Dear students,

Welcome to the world of Microprocessors!

You are about to learn, how these little chips became the central force behind the computer revolution. Brain child of Alan Turing, nurtured by genius minds like Von Neumann, Maurice Wilkes, and materialized by corporations like Intel, Samsung, Apple… these chips transformed the way the world computed.

From traffic signals to air traffic control, drones to satellites, fitness bands to pace makers, Microprocessors have brought to you, ever improving standards of health, travel, entertainment, communication etc.

In this book, you begin with learning the 8086 Architecture.
You then learn 8086 in depth including its Memory module, Instruction set, Programming and interfacing with its set of coprocessors and peripherals.

You will then proceed to learning powerful microprocessors like Pentium.
You will see the speed advantage achieved in Pentium using superscalar architecture and on chip cache memories. You will then witness the brilliance of Branch Prediction Algorithm which minimized the biggest drawback of pipelining.

Before we start, allow me to take the opportunity to thank my mother,
Prof. Veena D. Acharya.
A teacher all her life, she was the primary inspiration for me to pursue this noble profession.
Her blessings are reflected in the enthusiasm shown in this book and in my lectures.

Bharat D. Acharya.
B. E. Computer Science.
Founder, Bharat Acharya Education.
Teaching Microprocessors & Microcontrollers since 2000.

INTRODUCTION TO MICROPROCESSORS

Drones, stump vision cameras, mobile phones, RFID sensors, autonomous cars, streaming servers, your favorite shopping website… all of these are fruits of a seed called **"Microprocessor"**, planted years ago in mid 1970s and in fact conceived even earlier in 1940s.

So what does this Microprocessor (henceforth called µP) actually do… Why do we need to learn it… And most importantly, after completing our education, how will this knowledge be useful to us…

Where do we use a µP? Is a µP used in a fan, or in a tube light, or in a switch board? No, none of them. Is it used in a mobile phone, a computer, a microwave oven, a washing machine? Yes, all of them! This is because they run on programs, and all those programs are executed by the microprocessor within these devices. **This is the main function of a µP, to execute programs**.

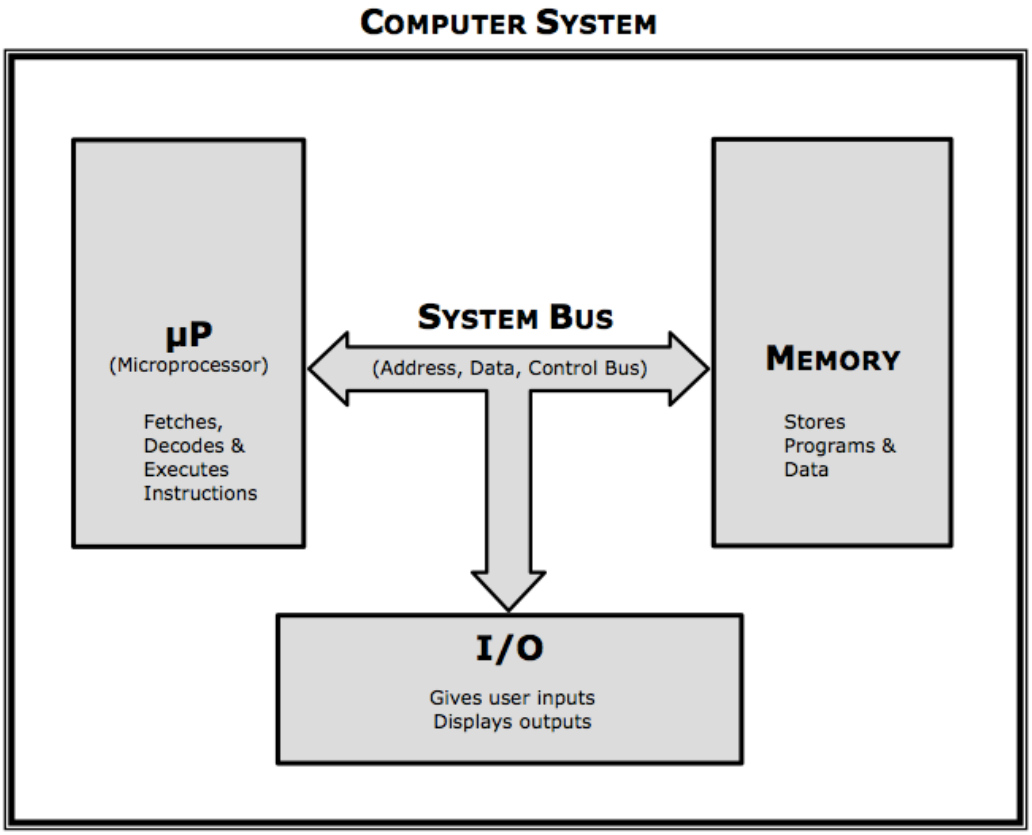
In our day to day encounters we come across several devices and appliances. If you feel any of them works on a program, you should most certainly realize, it must contain a microprocessor. Take a microwave oven as an example. The µP inside the oven isn't directly cooking the food. It is running programs that are responsible for rotating the dish, maintaining the correct temperature, counting the desired number of seconds, displaying the time remaining on the screen, and finally ringing the sweet alarm (ding!) informing us that the cooking is complete. All of these require programs, that are executed by the oven's µP. And who writes the programs? The engineer, yes that's you! It is this this combination of the engineers mind and the microprocessors execution abilities that has transformed the world in the past few decades and will continue to do so as both are getting ever so smart in the new age.

The simplest example of the use of a microprocessor, is in a **computer**. When you imagine a computer, lots of devices come to our mind, like the keyboard, mouse, printer, monitor and the big "box" also casually referred to by laymen as the CPU. Of course, you know their roles! The Keyboard and mouse are called input devices. Are they executing our programs? No! When we want to add two numbers, neither is the keyboard adding them nor is the mouse. So, are they required? Yes! To give inputs. That's their role. To provide inputs to the system. Similarly, the printer and the monitor are used to produce outputs. Hence these are called I/O devices (Input and Output devices). This leaves us with that big "box". Open it and you see an electric circuit board also called the motherboard. **On the motherboard, pretty much around the center lies a big chip, around the size of our palm, that's your µP**. You may notice in modern motherboards the µP is generally covered with a fan, to dissipate the heat generated by relentlessly performing millions and sometimes billions of operations per second. So how important is the µP? Well, remove it and our computer becomes a piece of junk! Every activity of your computer needs a program, lets get you in agreement with that. You watch a movie, play a song, surf the net, be on social media etc., all of these are programs. Executing them keeps the µP busy round the clock, hence the heat and the fan.

Let's say we are headed to the market right now, to buy the latest computer. Which microprocessor will you find inside? Yes, the Intel Core i7, or maybe Core i5 or i3, Intel also has recently released the Core i9 but its way too expensive to be mass produced as yet. So are we learning Core i7, i5, i3… not so early! These are a result of over 40 years of cutting-edge development making them way too advanced, to be understood by a beginner. We begin with learning the µP that started this whole x86 family… **The 8086 Microprocessor**.

BASIC ORGANIZATION OF A COMPUTER

A computer system, as we know it, consists of various components.
They can be broadly classified into three sections:
The Processor, Memory and I/O.



THE PROCESSOR – "µP"

The heart of the computer is its µP (Microprocessor). Current generation computers use processors like Intel Core i3, i5 or i7 and so on. They have come a long way from the initial processors that you are about to learn E.g.: 8085, 8086 etc.

Back in the day (1940s), when micro-electronics was not invented, processors looked very different and were certainly not "micro" in appearance. They were created using huge arrays of physical switches which were operated manually and often occupied large rooms.

In the following decades, with the invention of micro-electronics, scientists managed to embed thousands of microscopic switches (transistors) inside a small chip, and called it a "**Micro-processor**".

Over the years, microprocessors grew in strength. From housing a few thousand transistors (8085) to containing more than a billion transistors (Core i7), the computational power has been increasing exponentially. Having said that, some of the basics still remain the same.

To put it simply, **the main function of a µP is to Fetch, Decode and Execute instructions.**

Instructions are a part of programs. Programs are stored in the memory. First, **µP fetches an instruction** from the memory. It then **decodes the instruction**. This means, it "understands" the binary pattern of the instruction, also called its opcode. Every instruction when stored in the memory is in its unique binary form, which indicates the operation to be performed. **This is called its opcode**. Upon decoding the opcode, µP understands the operation to be performed and hence "executes" the instruction. This entire process is called an **"Instruction cycle"**. This process is repeated for the next instruction. Like this, one by one, all instructions of a program are executed.
Of course, by new concepts like **pipelining**, **multitasking**, **multiprocessing** etc., this procedure has become very advanced and efficient today. You will get to learn all of them, in the due course of this ever-intriguing subject.

We begin learning with basic processors like **8085** or **8086**, but make no mistake, none of this is "outdated". Yes, your mobile phone or your computer today uses the most advanced processors (Apple's **A12 Bionic** etc.), but to run a **traffic light** or **TV remote** control you don't need a core i7 now, do you? And these are used by the millions across the world. They simply use processors of the same grade as an 8085 or an 8086, with different product numbers as they are made by various manufacturers.

MEMORY

Memory is used to **store information**.
It stores two kinds of information… **programs and data**. Yes, think about it! Everything that's stored in your computer's memory is either some program or some data. MS Word is a program, the Word Documents are data. Your Image Viewer is a program, the images are data. WhatsApp is a program, its messages are data. Instagram is a program, the feed on the wall is the data… and so on!

All programs and data are stored in the memory, in digitized form, where every information is represented in 1s and 0s called binary digits or simply bits.

There are various forms of memory devices.
The main memory also called primary memory consists of Ram and ROM.
Other memory devices like Hard disk, Floppy, CD/ DVD etc. are secondary storage devices.

Additionally there is also a high speed memory called Cache composed of SRAM.
For the majority portion of this book, you are dealing with the initial processors like 8086.
It will be in your best interest to think of Primary Memory only, whenever we speak of memory. That is because, secondary memory and high speed memories were implemented much later in the evolution of processors as the demand for mass storage and high speed performance started increasing. So, from now on in this book, unless specified otherwise, **the word memory refers to primary memory that is RAM and ROM.**

The memory is a series of locations.
Each location is identified by its own unique address.
Every memory location contains 1 Byte (8 bits) of data. There is a very good reason for this, and you will learn it when we discuss the topic of memory banking in 8086.

I/O DEVICES

I/O devices are used to enter programs and data as inputs and display or print the results as outputs.
We are all familiar with devices such as the keyboard, mouse, printer, monitor etc. Every form of computer system has a set of I/O devices for human interaction. A device like a touch screen performs dual functions of both input and output.

The µP, Memory and I/O are all connected to each other using the System Bus.

SYSTEM BUS

A Bus is a set of lines. They are used to transfer information, obviously in binary form.
A line connected to Vcc will carry a logic 1 and if connected to Gnd it will carry logic 0.
Hence one line can transfer 1 bit. If we want multiple bits to be transmitted together, then we use a set of lines grouped together and that's called a bus.

The size of a bus refers to the number of lines it contains and is always given in terms of bits.
E.g.:: An 8-bit bus has 8 lines, a 16 bit bus has 16 lines and so on.

There are three types of busses… Address, Data and Control Bus. Collectively they are called the system bus. Lets take a closer look at them.

i) ADDRESS BUS

It carries the address where the operation has to be performed. Say the processor wants to write some data at a memory location. Firstly the processor will give the desired address on the address bus. This address will select a unique memory location. That's when data will be transferred with that location.
It is therefore obvious that **bigger the address bus, more is the number of memory locations** it can address and hence bigger the total size of the memory. Here is the relation between size of the address bus and size of the memory.
An address bus of 1 bit can give a total of two addresses: 0 and 1.
Hence can access a total memory of two locations.
A 2-bit address bus can generate a total of 4 addresses 00, 01, 10, 11 and hence can access a total of 4 locations.
3-bit address bus… 8 locations and so on.

So here is the rule,

An N-bit address bus can totally access 2^N locations.

As mentioned earlier, one memory location contains one byte of data.
This brings us to the following conclusion.
A processor with an N-bit address bus can access a memory of 2^N Bytes.

Lets solve a typical VIVA (oral exam) question.

Given the size of address bus, you have to figure out the size of the memory.

ADDRESS BUS (N - BIT)	MEMORY (2^N BYTES)
4 – bit	2^4 = 16 Bytes
6 – bit	2^6 = 64 Bytes
10 – bit	2^{10} = 1024 Bytes = 1 KB
16 – bit	$2^{16} = 2^6 \times 2^{10}$ = 64 x 1 K = 64 KB … (8085)
20 – bit	$2^{20} = 2^{10} \times 2^{10}$ = 1 K x 1 K = 1 MB … (8086)
30 – bit	$2^{30} = 2^{10} \times 2^{20}$ = 1 K x 1 M = 1 GB
32 – bit	$2^{32} = 2^2 \times 2^{30}$ = 4 x 1 G = 4 GB …. (80386 and Pentium)
40 – bit	$2^{40} = 2^{10} \times 2^{30}$ = 1K x 1G = 1 TB … (Typical Hard Disk)

In case you found it a little difficult after the 4th row, that is because you may have got a little confused with the powers of 2. This issue will persist all along the subject. The smarter thing to do is once for all, lets get this hurdle past us. Lets get all powers of 2 clearly sorted. You will realize in the due course of this subject, how helpful this small exercise will prove to be. Frankly speaking there's rarely a topic in this subject that is not connected with some power of 2. Lets sort this, totally!

Powers of 2

2^N	VALUE
2^0	0
2^1	2
2^2	4
2^3	8
2^4	16
2^5	32
2^6	64
2^7	128
2^8	256
2^9	512
2^{10}	1024 = 1K
2^{20}	$2^{10} \times 2^{10}$ = 1K x 1K = 1M
2^{24}	$2^4 \times 2^{20}$ = 16 x 1M = 16M
2^{28}	$2^8 \times 2^{20}$ = 256 x 1M = 256M
2^{30}	$2^{10} \times 2^{20}$ = 1K x 1M = 1G
2^{36}	$2^6 \times 2^{30}$ = 64 x 1G = 64G
2^{40}	$2^{10} \times 2^{30}$ = 1K x 1G = 1T

Similarly, there is one more very important fundamental that needs to be sorted out before we start learning the bigger concepts. You need to be sure of number representation and number conversions. You may have been familiar with various number systems like Decimal, Hexadecimal, Binary etc… Out of all of them, Hexadecimal and Binary are the two most widely used number systems in our course of learning this subject.

Every number we write, either in programs or in theory examples, is a hexadecimal number. When this number gets stored inside the computer, it is in binary form This simply means, you must be very well versed with hex-binary conversions as this will be needed while understanding various examples.

