprocessor using x64 architecture is slightly different than the x86 processor. An x64 processor is capable of processing not only 32-bit instructions but also 64-bit instructions as well. Because of the increased capability of the x64 microprocessor, a computer that utilizes an x64 microprocessor is also capable of utilizing more memory (128 GB maximum vs. 4 GB maximum) than a computer with an x86 microprocessor.

An x64 microprocessor, therefore, would be the better choice if you plan to use the computer for memory-intensive applications, or if you need better overall performance out of your computer system.

4.2 Architecture of Microprocessor

A microprocessor is the central processing unit (CPU) of a computer. It is where processing of program instructions and data occurs. A basic computer consists of a microprocessor, external memory, and input and output devices.

The following sections describe the basic components of simple microprocessor architecture. It is closer to early microprocessors. However, it is still the foundation of today’s more complex microprocessors.



4.2.1 Arithmetic Logic Unit

Short for **Arithmetic Logic Unit**, ALU is one of the many components within a computer processor. The ALU performs mathematical, logical and decision operations in a computer and is the final

processing performed by the processor. After the information has been processed by the ALU, it is sent to the computer memory.

Notes

In some computer processors, the ALU is divided into two distinct parts, the AU and the LU. The AU performs the arithmetic operations and the LU performs the logical operations.

4.2.2 Accumulator

The Accumulator holds one of the operands as well as the result in operations performed by the ALU.

Accumulator may refer to:

- **Accumulator (computing)**, in a CPU, a processor register for storing intermediate results
- **Accumulator (energy)**, an apparatus for storing energy or power
 - **Capacitor**, in electrical engineering, also known by the obsolete term accumulator
 - **Electrochemical cell**, a cell that stores electrical energy, typically used in rechargeable batteries
 - **Hydraulic accumulator**, an energy storage device using hydraulic fluid under pressure
- **Accumulator (bet)**, a parlay bet
- **Accumulator (structured product)**, a financial contract used by clients (usually individuals) to accumulate stock positions over time
- **Accumulator 1**, a Czech film
- **Dynamic accumulator**, a plant that mines nutrients from the soil through its roots

4.2.3 Program Counter (PC)

The program counter contains the memory address of the next program instruction to be executed.

The program counter or PC (also called the instruction pointer to a seminal Intel instruction set, such as the 8080 or 4004, or instruction address register, or just part of the instruction sequencer in some computers) is a processor register that indicates where the computer is in its instruction sequence. Depending on the details of the particular computer, the PC holds either the address of the instruction being executed or the address of the next instruction to be executed.

In most processors, the program counter is incremented automatically after fetching a program instruction so that instructions are normally retrieved sequentially from memory, with certain instructions, such as branches, jumps and subroutine calls and returns, interrupting the sequence by placing a new value in the program counter.

Such jump instructions allow a new address to be chosen as the start of the next part of the flow of instructions from the memory. They allow new values to be loaded (written) into the program counter register. A subroutine call is achieved simply by reading the old contents of the program counter, before they are overwritten by a new value, and saving them somewhere in memory or in another register. A subroutine return is then achieved by writing the saved value back into the program counter again.



Task

Give the basic concept of accumulator. Explain with example.

Notes

4.2.4 Address, Data and Status Registers and Stack Pointer

The Address Register contains address of a memory location to be accessed.

The Data Register contains the data coming from or going to memory or an I/O port.

The Status Register contains information about the result of the previous ALU operation.

The Stack Pointer Register contains the address of the block of memory (the stack) where subroutine return addresses are stored.

4.2.5 Control Unit

The Control Unit contains the circuitry that controls the process of fetching, decoding and executing program instructions.

The control unit is the circuitry that controls the flow of information through the processor, and coordinates the activities of the other units within it. In a way, it is the “brain within the brain”, as it controls what happens inside the processor, which in turn controls the rest of the PC.

The functions performed by the control unit vary greatly by the internal architecture of the CPU, since the control unit really implements this architecture. On a regular processor that executes x86 instructions natively, the control unit performs the tasks of fetching, decoding, managing execution and then storing results. On a processor with a RISC core the control unit has significantly more work to do. It manages the translation of x86 instructions to RISC micro-instructions, manages scheduling the micro-instructions between the various execution units, and juggles the output from these units to make sure they end up where they are supposed to go. On one of these processors the control unit may be broken into other units (such as a scheduling unit to handle scheduling and a retirement unit to deal with results coming from the pipeline) due to the complexity of the job it must perform.

4.2.6 CISC and RISC

Historically, the first type of ISA was the complex instruction set computers (CISC), and the second type was the reduced instruction set computers (RISC). It is a common misunderstanding that RISC systems typically have a small ISA (fewer instructions) but make up for it with faster hardware. RISC system actually have “reduced instructions”, in the sense that each instruction does so little that it takes very little time to execute it. It is a common misunderstanding that CISC systems have more instructions, but typically pay a steep performance penalty for the added versatility. CISC systems actually have “complex instructions”, in the sense that at least one instruction takes a long time to execute for example, the “double indirect” addressing mode inherently requires two memory cycles to execute, and a few CPUs have a “string copy” instruction that may require hundreds of memory cycles to execute. MIPS and SPARC are examples of RISC computers. Intel x86 is an example of a CISC computer.

The two main types of microprocessor architecture are complex instruction set computer (CISC) and reduced instruction set computer (RISC). CISC architecture is much more complex and thus can handle more complex commands. RISC uses a simpler architecture, so RISC microprocessors are smaller and faster.



RISC system actually have “reduced instructions”, in the sense that each instruction does so little that it takes very little time to execute it.

4.2.7 Manufacturing

Microprocessors are manufactured by etching different features (such as transistors and resistors) into a tiny wafer of pure silicon and then adding various layers of conductors and insulators. The

features of a microprocessor are so small that they are dwarfed by the smallest speck of dust; thus, the chips must be manufactured in a highly controlled, dust-free environment.

4.2.8 Multi-core Architecture

Microprocessor architecture has traditionally consisted of a single “core”; that is, the chips could only process one piece of information at a time. However, many processors are now being built with two, four, or even more cores, allowing one microprocessor to process multiple pieces of information simultaneously.

4.2.9 Von Neumann Architecture

Early computer programs were hard-wired. To reprogram a computer meant changing the hardware switches manually, that took a long time with potential errors. Computer memory was only used for storing data.

A Von Neumann microprocessor is a processor that follows this pattern:

Fetch: An instruction and the necessary data are obtained from memory.

Decode: The instruction and data are separated, and the components and pathways required to execute the instruction are activated.

Execute: The instruction is performed, the data is manipulated, and the results are stored.

This pattern is typically implemented by separating the task into two components, the control, and the data path.



Task

Draw the block diagram of Von Neumann Architecture.

Control: The control unit reads the instruction, and activates the appropriate parts of the data path.

4.2.10 Data Path

The data path is the pathway that the data takes through the microprocessor. As the data travels to different parts of the data path, the command signals from the control unit cause the data to be manipulated in specific ways, according to the instruction. The data path consists of the circuitry for transforming data and for storing temporary data. It contains ALUs capable of transforming data through operations such as addition, subtraction, logical AND, OR, inverting, and shifting.

4.2.11 Harvard Architecture

In a Harvard Architecture machine, the computer system’s memory is separated into two discrete parts: data and instructions. In a pure Harvard system, the two different memories occupy separate memory modules, and instructions can only be executed from the instruction memory.

In a “Princeton Architecture” machine, the computer system’s memory comprises one uniform address space: data and instructions may be placed anywhere in this single memory.

Many DSPs are modified Harvard architectures, designed to simultaneously access three distinct memory areas: the program instructions, the signal data samples, and the filter coefficients (often called the P, X, and Y memories).

In theory, such three-way Harvard architectures can be three times as fast as a Princeton architecture that is forced to read the instruction, the data sample, and the filter coefficient, one at a time.

In Princeton architecture systems, there is only one memory area. Any particular memory location that can be written to and read as data at any one time can also be executed as an instruction at

Notes

other times. Several common computer security problems arise because modern processors are not pure Harvard systems and manipulate instructions as if they were data, and vice versa.

4.2.12 Dual-Core vs Quad-Core Architecture

Many microprocessors made by Intel and AMD are multi-core processors. What this means is that within one microprocessor, there are two or more central processing units (cores) within one integrated circuit package.

As their names imply, “dual-core” multiprocessors have two CPU cores in the multiprocessor package, and “quad-core” multiprocessors have four CPU cores. Depending upon the software being used, the operating system the computer system has installed and the amount of memory available to the computer system, a quad-core system is capable of running more simultaneous processes than a dual-core system.

However, some older software or operating systems (such as Windows 95, Windows 98 or Windows Me) are not capable of utilizing multi-core microprocessors. If the software or operating system is incapable of utilizing the resources available, using a multi-core microprocessor is no different than using a single-core microprocessor.

4.3 Microprocessor Operations

A microprocessor manipulates data in a computer system. The central processing unit acts as the brain of a computer and consists of one or more microprocessors made up of several thousand transistors on a single integrated circuit. The microprocessor works in conjunction with other parts of the computer to compute arithmetic and logic functions to handle tasks using an instruction set to perform all tasks within a computer.

4.3.1 Input and Output

The microprocessor accepts input from devices, such as a mouse, keyboard or scanner, and performs a function on that data. It makes a decision based on the data, the microprocessor computes the information and then it sends the results to the output devices, such as a monitor or printer, as readable information for the user. For example, if a user using a word processor presses “m” on the keyboard, the microprocessor will accept that and send the letter “m” to the monitor.

Input/Output Devices

Keyboards, Floppy disk are the examples of input devices. Printer, LED / LCD display, CRT Monitor are the examples of output devices.

4.3.2 Arithmetic Logic Unit

The arithmetic logic unit gathers information as input from the CPU registers and operands and then does the arithmetic operations (addition, subtraction, multiplication and division) and logic operations (AND, OR and XOR). During data processing, the ALU tests conditions and prepares to take different actions based on results. The ALU also gathers data from additional sources, including number systems, instructions, timing and data routing circuits, such as adders and subtracters.

4.3.3 Memory

The microprocessor accesses and stores binary instructions into memory, or circuits that store bits. Random access memory is a control memory that uses registers to temporarily store data. The microprocessor stores volatile data used by programs in RAM. Read-only memory stores data permanently on chips with instructions built in. It takes longer to access the information in ROM, but it does not lose information when a computer shuts down as does RAM.

4.3.4 Control Unit

The control unit directs the flow of operations and data by selecting one program statement at a time, interpreting it and sending messages to the ALU or registers to carry out the instruction. It

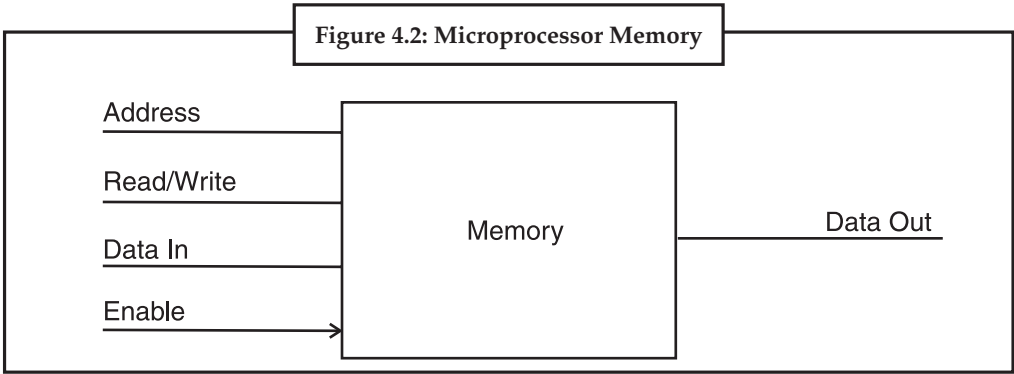
also decides where to keep information in memory and which devices to communicate with by interfacing with the ALU, memory and input/output devices. The control unit can also shut down a computer if it or another device, such as the power source, detects abnormal conditions.

4.3.5 Information Exchange

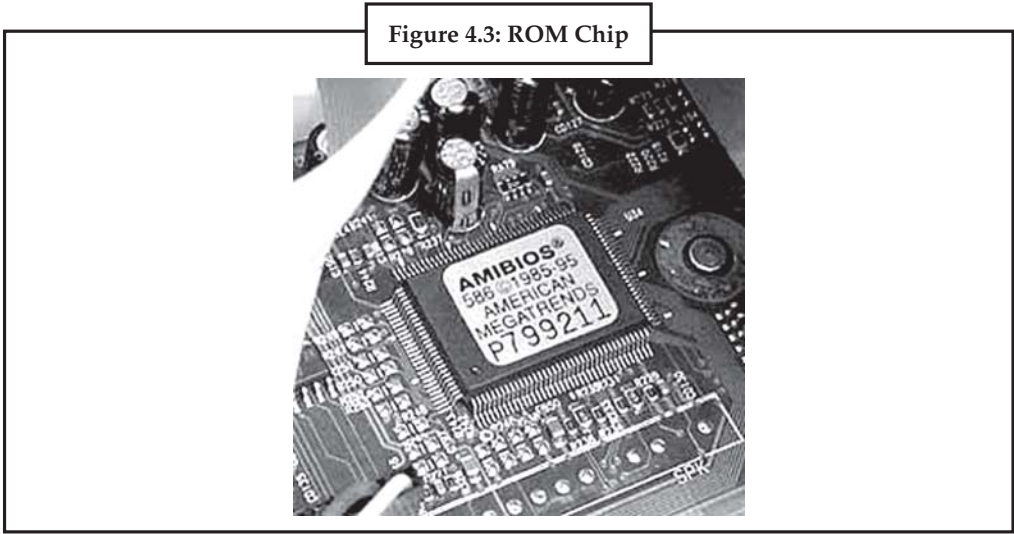
The system bus connects the microprocessor to the peripherals, such as a keyboard, mouse, printer, scanner, speaker or digital camera. The microprocessor sends and receives data through the system bus to communicate with the peripherals. It only communicates with one peripheral at a time so as to not mix up any information and send it to the wrong place. The control unit controls the timing of the information exchange.

4.4 Microprocessor Memory

The previous section talked about the address and data buses, as well as the RD and WR lines. These buses and lines connect either to RAM or ROM — generally both. In our sample microprocessor, we have an address bus 8 bits wide and a data bus 8 bits wide. That means that the microprocessor can address (2⁸) 256 bytes of memory, and it can read or write 8 bits of the memory at a time. Let’s assume that this simple microprocessor has 128 bytes of ROM starting at address 0 and 128 bytes of RAM starting at address 128.



ROM stands for read-only memory. A ROM chip is programmed with a permanent collection of pre-set bytes. The address bus tells the ROM chip which byte to get and place on the data bus. When the RD line changes state, the ROM chip presents the selected byte onto the data bus.



RAM stands for random-access memory. RAM contains bytes of information, and the microprocessor can read or write to those bytes depending on whether the RD or WR line is signalled. One problem with today’s RAM chips is that they forget everything once the power goes off. That is why the computer needs ROM.

Notes



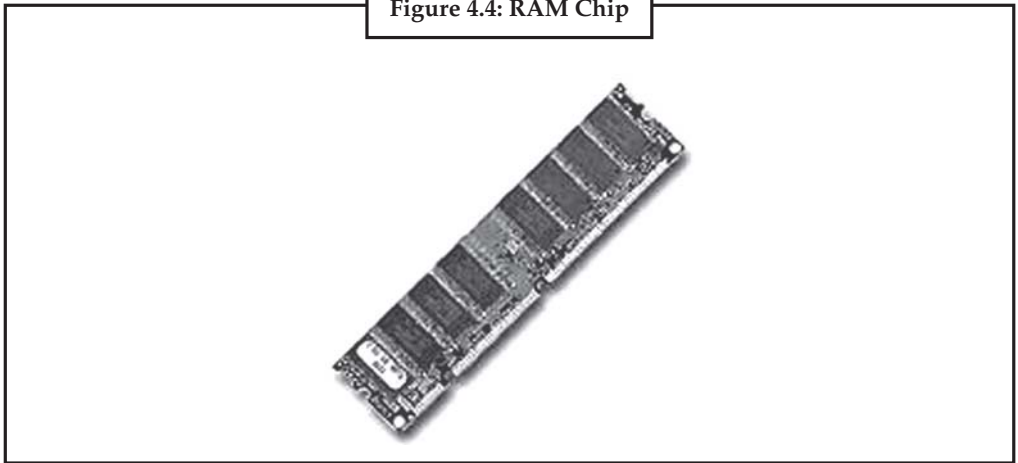
ROM cannot be used as stack because it is not possible to write to ROM.

By the way, nearly all computers contain some amount of ROM (it is possible to create a simple computer that contains no RAM; many microcontrollers do this by placing a handful of RAM bytes on the processor chip itself but generally impossible to create one that contains no ROM). On a PC, the ROM is called the BIOS (Basic Input/output System). When the microprocessor starts, it begins executing instructions it finds in the BIOS. The BIOS instructions do things like test the hardware in the machine, and then it goes to the hard disk to fetch the boot sector (How Hard Disks Work—for details). This boot sector is another small program, and the BIOS store it in RAM after reading it off the disk. The microprocessor then begins executing the boot sector’s instructions from RAM. The boot sector program will tell the microprocessor to fetch something else from the hard disk into RAM, which the microprocessor then executes, and so on. This is how the microprocessor loads and executes the entire operating system.



RAM chip addresses size is 7 bit and ROM chip addresses size is 8 bit.

Figure 4.4: RAM Chip



4.4.1 Actions of the Memory Unit

- **Program Memory:** A memory that contains the program (which we had written), after we’ve burned it. As a reminder, Program Counter executes commands stored in the program memory, one after the other.
- **Data Memory:** This is RAM memory type, which contains a special registers like SFR (Special Faction Register) and GPR (General Purpose Register). The variables that we store in the Data Memory during the program are deleted after we turn off the micro.

These two memories have separated data buses, which makes the access to each one of them very easy.

- **Data EEPROM (Electrically Erasable Programmable Read-Only Memory):** A memory that allows storing the variables as a result of burning the written program.



Cache memory is a small high-speed memory. It is used for temporary storage of data and information between the main memory and the CPU. The cache memory is only in RAM.

Each one of them has a different role. Program Memory and Data Memory are two memories that are needed to build a program, and Data EEPROM is used to save data after the microcontroller is turned off.

Program Memory and Data EEPROM are non-volatile memories, which store the information even after the power is turned off. These memories called Flash or EEPROM. In contrast, Data Memory does not save the information because it needs power in order to maintain the information stored in the chip.

4.4.2 Memory and Addressing

There are several different types of memory in a micro. One is Program memory. This is where the program is located. Another is Data memory. This is where data that might be used by the program is located. The neat (or strange) thing is that they both reside in the same memory space and can be altered by the program. That is right; a program can actually alter itself if that was necessary. Two terms are used when talking about memory: Reading (load) is getting a value from memory and writing (store) is putting a value into memory.

There are three buses associated with the memory subsystem. One is the address bus, the second is the data bus, and the third is the control bus. It's important for you to know exactly how all this works, because these buses transport data and addresses everywhere. All three are connected to the memory subsystem. It's also good to know the function of each to better understand what's happening. In the 8085 CPU, the address bus is 16 bits wide. It acts to select one of the unique 216 (64K) memory locations. The control bus determines whether this will be a read or a write. In the case of an instruction fetch, the control bus is set up for a read operation. Data is read or written through the data bus, which is 8 bits wide. This is why all registers and memory are 8 bits wide, it's the width of the data bus on the 8085 CPU. A bus is just a group of connections that all share a common function. Instead of speaking of each bit or connection in the address separately, for example, all 16 are taken together and referred to simply as the address bus. The same is true for the control and data buses.

A byte is the most used number in a micro because each memory location or register is one byte wide. Memory has to be thought of as a sort of file cabinet with each location in it being a folder in the cabinet. In a file cabinet, you go through the tabs on the folders until you find the right one. To get to each memory location, a different method is used. Instead, a unique address is assigned to each location. In most micros this address is a word or 16 bits, or 4-digit hex. This allows for a maximum of 65536 (216 or 64K) unique addresses or memory locations that can be accessed. These addresses are usually referred to by a 4-digit hex number. Memory usually starts at address 0000h and could go up to FFFFh (216 or 64K or 65536 in total). To access these locations, a 16-bit address is presented to memory and the byte at that location is either read or written.



Case Study

Small Cabbage Processor

This small business case study prepares, ferments, and packages sauerkraut. Their busiest time is August-November, when they employ seasonal workers (many who do not speak English). The process starts with the cleaning, coring, and shredding of raw cabbage. The outer leaves are peeled off by hand as the cabbage passes on a conveyor. The equipment in this room is noisy when running; employees wear earplugs and gloves. The fermentation processes occurs in large, two-storey tanks. Workers enter the tanks to load the sauerkraut into a vacuum hose, which carries it to the packaging lines. They wear harnesses hooked to overhead safety lines while performing this work. A confined-space entry permit program is followed. All employees are required to wear safety glasses, earplugs, bump caps, and hairnets. On the packaging floor, there are lines that run metal and glass containers.

Contd...

Notes	<p>They are rinsed, filled, pasteurized, packed into boxes, palleted, and shrink-wrapped for shipment. In another room, employees pack bagged sauerkraut into boxes and lift them onto a conveyor line.</p> <p>Grant Work: After an initial gap analysis and interview several issues arose, the business had recently broke away from being part of a major corporation, leaving the business with limited staffing resources, causing some problems with keeping paperwork and procedures up to date. Aside from that problem the business had a decent health and safety program left over from the parent company, the person left in charge of the safety and health program was also the director of human resources, they were beginning to bring in other people to help with running the safety and health programs. The business conducts scheduled safety meetings and has a good system for addressing and investigating any accidents. There appears to be good communication of safety issues to employees.</p> <p>The business had recently received an OSHA inspection, which cited several guarding issues around the facility, by the time the RIT assistants scheduled a visit majority of the cited issues were corrected. This showed that the business was interested in improving upon its safety and health. OSHA did not cite several other issues that the RIT assistant felt needed addressing.</p> <p>The first of the issues involved injuries, the most common injuries are lifting and ergonomic related. The worst of the ergonomic issues was in the palleting area, the employees had to lift cases of jars from the conveyor line and in most cases had to spin 180 degrees and lower the cases to a pallet that is located on the floor. The cases weighed between 15 and 50 lbs. The RIT assistants ran the NIOSH lifting equation on a worst and best case scenario to help determine and design the best lift system possible for their operation. For the worst case scenario, which was present in the facility, the recommended weight limit (RWL) was only 2.68 pounds, for the best case which was not found the RWL to be 113.06 pounds. From an attempt to design for best case was begun, the RIT assistant with the HR director and Operations supervisor, the owners of the facility would only accept a very low cost solution which prevented the purchases of lift tables, this case was later referred to the RIT Centre for Integrated Manufacturing studies which provides ergonomic services under a NYS Hazard Abatement grant for more expertise. The business has realized this to be a problem before participating in the Harwood grant and has sent employees to undergo ergonomic training by an outside consultant.</p> <p>Much of the facility had a noise level that was over 85dBs. The cutting and coring room which was not operating at the times of visits was described to be of deafening sound levels, in this room there is much mechanical motor noise and noise from metal cutting blades hitting metal tables during the coring process. Discussion about sound dampening and guarding techniques was held. Hearing protection is required around much of the facility but there was no formal audiometric testing at the time. The facility has contracted with an outside consultant to conduct baseline hearing assessment and annual test. Since half of their staff is seasonal that will provide some problems.</p> <p>Questions:</p> <ol style="list-style-type: none">1. Explain the basic concept of Small Cabbage Processor.2. What are the basic uses of Small Cabbage Processor in business?
-------	---

4.5 Summary

- Microprocessor architecture research has always been shaped by underlying technology trends, making it a rapidly changing and vigorous field.
- Microprocessors function as the “brain” of a computer system.

- The microprocessor can be programmed to perform functions on given data by writing specific instructions into its memory.
- Microprocessor architecture investigates ways to increase the speed at which the microprocessor executes programs.

4.6 Keywords

- **Arithmetical logical circuit (ALU):** ALU is the part of the microprocessor that performs arithmetic operations. ALUs can typically add, subtract, divide, multiply, and perform logical operations of two numbers (and, or, nor, not, etc).
- **Complex instruction set computer (CISC):** A complex instruction set computer (CISC) is a computer where single instructions can execute several low-level operations (such as a load from memory, an arithmetic operation, and a memory store) and/or are capable of multi-step operations or addressing modes within single instructions.
- **Data-level parallelism (DLP):** *Data-level parallelism* pertains to performing the same operation on many data elements at once.
- **Instruction-level parallelism (ILP):** *Instruction-level parallelism* pertains to concurrency among individual instructions.
- **Reduced instruction set computing (RISC):** Reduced instruction set computing or RISC is a CPU design strategy based on the insight that simplified (as opposed to complex) instructions can provide higher performance if this simplicity enables much faster execution of each instruction.
- **Thread-level parallelism (TLP):** *Thread-level parallelism* involves identifying large tasks within the program, each comprised of many instructions that are conjectured to be thread.



1. Draw the block diagram of the Microprocessor Memory and all the parts of units.

Lab Exercise 2. Give the basic difference between RISC and CISC.

4.7 Self-Assessment Questions

1. Stack pointer and Program counter all have 16 bits.
(a) True (b) False
2. 8085 is a address microprocessor.
(a) One (b) Two
(c) Three (d) Four
3. Intel x86 is an example of a
(a) RISC (b) DSP
(c) None (d) CISC
4. MIPS and SPARC are examples of CISC.
(a) True (b) False
5. The length of a register is known as the word length of the computer.
(a) True (b) False

Notes

6. The total addressable memory size
 - (a) 16 bits
 - (b) 32 kb
 - (c) 64 kb
 - (d) 128 bits
7. Program counters a bit register.
 - (a) 16 bit
 - (b) 8 bit
 - (c) 64 bit
 - (d) 32 bit
8. In a pure Harvard system, there are different memories.
 - (a) Two
 - (b) Three
 - (c) Four
 - (d) Five
9. RISC systems actually have “complex instructions”.
 - (a) True
 - (b) False

4.8 Review Questions

1. What is a Microprocessor?
2. What do you mean by address bus?
3. Why is the data bus bi-directional?
4. Explain the architecture of the 8085 microprocessor.
5. Differentiate between X64 and X86.
6. Explain the Von Neumann Architecture microprocessor.
7. What is Harvard Architecture?
8. What are the differences between Dual-Core and Quad-Core Architecture?
9. What are input and output devices?
10. Why does microprocessor contain ROM chips?

Answers for Self-Assessment Questions

- | | | | | |
|--------|--------|--------|--------|--------|
| 1. (a) | 2. (a) | 3. (d) | 4. (b) | 5. (a) |
| 6. (c) | 7. (a) | 8. (a) | 9. (b) | |

4.9 Further Reading



Books

Microprocessor Architecture, Jean-Loup Baer



Online link

<http://books.google.com/books?>

Unit 5: Microcomputer System

Notes

CONTENTS

Objectives

Introduction

5.1 Microcontroller Systems (Microcomputer System)

5.2 Microcontroller Architectures

5.2.1 RISC and CISC

5.3 Microcontroller Features

5.3.1 Supply Voltage

5.3.2 The Clock

5.3.3 Timers

5.3.4 Watchdog

5.3.5 Reset Input

5.3.6 Interrupts

5.3.7 Brown-out Detector

5.3.8 Analogue-to-Digital Converter

5.3.9 Serial Input/Output

5.3.10 EEPROM Data Memory

5.3.11 LCD Drivers

5.3.12 Analogue Comparator

5.3.13 Real-time Clock

5.3.14 Sleep Mode

5.3.15 Power on Reset

5.3.16 Low Power Operation

5.3.17 Current Sink/Source Capability

5.4 Example of Microcomputer

5.5 I/O Devices of Microcomputer System

5.6 Summary

5.7 Keywords

5.8 Self-Assessment Questions

5.9 Review Questions

5.10 Further Reading

Notes

Objectives

After studying this unit, you will be able to understand the following:

- Describe Microcontroller Systems (Microcomputer System)
- Understand Microcontroller Architectures
- Explain Microcontroller Features
- Example of Microcomputer
- Define I/O Devices of Microcomputer System

Introduction

The term microcomputer is used to describe a system that includes a minimum of a microprocessor, program memory, data memory, and input/output (I/O). Some microcomputer systems include additional components such as timers, counters, analogue-to-digital converters and so on. Thus, a microcomputer system can be anything from a large computer having hard disks, floppy disks and printers to a single chip computer system.

5.1 Microcontroller Systems (Microcomputer System)

The microcomputers consist of single silicon chip. Such microcomputer systems are also called microcontrollers.

Microcontrollers are general purpose microprocessors which have additional parts that allow them to control external devices. Basically, a microcontroller executes a user program which is loaded in its program memory. Under the control of this program data is received from external devices (inputs), manipulated and then sent to external output devices. A microcontroller is a very powerful tool that allows a designer to create sophisticated input/output data manipulation. Microcontrollers are classified by the number of bits in a data word. 8-bit microcontrollers are the most popular ones and are used in many applications. 16- and 32-bit microcontrollers are much more powerful, but usually more expensive and not required in many small to medium general purpose applications where microcontrollers are used.

The simplest microcontroller architecture consists of a microprocessor, memory, and input/output. The microprocessor consists of a central processing unit (CPU) and the control unit (CU).

The CPU is the brain of a microprocessor and is where all of the arithmetic and logical operations are performed. The control unit controls the internal operations of the microprocessor and sends out control signals to other parts of the micro-processor to carry out the required instructions.



Caution

How to differentiate between ROM and RAM?

5.2 Microcontroller Architectures

Basically, two types of architectures are used in microcontrollers: Von Neumann architecture and Harvard architecture. Von Neumann architecture is used by a very large percentage of microcontrollers and here all memory space is on the same bus, and instruction and data are treated identically. In the Harvard architecture (used by the PIC microcontrollers), code and data storage are on separate buses and this allows code and data to be fetched simultaneously, resulting in a more efficient implementation.

5.2.1 RISC and CISC

RISC (Reduced Instruction Set Computer) and CISC (Complex Instruction Set Computer) refer to the instruction set of a microcontroller. In a RISC microcontroller, instruction words are more

than 8 bits wide (usually 12, 14, or 16 bits) and the instructions occupy one word in the program memory. RISC processors (e.g. PIC) have no more than about 35 instructions, and offer higher speeds. CISC microcontrollers have 8-bit wide instructions and they usually have over 200 instructions. Some instructions (e.g. branch) occupy more than one program memory location.

5.3 Microcontroller Features

Microcontrollers from different manufacturers have different architectures and different capabilities. Some may suit a particular application while others may be totally unsuitable.

5.3.1 Supply Voltage

Most microcontrollers operate with the standard +5 V supply. Some microcontrollers can operate at as low as 2.7 V and some will tolerate 6 V without any problems. You should check the manufacturers' data sheets about the allowed limits of the supply voltage.

5.3.2 The Clock

All microcontrollers require an oscillator (known as a clock) to operate. Most microcomputers will operate with a crystal and two capacitors. Some will operate with resonators or with external resistor capacitor pair. Some microcontrollers have built-in resistor–capacitor type oscillators and they do not require any external timing components (e.g. PIC12C672). If your application is not time-sensitive you should use external or internal (if available) resistor–capacitor timing components for simplicity and low cost.

5.3.3 Timers

Timers are an important part of any microcontroller. A timer is basically a counter which is driven from an accurate clock (or a division of this clock). Timers can be 8-bits or 16-bits long. Data can be loaded into the timers and they can be started and stopped under software control. Most timers can be configured to generate an interrupt when they reach a certain count (usually when they overflow). Some microcontrollers offer capture and compare facilities where a timer value can be read when an external event occurs, or the timer value can be compared to a preset value and interrupts can be generated when this value is reached. It is typical to have at least one timer on every microcontroller. Some microcontrollers may have three or more while others may have two timers.

5.3.4 Watchdog

Many microcontrollers have at least one watchdog facility. The watchdog is usually refreshed by the user program and a reset occurs if the program fails to refresh the watchdog. Watchdog facilities are commonly used in real-time systems where it is required to check the proper termination of one or more activities.

5.3.5 Reset Input

This input resets the microcomputer. Most microcontrollers have a resistor connected to the supply voltage and this ensures that the microcontroller starts properly after the application of power. Some microcontrollers have internal reset circuitry which does not require any external components.

5.3.6 Interrupts

Interrupts are a very important concept in microcontrollers. An interrupt causes a microcontroller to respond to external and internal (e.g. timer) events very quickly. When an interrupt occurs the

Notes

microcontroller leaves its normal flow of execution and jumps directly to the interrupt service routine. Interrupts can in general be nested such that a new interrupt can suspend the execution of another interrupt. Most microcontrollers have at least one, though some have several interrupt sources.

5.3.7 Brown-out Detector

Brown-out detectors are also common in many microcontrollers and they reset a microcontroller if the supply voltage falls below a nominal value. Brown-out detectors are usually employed to prevent unpredictable operation at low voltages, especially to protect the contents of EEPROM type memories.

5.3.8 Analogue-to-Digital Converter

Some microcontrollers are equipped with analogue-to-digital converter circuits. Usually these converters are 8 bits, but some microcontrollers have 10- or even 12-bit converters. A/D converters usually generate interrupts when a conversion is complete so that the user program can read the converted data very quickly. A/D converters are very useful in control and monitoring applications since most sensors produce analogue output voltages.

5.3.9 Serial Input/Output

Some microcontrollers contain hardware to implement a serial asynchronous communications interface. The baud rate and the data format can usually be selected in software. If serial input/output hardware is not provided, it is easy to develop software to implement serial data transfer using any I/O pin of a microcontroller. Some microcontrollers incorporate SPI (Serial Peripheral Interface) or IC (Integrated InterConnect) bus interfaces. These enable a microcontroller to interface to other compatible devices easily.

5.3.10 EEPROM Data Memory

EEPROM type memory is also very common in many microcontrollers. The programmer can store non-volatile data in such memory and can also change this data whenever required. Some microcontroller types provide between 64 and 256 bytes of EEPROM data memories, while some others do not have any such memories.

5.3.11 LCD Drivers

LCD drivers enable a microcontroller to be connected to an external LCD display directly. These drivers are not very common since most of the functions provided by them can be implemented by software.

5.3.12 Analogue Comparator

Analogue comparators enable analogue signals to be compared easily. These circuits are not very common and are only implemented in some microcontrollers.

5.3.13 Real-time Clock

The real-time clock is another feature which is implemented in some microcontrollers. These microcontrollers usually keep the date and time of day and they are intended for the consumer market

5.3.14 Sleep Mode

Some microcontrollers (e.g. PIC) offer sleep modes where executing this instruction puts the microcontroller into a mode where the internal oscillator is stopped and the power consumption

is extremely low. The devices usually wake up from the sleep mode by external reset or by a watchdog time-out.

5.3.15 Power on Reset

Some microcontrollers (e.g. PIC) provide an on-chip power-on reset circuitry which keeps the microcontroller in reset state until all the internal clock and the circuitry are initialized properly.

5.3.16 Low Power Operation

Low power operation is important in portable applications. Some microcontrollers (e.g. PIC) can operate with less than 2mA with 5 V supply, and around 15 A at 3 V supply. Some other microcontrollers may consume as much as 80 mA or more at 5 V supply.

5.3.17 Current Sink/Source Capability

This is important if the microcontroller is to be connected to an external device which draws large current for its operation. Some microcontrollers can sink and source only a few mA of current and driver circuits are required if they have to be connected to devices with large current requirements. PIC microcontrollers can sink and source up to 25 mA of current from each I/O pin which is suitable for most small applications, e.g. they can be connected to LEDs without any driver circuits.

5.4 Example of Microcomputer

Microcomputers, personal computers (PCs), are small, lightweight, and portable. Some of them are more powerful than some of the older, larger mainframes and minicomputers. Microcomputers are unique in that the heart of the computer (the CPU) is contained on a single integrated chip (IC) and the entire computer system is contained on a handful of printed circuit boards located inside a small compact frame or cabinet. In some cases a complete microcomputer is located on a single chip; the CPU, co-processor, and memory. Some micros/PCs are high-speed, multi-user, multi-tasking units. Traditionally micros are used for word processing, database management, spreadsheets, graphics, desktop publishing, and other general office applications. Currently, micros and PCs are being used for tactical support systems, such as Naval Intelligence Processing Systems (NIPS) and Joint Operational Tactical System (JOTS). Micros and PCs can also be used as a SNAP system for shore-based operational commands, such as ASWOC. The operational programs for PCs used for a tactical support system are supported externally by technical teams. These operational programs are also updated as systems are added or replaced. Programs that are used for word processing, graphics, and so on are abundant and can be obtained through civilian vendors and software support teams such as Commander Naval Computer and Telecommunications Command (COMNAVCOMTELCOM). Training for microcomputers is obtained through formal A schools, civilian contractor schools, and OJT. Training for micros is not NEC producing.

The following is a brief description of a typical PC/desktop system:

Small compact frame or cabinet: PCs are unique in that the frame or cabinet contains the majority of the components for a complete system. A typical PC frame or cabinet contains the following components:

- Backplane or motherboard for printed circuit boards
- A central processor unit (CPU) and memory printed circuit board(s) (pcb)
- Input/output pcb
- Disk controller pcb

Notes

- Video controller pcb
- Data storage devices: Hard disk drive units, floppy disk drive units, and/or tape cassette units
- I/O connector: Parallel or serial communications—A small fan: No special cooling requirements; the unit relies on ambient temperature of the room or space
- Power supply
- No special requirements—Display monitors are output



In some cases the CPU and memory are located on the same pcb.

5.5 I/O Devices of Microcomputer System

The computer will be of no use if it is not communicating with the external world.

Thus, a computer must have a system to receive information from the outside world and must be able to communicate results to the external world. For this purpose computers use input/output devices. Input and output devices can also be written as I/O devices.

Input and output devices of a computer system are the devices that connect you to computer. Input devices let you to transfer data and user command into the computer. Input devices technologies are rapidly developing and are used to interact with the computer system. For example, you can type in data by using a keyboard, or you can input data in picture form by using a scanner in computer system. Inputs are data or signals received by the computer system and outputs are the data and signals which are sent by it as result after processing the input.

The output devices display the result of input data or signals after processing it. Examples of these could be your computer's monitor, which displays all the programs which are running into the computer, as well as the printer, which will print out a hard copy of the information which is saved in your computer.

Input and output devices allow the computer system to interact with the outside world by moving data into and out of the system. An input device is used to bring data into the system. Examples of some input devices are:

- Keyboard
- Mouse
- Joystick
- Microphone
- Barcode reader
- Graphics tablet

An output device is used to send data out of the system. Examples of some output devices are:

- Monitor
- Printer
- Plotter
- Speaker

Input and output devices in short are also called I/O devices. They are directly connected to an electronic module called I/O module or device controller. For example, the speakers of a

multimedia computer system are directly connected to a device controller called an audio card which in turn is connected to the rest of the system.



Hand-held scanners are commonly seen in big stores to scan codes and price information for each of the items. They are also termed the barcode readers.

Input and output devices are similar in operation but perform opposite functions. It is through the use of these devices that the computer is able to communicate with the outside world.



Task Search about the details of I/O devices.



History of Microcomputer

The term “microcomputer” came into popular use after the introduction of the minicomputer, although Isaac Asimov used the term *microcomputer* in his short story “The Dying Night” as early as 1956 (published in *The Magazine of Fantasy and Science Fiction* in July that year). Most notably, the microcomputer replaced the many separate components that made up the minicomputer’s CPU with a single integrated microprocessor chip.