A single hexadecimal digit ranges from 0H… FH. This has 16 options. Hence, to represent the number in binary we need 4 bits as 2^4 =16. The following table shows this conversion.

HEX	BINARY
0 H	0000
1 H	0001
2 H	0010
3 H	0011
4 H	0100
5 H	0101
6 H	0110
7 H	0111
8 H	1000
9 H	1001
A H	1010
B H	1011
C H	1100
D H	1101
E H	1110
F H	1111

NO! You do not need to mug this up. There is a simple trick of “8421” that can get you any value from the above table. Consider the number 5 which is basically 4+1. Now consider the four binary bits corresponding to values 8, 4, 2 and 1. Since we need the equivalent of 5 which is 4 plus 1, we need 4 and 1 but we don’t need 8 and 2 so in positions of 8 and 2 we put a “0” and in positions of 4 and 1 we put a “1”. So we get 0101. Lets take the example of 0 H. We don’t need any of them (8 or 4 or 2 or 1). So we put a “0” for all of them and hence get 0000. If we need F (which is 15), it is 8 + 4 + 2 + 1 so we need all of them and hence the binary equivalent is 1111. Take 9 as an example, 9 = 8 + 1. So we need 8 and 1 but not 4 and 2 so we put a “1” for 8 and 1 positions and a “0” for 4 and 2 positions giving us 1001. Hope you can now convert any hex digit into binary.

Now consider a two digit number like 25H. To convert this to binary we need to substitute the 4 bit equivalents of 2 and 5 respectively. 2H is 0010 and 5H is 0101.

Hence the number 25H in binary will be 0010 0101.

Similarly a number like 74H will be 0111 0100 in binary. 89H will be 1000 1001 and so on.

As you noticed, the numbers 25H, 74H and 89H are all 2 digits in hexadecimal and hence need 8 bits in binary. Such numbers are called 8 bit numbers. The range of 8 bit and 16 bit numbers is mentioned below:

8 BIT NUMBERS (ALSO CALLED A "BYTE")

HEX	BINARY
00 H	0000 0000
01 H	0000 0001
...	...
68 H	0110 1000
...	...
FE H	1111 1110
FF H	1111 1111

16 BIT NUMBERS (ALSO CALLED A "WORD")

HEX	BINARY
0000 H	0000 0000 0000 0000
0001 H	0000 0000 0000 0001
...	...
4831 H	0100 1000 0011 0001
...	...
FFFE H	1111 1111 1111 1110
FFFF H	1111 1111 1111 1111

ii) DATA BUS

It carries data to and from the processor.
The size of data bus determines how much data can be transferred in one operation (cycle).
Bigger the data bus, faster the processor, as it can transfer more data in one cycle.

iii) CONTROL BUS

It Carries control signals like RD, WR etc.
These signals determine the kind of operation that will be performed on the system bus.

Bharat Acharya Education
Learn...
8085 | 8086 | 80386 | Pentium |
8051 | ARM7 | COA

Fees: 1199/-
Duration: 6 months
Activation: Immediate
Certification: Yes
Free: PDFs of theory explanation
Free: VIVA questions and answers
Free: PDF of Multiple Choice Questions

Start Learning... NOW!
Bharat Acharya Education

Order our Books here...

8086 Microprocessor book
Link: https://amzn.to/3qHDpJH

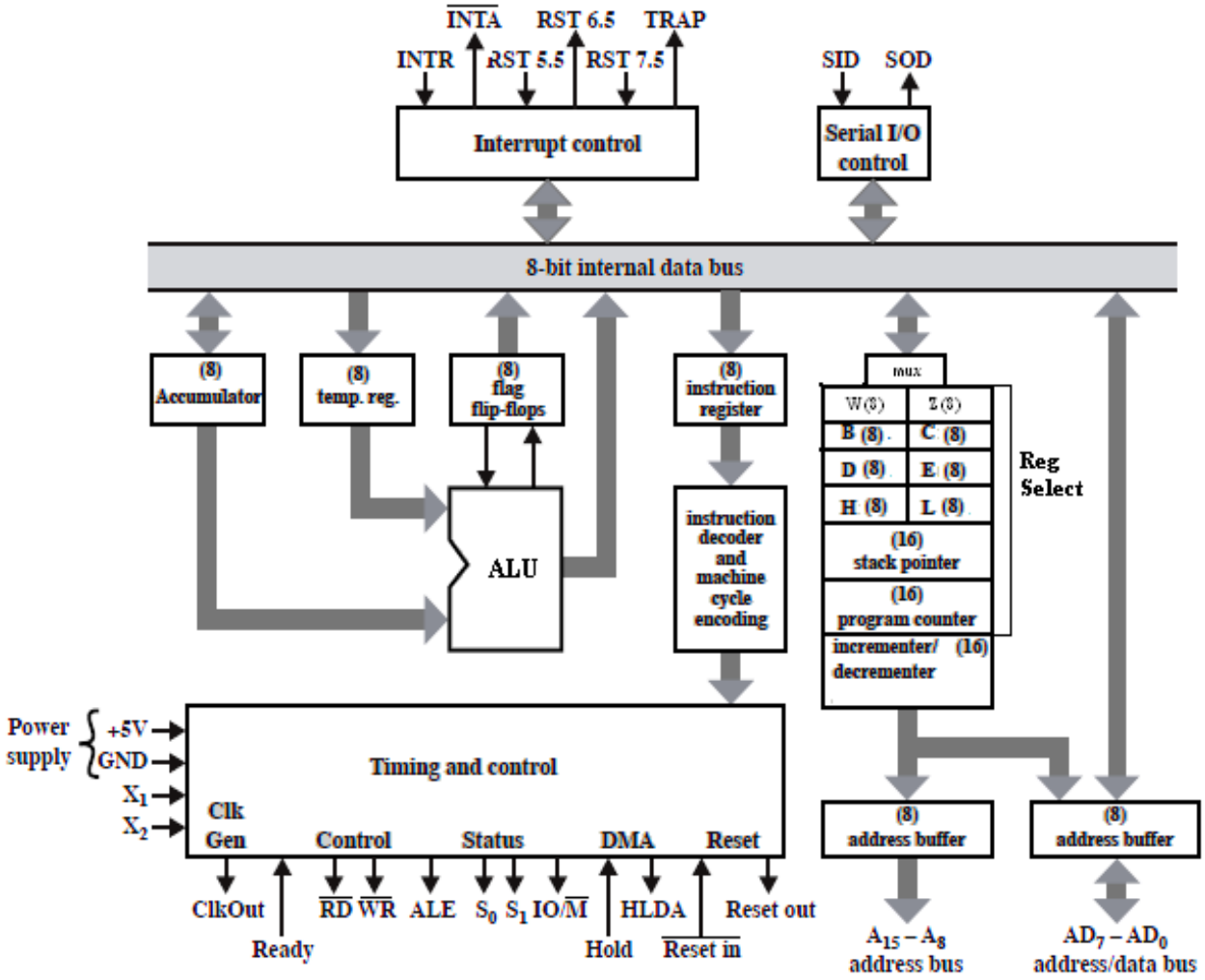
8051 Microcontroller book
Link: https://amzn.to/3aFQkXc

Official WhatsApp number:
+91 9136428051

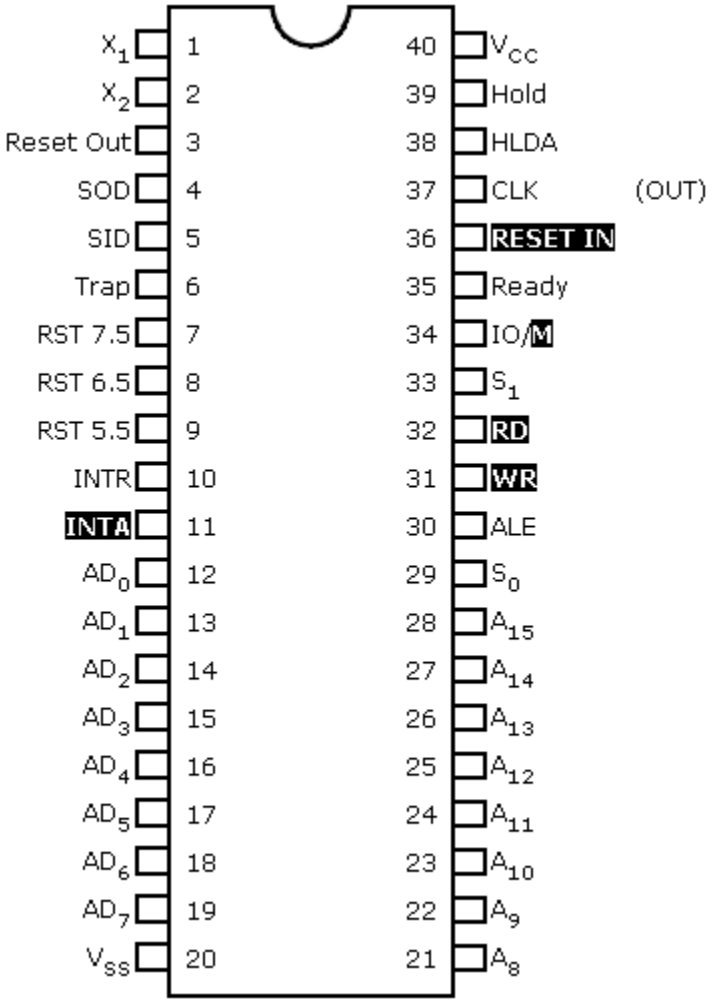
8085 ARCHITECTURE, PINS AND FLAG REGISTER

Note: Dear Students, Architecture contains all the pins and also the Flag Register. Hence I have made a common PDF for all the three topics so that you get the complete picture.

ARCHITECTURE OF 8085



PIN CONFIGURATION OF 8085



Registers

Program Counter (PC, 16-bits):

It is a 16-bit Special-Purpose register. It holds **address** of the **next instruction**.
PC is incremented by the INR/DCR after every instruction byte is fetched.

Stack Pointer (SP, 16-bits):

It is a 16-bit Special-Purpose register. It holds **address** of the **top of the Stack**.
Stack is a set of memory locations operating in LIFO manner.
SP is **decremented** on every **PUSH** operation and **incremented** on every **POP**.

B, C, D, E, H, L registers 8-bits each:

These are 8-bit General-Purpose registers**.**
They can also be used to store 16-bit data in register pairs.
The possible register **pairs** are **BC** pair, **DE** pair and **HL** pair.
The **HL** pair also holds the **address** for the Memory Pointer **"M"**.

Temporary Registers (WZ, 16-bits):

This is a 16-bit register pair.
It is **used by µP** to hold **temporary** values in some instructions like CALL/JMP/LDA etc.The **programmer** has **no access** to this register pair.

INR/DCR Register (16-bits):

This is a 16-bit shift register.
It is used to **increment PC after every instruction byte is fetched** and **increment** or **decrement SP** after a Pop or a Push operation respectively.
It is not available to the programmer.

A - Accumulator (8-bits):

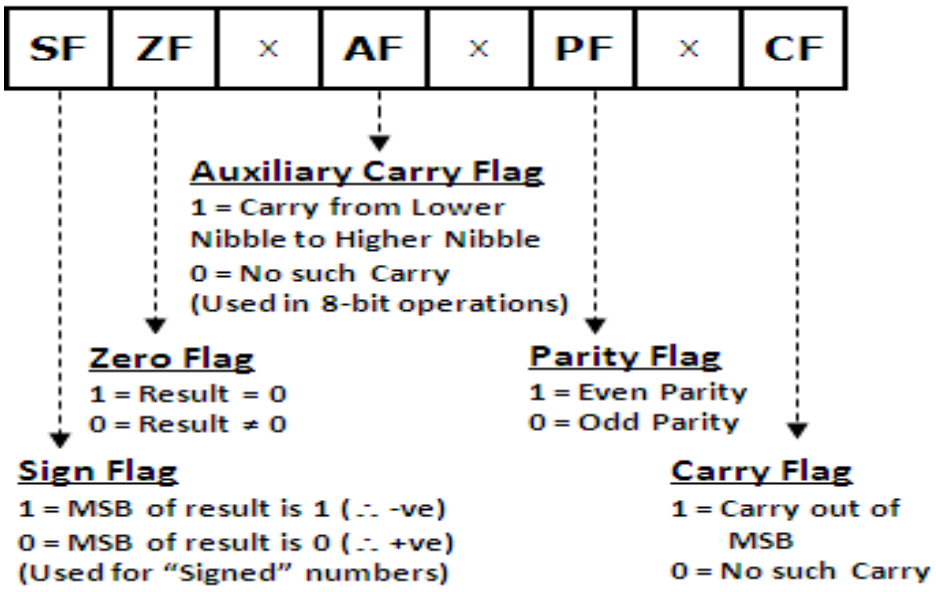
It is an 8-bit programmable register.The user can read or write this register.
It has two **special properties** viz:

- It **holds one** of the **operands** during most of the arithmetic operations.
- It **holds** the **result** of most of the arithmetic and logic operations

Temp Register (8-bits):

This is an 8-bit register.
It is **used by µP** for storing one of the operands during an operation.
The **programmer** has **NO ACCESS** to this register.

Flag Register



S - Sign Flag:

It is **set** (1) when **MSB** of the result is **1** (i.e. result is a **-**VE number).
It is reset (0) when MSB of the result is 0 (i.e. result is a **+**VE number).

Z - Zero Flag:

It is **set** when the **result** is **= zero**.
It is reset when the result is not = zero.

AC - Auxiliary Carry Flag:

It is **set** when an **Auxiliary Carry** / Borrow is **generated**.
It is reset when an Auxiliary Carry / Borrow is not generated.
Auxiliary Carry is the Carry generated **between** the **lower nibble and the higher nibble** for an 8-bit operation.
It is not affected after a 16- bit operation. It is used only in DAA operation.

P - Parity Flag:

It is **set (1)** when result has **even parity**. It is reset when result has odd parity.

C - Carry Flag:

It is **set** when a **Carry / Borrow** is **generated from the MSB**.
It is reset when a Carry / Borrow is not generated from the MSB.

 # In the exam, Show at least 2 examples from Bharat Sir's lecture notes

Interrupt Control

This Block is responsible for controlling the **hardware interrupts** of 8085.
8085 supports the following hardware interrupts:

TRAP:

This is an **edge as well as level triggered**, **vectored** interrupt.
It cannot be masked by SIM instruction and can neither be disabled by DI instruction.
It has the **highest priority**.
Its vector address is **0024H**.

RST 7.5:

This is an **edge triggered**, **vectored** interrupt.
It can be masked by SIM instruction and can also be disabled by DI instruction.
It has the **second highest priority**.
Its vector address is **003CH**.

RST 6.5:

This is a **level triggered**, **vectored** interrupt.
It can be masked by SIM instruction and can also be disabled by DI instruction.
It has the **third highest priority**.
Its vector address is **0034H**.

RST 5.5:

This is a **level triggered**, **vectored** interrupt.
It can be masked by SIM instruction and can also be disabled by DI instruction.
It has the **fourth highest priority**.
Its vector address is **002CH**.

INTR:

This is a **level triggered**, **non-vectored** interrupt.
It cannot be masked by SIM instruction but can be disabled by DI instruction.
It has the **lowest priority**.
It has an **acknowledgement signal $\overline{\text{INTA}}$** .
The address for the ISR is **fetched from external hardware**.

$\overline{\text{INTA}}$:

This is an **acknowledgement** signal **for INTR** (only).
This signal is used to **get** the Op-Code (and hence the ISR address) from External hardware in order to execute the ISR. ☺ In case of doubts, contact Bharat Sir: - 98204 08217.

ALL Interrupts **EXCEPT TRAP** can be **disabled** though the **DI** instruction.
These interrupts can be **enabled** again by the **EI** Instruction.
Interrupts can be individually **masked or unmasked by SIM instruction**.
TRAP and INTR are not affected by SIM instruction.

Serial Control

This Block is responsible for transferring data Serially to and from the µP.

SID - Serial In Data:
µP receives data, bit-by-bit through this line.

SOD - Serial Out Data:
µP sends out data, bit-by-bit through this line.
Serial transmission can be done by **RIM** and **SIM** Instructions.

ALU – Arithmetic Logic Unit

8085 has an **8-bit ALU**.
It performs 8-bit arithmetic operations like Addition and Subtraction.
It also performs logical operations like AND, OR, EX-OR NOT etc.
It takes **input** from the **Accumulator** and the **Temp** register.
The **output** of most of the ALU operations is stored back **into** the **Accumulator**.

Instruction Register and Decoder

Instruction Register:
The 8085 places the contents of the PC onto the Address bus and fetches the instruction.
This fetched instruction is stored into the Instruction register.

Instruction Decoder:
The fetched instruction from the Instruction register enters the Instruction Decoder.
Here the instruction is decoded and the decode information is given to the Timing and Control Circuit where the instruction is executed..

Timing and Control Circuit

The timing and control circuit issues the various internal and external control signals for executing and instruction.
The external pins connected to this circuit are as follows:

X1 and X2:
These pins provide the **Clock Input to the µP**.
Clock is provided from a crystal oscillator.

ClkOut:
8085 provides the **Clock input** to all **other peripherals** through the ClockOut pin.
This takes care of **synchronizing** all peripherals with 8085.

$\overline{\text{ResetIn}}$:
This is an active low signal activated when the manual reset signal is applied to the µP. This signal **resets the µP**. On Reset PC contains **0000H**. Hence, the **Reset Vector Address** of 8085 is 0000H.

ResetOut:
This signal is connected to the reset input of all the peripherals.

It is used to **reset the peripherals once** the **µP** is **reset**.

READY:

This is an active high input.
It is used to **synchronize** the µP with **"Slower"** Peripherals.
The **µP samples** the **Ready** input in the beginning of every Machine Cycle.
If it is found to be **LOW**, the **µP executes** one **WAIT CYCLE** after which it re-samples the ready pin till it finds the Ready pin HIGH.
∴ The **µP remains** in the **WAIT STATE until** the **READY** pin becomes **high** again.
Hence, **if** the **Ready** pin is **not required** it should be **connected** to the **Vcc,** and not, left unconnected, **otherwise** would cause the µP to execute **infinite wait cycles**. *#Please refer Bharat Sir's Lecture Notes for this ...*

ALE - Address Latch Enable:

This signal is **used to latch address** from the multiplexed Address-Data Bus (**AD0-AD7**). When the Bus contains **address**, **ALE** is **high**, **else** it is **low**.

IO/ $\overline{M}$:

This signal is used to distinguish between an **IO** and a **Memory operation**.
When this signal is high it is an IO operation else it is a Memory operation.

$\overline{RD}$:

This is an active low signal used to indicate a **read operation**.

$\overline{WR}$:

This is an active low signal used to indicate a **write operation**.

S_1 and S_0:

These lines denote the status of the µP

S_1 S_0	Status
0 0	Idle
0 1	Write
1 0	Read
1 1	Opcode fetch

HOLD and HLDA:

The Hold and Hold Acknowledge signals are used for **Direct Memory Access** (DMA).
The **DMA Controller issued** the **Hold** signal to the µP.
In response the **µP releases** the **System bus**.
After releasing the system bus the **µP acknowledges** the Hold signal with **HLDA** signal.
The **DMA Transfer** thus **begins**.
DMA **Transfer** is **terminated** by **releasing** the **HOLD** signal.

Bharat Acharya Education
Learn...
8085 | 8086 | 80386 | Pentium |
8051 | ARM7 | COA

Fees: 1199/-
Duration: 6 months
Activation: Immediate
Certification: Yes
Free: PDFs of theory explanation
Free: VIVA questions and answers
Free: PDF of Multiple Choice Questions

Start Learning... NOW!
Bharat Acharya Education

Order our Books here...

8086 Microprocessor book
Link: https://amzn.to/3qHDpJH

8051 Microcontroller book
Link: https://amzn.to/3aFQkXc

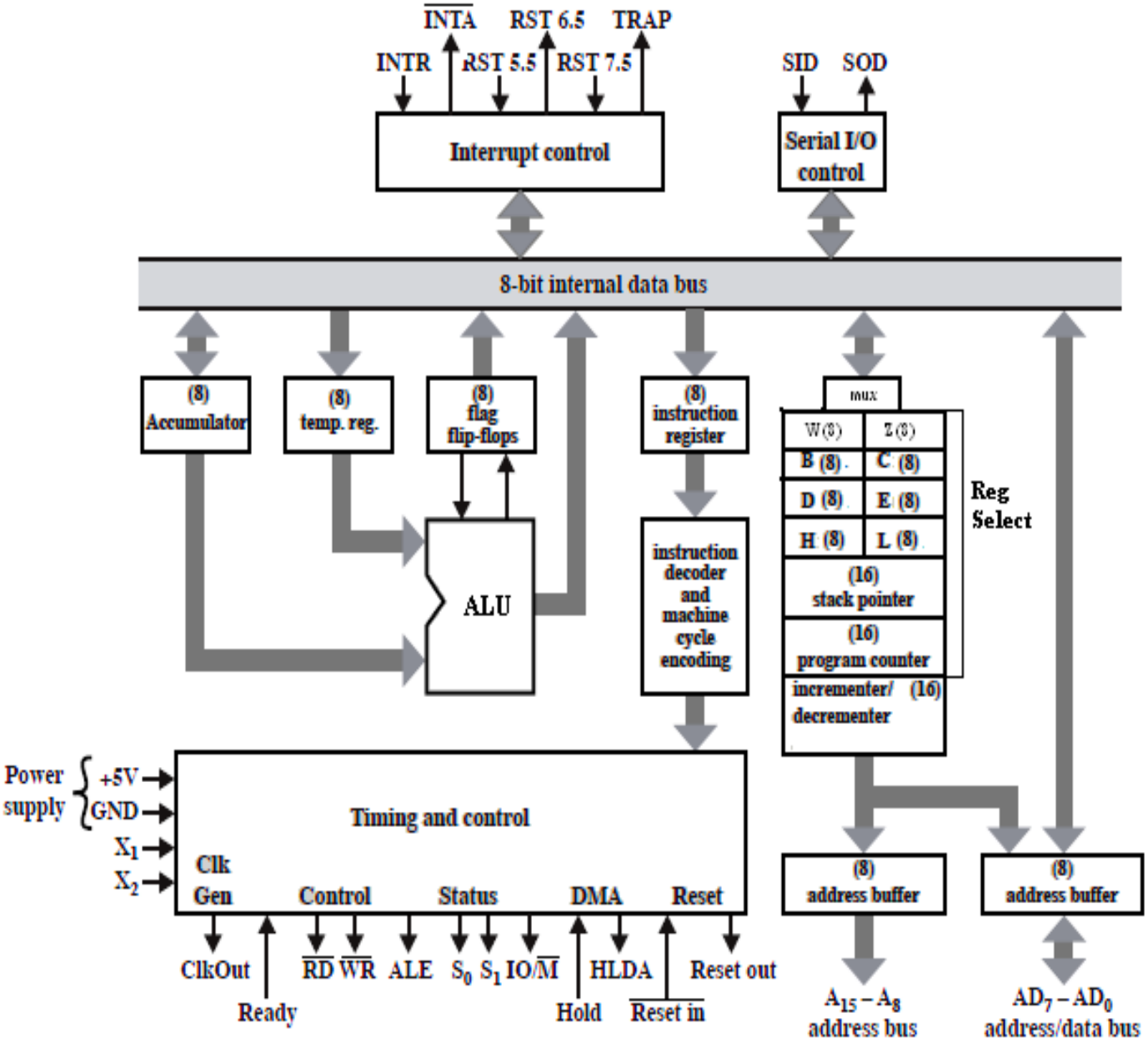
Official WhatsApp number:
+91 9136428051

8085 ARCHITECTURE, PINS AND FLAG REGISTER

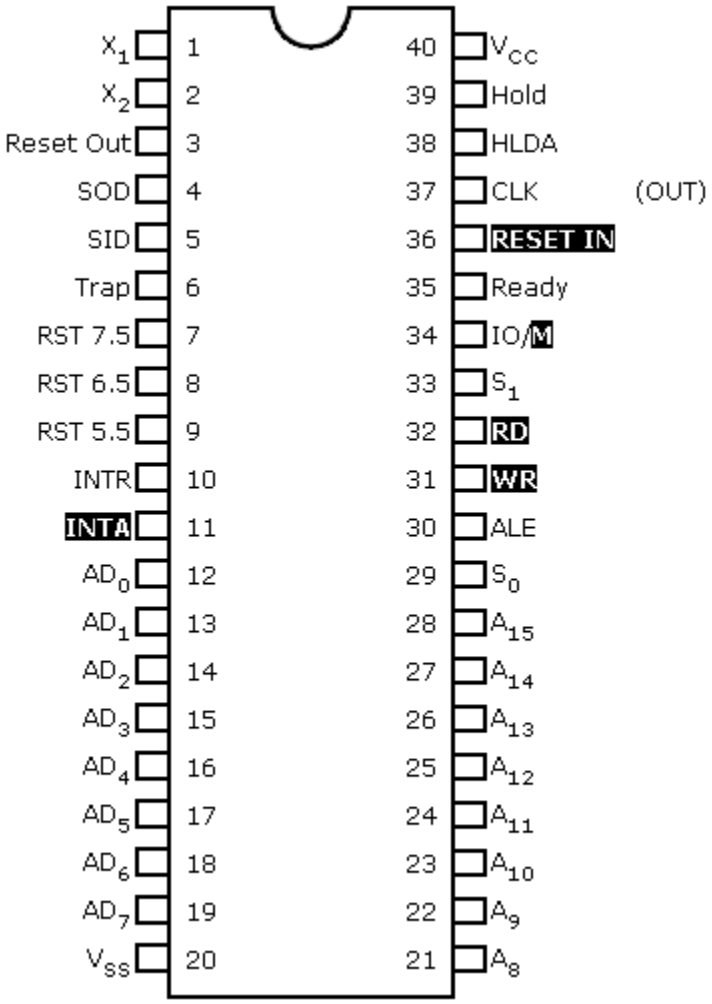
Note

Dear Students, Architecture contains all the pins and the Flag Register. Hence, I have made a common PDF for all these three topics.

8085 ARCHITECTURE



8085 PIN DIAGRAM



REGISTERS

Program Counter (PC, 16-bits)

It is a 16-bit Special-Purpose register. It holds **address** of the **next instruction**.
PC is incremented by the INR/DCR after every instruction byte is fetched.

Stack Pointer (SP, 16-bits)

It is a 16-bit Special-Purpose register. It holds **address** of the **top of the Stack**.
Stack is a set of memory locations operating in LIFO manner.
SP is **decremented** on every **PUSH** operation and **incremented** on every **POP**.

B, C, D, E, H, L (8-bits)

These are 8-bit General-Purpose registers.
They can also be used to store 16-bit data in register pairs.
The possible register **pairs** are **BC** pair, **DE** pair and **HL** pair.
The **HL** pair also holds the **address** for the Memory Pointer **"M"**.

Temporary Register Pair (WZ, 16-bits)

This is a 16-bit register pair.
It is **used by µP** to hold **temporary** values in some instructions like CALL/JMP/LDA etc.
The **programmer** has **no access** to this register pair.

INR/DCR Register (16-bits)

This is a 16-bit shift register.
It is used to **increment PC after every instruction byte is fetched**
It also **increments** or **decrements SP** after a Pop or a Push operation respectively.
It is not available to the programmer.

A - Accumulator (8-bits)

It is an 8-bit programmable register.
The user can read or write this register.
It has two **special properties** viz:

- It **holds the first operand** during most arithmetic operations.
- It **holds** the **result** of most of the arithmetic and logic operations
  Eg: ADD B; This instruction will do A + B and store the result in A.
  Eg: SUB B; This instruction will do A - B and store the result in A.

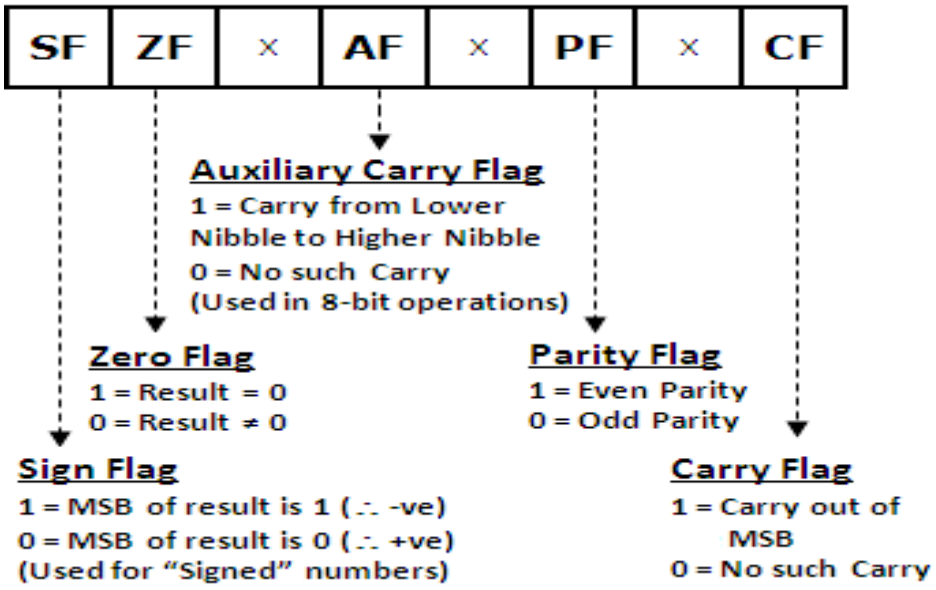
Temp Register (8-bits)

This is an 8-bit register.
It is **used by µP** for storing one of the operands during an operation.
The **programmer** has **NO ACCESS** to this register.