The earliest models often sold as kits to be assembled by the user, and came with as little as 256 bytes of RAM, and no input/output devices other than indicator lights and switches. However, as microprocessor design advanced rapidly and semiconductor memory became less expensive from the early-to-mid-1970s onwards, microcomputers in turn grew faster and cheaper. This resulted in an explosion in their popularity during the late 1970s and early 1980s.

The increasing availability and power of desktop computers for personal use attracted the attention of more software developers. As time went on and the industry matured, the market for personal (micro) computers standardized around IBM PC compatibles running MS-DOS (and later Windows).

Modern desktop computers, video game consoles, laptop computers, tablet PCs, and many types of hand-held devices, including mobile phones and pocket calculators, as well as industrial embedded systems, may all be considered examples of microcomputers according to the definition given above.

Colloquial use of the term

Everyday use of the expression “microcomputer” (and in particular the “micro” abbreviation) has declined significantly from the mid-1980s onwards, and is no longer commonplace. It is most commonly associated with the first wave of all-in-one 8-bit home computers and small business microcomputers (such as the Apple II, Commodore 64, BBC Micro, and TRS 80). Although—or perhaps because—an increasingly diverse range of modern microprocessor-based devices fit the definition of “microcomputer,” they are no longer referred to as such in everyday speech.

In common usage, “microcomputer” has been largely supplanted by the description “personal computer” or “PC,” which describes that it has been designed to be used by one person at a time. IBM first promoted the term “personal computer” to differentiate themselves from other microcomputers, often called “home computers” and also IBM’s own mainframes and minicomputers. Unfortunately for IBM, the microcomputer itself was widely imitated, as well as the term. The component parts were commonly available to manufacturers and the BIOS was reverse engineered through clean room design techniques. IBM PC compatible

Contd...

Notes

"clones" became commonplace, and the terms "Personal Computer," and especially "PC", stuck with the general public.

Questions:

1. Give brief description of the "home computers."

2. Differentiate between minicomputer and microcomputer.

5.6 Summary

- The term microcomputer is used to describe a system that includes a minimum of a microprocessor, program memory, data memory, and input/output (I/O).
- The computer will be of no use unless it is able to communicate with the outside world. Input/output devices are required for users to communicate with the computer.
- A microcontroller executes a user program which is loaded in its program memory.
- Memory is an important part of a microcomputer system. Depending upon the application we can classify memories into two groups: program memory and data.
- Basically, two types of architectures are used in microcontrollers: Von Neumann architecture and Harvard architecture.

5.7 Keywords

- **Backspace:** This key is used to move the cursor one position to the left and also delete the character in that position.
- **Caps Lock:** Cap Lock is used to toggle between the capital lock features. When 'on', it locks the alphanumeric keypad for capital letters input only.
- **Enter:** It is similar to the 'return' key of the typewriter and is used to execute a command or program.
- **Numeric Keypad:** Numeric keypad is located on the right side of the keyboard and consists of keys having numbers (0 to 9) and mathematical operators (+ - * /) defined on them. This keypad is provided to support quick entry for numeric data.
- **Shift:** This key is used to type capital letters when pressed along with an alphabet key. Also used to type the special characters located on the upper side of a key that has two characters defined on the same key.
- **Spacebar:** It is used to enter a space at the current cursor location.
- **Tab:** Tab is used to move the cursor to the next tab position defined in the document. Also, it is used to insert indentation into a document.



Lab Exercise

1. Give the Connecting the I/O devices into the computer system.

2. Explain the Register in the term of processing.

5.8 Self-Assessment Questions

1. Double Click : Used to a program or open a file.

(a) Start

(b) Select

(c) Move

(d) Open

Notes

2. Scanner is an input device used for direct data entry from the source document into the computer system.
(a) True (b) False
3. Microcomputers are not unique in that the heart of the computer (the CPU) is contained on a single integrated chip.
(a) True (b) False
4. The simplest microcontroller architecture consists of a microprocessor, memory, and input/output.
(a) True (b) False
5. The three components of the microcomputer system is connected by buses.
(a) Two (b) Three
(c) Four (d) Five
6. The address bus is 'unidirectional', over which the microprocessor sends an address code to the memory or input/output.
(a) True (b) False
7. The data bus is 'unidirectional', on which data or instruction codes are transferred into the microprocessor or on which the result of an operation or computation is sent out from the microprocessor to the memory or input/output.
(a) True (b) False
8. Input and output devices in short are also called I/O devices.
(a) True (b) False
9. Input and output devices are not similar in operation but perform opposite functions.
(a) True (b) False

5.9 Review Questions

1. What is the purpose of the Instruction Register (IR)?
2. Draw the programming model of the Z80 microprocessor.
3. What are the differences between a general purpose with a special purpose register?
4. What is the answer register?
5. Explain the I/O devices.
6. Define the Microcontroller system.
7. Explain the Microcomputer system memory.
8. What are the features of Microcontroller system?
9. Describe the architecture of microcomputer system.
10. Describe the parts of microcomputer system.

Answers for Self-Assessment Questions

1. (a) 2. (a) 3. (b) 4. (a) 5. (a)
6. (a) 7. (b) 8. (a) 9. (b)

Notes

5.10 Further Reading



Books

8080A/8085 Assembly Language Programming, Lance A. Leventhal



Online link

<http://books.google.com/books?>

Unit 6: The 8085 Microprocessor Architecture

Notes

CONTENTS

Objectives

Introduction

6.1 The 8085 Architecture

6.1.1 The 8085 Pin Diagram

6.1.2 Pin Details

6.2 Internal Architecture of 8085

6.2.1 Register organization

6.2.2 Timing and Control Unit

6.2.3 Address Buffer and Address Data Buffer

6.2.4 Interrupt Controls

6.2.5 Serial Input/Output Control

6.2.6 Addressing Modes of 8085

6.2.7 Timing Diagram of 8085

6.2.8 Read Cycle of 8085

6.2.9 Write Machine Cycle

6.2.10 Power on Reset of CPU

6.3 Summary

6.4 Keywords

6.5 Self-Assessment Questions

6.6 Review Questions

6.7 Further Reading

Objectives

After studying this unit, you will able to understand the following:

- Discuss the 8085 architecture
- Explain the internal architecture of 8085

Introduction

8085 microprocessor was introduced by Intel in the year 1976. This microprocessor is an update of 8080 microprocessor. The 8080 processor was updated with Enable/Disable instruction pins and Interrupt pins to form the 8085 microprocessor. Let us discuss the architecture of 8085 microprocessor in detail.

Before knowing about the 8085 architecture in detail, lets us briefly discuss about the basic features of 8085 processor. 8085 microprocessor is an 8-bit microprocessor with a 40 pin dual in line package.

Notes

The address and data bus are multiplexed in this processor which helps in providing more control signals. 8085 microprocessor has 1 Non-maskable interrupt and 3 maskable interrupts. It provides serial interfacing with serial input data (SID) and serial output data (SOD). It has a set of registers for performing various operations. The various registers include:

- Accumulator (register A)
- Registers: B, C, D, E, H and L

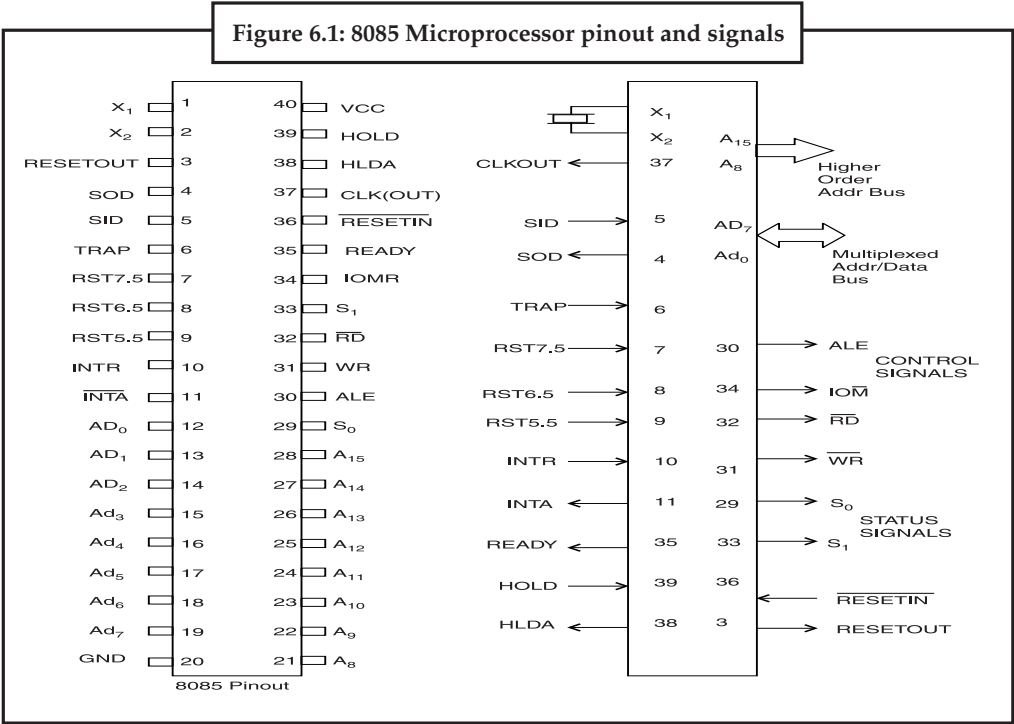
6.1 The 8085 Architecture

The 8085 is an 8-bit microprocessor. This design is the enhanced design of the earlier microprocessor 8080, which had three operating voltages and data lines/address lines independent. In 8085 the bus structure is multiplexed with 8 lower address lines and requires only 5 V as single power supply. The external chips for clock generation and control system in 8080 are not available in 8085 microprocessor these are designed using IVMOS technology. The operating clock frequency for 8085 is faster than the 8080. The operating clock frequencies further high for enhanced versions 8085 AH which is designed using the HMOS technology, the operating frequency is from 3 MHz to 6 MHz.

All the signals can be classified into six groups:

1. Address bus
2. Data bus
3. Control and status signals
4. Power supply and frequency signals
5. Externally initiated signals
6. Serial I/O ports.

6.1.1 The 8085 Pin Diagram



6.1.2 Pin Details

The 8085A (commonly known as 8085) is an 8-bit general purpose microprocessor capable of addressing 64k of memory and consist of approximately 6200 transistors. It is the most popular

and easily available 8-bit microprocessor chip in the market. It is the improved version of the 8080A and the instruction set includes all the 8080A instructions plus some additional instructions. The device has 40 pins and requires a single power supply of +5 volts as shown in Figure 6.1. It has its own built in clock and control circuits. Its clock frequency is 3.125 MHz. It uses a 6.25 MHz crystal to generate the 3.125 MHz clock and the driving frequency must be at least 1.0 MHz.

X1 and X2 (Pin 1 and 2)

The 8085A has an in-built oscillator on the chip excepting the crystal, and an LC tank or RC network to control the frequency of oscillation when connected across X1 and X2 terminals. A 6.25 MHz crystal provides a 3.125 MHz internal clock frequency.

Reset Out (Pin 3)

This signal is taken to RESET peripherals in a microprocessor system. A high on this pin indicates that the microprocessor is being RESET, i.e., all the registers and counters are being RESET to zero. This signal can also be used to RESET other devices in the microprocessor system. The output of this pin remains as long as the RESET IN is kept low. When power supply to the system is switched on, the whole system is RESET or initialized. After the RESET OUT becomes low, the processing starts.

Serial Output Data (SOD) (Pin 4)

The SOD line provides a 1-bit output port on the 8085A. The SIM (Send Interrupt Mask) instruction is essential to output data serially from the SOD line. With the help of SIM instruction the value of D₇ (bit 7) of the Accumulator is loaded in SOD latch only when the D₆ (bit 6) of the Accumulator is logic high i.e., D₆.

If D₆ = 0, the SOD latch remains unaffected. Whenever the 8085A is reset, the SOD latch is also set to logic low i.e., 0 automatically.



The 8085 can access locations with numbers from 0 to 65,536. Or, the 8085 can access a total of 64K addresses.

Serial Input Data (SID) (Pin 5)

Serial input data pin is a 1-bit input port of the 8085A, RIM (Read Interrupt Mask) instruction can be used to transfer the input present in the pin to bit-7 of the Accumulator.

Interrupt (Pin 6 to Pin 10)

The 8085A has five interrupt pins: TRAP (Pin 6), RST7.5 (Pin 7), RST 6.5 (Pin 8), RST5.5 (Pin 9) and INTR (Pin 10).

The TRAP interrupt has the highest priority and is a non-maskable interrupt. It is both edge and level sensitive. The second priority is assigned to RST7.5, which is positive edge sensitive only. The next two interrupts i.e., RST6.5 and RST5.5 are both level sensitive. The INTR is the lowest priority interrupt request in 8085A and is used as the general-purpose interrupt. A logic high i.e., INTR = 1 on this pin, indicates that some of the peripheral devices are requesting the microprocessor to stop the execution of the program in sequence and to do some other work for the peripherals.

Different interrupts have different memory locations fixed for transfer of control from the normal execution of the routine. The above-mentioned interrupts except INTR and TRAP, are called vector interrupts or Restart interrupt (RST). Each of the vectors interrupts points to a particular memory location. As soon as the signals appear on these interrupts, the normal execution of the program stops and the program control is transferred to the corresponding memory locations. They have higher priorities than the INTR interrupt. Among these three, the priority order is 7.5, 6.5 and 5.5.

Notes

INTA (Pin 11)

It stands for interrupt acknowledge. A low on this pin indicates that the microprocessor has acknowledged a request from some of the peripheral devices and is also used to activate the interrupt controller.

Multiplexed Address/Data bus (Pin 12 to Pin 19)

The pins 12 to 19 serve the dual purpose of transmitting the lower order address and data type is time division multiplexing. During the 1st clock of each machine cycle, the lower order address byte is output on these lines. In the remaining part of the clock of the machine cycle, it becomes the data bus. The lower order address and data bus can be separated using an address latch. These are bidirectional bus. ($AD_0 - AD_7$)

Latch

It is commonly used to connect the output devices to the microprocessor. The data is available on the address/data bus for only a few microseconds. The data in the form of address must be present on the address bus so long as the reading or writing operation is not complete or otherwise wrong data transfer will take place. Since the lower order address byte is present in the 1st clock of the machine cycle, it must be latched (stored to hold the lower order address byte). V_{ss} (Pin 20) is used for ground references.

Address Bus (Pin 21 to Pin 28)

8085A has eight unidirectional (A_8 to A_{15}) signal lines corresponding to; pin 21 to pin 28. It outputs 8-MSB's (Most Significant Bit) i.e., higher order address bus of the memory address. It remains in the high impedance state during HOLD, HALT and RESET modes.

Control and Status Signals

\overline{RD} and \overline{WR} are two control signals and $10/\overline{M}$, S_1 and S_0 are the three status signals. The ALE is a very special control signal, which indicates the beginning of the operation.

S_1 (Pin 33) and S_0 (Pin 29)

S_0 and S_1 are the status signals similar to $10/\overline{M}$, along with IO/\overline{M} , S_0 and S_1 are used to identify various operations as indicated in the Table 6.1:

Table 6.1: 8085 Machine Cycle, Status and Control Signals				
Machine cycle	$10/\overline{M}$	S_1	S_0	Control signals
Opcode Fetch	0	1	1	$\overline{RD} = 0$
Memory Read	0	1	0	$\overline{RD} = 0$
Memory Write	0	0	1	$\overline{WR} = 0$
I/O Read	1	1	0	$\overline{RD} = 0$
I/O Write	1	0		$\overline{WR} = 0$
Interrupt Ack	1	1	1	$\overline{INTA} = 0$
Halt	Z	0	0	
Hold	Z	X	X	$\overline{RD}, \overline{WR} = Z$
Reset	Z	X	X	$\overline{INTA} = 1$
Z = High Impedance State or tristate X = Unspecified				

ALE (Pin 30)

All memory chips have a memory address register (MAR), also called the address latch. This stores the address from the address bus and is connected to the memory chip. The falling edge of the ALE signal loads the address bus into the MAR of the memory chip. ALE stands for address latch enable and is used to indicate that the information carried on the multiplexed address/data bus (AD_0 to AD_7), is the lower order address byte when a logic high condition appears on this line. Hence this control line is used to latch the lower order address A_7 to A_0 from the multiplexed address/data bus in the 1 clock of the machine cycle.

 \overline{WR} (Pin 31)

A low condition i.e., 0 on the \overline{WR} pin intends to output (write) data on the data bus are to be written into the selected I/O devices or memory.

 \overline{RD} (Pin 32)

A low condition i.e., 0 on the \overline{RD} pin indicates that the microprocessor intends to receive (read) the data from the I/O devices or memory locations and data are available on the data bus.

 $1O/\overline{M}$ (Pin 34)

This pin indicates whether the address on the address bus is meant for the I/O devices or for the memory locations. A logic high i.e., 1 on this pin indicates that the address on the address bus is meant for the I/O devices while the logic low i.e., 0 on this pin indicates that the address on the address bus is meant for the memory locations.

Ready (Pin 35) (Input)

The I/O or memory devices are not as fast as the microprocessor and a mechanism is required to tell the microprocessor that the data from the I/O devices or from the memory are not readily available at a particular moment. It indicates the microprocessor to wait up to the time the data becomes available. When $READY = 0$, the microprocessor waits till $READY = 1$ and when $READY = 1$, the microprocessor knows that the data is now available from the I/O devices or from the memory.

Reset In (Pin 36)

When the signal on this pin becomes low for at least 600 nano seconds, it forces the microprocessor to do the following:

- $(PC) = 0$
- Instruction register cleared
- All interrupts (except TRAP) disabled
- $SOD = 0$
- Data, address and control bus are floated

CLK OUT (Pin 37)

The clock signal is derived from the on chip: oscillator and the CLK derive the peripherals to synchronize their timings. This signal can be used as the system clock for other devices.

Hold (Pin 38) (Output)

In order to speed up the data transfer between memory and a peripheral device, the microprocessor need not be involved. This is implemented by the DMA (Direct memory address). Hence HOLD and HLDA control signals are required for the DMA operation. A logic high i.e., 1 on this pin input by any I/O device indicates that the data is ready for DMA transfer.

HLDA (Pin 39) (Input)

This stands for hold acknowledge. A logic high condition on this pin indicates; that the microprocessor has received the information of request from the I/O devices and will relinquish the data, address and control buses after completion of the current bus transfer.

Notes



The lower order address byte is present in the 1st clock of the machine cycle; it must be latched (stored to hold the lower order address byte). V_{ss} (Pin 20) is used for ground references.

V_{cc} (Pin 40)

An external DC supply voltage of +5V (17 milli-amps) is connected to this pin for the operation of the 8085A as shown in Table 6.2.

Table 6.2: Pin description

Pin number	Pin name	Description	Type
1, 2	X_1, X_2	Crystal	Input
3	Reset out	Peripheral Reset Output Signal	Output
4	SOD	Serial output data	Output
5	SID	Serial Input data	Input
6	TRAP	Non-maskable interrupt	Input
7	RST 7.5	Hardware vectored interrupt	Input
8	RST 6.5	Hardware vectored interrupt	Input
9	RST 5.5	Hardware vectored interrupt	Input
10	INTR	Interrupt Request	Input
11	\overline{INTA}	Interrupt Acknowledgement	Output
12-19	AD_0-AD_7	Address/ Data Bus	Bidirectional, tristate
20	V_{ss}	Ground	Input
21-28	A_8-A_{15}	Address Bus (Higher byte)	Output, tristate
29	S_0	Bus state	Output
30	ALE	Address Latch Signal	Output, tristate
31	\overline{WR}	Write control signal	Output, tristate
32	\overline{RD}	Read control signal	Output, tristate
33	S_1	Bus state	Output
34	IO/\overline{M}	I/O or memory status	Output, tristate
35	READY	Wait state request	Input
36	RESET IN	System Reset	Input
37	CLK (OUT)	Clock signal	Output
38	HLDA	Hold Acknowledgment	Output
39	HOLD	Hold Request	Input
40	V_{cc} (+5V)	Power	



Task Give the basic pin description of the microprocessor architecture.

6.2 Internal Architecture of 8085

The 8-bit microprocessor 8085 was most popular among the available series. The internal architecture of 8085 microprocessor can be shown in Figure. 6.2. The architecture can be divided into the following six sections:

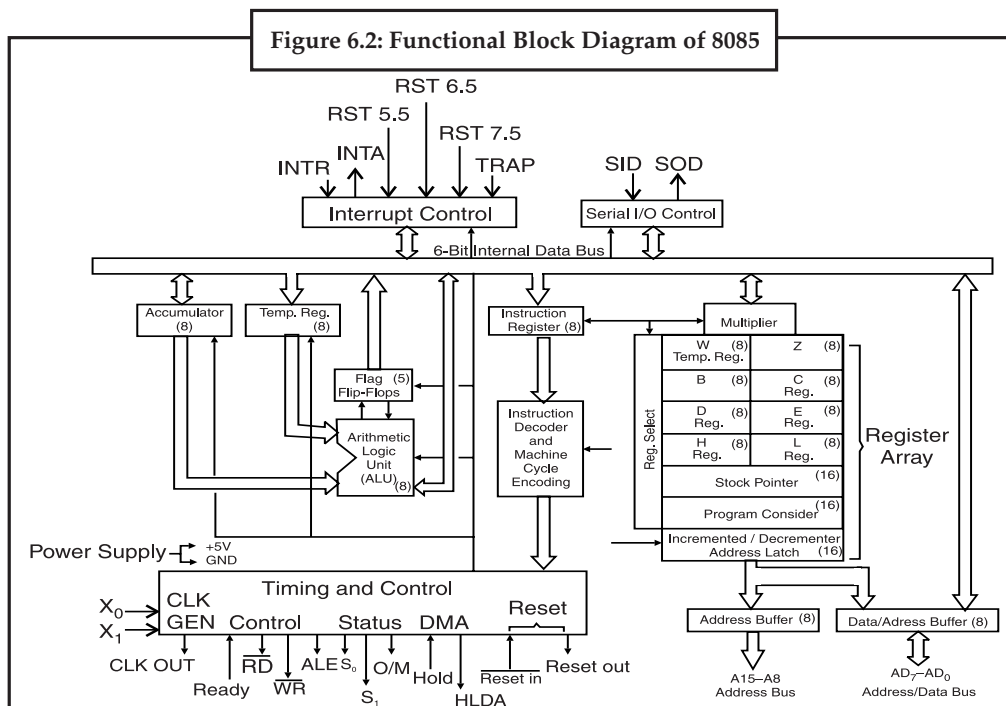
1. Register organization
2. Timing and control unit
3. Address buffer and address/data buffer
4. Interrupt control
5. Serial Input/output control
6. Arithmetic and logical unit (ALU)

6.2.1 Register organization

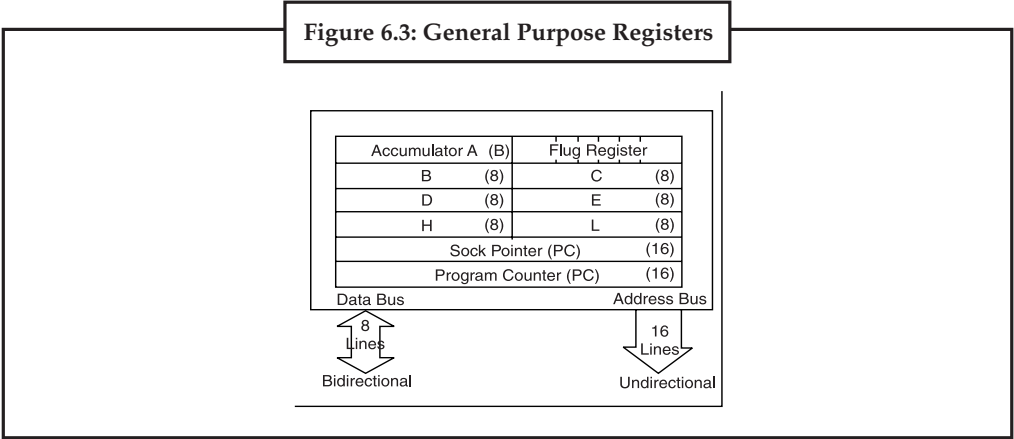
A register is a very small storage area. Most of the 8085A store only one byte of data. It is used for temporary storage of instructions, data or address. In most of the 8-bit microprocessor, the registers are either 8-bit for data storage or 16-bit for address storage.

The Figure 6.3 shows the registers and sources of address for the address register. Every microprocessor is provided with a set of registers for temporary storage of information at various stages of instruction execution. The number and types of registers available for temporary storage of operands or address affects the following:

- Memory space occupied by the program
- Time of execution of the program
- Ease of programming



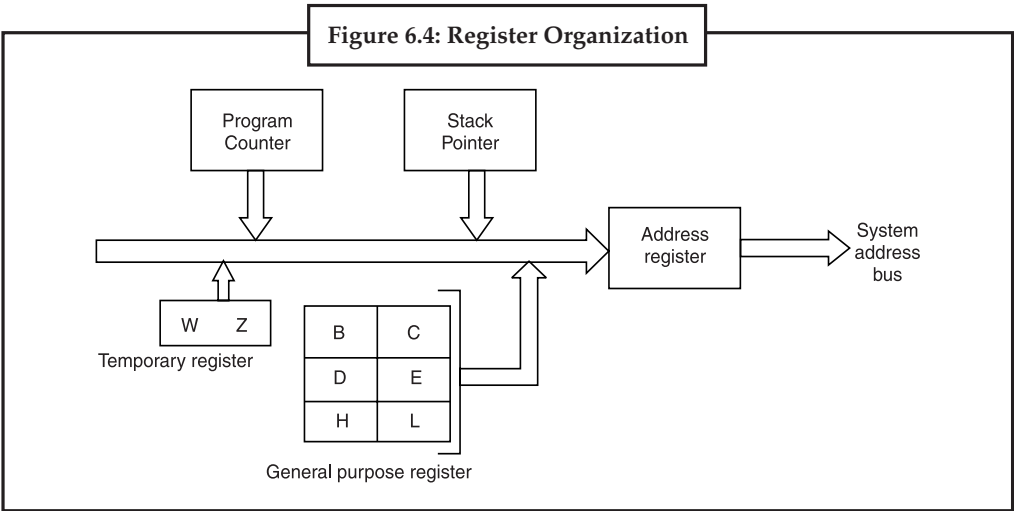
Notes The register available inside the microprocessor can be classified into; register accessible to user and registers not accessible to the user. The registers available to the user can be further classified into; general purpose registers and special purpose registers.



Differentiate between internal and external architecture of microprocessor in the term of pins.

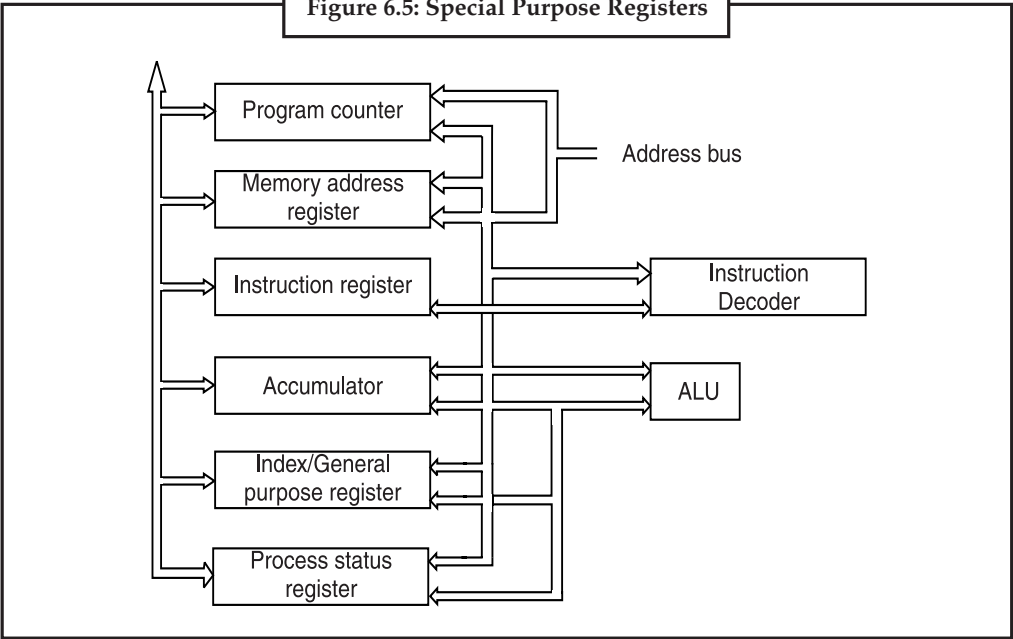
(a) General Purpose Registers: The general-purpose registers are used for storing data as well as the address. 8085A has six general-purpose registers. They are B, C, D, E, H and L. In 8085A microprocessor, the data size is 8-bit and address size is 16-bit. Hence they can be used in pairs like B-C, D-E and H-L to store 16-bit address as show in Figure 6.4.

When these registers are used in pair the higher byte will reside in first registers i.e., B, D and H and lower byte in second registers i.e., C, E and L. For example in case of B-C pair the higher byte will reside in B and lower byte in C register. The register pair H-L function as data pointer and can holds memory addresses in the register indirect addressing mode.



(b) Special Purpose Registers: They are a set of registers provided for some specific applications. Some of the special purpose registers are: Accumulator (A), Program Counter (PC), Stack Pointer (SP) and Status Flags. Thus the microprocessor has six programmable registers inside the microprocessor as shown in Figure 6.5.

Figure 6.5: Special Purpose Registers

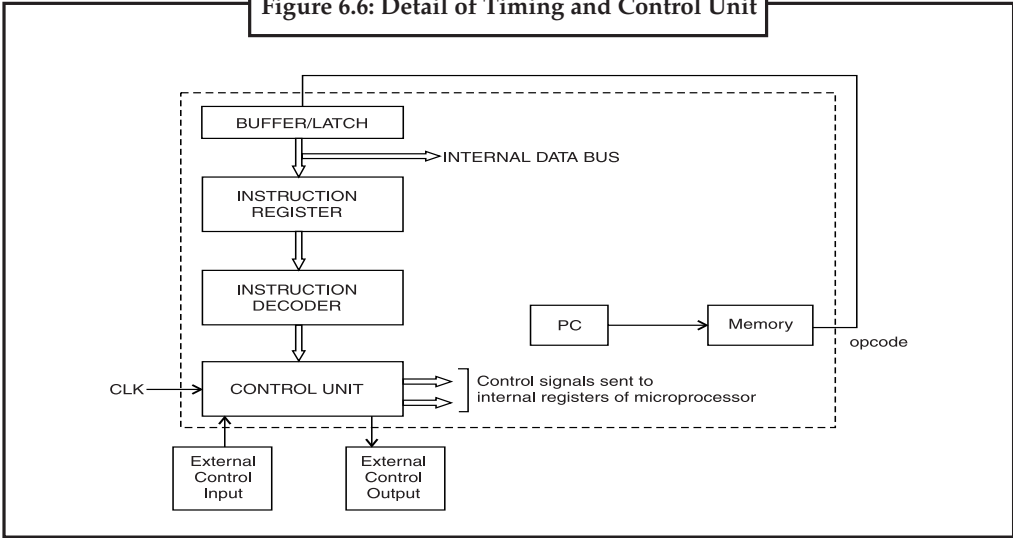


6.2.2 Timing and Control Unit

This unit controls and synchronizes all the operations inside and outside of the microprocessor in the system. The timing and control signals that regulate the transfers and transformations in the system associated with each instruction are derived from the master clock (CLK). The control unit also accepts the control signals generated by other devices associated with the microprocessor system. These control signals generated from outside of the microprocessor also alters the state of the microprocessor as shown in Figure 6.6.

Most of the microprocessor utilizes an external quartz crystal to determine the clock frequency (CLK) from which other timing and control signals are developed. The speed of the microprocessor is related directly to the clock speed, since most of the internal functions are timed by the reference signal.

Figure 6.6: Detail of Timing and Control Unit



Notes

6.2.3 Address Buffer and Address Data Buffer

The contents of the stack pointer or program counter can be loaded into the address buffer and address-data buffer. The output of these buffers then drives the external address bus and address-data latches. Memory and I/O chips are also connected to these buses through buffers decoders. The CPU can thus send the address of the desired data to the memory and I/O chips. The 8-bit internal data bus is also connected to the address-data buffer. The input to this section is from incrementer/decrementer section below registers.

6.2.4 Interrupt Controls

To answer a request from an I/O device, it is sometimes necessary to interrupt the execution of the main program currently being executed. The CPU temporarily stops execution and attends the I/O device which has interrupted, the CPU, then returns to what it was doing.

The 8085 A has five hardware interrupts: INTR, RST 5.5, RST 6.5, RST 7.5 and TRAP. These can be classified into three types depending on their maskability. The first of its types is INTR. INTR (Input) i.e., Interrupt request. This is a general purpose interrupt. INTA (Output) i.e., Interrupt acknowledge. This is used to acknowledge an interrupt. In the second group is RST 5.5, RST i-5 and RST 7.5. These are input-restart interrupts. These are vector interrupts that transfer the program control to specific memory locations. They have higher priorities than the INTR interrupt: Among these three, the priority order is 7.5, 6.5 and 5.5. These are vector interrupt. The TRAP is the third kind of hardware interrupt.

If two or more interrupt go high at the same time, the CPU will service them in the order of their priority. TRAP has the highest priority, which is non-maskable. The priority is in Table 6.3:

Table 6.3: Interrupt Priorities		
Interrupt Name	Priority of Interrupt	Call Location
TRAP	1	0024H
RST 7.5	2	003CH
RST 6.5	3	0034H
RST 5.5	4	002CH
INTR	5	—

6.2.5 Serial Input/Output Control

The 8085 CPU works on 8-bit parallel data. Sometimes I/O devices with serial data, when want to communicate, the serial data stream from an input device must be converted to 8-bit parallel data before this input enters the 8085 and the CPU can use. Same way the 8-bit data out of CPU must be converted to serial form before it is fed to a serial output device. The SID input is used where serial data enters the 8085 and SOD output is used where serial data leaves the 8085 to be used for the I/O devices which work with the serial data.

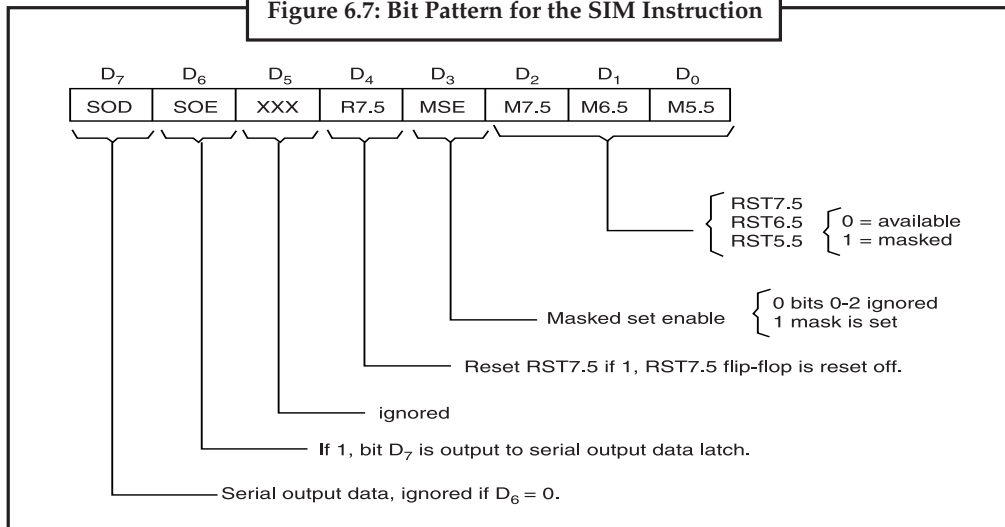
The SIM (Set Interrupt Mask) and RIM (Read Interrupt Mask) instruction allows to perform the conversion needed for serial I/O devices.

SIM is a 1-byte instruction can be used for three different functions:

- (a) Set mask for RST 7.5, 6.5 and 5.5 interrupts
- (b) Reset RST 7.5 flipflop.
- (c) Implement serial I/O.

Bit D_7 and D_6 of the Accumulator are used for serial I/O and do not affect the interrupts. Bit $D_6 = 1$ enables the serial I/O and bit D_7 is used to transmit output bits.

Figure 6.7: Bit Pattern for the SIM Instruction



RIM is a I-byte instruction that can be used for three functions:

1. To read interrupt masks
2. To identify pending interrupts
3. To receive serial data.

Bit D_7 is used to receive serial data. Bits D_4 , D_5 and D_6 identify the pending interrupts as shown in Figure 6.7. The RIM instruction loads the Accumulator with the following information.

6.2.6 Addressing Modes of 8085

Addressing is the technique to transfer data from memory or I/O ports to or from CPU for execution of an instruction. There are five type of addressing used in 8085 microprocessor system. They are:

- Immediate Addressing:** This is the technique of addressing by which the data to be operated upon by the CPU is given to processor without using memory locations during transfer. In this mode 8-bit data is given to processor and uses 2-byte instructions, the first byte is opcode and second byte is 8-bit data. The 16-bit data can also be loaded by using 3-byte instruction. The first byte is the opcode while 2nd and 3rd bytes are the address of memory locations. For example MVI B, 02H loads register B with value 02H and LXI H, 2050H loads register H with 20H and register L with 50H.
- Direct Addressing:** It is the mode of addressing in which the address of memory location where data is residing is given in the instruction. These instructions contain three bytes. Where first byte is opcode and two bytes are for address of data the lower byte of address is in byte 2 and higher order byte is in byte 3 of instruction. For example LDA 2050H loads the Accumulator with data contents available at 2050H, STA 2051H transfers 8-bit data from Accumulator to 2051H address of memory.
- Register Addressing:** In this mode of addressing the operands resides in the general purpose registers the operands are moved within the registers by using opcodes. For example: MOVA, B moves the contents of the register B into Accumulator and instruction is in the coded form. This means that the instruction is single byte instruction and opcode itself is having register address. Another example may be ADD B which adds contents of register B to the contents of the Accumulator and result is stored in the Accumulator.

Notes

(d) **Register Indirect Addressing:** In this mode of addressing the operands are addressed through register pairs. It means that the data location is specified by indirect method. The register pair is having address of memory location from where data is to be accessed. For example LXI H, 2050H which loads the HL register pair with 2050H and MOVA, M moves the contents of the memory location whose address is in the HL pair to the Accumulator. ADD, M is another example of this mode of addressing.

(e) **Implicit Addressing:** This is the mode of addressing in which the contents data of the Accumulator are operated upon and does not require the address of the operands. For example the instructions CMA, RAL, RAR etc. does not require other operands then the available in the Accumulator.

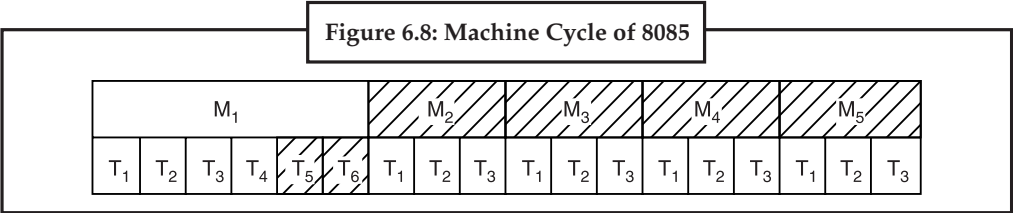


Program, data and stack memories occupy the same memory space. The total addressable memory size is 64 KB.

6.2.7 Timing Diagram of 8085

For execution of an instruction, a microprocessor fetches the instruction and executes it. The time taken for the execution of an instruction is called Instruction Cycle. An instruction cycle consists of a fetch cycle and executes cycle. The execution of any program consists of READ or WRITE operations, of which each transfer a byte of data between the CPU and a particular I/O device or memory.

Each READ or WRITE operation of the CPU is referred to as Machine Cycle. Machine cycle is defined as the time required to complete one operation of accessing memory, I/O or acknowledging an external request. An instruction’s execution consists of a number of machine cycles. These cycles vary from one to five depending on the instruction. Each machine cycle contains a number of clock cycles which are called states. T-State is defined as one subdivision of the operation performed in one clock period. The first machine cycle will be executed by either four or six clock periods or the machine cycles that follow will have three clock periods. The 8085A machine cycle has been shown in Figure 6.8.



The shaded clock periods (T₅ and T₆) mean that these are needed in M₁ by some of the instructions.

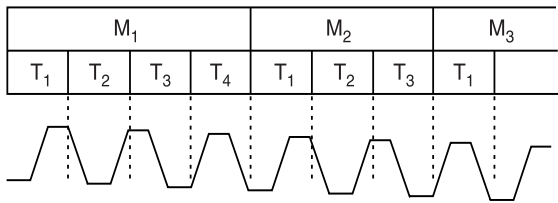
The control and statue signals are shown below:

Machine	Status				Control	
	IO/ \overline{M}	S ₁	S ₀	\overline{RD}	\overline{WR}	$\overline{INT A}$
Opcode Fetch(OF)	0	1	1	0	1	1
Memory Read	0	1	0	0	1	1
Memory write	0	0	1	1	0	1
I/O Read	1	0	0	1	1	
I/O Write		0	1	1	0	1
INTR ACK	1	1	1	1	1	0
HALT	TRISTATED	0	0	TRISTATE	TRISTATE	1

Notes

The clock periods within a machine cycle can be illustrated as shown in Figure 6.9. The beginning of a new machine cycle is indicated by outputting the address latch enable' signal 'High' and during this time, ADO- AD7are used for placing the low byte of the address. When the ALE goes 'low', the low byte of the address are Latched so that the ADO - AD7 lines can be used for transferring data.

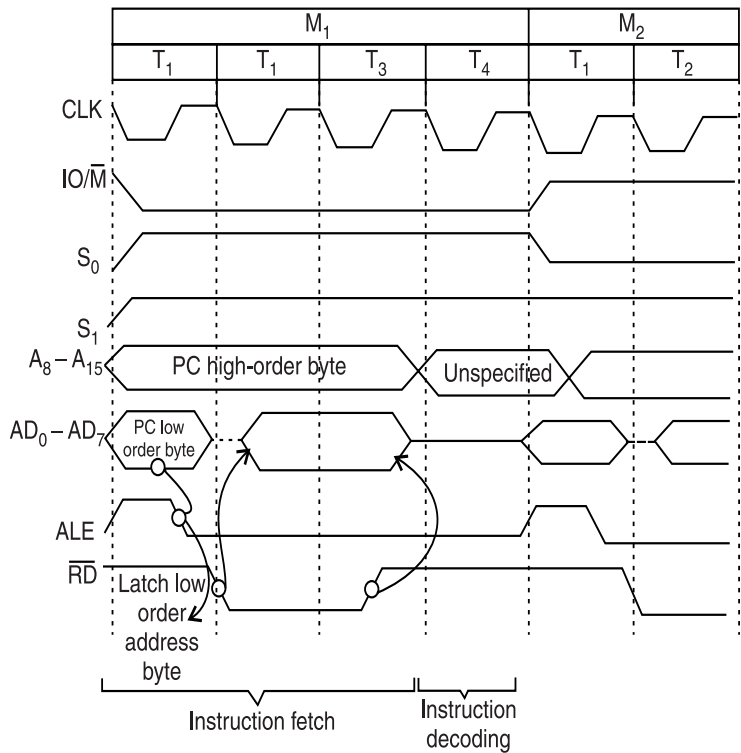
Figure 6.9: Clock Periods of a Machine Cycle



OPCODE Fetch

The instruction fetch cycle requires either four or six clock periods as shown in Figure. 6.10. The machine cycles that follow will need three clock periods. The purpose of an instruction fetch is to read the contents of a memory location containing an instruction addressed by the program counter and to place it in the Instruction Register.

Figure 6.10: Opcode Fetch Timing Diagram of 8085



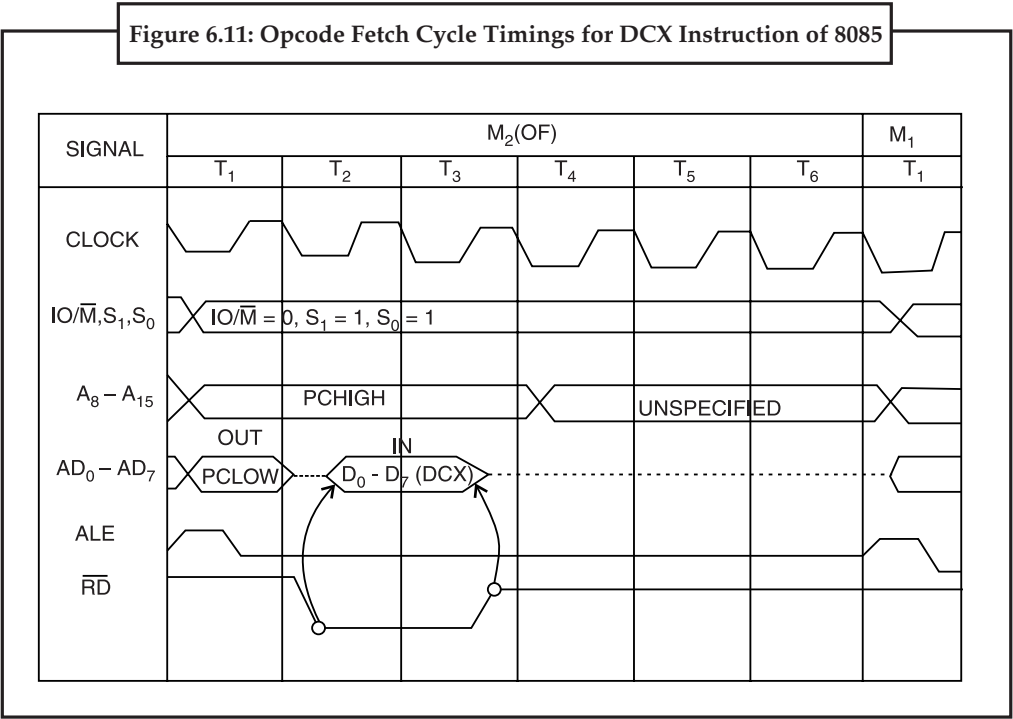
Notes

- The instruction fetch timing diagram can be in various steps.
- Initially the $10/\overline{M}$ are low and S_0, S_1 are high to indicate the opcode fetch operation. In this state ALE is high and PC 's higher byte is placed on the A_8-A_{15} address lines and lower byte is placed on to A_0-A_7 by demultiplexing AD lines.
 - In the second state the CPU sends low on \overline{RD} pin by which $MEMRD$ signal is generated for read operation.
 - In third state the opcode is read from memory location addressed by program counter. The read signal is high at the end of this cycle.
 - In fourth state the opcode is decoded and is ready to receive next instruction or machine cycle.

Machine Cycle for DCX: Figure 6.11 shows the timing for an opcode fetch machine cycle. The particular instruction is DCX (decrement register pair,) whose timing for the opcode fetch differs from other instructions as it has six clock periods, while other instructions require only four clock periods for opcode fetch machine cycle.

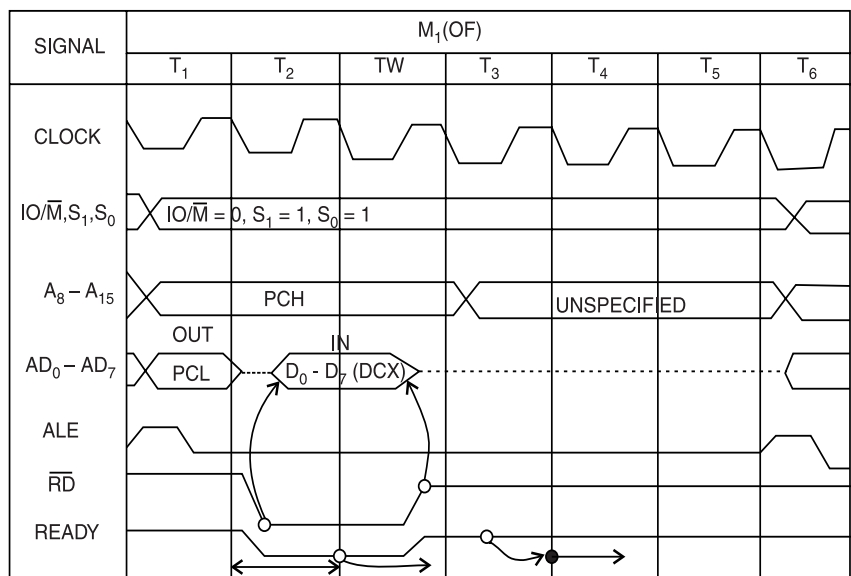
In the first state the CPU sends out three status signals ($10/\overline{M}, S_0, S_1$) that defines the type of machine cycle which is about to take place. As shown in Figure. 6.12 the CPU will send out $10/\overline{M} = 0; S_1 = S_0 = 1$ at the beginning of the machine cycle to identify it as $READ$ from a memory location to obtain an opcode from memory location.

In this case the contents of the program counter are placed on the address bus. The high order byte is placed on the A_8-A_{15} lines, whose three-state drivers are enabled the lower byte of address is put on the time multiplexed lines (AD_0-AD_7) for one clock cycle unlike the PCH . The AD lines latch the address by ALE signal.



Since the address information on AD_0-AD_7 is of a transitory nature, \overline{RD} remains high; therefore, at this stage memory reading does not take place.

Figure 6.12: Opcode Fetch Machine Cycle with One wait State



In the second state the lower address byte (PCL) disappears from AD₀-AD₇ lines. However, A₀A₇ information remains available. Now \overline{RD} signal from CPU goes low which enables memory devices and opcode for DCX is put on to the data lines.

During third state the CPU loads opcode into instruction register and \overline{RD} signal is made, high which disables addressed memory device.

During fourth state (T₄), the opcode -is decoded by the instruction decoder and enters into the fifth state in this case.

During fifth and sixth states (T₅ and T₆) CPU determines register and decrements it which internally transfers to INC/DCR circuit of CPU.

Opcode fetch with wait state: When extra time is required to fetch opcode from slow memory devices, the CPU should get READY signal low while checking it in second state (T). It introduces one wait cycle between T₂ and T₃. When it receives it high on READY then only it proceeds to the next state.

6.2.8 Read Cycle of 8085

There are two types of read operations associated with this micro-processor

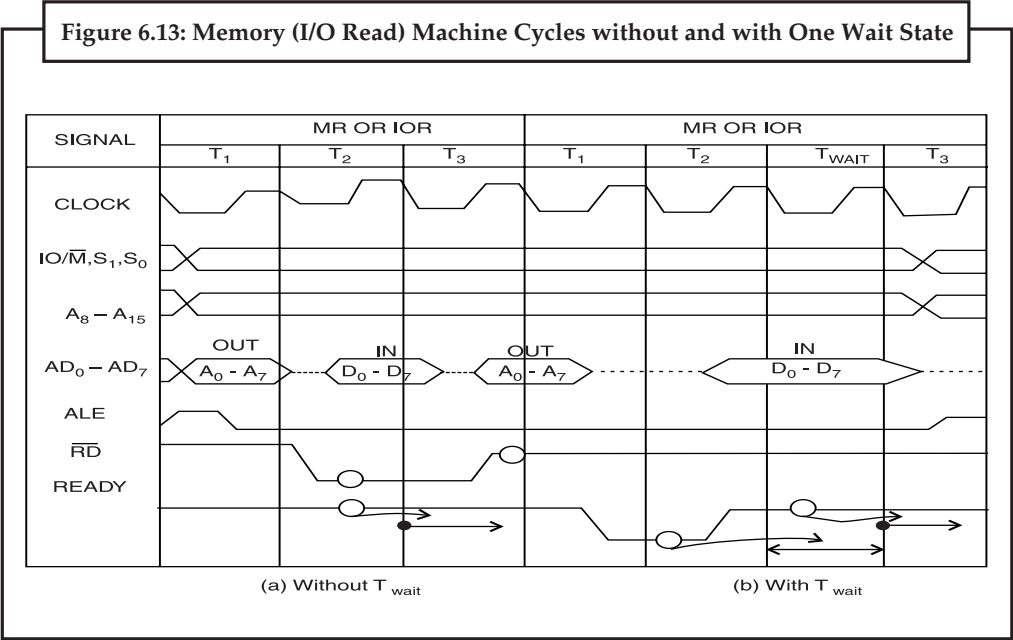
- (i) Memory Read, (ii) I/O Read operation

Memory Read Cycle

We can consider two successive memories read machine cycles which are shown in Figure 6.13 without T_{wait} state and second with one T_{wait} state. The timing during T₁ - T₃ is absolutely identical to the Opcode Fetch (OF) machine cycle, with the exception that the status sent out during T₁ is IO/ M= 0; S₁=1; S₀= 0, identifying the cycles as a READ from a memory location. The memory address used in OF cycle is always the contents of the program counter, which points to the

Notes

current instruction, while the address used in the Memory read cycle can, have several possible origins. The data read during memory read cycle is placed in the appropriate register and not in the instruction register.



I/O Read Machine Cycles

The I/O read machine cycle can be understood by considering Figure 6.13 where two successive I/O read cycles are shown one without wait state and second with wait state. The \overline{IOR} machine cycle is identical to Memory read as explained earlier except status line IO/ M is 1 in place of 0.

6.2.9 Write Machine Cycle

Similar to the read operation, there are two types of write operations which are:

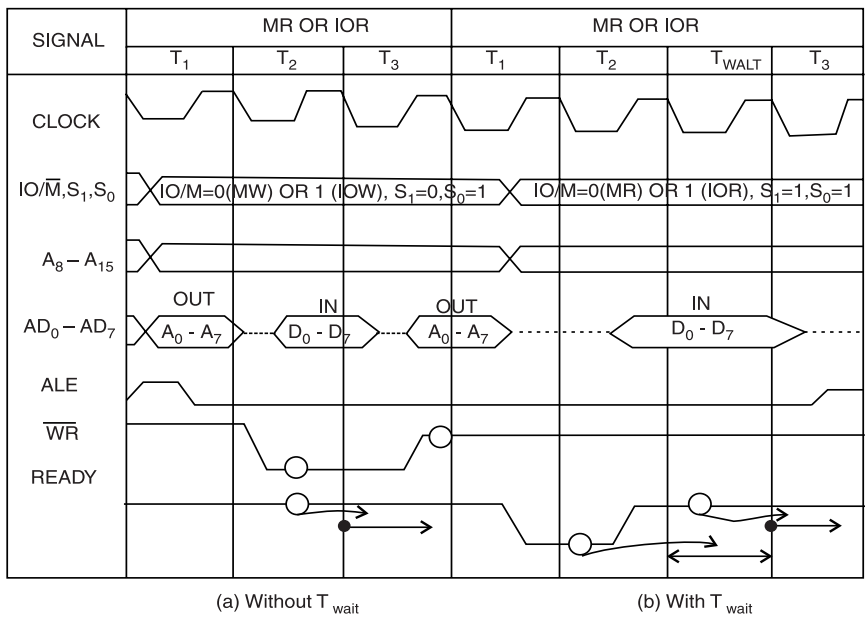
- (i) Memory write (ii) I/O write

Memory Write Machine Cycle

Figure 6.14 shows the timings for two successive Memory Write machine cycles, the first without a wait state and second with one wait state. They are similar to Memory Read except for the values of IO/ \overline{M} , S₁, S₀, \overline{RD} and \overline{WR} signals. In MW, IO/ \overline{M} = 0, S₁ = 0, S₀ = 1. In READ cycle, at the end of T₁, the AD₀-AD₇ drivers are disabled during T₂-T₃ of data output from the memory, but in a WRITE cycle, the AD₀ - AD₇ drivers are kept active and not disable because the data output takes place from the CPU through the AD₀ - AD₇ lines during T₂ for writing into the addressed memory locations; and \overline{WR} is lowered so that writing 0 can take place. During third state, \overline{WR} goes high, which disables the memory device and terminates the WRITE operation. During the next state, T₁ of the next machine cycle is started. Till the start of the new state the contents of the address and data lines unchanged.

Notes

Figure 6.14: Memory write I/O write Machine Cycles without and with Wait States



The READY line is checked during T₂ to see whether a wait state is required. Extension due to insertion of wait state is identical as in memory or I/O read cycles.

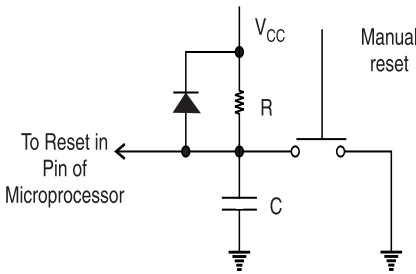
I/O Write Machine Cycle

The timing for an I/O Write machine cycle is the same as in memory write machine cycle except that $\overline{IO/M} = 1$ during the IOW cycle. The address used in an IOW cycle is the I/O port number which is duplicated on high and low bytes of the address bus. In the case of IOW, the port number comes from the second byte of an OUTPUT instruction as the instruction is executed.

6.2.10 Power on Reset of CPU

The 8085 microprocessor employs an external RC circuit for initializing microprocessor by resulting it along with peripheral chips when power supply is switched on. Initially when it is switched on the capacitor with reset in pin is discharged and starts charging during which it gets low input signal which causes CPU to reset and high signal is available at reset out pin which is used to reset the peripheral chips. The capacitor when charged will produce high logic causing CPU to come out of reset state and is ready to receive further has instructions. When CPU is resetting all buses are in floating condition and function internal flip flops are synchronized for use with various register operations and machine cycles as show in Figure 6.15.

Figure 6.15: Power on Reset Circuit



Notes



Virtual 8085

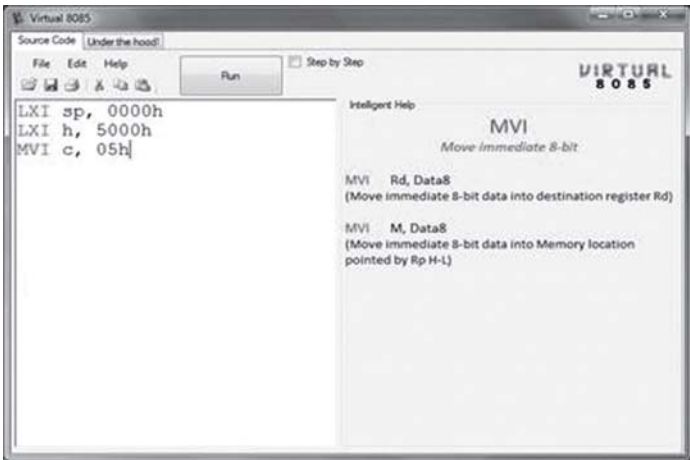
Virtual 8085 is a tool which enables students to run programs written in 8085 assembly language on a personal computer instead of a microprocessor kit. Virtual 8085 do not actually simulate the real hardware of Intel 8085, but it interprets the 8085 assembly language programs.

When we study the C language, we always have a compiler and a computer to practice our programs at home. In case of 8085 microprocessor and its assembly language it is totally different. If we write an 8085 assembly language program, we have to test it or run it on a microprocessor kit kept in our college labs. And for no reason one would want to purchase a costly microprocessor kit to study at home.

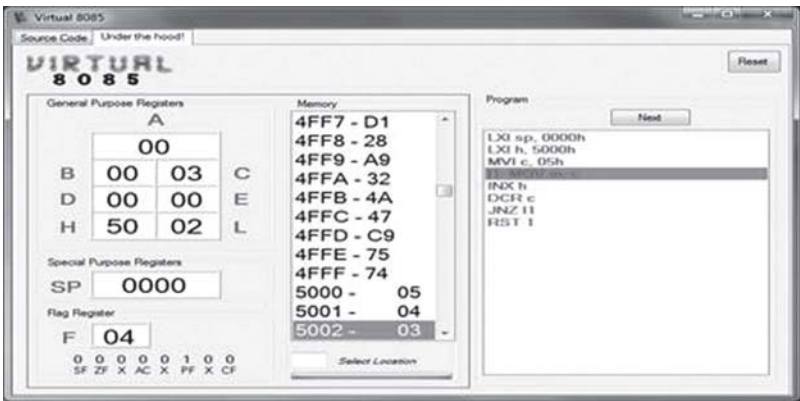
Also, Virtual 8085 updates the graphical user interface (GUI) after each instruction, so that students understand what each instruction performs. Virtual 8085 also provide real time help for the instruction which the programmer is currently typing. Another useful feature is syntax highlighting which enables better readability of program. Virtual 8085 do not need to take the program in form of op-codes, it understands mnemonics.

Another part of Virtual 8085's agenda is to make easier, the learning process of the 8085 assembly language. We achieve this by providing real time and intelligent help (something similar to the IntelliSense feature of Visual Studio IDE). One of our secondary objectives is to make Virtual 8085 free and open source forever. This will enable people to use it freely and other talented guys to develop it even further.

Input Screen



Output Screen



Contd...

Questions:

1. Explain the Virtual 8085 tool.
2. How to make easier the learning process of the 8085 assembly language?

Notes**6.3 Summary**

- In 8085 the bus structure is multiplexed with 8 lower address lines and requires only 5 V as single power supply.
- The TRAP interrupt has the highest priority and is a non-maskable interrupt.
- 8085A has *eight unidirectional* (A_8 to A_{15}) signal lines corresponding to; pin 21 to pin 28.
- 8085A store only one byte of data. It is used for temporary storage of instructions, data or address.
- Program Counter (PC) is common to all microprocessors and is called a *memory pointer*.

6.4 Keywords

- **Immediate Addressing:** This is the technique of addressing by which the data to be operated upon by the CPU is given to processor without using memory locations during transfer
- **MAR:** Memory chips have a memory address register (MAR), also called the *address latch*. This stores the address from the address bus and is connected to the memory chip.
- **SID:** Serial input data (SID) pin is a 1-bit input port of the 8085A
- **SOD:** The Serial Output Data (SOD) line provides a 1-bit output port on the 8085A
- **SP:** Stack pointer is a dedicated 2-byte (16-bit) register. A stack is set of memory locations in the RAM area, which is specifically used to store certain temporary data. The starting address of the stack area is stored in the 16-bit CPU register called the *stack pointer*

*Lab Exercise*

1. Write fetching instruction opcode of 8085.
2. Show the read cycle of 8085 microprocessor.

6.5 Self-Assessment Questions

1. The instruction is essential to output data serially from the SOD line.
 - (a) SIM
 - (b) RESET
 - (c) RST
 - (d) None of these
2. is commonly used to connect the output devices to the microprocessor.
 - (a) Address bus
 - (b) Latch
 - (c) Register
 - (d) Buffer
3. 8085A has signal lines corresponding to; pin 21 to pin 28.
 - (a) five unidirectional
 - (b) six unidirectional
 - (c) seven unidirectional
 - (d) eight unidirectional
4. A stack is set of memory locations in the RAM area, which is specifically used to store certain.
 - (a) Permanent data
 - (b) Secondary data
 - (c) Temporary data
 - (d) All of these

Notes

5. The first step in the execution of an instruction by the CPU is
 - (a) fetching the instruction
 - (b) searching the instruction
 - (c) copy the instruction
 - (d) copy the instruction from register
6. Which flag is set (1) if the ALU operation results in the Accumulator being equal to 0 else the flag will be reset (0), if the result is not 0
 - (a) sign flag
 - (b) status flag
 - (c) carry flag
 - (d) zero flag

6.6 Review Questions

1. What are the various registers of 8085? Discuss their function?
2. What is the function of HOLD and HLDA?
3. What is function of ALE signal in 8085?
4. Discuss the function of ALU of 8085.
5. Explain the generation of control signals in 8085.
6. Explain clock generation of 8085CPU.
7. What is timing diagram? What do you mean by machine cycle, fetch cycle, and T-state?
8. Discuss opcode fetch operation with timing diagram.
9. Discuss the timing diagram of I/O read and I/O write operation.
10. Explain with neat diagram the pin structure of 8085.
11. Discuss with functional diagram the internal architecture of 8085.

Answers for Self-Assessment Questions

1. (a)
2. (b)
3. (d)
4. (c)
5. (a)
6. (d)

6.7 Further Reading



Books

Microprocessor architecture, programming, and systems featuring the 8085, William A. Rott



Online link

<http://books.google.com/books?>

Unit 7: Memory Interfacing

Notes

CONTENTS

Objectives

Introduction

- 7.1 Memory Structure and its Requirements
- 7.2 Basic Concepts in Memory Interfacing
 - 7.2.1 Address Decoding
 - 7.2.2 Interfacing the 2764 EPROM
 - 7.2.3 Interfacing CMOS 6116 Static R/W Memory
 - 7.2.4 Memory Address Range
 - 7.2.5 Designing Memory for the MCTS Project
- 7.3 The 8085 MPU
- 7.4 Microprocessor Based Controllers
 - 7.4.1 Single-board Microcomputers (Microcontrollers)
 - 7.4.2 Input and Output
 - 7.4.3 Bus
 - 7.4.4 Communications and User Interfaces
 - 7.4.5 Programming
 - 7.4.6 EPROM Burning
 - 7.4.7 Keypad Monitors
 - 7.4.8 Single-chip Microcontrollers
 - 7.4.9 Program Memory
 - 7.4.10 Single-board Microcontrollers Today
- 7.5 Summary
- 7.6 Keywords
- 7.7 Self-Assessment Questions
- 7.8 Review Questions
- 7.9 Further Reading

Objectives

After studying this unit, you will be able to understand the following:

- Discuss the memory structure and its requirements
- Explain the basic concepts in memory interfacing
- Describe the 8085 MPU
- Discuss the microprocessor based controllers

Notes

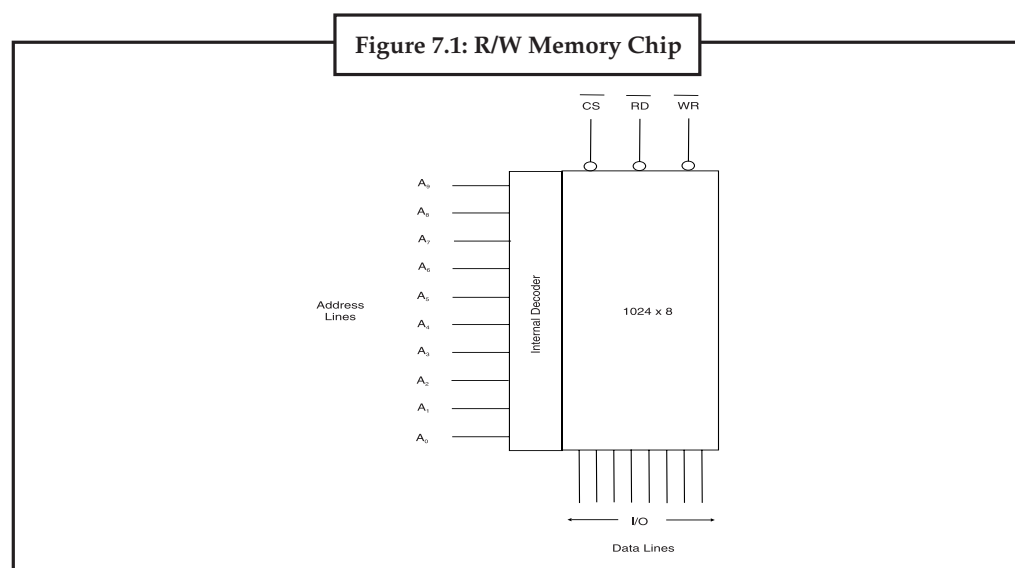
Introduction

While executing a program, the microprocessor needs to access memory frequently to read instruction codes and data stored in memory and the interfacing circuit enables that access. Memory is basically a data storage device. For any microprocessor system we require memories to store monitor program, to store data. So memory is an essential component in microprocessor system which will allow the user to store the program and data. Memory consists of thousand of memory cells is capable of storing 1-bit.

7.1 Memory Structure and its Requirements

Read/Write Memory (R/WM) is a group of registers to store binary information. Figure 7.1 shows a typical R/W memory chip, it has 1024 registers, each of which can store 8 bits indicated by 8 I/O lines.

1. An address should be placed on the address lines. The low-order address lines are decoded by the internal decoder of the memory chip, and the addressed register is identified.
2. The high-order address should be decoded to generate a Chip Select signal, and the memory chip is selected by asserting the Chip Select \overline{CS} low.
3. To read from the addressed register, \overline{RD} should be asserted low to enable the output buffer, and then the data byte from the register will be placed on the I/O lines.
4. To write into the addressed register, \overline{WR} should be asserted low to enable the input buffer, and then data bits from the data lines are stored into the register.



To read from memory, the Z80 performs the following steps:

1. Places a 16 bit address on its address bus.
2. Asserts to \overline{MREQ} indicate that the address bus holds a valid address.
3. Asserts the \overline{RD} signal low to indicate that it wants to read.

To write into memory, the Z80 performs the following steps:

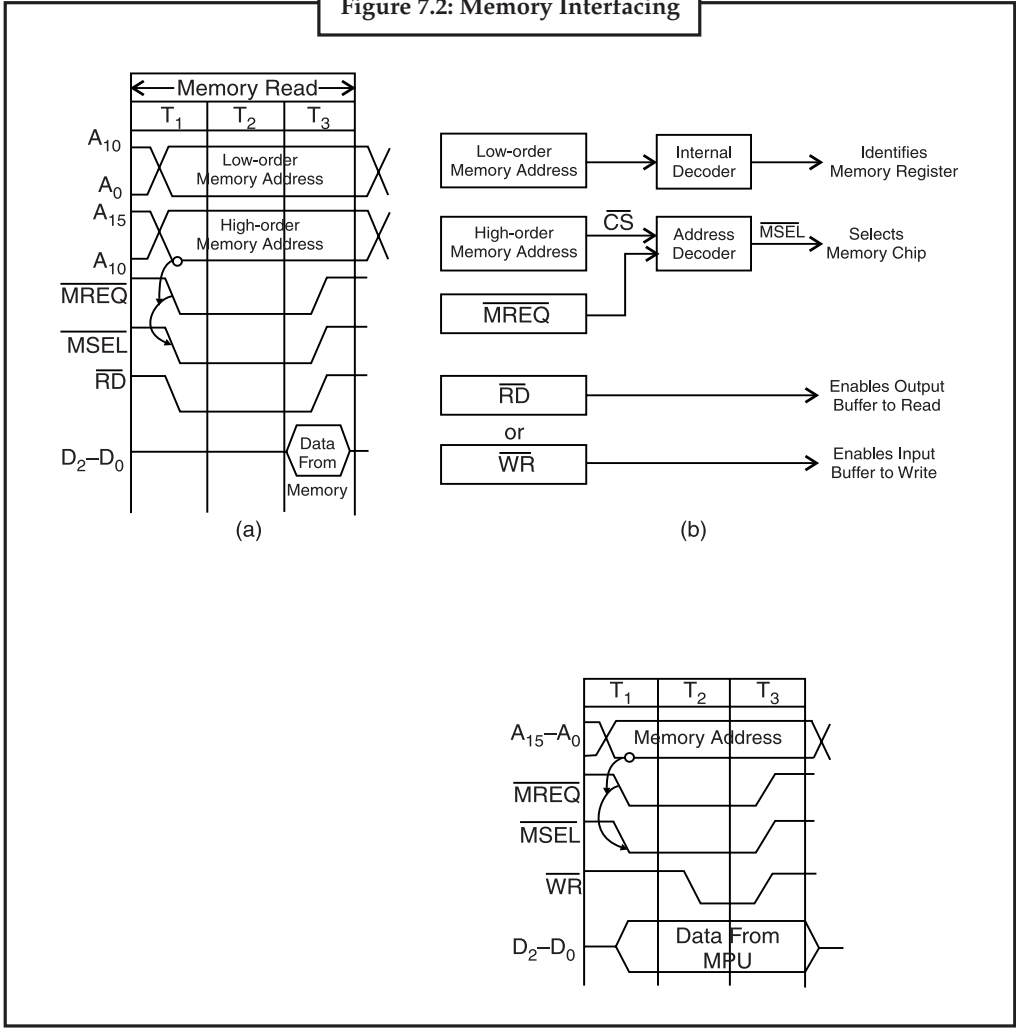
1. Places a 16 bit address on the address bus.
2. Asserts \overline{MREQ} and places data on the data bus.
3. Asserts \overline{WR} signal.

7.2 Basic Concepts in Memory Interfacing

The primary function of memory interfacing is to allow the microprocessor to read from and write into a given register of memory as shown in Figure 7.2.

- 1. Be able to select the chip
- 2. Identify the register
- 3. Enable the appropriate buffer.

Figure 7.2: Memory Interfacing

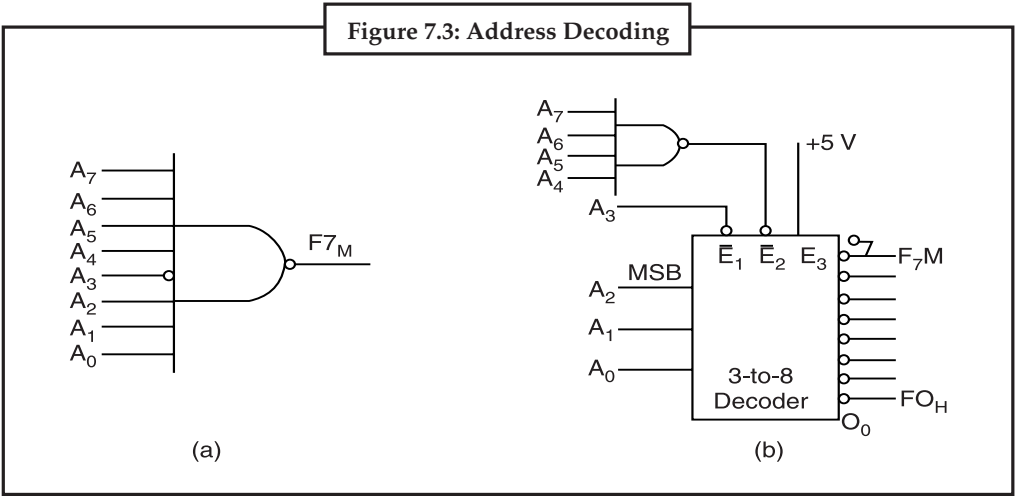


RAM is temporary memory so save important data in secondary memory.

7.2.1 Address Decoding

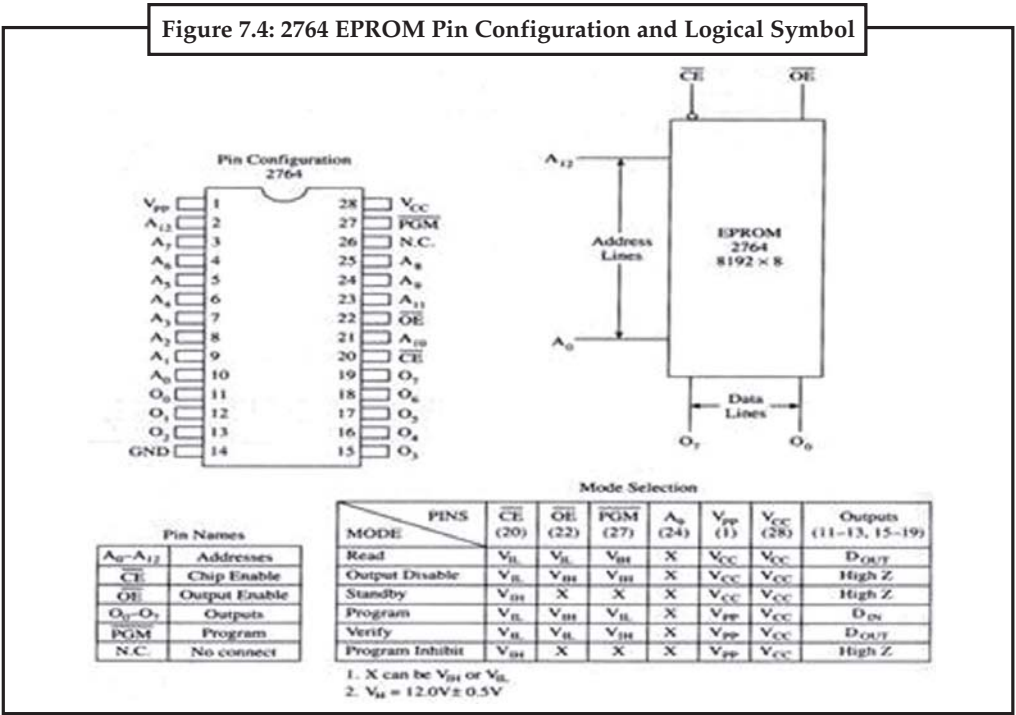
The process of address decoding should result in identifying a register with a given address; we should be able to generate a unique pulse for that address as in Figure 7.3.

Notes



7.2.2 Interfacing the 2764 EPROM

This is a memory chip commonly used in industry to develop microprocessor-based products. This is an 8k (8192 × 8) memory chip with 8 data lines and is housed in a 28-pin package as in Figure 7.4.

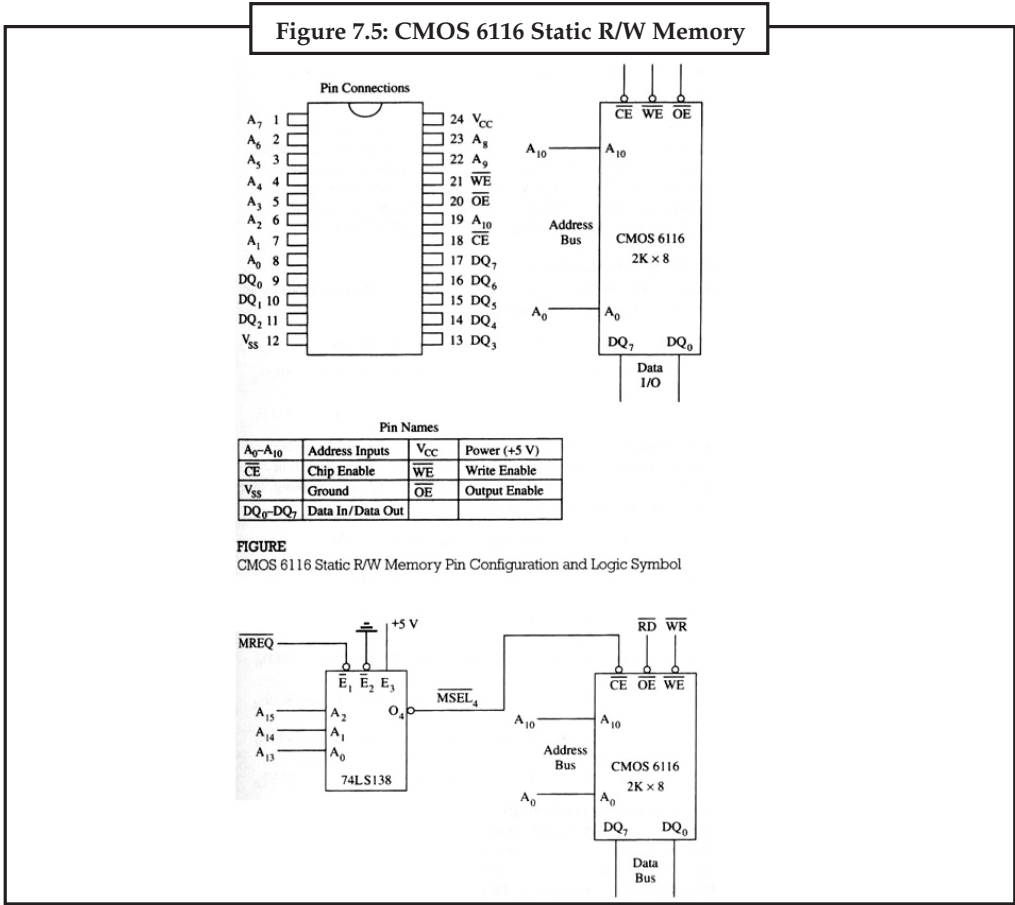


Address Range and Memory Map															
A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0 = 0000H
MSEL 0			↓	↓											↓
0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1 = FFFFH

7.2.3 Interfacing CMOS 6116 Static R/W Memory

Notes

This is a 2k static memory chip, organized as 2048 ´ 8 format. It has 11 address lines (A10-A0), 8 data lines and 3 control signals: \overline{CE} , \overline{OE} and \overline{WE} (write enable) as in Figure 7.5.