Special Note:

If you are learning this by piracy, then you are not my student.
You are simply a thief! #PoorUpbringing

8085 FLAG REGISTER



S - Sign Flag

It is **set** (1) when **MSB** of the result is **1** (i.e. result is a **-**VE number).
It is reset (0) when MSB of the result is 0 (i.e. result is a **+**VE number).

Z - Zero Flag

It is **set** when the **result** is **= zero**.
It is reset when the result is not = zero.

AC - Auxiliary Carry Flag

It is **set** when an **Auxiliary Carry** / Borrow is **generated**.
It is reset when an Auxiliary Carry / Borrow is not generated.
Auxiliary Carry is the Carry generated **between** the **lower nibble and the higher nibble** for an 8-bit operation. It is not affected after a 16- bit operation. It is used only in DAA operation.

P - Parity Flag

It is **set (1)** when result has **even parity**. It is reset when result has odd parity.

C - Carry Flag

It is **set** when a **Carry / Borrow** is **generated from the MSB**.
It is reset when a Carry / Borrow is not generated from the MSB.

 # In the exam, Show at least 2 examples from Bharat Sir's video lecture

INTERRUPT CONTROL

This Block is responsible for controlling the **hardware interrupts** of 8085.
8085 supports the following hardware interrupts:

TRAP

This is an **edge as well as level triggered**, **vectored** interrupt.
It cannot be masked by SIM instruction and can neither be disabled by DI instruction.
It has the **highest priority**.
Its vector address is **0024H**.

RST 7.5

This is an **edge triggered**, **vectored** interrupt.
It can be masked by SIM instruction and can also be disabled by DI instruction.
It has the **second highest priority**.
Its vector address is **003CH**.

RST 6.5

This is a **level triggered**, **vectored** interrupt.
It can be masked by SIM instruction and can also be disabled by DI instruction.
It has the **third highest priority**.
Its vector address is **0034H**.

RST 5.5

This is a **level triggered**, **vectored** interrupt.
It can be masked by SIM instruction and can also be disabled by DI instruction.
It has the **fourth highest priority**.
Its vector address is **002CH**.

INTR

This is a **level triggered**, **non-vectored** interrupt.
It cannot be masked by SIM instruction but can be disabled by DI instruction.
It has the **lowest priority**.
It has an **acknowledgement signal $\overline{\text{INTA}}$** .
The address for the ISR is **fetched from external hardware**.

$\overline{\text{INTA}}$

This is an **acknowledgement** signal **for INTR** (only).
This signal is used to **get** the Op-Code (and hence the ISR address) from External hardware in order to execute the ISR. ☺ In case of doubts, contact Bharat Sir: - 98204 08217.

ALL Interrupts **EXCEPT TRAP** can be **disabled** though the **DI** instruction.
These interrupts can be **enabled** again by the **EI** Instruction.
Interrupts can be individually **masked or unmasked by SIM instruction**.
TRAP and INTR are not affected by SIM instruction.

SERIAL CONTROL

This Block is responsible for transferring data Serially to and from the µP.

SID - Serial In Data

µP receives data, bit-by-bit through this line.

SOD - Serial Out Data

µP sends out data, bit-by-bit through this line.
Serial transmission can be done by **RIM** and **SIM** Instructions.

ALU

8085 has an **8-bit ALU**.
It performs 8-bit arithmetic operations like Addition and Subtraction.
It also performs logical operations like AND, OR, EX-OR NOT etc.
It takes **input** from the **Accumulator** and the **Temp** register.
The **output** of most of the ALU operations is stored back **into** the **Accumulator**.

INSTRUCTION REGISTER AND DECODER

Instruction Register

The 8085 places the contents of the PC onto the Address bus and fetches the instruction.
This fetched instruction is stored into the Instruction register.

Instruction Decoder:

The fetched instruction from the Instruction register enters the Instruction Decoder. Here the instruction is decoded and the decode information is given to the Timing and Control Circuit where the instruction is executed.

TIMING AND CONTROL CIRCUIT

The timing and control circuit issues the various internal and external control signals for executing and instruction.
The external pins connected to this circuit are as follows:

X1 and X2

These pins provide the **Clock Input to the µP**.
Clock is provided from a crystal oscillator.

ClkOut

8085 provides the **Clock input** to all **other peripherals** through the ClockOut pin. This takes care of **synchronizing** all peripherals with 8085.

$\overline{\text{ResetIn}}$

This is an active low signal activated when the manual reset signal is applied to the µP. This signal **resets the µP**. On Reset PC contains **0000H**. Hence, the **Reset Vector Address** of 8085 is 0000H.

ResetOut

This signal is connected to the reset input of all the peripherals.
It is used to **reset the peripherals once** the **µP** is **reset**.

READY

This is an active high input.
It is used to **synchronize** the µP with **"Slower"** Peripherals.
The **µP samples** the **Ready** input in the beginning of every Machine Cycle.
If it is found to be **LOW**, the **µP executes** one **WAIT CYCLE** after which it re-samples the ready pin till it finds the Ready pin HIGH.
∴ The **µP remains** in the **WAIT STATE until** the **READY** pin becomes **high** again.
Hence, **if** the **Ready** pin is **not required** it should be **connected** to the **Vcc,** and not, left unconnected, **otherwise** would cause the µP to execute **infinite wait cycles**.
#Please refer Bharat Sir's video Lecture for this ...

ALE - Address Latch Enable

This signal is **used to latch address** from the multiplexed Address-Data Bus (**AD0-AD7**). When the Bus contains **address**, **ALE** is **high**, **else** it is **low**.

IO/ $\overline{M}$

This signal is used to distinguish between an **IO** and a **Memory operation**.
When this signal is high it is an IO operation else it is a Memory operation.

$\overline{RD}$

This is an active low signal used to indicate a **read operation**.

$\overline{WR}$

This is an active low signal used to indicate a **write operation**.

S_1 and S_0

These lines denote the status of the µP

S_1	S_0	Status
0	0	Idle
0	1	Write
1	0	Read
1	1	Opcode fetch

HOLD and HLDA

The Hold and Hold Acknowledge signals are used for **Direct Memory Access** (DMA).
The **DMA Controller issued** the **Hold** signal to the µP.
In response the **µP releases** the **System bus**.
After releasing the system bus the **µP acknowledges** the Hold signal with **HLDA** signal. The **DMA Transfer** thus **begins**.
DMA **Transfer** is **terminated** by **releasing** the **HOLD** signal.

Bharat Acharya Education
Learn...
8085 | 8086 | 80386 | Pentium |
8051 | ARM7 | COA

Fees: 1199/-
Duration: 6 months
Activation: Immediate
Certification: Yes
Free: PDFs of theory explanation
Free: VIVA questions and answers
Free: PDF of Multiple Choice Questions

Start Learning... NOW!
Bharat Acharya Education

Order our Books here...

8086 Microprocessor book
Link: https://amzn.to/3qHDpJH

8051 Microcontroller book
Link: https://amzn.to/3aFQkXc

Official WhatsApp number:
+91 9136428051

8085 ADDRESSING MODES

Addressing Modes are the different ways by which the µP address (specifies) the operands in an instruction. 8085 supports the following Addressing Modes:

1) Immediate Addressing Mode

In this mode, the **Data** is specified **in** the **Instruction** itself.

```
Eg:   MVI A, 35H      ; Move immediately the value 35 into the Accumulator.
                      ; i.e. A ← 35H
      LXI B, 4000H    ; Move immediately the value 4000 into the register pair BC.
                      ; i.e. BC ← 4000H
```

Advantage:
Programmer can easily **identify** the **operands**.

Disadvantage:
Always more than one byte hence requires **more space**.
The µP requires **two or three machine cycles** to fetch the instruction hence **slow**.

Special Notes:
The "I" in the instruction indicates "Immediate Addressing Mode"
Hence the number in the instruction must be DATA.
Hereafter, when you see a number in any instruction look for an "I".
If "I" is present then the number is DATA else its an address.
The "X" in the instruction (LXI) indicates "Register Pair".

2) Register Addressing Mode

In this mode, the **Data** is specified **in Registers**.

```
Eg:   MOV B, C        ; Move the Contents of C-Register into B-Register.
                      ; i.e. B ← C
      INR B           ; Increments the contents of B-Register.
                      ; i.e. B ← B + 1
```

Advantage:
The µP requires **only one machine cycle** to Fetch the instruction.

Disadvantage:
Operands **cannot** be easily **identified**.

3) Direct Addressing Mode

In this mode, the **Address** of the operand is specified **in** the **Instruction** itself.

Eg: **LDA 2000H** ; Loads the Accumulator with the Contents of Location 2000.
; i.e. A ← [2000]
STA 2000H ; Stores the Contents of the Accumulator at the Location 2000.
; i.e. [2000] ← A

Advantage:
The programmer **can identify** the address of the operand.

Disadvantage:
These are **three byte instructions** hence require three fetch cycles.

4) Indirect Addressing Mode

In this mode, the **Address** of the operand is specified **in Registers.**
Hence, the instruction indirectly points to the operands.
Even the Memory Pointer "M" can be used as it is pointed by the HL register pair.

Eg: **STAX B** ; Stores the contents of the Accumulator at the location
; pointed by the contents of BC pair.
; i.e. [[BC]] ← A.
; So if contents of BC pair = 4000 i.e. [BC] = 4000 then
; [4000] ← A. *#Please refer Bharat Sir's Lecture Notes for this ...*
INR M ; Increments the contents of the location pointed by HL pair
; (i.e. M) i.e. [[HL]] ← [[HL]] + 1

Advantage:
Address of the operand is **not fixed** and hence can be used in a **loop**.
Size of the instruction is **small** as compared to direct addressing mode.

Disadvantage:
Requires initialization of the register pair hence requires atleast one more instruction.

Special Notes:
Remember, during programming when you want to access only 1 or 2 locations, use Direct addressing mode as it is simpler.
But when you want to access a series of locations, use Indirect addressing mode.
Initialize the first address in a register pair.
Thereafter increment/decrement that pair in a loop to access a series of locations.

5) Implied Addressing Mode

In this mode, the **Operand** is **implied** in the instruction.
This instruction will work only on that implied operand, and not on any other operand.

Eg: **STC** ; Sets the Carry Flag in the Flag register.
; Cy ← 1.
CMC ; Complements the Carry Flag in the Flag register.

Advantage:
Instructions are generally **only one byte**.

Disadvantage:
Programmer **cannot** easily **identify** the value of the operand.

Bharat Acharya Education
Learn...
8085 | 8086 | 80386 | Pentium |
8051 | ARM7 | COA

Fees: 1199/-
Duration: 6 months
Activation: Immediate
Certification: Yes
Free: PDFs of theory explanation
Free: VIVA questions and answers
Free: PDF of Multiple Choice Questions

Start Learning... NOW!
Bharat Acharya Education

Order our Books here...

8086 Microprocessor book
Link: https://amzn.to/3qHDpJH

8051 Microcontroller book
Link: https://amzn.to/3aFQkXc

Official WhatsApp number:
+91 9136428051

8085 CLASSIFICATION OF INSTRUCTIONS

The Instruction Set of 8085 is classified into the following 5 groups:

1) Data Transfer Group

This group of instructions transfer data from one place to another place.
The data can be transferred from Register to Register, Memory to Register & Register to Memory

Eg: **MOV A, B** ; Content of B register is transferred to accumulator.
MVI A, 03H ; Move immediately 03 into the accumulator.

2) Arithmetic Group

This group of instructions performs arithmetic operation on the data.
The arithmetic operation may be data addition, subtraction, increment, decrement etc.

Eg: **ADI 03H** ; Add immediately 03 with the content of accumulator, store
; result in accumulator.
SUB B ; Subtract content of B Register from the accumulator, store
; result in accumulator.

3) Logic Group

This group of instructions performs logical operations on the data.
These instructions include AND, OR, EX-OR etc.

Eg: **ANI 03H** ; AND logically 03 with the content of accumulator, store result
; in accumulator.
ORA B ; OR logically content of B Register with the accumulator, store
; result in accumulator.

4) Branch Group

These instructions change the sequence of flow of the program.
There are two types of Branch Instructions

Conditional Branch instructions
In this type, control is transferred to another memory location only if a particular condition is satisfied.
Eg: **JNZ 4000** ; Program Control is transferred to memory location 4000 provided result of
; the previous operation is non-zero else control continues sequentially.

Unconditional Branch instruction
In this type, control is transferred to another memory location without any condition.
Eg: **JMP 4000** ; Simply control is transferred to the memory location 4000.

5) Stack, IO and Machine Control Instruction

These instructions control the STACK (Push and Pop), I/O (Input and Output) and Machine hardware.

Eg: **PUSH B** ; The content of BC register pair is pushed to the stack, at the
; location pointed by SP.
IN 80H ; The data from the input device at port address 80 is placed in
; accumulator.

INSTRUCTION SET OF 8085

Common Notations:

1) Addr → 16 bit address.
2) Data → 8 or 16 bit data.
3) R → one of the registers.
4) Rp → register pair. BC pair is called B, DE → D and HL → L

Special Notes:
In the explanation of every instruction, I have mentioned its machine cycles and T-states. You will understand this part once you watch the two videos of timing diagrams

DATA TRANSFER GROUP

1) MOV $R_{Destination}$, R_{source}

The contents of register $\mathbf{R}_{source}$ is copied into register $\mathbf{R}_{Destination}$.
Eg: MOV A,B ; A ← B

.