7.2.4 Memory Address Range

Assuming the “don’t care” address lines A12 and A11 are at logic 0.

A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0	
1	0	0	X	X	0	0	0	0	0	0	0	0	0	0	0	= 8000H
MSEL 4			↓												↓	
1	0	0	X	X	1	1	1	1	1	1	1	1	1	1	1	= 87FFH

7.2.5 Designing Memory for the MCTS Project

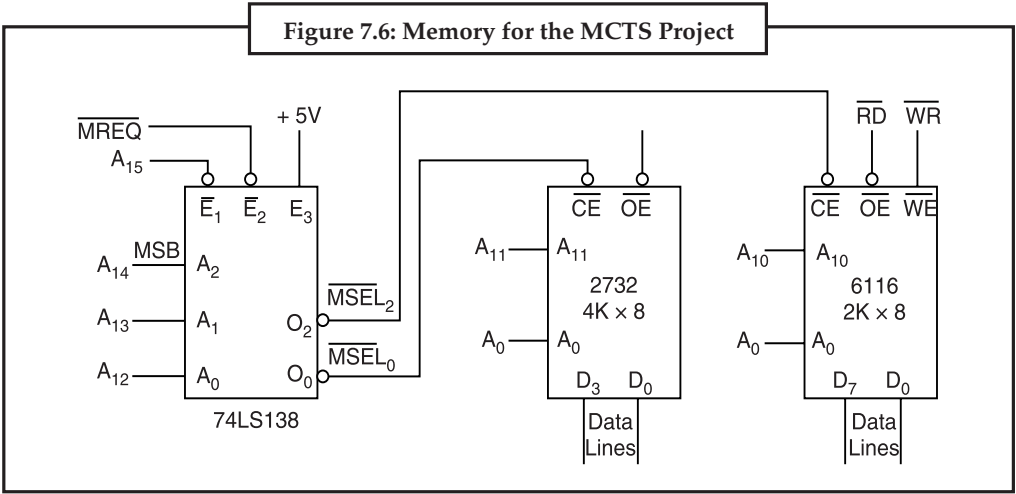
MCTS is Microprocessor-Controlled Temperature System as in Figure 7.6.

Memory Design: Problem Statement

Given the components as listed;

- 74LS 138 : 3 to 8 decoder
- 2732 (4k × 8) : EPROM
- 6116 (2k × 8) : CMOS R/W memory

Notes



Circuit Analysis

1. When the logic levels of A₁₅-A₁₂ are all 0, and the processor asserts $\overline{\text{MRE}}$ to read from memory, the output O₀ goes active and selects the chip. The address range of the EPROM is as follows:

A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0 = 0000H
$\overline{\text{MSEL}}_0$				↓											↓
				1	1	1	1	1	1	1	1	1	1	1	1 = 0FFFH

2. The 6116 R/W memory is selected by the output signal of the decoder O₂

A15	A14	A13	A12	A11	A10	A9	A8	A7	A6	A5	A4	A3	A2	A1	A0
-----	-----	-----	-----	-----	-----	----	----	----	----	----	----	----	----	----	----

7.3 The 8085 MPU

The Microprocessor is a programmable logic device; designed with registers, flip flops, and timing elements. The 8085 (8085A) is an 8 bit general purpose microprocessor capable of addressing 64K of memory. It has 40 pins, requires a +5V single power supply, and can operate with a 3 MHz single phase clock.

The 8085 is a complex IC of sequential circuits. The sequential circuits are designed to do some operation depending on what is the input on their lines. The vital inputs on the lines are what therefore determine what operation will be done by the sequential circuits inside it. Since we can find a way to put different values of inputs to the input lines of the processor at different times, we can make it execute different operations in a sequence that we desire. Thus, in other words we can make the processor execute a programme to do useful things for us. These inputs then could alternately, be called instructions. The inputs that we talked of so far are actually voltages to represent 1s and 0s. They can only be 1s or 0s, they are called digital values. The number of input lines that the processor provides are some estimation of the largeness of the instruction set that it can support.

The 8085 has 8 bit input (actually called the input lines of the data bus), meaning that we could have only a maximum of $2^8 = 256$ different combination of 1s and 0s as inputs to the processor. Well! The 8085 supports almost that many different instructions. The programme that we desire

part of the 8085 processor. The memory chips are again sequential circuits consisting of flip flops which are capable of storing digital values. Since we would be interested in storing a huge number of such digital values, a large number of these memories are packed together with a scheme of addresses, so that we can address them individually. Generally the memories are arranged in large numbers of 8 bit bunches each.

The addressing is also done by the voltages (1s or 0s) on the address bus. The 8085 has a address bus which is 16 bit wide. Therefore it can put 2^{16} different digital values on it, and therefore it can address a maximum of 2^{16} different address locations. This is called the addressing space and it is 64 kilobyte for the 8085, because $2^{16} = 65535$. And then we ask the processor to execute those instructions from a particular memory location onwards. It goes on executing those instructions one after another. And that is all it does. The memory location from which it is to pick up the next instruction for execution is maintained in an internal memory location (Register) called Programme Counter (PC). As would be expected, it has to hold a 16 bit memory address. But we will always need to start from somewhere. As soon as the power is turned on, the 8085 does a reset of its programme counter. It is reset to 0000H, on start up. After which it floats that value on the address bus. The address bus is connected in a parallel fashion to the entire memory. When the voltages on the address bus are indicating at 0000H, only that memory location is activated. And whatever is the content of that memory location is now floated on the data bus. The data bus feeds that value back to the processor 8085. This is also called an opcode fetch cycle. (an instruction fetch cycle). The 8085 executes that instruction and waits for the next instruction. Apart from executing instructions from consecutive memory locations, the processor can make changes in the value of the program counter itself so that it will fetch the next instruction from some other memory location. This is how jumps are executed. Your programme could also write to the memory locations whose address you specify. The processor uses certain internal memory locations called Registers in doing all the operations that we ask it to do. The contents of those memory locations can be directly altered by the instructions that we give.



The operations can be very complex and therefore this chip is also called a processor.

7.4 Microprocessor Based Controllers

7.4.1 Single-board Microcomputers (Microcontrollers)

A microprocessor by itself is not a computer to be functional; the microprocessor must be connected to other integrated circuits that provide the memory and I/O capability. A microcontroller is a computer on a single IC, designed specifically for control applications. It consists of a microprocessor, memory (both RAM and ROM), I/O ports, and possibly other features such as timers and ADCS/DACS. Having the complete controller on a single chip allows the hardware design to be simple and very inexpensive. Microcontrollers are showing up increasingly in products as varied as industrial applications, home appliances, and toys. In such uses as these, they are called embedded controllers because the controller is located physically in the equipment being controlled.

The main difference between microprocessors and microcontrollers is that microprocessors are being designed for use in microcomputers where greater speed and larger word size are the driving requirements, whereas microcontrollers are evolving toward reduced chip count by integrating more hardware functions on the chip. Most control applications do not need the 32 bit word size and 500-MHz (megahertz) speed of the newer microprocessors. Eight or 16 bits and 1 MHz will work just fine in many applications, and the single-chip microcontroller costs much less.

Another difference between microprocessors and microcontrollers concerns the instruction set. The microprocessor tends to be rich in instructions dealing with moving data into and out of

Notes

memory. The microcontroller has fewer memory-move instructions and more bit-handling instructions. The reason for the lack of memory-move instructions is that the microcontroller typically has only a small amount of RAM, which it uses only as a “scratch pad.” The additional bit-handling instructions were included because they are so useful in control system applications. For example, in a control system, each separate bit of a parallel output word might control a different device, such as a motor or indicator light. The bit-handling instructions allow the software to turn one device easily on or off without affecting the others.

In March 1976, Intel announced a single-board computer product that integrated all the support components required for their 8080 microprocessor, along with 1 kbytes of RAM, 4 kbytes of user-programmable ROM, and 48 lines of parallel digital I/O with line drivers. The board also offered expansion through a bus connector, but could be used without an expansion card cage where applications didn’t require additional hardware. Software development for this system was hosted on Intel’s Intellect MDS microcomputer development system; this provided assembler and PL/M support, and permitted in-circuit emulation for debugging.

Processors of this era required a number of support chips in addition. RAM and EPROM were separate, often requiring memory management or refresh circuitry for dynamic memory as well. I/O processing might be carried out by a single chip such as the 8255, but frequently required several more chips.

A single-board microcontroller differs from a single-board computer in that it lacks the general purpose user interface and mass storage interfaces that a more general-purpose computer would have. Compared to a microprocessor development board, a microcontroller board would emphasize digital and analog control interconnections to some controlled system, where a development board might by comparison have only a few or no discrete or analog input/output devices. The development board exists to showcase or to train on some particular processor family and this internal implementation is more important than the external function.



Did u know?

Single-board microcontrollers appeared in the late 1970s when the first generations of microprocessors, such as the 6502 and the Z80, made it practical to build an entire controller on a single board, and affordable to dedicate a processor chip to such a relatively minor task.

7.4.2 Input and Output

Microcontroller systems provide multiple forms of input and output signals to allow application software to control an external “real-world” system. Discrete digital I/O provides a single bit of data (on, or off). Analog signals, representing a continuously variable range such as temperature or pressure, can also be inputs and outputs for microcontrollers.

Discrete digital inputs and outputs might only be buffered from the microprocessor data bus by an addressable latch, or might be operated by a specialized input/output integrated circuit such as an Intel 8155 or Motorola 6821 parallel input/output adapter. Later single-chip microcontrollers have input and output pins available. The input/output circuits usually do not provide enough current to directly operate such devices as lamps or motors, so solid-state relays are operated by the microcontroller digital outputs, and inputs are isolated by signal conditioning level-shifting and protection circuits.

One or more analog inputs, with an analog multiplexer and common analog to digital converter, are found on some microcontroller boards. Analog outputs may use a digital-to-analog converter, or on some microcontrollers may be controlled by pulse-width modulation. As for discrete inputs, external signal conditioning may be required to scale inputs, or provide such functions as bridge excitation or cold-junction compensation.

To control component costs, many boards were designed with extra hardware interface circuits but the components for these circuits weren't installed and the board was left bare. The circuit was only added as an option on delivery, or could be populated later.

It is common practice for boards to include "prototyping areas", areas of the board already laid out as a solderable breadboard area with the bus and power rails available, but without a defined circuit. Several controllers, particularly those intended for training, also included a pluggable re-usable breadboard for easy prototyping of extra I/O circuits that could be changed or removed for later projects.

7.4.3 Bus

Some microcontroller boards using a general-purpose microprocessor can bring the address and data bus of the processor to an expansion connector, allowing additional memory or peripherals to be added. This would provide resources not already present on the single board system. Since not all systems require expansion, the connector may be an option, with a mounting position provided for the connector for installation by the user if desired.

7.4.4 Communications and User Interfaces

Communications interfaces vary depending on the age of the microcontroller system. Early systems might implement a serial port to provide RS_232 or current loop. The serial port could be used by the application program, or could be used, in conjunction with a monitor ROM, to transfer programs into the microcontroller memory. Current microcontrollers may support USB, wireless network ports, or provide an Ethernet connection. Some devices have firmware available to implement a Web server, allowing an application developer to rapidly build a Web-enable instrument or system.

7.4.5 Programming

Many of the earliest systems had no internal facility for programming at all, and relied on a separate "host" system. This programming was typically in assembly language, sometimes C or even PL/M, and then cross-assembled or cross-compiled on the host.

7.4.6 EPROM Burning

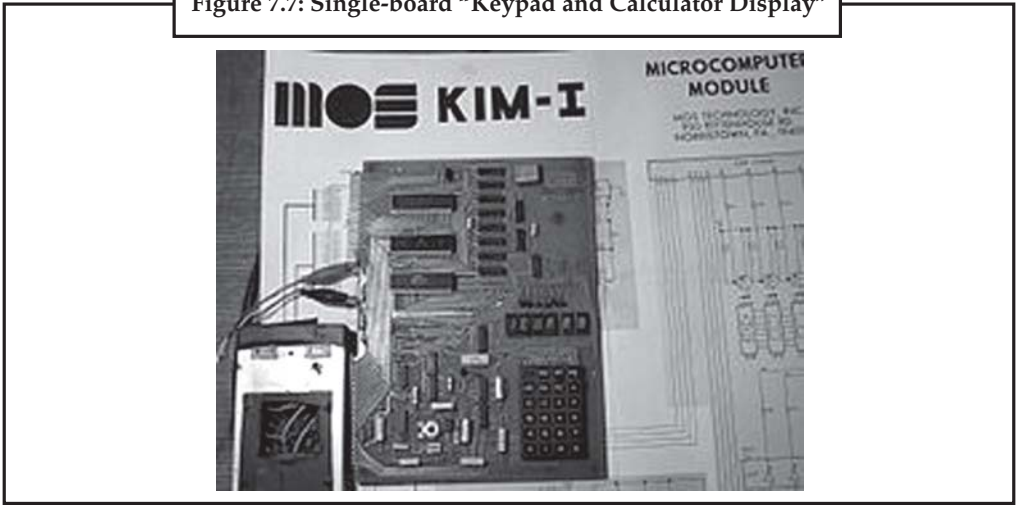
The completed object code from the host system (usually in Intel HEX format) would then be "burned" onto an EPROM with an EPROM programmer, this EPROM then being physically plugged into the board. As the EPROM would be removed and replaced many times during program development, it was usual to provide a ZIF socket to avoid wear or damage. As "washing" an EPROM with a UV eraser takes a considerable time, it was also usual for a developer to have several EPROMs in circulation at any one time.

7.4.7 Keypad Monitors


Where the single-board controller formed the entire development environment (typically in education) the board might also be provided with a simple hexadecimal keypad, calculator-style LED display and a "monitor" program set permanently in ROM. This monitor allowed machine code programs to be entered directly through the keyboard and held in RAM. These programs were in machine code, not even in assembly language, and were assembled by hand on paper first. It's arguable as to which process was more time-consuming and error prone: assembling by hand, or keying byte-by-byte.

Notes

Figure 7.7: Single-board “Keypad and Calculator Display”



Single-board “keypad and calculator display” microcontrollers of this type were very similar to some low-end microcomputers of the time, such as the KIM-1 or the Microprocessor I. Some of these microprocessor “trainer” systems are still in production today, as a very low-cost introduction to microprocessors at the hardware programming level.



Task Explain the Discrete digital I/O with the related example.

7.4.8 Single-chip Microcontrollers

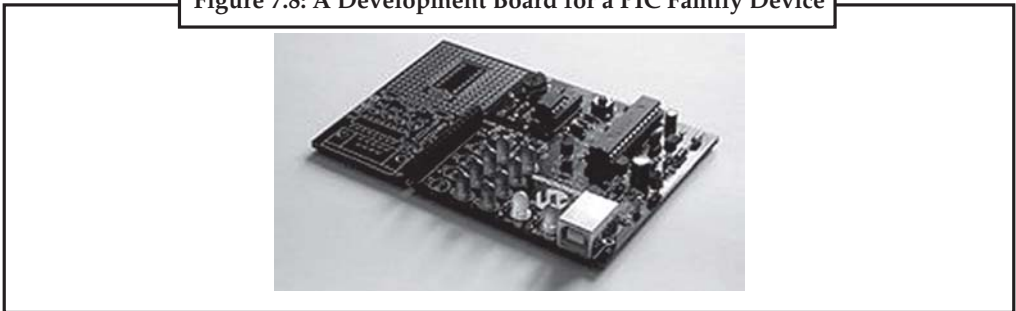
With the development of single-chip microcontrollers such as the 8748, it became possible to combine most of the features of the previous board into a single IC package. Often only the clock generator remained separate, as it remained easier to use a quartz crystal oscillator than to integrate a stable clock circuit into a low-cost IC.

Single-chip microcontrollers integrate their necessary memory (both RAM and ROM) on-package and so do not need to expose their bus through the IC package’s pins. These pins are thus freed-up for I/O lines. Both of these changes make the design of a single-board microcontroller simpler, but also less necessary. Single-board development and training systems remained popular, but there was now less need to provide single-board systems as embeddable components for production systems.

7.4.9 Program Memory

For production use as embedded systems, the on-board ROM would be either mask programmed at the chip factory or one-time programmed (OTP) by the developer as a PROM. PROMs often used the same UV EPROM technology for the chip, but in a cheaper package without the transparent erasure window. During program development it was still necessary to burn EPROMs, this time the entire controller IC, and so ZIF sockets would be provided.

Figure 7.8: A Development Board for a PIC Family Device



Notes

With the development of affordable EEPROM, EAROM and eventually flash memory, it became practical to attach the controller permanently to the board and to download program code to it through a serial connection to a host computer. This was termed “in-circuit programming”. Erasure of old programs was carried out by either over-writing them with a new download, or bulk erasing them electrically (for EEPROM) which was slower, but could be carried out in-situ.

The main function of the controller board was now to carry the support circuits for this serial interface, or USB on later boards. As a further convenience feature during development, many boards also carried low-cost features like LED monitors of the I/O lines or reset switches mounted on-board.

- 8748
- PIC
- Atmel AVR

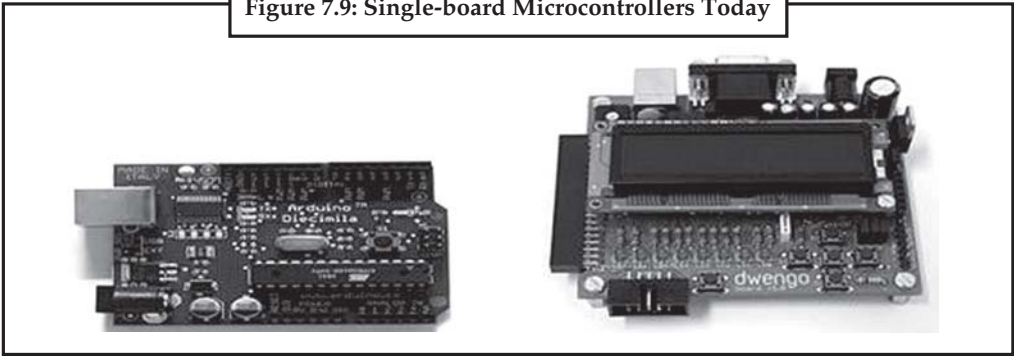
7.4.10 Single-board Microcontrollers Today

Microcontrollers are now cheap and simple to design circuit boards for. Development host systems are also cheap, especially when using open source software. Higher level programming languages abstract details of the hardware, making differences between specific processors less obvious to the application programmer. Rewritable flash memory has replaced slow programming cycles, at least during program development. Accordingly almost all development now is based on cross-compilation from personal computers and downloads to the controller board through a serial-like interface, usually appearing to the host as a USB device.

The original market demand of a simplified board implementation is no longer so relevant to microcontrollers. Single-board microcontrollers are still important, but have shifted their focus to:

- Easily accessible platforms aimed at traditionally “non-programmer” groups, such as artists. These controllers may be embedded to form part of a physical computing project. Popular choices for this work are the Arduino, Dwengo or the Wiring project.
- Technology demonstrator boards for innovative processors or peripheral features:
 - AVR Butterfly
 - Parallax Propeller

Figure 7.9: Single-board Microcontrollers Today



Case Study

Flash Memory Analyzer

The integrity of a Flash memory device degrades over time primarily due to the impact of erase cycles. Wear-levelling techniques are internally employed by the device to slow its degradation and prolong its life span. Our product, the flash memory analyser, allows a variety of flash memory devices to be subjected to stress tests and their aging to be

Contd...

Notes

analysed in detail (in the spirit of investigating techniques to improve life span). The analyser is controlled by a user friendly application on a network connected PC. Multiple analysers can be driven simultaneously by a single PC.	
Embedded Device (Flash Analyser)	PC/Server (Analyser Interface)
Single board computer (embedded ARM9)	Microsoft Windows XP & Vista
Custom daughter board for Flash devices	Graphical User Interface
Device driver development	Virtual Displays appear on Extended Desktop
Device Driver Adaptation	Remote management of devices via TCP/IP
Data acquisition and reporting	Memory read/write test case automation
Remote management via TCP/IP	Microsoft C, C++, .NET
Questions:	
1. Explain the working of Flash Memory Analyzer.	
2. Differentiate between embedded device and PC/server analyze interface.	

7.5 Summary

- Memory is an essential component in microprocessor system which will allow the user to store the program and data. Memory consists of thousand of memory cells is capable of storing 1-bit.
- To write into the addressed register, should be asserted low to enable the input buffer, and then data bits from the data lines are stored into the register.
- The primary function of memory interfacing is to allow the microprocessor to read from and write into a given register of memory chip.
- The 8085 has a 8 bit input (actually called the input lines of the data bus), meaning that we could have only a maximum of $2^8=256$ different combination of 1s and 0s as inputs to the processor.
- A single-board microcontroller differs from a single-board computer in that it lacks the general purpose user interface and mass storage interfaces that a more general-purpose computer would have.

7.6 Keywords

- *Address decoding*: address decoding is used the result in identifying a register with a given address.
- *MPU*: The Microprocessor is a programmable logic device; designed with registers, flip flops, and timing elements.
- *OTP*: The on-board ROM would be either mask programmed at the chip factory or one-time programmed (OTP) by the developer as a PROM.
- *R/W*: Read/Write Memory is a group of registers to store binary information.



Lab Exercise

1. Explain the memory structure in the single board microprocessor.
2. Give the circuit analysis of the 8085 microprocessor IC.

7.7 Self-Assessment Questions

Notes

- The should be decoded to generate a Chip Select signal.
 - high-order address
 - low-order address
 - high order signal
 - low-order signal
- instructions do not affect any of the status flags.
 - MOV
 - Carry
 - Data copy
 - None of these

7.8 Review Questions

- Describe the memory structure and requirements.
- Discuss memory interfacing and how it play important role in microprocessor.
- Describe EPROM interfacing in microprocessor.
- Explain interfacing CMOS 6116 static R/W memory.
- Discuss in detail 8085 MPU.
- Describe the microcontrollers.
- How single based microcontroller useful today?
- What are Single-chip microcontrollers?
- Define EPROM burning.
- How it is useful in single chip microcontroller?

Answers for Self-Assessment Questions

- (a)
- (c)

7.9 Further Reading



Books

Microprocessor 8085, 8086, By Abhishek Yadav

Online link

<http://books.google.com/books?>

Unit 8: Interfacing I/O Devices

CONTENTS

Objectives

Introduction

8.1 Basic Interface Concepts

8.2 Graphical User Interface

8.2.1 Physical Devices

8.2.2 Virtual Devices

8.2.3 Polling and Sampling

8.2.4 Device Association

8.2.5 Device Input Modes

8.2.6 Application Structure

8.2.7 Event Queues

8.2.8 Toolkits and Callbacks

8.3 I/O Devices in Microprocessor

8.3.1 Dealing with I/O Devices

8.3.2 Peripheral I/O Instructions

8.3.3 The Execution of the OUT Instruction

8.3.4 The Execution of the IN Instruction

8.3.5 The Interfacing of Output Devices

8.3.6 Interfacing of Input Devices

8.3.7 Interfacing the LEDs

8.3.8 Interfacing the LEDs (Control Circuit)

8.3.9 Interfacing the LEDs (Latch & LEDs)

8.3.10 Interfacing the LEDs (The Program)

8.3.11 Interfacing the Switches

8.3.12 Interfacing the Switches (Control Circuit)

8.3.13 Interfacing the Switches (Latch & Switches)

8.3.14 Interfacing the Switches (The Program)

8.4 Memory Mapped I/O

8.4.1 Memory Barriers

8.4.2 Port Mapped vs. Memory Mapped I/O

8.5 Summary

8.6 Keywords

8.7 Self-Assessment Questions

8.8 Review Questions

8.9 Further Reading

Objectives

Notes

After studying this unit, you will be able to understand the following:

- Describe Basic interface concepts
- Explain graphical user interface
- Discuss I/O devices in microprocessor
- Explain the memory mapped I/O

Introduction

An I/O interface is required whenever the I/O device is driven by the processor. The interface must have necessary logic to interpret the device address generated by the processor. Handshaking should be implemented by the interface using appropriate commands (like BUSY, READY, and WAIT), and the processor can communicate with an I/O device through the interface. If different data formats are being exchanged, the interface must be able to convert serial data to parallel form and vice-versa. There must be provision for generating interrupts and the corresponding type numbers for further processing by the processor if required.

A computer that uses memory-mapped I/O accesses hardware by reading and writing to specific memory locations, using the same assembly language instructions that computer would normally use to access memory.

8.1 Basic Interface Concepts

Interfacing is a textile used on the unseen or “wrong” side of fabrics to make an area of a garment more rigid.

Interfacings can be used to:

- stiffen or add body to fabric, such as the interfacing used in shirt collars
- strengthen a certain area of the fabric, for instance where buttonholes will be sewn
- keep fabrics from stretching out of shape, particularly knit fabrics

Interfacings come in a variety of weights and stiffnesses to suit different purposes. Generally, the heavier weight a fabric is the heavier weight an interfacing it will use. Most modern interfacings have heat-activated adhesive on one side. They are affixed to a garment piece using heat and moderate pressure, from a hand iron for example. This type of interfacing is known as “fusible” interfacing. Non-fusible interfacings do not have adhesive and must be sewn by hand or machine.

There are a lot of places where you’ll see text that looks like this. Apart from this example, that color and style of text indicates a tip. Just hover your mouse over the text and you’ll get a hovering window that gives more information. Generally, the game’s most crucial concepts will be explained here, so make good use of these elements!

Until you turn them off, most pages will come standard with page tips that tell you a bit about what the page are all about. Sometimes these tips are very useful! Try to keep the tips on until you’re truly familiar with the inner workings of the game

8.2 Graphical User Interface

It is a type of user interface that allows users to interact with electronic devices with images rather than text commands. GUIs can be used in computers, hand-held devices such as MP3 players, portable media players or gaming devices, household appliances and office equipment. A GUI represents the information and actions available to a user through graphical icons and

Notes

visual indicators such as secondary notation, as opposed to text-based interfaces, typed command labels or text navigation. The actions are usually performed through direct manipulation of the graphical elements

A short outline of input devices and the implementation of a graphical user interface is given:

- Physical input devices used in graphics
- Virtual devices
- Polling is compared to event processing
- UI toolkits are introduced by generalizing event processing

8.2.1 Physical Devices

Actual, physical input devices include:

- Dials (Potentiometers)
- Selectors
- Pushbuttons
- Switches
- Keyboards (collections of pushbuttons called “keys”)
- Trackballs (relative motion)
- Mice (relative motion)
- Joysticks (relative motion, direction)
- Tablets (absolute position) etc.

8.2.2 Virtual Devices

Devices can be classified according to the kind of value they return:

Button: Return a Boolean value; can be *depressed* or *released*.

Key: Return a “character”; that is, one of a given set of code values.

String: Return a sequence of characters.

Selector: Return an integral value (in a given range).

Choice: Return an option (menu, callback).

Valuator: Return a real value (in a given range).

Locator: Return a position in (2D/3D) space (e.g. several ganged valuator).

Stroke: Return a sequence of positions.

Pick: Return a scene component.

Each of the above is called a virtual device.

8.2.3 Polling and Sampling

In polling,

- Value of input device constantly checked in a tight loop
- Wait for a change in status

Generally, polling is inefficient and should be avoided, particularly in time-sharing systems.

Notes

In sampling, value of an input device is read and then the program proceeds.

- No tight loop
- Typically used to track sequence of actions (the mouse)

8.2.4 Device Association

To obtain device independence:

- Design an application in terms of virtual (abstract) devices.
- Implement virtual devices using physical devices

There are certain natural associations:

e.g. Valuator—Mouse-X

But if the naturally associated device does not exist on a platform, one can make do with other possibilities:

e.g. Valuator—number entered on keyboard

8.2.5 Device Input Modes

Return values from devices may be provided in:

Request Mode:

Alternating application and device execution

- application requests a return and then suspends execution;
- device wakes up, provides input and then suspends execution;
- application resumes execution.

Sample Mode:

Concurrent application and device execution

- device continually updates register(s) or memory location(s);
- application may read at any time.

Event Mode:

Concurrent application and device execution together with a concurrent queue management service

- device continually offers input to the queue
- application may request selections and services from the queue (or the queue may interrupt the application).

8.2.6 Application Structure

With respect to device input modes, applications may be structured to engage in

- requesting
- polling or sampling
- event processing

Notes

Events may or may not be interruptive. If not interruptive, they may be read in a

- blocking
- non-blocking

8.2.7 Event Queues

- Device is monitored by an asynchronous process.
- Upon change in status of device, this process places a record into an event queue.
- Application can request read-out of queue:
 - Number of events
 - 1st waiting event
 - Highest priority event
 - 1st event of some category
 - All events
- Application can also
 - Specify which events should be placed in queue
 - Clear and reset the queue etc.
- Queue reading may be blocking or non-blocking
- Processing may be through callbacks
- Events may be processed interactively
- Events can be associated with more than devices

Without interrupts, the application will engage in an *event loop*

- not a tight loop
- a preliminary of register *event* actions followed by a repetition of test for event actions.

For more sophisticated queue management,

- application merely registers event-process pairs
- queue manager does all the rest

if event XXX then invoke process YYY.”

- The cursor is usually bound to a pair of valuator, typically MOUSE_X and MOUSE_Y.
- Events can be restricted to particular areas of the screen, based on the cursor position.
- Events can be very general or specific:
 - A mouse button or keyboard key is depressed.
 - A mouse button or keyboard key is released.
 - The cursor enters a window.
 - The cursor has moved more than a certain amount.
 - An Expose event is triggered under X when a window becomes visible.
 - A Configure event is triggered when a window is resized.
 - A timer event may occur after a certain interval.

- Simple event queues just record a code for event (Iris GL).
- Better event queues record extra information such as time stamps (X windows).

8.2.8 Toolkits and Callbacks

Event-loop processing can be generalized:

- Instead of switch, use table lookup.
- Each table entry associates an event with a callback function.
- When event occurs, corresponding callback is invoked.
- Provide an API to make and delete table entries.
- Divide screen into parcels, and assign different callbacks to different parcels (X Windows does this).
- Event manager does most or all of the administration.

Modular UI functionality is provided through a set of widgets:

- Widgets are parcels of the screen that can respond to events.
- A widget has a graphical representation that suggests its function.
- Widgets may respond to events with a change in appearance, as well as issuing callbacks.
- Widgets are arranged in a parent/child hierarchy.
 - Event-process definition for parent may apply to child, and child may add additional event-process definitions
 - Event-process definition for parent may be redefined within child
- Widgets may have multiple parts, and in fact may be composed of other widgets in a hierarchy.

Some UI toolkits: Xm, Xt, SUIT, FORMS, Tk



Task

Distinguish between Physical and virtual devices.

8.3 I/O Devices in Microprocessor

Using I/O devices data can be transferred between the microprocessor and the outside world. This can be done in groups of 8 bits using the entire data bus. This is called parallel I/O. The other method is serial I/O where one bit is transferred at a time using the SI and SO pins on the Microprocessor.

8.3.1 Dealing with I/O Devices

There are two ways to deal with I/O devices.

1. Consider them like any other memory location.
 - They are assigned a 16-bit address within the address range of the 8085.
 - The exchange of data with these devices follows the transfer of data with memory. The user uses the same instructions used for memory.
 - This is called memory-mapped I/O.

Notes

2. Treat them separately from memory:

- I/O devices are assigned a “port number” within the 8-bit address range of 00H to FFH.
- The user in this case would access these devices using the IN and OUT instructions only.
- This is called I/O-mapped I/O or Peripheral-mapped I/O.

The first step in interfacing an I/O device would be to determine which instructions will be used to access it. If you want the user to use the IN/OUT instructions, then it should be interfaced as a peripheral-mapped I/O device. If the user should use regular data transfer instructions (LDA, STA, etc.) then it should be interfaced as a memory-mapped I/O device.

8.3.2 Peripheral I/O Instructions

There are two instructions:

- IN brings data (8-bits) from an input device to the accumulator.
- OUT brings data (8-bits) from the accumulator to an output device.

They are both 2 byte instructions with the second byte holding the 8-bit address of the device.



Did u know?

There are separate instructions for input and output, the 8085 can actually communicate with 256 different input devices AND an additional 256 different output devices.

8.3.3 The Execution of the OUT Instruction

The OUT instruction requires 3 machine cycles and 10 T-states.

- The first cycle is an opcode fetch cycle to fetch the 1st byte of the instruction from memory (OUT).
- The second cycle is a memory read cycle to bring the 8-bit port number from the next location.
- The third cycle is an I/O write cycle. In this cycle, the 8085 places the port number on AD0-AD7 AND A8-A15 and the signal WR is set low (active).

Since the device address is placed on both AD0-AD7 as well as A8-A15, there is no need for demultiplexing AD0-AD7. A8-A15 can be used directly to identify the device.

8.3.4 The Execution of the IN Instruction

The execution of the IN instruction is almost identical to that of the OUT instruction. 3 machine cycles, 10 T-states.

- The first machine cycle is the opcode fetch.
- The second cycle is the memory read to get the port number.
- The third is an I/O Read cycle.

Again, in T1 the port address (8-bits) is placed on both AD0-AD7, and A8-A15. The IO/M signal is set high to indicate an I/O operation. At the beginning of T2, the RD signal is set low (active) and the I/O device responds by placing the 8-bit data on the data bus.

8.3.5 The Interfacing of Output Devices

Output devices are usually slow. Also, the output is usually expected to continue appearing on the output device for a long period of time. Given that the data will only be present on the data lines for a very short period (microseconds), it has to be latched externally.

To do this the external latch should be enabled when the port's address is present on the address bus, the IO/M signal is set high and WR is set low. The resulting signal would be active when the output device is being accessed by the microprocessor. Decoding the address bus (for memory-mapped devices) follows the same techniques discussed in interfacing memory.

8.3.6 Interfacing of Input Devices

The basic concepts are similar to interfacing of output devices. The address lines are decoded to generate a signal that is active when the particular port is being accessed. An IORD signal is generated by combining the IO/M and the RD signals from the microprocessor.

A tri-state buffer is used to connect the input device to the data bus. The control (Enable) for these buffers is connected to the result of combining the address signal and the signal IORD.

Examples of Interfacing I/O Devices

To illustrate the techniques of interfacing I/O devices we will design the circuits needed to interface 8 LEDs to display the contents of the accumulator as well as 8 switches to set the contents of the accumulator.

8.3.7 Interfacing the LEDs

Let's first design the external circuit. The data on the data bus from the microprocessor stays for an extremely short amount of time. So, in order to keep it long enough for displaying, we will need an external latch.

We will use an 8-bit latch to hold the data we need to connect the 8 LED to the latches outputs.

However, the latch will not be able to source enough current. So, we will use the inverted outputs and make it sink the current instead.

When should the latch be enabled?

- It needs to be enabled when the data is on the data bus.
- That happens when the ALE signal is low. However, we only want to display the data that is being sent to the I/O, we don't want to display the data being saved in memory.
- So, the latch needs to be enabled only during I/O operations. That happens when IO/M=1
- Finally we only want to display data intended for our port. We must decide on a port number.

Let's say FFH.

Now, we can design the control circuit.

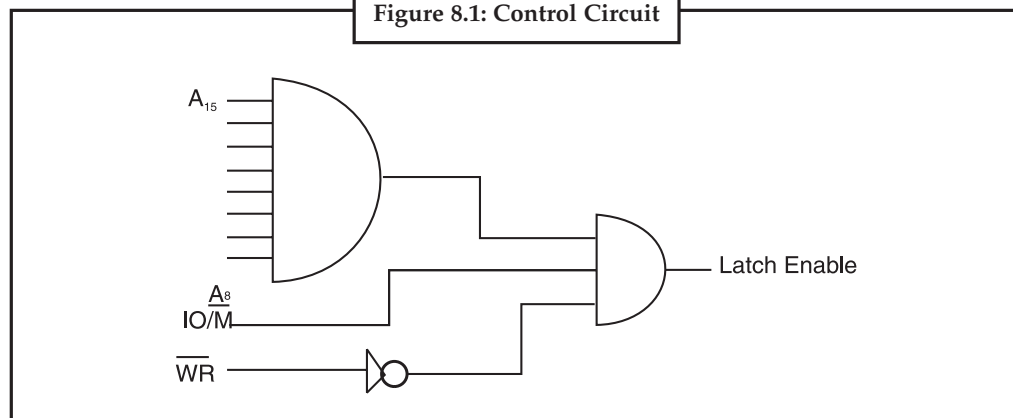
8.3.8 Interfacing the LEDs (Control Circuit)

The Latch will be enabled when:

- $WR = 0$
- $IO/M = 1$
- The address on A8 – A15 = FFH

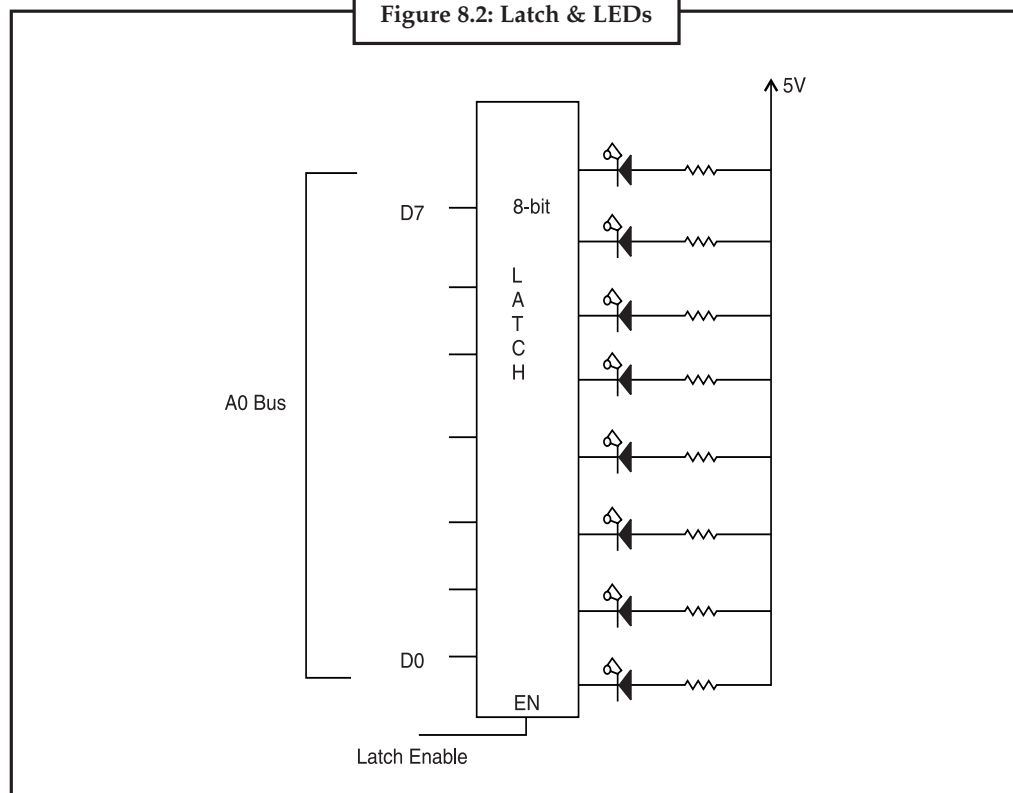
Notes

Figure 8.1: Control Circuit



8.3.9 Interfacing the LEDs (Latch & LEDs)

Figure 8.2: Latch & LEDs



8.3.10 Interfacing the LEDs (The Program)

- When a bit on the AD bus is 1, the corresponding Q' will be zero and the LED will have 5 volts on the anode and 0 on the cathode. Therefore, it will be on.
- Finally, to write the program:


```
MVI A, Data    ;load the data to be displayed
OUT FF         ;send the data to output port FF
HLT           ;End
```

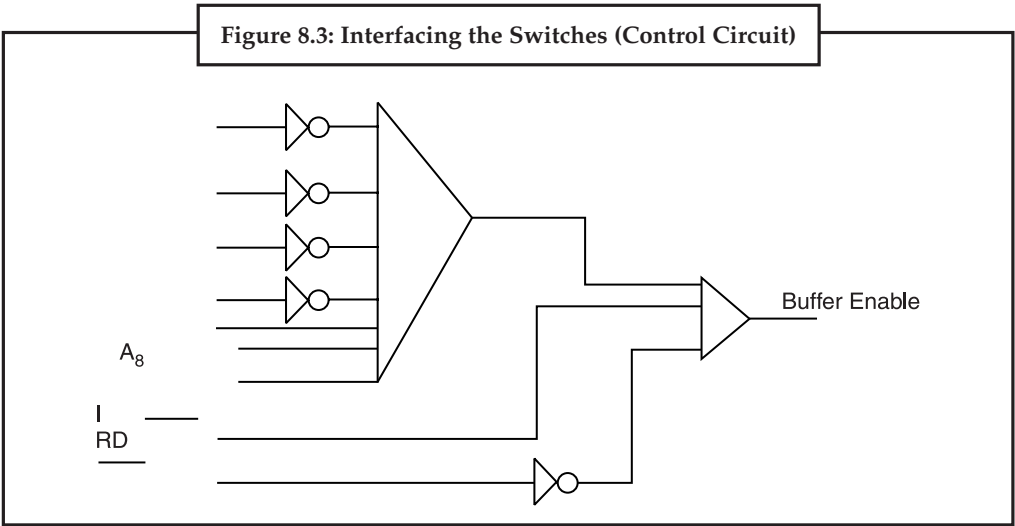
8.3.11 Interfacing the Switches

Notes

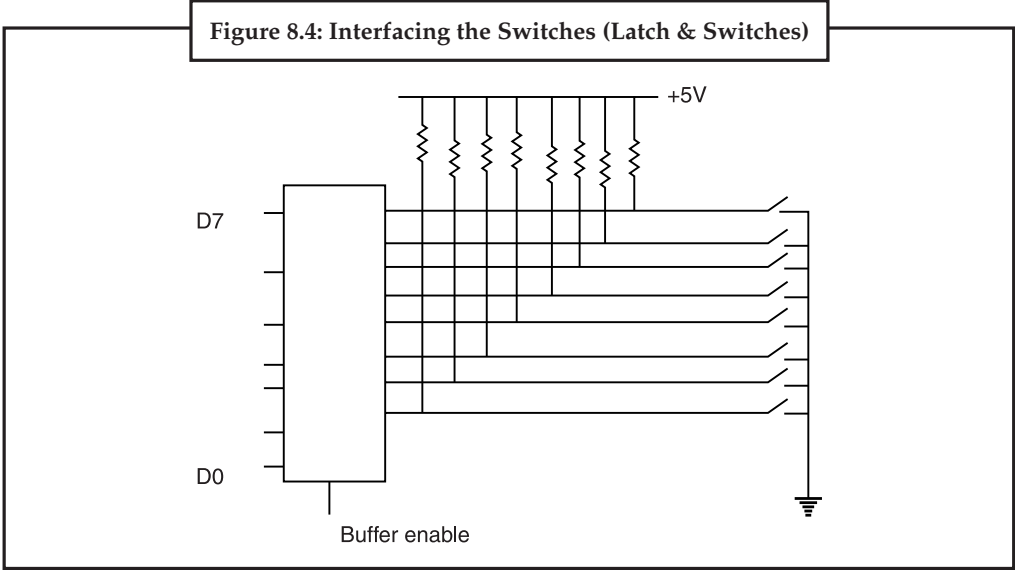
- The binary value from the switches will have to be carried by the data bus. However, the data bus is a shared bus. So, the switches must be connected to the data bus using Tri-state buffers.
- Similar to the latch, the buffers must be enabled only on I/O Read operation from this I/O port.
- Lets choose I/O port 0FH for the switches. So, the buffers must be enabled when:
 - RD = 0
 - IO/M = 1

A8-A15 = 0FH

8.3.12 Interfacing the Switches (Control Circuit)



8.3.13 Interfacing the Switches (Latch & Switches)



Notes

8.3.14 Interfacing the Switches (The Program)

- Finally, the program:

```
IN 0FH          ;input data from port 0F into A
HLT            ;END
```
- If we combine both circuits, then we can write the following program:

```
INPUT:         IN 0FH
               OUT FFH
               JMP INPUT
```

8.4 Memory Mapped I/O

A peripheral device that assigns specific memory location to inputs and outputs. For example, in a memory mapped display, each pixel or text character derives its data from a specific memory byte or bytes. The instant this memory is updated by software, the screen is displaying the new data.

Memory-mapped I/O (MMIO) and port I/O (also called isolated I/O or port-mapped I/O abbreviated PMIO) are two complementary methods of performing input/output between the CPU and peripheral devices in a computer. An alternative approach, not discussed here, is using dedicated I/O processors commonly known as channels on mainframe computers that execute their own instructions.

Memory-mapped I/O (not to be confused with memory-mapped file I/O) uses the same address bus to address both memory and I/O devices - the memory and registers of the I/O devices are mapped to (associated with) address values. So when an address is used by the CPU it may refer to a portion of physical RAM, or it can instead refer to memory of the I/O device. Thus, the CPU instructions used to access the memory are also used for accessing devices. Each I/O device monitors the CPU's address bus and responds to any of the CPU's access of address space assigned to that device, connecting the data bus to a desirable device's hardware register. To accommodate the I/O devices, areas of the addresses used by the CPU must be reserved for I/O and not be available for normal physical memory. The reservation might be temporary the Commodore 64 could bank switch between its I/O devices and regular memory or permanent.

Port-mapped I/O uses a special class of CPU instructions specifically for performing I/O. This is generally found on Intel microprocessors, specifically the IN and OUT instructions which can read and write one to four bytes (outb, outw, outl) to an I/O device. I/O devices have a separate address space from general memory, either accomplished by an extra "I/O" pin on the CPU's physical interface, or an entire bus dedicated to I/O. Because the address space for I/O is isolated from that for main memory, this is sometimes referred to as isolated I/O.

A device's direct memory access (DMA) is not affected by those CPU-to-device communication methods, especially it is not affected by memory mapping. This is because by definition, DMA is a memory-to-device communication method that bypasses the CPU.

Hardware interrupt is yet another communication method between CPU and peripheral devices. However, it is always treated separately for a number of reasons. It is device-initiated, as opposed to the methods mentioned above, which are CPU-initiated. It is also unidirectional, as information flows only from device to CPU. Lastly, each interrupt line carries only one bit of information with a fixed meaning, namely "an event that requires attention has occurred in a device on this interrupt line".

8.4.1 Memory Barriers

Memory-mapped I/O is the cause of memory barriers in older generations of computers the 640 KiB barrier is due to the IBM PC placing the Upper Memory Area in the 640–1024 KiB range (of its 20-bit memory addressing)



This memory map contains gaps; that is also quite common.



Example: Consider a simple system built around an 8-bit microprocessor. Such a CPU might provide 16-bit address lines, allowing it to address up to 64 kibibytes (KiB) of memory. On such a system, perhaps the first 32 KiB of address space would be allotted to random access memory (RAM), another 16K to read only memory (ROM) and the remainder to a variety of other devices such as timers, counters, video display chips, sound generating devices, and so forth. The hardware of the system is arranged so that devices on the address bus will only respond to particular addresses which are intended for them; all other addresses are ignored. This is the job of the address decoding circuitry, and it is this that establishes the memory map of the system.



The principle function of memory interfacing is to enable the microprocessor to read or write into a register of the memory chip.



Task

Explain the features of memory mapped I/O.

8.4.2 Port Mapped vs Memory Mapped I/O

Microprocessors normally use two methods to connect external devices: memory mapped and port mapped I/O. To understand how to emulate microprocessors (for gaming or other purposes) it is important to understand this subtle difference. As far as the peripheral is concerned, both methods are really identical. A device connected to a microprocessor must decode its address from various numbers of address lines and read/write its data from various numbers of data lines. The difference between the two schemes occurs within the microprocessor. Intel has, for the most part, used the port mapped scheme for their microprocessors and Motorola has used the memory mapped scheme. It is certainly possible for a hardware engineer to design a system with memory mapped I/O which uses a CPU supporting port mapped I/O; below you can see why that would not necessarily be a good idea. Here are the basic differences:

Memory Mapped I/O devices are mapped into the system memory map along with RAM and ROM. To access a hardware device, simply read or write to those ‘special’ addresses using the normal memory access instructions. The advantage to this method is that every instruction which can access memory can be used to manipulate an I/O device. The disadvantage to this method is that the entire address bus must be fully decoded for every device. For example, a machine with a 32-bit address bus would require logic gates to resolve the state of all 32 address lines to properly decode the specific address of any device. This increases the cost of adding hardware to the machine.

Port Mapped I/O devices are mapped into a separate address space. This is usually accomplished by having a different set of signal lines to indicate a memory access versus a port access. The address lines are usually shared between the two address spaces, but less of them are used for accessing ports. An example of this is the standard PC which uses 16 bits of port address space, but 32 bits of memory address space. The advantage to this system is that less logic is needed to decode a discrete address and therefore less cost to add hardware devices to a machine. On the

Notes

older PC compatible machines, only 10 bits of address space were decoded for I/O ports and so there were only 1024 unique port locations; modern PC's decode all 16 address lines. To read or write from a hardware device, special port I/O instructions are used. From a software perspective, that this is a slight disadvantage because more instructions are required to accomplish the same task.



If you wanted to test one bit on a memory mapped port, there is a single instruction to test a bit in memory, but for ports you must read the data into a register, then test the bit.

The way this applies to arcade emulation is that the port I/O handling routines of your CPU emulator do not need to have special flags or logic for handling reads and writes to the port address space since every access to a port will be for the purpose of communicating with a hardware device. Therefore, in the simplest case, the port I/O handler routine will simply be a switch statement on the port address. Memory mapped I/O is harder to emulate because you must maintain some sort of flag list for each address to know if it is ROM, RAM or a 'special' address referencing a hardware device.



Case Study

Touch Tablets

Touch tablets are an interesting subject for a case study. On the one hand, they are simplicity personified. They are just a flat surface that can sense that it has been touched and communicate to the computer the location of where that touch occurred. On the other hand, they form the basis for an extremely broad and diverse set of physical and logical manifestations, as well as interaction techniques. Hence, they constitute a rich source for improving our understanding of input.

In simplest terms, a touch tablet is typically mounted horizontally on a working surface and operated with a single finger. But from this basic configuration is a broad range of variations. They can range in size from an inch per side to several feet. They differ in how much pressure is required before a touch is registered. Some are capable of continuously reporting to the computer the amount of pressure being applied by the touch. Some are able to be operated with a stylus as well as a finger, while others are capable of independently sensing the location (and sometimes pressure) of multiple simultaneous points of touch. And, as is illustrated in Figure 8.5, they can be integrated into other devices such as a keyboards or mice.

The biggest problem in any discussion around touch tablets stems from confusing them with touch screens. The problem is legitimate since the differences between the two are not always a clear cut as one might first think. Both are controlled by touch. With touch screens, the touch technology is superimposed over a display. But what about the input device. It is technically a touch screen, since the touch sensor is over a display. On the other hand, it is more like a touch tablet, since it is not mounted on the primary visual display, and is horizontally mounted in a tablet-like fashion.



Contd...

Questions:

1. Explain working of Touch tablet.
2. What are the basic problems in Touch tablet?

Notes

8.5 Summary

- GUIs can be used in computers, hand-held devices such as MP3 players, portable media players or gaming devices, household appliances and office equipment.
- Facility type labels and V/C (volume delay) curves for each facility type from the user's link file are set in this worksheet.
- Highway network data describe network nodes and links.
- Memory-mapped I/O uses the same address bus to address both memory and I/O devices - the memory and registers of the I/O devices are mapped to address values.

8.6 Keywords

- **ACM(Alternatives comparison module):** The alternatives comparison module (ACM) calculates user benefits resulting from the ITS deployments.
- **DMA(Direct memory access):** Direct memory access is a feature of modern computers and microprocessors that allows certain hardware subsystems within the computer to access system memory for reading and/or writing independently of the central processing unit (CPU).
- **MMIO (Memory-mapped I/O):** Memory-mapped I/O is an I/O scheme where the device's own on-board memory is mapped into the processor's address space. Data to be written to the device is copied by the driver to the device memory, and data read in by the device is available in the shared memory for copying back into the system memory.
- **TDM(Travel Demand Model):** Travel demand modeling is the utilization of computer software package to replicate to the real world transportation system around us (road traffic control devices etc).

**Lab Exercise**

1. How to interface 8085 I/O without using interfacing devices?
2. Explain the different types of file with the help of example.

8.7 Self-Assessment Questions

1. Memory mapped IO uses a portion of to handle IO requests.
 - (a) RAM
 - (b) ROM
 - (c) DRAM
 - (d) SRAM
2. The principle function of memory interfacing is not to enable the microprocessor to read or write into a register of the memory chip.
 - (a) True
 - (b) False
3. In I/O mapped I/O; I/O devices are handled distinctly by the
 - (a) Monitor
 - (b) Pointer
 - (c) CU
 - (d) CPU

Notes

4. Port-mapped I/O uses a special class of CPU instructions specifically for performing I/O.
(a) True (b) False
5. A peripheral device that assigns specific memory locations to input and output.
(a) True (b) False

8.8 Review Questions

1. Explain advantages and disadvantages of interfacing.
2. What do we mean by direct memory access?
3. What is DMA?
4. Distinguish between Ports mapped vs. memory mapped I/O.
5. What is interfacing of memory and IO?
6. What are the advantages of memory mapped IO?
7. What is the need of memory interfacing in microprocessor?
8. What is the difference between memories and mapped I/O?
9. What are I/O devices interfacing?
10. Explain the graphical user interface.

Answers for Self-Assessment Questions

1. (a)
2. (b)
3. (d)
4. (a)
5. (a)

8.9 Further Reading



Books

Microprocessors and its Applications, A.P.Godse, D.A.Godse



Online link

<http://www.zap.org.au/elec2041-cdrom/unsw/elec2041/experiment4.pdf>

Unit 9: Introduction to 8085 Instructions

Notes

CONTENTS

- Objectives
- Introduction
 - 9.1 Data Transfer (Copy) Operations
 - 9.2 Arithmetic Operations
 - 9.3 Logical Operations
 - 9.4 Branching Operations
 - 9.5 Machine Control Operations
 - 9.6 Stack I/O Operations
 - 9.7 Instruction Format
 - 9.7.1 Instruction Word Size
 - 9.8 Summary
 - 9.9 Keywords
 - 9.10 Self-Assessment Questions
 - 9.11 Review Questions
 - 9.12 Further Reading

Objectives

After studying this unit, you will be able to understand the following:

- Discuss about the data transfer (copy) operations
- Discuss about the arithmetic operations
- Discuss about the logical operations
- Discuss about the branching operations
- Discuss about the machine-control operations
- Explain about the instruction format

Introduction

An instruction is a binary pattern designed inside a microprocessor to perform a specific function. The entire group of instructions, called the instruction set, determines what functions the microprocessor can perform. These instructions can be classified into the following five functional categories: data transfer (copy) operations, arithmetic operations, logical operations, branching operations, and machine-control operations.

9.1 Data Transfer (Copy) Operations

This group of instructions copy data from a location called a source to another location called a destination, without modifying the contents of the source. In technical manuals, the term data transfer is used for this copying function. However, the term transfer is misleading; it creates the

Notes

impression that the contents of the source are destroyed when, in fact, the contents are retained without any modification. The various types of data transfer (copy) are listed below together with examples of each type:

Types	Examples
1. Between Registers.	1. Copy the contents of the register B into register D.
2. Specific data byte to a register or a memory location.	2. Load register B with the data byte 32H.
3. Between a memory location and a register.	3. From a memory location 2000H to register B.
4. Between an I/O device and the accumulator.	4. From an input keyboard to the accumulator.

The data transfer instructions move data between registers or between memory and registers.

MOV: Move

MVI: Move Immediate

LDA: Load Accumulator Directly from Memory

STA: Store Accumulator Directly in Memory

LHLD: Load H & L Registers Directly from Memory

SHLD: Store H & L Registers Directly in Memory

An ‘X’ in the name of a data transfer instruction implies that it deals with a register pair (16-bits);


LXI: Load Register Pair with Immediate data

LDAX: Load Accumulator from Address in Register Pair

STAX: Store Accumulator in Address in Register Pair

XCHG: Exchange H & L with D & E

XTHL: Exchange Top of Stack with H & L



Did u know?

The microprocessor, also known as a CPU (central processing unit).

9.2 Arithmetic Operations

These instructions perform arithmetic operations such as addition, subtraction, increment, and decrement.

Addition: Any 8-bit number, or the contents of a register or the contents of a memory location can be added to the contents of the accumulator and the sum is stored in the accumulator. No two other 8-bit registers can be added directly (e.g., the contents of register B cannot be added directly to the contents of the register C). The instruction DAD is an exception; it adds 16-bit data directly in register pairs.

Subtraction: Any 8-bit number, or the contents of a register, or the contents of a memory location can be subtracted from the contents of the accumulator and the results stored in the accumulator. The subtraction is performed in 2’s compliment, and the results if negative, are expressed in 2’s complement. No two other registers can be subtracted directly.

Notes

Increment/Decrement: The 8-bit contents of a register or a memory location can be incremented or decrement by 1. Similarly, the 16-bit contents of a register pair (such as BC) can be incremented or decrement by 1. These increment and decrement operations differ from addition and subtraction in an important way; i.e., they can be performed in any one of the registers or in a memory location.

The arithmetic instructions add, subtract, increment, or decrement data in registers or memory.

ADD: Add to Accumulator

ADI: Add Immediate Data to Accumulator

ADC: Add to Accumulator Using Carry Flag

ACI: Add Immediate data to Accumulator Using Carry

SUB: Subtract from Accumulator

SUI: Subtract Immediate Data from Accumulator

SBB: Subtract from Accumulator Using Borrow (Carry) Flag

SBI: Subtract Immediate from Accumulator Using Borrow (Carry) Flag

INR: Increment Specified Byte by One

DCR: Decrement Specified Byte by One

INX: Increment Register Pair by One

DCX: Decrement Register Pair by One

DAD: Double Register Add; Add Content of Register

Pair to H & L Register Pair



Task

Write an assembly program to implement subtraction on two numbers.

9.3 Logical Operations

These instructions perform various logical operations with the contents of the accumulator. AND, OR Exclusive-OR - Any 8-bit number, or the contents of a register, or of a memory location can be logically ANDed, Ored, or Exclusive-ORed with the contents of the accumulator. The results are stored in the accumulator. Rotate- Each bit in the accumulator can be shifted either left or right to the next position. Compare- Any 8-bit number or the contents of a register, or a memory location can be compared for equality, greater than, or less than, with the contents of the accumulator. Complement - The contents of the accumulator can be complemented. All 0s are replaced by 1s and all 1s are replaced by 0s.



Caution

The logical AND, OR, and Exclusive OR instructions enable you to set specific bits in the accumulator ON or OFF.

ANA: Logical AND with Accumulator

ANI: Logical AND with Accumulator Using Immediate Data

ORA: Logical OR with Accumulator

OR: Logical OR with Accumulator Using Immediate Data

Notes

XRA: Exclusive Logical OR with Accumulator

XRI: Exclusive OR Using Immediate Data

The Compare instructions compare the content of an 8-bit value with the contents of the accumulator;

CMP: Compare

CPI: Compare Using Immediate Data

The rotate instructions shift the contents of the accumulator one bit position to the left or right:

RLC: Rotate Accumulator Left

RRC: Rotate Accumulator Right

RAL: Rotate Left Through Carry

RAR: Rotate Right Through Carry

Complement and carry flag instructions:

CMA: Complement Accumulator

CMC: Complement Carry Flag

STC: Set Carry Flag



Did u know?

In 1971, the Intel 4004 processor held 2,300 transistors. In 2010, an Intel Core processor that includes a 32nm processing die with second-generation High-k metal gate silicon technology holds 560 million transistors.

9.4 Branching Operations

This group of instructions alters the sequence of program execution either conditionally or unconditionally.

Jump -Conditional jumps are an important aspect of the decision-making process in the programming. These instructions test for a certain conditions (e.g., Zero or Carry flag) and alter the program sequence when the condition is met. In addition, the instruction set includes an instruction called unconditional jump.

Call, Return, and Restart - These instructions change the sequence of a program either by calling a subroutine or returning from a subroutine. The conditional Call and Return instructions also can test condition flags.

The unconditional branching instructions are as follows:

JMP: Jump

CALL: Call

RET: Return

Conditional branching instructions examine the status of one of four condition flags to determine whether the specified branch is to be executed. The conditions that may be specified are as follows:

NZ: Not Zero ($Z = 0$)

Z: Zero ($Z = 1$)

NC: No Carry ($C = 0$)

C: Carry ($C = 1$)


- PO:** Parity Odd (P = 0)
- PE:** Parity Even (P = 1)
- P:** Plus (S = 0)
- M:** Minus (S = 1)

Thus, the conditional branching instructions are specified as follows:

Jumps	Calls	Returns
C	CC	RC (Carry)
INC	CNC	RNC (No Carry)
JZ	CZ	RZ (Zero)
JNZ	CNZ	RNZ (Not Zero)
JP	CP	RP (Plus)
JM	CM	RM (Minus)
JPE	CPE	RPE (Parity Even)
JPO	CPO	RPO (Parity Odd)

Two other instructions can affect a branch by replacing the contents or the program counter:

- PCHL:** Move H & L to Program Counter
- RST:** Special Restart Instruction Used with Interrupts



Task Give the difference between conditional and unconditional branching instructions.

9.5 Machine Control Operations

These instructions control machine functions such as Halt, Interrupt, or do nothing. The microprocessor operations related to data manipulation can be summarized in four functions:

1. Copying data
2. Performing arithmetic operations
3. Performing logical operations
4. Testing for a given condition and alerting the program sequence

Some important aspects of the instruction set are noted below:

1. In data transfer, the contents of the source are not destroyed; only the contents of the destination are changed. The data copy instructions do not affect the flags.
2. Arithmetic and Logical operations are performed with the contents of the accumulator, and the results are stored in the accumulator (with some expectations). The flags are affected according to the results.
3. Any register including the memory can be used for increment and decrement.

Notes

4. A program sequence can be changed either conditionally or by testing for a given data condition.

The Machine Control instructions are as follows:

EI: Enable Interrupt System

DI: Disable Interrupt System

HLT: Halt

NOP: No Operation



Did u know?

Intel was the first company to produce a microprocessor for commercial use. Called the 4004, it was released in the early 1970s and contained slightly more than 2,000 transistors.

9.6 Stack I/O Operations

The following instructions affect the Stack and/or Stack Pointer:

PUSH: Push Two bytes of Data onto the Stack

POP: Pop Two Bytes of Data off the Stack

XTHL: Exchange Top of Stack with H & L

SPHL: Move content of H & L to Stack Pointer

The I/O instructions are as follows:

IN: Initiate Input Operation

OUT: Initiate Output Operation

9.7 Instruction Format

An instruction is a command to the microprocessor to perform a given task on a specified data. Each instruction has two parts: one is task to be performed, called the operation code (opcode), and the second is the data to be operated on, called the operand. The operand (or data) can be specified in various ways. It may include 8-bit (or 16-bit) data, an internal register, a memory location, or 8-bit (or 16-bit) address.

In some instructions, the operand is implicit.

9.7.1 Instruction Word Size

The 8085 instruction set is classified into the following three groups according to word size:

1. One-word or 1-byte instructions
2. Two-word or 2-byte instructions
3. Three-word or 3-byte instructions

In the 8085, “byte” and “word” are synonymous because it is an 8-bit microprocessor.

However, instructions are commonly referred to in terms of bytes rather than words.

One-Byte Instructions

A 1-byte instruction includes the opcode and operand in the same byte. Operand(s) are internal register and are coded into the instruction.

For example:

Notes

Task	Op code	Operand	Binary Code	Hex Code
Copy the contents of the accumulator in the register C.	MOV	C,A	01001111	4FH
Add the contents of register B to the contents of the accumulator	ADD	B	1000 0000	80H
Invert (compliment) each bit in the accumulator.	CMA		00101111	2FH

These instructions are 1-byte instructions performing three different tasks. In the first instruction, both operand registers are specified. In the second instruction, the operand B is specified and the accumulator is assumed. Similarly, in the third instruction, the accumulator is assumed to be the implicit operand. These instructions are stored in 8- bit binary format in memory; each requires one memory location.

```
MOV rd, rs
rd <- rs copies contents of rs into rd.
```

Coded as 01 ddd sss where ddd is a code for one of the 7 general registers which is the destination of the data, sss is the code of the source register.

 Example: MOV A, B

Coded as 01111000 = 78H = 170 octal (octal was used extensively in instruction design of such processors).

```
ADD r
A <- A + r
```

Two-Byte Instructions

In a two-byte instruction, the first byte specifies the operation code and the second byte specifies the operand. Source operand is a data byte immediately following the opcode. For example:

Task	Op code	Operand	Binary Code	Hex Code	
Load an 8-bit data byte in the accumulator.	MVI	A,	0011 1110	Data 3E	First Byte
				Data	Second Byte


Assume that the data byte is 32H. The assembly language instruction is written as

Mnemonics	Hex code
MVI A, 32H	3E 32H

Notes

The instruction would require two memory locations to store in memory.

```
MVI r, data
r ← data
```

 Example: MVI A, 30H coded as 3EH 30H as two contiguous bytes. This is an example of immediate addressing.

```
ADI data
A ← A + data
OUT port
0011 1110
DATA
```

Where port is an 8-bit device address. (Port) ← A. Since the byte is not the data but points directly to where it is located this is called direct addressing.

Three-Byte Instructions

In a three-byte instruction, the first byte specifies the opcode, and the following two bytes specify the 16-bit address.

opcode + data byte + data byte
For example:

Task	Op code	Operand	Binary Code	Hex Code	
Transfer the program sequence to the memory location 2085H.	JMP	2085H	1100 0011	C3	First byte
			1000 0101	85	Second Byte
			0010 0000	20	Third Byte

 The second byte is the low-order address and the third byte is the high-order address.
Caution

This instruction would require three memory locations to store in memory. Three byte instructions - opcode + data byte + data byte

```
LXI rp, data16
```

rp is one of the pairs of registers BC, DE, HL used as 16-bit registers. The two data bytes are 16-bit data in L H order of significance.

```
rp ← data16
```

 Example: LXI H, 0520H coded as 21H 20H 50H in three bytes. This is also immediate addressing.

```
LDA addr
```

A ← (addr) Addr is a 16-bit address in L H order. Example: LDA 2134H coded as 3AH 34H 21H. This is also an example of direct addressing.



Case Study

The History of Intel Organization

Intel began in 1968. It was founded by Gordon E. Moore who is also a physicist and chemist. He was accompanied by Robert Noyce, also a fellow physicist and co-creator of integrated circuitry, after they both had left Fairchild Semiconductor. During the 1980's Intel was run by a chemical engineer by the name of Andy Grove, who was the third member of the original Intel family. Many other Fairchild employees participated in other Silicon Valley companies. Andy Grove today is considered to be one of the company's essential business and strategic leaders. As the 1990's concluded, Intel had become one of the largest and by far the most successful businesses in the entire world. Intel has gone through many faces and phases. In the beginning Intel was set apart by its ability primarily to create memory chips or SRAM.

When the firm was founded, Gordon Moore and Robert Noyce had the idea to name their company Moore Noyce. However when the name is spoken it is heard as "More Noise" This idea was quickly abandoned and the pursuit of a more suitable name – one which was not associated with a bad interface. The name NM Electronics was shortly thereafter chosen and used for nearly a year, when the company experienced a name change to Integrated Electronics, or INTEL for short. The rights to the name however had to be purchased as it was already in use by a fairly well known hotel chain.

Though Intel had mastered the first microprocessor called the Intel 4004 in 1971 and also one of the worlds very first microcomputers in 1972, in the early 80's the focus was primarily on Random Access Memory chips. A new client in the early 70's from Japan wanted to enlist the services of Intel to design twelve chips for their calculators. Knowing that they did not have the manpower or the resources to complete this job effectively, Ted Hoff agreed to the job just the same. His idea was: What if we can design one computer chip which could function the same as twelve microchips? Hoof's idea was completely embraced by Moore and Noyce. If this project were successful the chip would have the ability to receive command functions. This is where the 4004 model came from. After a painstaking 9 months. It measured 1/8th inch by 1/6th inch long and contained 2,300 transistors. History was made and changed that day.

The Pentium Pro processor had 5.5 million transistors, making the chip so affordable that it could be imbedded in common household appliances. After this success Intel decided to completely embrace this and to pursue its production. Some notable dates in the history of Intel are:

1968 Robert Noyce and Gordon Moore incorporate NM Electronics

1970 The development of DRAM and dynamic RAM

1971 The world's first microcomputer is introduced

1974 The first general purpose microprocessor is introduced to the world

1980 The Intel microprocessor is chosen by IBM for the first ever personal computer. 1992 Intel's net income tops the one billion dollar point

1993 The Pentium is introduced, a fifth generation chip

1996 Intel's revenue exceeds twenty billion dollars and the net income surpasses five billion dollars

1997, The Pentium 11 microprocessor is introduced to the world

1999 Intel is added to Dow Jones Averages.

Contd...

Notes

2000 The world's very first Intel 1 gigahertz processor hits the shelves.

To this day Intel continues to make strides in the computing and micro computing world.

Questions:

1. Name the founder of Intel Company?
2. Give the year name when Intel launches first processor.

9.8 Summary

- Instruction is a binary pattern designed inside a microprocessor to perform a specific function.
- Data transfer operation copy data from a location called a source to another location called a destination, without modifying the contents of the source.
- An arithmetic instruction performs arithmetic operation on given pattern.
- Logical instruction use to perform logical operations.
- Branching instruction alters the sequence of program execution either conditionally or unconditionally.
- Machine control instructions control machine functions.

9.9 Keywords

ADD: Add data to Accumulator

ANA: Logical AND with Accumulator

LDA: Load Accumulator Directly from Memory

MOV: Move data between memory and registers

RAL: Rotate Left Through Carry

SBB: Subtract from Accumulator Using Borrow (Carry) Flag

SHLD: Store H & L Registers Directly in Memory

STA: Store Accumulator Directly in Memory

SUB: Subtract data from Accumulator

XCHG: Exchange H & L with D & E

XRI: Exclusive OR Using Immediate Data

XTHL: Exchange Top of Stack with H & L



Lab Exercise

1. Write an assembly program to add two numbers Program.
2. Write an assembly program to find greatest between two numbers Program.

9.10 Self-Assessment Questions

1. The data transfer instruction use to perform the logical operations on pattern.
(a) True (b) False

Notes

- 2. The instruction INR use to:
 - (a) Decrement specified byte by one
 - (b) Increment register pair by one
 - (c) Increment specified byte by one
 - (d) Decrement register pair by one
- 3. The instruction INX use to:
 - (a) Decrement specified byte by one
 - (b) Increment register pair by one
 - (c) Increment specified byte by one
 - (d) Decrement register pair by one
- 4. The instruction DCX use to:
 - (a) Decrement specified byte by one
 - (b) Increment register pair by one
 - (c) Increment specified byte by one
 - (d) Decrement register pair by one
- 5. The instruction CMP use to:
 - (a) Decrement specified byte by one
 - (b) Increment register pair by one
 - (c) Compare data
 - (d) Decrement register pair by one
- 6. operations alter the sequence of program execution either conditionally or unconditionally.
 - (a) Logical
 - (b) Arithmetic
 - (c) Branch
 - (d) None
- 7. JMP is a conditional branching instruction.
 - (a) True
 - (b) False
- 8. EI is used to enable interrupt system.
 - (a) True
 - (b) False
- 9. instruction use to move content of H & L to Stack Pointer.
 - (a) PUSH
 - (b) XTHL
 - (c) POP
 - (d) SPHL
- 10. instruction use to exchange top of stack with H & L.
 - (a) PUSH
 - (b) XTHL
 - (c) POP
 - (d) SPHL

9.11 Review Questions

- 1. What is an instruction?
- 2. What is the function of the accumulator?
- 3. Give the functional categories of 8085 microinstructions.
- 4. Define Opcode and operand.
- 5. Define the types of branching operations.
- 6. Define two-byte instruction with one example.
- 7. Write instructions to load the hexadecimal numbers 65H in register C, and 92h in the accumulator A .Display the number 65H at PORT0 and 92H at PORT1.
- 8. How is the instruction set classified?

Notes

9. What is an OUT instruction?
10. Give the difference between JZ and JNZ.
11. What is CMA?
12. What is CALL instruction?

Answers for Self-Assessment Questions

- | | | | | |
|--------|--------|--------|--------|---------|
| 1. (b) | 2. (c) | 3. (b) | 4. (d) | 5. (c) |
| 6. (c) | 7. (b) | 8. (a) | 9. (d) | 10. (b) |

9.11 Further Reading



Books

Microprocessor 8085 and its Interfacing, By Mathur



Online link

<http://books.google.com/books?>

**Unit 10: Programming Techniques with
Additional Instructions**

Notes

CONTENTS

Objectives

Introduction

10.1 Looping, Counting and Indexing

 10.1.1 Continuous Loop

 10.1.2 Conditional Loop

10.2 Additional Data Transfer Instructions

 10.2.1 16-bit Data Transfer to Registers Pairs

 10.2.2 Data Transfer between Memory and Microprocessor

10.3 Additional Arithmetic Operations

 10.3.1 Related to Memory

 10.3.2 Related to 16-bits (Register Pairs)

10.4 Advanced Logical Operations

 10.4.1 Rotate

 10.4.2 Compare

10.5 Summary

10.6 Keywords

10.7 Self-Assessment Questions

10.8 Review Questions

10.9 Further Reading

Objectives

After studying this unit, you will able to understand the following:

- Discuss about the looping, counting and indexing
- Explain about the additional data transfer instruction
- Explain about the additional arithmetic operations
- Explain about the additional logical operations

Introduction

In this unit, we are going to study how to implement programming techniques and some programming examples using them. Before going to implement these techniques, we get conversant with these techniques and understand the use of them.

10.1 Looping, Counting and Indexing

When repetitive task are considered, computers are more efficient than human beings fast and accurate, when have to add hundreds numbers or transferring a thousand bytes of data.

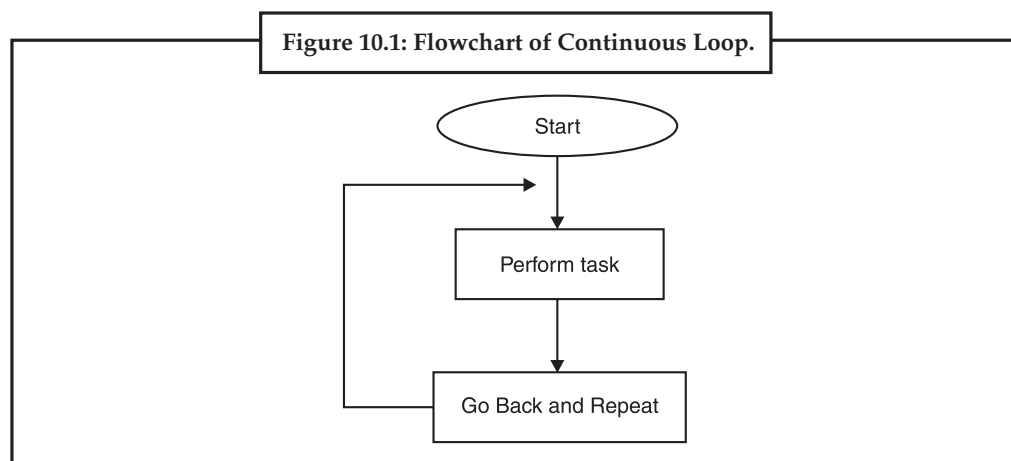
Looping is the programming technique used to instruct the microprocessor to repeat tasks. A loop is the procedure which asks the microprocessor to change the sequence of execution and perform the task repeatedly. These set of instructions also include counting i.e., how many times the task should be repeated and indexing is use to keep track of the sequential execution after the jump. Counting is technique allows programmer to count how many times the instruction/set of instructions are executed. Indexing allows programmer to point or refer the data stored in sequential memory locations one by one.

Loops can be classified into two jumps:

1. Continuous loop-repeats the tasks continuously.
2. Conditional loop-repeats the task until certain data conditions are met.

10.1.1 Continuous Loop

A continuous loop is setup by using the unconditional jump instructions. The unconditional jump instruction creates a continuous loop which can be stopped only by the resetting of the system.



10.1.2 Conditional Loop

A conditional loop is setup by the conditional jump instructions. These instruction check flags: zero, carry, sign etc. and repeat the specific tasks if the conditions are satisfied. These loop usually include counting and indexing.

The conditional loops are executed in the following steps:

Step 1: Counter is setup by loading appropriate value in the register.