Addr. Mode	Flags Affected	Cycles	T-States
Register	None	1	4

2) MOV R, M

The contents of the memory location pointed by HL (memory pointer) is copied into register R.
Eg: MOV B,M ; B ← [[HL]] i.e. B ← M

Addr. Mode	Flags Affected	Cycles	T-States
Indirect	None	2	7

3) MOV M, R

The contents of register R is copied into the memory location pointed by HL (memory pointer).
Eg: MOV M,B ; [[HL]] ← B i.e. M ← B

Addr. Mode	Flags Affected	Cycles	T-States
Register	None	2	7

4) MVI R, 8-bit data

The 8-bit data is immidiately moved into the register specified in the instruction.
Eg: MVI C, 23 ; C ← 23H

Addr. Mode	Flags Affected	Cycles	T-States
Immidiate	None	2	7

5) LXI Rp, 16-bit data

The 16-bit data is immidiately moved into the register pair specified in the instruction.
Eg: MVI B, 2300H ; BC ← 2300H i.e. B← 23, C← 00

Addr. Mode	Flags Affected	Cycles	T-States
Immidiate	None	3	10

6) MVI M, 8-bit data

The 8-bit data is immidiately moved into the memory location pointed by HL (memory pointer).
Eg: MVI M, 23 ; [[HL]] ← 23H

Addr. Mode	Flags Affected	Cycles	T-States
Immediate	None	3	10

7) LDA 16-bit address

The accumulator is loaded withb the contents of the memory location having the given address.
Eg: LDA 2000H ; A ← [2000]

Addr. Mode	Flags Affected	Cycles	T-States
Direct	None	4	13

8) STA 16-bit address

The accumulator is stored into the memory location having the given address.
Eg: STA 2000H ; [2000] ← A

Addr. Mode	Flags Affected	Cycles	T-States
Direct	None	4	13

9) LHLD 16-bit address

The HL pair is loaded with the contents of the loactions ppointed by the given address and address + 1. ☺ In case of doubts, contact Bharat Sir: - 98204 08217.
Eg: LHLD 2000 ; HL ← [2000] & [2001] i.e. L ← [2000], H ← [2001]

Addr. Mode	Flags Affected	Cycles	T-States
Direct	None	5	16

10) SHLD 16-bit address

The HL pair is stored into the loactions ppointed by the given address and address + 1.
Eg: SHLD 2000 ; [2000] & [2001] ← HL i.e. [2000] ← L, [2001] ← H

Addr. Mode	Flags Affected	Cycles	T-States
Direct	None	5	16

11) LDAX rp

The accumulator is loaded with the contents of memory location pointed by value of the given register.

Eg: LDAX B ; A ← [[BC]] i.e. if [BC] = 2000, A gets the value from location
; 2000 i.e. A ← [2000]

Addr. Mode	Flags Affected	Cycles	T-States
Indirect	None	2	7

12) STAX rp

The accumulator is stored into the location pointed by value of the given register.

Eg: STAX B ; [[BC]] ← A i.e. if [BC] = 2000, location 2000 will get the
; value of A i.e. [2000] ← A.

Addr. Mode	Flags Affected	Cycles	T-States
Indirect	None	2	7

13) PCHL

The Program Counter gets the contents of the HL register pair.
This statement causes a branch in the sequence of the program.

Eg: PCHL ; PC ← HL

Addr. Mode	Flags Affected	Cycles	T-States
Register	None	1	**6**

14) SPHL

The Stack Pointer gets the contents of the HL register pair.
This statement reloates the stack in the 64 KB memory.

Eg: SPHL ; SP ← HL

Addr. Mode	Flags Affected	Cycles	T-States
Register	None	1	**6**

15) XCHG

This instruction exchanges the contents of HL pair and DE pair.

Eg: XCHG ; HL ←→ DE
#Please refer Bharat Sir's Lecture Notes for this ...

Addr. Mode	Flags Affected	Cycles	T-States
Register	None	1	4

16) XTHL

This instruction exchanges the DE pair with the contents of location pointed by the SP and SP+1.

Eg: XTHL ; HL ←→ [[SP]] and [[SP]+1]
; i.e. if [SP]=2000 then L ←→ [2000] and H ←→ [2001]
#Please refer Bharat Sir's Video fo rmore on this ...

Addr. Mode	Flags Affected	Cycles	T-States
Register	None	5	16

Bharat Acharya Education
Learn...
8085 | 8086 | 80386 | Pentium |
8051 | ARM7 | COA

Fees: 1199/-
Duration: 6 months
Activation: Immediate
Certification: Yes
Free: PDFs of theory explanation
Free: VIVA questions and answers
Free: PDF of Multiple Choice Questions

Start Learning... NOW!
Bharat Acharya Education

Order our Books here...

8086 Microprocessor book
Link: https://amzn.to/3qHDpJH

8051 Microcontroller book
Link: https://amzn.to/3aFQkXc

Official WhatsApp number:
+91 9136428051

ARITHMETIC GROUP

Special Note:
In the explanation of every instruction, I have mentioned its machine cycles and T-states. You will understand this part once you watch the two videos of timing diagrams

1) ADD R

This instruction adds the contents of register R with the accumulator, stores result in the accumulator.
Eg: ADD B ; A ← A + B

Addr. Mode	Flags Affected	Cycles	T-States
Register	All	1	**4**

2) ADD M

This instruction adds the contents of the memory location pointed by HL, with the accumulator, and stores the result in the accumulator.
Eg: ADD M ; A ← A + [[HL]]

Addr. Mode	Flags Affected	Cycles	T-States
Indirect	All	2	**7**

3) ADI 8-bit data

This instruction adds the immidiate data with the accumulator, and stores the result in A.
Eg: ADI 25 ; A ← A + 25

Addr. Mode	Flags Affected	Cycles	T-States
Immediate	All	2	**7**

4) ADC R

This instruction adds the contents of the register R with the accumulator, and also adds the carry flag, and stores the result in the accumulator. It is used **while adding large numbers**.
Refer example from our video at **www.BharatAcharyaEducation.com**

Eg: ADC B ; A ← A + B + Cy

Addr. Mode	Flags Affected	Cycles	T-States
Register	All	1	**4**

5) ADC M

This instruction adds the contents of the memory location pointed by HL, with the accumulator, and also adds the carry flag, and stores the result in the accumulator.

Eg: ADC M ; A ← A + [[HL]] + Cy

Addr. Mode	Flags Affected	Cycles	T-States
Indirect	All	2	**7**

6) ACI 8-bit data

This instruction adds the immidiate data with the accumulator, and also adds the carry flag, and stores the result in the accumulator.

Eg: ACI 25 ; A ← A + 25 +Cy

Addr. Mode	Flags Affected	Cycles	T-States
Immediate	All	2	**7**

Simillarly subtraction is also done as above.

7) SUB R
8) SUB M
9) SUI 8-bit data
10) SBB R
11) SBB M
12) SBI 8-bit data

Special Note:
During subtraction if a borrow is taken, then carry flag CY becomes 1.

Special Note:
SBB is used to subtract two large numbers like 16-bit numbers.
First, we subtract the lower bytes using SUB
Then, we subtract the higher bytes using SBB to include the borrow taken by the lower byte.

INR R

This instruction increments the contents of the specified register.
The incremented value is stored back in the same register.
Eg: INR B ; B ← B + 1

Addr. Mode	Flags Affected	Cycles	T-States
Register	All except carry	1	4

14) INR M

This instruction increments the contents of memory location pointed by HL pair.
The incremented value is stored back at the same location.
Eg: INR M ; M ← M + 1 i.e. [[HL]] ← [[HL]] +1

Addr. Mode	Flags Affected	Cycles	T-States
Indirect	All except carry	3	10

15) INX Rp

This instruction increments the contents of the specified register **pair**.
The incremented value is stored back in the same register **pair**.
Eg: INX BC ; BC ← BC + 1 i.e. if [BC]=3000 then [BC] becomes 3001.

Addr. Mode	Flags Affected	Cycles	T-States
Register	**NONE**	1	**6**

16) DCR R

This instruction decrements the contents of the specified register.
The decremented value is stored back in the same register.
Eg: DCR B ; B ← B + 1

Addr. Mode	Flags Affected	Cycles	T-States
Register	All except carry	1	4

17) DCR M

This instruction decrements the contents of memory location pointed by HL pair.
The decremented value is stored back at the same location.
Eg: DCR M ; M ← M - 1 i.e. [[HL]] ← [[HL]] - 1

Addr. Mode	Flags Affected	Cycles	T-States
Indirect	All except carry	3	10

18) DCX Rp

This instruction decrements the contents of the specified register **pair**.
The decremented value is stored back in the same register **pair**.
Eg: DCX BC ; BC ← BC - 1 i.e. if [BC]=3001 then [BC] becomes 3000.

Addr. Mode	Flags Affected	Cycles	T-States
Register	**NONE**	1	**6**

19) DAD Rp

This instruction adds the contents of the given register pair with HL pair.
The result is stored in the HL pair.
Eg: DAD B ; HL ← HL + BC

Addr. Mode	Flags Affected	Cycles	T-States
Register	**Only Carry**	3	**10**

20) DAA

This instruction is used to get the answer in BCD form.
It adjusts the result of an addition operation to make the addition work like a decimal addition.
It is implied addressing and works strictly on A register.
It checks the nibbles of A as follows
If LN > 9 or AC = 1 then add 06H, If HN > 9 or CY = 1 then add 60H
#For examples, Please refer Bharat Sir's video fro more on this ...

Addr. Mode	Flags Affected	Cycles	T-States
Implied	**ALL**	1	**4**

Bharat Acharya Education
Learn...
8085 | 8086 | 80386 | Pentium |
8051 | ARM7 | COA

Fees: 1199/-
Duration: 6 months
Activation: Immediate
Certification: Yes
Free: PDFs of theory explanation
Free: VIVA questions and answers
Free: PDF of Multiple Choice Questions

Start Learning... NOW!
Bharat Acharya Education

Order our Books here...

8086 Microprocessor book
Link: https://amzn.to/3qHDpJH

8051 Microcontroller book
Link: https://amzn.to/3aFQkXc

Official WhatsApp number:
+91 9136428051

LOGIC GROUP

Special Note:
In the explanation of every instruction, I have mentioned its machine cycles and T-states. You will understand this part once you watch the two videos of timing diagrams

1) ANA R

Logically AND the contents of the specified register with accumulator, store result in accumulator.
Eg: ANA B ; A ← A AND B

Addr. Mode	Flags Affected	Cycles	T-States
Register	**ALL**	1	**4**

2) ANA M

Logically AND the contents of the memory location pointed by HL pair, with the accumulator.
Eg: ANA M ; A ← A AND M

Addr. Mode	Flags Affected	Cycles	T-States
Indirect	**ALL**	2	**7**

3) ANI 8-bit data

Logically AND the immidiate 8-bit data, with the accumulator.
Eg: ANA 25; A ← A AND 25

Addr. Mode	Flags Affected	Cycles	T-States
Immidiate	**ALL**	2	**7**

Special Note: Use of AND operation
To “Clear any bit”, we must “AND” that bit with “0” and the remaining bits with “1”.
Eg: ANI F0H will Clear the Lower Nibble of A while the Higher Nibble will remain the same.

Simillarly we have the other logic instrctions as follows:

4) ORA R
5) ORA M
6) ORI 8-bit data

Special Note: Use of OR operation
To "Set any bit", we must "OR" that bit with "1" and the remaining bits with "0".
Eg: ORI 0FH will Set the Lower Nibble of A while the Higher Nibble will remain the same.

7) XRA R
8) XRA M
9) XRI 8-bit data

Special Note: Use of XOR operation
To "Complement any bit", we must "XOR" that bit with "1" and the remaining bits with "0".
Eg: XRI 0FH will Complement the Lower Nibble while the Higher Nibble will remain the same.

10) CMP R

Compares the contents of register R and accumulator.
Comparision essntially is subtraction. Hence, this instruction performs A – R.
It is very important to **remember** that the **result** of this comparision is **NOT stored** in **accumulator**, only the Flags are afftected. ☺ In case of doubts, contact Bharat Sir: - 98204 08217.
Eg: CMP B ; Compares A and B i.e. A – B (and not B - A)
We decide which one of the two is greater by checking the flags affected as follows:

Conclusion	Zero Flag 'Z'	Carry Flag 'Cy'
A > B	0	0
A = B	1	0
A < B	0	1

Addr. Mode	Flags Affected	Cycles	T-States
Register	**ALL**	1	**4**

Simillarly we have the other comparision instrctions as follows:

11) CMP M
12) CPI 8-bit data

13) STC

Sets the carry flag.
Cy ← 1.

Addr. Mode	Flags Affected	Cycles	T-States
Implied	**Only Carry**	1	**4**

14) CMC

Complements the carry flag.
Cy ← Cy.

Addr. Mode	Flags Affected	Cycles	T-States
Implied	**Only Carry**	1	**4**

15) CMA

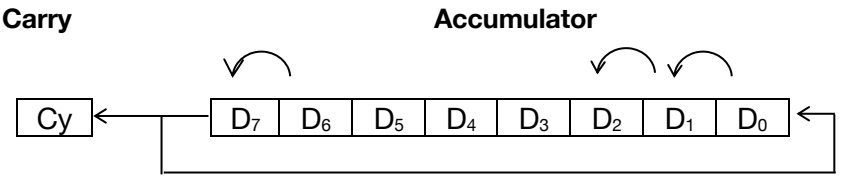
Complements the accumulator.
A ← 1's complement of A.

Addr. Mode	Flags Affected	Cycles	T-States
Implied	**None**	1	**4**

Special Note: Use of CMA operation
It acts as a NOT gate. AND followed by NOT gives a NAND.
Hence using CMA we can derive NAND, NOR and XNOR operations.

16) RLC

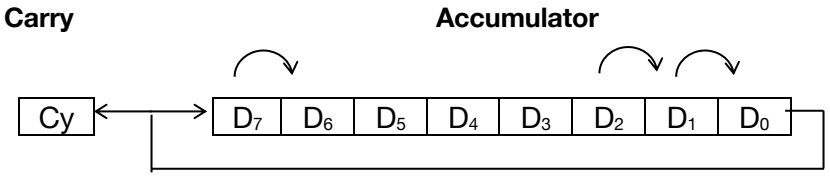
The Contents of accumulator are rotated left by 1.
The MSB goes to the Carry AND the LSB.



Addr. Mode	Flags Affected	Cycles	T-States
Implied	**Carry**	1	**4**

17) RRC

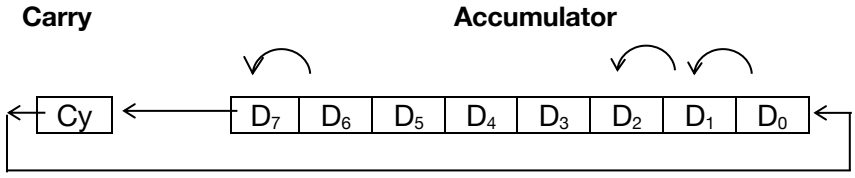
The Contents of accumulator are rotated right by 1.
The LSB goes to the Carry AND the MSB.



Addr. Mode	Flags Affected	Cycles	T-States
Implied	**Carry**	1	**4**

18) RAL

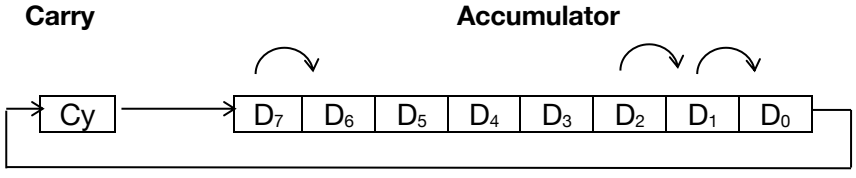
The Contents of accumulator are rotated left by 1.
The MSB goes to the Carry and THE CARRY goes to LSB.