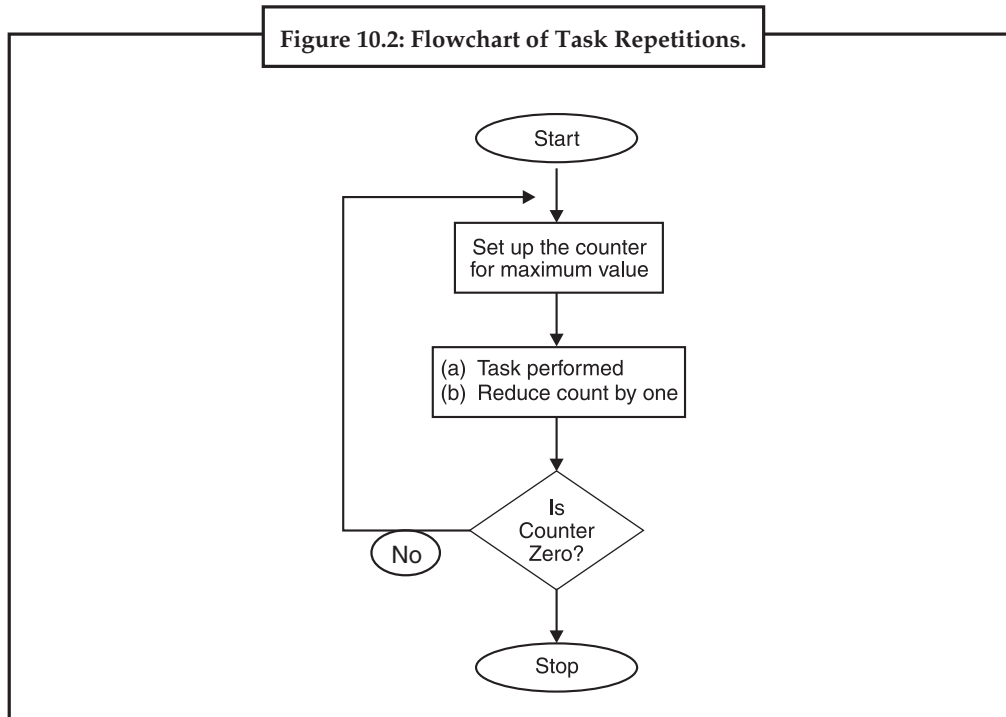
Step 2: Counting can be performed either by incrementing or decrementing the counter.

Step 3: Loops are setup by a conditional jump instructions.

Step 4: Flag denotes the end of counting.

It is easier to count down to zero than to count up. In counting down, the zero flag is set when the register value becomes zero. In counting up requires the compare instruction.

The flowchart for these steps can be as follows:



Another type of loops include indexing along with the counter.



Indexing is defined as a pointer or reference to an object with sequential numbers. Data bytes are stored in memory locations and those data bytes are referred by their memory location.

These loops are executed in the following steps:

Step 1: Counter is setup by loading appropriate value in the register.

Step 2: An index or a memory pointer is required to locate where data bytes are stored.

Step 3: Data should be transferred from memory location to the microprocessor (ALU).

Step 4: Perform all data manipulation using ALU.

Step 5: The partial results should be temporarily stored in the register

Step 6: Decision making step, it decides whether the task is-completed or not indicated by a flag check.

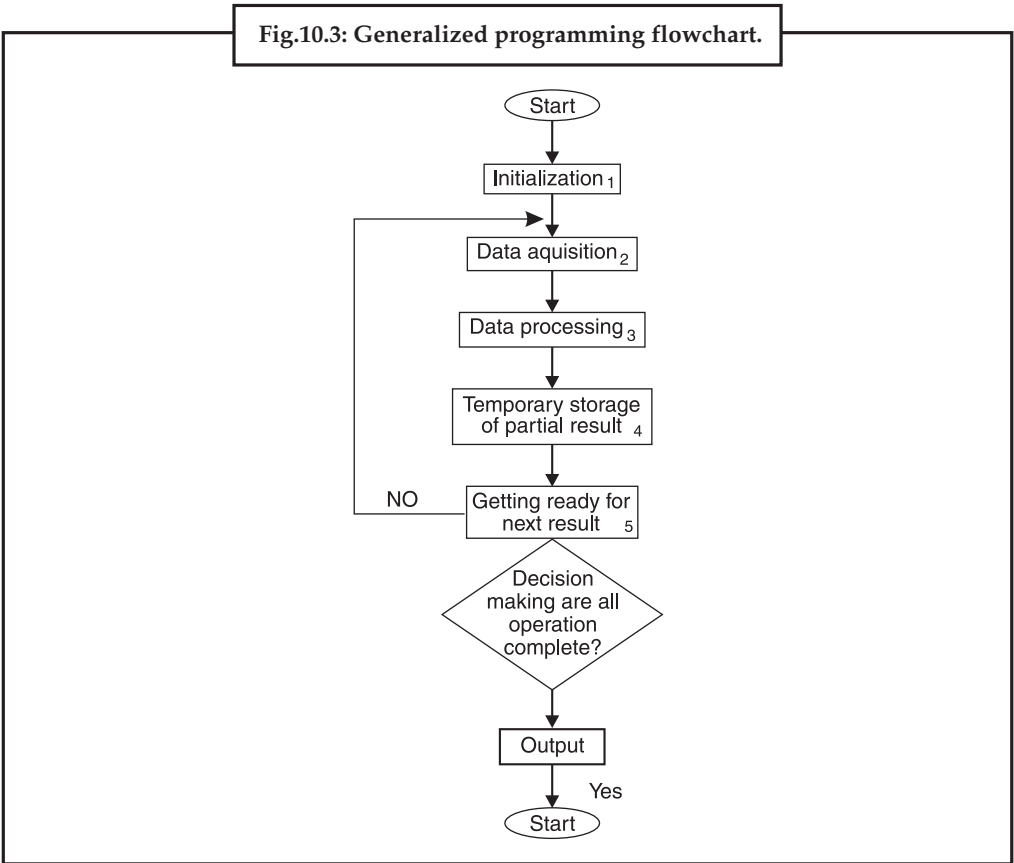
Step 7: The result should be stored or send to an output port.

These steps can be converted into a flowchart as shown in Fig. 10.2. It is a generalized flowchart which can be used to solve many problems. Some steps or blocks can be expanded with additional loops or some blocks may need to be interchanged in their positions.


Key points to remembers:

1. Computer is best in repeating tasks.
2. It is fast and accurate.
3. Loops are setup by using the looping technique along with counting and indexing.
4. The computer is a versatile and powerful tools.
5. setting of loops are capable in making the decision which are based on data conditions.

Notes



- 1. Planning stage: In this step setting up is done. It is most difficult step as initial setup depends on the requirement of the following step.
- 2. Data are generally stored in memory or read from the input port. In this step the data is brought into the microprocessor for operations.
- 3. All data manipulations are carried out in this step.
- 4. Storage of partial results in registers is done in this step.
- 5. The microprocessor does not know whether it has completed all the operations. Before it can repeat the task, it needs to get ready for the next operations.
- 6. This is a decision-making step. It decides whether to change the sequence of execution and repeat or to go to the next instruction.
- 7. This step involves either sending the result to an output port or storing in the memory.

 *Example:* The following block of data is stored in the memory locations from 2050H to 2055H. Transfer the data to the location 4055H to 405AH in the reverse order.

Data (H): 22, B4, 1A, 27, 78, 99

Solution: The assembly language program for the above stated problem is as follows:

Memory	Label	Opcode	Operands	Remarks
21	START:	LXI	H, 2050H	; Index for data source
50				
11		LXI	D, 405AH	; Index for data

Contd. ...

5A				; destination, Starting
40				; at last location
06		MVI	B, 06H	; Byte counter
06				
7E	NEXT:	MOV	A, M	; Get data byte
12		STAX	D	; Store data byte
23		INX	H	; Next location
1B		DCX	D	
05		DCR	B	; Next count
C2		JNZ	NEXT	; If counter is not 0, go
X				; back to transfer next byte
X				
76.		HLT		; End of program.

Notes

10.2 Additional Data Transfer Instructions

Additional data transfer instructions are between the microprocessor and memory.

10.2.1 16-bit Data Transfer to Registers Pairs

This is done with LXI mnemonic of assembly language. LXI is similar to MVI instruction except LXI loads 16-bit data in register pairs and the stack pointer register. No flags are affected by these instructions. LXI is a 3-byte instruction where as MVI is a 2-byte instruction.

Table 10.1 List of Instructions of 16-bit Data Transfer to Register Pairs (LXI).

Opcode	Operand	Description
LXI	R, 16-bit	Load Register Pair Immediate
LXI	B, 16-bit	(a) It is a 3-byte instruction
LXI	D, 16-bit	(b) Second byte is loaded in the low-order register of the register pair
LXI	H, 16-bit	(c) Third byte is loaded in the high-order register of the register pair.
LXI	SP, 16-bit	(d) The operands B, D, H, represents BC, DE, HL registers and SP represents the stack pointer register.

The basic difference between MVI and LXI are:

1. LXI can load 16-bit data in register pairs and stack pointer register where as MOV can copy data from source register to destination register and MVI can load 8-bit data into the specified register.
2. LXI is a 3-byte instruction whereas MVI and MOV are 2-byte and 1-byte instructions.

Notes

3. When data is loaded using LXI instruction then low-order byte first followed by the high-order byte where as in MVI high-order byte first followed by the low-order byte.

10.2.2 Data Transfer between Memory and Microprocessor

The 8085 instruction set includes instructions having data transfer:

- 1. From memory to the microprocessor.
- 2. From microprocessor to the memory or directly into the memory.

(1) From Memory to the Microprocessor

The 8085 instruction set includes three types of memory transfer instructions. These instructions do not affect the flags.

Table 10.2: List of Instructions of Data Transfer from Memory to the Microprocessor.

Opcode	Operands	Remarks
MOV	R, M	<i>Move data from memory to register</i> (a) It is a 1-byte instruction (b) Copies data from memory location into a register
LDAX	B/D	(c) (HL) register specifies the memory location
LDAX	B	(d) Indirect addressing mode is used
LDAX	D	<i>Load accumulator indirect</i> (a) It is a 1-byte instruction (b) Copies data from memory location into the Accumulator (c) (BC) or (DE) register contents specifies the memory location.
	16-bit	(d) Indirect addressing mode is used.
LDA		<i>Load accumulator direct.</i> (a) It is a 3-byte instruction (b) Copies data from memory location specified by the 16-bit address in the second and third byte. (c) Second byte is a line number and third byte is a page number. (d) Direct addressing mode is used.



Caution

Take care of basic difference between MVI and LXI.

R can be A, B, C, D, E, H, L any of the register.


In the above stated three instructions, it is observed that indirect addressing mode requires more 1 byte as compared to the direct addressing mode when data to be transfer is only one byte then

LDA proves to be more efficient as compared to LDAX and MOV R, M. But for a block of memory transfer, the instruction LDA (three bytes) will have to be repeated for each memory. On the other hand, a loop can be set up with two other instructions and the contents of a register pair can be incremented or decremented.

(2) From Microprocessor to Memory or Directly into Memory

The instructions belonging to this category are listed in Table 10.3.

Table 10.3: List of Instructions Related to Data Transfer from Microprocessor to Memory.		
Opcode	Operands	Description
MOV	M, R	<i>Move data from register to memory</i> (a) It is a 1-byte instruction (b) Copies the data from a register R, into the memory location specified by the contents of HL register pair (c) Uses indirect addressing mode.
STAX	B/D	<i>Store Accumulator indirect</i> (a) It is a 1-byte instruction (b) Copies data from the Accumulator into the memory location specified by the contents of either BC or DE registers. (c) Uses indirect addressing mode
STA	16-bit	<i>Store Accumulator direct</i> (a) It is a 3-byte instruction (b) Copies data from the Accumulator into the memory location specified by the 16-bit operand. (c) Uses direct addressing mode.
MVI	M, 8-bit	<i>Load 8-bit data is memory</i> (a) It is a 2-byte instruction (b) Second byte specifies 8-bit data (c) Memory location is specified by the contents of the HL register. (d) Uses immediate addressing mode.

 *Example:* Register B contains 48H. Use memory location 8000H for indirect addressing mode for instructions **MOV, STAX, MVI**. **Also use direct** addressing mode for STA instruction.

Solution: In MOV instructions, the byte 48H is copied from register B into memory location 8000H by using the HL as a memory pointer. In STAX instructions, DE register is used as a memory pointer, the byte 48H must be copied from B into the Accumulator first as the STAX instruction copies only from the Accumulator.

The STA instruction copies 48H from the Accumulator into the memory location 8000H.

The memory address is specified as the operand. The MVI instruction, load a byte directly in memory location by using the HL as a memory pointer.

Notes	(a)	Machine code	Opcode	Operands
		21	LXI	H, 8000H
		00		
		80		
	(b)	70	MOV	M, B
		Machine code	Opcode	Operands
		11	LXI	D, 8000H 00
		80		
		78	MOV	A, B
		12	STAX	D

10.3 Additional Arithmetic Operations

Simple arithmetic operations describe the operations between numbers or contents of the microprocessor register. The advanced arithmetic operations include memory as one of its operand. The arithmetic instructions referenced to memory perform two tasks:

- (a) Copy a byte from a memory location to the microprocessor.
- (b) Perform arithmetic operation.

10.3.1 Related to Memory

ADD and SUB instructions implicitly assume that one of the operand is register A i.e., Accumulator. After an operation, the previous contents of the Accumulator are replaced by the result. All the flags are modified according to the result.

Table 10.4: List of Memory Related Arithmetic Operations.		
Opcode	Operands	Description
ADD	M	<i>Add memory contents</i> (a) It is a 1-byte instruction. (b) Adds (M) to (A) and stores result in A. (c) Contents of HL register specifies the memory location.
SUB		<i>Subtract memory</i> (a) It is a 1-byte instruction. (b) Subtract (M) from (A) and stores the result in A (c) Contents of HL register specifies the memory location.
INR	M	<i>Increments memory contents</i> (a) It is a 1-byte instruction (b) Increments the contents of memory by 1, not the memory address (c) (HL) register specifies the memory location.

Contd. ...

DCR	M	Decrements memory contents (a) It is a 1-byte instruction (b) Decrements the contents of memory by 1, not the memory address (c) (HL) register specifies the memory location.
-----	---	--

Notes

INR and DCR instructions affect all the flags except the carry (CY) flag. As all these instructions involve a memory location and a register and so uses register indirect addressing more.



Example: Write instructions to add the contents of the memory location 8000H to (A) and subtract the contents of the memory 8001H from the first sum. Assume the Accumulator has 68H the memory location 8000H has 30H, and the location 8001H has 7FH.

Solution: The contents of HL pair 8000H specify the memory location. The instruction ADD M adds 30H the contents of memory location 8000H, to the contents of the Accumulator (68H) the instruction INX H points to the next memory location 8001H and the instruction SUB M subtracts the contents (7FH) of memory location 8001 from the previous sum. The assembly language program is as follows:

Machine code (Hex)	Opcode	Operands
21	LXI	H, 8000H
00		
80		
86	ADD	M
23	INX	H
96	SUB	M

10.3.2 Related to 16-bits (Register Pairs)

The instructions related to incrementing/ decrementing 16-bit contents in a register pair are listed in Table 10.5. These instructions do not affect any of the status of flags.

Table 10.5: List of Arithmetic Operations Related to 16-bit.

Opcode	Operands	Description
INX	R	<i>Increments register pairs</i>
INX	B	(a) It is a 1-byte instruction.
INX	D	(b) Assumes the contents of two registers as one
INX	H	16-bit number
INX	SP	(c) Increments the contents by 1
DCX	R	<i>Decrements register pairs</i>
DCX	B	(a) It is a 1-byte instruction
DCX	D	(b) Assumes the contents of two registers as one
DCX	H	16-bit number.
DCX	SP	(c) Decreases the contents of registers pair by 1

Notes



Example: Write instructions to load the number 3040H in the register pair BC. Increment the number using the instruction INX B.

Solution: The assembly language program for the above stated problem is as follows:

Machine code (Hex)	Opcode	Operands
01	LXI	B, 3040H 40
30		
03	INX	B



Example: Write instructions to load the number 2054H in the register pair BC. Decrement the number using the DCX B.

Solution: The assembly language program for the above stated problem is as follows:

Machine code (Hex)	Opcode	Operands
01	LXI	B, 2054H
54		
20		
0B	DCX	B

10.4 Advanced Logical Operations

The advanced logical operations include rotate and compare instructions. The rotate instruction is related to rotating the Accumulator bits. The compare instruction compares the data byte or register/memory contents with the contents of the Accumulator.

10.4.1 Rotate

Rotate instructions rotate the Accumulator bits. This group includes four instructions. Two for rotating left and two are for rotating right.

Table 10.6: List of Rotate Instructions.

Opcode	Description
RLC	<i>Rotate Accumulator left</i> (a) Each bit is shifted to the adjacent left position (b) Bit D, becomes Do (c) CY flag is modified according to bit D7
RAL	<i>Rotate Accumulator left through carry</i> (a) Each bit is shifted to the adjacent left position (b) D7 becomes the carry bit . (c) Carry bit is shifted into Do
RRC	<i>Rotate Accumulator right</i> (a) Each bit is shifted right to the adjacent position

Contd. ...

	(b) Bit Do becomes D, (c) CY flag is modified according to bit Do
RAR	<i>Rotate Accumulator right through carry</i> (a) Each bit is shifted right to the adjacent position (b) Do bit becomes carry bit (c) Carry bit is shifted into D ₇

Notes




Example: If contents of Accumulator are 24H then rotate left by one bit and rotate right by one bit.

Solution: The solution to, above stated problem is as follows:

Binary value of 24H = 0 010 010 0 (a) Rotate 24H by 1-bit right
0001 0010=12H

This is equivalent to dividing by 2 (b) Rotate 24H by 1-bit left
0100 1000=48H

This is equivalent to multiplying by 2.



Task

Give the code for rotated instruction with example.

10.4.2 Compare

Compare instructions compares a data byte or register /memory contents with the contents of the Accumulator by subtracting the data byte from contents of A denoted by (A). It indicates whether the data byte is greater than equal to \geq or less than equal to (\leq) by modifying the flags. The contents of Accumulator are not modified. These are two types of compare operations listed in Table 10.7.

Table 10.7: List of Compare Operations.

Opcode	Operands	Description
CMP	R/M	<i>Compare register or memory contents with contents of Accumulator</i> (a) It is a 1-byte instruction. (b) If (A) <(R/M), CY is set and zero flag is reset (c) If (A) =(R/M), zero flag is set, CY flag is reset (d) If (A) >(R/M), CY and zero flag are reset (e) If memory is operand, its address is specified by (HL) (f) No contents are modified (g) All remaining flags i.e., S, P, AC are affected according to the result of subtraction.

Contd. ...

Notes

CPI	8-bit	<i>Compare immediate with Accumulator</i> (a) It.is 2-byte instruction (b) Second type is 8-bit data (c) Compares second byte with (A) (d) If (A) < 8-bit data, CY flag is set and zero flag is reset (e) If (A) = 8-bit data, zero flag is set and CY flag is reset (f) If (A) > 8-bit data, CY and zero flags are reset (g) No contents are modified (h) All remaining flags i.e., S, P, AC are affected according to the result of subtraction.
-----	-------	--



Case Study

Human-Computer Interaction (HCI)

Human-computer interaction (HCI) is an area of research and practice that emerged in the early 1980s, initially as a specialty area in computer science. HCI has expanded rapidly and steadily for three decades, attracting professionals from many other disciplines and incorporating diverse concepts and approaches. To a considerable extent, HCI now aggregates a collection of semi-distinct fields of research and practice in human-centered informatics. However, the continuing synthesis of disparate conceptions and approaches to science and practice in HCI has produced a dramatic example of how different epistemologies and paradigms can be reconciled and integrated.

Until the late 1970s, the only humans who interacted with computers were information technology professionals and dedicated hobbyists. This changed disruptively with the emergence of personal computing around 1980. Personal computing, including both personal software (productivity applications, such as text editors and spreadsheets, and interactive computer games) and personal computer platforms (operating systems, programming languages, and hardware), made everyone in the developed world a potential computer user, and vividly highlighted the deficiencies of computers with respect to usability for those who wanted to use computers as tools.

The challenge of personal computing became manifest at an opportune time. The broad project of cognitive science, which incorporated cognitive psychology, artificial intelligence, linguistics, cognitive anthropology, and the philosophy of mind, had formed at the end of the 1970s. Part of the programme of cognitive science was to articulate systematic and scientifically-informed applications to be known as “cognitive engineering”. Thus, at just the point when personal computing presented the practical need for HCI, cognitive science presented people, concepts, skills, and a vision for addressing such needs. HCI was one of the first examples of cognitive engineering.

Other historically fortuitous developments contributed to establishment of HCI. Software engineering, mired in unmanageable software complexity in the 1970s, was starting to focus on nonfunctional requirements, including usability and maintainability, and on non-linear software development processes that relied heavily on testing. Computer graphics and information retrieval had emerged in the 1970s, and rapidly came to recognize that interactive systems were the key to progressing beyond early achievements. All these threads of

development in computer science pointed to the same conclusion: The way forward for computing entailed understanding and better empowering users.

Finally human factors engineering, which had developed many techniques for empirical analysis of human-system interactions in so-called control domains such as aviation and manufacturing, came to see HCI as a valuable and challenging domain in which human operators regularly exerted greater problem-solving discretion. These forces of need and opportunity converged around 1980, focusing a huge burst of human energy, and creating a highly visible interdisciplinary project.

From cabal to community

The original and abiding technical focus of HCI is on the concept of *usability*. This concept was originally articulated naively in the slogan “easy to learn, easy to use”. The blunt simplicity of this conceptualization gave HCI an edgy and prominent identity in computing. It served to hold the field together, and to help it influence computer science and technology development more broadly and effectively. However, inside HCI the concept of usability has been reconstructed continually, and has become increasingly rich and intriguingly problematic. Usability now often subsumes qualities like fun, well-being, collective efficacy, aesthetic tension, enhanced creativity, support for human development, and many others. A more dynamic view of usability is that of a programmatic objective that should continue to develop as our ability to reach further toward it improves.

Although the original academic home for HCI was computer science, and its original focus was on personal productivity applications, mainly text editing and spreadsheets, the field has constantly diversified and outgrown all boundaries. It quickly expanded to encompass visualization, information systems, collaborative systems, the system development process, and many areas of design. HCI is taught now in many departments/faculties that address information technology, including psychology, design, communication studies, cognitive science, information science, science and technology studies, geographical sciences, management information systems, and industrial, manufacturing, and systems engineering. HCI research and practice draws upon and integrates all of these perspectives.

A result of this growth is that HCI is now less singularly focused with respect to core concepts and methods, problem areas and assumptions about infrastructures, applications, and types of users. Indeed, it no longer makes sense to regard HCI as a specialty of computer science; HCI has grown to be broader, larger and much more diverse than computer science. It expanded from individual and generic user behavior to include social and organizational computing, creativity, and accessibility for the elderly, the cognitively impaired, and for all people. It expanded from desktop office applications to include games, e-learning, e-commerce, military systems, and process control. It expanded from early graphical user interfaces to include myriad interaction techniques and devices, multi-modal interactions, and host of emerging ubiquitous, handheld and context-aware interactions.

There is no unified concept of an HCI professional. In the 1980s, people often contrasts the cognitive science side of HCI with the software tools and user interface side of HCI. The HCI landscape is far more differentiated and complex now. HCI academic programs train many different types of professionals now: user experience designers, interaction designers, user interface designers, application designers, usability engineers, user interface developers, application developers, technical communicators/online information designers, and more. And indeed, many of the sub-communities of HCI are themselves quite diverse. For example, ubiquitous computing (aka ubicomp) is subarea of HCI, but it is also a superordinate area integrating several mutually diverse subareas (e.g., mobile computing, geo-spatial information systems, in-vehicle systems, community informatics, distributed systems, handhelds, wearable devices, ambient intelligence, sensor networks, and specialized views of usability evaluation, programming tools and techniques, application infrastructures, etc.). The relationship between ubiquitous computing and HCI is becoming paradigmatic: HCI is the name for a community of communities.

Contd. ...

Notes

In the early 1980s, HCI was a small and focused specialty area. It was a cabal trying to establish what was then a heretical view of computing. Today, largely due to the success of that endeavor, HCI is a vast and multifaceted community, loosely bound by the evolving concept of usability, and the integrating commitment to value human concerns as the primary consideration in creating interactive systems.

Questions

1. Describe brief about Human-computer interaction (HCI).

2. What are the concepts of *usability* of Human-computer interaction (HCI)?

10.5 Summary

- A loop is the procedure which asks the microprocessor to change the sequence of execution and perform the task repeatedly.
- Counting is technique allows programmer to count how many times the instruction/set of instructions are executed.
- The unconditional jump instruction creates a continuous loop which can be stopped only by the resetting of the system.
- ADD and SUB instructions implicitly assume that one of the operand is register.

10.6 Keywords

Counting: Counting is technique allows programmer to count how many times the instruction/ set of instructions are executed.

Looping: Looping is the programming technique used to instruct the microprocessor to repeat tasks.

Machine code: Machine code or machine language is a system of instructions and data executed directly by a computer's central processing unit.

Registers: A register, among other definitions, is a record in writing. As a verb it is to record or to be recorded in an official list.

Rotate instructions: Rotate instructions rotate the Accumulator bits. This group includes four instructions.



Lab Exercise

1. Explain the machine code with example.
2. Give the code for looping instruction. Explain.

10.7 Self-Assessment Questions

1. Simple arithmetic operations describes the operations between number or contents of the microprocessor
- (a) instruction

(b) repeater

(c) register

(d) None of these
2. The rotate instruction is related to rotating the bits
- (a) accumulator

(b) register

(c) instruction

(d) None of these

3. Looping is the programming technique used to instruct the microprocessor to tasks. Notes
- (a) continue (b) repeat
(c) stop (d) None of these
4. A conditional loop is setup by the conditional instructions
- (a) jump (b) break
(c) end (d) None of these
5. is defined as a pointer or reference to an object with sequential numbers.
- (a) Looping (b) Repeating
(c) Indexing (d) None of these

10.8 Review Questions

1. What do you mean by counting and indexing?
2. Explain looping with the help of example.
3. Explain the need of software timers.
4. Describe different types of loops.
5. Distinguish between MVI and LXI.
6. What are arithmetic operations?
7. Explain the features of arithmetic operations related to 16-bit.
8. What are logical operations?
9. How compare operation works?
10. How a data can be transfer between memory and microprocessor?

Answers for Self-Assessment Questions

1. (c) 2. (a) 3. (b) 4. (a) 5. (c)

10.9 Further Reading



Books

Microprocessor architecture, programming, and systems featuring the 8085, William A. Routt



Online link

<http://books.google.com/books?>

Unit 11: Counters and Time Delays

CONTENTS

Objectives

Introduction

11.1 Program Counter

11.1.1 Updating the PC

11.2 Branching

11.2.1 Non-Offset Branching

11.2.2 Offset Branching

11.2.3 Offset and Non-Offset Branching

11.3 Time Delays

11.4 Programmed Time Delays

11.5 Using a Register Pair as a Loop Counter

11.5.1 Nested Loops

11.5.2 Nested Loops for Delay

11.5.3 Delay Calculation of Nested Loops

11.5.4 Increasing the Delay

11.6 Programs using If Loops Counters Accumulators

11.7 Hexadecimal Counter

11.8 Summary

11.9 Keywords

11.10 Self-Assessment Questions

11.11 Review Questions

11.12 Further Reading

Objectives

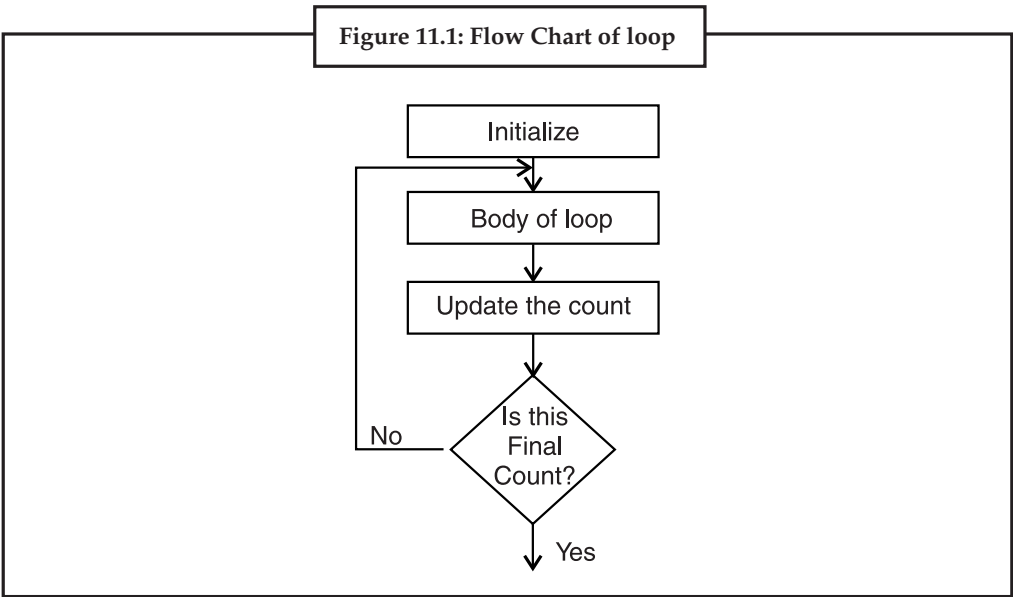
After studying this unit, you will be able to understand the following:

- Explain program counter
- Understand the concept of branching
- Describe time delays
- Explain programmed time delays
- Understand register pair as a loop counter
- Explain programs using if loops counters accumulators
- Describe hexadecimal counter

Introduction

Notes

- A loop counter is set up by loading a register with a certain value
- Then using the DCR (to decrement) and INR (to increment) the contents of the register are updated.
- A loop is set up with a conditional jump instruction that loops back or not depending on whether the count has reached the termination count.
- The operation of a loop counter can be described using the following flowchart.



11.1 Program Counter

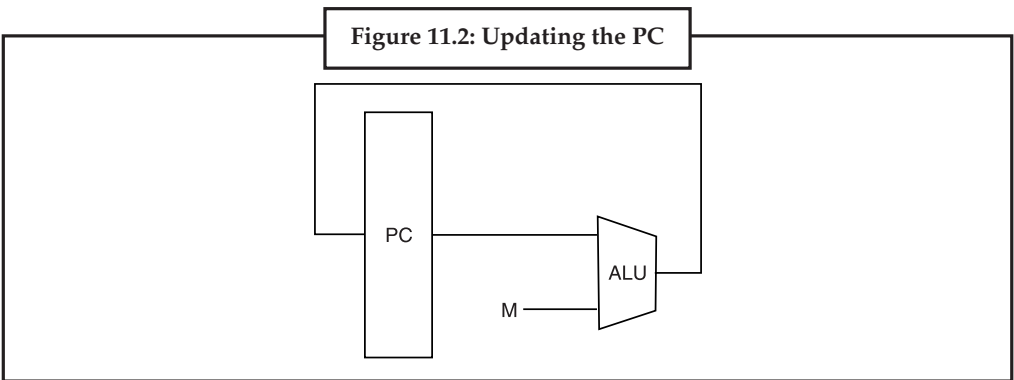
The **Program Counter** (PC) is a register structure that contains the address pointer value of the current instruction. Each cycle, the value at the pointer is read into the instruction decoder and the program counter is updated to point to the next instruction. For RISC computers updating the PC register is as simple as adding the machine word length (in bytes) to the PC. In a CISC machine, however, the length of the current instruction needs to be calculated, and that length value needs to be added to the PC.

11.1.1 Updating the PC


The PC can be updated by making the enable signal high. Each instruction cycle the PC needs to be updated to point to the next instruction in memory. It is important to know how the memory is arranged before constructing your PC update circuit.

Harvard-based systems tend to store one machine word per memory location. This means that every cycle the PC needs to be incremented by 1. Computers that share data and instruction memory together typically are *byte addressable*, which is to say that each byte has its own address, as opposed to each machine word having its own address. In these situations, the PC needs to be incremented by the number of bytes in the machine word.

Notes



In figure, the letter *M* is being used as the amount by which to update the PC each cycle. This might be a variable in the case of a CISC machine.

 *Example: MIPS*

The MIPS architecture uses a byte-addressable instruction memory unit. MIPS is a RISC computer, and that means that all the instructions are the same length: 32-bits. Every cycle, therefore, the PC needs to be incremented by 4 (32 bits = 4 bytes).

 *Example: Intel IA32*

The Intel IA32 (better known by some as “x86”) is a CISC architecture, which means that each instruction can be a different length. The Intel memory is byte-addressable. Each cycle the instruction decoder needs to determine the length of the instruction, in bytes, and it needs to output that value to the PC. The PC unit increments itself by the value received from the instruction decoder.

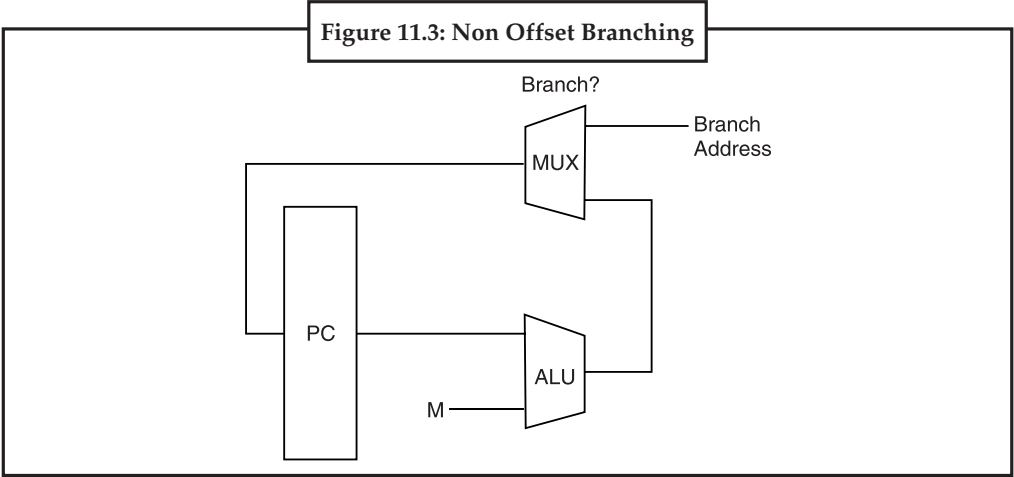
11.2 Branching

Branching occurs at one of a set of special instructions known collectively as “branch” or “jump” instructions. In a branch or a jump, control is moved to a different instruction at a different location in instruction memory.

During a branch, a new address for the PC is loaded, typically from the instruction or from a register. This new value is loaded into the PC, and future instructions are loaded from that location.

11.2.1 Non-Offset Branching

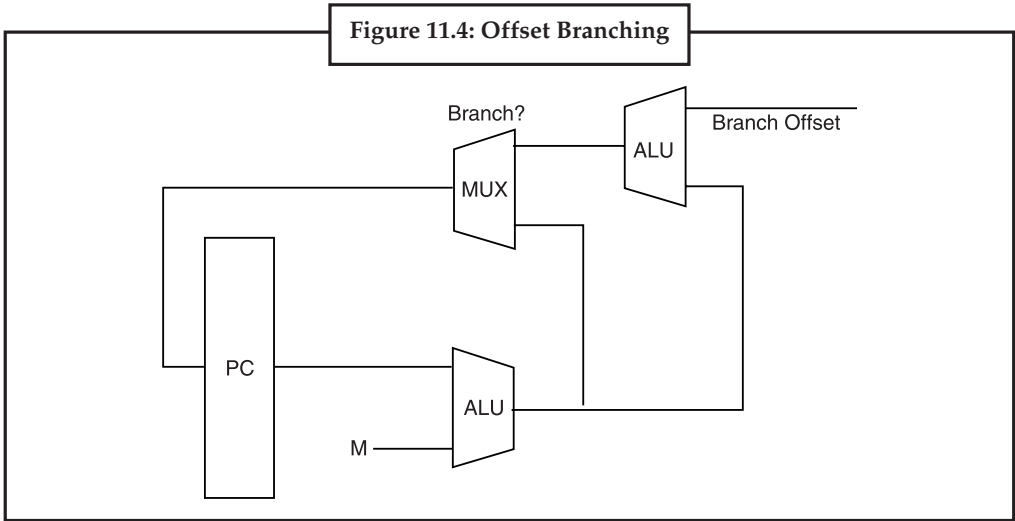
A non-offset branch, frequently referred to as a “jump” is a branch where the previous PC value is discarded and a new PC value is loaded from an external source.



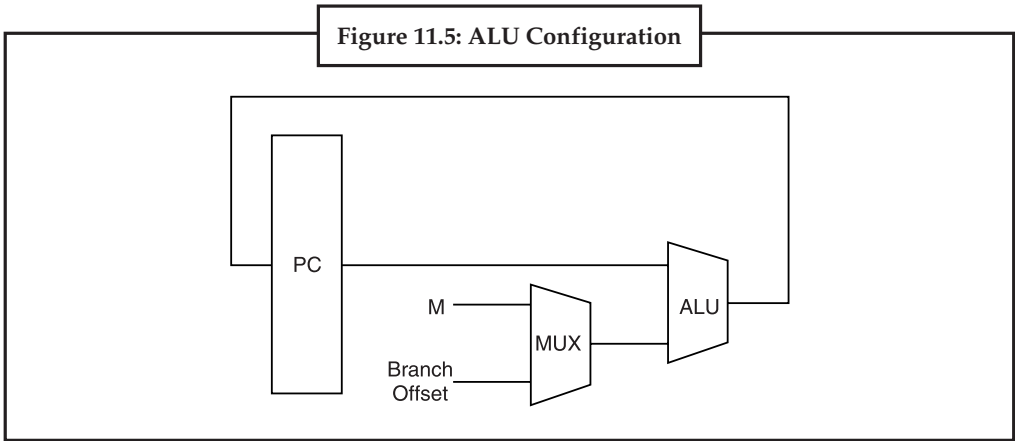
In figure, the PC value is either loaded with an updated version of itself, or else it is loaded with a new *Branch Address*. For simplification we do not show the control signals to the MUX.

11.2.2 Offset Branching

An offset branch is a branch where a value is added (or subtracted) to the current PC value to produce the new value. This is typically used in systems where the PC value is larger then a register value or an immediate value, and it is not possible to load a complete value into the PC.



In figure, there is a second ALU unit. Notice that we could simplify this circuit and remove the second ALU unit if we use the configuration mentioned in Figure 11.5:



These are just two possible configurations for this circuit.

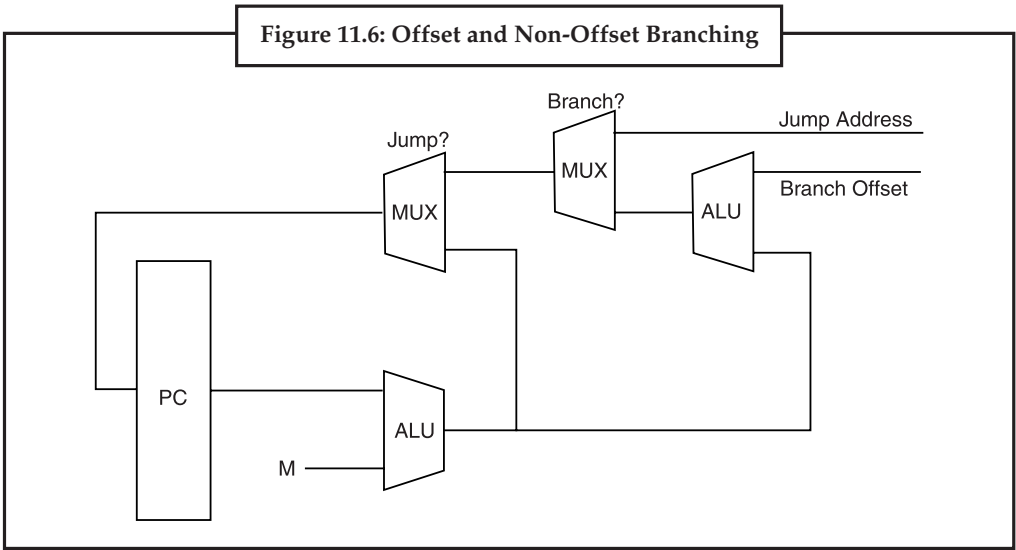


Offset branch is used to support reloadable binaries which may be loaded at an arbitrary base address.