Addr. Mode	Flags Affected	Cycles	T-States
Implied	**Carry**	1	**4**

19) RAR

The Contents of accumulator are rotated right by 1.
The LSB goes to the Carry and the CARRY goes to the MSB.



Addr. Mode	Flags Affected	Cycles	T-States
Implied	**Carry**	1	**4**

Bharat Acharya Education

Learn...
8085 | 8086 | 80386 | Pentium |
8051 | ARM7 | COA

Fees: 1199/-
Duration: 6 months
Activation: Immediate
Certification: Yes
Free: PDFs of theory explanation
Free: VIVA questions and answers
Free: PDF of Multiple Choice Questions

Start Learning... NOW!

Bharat Acharya Education

Order our Books here...

8086 Microprocessor book
Link: https://amzn.to/3qHDpJH

8051 Microcontroller book
Link: https://amzn.to/3aFQkXc

Official WhatsApp number:

+91 9136428051

BRANCH CONTROL GROUP

Special Note:
In the explanation of every instruction, I have mentioned its machine cycles and T-states. You will understand this part once you watch the two videos of timing diagrams

1) JMP 16-bit address (Unconditional Jump)

Loads PC with the 16-bit address specified in the instruction.
Eg: JMP 2500 ; PC ← 2500

Addr. Mode	Flags Affected	Cycles	T-States
Immidiate	**None**	3	**10**

Special Note:
Remember the point discussed in the video…
PC will contain address of the next instruction before it gets the branch address from WZ pair.

2) JCondition 16-bit address (Conditional Jump)

It is the same as UnConditional JUMP except that the action takes place ONLY if the condition is true.
Eg: JZ 2500 ; PC ← 2500 if Z =1

Addr. Mode	Flags Affected	Cycles	T-States
Immidiate	**None**	2/3	**7/10**

Conditions

Condition	Description	True if:
NZ	No 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**

3) CALL 16-bit address (Unconditional Call)

Loads PC with the 16-bit address specified in the instruction.
Before doing so, it also Pushes the Current PC into the Stack.

Eg: JMP 2500 ; SP ← SP – 1
[SP] ← PC_H
SP ← SP - 1
[SP] ← PC_L
PC ← 2500

#Please refer Bharat Sir's video for clear understanding of this instruction ...

Addr. Mode	Flags Affected	Cycles	T-States
Immidiate	**None**	5	**18**

4) CCondition 16-bit address (Conditional Call)

It is the same as UnConditional Call except that the action takes place ONLY if the condition is true.

Eg: CNZ 2500 ; IF Z = 0 then
SP ← SP – 1
[SP] ← PC_H
SP ← SP - 1
[SP] ← PC_L
PC ← 2500

Addr. Mode	Flags Affected	Cycles	T-States
Immidiate	**None**	2/5	**9/18**

5) RET (Unconditional Return)

This instruction is written at the end of the sub-routine and enables the control to go back to the main program.
We enter a subroutine using CALL instruction in which we push the return address into the stack.
In RET instruction we do the opposite i.e. we POP the return address from the stack into PC.

Eg: RET ; PC_L ← [SP]
SP ← SP + 1
PC_H ← [SP]
SP ← SP + 1

Addr. Mode	Flags Affected	Cycles	T-States
Indirect	**None**	3	**10**

6) RCondition (Conditional Return)

It is the same as UnConditional Return except that the action takes place ONLY if the condition is true.

Eg: RC ; IF C = 1 then
$PC_L \leftarrow [SP]$
$SP \leftarrow SP + 1$
$PC_H \leftarrow [SP]$
$SP \leftarrow SP + 1$

#Please refer Bharat Sir's Lecture Notes for this ...

Addr. Mode	Flags Affected	Cycles	T-States
Indirect	**None**	1/3	**6/12**

7) RSTn (Restart n)

This instruction is very similar to the CALL instruction.
Here the branch address, instead of being specified directly in the instruction, is calculated as (n × 8).
The current PC is Pushed into the stack.
The new value of PC is (n × 8).
The value of n = 0,1,2 … 7.
These instructions are called as Software Interrupts.
Operationally it is thus simillar to CALL except that it is a **1 byte instruction**.

Eg: RST1 ; $SP \leftarrow SP - 1$
$[SP] \leftarrow PC_H$
$SP \leftarrow SP - 1$
$[SP] \leftarrow PC_L$
$PC \leftarrow 0008$ ($\because 1 \times 8 = 0008$)

#Please refer Bharat Sir's Lecture Notes for this ...

Addr. Mode	Flags Affected	Cycles	T-States
Indirect	**None**	3	**12**

Special Note:
PCHL also causes a branch in the program flow (to the location pointed by the HL Pair), but is already included in the data transfer group. It is a very important instruction and during programming, you should remember that it can also cause a branch.

Bharat Acharya Education

Learn...
8085 | 8086 | 80386 | Pentium |
8051 | ARM7 | COA

Fees: 1199/-
Duration: 6 months
Activation: Immediate
Certification: Yes
Free: PDFs of theory explanation
Free: VIVA questions and answers
Free: PDF of Multiple Choice Questions

Start Learning... NOW!

Bharat Acharya Education

Order our Books here...

8086 Microprocessor book
Link: https://amzn.to/3qHDpJH

8051 Microcontroller book
Link: https://amzn.to/3aFQkXc

Official WhatsApp number:

+91 9136428051

STACK, I/O AND MACHINE CONTROL GROUP

Special Note:
In the explanation of every instruction, I have mentioned its machine cycles and T-states. You will understand this part once you watch the two videos of timing diagrams

STACK INSTRUCTIONS

1) PUSH Rp

It pushes the given register pair into the stack.

Eg: PUSH B ; SP ← SP – 1
[SP] ← B
SP ← SP - 1
[SP] ← C

Addr. Mode	Flags Affected	Cycles	T-States
Register	**None**	3	**12**

Special Notes:

All Stack operations (Push and Pop) are compulsorily **16-bit operations**.
We can push register pairs, and not individual 8-bit registers.

Push and Pop only support **Register Addressing Mode**. Hence if we want to push a number like 2000H, we must first move it into a register pair and then push it.

Eg: LXI B, 2000H
Push B

2) POP Rp

It pops the top 2 elements (2 × 8-bit ∴ 16-bit) from the stack into the given register pair.
The lower byte comes out first as the higher byte was pushed in first and stack operates in LIFO manner.

Eg: POP B ; C ← [SP]
SP← SP + 1
B ← [SP]
SP← SP + 1

Addr. Mode	Flags Affected	Cycles	T-States
Register	**None**	3	**10**

3) PUSH PSW

It pushes the PSW (Program Status Word) into the stack.
The PSW is the combination of the accumulator and the Flag register, accumulator being the higher byte.
∴ **PSW ➔ AF**
PSW can **ONLY** be used in PUSH and POP instructions.

Eg: PUSH PSW ; SP ← SP – 1
[SP] ← A
SP ← SP - 1
[SP] ← F

Addr. Mode	Flags Affected	Cycles	T-States
Register	**None**	3	**12**

4) POP PSW

It pops the top 2 elements (2 × 8-bit ∴ 16-bit) from the stack into the PSW.

Eg: POP PSW ; F ← [SP]
SP← SP + 1
A ← [SP]
SP← SP + 1

Addr. Mode	Flags Affected	Cycles	T-States
Register	**None**	3	**10**

Please Note: There are other instructions like XTHL, SPHL, LXI SP, CALL RET etc which directly or indirectly affect the stack and are already included in various groups above.

Special Note:
PSW is only created to allow push and pop of Accumulator and Flags.
Hence PSW can only be used in Push and Pop instructions.

I/O INSTRUCTIONS

5) IN 8-bit I/O Port address

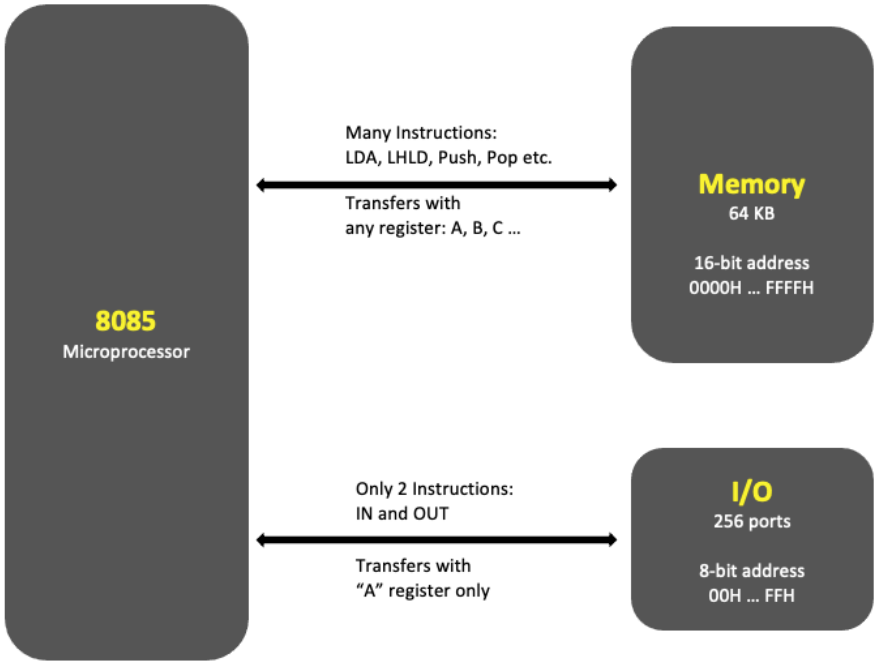
8085 has 256 I/O Ports having 8-bit addresses 00H … FFH.
This instruction is used to read data from an I/O Port, whose address is given in the instruction.
This data can be read into the Accuulator ONLY.
Eg: IN 80 ; A ← $[80]_{I/O}$

Addr. Mode	Flags Affected	Cycles	T-States
Direct	**None**	3	**10**

6) OUT 8-bit I/O Port address

This instruction is used to send data from the accumulator to an I/O Port, whose address is in the instruction.
This data can be sent from the Accuulator ONLY.
Eg: OUT 80; $[80]_{I/O}$ ← A

Addr. Mode	Flags Affected	Cycles	T-States
Direct	**None**	3	**10**



MACHINE CONTROL INSTRUCTIONS

7) SIM (Set Interrupt Mask)

This instruction is used to set the interrupt masking pattern for the µP.
The appropriate bit pattern is **loaded into** the accumulator an then this instruction is executed.
It is basically used to **mask/unmask** the **interrupts** except TRAP and INTR.
It can also be used to send a '**bit**' out through the serial out pin **SID**.
Eg: SIM ; µP accepts the masking pattern through the accumulator.

Addr. Mode	Flags Affected	Cycles	T-States
Implied	**None**	1	**4**

8) RIM (Read Interrupt Mask)

This instruction is used to read the interrupt masking pattern for the µP.
After executing this instruction, the µP loads the bit pattern **into** the accumulator.
It can also be used to receive a '**bit**' out through the serial in pin **SID**.
Eg: RIM ; µP loads the masking pattern into the accumulator.

Addr. Mode	Flags Affected	Cycles	T-States
Implied	**None**	1	**4**

Please Note: For the bit patterns of SIM and RIM instruction, please refer the chapter on Interrupts.

9) EI (Enable Interrupts)

This instruction is used to enable the interrupts in the µP.
This instrusetion sets the INTE flip-flop (Interrupt Enable Flip-Flop).
This instruction effects all the interrupts except TRAP.
Eg: EI ; $INTE_{F/F} \leftarrow 1$

Addr. Mode	Flags Affected	Cycles	T-States
Implied	**None**	1	**4**

10) DI (Disable Interrupts)

This instruction is used to disable the interrupts in the µP.
This instrusetion resets the INTE flip-flop.
This instruction effects all the interrupts except TRAP.
Eg: DI ; $INTE_{F/F} \leftarrow 0$

Addr. Mode	Flags Affected	Cycles	T-States
Implied	**None**	1	**4**

11) NOP (No Operation)

This instruction performs no operation, but consumes time of the μP.
It is the simplest method of causing a software delay.
Eg: NOP ; ----------

Addr. Mode	Flags Affected	Cycles	T-States
Implied	**None**	1	**4**

12) HLT (Halt)

This instruction signifies the end of the program.
It causes the μP to stop fetching any further instruction, hence program execution is stopped.
It makes the Halt Flip Flop inside the μP = 1.
μP checks the Halt Flip Flop in the beginning (1st T-State) of the next Machine Cycle and Stops all operations.

Eg: HLT ; Halt Flip-Flop ← 1

Addr. Mode	Flags Affected	Cycles	T-States
Implied	**None**	1 + 1T	**5**

Bharat Acharya Education

Learn...
8085 | 8086 | 80386 | Pentium |
8051 | ARM7 | COA

Fees: 1199/-
Duration: 6 months
Activation: Immediate
Certification: Yes
Free: PDFs of theory explanation
Free: VIVA questions and answers
Free: PDF of Multiple-Choice Questions

Start Learning... NOW!

Bharat Acharya Education

Order our Books here...

8086 Microprocessor book
Link: https://amzn.to/3qHDpJH

8051 Microcontroller book
Link: https://amzn.to/3aFQkXc

Official WhatsApp number:

+91 9136428051

8085 TIMING DIAGRAMS

Instruction Cycle:
This is the time required by the µP to fetch and execute one complete instruction.
The instruction cycle is in two parts:
1. Fetch Cycle
2. Execute Cycle

Fetch Cycle:
This is the time required by the µP to fetch all bytes of an instruction
The length of the fetch cycle is thus determined by the no of bytes in an instruction.

Execution Cycle:
This is the time required by the µP to execute a fetched instruction.

An Instruction cycle consists of various machine cycles.
The most important and also compulsory machine cycle of every instruction is OPCODE FETCH.
There are other machine cycles such as memory read, memory write, I/O read etc.
Machine cycles contain T-states

T-State:
A T-State is one clock cycle of the µP.
∴T = Clock Period = 1/Clock Frequency
Since 8085 (standard version) works at 3 MHz, one T-state = 1÷3 microseconds = 0.333 microseconds.

MACHINE CYCLES

It is the time required by the μP doing one operation and accessing one byte from the external module (Memory or I/O). As the data bus of 8085 is 8-bit, one machine cycle will transfer one byte (8-bits). Instructions that require to read or write 16-bit data from memory, need multiple machine cycles.