11.2.3 Offset and Non-Offset Branching

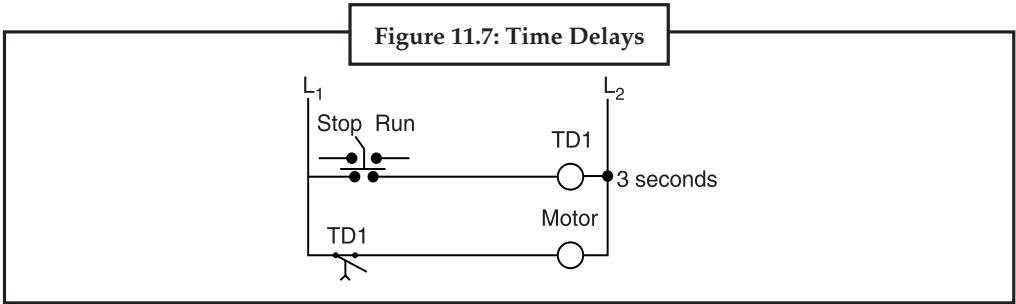
Many systems have capabilities to use both offset and non-offset branching. Some systems may differentiate between the two as “far jump” and “near jump”, respectively, although this terminology is archaic.

Notes



11.3 Time Delays

A special class of electromechanical relays called *time-delay* relays provide delayed action, either upon power-up or power-down, and are commonly denoted in ladder logic diagrams by “TD” or “TR” designations near the coil symbols and arrows on the contact symbols. Here is an example of a time-delay relay contact used in a motor control circuit:



- Knowing the combinations of cycles, one can calculate how long such an instruction would require to complete.
- Knowing how many T-States an instruction requires, and keeping in mind that a T-State is one clock cycle long, we can calculate the time using the following formula:

Delay = No. of T-States / Frequency

For example a “MVI” instruction uses 7 T-States.

Therefore, if the Microprocessor is running at 2 MHz, the instruction would require 3.5 mSeconds to complete.

- We can use a loop to produce a certain amount of time delay in a program.
- The following is an example of a delay loop:

```
MVI C, FFH    7 T-States
LOOP          DCR C    4 T-States
JNZ LOOP      10 T-States
```

- The first instruction initializes the loop counter and is executed only once requiring only 7 T-States.

The following two instructions form a loop that requires 14 T-States to execute and is repeated 255 times until C becomes 0

- We need to keep in mind though that in the last iteration of the loop, the JNZ instruction will fail and require only 7 T-States rather than the 10.
- Therefore, we must deduct 3 T-States from the total delay to get an accurate delay calculation.
- To calculate the delay, we use the following formula:

$$T_{\text{delay}} = T_O + T_L$$

- T_{delay} = total delay
- T_O = delay outside the loop
- T_L = delay of the loop

- T_O is the sum of all delays outside the loop.
- T_L is calculated using the formula

$$T_L = T \times \text{Loop T-States} \times N_{10}$$

- Using these formulas, we can calculate the time delay for the previous example:
 - $T_O = 7$ T-States
 - Delay of the MVI instruction
- $T_L = (14 \times 255) - 3 = 3567$ T-States
 - 14 T-States for the 2 instructions repeated 255 times ($FF_{16} = 255_{10}$) reduced by the 3 T-States for the final JNZ.
- $T_{\text{Delay}} = (7 + 3567) \times 0.5 \text{ mSec} = 1.787 \text{ mSec}$
 - Assuming $f = 2 \text{ MHz}$

11.4 Programmed Time Delays

Each 8085 instruction requires a specific amount of execution time. In most applications, the programmer will attempt to minimize this execution time by writing the program in an efficient manner. There are many situations however, in which desirable to generate a time delay. This delay time can serve as a time to generate periodic waveforms or to sequence an industrial process. The execution time required by each instruction is function of the number of (T-states) in its instruction cycle. For example, if an 8085 microprocessor has a clock frequency of 2 MHz, then each T-state equal 0.5 μs . assuming this clock frequency, the instruction MVI A, byte, which consumes seven T-states, would be executed in.



Each T-state is equal to one period of the 8085 system clock. Clock frequency of the system $f = 2\text{MHz}$



Did u know?

Compute the same calculation when clock frequency=3.125MHz, then each T-state equal 320ns. The time delay generated by a program is compounded by nesting loops within other loops. The total delay will then be the product of the individual delay for each loop. If longer loops are required, register pairs may be used for loop counters. Statements such as XTHL are often placed inside delay loops as “padding”, to increase the total delay time.

Notes

Useful Instructions: NOP: No operation is performed. The instruction is fetched and decoded; however, no operation is executed. The instruction is used to fill in time delays and insert instructions while troubleshooting.

Instruction	Type	No. of Bytes	Function	Effect
NOP	Machine control	1	No operation	None



Example:

		T-states
Loop:	MVI C, 0FFh	7
	DCR C	5
	JNZ loop	10

In this example, register C is loaded with the count ff (255) by the instruction MVI, which is executed once and takes seven T-states. The next two instructions, DCR and JNZ, from a loop with a total of fifteen (5+10) T-states. The loop is repeated 255 times until C=0. The time delay in the loop TL with 2 MHZ clock frequency is calculated as:-

$$TL = (T^* \text{ loop } T\text{-states} * N10)$$

Where TL : Time delay in the loop

T: system clock period

N10: equivalent decimal number of the hexadecimal count loaded in the delay register.

$$= (0.5 \cdot 10^{-6} \cdot 15 \cdot 255)$$

$$= 1912.5 \mu s$$

However, to calculate the time delay more accurately, the time for the execution of the initial instruction MVI should be included in the total time delay TD as:

Delay TD = Time to execute instructions outside loop + time to execute loop instructions

$$= T0+ TL$$

$$= (7 \times 0.5 \mu s) + 1912.5 \mu s$$

$$= 1916\mu\text{s}$$



Did u know?

INX rp and DCX rp instruction do not modify the Z flag. In other words, a JZ or JNZ instruction immediately following a DCX rp or INX rp instruction is not satisfactory, and an arithmetic or logic operation must be executed first

11.5 Using a Register Pair as a Loop Counter

- Using a single register, one can repeat a loop for a maximum count of 255 times.
- It is possible to increase this count by using a register pair for the loop counter instead of the single register.
 - A minor problem arises in how to test for the final count since DCX and INX do not modify the flags.
- The following is an example of a delay loop set up with a register pair as the loop counter.

LXI B, 1000H

10 T-States

Notes

LOOP DCX B	6 T-States
MOV A, C	4 T-States
ORA B	4 T-States
JNZ LOOP	10 T-States

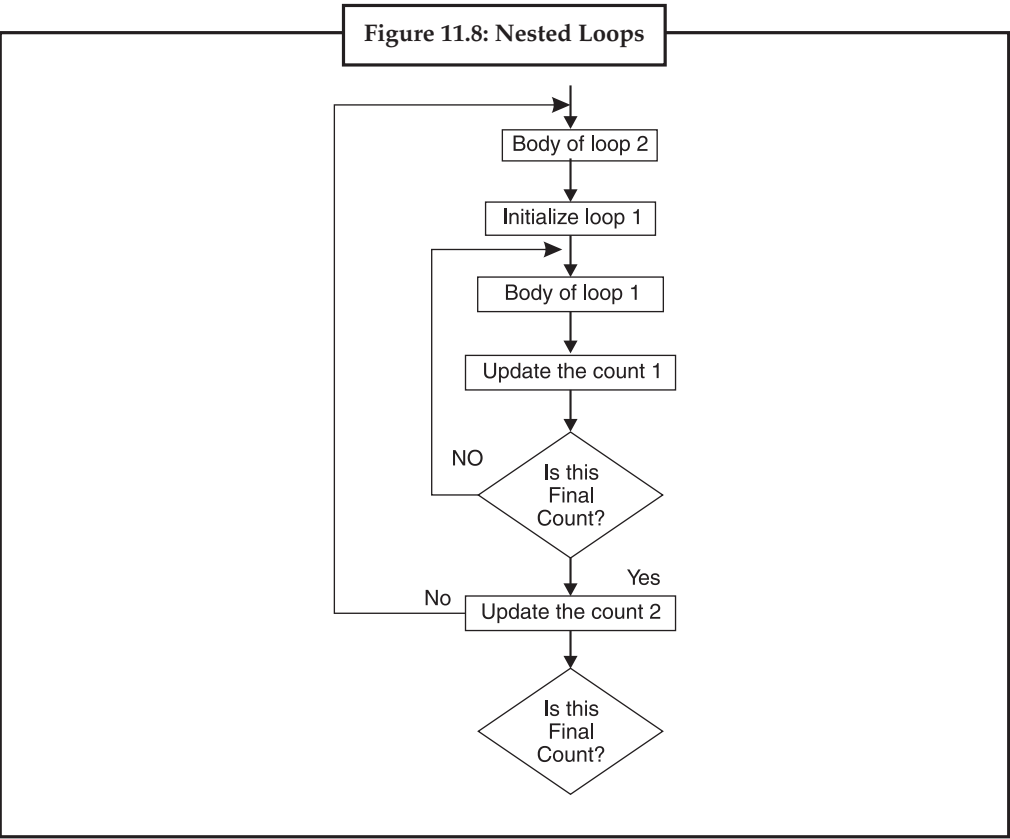
- Using the same formula from before, we can calculate:
- $T_O = 10$ T-States
 - The delay for the LXI instruction
- $T_L = (24 \times 4096) - 3 = 98301$ T- States
 - 24 T-States for the 4 instructions in the loop repeated 4096 times ($1000_{16} = 4096_{10}$) reduced by the 3 T-States for the JNZ in the last iteration.
- $T_{Delay} = (10 + 98301) \times 0.5 \text{ mSec} = 49.155 \text{ mSec}$



If the loop is looking for when the count becomes zero, we can use a small trick by ORing the two registers in the pair and then checking the zero flag.

11.5.1 Nested Loops

- Nested loops can be easily setup in Assembly language by using two registers for the two loop counters and updating the right register in the right loop.
 - In the Figure, the body of loop2 can be before or after loop1.



Notes

11.5.2 Nested Loops for Delay

- Instead (or in conjunction with) Register Pairs, a nested loop structure can be used to increase the total delay produced.

MVI B, 10H	7	T-States
LOOP2 MVI C, FFH	7	T-States
LOOP1 DCR C	4	T-States
JNZ LOOP1	10	T-States
DCR B	4	T-States
JNZ LOOP2	10	T-States

11.5.3 Delay Calculation of Nested Loops

- The calculation remains the same except that it the formula must be applied recursively to each loop.
 - Start with the inner loop, then plug that delay in the calculation of the outer loop.
- Delay of inner loop
 - $T_{O1} = 7$ T-States
- MVI C, FFH instruction
 - $T_{L1} = (255 \times 14) - 3 = 3567$ T-States
- 14 T-States for the DCR C and JNZ instructions repeated 255 times ($FF_{16} = 255_{10}$) minus 3 for the final JNZ.
 - $T_{LOOP1} = 7 + 3567 = 3574$ T-States
- Delay of outer loop
 - $T_{O2} = 7$ T-States
- MVI B, 10H instruction
 - $T_{L1} = (16 \times (14 + 3574)) - 3 = 57405$ T-States
- 14 T-States for the DCR B and JNZ instructions and 3574 T-States for loop1 repeated 16 times ($10_{16} = 16_{10}$) minus 3 for the final JNZ.
 - $T_{Delay} = 7 + 57405 = 57412$ T-States
- Total Delay
 - $T_{Delay} = 57412 \times 0.5 \text{ mSec} = 28.706 \text{ mSec}$



Search more about the looping counter.

11.5.4 Increasing the Delay

The delay can be further increased by using register pairs for each of the loop counters in the nested loops setup.

It can also be increased by adding dummy instructions (like NOP) in the body of the loop.

11.6 Programs Using If Loops Counters Accumulators

Notes

Program to print possible sum combinations of a number

```
class NumberCombinations
{
    void makeCombinations(int n)
    {
        int sum,x,y,z,p;
        for(x=1 ; x<=9 ; x++)
        {
            for(y=x ; y<=9 ; y++)
            {
                sum =x+y;
                if(sum==n)
                    System.out.println(x + " + " + y);
                for(z=1 ; z<=9 ; z++)
                {
                    sum=sum+z;
                    if(sum<n)
                        continue;
                    else
                        break;
                }
                //for ends
                if(sum==n)
                {
                    System.out.print(x + " + " + y);
                    for(p=1 ; p<=z ; p++)
                        System.out.print(" + " + p);
                    System.out.println();
                }
            }
        }
    }
}
//if ends
    }
//for ends
}
//for ends
    }
//method ends
}
//class ends
```

11.7 Hexadecimal Counter

Hexadecimal means 16 and we are using what is called 'Base 16' when we count in Hex (Hexadecimal). Base 16 means that each column can hold up to 16 numbers before it has to be zeroed out, and 1 added to the next column to the left. We can put any one of 16 numbers in Column 1 (zero and 1-9 and A-F).

When we are counting and we reach F (the equivalent of 15 in decimal) in Column 1, Column 1 is full and we zero it out. We then add 1 to Column 2 and a put a 0 in Column 1.

If we keep on counting in this manner, adding 1 to col 2 every time col 1 reaches F, we fill up cols 1 and 2 to a value of FF.

Notes

Adding 1 more means we have to zero out col 1 and add 1 to col 2. However, this makes col 2 = F+1. So we have also to zero out col 2 and add 1 to col 3



Case Study

Performance Analysis and Monitoring Using Hardware Counters

The UltraSPARC and Pentium microprocessors contain hardware performance counters that allow counting a series of processor events, such as cache misses, pipeline stalls and floating-point operations. Statistics of processor events can be collected in hardware with little or no overhead, making these counters a powerful means to monitor an application and analyze its performance. Such counters, to their advantage, are non-intrusive, do not require recompilation of applications and are available in every UltraSPARC-based system that Sun Microsystems currently ships. Nonetheless, these counters are not widely used beyond hardware specialists because of the lack of programming interfaces and sparse documentation.

Starting with release 8 of the Solaris OE, there is an increasing support of hardware counters, with the availability of utility tools and public programming interfaces. This support for hardware counters at the operating system level is a major milestone for the Solaris software developer community but is not a direct answer to its needs. Application programmers work at a higher level where the common metrics are a combination of the low-level event counts accumulated in hardware. There is a need for higher-level tools that hide the complexity of the underlying hardware design and present in a uniform manner the common metrics the programmers expect: million instructions per second (MIPS), floating-point operations per second (FLOPS), cache miss rate.

In this study, the Solaris 8 platform's support for hardware performance counters is covered, and a tool, the Hardware Activity Reporter that builds on the new Solaris interfaces and aims to address the needs of application programmers is introduced.

Solaris 8 OE Support for Hardware Performance Counters

In February 2000, Sun Microsystems announced a new release of the Solaris Operating Environment, Solaris 8. With the latest release of Solaris 8 OE, Sun began to deliver public interfaces to the UltraSPARC and Pentium hardware performance counters. A series of application programming interfaces (APIs) have been made available as shared libraries, to program various hardware counters and the utility tools `cpustat` and `cputrack` to access the microprocessor (CPU) hardware counters from the command line. The Solaris 8 platform also provides similar libraries and tools to access the hardware counters of the system bus and I/O boards.

Solaris 8 OE Libraries

Solaris 8 software ships with two libraries directly related to the usage of the CPU performance counters: `libcpc`, to access CPU counters and `libpctx`, to track a process. Using these APIs, one can instrument code to access the performance hardware counters and collect performance information. The steps to instrument a piece of code are:

1. Check the versions and accessibility of the hardware performance counters with `cpc_version()` and `cpc_access()`;
2. Initialize a `cpc_event_t` data structure, using `cpc_getcpuver()` and `cpc_strtoevent()`, to be used in conjunction with the counters;
3. Bind the data structure to the CPU using `cpc_bind_event()`;

Contd...

Notes

- 4. Read the counters, as desired, using `cpc_take_sample()`;
- 5. Release the CPU using `cpc_rele()` when finished.

The attractive aspect of these interfaces is their simplicity. Once the counters are initialized, a program reads them by a single call to `cpc_take_sample()`. No matter how easy-to-use these APIs are, however, people are usually reluctant to instrument code. To avoid incrementing code, Solaris 8 software also ships with a couple of command-line based utilities, `cpustat` and `cpustrack`, that report on CPU performance counters.

Solaris 8 OE Utility Tools

The `cpustat` utility reports on CPU performance counters in a system-wide fashion. `cpustat` is invoked from the command line, and a pair of processor events to monitor must be passed as an argument. Events go by pair because the current UltraSPARC and Pentium microprocessors have two hardware performance counters that must be programmed simultaneously. Optional arguments are the sampling interval and count, i.e., the frequency at which the counters are read and the number of times the counters are read. Multiple pairs of events can be specified; in that case, the system alternates between the multiple pairs. Because `cpustat` is a system-wide utility, it must be run as root.

Questions:

- 1. What you understand to UltraSPARC and Pentium microprocessors?
- 2. Differentiate between MIPS and FLOPS.

11.8 Summary

- A loop is set up with a conditional jump instruction that loops back or not depending on whether the count has reached the termination count.
- PC value is either loaded with an updated version of itself, or else it is loaded with a new *Branch Address*.
- Delay time can serve as a time to generate periodic waveforms or to sequence an industrial process.
- Calculation remains the same except that it the formula must be applied recursively to each loop.

11.9 Keywords

Jump: Branching occurs at one of a set of special instructions known collectively as jump.

Offset branch: It is a branch where a value is added to the current PC value to produce the new value.

Program counter (PC) is a register structure that contains the address pointer value of the current instruction.

Time-delay: A special class of electromechanical relays called *time-delay*.



The following program uses register pair (BC) to control the time delay length.

- 1. Explain why the program as written does not perform properly. Correct the program so that it functions as required.

LXI B,0FFFF h : Initialize loop counter.

Loop: DCX B : Decrement loop counter.

Notes

JNZ Loop : Loop until reaches zero.

RST1 : Stop.

2. Calculate the total time required to execute the following delay subroutine.

Delay: LXI D,02F1h

Loop: DCX D

MOV A,E

ORA D JNZ

Loop RST1

11.10 Self-Assessment Questions

1. PC can be updated by making the enable signal high.
(a) True (b) False
2. set up by loading a register with a certain value.
(a) INR (b) DCR
(c) Loop counter (d) None of the above
3. Harvard-based systems tend to store machine word per memory location.
(a) one (b) two
(c) three (d) None of the above
4. The first instruction initializes the loop counter and is executed only once requiring only 7 T-States.
(a) True (b) False
5. The required by each instruction is function of the number of T-states in its instruction cycle.
(a) installation time (b) execution time
(c) process time (d) None of the above
6. Single register can repeat a loop for a maximum count of 205 times.
(a) True (b) False

11.11 Review Questions

1. What do you mean by loop counter?
2. Define PC.
3. Explain branching. and name its types.
4. Explain programmed time delays.
5. Define time delays.
6. Describe the steps to use register as a loop counter with example.
7. What are nested loops?
8. What do you mean by hexadecimal?

9. Write the difference between offset and non-offset branching.
10. Explain an example of hexadecimal counter.

Notes

Answers for Self-Assessment Questions

1. (a)
2. (c)
3. (a)
4. (a)
5. (b)
6. (b)

11.12 Further Reading



Books

Microprocessor architecture, Jean-Loup Baer



Online link

<http://books.google.com/books?>

Unit 12: The Stacks

CONTENTS

Objectives

Introduction

12.1 Stack

12.1.1 Stack in main memory

12.1.2 Stack in registers or dedicated memory

12.2 Saving Information on the Stack

12.3 PUSH & POP

12.3.1 The PUSH Instruction

12.3.2 The POP Instruction

12.3.3 Operation of the Stack

12.4 The PSW Register Pair

12.4.1 PUSH PSW (1 Byte Instruction)

12.4.2 Pop PSW Register Pair

12.4.3 Modify Flag Content using PUSH/POP

12.5 Summary

12.6 Keywords

12.7 Self-Assessment Questions

12.8 Review Questions

12.9 Further Reading

Objectives

After studying this unit, you will be able to understand the following:

- Discuss about the use of stack in memory
- Explain about how to save the information on stack
- Discuss about the operations on stack

Introduction

The stack is one of the most important things you must know when programming. Think of the stack as a deck of cards. When you put a card on the deck, it will be the top card. Then you put another card, then another. When you remove the cards, you remove them backwards, the last card first and so on. The stack works the same way, you put (push) words (addresses or register pairs) on the stack and then remove (pop) them backwards. That's called LIFO, Last In First Out.

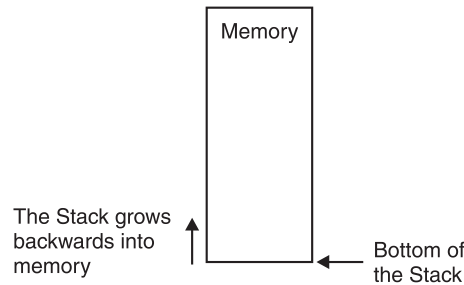
The 8085 uses a 16 bit register to know where the stack top is located, and that register is called the SP (Stack Pointer). There are instructions that allow you to modify its contents but you should NOT change the contents of that register if you don't know what you're doing?

12.1 Stack

Notes

The stack is an area of memory identified by the programmer for temporary storage of information. The stack is a LIFO structure (Last In First Out). The stack normally grows backwards into memory.

In other words, the programmer defines the bottom of the stack and the stack grows up into reducing address range.



Given that the stack grows backwards into memory, it is customary to place the bottom of the stack at the end of memory to keep it as far away from user programs as possible.

In the 8085, the stack is defined by setting the SP (Stack Pointer) register.

LXI SP, FFFFH

This sets the Stack Pointer to location FFFFH (end of memory for the 8085). The Size of the stack is limited only by the available memory



The stack was first proposed in 1955, and then patented in 1957, by the German Friedrich L. Bauer. The same concept was developed independently, at around the same time, by the Australian Charles Leonard Hamblin.

12.1.1 Stack in main memory

Most CPUs have registers that can be used as stack pointers. Processor families like the x86, Z80, 6502, and many others have special instructions that implicitly use a dedicated (hardware) stack pointer to conserve opcode space. Some processors, like the PDP-11 and the 68000, also have special addressing modes for implementation of stacks, typically with a semi-dedicated stack pointer as well (such as A7 in the 68000). However, in most processors, several different registers may be used as additional stack pointers as needed (whether updated via addressing modes or via add/sub instructions).

12.1.2 Stack in registers or dedicated memory

The x87 floating point architecture is an example of a set of registers organized as a stack where direct access to individual registers (relative the current top) is also possible. As with stack-based machines in general, having the top-of-stack as an implicit argument allows for a small machine code footprint with a good usage of bus bandwidth and code caches, but it also prevents some types of optimizations possible on processors permitting random access to the register file for all (two or three) operands. A stack structure also makes superscalar implementations with register renaming (for speculative execution) somewhat more complex to implement, although it is still feasible, as exemplified by modern x87 implementations.

Sun SPARC, AMD Am29000, and Intel i960 are all examples of architectures using register windows within a register-stack as another strategy to avoid the use of slow main memory for function arguments and return values.

Notes

There are also a number of small microprocessors that implements a stack directly in hardware and some microcontrollers have a fixed-depth stack that is not directly accessible. Examples are the PIC microcontrollers, the Computer Cowboys MuP21, the Harris RTX line, and the Novix NC4016. Many stack-based microprocessors were used to implement the programming language Forth at the microcode level. Stacks were also used as a basis of a number of mainframes and mini computers. Such machines were called stack machines, the most famous being the Burroughs B5000.



Search which stack was used in 8085 microprocessor.

12.2 Saving Information on the Stack

Information is saved on the stack by PUSHing it on. It is retrieved from the stack by POPing it off. The 8085 provides two instructions: PUSH and POP for storing information on the stack and retrieving it back.



Processor families like the x86, Z80, 6502, and many others have special instructions that implicitly use a dedicated (hardware) stack pointer to conserve opcode space.

12.3 PUSH & POP

As you may have guessed, push and pop "pushes" bytes on the stack and then takes them off. When you push something, the stack counter will decrease with 2 (the stack "grows" down, from higher addresses to lower) and then the register pair is loaded onto the stack. When you pop, the register pair is first lifted of the stack, and then SP increases by 2.

N.B: Push and Pop only operate on words (2 bytes i.e.: 16 bits).

You can push (and pop) all register pairs: BC, DE, HL and PSW (Register A and Flags). When you pop PSW, remember that all flags may be changed. You can't push an immediate value. If you want, you'll have to load a register pair with the value and then push it. Perhaps it's worth noting that when you push something, the contents of the registers will still be the same; they won't be erased or something. Also, if you push DE, you can pop it back as HL (you don't have to pop it back to the same register where you got it from).

The stack is also updated when you CALL and RETURN from subroutines. The PC (program counter which points at the next instruction to be executed) is pushed to the stack and the calling address is loaded into PC. When returning, the PC is loaded with the word popped from the top of the stack (TOS). So, when is this useful? It's almost always used when you call subroutines.

For example, you have an often used value stored in HL. You have to call a subroutine that you know will destroy HL. Instead of first saving HL in a memory location and then loading it back after the subroutine, you can push HL before calling and directly after the calling pop it back. Of course, it's often better to use the pushes and pops inside the subroutine. All registers you know will be changed are often pushed in the beginning of a subroutine and then popped at the end, in reverse order! Don't forget - last in first out. If you want to only push one 8 bit register, you still have to push its "friend". Therefore, be aware that if you want to store away D with pushing and popping, remember that E will also be changed back to what it was before. In those cases, if you don't want that to happen, you should try first to change register (try to store the information in E in another register if you can) or else you have to store it in a temporary variable.

Before executing a program, you should keep track of your pushes and pops, since they are responsible for 99% of all computer crashes! For example, if you push HL and then forget to pop

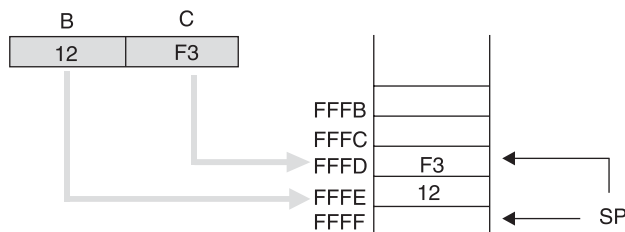
Notes

it back, the next RET instruction will cause a jump to HL, which can be anywhere in the ROM/RAM and the computer will crash. Note however, it's also a way to jump to the location stored in HL, but then you should really use the JMP instruction, to do the same thing. Push and pop doesn't change any flags, so you can use them between a compare and jump instructions, depending on a condition, which is often very useful.

12.3.1 The PUSH Instruction

PUSH B (1 Byte Instruction)

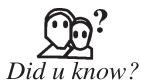
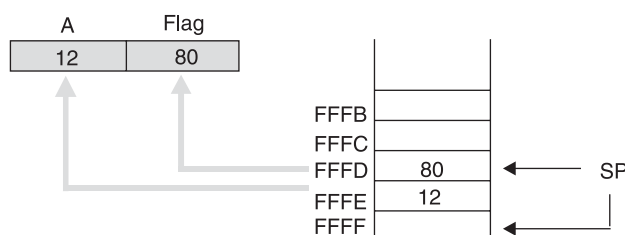
- Decrement SP
- Copy the contents of register B to the memory location pointed to by SP
- Decrement SP
- Copy the contents of register C to the memory location pointed to by SP



12.3.2 The POP Instruction

POP D (1 Byte Instruction)

- Copy the contents of the memory location pointed to by the SP to register E
- Increment SP
- Copy the contents of the memory location pointed to by the SP to register D
- Increment SP



The SP pointer always points to "the top of the stack".

12.3.3 Operation of the Stack

During pushing, the stack operates in a "decrement then store" style. The stack pointer is decremented first, then the information is placed on the stack. During popping, the stack operates in a "use then increment" style. The information is retrieved from the top of the stack and then the pointer is incremented.

Notes

LIFO

The order of PUSHs and POPs must be opposite of each other in order to retrieve information back into its original location.

PUSH B

PUSH D

...

POP D

POP B

Reversing the order of the POP instructions will result in the exchange of the contents of BC and DE.



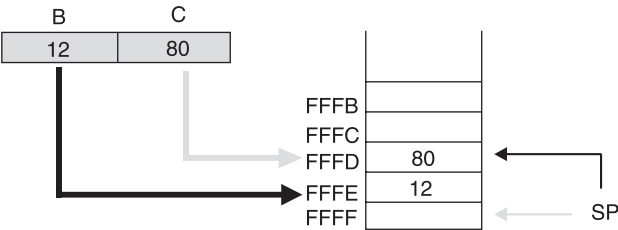
The top of the stack is also known as peek.

12.4 The PSW Register Pair

The 8085 recognizes one additional register pair called the PSW (Program Status Word). This register pair is made up of the Accumulator and the Flag registers. It is possible to push the PSW onto the stack, do whatever operations are needed, then POP it off of the stack. The result is that the contents of the Accumulator and the status of the Flags are returned to what they were before the operations were executed.

12.4.1 PUSH PSW (1 Byte Instruction)

- Decrement SP
- Copy the contents of register A to the memory location pointed to by SP
- Decrement SP
- Copy the contents of Flag register to the memory location pointed to by SP



12.4.2 POP PSW Register Pair

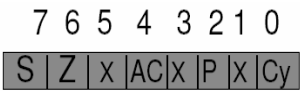
POP PSW (1 Byte Instruction)

- Copy the contents of the memory location pointed to by the SP to Flag register
- Increment SP
- Copy the contents of the memory location pointed to by the SP to register A
- Increment SP

12.4.3 Modify Flag Content using PUSH/POP

Notes

- Let, We want to Reset the Zero Flag
- 8085 Flag :



- Program:
- LXI SP FFFF
- PUSH PSW
- POP H
- MOV A L
- ANI BFH (BFH= 1011 1111) * Masking
- MOV L A
- PUSH H
- POP PSW



Give the basic concept of stack implementation.



Example: Write a program to clear all the flags and load OOH in the accumulator (A) and demonstrate that the zero flag is not affected by the data transfer instruction.

Solution: In this problem we have to examine the zero flag after instructions have been executed. There is no direct method of observing these flags and so they are stored on the stack using the PUSH PSW. The contents of the flag register can be retrieved in any one of the registers by using the instruction POP, and the flags can be displayed at an output port. The assembly language program for thee above stated problem is as follows:

Machine Code (Hex.)	Opcode	Operands	Comments
31	LXI	SP, XX99H	; Initialize the stack
99			
XX			
2E	MVI	L, OOH	; Clear L
00			
E5	PUSH	H	; Place (L) on stack
F1	POP	PSW	; Clear flags
3E	MVI	A, OOH	; Load OOH
00			
F5	PUSH	PSW	; Save flags on stack
E1	POP	H	; Retrieve flags in L
7D	MOV	A, L	
D3	OUT	PORT 0	; Display flags

Notes

PORTO

The stack pointer register is initialized at XX99H. All the flags are cleared then (L) is placed on the stack, which is subsequently placed into the flag register to clear all the flags. The flags are verified by PUSH and POP instructions after the execution of MVI A instructions. These instructions are used to clear the flags and the flags are displayed at PORTO.

Data transfer instructions do not affect the flags and so no flags should be set after the instructions MVI A, even if (A) is equal to zero and PORTO should display OOH. If the output ports are not available, the results can be stored in the stack memory.

Instruction	Stack Memory Contents
	XX99 Stack painter initialization
PUSH PSW	XX98 (A) = OOH
	XX97 (F) = OOH



Case Study

Automated Stacking System

Challenge An international building products company that provides support to general contractors, architects, engineers and governmental agencies needed an automated way to stack flat and tapered boards to both increase speed and alleviate large turnover in manpower.

Before working with Motion Controls Robotics, the company had no automation. Employees manually lifted and stacked the boards, often leading to injuries. The company also had a high employee turnover rate and used some temporary staff. In addition, the stacks created manually were often uneven, making it difficult to keep the products from leaning.



Solution —Motion Controls Robotics recommended a system using a robot with a vacuum-powered end-of-arm tool to position, square and pick up the boards from the existing equipment. The robot lifts the board and sets it onto a chain conveyor, rotating every other board. At a pre-determined count, the stacks are sent out to handling conveyors that move them to down-line equipment.

Motion Controls Robotics installed, tested and provided training and support during the startup to familiarize the associates on how to operate the equipment. Motion Controls Robotics technicians directly supervised start-up, eventually turning over operation to the newly trained employees.

Details -- The system used two Fanuc M-410iB robots, HandlingTool software and custom written software to handle the robot motion. Peripheral equipment included an Allen Bradley

505 plc, and Panel View Plus1000 HMI.

Result -- According to the company, this system allowed for three fewer employees per shift. By decreasing the number of employees, the company reduced manpower turnover and injuries. Straighter and neater stacks could not be achieved using the original manual system. With the automated system the robots can stack precisely and also rotate tapered boards every other part which allows for more even and straighter stacks.

Questions:

1. Explain the basic challenges which occur in automated stacking system.
2. How automated stacking systems overcome the manpower problems?

Notes

12.5 Summary

- A stack is a restricted data structure, because only a small number of operations are performed on it.
- Elements are removed from the stack in the reverse order to the order of their addition: therefore, the lower elements are those that have been on the stack the longest.
- If the stack is full and does not contain enough space to accept the given item, the stack is then considered to be in an overflow state.
- If the stack is empty then it goes into underflow state.

12.6 Keywords

LIFO: Last In First Out, this is the way to implement the stack. If a data insert first in stack it will remove at the last position.

POP: Pop is an operation to perform the deletion on the stack.

PSW: It stands for Program Status Word. It is an additional register pair in 8085, is made up of the Accumulator and the Flag registers.

PUSH: Push is an operation to insert the data into the stack.

SP: The 8085 uses a 16 bit register to know where the stack top is located, and that register is called the SP (Stack Pointer).



Lab Exercise

1. Write the steps to insert the data into the stack.
2. Write an assembly program to add two numbers.

12.7 Self-Assessment Questions

1. The stack is an area of memory identified by the programmer for temporary storage of information.
 - (a) True
 - (b) False
2. The stack is a Structure.
 - (a) Last In First Out
 - (b) First In Last Out
 - (c) both
 - (d) none

Notes

3. operation perform the insertion on stack.
- (a) POP (b) PUSH
- (c) INSERT (d) None
4. operation perform the deletion on stack.
- (a) POP (b) PUSH
- (c) INSERT (d) None
5. The 8085 uses a 16 bit register to know where the stack top is located, and that register is called the.....
- (a) TOP (b) Overflow
- (c) INSERT (d) Stack Pointer
6. The condition on which no any data can be insert in stack is called.....
- (a) underflow (b) overflow
- (c) insert (d) stack Pointer
7. The condition on which no any data can be delete from stack is called.....
- (a) underflow (b) overflow
- (c) insert (d) stack Pointer
8. Stacks were also used as a basis of a number of mainframes and mini computers.
- (a) True (b) False

12.8 Review Questions

1. What is Stack Pointer?
2. Which Stack is used in 8085?
3. Read the following program and answer the questions:
- 4000 LXI SP, 4100H DELAY : 4064 PUSH H

4003 LXI B, 0000H 4065 PUSH B

4006 PUSH B , 4066 LXI B, 80FFH

4007 POP PSW LOOP : 4069 DCX B

4008 LXI H, 400BH 406A MOV A, B

400B CALL 4064H 406B ORA C

400E OUT 01H 406C JNC LOOP

4010 HLT 406F POP B

4070 RET
- (a) What is the status of the flags and the contents of the Accumulator after the execution of the POP instruction located at 4007H?
- (b) Specify the stack locations and their contents after the execution of the CALL instruction (not the call subroutine)
- (c) What are the contents of the stack pointer register and the program counter?

Notes

- (d) Specify the memory location where the program returns after the subroutine.
- (e) What is the output of this program?
4. What do you mean by stacks? Why are they needed?
 5. What is the mean of PUSH and POP in stack?
 6. What do you understand by term stack in memory?

Answers for Self-Assessment Questions

- | | | | | |
|--------|--------|--------|--------|--------|
| 1. (a) | 2. (a) | 3. (b) | 4. (a) | 5. (d) |
| 6. (b) | 7. (a) | 8. (a) | | |

12.9 Further Reading

Books

8080A/8085 Assembly Language Programming, Lance A. Leventhal



Online link

<http://books.google.com/books?>

Unit 13: Subroutines

CONTENTS
Objectives
Introduction
13.1 Subroutine
13.1.1 Definition
13.1.2 Calling a Subroutine
13.1.3 Exiting a Subroutine
13.2 Restart Sequence
13.3 Conditional Call Instruction
13.3.1 Subroutine call
13.3.2 Trap subroutine call activation/deactivation
13.4 Return Instruction
13.5 Summary
13.6 Keywords
13.7 Self-Assessment Questions
13.8 Review Questions
13.9 Further Reading

Objectives

After studying this unit, you will able to understand the following:

- Explain the definition of subroutine
- Describe how to create a subroutine
- Explain the procedure to call a subroutine
- Describe about restart sequence
- Define conditional call instruction and its uses; and
- Explain return instruction

Introduction

A subroutine is a container that holds a series of VBScript statements. Suppose you'd like to create a block of code that accomplishes some specific task. Maybe you need to accomplish that task in various places throughout your code. All you need to do is create, or declare, the subroutine in your script. Once you've declared the subroutine, you can call it anywhere within your code. When your program calls a subroutine, the flow of the code is temporarily diverted to the statements within the subroutine.

Once the subroutine has finished executing, control returns to the code that called the subroutine and execution picks up from there. A subroutine is a block of code that can be called from anywhere in a program to accomplish a specific task. Subroutines can accept starting data through subroutine

declaration variables called parameters. However, subroutines do not automatically return a result code or an argument to the caller.

You declare subroutines using the Sub keyword and end them using the End Sub statement.

13.1 Subroutine

13.1.1 Definition

Subroutines are a powerful programming tool, and the syntax of many programming languages includes support for writing and using them. Judicious use of subroutines (for example, through the structured programming approach) will often substantially reduce the cost of developing and maintaining a large program, while increasing its quality and reliability. Subroutines, often collected into libraries, are an important mechanism for sharing and trading software. The discipline of object-oriented programming is based on objects and methods (which are subroutines attached to these objects or object classes).

The structure of a subroutine is where Subroutine Name is the name of the subroutine and argument through arguments are optional arguments, often called parameters that you can pass to the subroutine. If you choose not to pass any arguments to the subroutine, the parentheses are optional, as you will see in a moment. The name of a subroutine should adequately describe what the subroutine is for. You must name subroutines using the same rules as variables; letters and numbers are fine as long as the first character is not a number, and you cannot use symbols. If you enter an invalid name for a subroutine

The following are some valid subroutine names:

- WelcomeTheUser
- PrintInvoice
- Meters2Yards

The following are some unacceptable subroutine names:

- User.Welcome
- 2Printer
- Miles 1.609

If you want, a subroutine can require that the code statement that calls that subroutine provide one or more arguments or variables that the subroutine can work with. Any time you need preexisting data to perform the task within the subroutine, arguments are very helpful. For example, the following subroutine accepts an argument to be used in the subroutine as a message to be displayed to the user:

13.1.2 Calling a Subroutine

Now that you've learned how to create a subroutine, how do you call one? You can call a subroutine throughout the rest of the application once you've declared and created it. You can call subroutines by using the Call keyword or just entering the name of the subroutine on a line of code. For example, to call a subroutine called ShowMessage, you could enter



If you use Call, you must enclose the arguments in parentheses. This is simply a convention that code requires.

13.1.3 Exiting a Subroutine

The code within your subroutine will execute until one of two things happens. First, the subroutine might get down to the last line, the End Sub line, which terminates the subroutine and passes the

Notes baton back to the caller. This statement can appear only once at the end of the subroutine declaration.

13.2 Restart Sequence

The restart sequence is made up of three machine cycles

In the 1st machine cycle:

The microprocessor sends the INTA signal. While INTA is active the microprocessor reads the data lines expecting to receive, from the interrupting device, the opcode for the specific RST instruction.

In the 2nd and 3rd machine cycles:

The 16-bit address of the next instruction is saved on the stack.


Then the microprocessor jumps to the address associated with the specified RST instruction.

How does the external device produce the opcode for the appropriate RST instruction?

The opcode is simply a collection of bits. So, the device needs to set the bits of the data bus to the appropriate value in response to an INTA signal.

During the interrupt acknowledge machine cycle, (the 1st machine cycle of the RST operation):

The Microprocessor activates the INTA signal. This signal will enable the Tri-state buffers, which will place the value EFH on the data bus.



Task

Explain, How we work with subroutines?

13.3 Conditional Call Instruction

The call instruction is used for two purposes.

1. to invoke an internal or external procedure as a subroutine
2. to activate/deactivate a condition trap subroutine handler

13.3.1 Subroutine call

```
call procedureName [ argumentExpression [ , argumentExpression [ ...etc ] ]
```

This form of the call instruction invokes an internal or external procedure named procedureName, or a built-in function, as a subroutine. A series of arguments can optionally be prepared by the argument Expression values. These arguments can be acquired in the subroutine by using either the parse arg or arg instructions, or the arg built-in function. The subroutine can optionally return a result, which can be acquired by referencing the special RESULT variable. Often the result can be processed directly in-place, by using a function call instead.

The name provided as procedureName can be provided as a quoted string -- e.g. 'X2C'. When it is a quoted string, an internal procedure is not invoked, and a built-in function or external procedure is invoked instead. When a built-in function is invoked, using this technique, the name in quotes should be in upper case.

The search order for the target procedure, or built-in function is:

1. internal procedure--unless the procedure name was enclosed in quotes
2. built-in function
3. external procedure

The mechanism for locating and activating an external procedure is implementation-dependent.

Here is an example of a subroutine invocation using the call instruction.

```
/* main program */
do n=1 to 5
  call factorial n
say 'The factorial of' n 'is:' RESULT
end
return
```

```
factorial : procedure
n = arg(1)
if n = 1 then
return 1
return n * factorial( n - 1 )
```

The above program would normally use a function call instead. The following shows how the main program would be coded in this case.

```
/* main program */
do n=1 to 5
  say 'The factorial of' n 'is:' factorial( n )
end
return
```

Normally, you use the call instruction to invoke a function, when you will ignore (or not require) a return value. The following shows how the value built-in function is invoked as a subroutine.

```
call VALUE 'Destination', 'Nice', 'Vacation' /* assigns new destination */
```

Processing state information is SAVED during function and subroutine calls

The processing state of the following information is saved when a function or subroutine call is performed. The state of this information is restored when the function or subroutine returns pending control structures

if ... then ... else ...

do ... end

do i=1 to n ... end [and similar do loops]

select when ... then ... etc otherwise ... end

numeric option settings

digits, fuzz, form

command address settings

current and saved destinations

Notes

pending condition handler information

call/signal on : error, failure, halt, notready, novalue, syntax

pending condition information

information that is returned by the condition built-in function

settings associated with the options instruction

elapsed time clock

a function or subroutine can alter the setting of the elapsed time clock, which is provided by the time built-in function, without affecting the setting that is used in the calling procedure.

trace settings

a function or subroutine can alter trace settings, which are established by the trace instruction or the trace built-in function, without affecting the settings that are used in the calling procedure.



Support for the options instruction is implementation-dependent, and the settings may not be saved and restored during function or subroutine calls.

13.3.2 Trap subroutine call activation/deactivation

call ON conditionName [NAME trapLabel]

call OFF conditionName

The call ON or OFF instructions activate or deactivate the handling of condition traps by an associated procedure. When active, a conditional trap will cause the trapLabel to be invoked as a subroutine. In a call ON instruction, if the trapLabel is absent, the label that is invoked is the same as the condition name. The handling of condition traps can alternatively be activated or deactivated by the signal ON or OFF instructions.

The conditionName in the call ON or OFF is one of the following:

ERROR

FAILURE

HALT

NOTREADY

Unlike the signal ON or OFF trap anticipation instructions, the following condition names are ineligible in a call ON or OFF instruction:

SYNTAX

NOVALUE



The call ON or OFF instructions activate or deactivate the handling of condition traps by an associated procedure.

13.4 Return Instruction

In computer programming, a return statement causes execution to leave the current subroutine and resume at the point in the code immediately after where the subroutine was called, known as its return address. The return address is saved, usually on the process's call stack, as part of the operation of making the subroutine call. Return statements in many languages allow a function to specify a return value to be passed back to the code that called the function.



Case Study

Subroutine History

Notes

Language support

In the (very) early assemblers, subroutine support was limited. Subroutines were not explicitly separated from each other or from the main program, and indeed the source code of a subroutine could be interspersed with that of other subprograms. Some assemblers would offer predefined macros to generate the call and return sequences. Later assemblers (1960s) had much more sophisticated support for both in-line and separately assembled subroutines that could be linked together.

Self-modifying code

The first use of subprograms was on early computers that were programmed in machine code or assembly language, and did not have a specific call instruction. On those computers, each subroutine call had to be implemented as a sequence of lower level machine instructions that relied on self-modifying code. By replacing the operand of a branch instruction at the end of the procedure's body, execution could then be returned to the proper location (designated by the return address) in the calling program (usually just after the instruction that jumped into the subroutine).

Subroutine libraries

Even with this cumbersome approach, subroutines proved very useful. For one thing they allowed the same code to be used in many different programs. Moreover, memory was a very scarce resource on early computers, and subroutines allowed significant savings in program size.

In many early computers, the program instructions were entered into memory from a punched paper tape. Each subroutine could then be provided by a separate piece of tape, loaded or spliced before or after the main program; and the same subroutine tape could then be used by many different programs. A similar approach was used in computers whose main input was through punched cards. The name "subroutine library" originally meant a library, in the literal sense, which kept indexed collections of such tapes or card decks for collective use.

Return by indirect jump

To remove the need for self-modifying code, computer designers eventually provided an "indirect jump" instruction, whose operand, instead of being the return address itself, was the location of a variable or processor register containing the return address.

On those computers, instead of modifying the subroutine's return jump, the calling program would store the return address in a variable so that when the subroutine completed, it would execute an indirect jump that would direct execution to the location given by the predefined variable.

Jump to subroutine

Another advance was the "jump to subroutine" instruction, which combined the saving of the return address with the calling jump, thereby minimizing overhead significantly.

In the IBM System/360, for example, the branch instructions BAL or BALR, designed for procedure calling, would save the return address in a processor register specified in the instruction. To return, the subroutine had only to execute an indirect branch instruction (BR) through that register. If the subroutine needed that register for some other purpose (such as calling another subroutine), it would save the register's contents to a private memory location or a register stack.

Contd. ...

Notes

In the HP 2100, the JSB instruction would perform a similar task, except that the return address was stored in the memory location that was the target of the branch. Execution of the procedure would actually begin at the next memory location. In the HP 2100 assembly language, one would write, for example

...

JSB MYSUB (Calls subroutine MYSUB.)

BB ... (Will return here after MYSUB is done.)

to call a subroutine called MYSUB from the main program. The subroutine would be coded as

MYSUB NOP (Storage for MYSUB's return address.)

AA ... (Start of MYSUB's body.)

...

JMP MYSUB,I (Returns to the calling program.)

The JSB instruction placed the address of the NEXT instruction (namely, BB) into the location specified as its operand (namely, MYSUB), and then branched to the NEXT location after that (namely, AA = MYSUB + 1). The subroutine could then return to the main program by executing the indirect jump JMP MYSUB,I which branched to the location stored at location MYSUB.

Compilers for Fortran and other languages could easily make use of these instructions when available. This approach supported multiple levels of calls; however, since the return address, parameters, and return values of a subroutine were assigned fixed memory locations, it did not allow for recursive calls.

Incidentally, a similar technique was used by Lotus 1-2-3, in the early 1980s, to discover the recalculation dependencies in a spreadsheet. Namely, a location was reserved in each cell to store the "return" address. Since circular references are not allowed for natural recalculation order, this allows a tree walk without reserving space for a stack in memory, which was very limited on small computers such as the IBM PC.

Questions:

1. Define Self-modifying code in the term of subroutines.
2. Explain the concept of jumping in the subroutines.

13.5 Summary

- A subroutine is a block of code that can be called from anywhere in a program to accomplish a specific task.
- The discipline of object-oriented programming is based on objects and methods.
- The name of a subroutine should adequately describe what the subroutine is for.
- You can call subroutines by using the Call keyword or just entering the name of the subroutine on a line of code.
- The Microprocessor activates the INTA signal. This signal will enable the Tri-state buffers, which will place the value EFH on the data bus.

13.6 Keywords

Notes

Exit Sub is use to exit a subroutine, particularly when handling errors.

Opcode: It is simply a collection of bits.

Return address: The return address is saved, usually on the process's call stack

Subroutine_Name is the name of the subroutine and through arguments are optional arguments, often called parameters

VB Script: In this arguments must be supplied in the order specified by the procedure



Lab Exercise

1. Give code for Subroutine call in microprocessor.
2. Write the opcode for Restart Sequence.

13.7 Self-Assessment Questions

1. A subroutine is a container that holds a series of Statements.
 - (a) End Sub
 - (b) VBScript
 - (c) MsgBox
 - (d) None of these
2. Declare subroutines using the Sub keyword and end them using the Statement.
 - (a) End Sub
 - (b) VBScript
 - (c) MsgBox
 - (d) None of these
3. Which among the following are the valid subroutine names:
 - (a) WelcomeTheUser
 - (b) PrintInvoice
 - (c) Meters2Yards
 - (d) All of the above
4. The is simply a collection of bits.
 - (a) INTA
 - (b) opcode
 - (c) EFH
 - (d) None of these
5. When a built-in function is invoked, using this technique, the name in quotes should be in
 - (a) upper case
 - (b) lower case
 - (c) middle case
 - (d) None of these
6. The call instructions activate or deactivate the handling of condition traps by an associated procedure.
 - (a) OFF or ON
 - (b) UP and Down
 - (c) ON or OFF
 - (d) None of these

13.8 Review Questions

1. Explain the definition of Subroutine in brief.
2. Describe how to create a Subroutine.

Notes

3. Explain the procedure to call a Subroutine.
4. Describe about Restart Sequence.
5. Define Conditional Call instruction and its uses.
6. Discuss Return Instruction in brief.
7. Define:
 - (a) Subroutine_Name,
 - (b) VBScript,
 - (c) CurrentMessage,
 - (d) procedureName
8. What do you mean by an argument or a parameter?
9. What are the machine cycles in a restart sequence?
10. Describe the processing state of information which is saved when a function or subroutine call is performed.

Answers for Self-Assessment Questions

1. (b) 2. (a) 3. (d) 4. (b) 5. (a) 6. (c)

13.9 Further Reading



Books

Subroutine by: Frederic P. Miller, Agnes F. Vandome, John McBrewster.



Online link

<http://www.go4expert.com/forums/showthread.php?t=4098>

Unit 14: Interrupts

Notes

CONTENTS

Objectives

Introduction

14.1 8085 Interrupts

14.2 Type 8085 Interrupts

 14.2.1 Makeable Interrupts

 14.2.2 Non-Maskable Interrupt

14.3 Interrupt 8085 Microprocessor

14.4 Differences between Intel 8080 and 8085 Processors

 14.4.1 Arithmetic

 14.4.2 Logical

 14.4.3 Compliment

 14.4.4 Rotate

 14.4.5 Clear

 14.4.6 Branching

 14.4.7 Jump on Condition (of a Bit)

 14.4.8 Call

14.5 Summary

14.6 Keywords

14.7 Self-Assessment Questions

14.8 Review Questions

14.9 Further Reading

Objectives

After studying this unit, you will able to understand the following:

- Overview of 8085 interrupts
- Type 8085 interrupts
- Overview of interrupt 8085 microprocessor
- Differences between Intel 8080 and 8085 processors

Introduction

A signal that gets the attention of the CPU and is usually generated when I/O is required. For example, hardware interrupts are generated when a key is pressed or when the mouse is moved.

Notes

Pressed or when the mouse is moved. Software interrupts are generated by a program requiring disk input or output.

An internal timer may continually interrupt the computer several times per second to keep the Time of day current or for time sharing purposes. When an interrupt occurs, control is transferred to the operating system, transferred to the operating system, which determines the action to be taken. Interrupts are prioritized; the higher the priority, the faster the interrupt will be serviced. Basically, a single computer can perform only one computer instruction at a time. But, because it can be interrupted, it can take turns in which programs or sets of instructions that it performs. This is instructions that it performs. This is known as multitasking.

An operating system usually has some code that is called an interrupt handler. The interrupt handler prioritizes the interrupts and saves them in a queue if more than one is waiting to be handled.

The operating system has another little program, sometimes called a scheduler. The operating system has another little program, sometimes called a scheduler.

14.1 8085 Interrupts

Interrupt is a process where an external device can get the attention of the microprocessor. The process starts from the I/O device and is asynchronous.

Interrupts in 8085 microprocessor are classified into Hardware interrupts and Software interrupts.

1. Hardware interrupt - TRAP, RST7.5, RST6.5, RST5.5, and INTR.
2. Software interrupt - RST0, RST1, RST2, RST3, RST4, RST5, RST6, RST7.

Intel 8085 microprocessor is the next generation of Intel 8080 CPU family. In addition to being faster than the 8080, the 8085 had the following enhancements:

- Intel 8085 had single 5 Volt power supply.
- Clock oscillator and system controller were integrated on the chip.
- The CPU included serial I/O port.
- Two new instructions were added to 8085 instruction set.

The CPU also included a few undocumented instructions. These instructions were supposed to be a part of the CPU instruction set, but at the last moment they were left undocumented because they were not compatible with forthcoming Intel 8086.



Interrupt is a signal send by external device to the processor so as to request the processor to perform a particular work.

14.2 Type 8085 Interrupts

The 8085 has facilities for servicing interrupts similar to the 8080. The functional items required are an Interrupt Request (INTR) pin, an Interrupt Acknowledge (INTA) pin, an Interrupt Enable (INTE) pin; eight interrupt vectors in low RAM, and the Restart instruction. These perform in the same way as the 8080 interrupt system. Here is a brief review:

1. A program is running normally in the system. The 8214 Priority Interrupt Controller or similar circuit has its compare mask set to some priority level. The Interrupt Enable bit has been set on by some previous routine, enabling interrupts.

2. A device wishes to interrupt the system. It raises its own line which connects directly to the 8214. The 8214 compares this request with the current status of the system. If the new request is higher in priority than the existing (if any), the interrupt will be allowed. If not, the interrupt will be latched for later use, but no further action is taken.
3. The Interrupts Enabled line exiting the 8085 is high, indicating that interrupts are permitted. The 8214 raises the Interrupt line, which causes the MP to finish the current instruction, and then enter an interrupt service cycle. The MP generates the Interrupts Acknowledge line at the beginning of this cycle to permit the 8214 to proceed.
4. Upon receipt of the INTA line, the 8214 along with an 8212 octal latch or similar circuit, generates a Restart instruction which it jams onto the data bus at T3 of the interrupt service cycle. The MP receives this, and removes from it the three-bit modulo-8 vector, which it then multiplies by 8 to find the vector in low RAM. This vector contains one or more instructions which can service the device causing the interrupt.
5. The execution of the Restart instruction causes the address of the next normal instruction to be executed, obtained from PC, to be placed onto the stack. The next machine cycle will be the M1 of the instruction located in the vector in low RAM. This instruction can now guide the MP to the routine to service the interrupt.
6. At the end of the interrupt service routine, a Return (RTN) instruction will cause the popping of the address off the stack which was of the next instruction to be serviced if the interrupt had not occurred.

The system now finds itself back where it came from

There are three possible variations to the above scenario. First, unlike the 8080, the 8085 will permit the interrupt as described above as long as no other interrupts are pending which are of greater importance. These, of course, are the 5.5, 6.5, 7.5, and Trap. If any of these are pending, they will be serviced first.

Secondly, while the 8214 was the original device to service interrupts on the 8080 system, the 8085 can work with the 8259A Programmable Interrupt Controller as well. This is a more complex device, programmable as to how it handles interrupts, and stackable to two levels, providing as many as 64 levels of interrupt for the '85. The 8259A, moreover, generates Call instructions as well as Restarts. This means that a Call may be jammed onto the data bus during T3 of the interrupt cycle, instead of Restart. While the Restart provides a vector to eight different places in low RAM, depending upon the modulo-8 bits it contains, the Call contains a full two-byte-wide address, which can effectively vector the MP to any-place within the 64K RAM address space. This obviously provides a vastly extended ability to handle interrupts more efficiently.

The third item to be aware of is that the Interrupt Enable flip-flop of the 8080 is now observable as the IE bit #3 of the byte obtained by executing the RIM instruction. It hitherto has not been available, and its status must be remembered by the programmer. Now the bit may be checked with the RIM instruction.

Interrupts can be classified into two types:

1. Makeable Interrupts (Can be delayed or rejected)
2. Non-makeable Interrupts (Cannot be delayed or rejected)

14.2.1 Makeable Interrupts

Three maskable interrupts are provided in the 8085, each with their own pins. They are named RST 5.5, RST 6.5, and RST 7.5, respectively. To see where these names come from, study this chart

Notes

NAME: ADDRESS:

RST 0 00H

RST 1 08H

RST 2 10H

RST 3 18H

RST 4 20H

TRAP 24H

RST 5 28H

REST 5.5 2CH

RST 6 30H

RST 6.5 34H

RST 7 38H

RST 7.5 3CH



Caution

The normal vectors for the Restart instructions 0 through 7, as created by the 8214. They are 8 bytes apart, which is ample room for such jumps as are needed to obtain the interrupt servicing routines.

Now look at the bold face items. These items have vector areas which are between the original vectors in RAM. The 5.5, for instance, is half way between the RST 5 and the RST 6 vectors, hence the "0.5". If all the vectors were in use, those located above address 20H would each have only four bytes in which to locate and jump to the interrupt service routine. This should be enough room, however, if used wisely. Note also that the Trap interrupt is located at the 4.5 point in the vectors.

The 5.5, 6.5, and 7.5 vectors have several items in common. First, they each have their own pin directly into the 8085. These pins will accept asynchronous interrupt requests without the need for any sort of external priority interrupt device. Secondly, these interrupts are individually makeable. This is accomplished via the Set Interrupt Mask instruction. This instruction allows bits to be set or cleared which will permit or deny an interrupt on one of these lines to force the '85 into an interrupt service cycle. When an input is received on one of these lines and its respective mask bit is enabled (set to 0), the processor will finish the current machine cycle, then enter a interrupt service cycle in which an automatic jam inside the MP will vector it to 2CH, 34H, or 3CH for 5.5, 6.5, or 7.5 respectively. Those locations will assumedly have been previously set to contain directions to the interrupt servicing routines.

The RST 5.5 and RST 6.5 interrupts are "level sensitive". This means that the device wishing to interrupt will apply a steady high level to the appropriate pin and hold it there until the 8085 gets around to responding. When the '85 recognizes the applied high level, it will permit the interrupt to be serviced in the next machine cycle. The mask bits set by the SIM instruction will directly determine what the RIM instruction sees with respect to the 5.5 and 6.5 interrupt pending bits. If the mask bits are set high (to a 1), these interrupts are masked off. This means that a following RIM will not see them as pending. If the mask bits are set to 0 (enabled), a RIM will see the true condition in bits 4 and 5 of the mask byte.

The RST 7.5 interrupt is "edge sensitive". This means that a pulse applied to this pin, requesting an interrupt, can come and go before the processor gets around to servicing it. This is possible because, unlike the 5.5 and 6.5, the 7.5 has a flip-flop just inside its pin which instantly registers the fact that an interrupt request, albeit short, was applied to the device. This flip-flop provides a bit which is read in RIM instruction as bit 6. This bit will indicate an interrupt pending if a quick

pulse is applied to pin 7.5, even though bit 2 of the SIM instruction, the 7.5 mask bit, is turned on (disabled). Bit 2 of SIM byte, therefore, acts differently as a mask bit than does bits 0 and 1 for 5.5 and 6.5. Whereas bits 0 and 1 will mask off all indication of action on pins 5.5 and 6.5, bit 2 will allow the indication of a 7.5 interrupt pending, but will prevent the actual servicing of the 7.5 vector unless the mask is enabled for it. In this way, even though the mask set by the SIM prevents the MP from servicing a 7.5 interrupt, the fact that such an interrupt did occur, captured by the flip-flop, and is indicated to whatever routine next executes a RIM instruction.

While the normal interrupt and 5.5 and 6.5 interrupts' enable bits are reset when these are serviced, the 7.5 interrupt flip-flop must be turned off individually. This may be accomplished by actually responding to the interrupt, just like the other interrupts above; by having the 8085 receiving a / RESET IN, which would also reset the whole system; or by executing a SIM instruction in which bit 4 of the SIM byte is set on. This bit 4 is the "Reset RST 7.5" bit, and will reset the flip-flop if it is on when a SIM is executed.



Task

Differentiate between RAM and ROM.

14.2.2 Non-maskable Interrupt

The Trap instruction is a non-maskable interrupt provision for the 8085. There is no mask bit related to it, and no control bits of any kind. It is used for interrupts of a catastrophic nature, such as the impending doom of a power failure. It is essentially an edge-sensitive input, since its pin connects directly inside the '85 to a flip-flop to capture the fact that a request was made. However, the inside circuitry around the flip-flop requires that although the flip-flop is set, the asserted level be continually applied thereafter until the processor enters the service cycle. This is shown in a diagram in the documentation. The Trap, therefore, is called both edge-sensitive and level sensitive as well. The order of priority for all of the interrupts of the 8085, from least important to most important, are the Restart 0 through Restart 7, RST 5.5, RST 6.5, RST 7.5, and finally the Trap. Remember that through the use of the 8214, the RST 0 through 7 interrupts are also prioritized, with 0 as the least important and 7 as the most important. Collectively, the 8085 has a complete set of interrupt capabilities that should serve every need.

When the Microprocessor receives an interrupt signal, it suspends the currently executing program and jumps to an Interrupt Service Routine (ISR) to respond to the incoming interrupt. Each interrupt will most probably have its own ISR.

Responding to an interrupt may be immediate or delayed depending on whether the interrupt is maskable or non-maskable and whether interrupts are being masked or not.

There are two ways of redirecting the execution to the ISR depending on whether the interrupt is vectored or non-vectored.

Vectored: The address of the subroutine is already known to the Microprocessor

Non Vectored: The device will have to supply the address of the subroutine to the Microprocessor

When a device interrupts, it actually wants the MP to give a service which is equivalent to asking the MP to call a subroutine. This subroutine is called ISR (Interrupt Service Routine)



Did u know?

1. The 'EI' instruction is a one byte instruction and is used to Enable the non-maskable interrupts.
2. The 8085 has a single Non-Maskable interrupt. The non-maskable interrupt is not affected by the value of the Interrupt Enable flip flop.

Notes

The 8085 has 5 interrupt inputs:

1. The INTR input.
2. The INTR input is the only non-vectored interrupt.
3. INTR is maskable using the EI/DI instruction pair.
4. RST 5.5, RST 6.5, RST 7.5 are all automatically vectored.
5. RST 5.5, RST 6.5, and RST 7.5 are all maskable.
6. TRAP is the only non-maskable interrupt in the 8085.
7. TRAP is also automatically vectored.

An interrupt vector is a pointer to where the ISR is stored in memory.

All interrupts (vectored or otherwise) are mapped onto a memory area called the Interrupt Vector Table (IVT). The IVT is usually located in memory page 00 (0000H - 00FFH). The purpose of the IVT is to hold the vectors that redirect the microprocessor to the right place when an interrupt arrives.



Example: Let, a device interrupts the Microprocessor using the RST 7.5 interrupt line.

Because the RST 7.5 interrupt is vectored, Microprocessor knows, in which memory location it has to go using a call instruction to get the ISR address. RST7.5 is known as Call 003Ch to Microprocessor. Microprocessor goes to 003C location and will get a JMP instruction to the actual ISR address. The Microprocessor will then, jump to the ISR location.

The 8085 Non-Vectored Interrupt Process

1. The interrupt process should be enabled using the EI instruction.
2. The 8085 checks for an interrupt during the execution of every instruction.
3. If INTR is high, MP completes current instruction, disables the interrupt and sends INTA (Interrupt acknowledge) signal to the device that interrupted.
4. INTA allows the I/O device to send a RST instruction through data bus.
5. Upon receiving the INTA signal, MP saves the memory location of the next instruction on the stack and the program is transferred to 'call' location (ISR Call) specified by the RST instruction.
6. Microprocessor Performs the ISR.
8. ISR must include the 'EI' instruction to enable the further interrupt within the program.



Did u know?

Stack pointer is a 16 bit register. This register is always incremented / decremented by 2.

14.3 Interrupt 8085 Microprocessor

The 8085 microprocessor has 5 interrupts. They are presented below in the order of their priority (from lowest to highest):

INTR is maskable 8080A compatible interrupt. When the interrupt occurs the processor fetches from the bus one instruction, usually one of these instructions:

- One of the 8 RST instructions (RST0 - RST7). The processor saves current program counter into stack and branches to memory location $N * 8$ (where N is a 3-bit number from 0 to 7 supplied with the RST instruction).

- **CALL** instruction (3 byte instruction). The processor calls the subroutine, address of which is specified in the second and third bytes of the instruction.

RST5.5 is a maskable interrupt. When this interrupt is received the processor saves the contents of the PC register into stack and branches to 2Ch (hexadecimal) address.

RST6.5 is a maskable interrupt. When this interrupt is received the processor saves the contents of the PC register into stack and branches to 34h (hexadecimal) address.

RST7.5 is a maskable interrupt. When this interrupt is received the processor saves the contents of the PC register into stack and branches to 3Ch (hexadecimal) address.

Trap is a non-maskable interrupt. When this interrupt is received the processor saves the contents of the PC register into stack and branches to 24h (hexadecimal) address.

All maskable interrupts can be enabled or disabled using EI and DI instructions. RST 5.5, RST6.5 and RST7.5 interrupts can be enabled or disabled individually using SIM instruction.

I/O ports

256 Input ports

256 Output ports

Registers

Accumulator or A register is an 8-bit register used for arithmetic, logic, I/O and load/store operations.

Flag is an 8-bit register containing 5 1-bit flags:

- Sign - set if the most significant bit of the result is set.
- Zero - set if the result is zero.
- Auxiliary carry - set if there was a carry out from bit 3 to bit 4 of the result.
- Parity - set if the parity (the number of set bits in the result) is even.
- Carry - set if there was a carry during addition, or borrow during subtraction/comparison.

General registers:

- 8-bit B and 8-bit C registers can be used as one 16-bit BC register pair. When used as a pair the C register contains low-order byte. Some instructions may use BC register as a data pointer.
- 8-bit D and 8-bit E registers can be used as one 16-bit DE register pair. When used as a pair the E register contains low-order byte. Some instructions may use DE register as a data pointer.
- 8-bit H and 8-bit L registers can be used as one 16-bit HL register pair. When used as a pair the L register contains low-order byte. HL register usually contains a data pointer used to reference memory addresses.

Stack pointer is a 16 bit register. This register is always incremented/decremented by 2.

Program counter is a 16-bit register.

Instruction Set

Instruction set of Intel 8085 microprocessor consists of the following instructions:

- Data moving instructions.
- Arithmetic - add, subtract, increment and decrement.
- Logic - AND, OR, XOR and rotate.

Notes

- Control transfer - conditional, unconditional, call subroutine, return from subroutine and restarts.
- Input/Output instructions.
- Other - setting/clearing flag bits, enabling/disabling interrupts, stack operations, etc.

14.4 Differences between Intel 8080 and 8085 Processors

There were multiple versions of 8085 microprocessors. The original version of the 8085 microprocessor without suffix "A" was manufactured by Intel only, and was very quickly replaced with 8085A containing bug fixes. A few years after that, around 1980, Intel introduced 8085AH - HMOS version of 8085A. There was also 80C85A - CMOS version of the 8085A. It's not clear if 80C85 was ever manufactured by Intel or not, but it was produced by at least two second source manufacturers - OKI and Tundra Semiconductor. Tundra Semiconductor manufactured the fastest 8085 microprocessor running at 8 MHz.

Second source manufacturers: AMD, Mitsubishi, NEC, OKI, Siemens, Toshiba. Soviet Union also manufactured clones of Intel 8085 CPU.

14.4.1 Arithmetic

The arithmetic instructions usually include addition, subtraction, division, multiplication, incrementing, and decrementing although division & multiplication were not available in most early CPU's. There are two flags used with arithmetic that tell the program what was the outcome of an instruction. One is the Carry (C) flag. The other is the Zero (Z) flag. The C flag will be explained in the following example of addition. The Z flag, if set, says that the result of the instruction left a value of 0 in the accumulator. We will see the Z flag used in a later lesson.

Addition

This is straightforward and is simply to add two numbers together and get the result. However there is one more thing. If, in the addition, the result was too big to fit into the accumulator, part of it might be lost. There is a safeguard against this. Take the case of 11111111b (255) and 11111111b (255). These are the largest numbers that can fit into an 8-bit register or memory location. You can add these as decimal numbers. The binary value for 510 is 111111110b (9 bits). The accumulator is only 8 bits wide, it is a byte. How do you fit a 9-bit number into 8 bits of space? The answer is, you can't, and it's called an OVERFLOW condition. So how do we get around this dilemma? We do it with the CARRY (C) flag. If the result of the addition is greater than 8 bits, the CARRY (C) flag will hold the 9th bit. In this case the accumulator would have in 11111110b (254) and the C flag would be a 1, or set. This 1 has the value of 256 because this is the 9th bit. We haven't covered a 9-bit number, but they come up all the time as overflows in addition. Since we are using base 2, and we found out in lesson 2 that the 8th bit (bit 7) in a byte is worth 128, then the 9th bit is worth 2 times that, or 256. Adding 254 and 256, we get 510, the answer, and we didn't lose anything, because of the C flag. Had the result of the addition not caused an overflow, the C flag would be 0, or cleared.

Subtraction

In the case of subtraction, the process is more difficult to explain, and as such, I'm not going to cover it here. It involves 1's complement and 2's complement representation.

Multiplication and Division

In the micro we will be using, the 8085, multiply and divide instructions are not available so we will wait till later (EL31G-Microprocessors II) to talk about them. They do, however, do just what the names suggest.

Increment and Decrement**Notes**

Two other instructions are included in the arithmetic group. They are increment and decrement. These instructions are used to count events or loops in a program. Each time an increment is executed, the value is incremented by 1. A decrement, decrements the value by 1. These can be used with conditional jumps to loop a section of program, a certain number of times.

14.4.2 Logical

In micros there are other mathematical instructions called logical instructions. These are OR, AND, XOR, ROTATE, COMPLEMENT and CLEAR. These commands are usually not concerned with the value of the data they work with, but, instead, the value, or state, of each bit in the data.

OR

The OR function can be demonstrate by taking two binary numbers, 1010b and 0110b. When OR'ing two numbers, it doesn't matter at which end you start, right or left. Let's start from the left. In the first bit position there is a 1 in the first number and a 0 in the second number. This would result in a 1. The next bit has a 0 in the first number and a 1 in the second number. The result would be 1. The next bit has a 1 in the first number and a 1 in the second number. The result would be a 1. The last bit has a 0 in the first number and a 0 in the second number, resulting in a 0. So the answer would be 1110b. The rule that gives this answer says that with an OR, a 1 in either number result in a 1, or said another way, any 1 in, gives a 1 out.

AND

AND in uses a different rule. The rule here is a 0 in either number will result in a 0, for each corresponding bit position. Using the same two numbers 1010b and 0110b the result would be 0010b. You can see that every bit position except the third has a zero in one or the other number. Another way of defining an AND is to say that a 1 AND a 1 results in a 1.

XOR (Exclusive OR)

XOR'ing is similar to OR'ing with one exception. An OR can also be called an inclusive OR. This means that a 1 in either number or both will result in a 1. An eXclusive OR says that if either number has a 1 in it, but not both, a 1 will result. A seemingly small difference, but crucial. Using the same two numbers, the result would be 1100b. The first two bits have a 1 in either the first or the second number but not both. The third bit has a 1 in both numbers, which results in a 0. The fourth has no 1's at all, so the result is 0. The difference may seem small, even though the OR and XOR result in different answers. The main use of an XOR is to test two numbers against each other. If they are the same, the result will be all 0's, otherwise the answer will have 1's where there are differences.

14.4.3 Compliment

Complimenting a number results in the opposite state of all the 1's and 0's. Take the number 1111b. Complimenting results in 0000b. This is the simplest operator of all and the easiest to understand. Its uses are varied.

14.4.4 Rotate

These instructions rotate bits in a byte. The rotation can be left or right, and is done one bit each instruction. An example might be where the accumulator has a 11000011b in it. If we rotate left, the result will be 1000011b. You can see that bit 7 has now been moved into bit 0 and all the other bits have move 1 bit position in, the left direction

Notes

14.4.5 Clear

This instruction clears, or zero's out the accumulator. This is the same as moving a 0 into the accumulator. This also clears the C flag and sets the Z flag.

14.4.6 Branching

There are also program flow commands. These are branches or jumps. They have several different names reflecting the way they do the jump or on what condition causes the jump, like an overflow or under flow, or the results being zero or not zero. But all stop the normal sequential execution of the program, and jump to another location, other than the next instruction in sequence.

14.4.7 Jump on Condition (of a Bit)

These instructions let you make a jump based on whether a certain bit is set (a 1) or cleared (a 0). This bit can be the CY (carry) flag, the Z (zero) flag, or any other bit.

14.4.8 Call

There is also a variation on a jump that is referred to as a CALL. A CALL does a jump, but then eventually comes back to the place where the CALL instruction was executed and continues with the next instruction after the CALL. This allows the programmer to create little sub-programs, or subroutines, that do repetitive tasks needed by the main program. This saves programming time because once the subroutine is written; it can be used by the main program whenever it needs it, a kind of way to create your own instructions.

*Case Study***Automatic Verification of External Interrupt Behaviors
for Microprocessor Design**

Interrupt behaviors, particularly the external ones, are difficult to verify in a microprocessor. Because the external interrupt arrival time and the microprocessor response time must be precise, verification requires sophisticated hardware and software design. This paper proposes a computer-aided design tool, called processor exception verification tool (PEVT), to verify the external interrupt behaviors of microprocessors, including individual, multiple, and nested interrupts. An architecture description language extension, called Exception Description Language (EXPDL), is developed for the designer to capture the external interrupt behaviors for the microprocessor under verification. PEVT is responsible for generating the verification cases, consisting of both the hardware and software modules, which are then used to trigger the expected behaviors. A monitor is also generated from the EXPDL description to verify these cases. PEVT has been applied to the verification of an academic implementation of the ARM7 microprocessor core and a public domain scalable processor architecture (SPARC) microprocessor core. The ARM7 has had a system-on-a-chip test chip and software porting including multimedia applications (MP3/JPEG/ ...) and a real time operating system muC-OSII. PEVT successfully identified several sophisticated remaining bugs with 527 lines of EXPDL description and took only 4 204 961 cycles of register transfer language simulation with execution time of 4.5 h in a SUN Blade2000 workstation. The experiment shows that PEVT could generate highly focused verification cases, less than 98 cycles per case on the average, which identify potential bugs with much less simulation cycles at the early verification stage, compared with traditional manual-based approaches.

Questions:

1. What did you understand to EXPDL?
2. Explain the concept of External Interrupt.

14.5 Summary

Notes

- Interrupt is a signal send by external device to the processor so as to request the processor to perform a particular work.
- In the 8085 microprocessor are eight data lines, D_0 through D_7 .
- Accumulator or A register is an 8-bit register used for arithmetic, logic, I/O and load/store operations.
- Flag is an 8-bit register containing 5 1-bit flags: Sign - set if the most significant bit.

14.6 Keywords

INTR: INTR is maskable 8080A compatible interrupt. When the interrupt occurs the processor fetches from the bus one instruction.

RST5.5: RST5.5 is a maskable interrupt. When this interrupt is received the processor saves the contents of the PC register into stack and branches to 2Ch (hexadecimal) address.

RST6.5: RST6.5 is a maskable interrupt. When this interrupt is received the processor saves the contents of the PC register into stack and branches to 34h (hexadecimal) address.

RST7.5: RST7.5 is a maskable interrupt. When this interrupt is received the processor saves the contents of the PC register into stack and branches to 3Ch (hexadecimal) address.

Trap: Trap is a non-maskable interrupt. When this interrupt is received the processor saves the contents of the PC register into stack and branches to 24h (hexadecimal) address.



Lab Exercise

1. Give the instructions that perform the logical operations.
2. Draw the block diagram of CPU architecture.

14.7 Self-Assessment Questions

1. INTR stand for

(a) Interrupt Request	(b) Interrupt Acknowledge
(c) Interrupt Enable	(d) Internet Request
2. INTA stand for

(a) Interrupt Request	(b) Interrupt Acknowledge
(c) Interrupt Enable	(d) Internet Request
3. INTE stand for

(a) Interrupt Request	(b) Interrupt Acknowledge
(c) Interrupt Enable	(d) Internet Request
4. The 8085 has facilities for servicing interrupts similar to the

(a) 8086	(b) 8087
(c) 8080	(d) 8085

Notes

5. The arithmetic instructions usually include
 - (a) addition
 - (b) subtraction
 - (c) division
 - (d) All above
6. In 8085 the interrupts are classified as interrupts.
 - (a) hardware
 - (b) software
 - (c) Both
 - (d) None
7. Hardware interrupts are
 - (a) RST5.0
 - (b) RST5.2
 - (c) RST5.4
 - (d) RST5.5
8. Software interrupts are
 - (a) RST3
 - (b) RST5.0
 - (c) RST5.4
 - (d) RST5.5
9. Addressing modes are name.
 - (a) 2
 - (b) 3
 - (c) 4
 - (d) 5
10. RST 4.5 is called as
 - (a) RST5
 - (b) RST7
 - (c) TRAP
 - (d) RST7.5

14.8 Review Questions

1. What way interrupts are classified in 8085?
2. What are hardware interrupts?
3. What are input and output devices?
4. Differences between Intel 8080 and 8085 processors.
5. What are software interrupts?
6. What is interrupt?
7. Explain Different addressing modes?
8. Explain the type interrupts 8085.
9. What are the various flags used in 8085?
10. What is program counter?

Answers for Self-Assessment Questions

- | | | | | |
|--------|--------|--------|--------|---------|
| 1. (a) | 2. (b) | 3. (c) | 4. (c) | 5. (d) |
| 6. (c) | 7. (d) | 8. (a) | 9. (d) | 10. (c) |

14.9 Further Reading

Notes



Books

Microprocessor 8085 and its Interfacing, By Mathur



Online link

<http://books.google.com/books?>

LOVELY PROFESSIONAL UNIVERSITY

Jalandhar-Delhi G.T. Road (NH-1)
Phagwara, Punjab (India)-144411
For Enquiry: +91-1824-521360
Fax.: +91-1824-506111
Email: odl@lpu.co.in