The Machine cycles of 8085 are given below:

Name	IO/M	RD	WR	S1	S0	INTA	T-States
Opcode Fetch	0	0	1	1	1	1	**4/6**
Mem Read	0	0	1	1	0	1	3
Mem Write	0	1	0	0	1	1	3
IO Read	1	0	1	1	0	1	3
IO Write	1	1	0	0	1	1	3
Int. Acknowledge	1	1	1	1	1	0	**3 or 6**
Bus Idle	0	1	1	0	0	1	3

Opcode fetch is compulsory in every instruction and is the first machine cycle of every instruction.

Most instructions are memory based so they may need Memory Reads or Memory Writes after Opcode Fetch.

Only IN and OUT instructions involve IO Read and IO Write respectively.

Only DAD instruction needs Bus Idle

Interrupt Acknowledgment cycle is performed in response to INTR signal. This will be explained much later in the topic of interrupts

Special Note:

Be smart and focus mainly on Opcode Fetch, Memory Read and Memory Write in the beginning as the other machine cycles are used by only a handful of instructions.

OPCODE FETCH

- This cycle is used to **fetch** the **Opcode from** the **memory**.
- This is the **First** Machine Cycle of every instruction.
- It is a **compulsory** Machine Cycle
- It is **generally** of **4 T**-States but **some** instructions require a **6 T**-State Opcode Fetch.

During T1

- **A15-A8** contains the higher byte of the address (**PCH**)
- As **ALE** is **high AD7-AD0** contains the lower byte of the address (**PCL**).
- Since it is an Opcode fetch cycle, **S1** and **S0** go **high**.
- Since it is a memory operation, **IO/M** goes **low**.

During T2

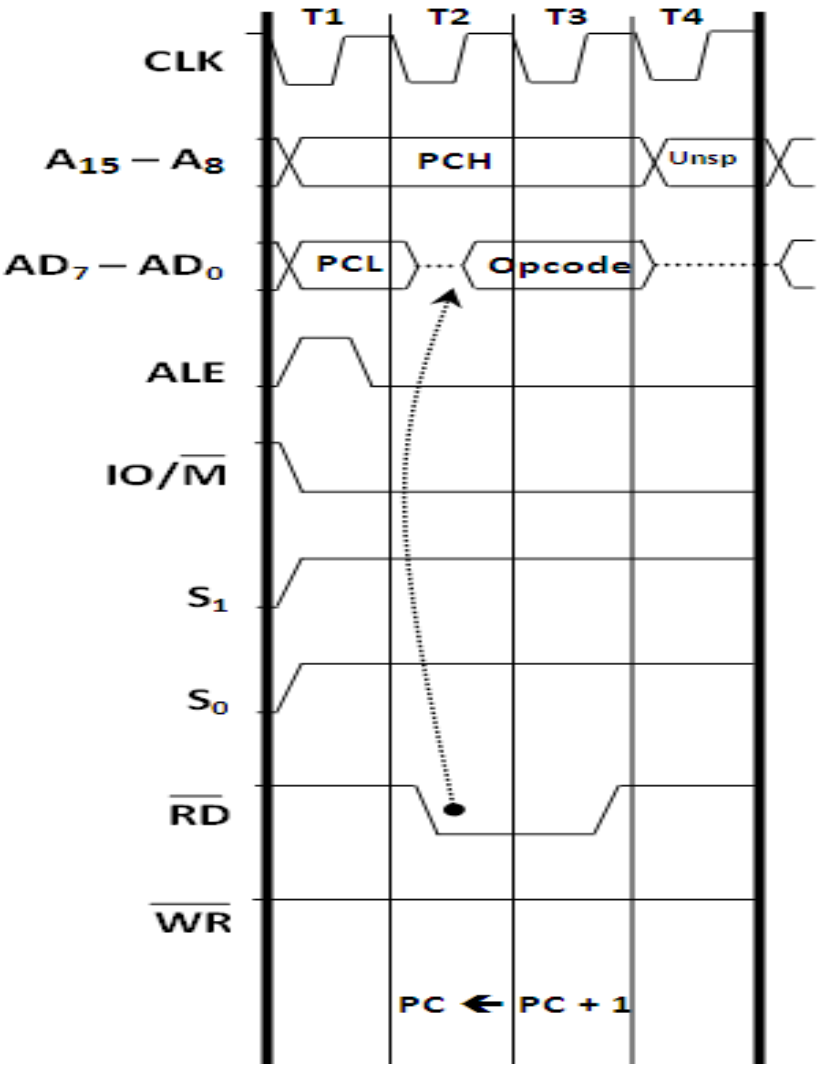
- As **ALE** goes **low** address is removed from AD7-AD0.
- As **RD** goes **low**, **data** appears **on AD7-AD0**. ☺ In case of doubts, contact Bharat Sir: - 98204 08217.
- As µP is reading the data (Opcode) from memory, there is a propagation delay between sending the address and arrival of data. This is the time required for data (Opcode) to travel from memory to µP. {It is shown by dots between address and data}
- The µP examines the state of the READY pin.
  If READY pin is "high", µP will continue, but if it is low, then it means the device is not ready, as it is slow. Hence the µP enters wait-state by executing wait cycles and remains in the wait-state until READY goes high.

During T3

- Data remains on AD7-AD0 till RD is low. This is the time given to µP to capture the data (Opcode) from the data bus.

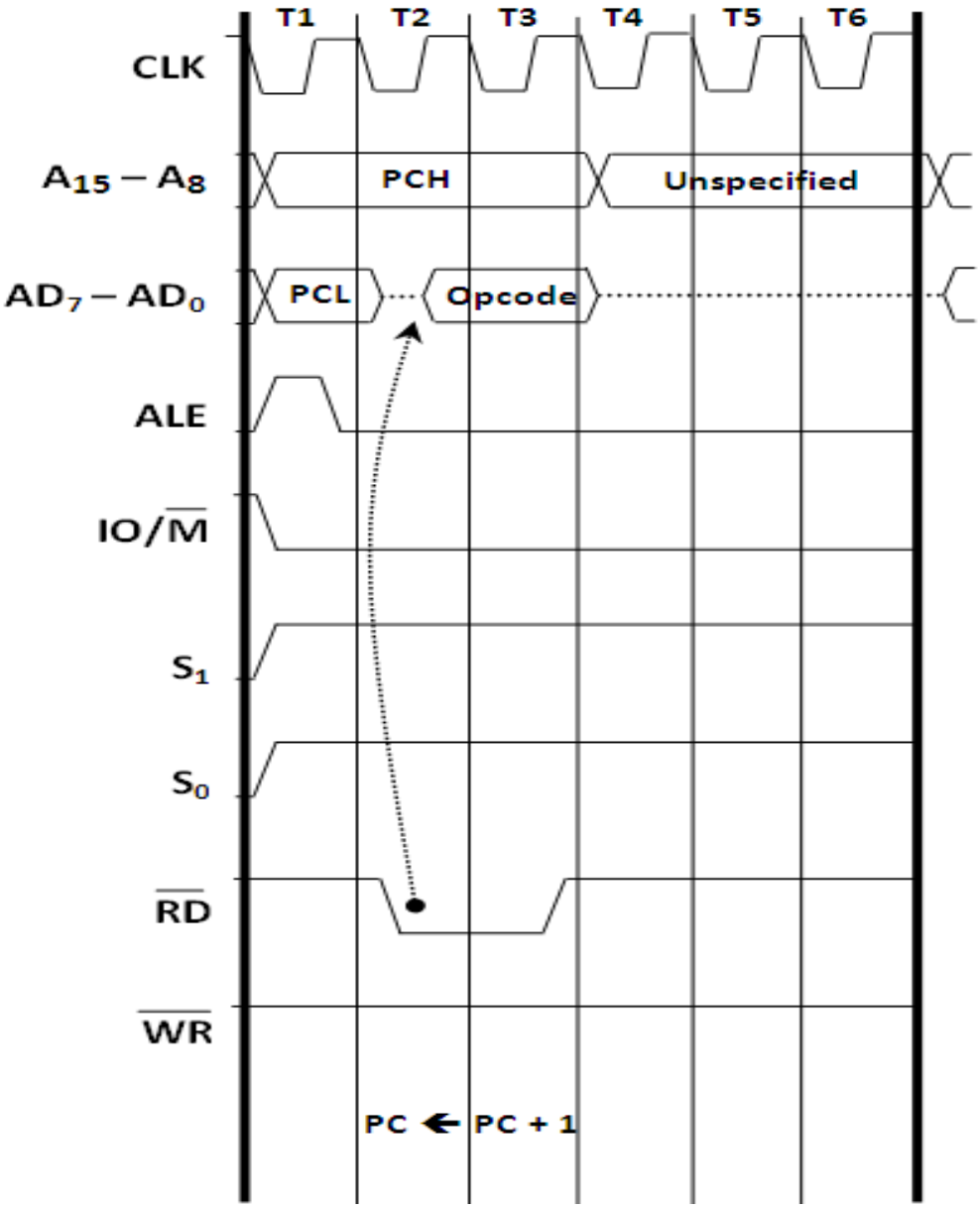
During T4

- T4 state is used by the µP to decode the Opcode.
- This is how an Opcode Fetch cycle is different from a Memory Read cycle.



Special Note:
Timing diagrams show what is happening in the buses. Decoding in an internal operation and hence is not shown in timing diagrams.

OPCODE FETCH OF 6 T-STATES



Instructions that use a 6T Opcode Fetch

PCHL
SPHL
INX
DCX
PUSH
CALL (all types)
RC (Conditional RET)
RSTn

You will of course, understand this later as you learn timing diagrams of instructions.

Special Note:
T5 and T6 are used for internal operation. Nothing happens on the buses during that time hence nothing is shown in the timing diagram.

MEMORY READ

- This cycle is used to **fetch one byte from** the **memory**.
- This cycle can be used to fetch the operand bytes of an instruction or any data from the memory.
- It is a not compulsory Machine Cycle
- It requires **3 T-States**.

During T1

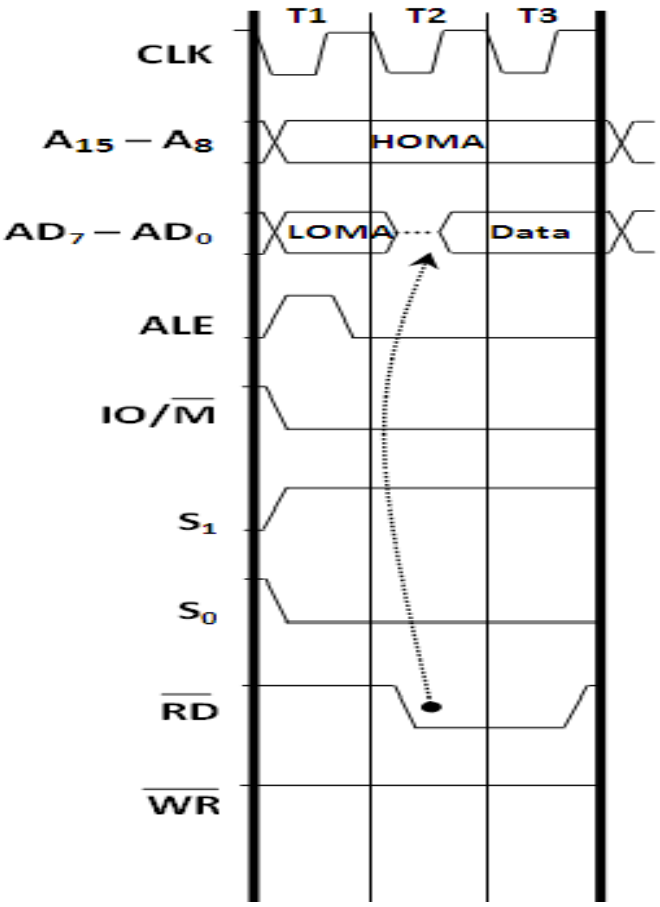
- **A15-A8** contains the higher byte of the address (**PCH**)
- As **ALE** is **high**, **AD7-AD0** contains the lower byte of the address (**PCL**).
- Since it is a Memory Read cycle, **S1** goes **high**.
- Since it is a memory operation, **IO/M** goes **low**.

During T2

- **ALE** goes **low**.
- Address is removed from AD7-AD0.
- As **RD** goes **low, data** appears **on AD7-AD0**.

During T3

- Data remains on AD7-AD0 till RD is low.



Special Note:

Notice the propagation delay, as this is a read operation.

MEMORY WRITE

- This cycle is used to **send** (write) **one byte into** the **memory**.
- It is a not compulsory Machine Cycle
- It requires **3 T-States**.

During T1

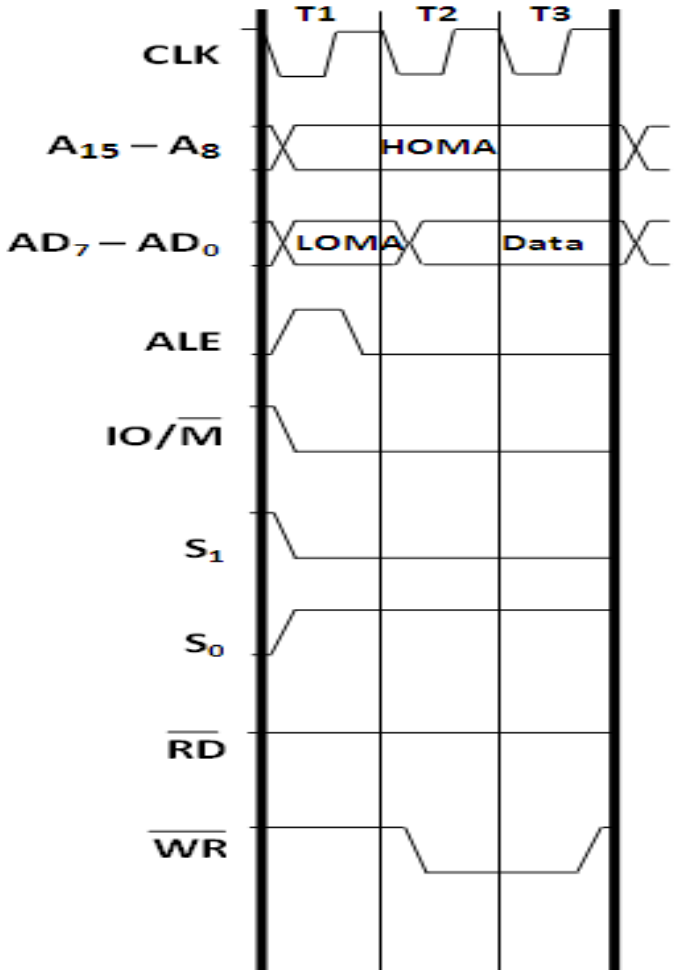
- **A15-A8** contains the higher byte of the address (**PCH**)
- As **ALE** is **high**, **AD7-AD0** contains the lower byte of the address (**PCL**).
- Since it is a Memory Write cycle, **S0** goes **high**.
- Since it is a memory operation, **IO/M** goes **low**.

During T2

- **ALE** goes **low**.
- Address is removed from AD7-AD0.
- **Data** appears **on AD7-AD0** and **WR** goes **low**.

During T3

- Data remains on AD7-AD0 till WR is low.



Special Note:

Notice, NO propagation delay, as this is a write operation.

IO READ

- This cycle is used to **fetch one byte from** an **IO Port**.
- It is a not compulsory Machine Cycle
- It requires **3 T-States**.

During T1

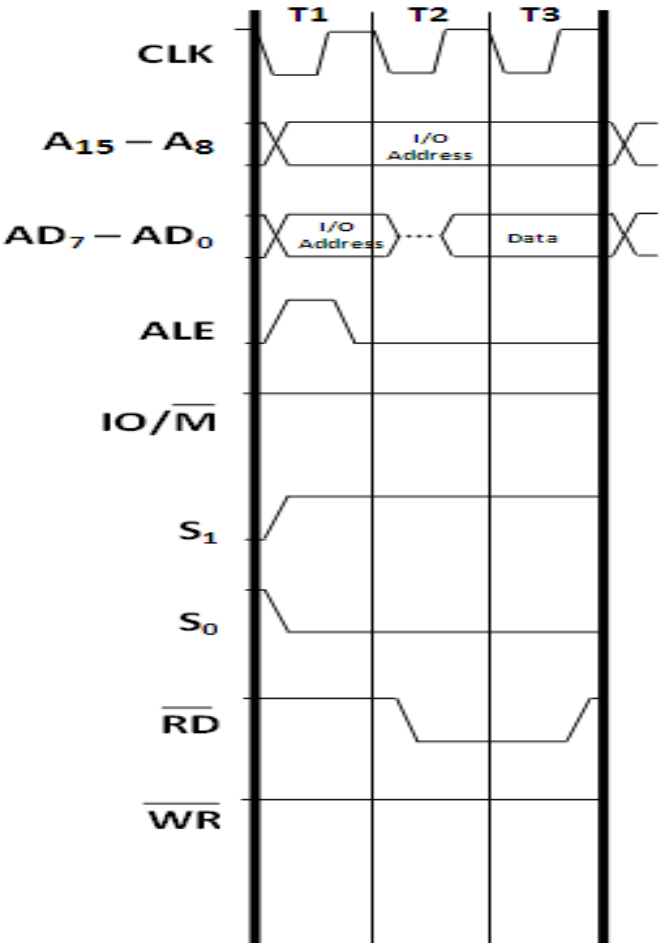
- The **lower 8 bits of the IO Port Address** are **duplicated into** the higher order address bus **A15-A8**.
- As **ALE** is **high AD7-AD0 contains** the **lower byte of** the **address**
- Since it is an IO **Read** cycle **S1** goes **high**.
- Since it is an **IO** operation **IO/M** goes **high**.

During T2

- **ALE** goes **low.**
- Address is removed from AD7-AD0.
- As **RD** goes **low**, **data** appears on **AD7-AD0**.

During T3

- Data remains on AD7-AD0 till RD is low.



Special Note:

Notice the propagation delay, as this is a read operation.

IO WRITE

- This cycle is used to **send** (write) **one byte into** an **IO Port**.
- It is a not compulsory Machine Cycle
- It requires **3 T-States.**

During T1

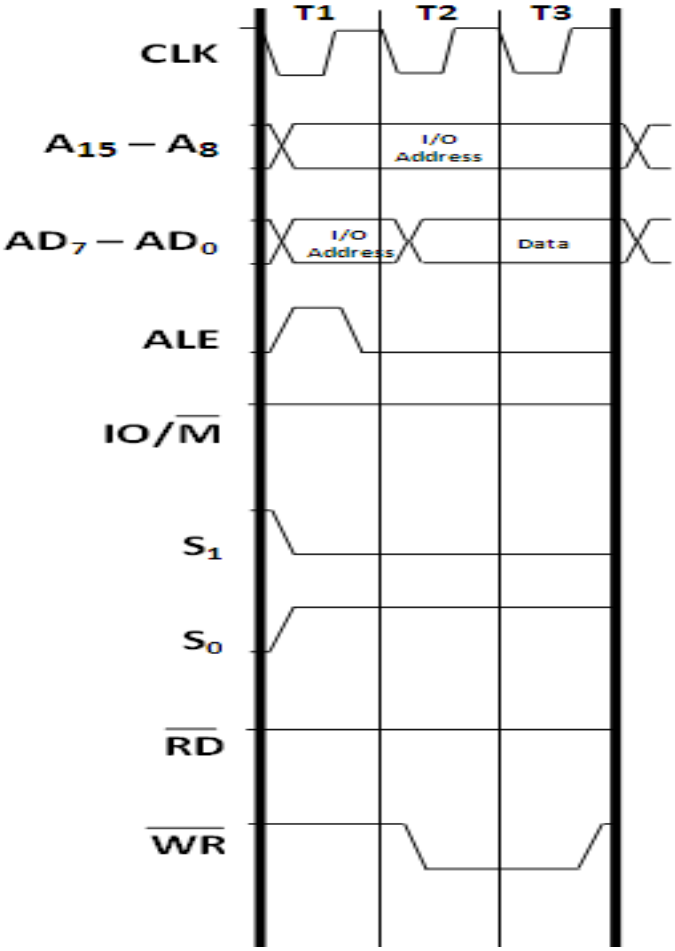
- The **lower 8 bits of the IO Port Address** are **duplicated into** the higher order address bus **A15-A8**.
- As **ALE** is **high AD7-AD0 contains** the **lower byte of** the **address**
- Since it is an IO **Write** cycle, **S0** goes **high**.
- Since it is an **IO** operation, **IO/M** goes **high**.

During T2

- **ALE** goes **low.**
- Address is removed from AD7-AD0.
- **Data** appears **on AD7-AD0** and **WR** goes **low**.

During T3

- Data remains on AD7-AD0 till RD is low.



Special Note:

Notice, NO propagation delay, as this is a write operation.

Bharat Acharya Education

Learn...
8085 | 8086 | 80386 | Pentium | 8051 | ARM7 | COA

Fees: 1199/- | Duration: 6 months | Activation: Immediate | Certification: Yes
Free: PDFs of Theory explanation, VIVA questions and answers, Multiple-Choice Questions

Start Learning... NOW!

Bharat Acharya Education

Order our Books here...

8086 Microprocessor

Link: https://amzn.to/3qHDpJH

8051 Microcontroller

Link: https://amzn.to/3aFQkXc

Official WhatsApp number:

+91 9136428051

8085 TIMING DIAGRAMS OF INSTRUCTIONS

Special Note:

Any operation performed INSIDE the microprocessor does not require a machine cycle and hence will not be shown in timing diagrams.

1) MVI B, 25H

B ← 25 H

Machine Cycle	Address Bus	Data Bus	T-States
Opcode Fetch	PC	Opcode (Eg: 3E)	**4**
Memory Read	PC + 1	25	**3**
		Total	**7**

2) LXI B, 2000H

BC ← 2000 H

Machine Cycle	Address Bus	Data Bus	T-States
Opcode Fetch	PC	Opcode (Eg: 3E)	**4**
Memory Read	PC + 1	00	**3**
Memory Read	PC + 2	20	**3**
		Total	**10**

3) LDA 2000H

A ← [2000H]

Machine Cycle	Address Bus	Data Bus	T-States
Opcode Fetch	PC	Opcode (Eg: 3E)	**4**
Memory Read	PC + 1	00 (Z)	**3**
Memory Read	PC + 2	20 (W)	**3**
Memory Read	2000	[2000]	**3**
		Total	**13**

4) STA 3000H

A → [3000H]

Machine Cycle	Address Bus	Data Bus	T-States
Opcode Fetch	PC	Opcode (Eg: 3E)	**4**
Memory Read	PC + 1	00 (Z)	**3**
Memory Read	PC + 2	30 (W)	**3**
Memory Write	3000	A	**3**
		Total	**13**

5) LDAX B

A ← [BC]

Machine Cycle	Address Bus	Data Bus	T-States
Opcode Fetch	PC	Opcode (Eg: 3E)	**4**
Memory Read	BC	[BC]	**3**
		Total	**7**

6) STAX D

A → [DE]

Machine Cycle	Address Bus	Data Bus	T-States
Opcode Fetch	PC	Opcode (Eg: 3E)	**4**
Memory Write	DE	A	**3**
		Total	**7**

7) LHLD 2000H

L ← [2000H], H ← [2001H]

Machine Cycle	Address Bus	Data Bus	T-States
Opcode Fetch	PC	Opcode (Eg: 3E)	**4**
Memory Read	PC + 1	00 (Z)	**3**
Memory Read	PC + 2	20 (W)	**3**
Memory Read	2000	[2000]	**3**
Memory Read	2001	[2001]	**3**
		Total	**16**

8) SHLD 5140H

L → [5140H], H → [5141H]

Machine Cycle	Address Bus	Data Bus	T-States
Opcode Fetch	PC	Opcode (Eg: 3E)	**4**
Memory Read	PC + 1	40 (Z)	**3**
Memory Read	PC + 2	51 (W)	**3**
Memory Write	5140	L	**3**
Memory Write	5141	H	**3**
		Total	**16**

9) MOV B,C

B ← C

Machine Cycle	Address Bus	Data Bus	T-States
Opcode Fetch	PC	Opcode (Eg: 3E)	**4**
		Total	**4**

10) PCHL

PC ← HL

6T
Opcode Fetch

Machine Cycle	Address Bus	Data Bus	T-States
Opcode Fetch	PC	Opcode (Eg: 3E)	**6**
		Total	**6**

11) SPHL

SP ← HL

6T
Opcode Fetch

Machine Cycle	Address Bus	Data Bus	T-States
Opcode Fetch	PC	Opcode (Eg: 3E)	**6**
		Total	**6**

12) ADD B {all 8 bit arithmetic operations using register addressing mode}

A ← A + B

Machine Cycle	Address Bus	Data Bus	T-States
Opcode Fetch	PC	Opcode (Eg: 3E)	**4**
		Total	**4**

13) INR B

B ← B + 1

Machine Cycle	Address Bus	Data Bus	T-States
Opcode Fetch	PC	Opcode (Eg: 3E)	**4**
		Total	**4**

14) INX B

6T
Opcode Fetch

BC ← BC + 1

Machine Cycle	Address Bus	Data Bus	T-States
Opcode Fetch	PC	Opcode (Eg: 3E)	**6**
		Total	**6**

Special Note:

- If you are learning this by piracy, then you are not my student.
- You are simply a thief!
  #PoorUpbringing

INSTRUCTIONS INVOLVING M – MEMORY POINTER

Special Note:

- Instructions involving M must be examined more carefully.
- Remember M is not a register. M is a MEMORY LOCATION pointed by HL pair.
- Taking data from M will need a Memory Read cycle.
- Putting data in M will need a Memory Write cycle.

15) MVI B, 25H

B ← B + 1

Machine Cycle	Address Bus	Data Bus	T-States
Opcode Fetch	PC	Opcode (Eg: 3E)	**4**
Memory Read	PC + 1	25	**3**
Memory Write	HL	25	**3**
		Total	**10**

16) MOV B, M

B ← M

Machine Cycle	Address Bus	Data Bus	T-States
Opcode Fetch	PC	Opcode (Eg: 3E)	**4**
Memory Read	HL	M	**3**
		Total	**7**

17) MOV M, B

M ← B

Machine Cycle	Address Bus	Data Bus	T-States
Opcode Fetch	PC	Opcode (Eg: 3E)	**4**
Memory Write	HL	B	**3**
		Total	**7**

18) INR M {Very Important}

M ← M + 1

Machine Cycle	Address Bus	Data Bus	T-States
Opcode Fetch	PC	Opcode (Eg: 3E)	**4**
Memory Read	HL	M	**3**
Memory Write	HL	M + 1	**3**
		Total	**10**

19) ADD M

A ← A + M

Machine Cycle	Address Bus	Data Bus	T-States
Opcode Fetch	PC	Opcode (Eg: 3E)	**4**
Memory Read	HL	M	**3**
		Total	**7**

STACK OPERATIONS

20) PUSH B

SP ← SP – 1 … internal operation
[SP] ← B … memory write
SP ← SP – 1 … internal operation
[SP] ← C … memory write

6T
Opcode Fetch

Machine Cycle	Address Bus	Data Bus	T-States
Opcode Fetch	PC	Opcode (Eg: 3E)	**6**
Memory Write	SP – 1	B	**3**
Memory Write	SP – 2	C	**3**
		Total	**12**

21) POP B

C ← [SP] … memory read
SP ← SP + 1 … internal operation
B ← [SP] … memory read
SP ← SP + 1 … internal operation

Machine Cycle	Address Bus	Data Bus	T-States
Opcode Fetch	PC	Opcode (Eg: 3E)	**4**
Memory Read	SP	[SP]	**3**
Memory Read	SP +1	[SP + 1]	**3**
		Total	**10**

BRANCH INSTRUCTIONS

22) JMP 2000H

PC ← 2000 H

Machine Cycle	Address Bus	Data Bus	T-States
Opcode Fetch	PC	Opcode (Eg: 3E)	**4**
Memory Read	PC + 1	00 (Z)	**3**
Memory Read	PC + 2	20 (W)	**3**
		Total	**10**

23) JC 2000H

If CF = 1 then condition is true hence,

Machine Cycle	Address Bus	Data Bus	T-States
Opcode Fetch	PC	Opcode (Eg: 3E)	**4**
Memory Read	PC + 1	00 (Z)	**3**
Memory Read	PC + 2	20 (W)	**3**
		Total	**10**

If CF = 0 then condition is false hence,

Machine Cycle	Address Bus	Data Bus	T-States
Opcode Fetch	PC	Opcode (Eg: 3E)	**4**
Memory Read (Idle)	---	---	**3**
		Total	**7**

24) Call 2000H

SP ← SP – 1 … internal operation
[SP] ← PCH … Memory Write
SP ← SP – 1 … internal operation
[SP] ← PCL … Memory Write
PC ← 2000 H … internal operation

6T
Opcode Fetch

Machine Cycle	Address Bus	Data Bus	T-States
Opcode Fetch	PC	Opcode (Eg: 3E)	**6**
Memory Read	PC + 1	00 (Z)	**3**
Memory Read	PC + 2	20 (W)	**3**
Memory Write	SP – 1	PCH	**3**
Memory Write	SP – 2	PCL	**3**
		Total	**18**

25) CC 2000H

If CF=1 then condition is true hence,

SP ← SP – 1 … internal operation
[SP] ← PCH … Memory Write
SP ← SP – 1 … internal operation
[SP] ← PCL … Memory Write
PC ← 2000 H … internal operation

6T
Opcode Fetch

Machine Cycle	Address Bus	Data Bus	T-States
Opcode Fetch	PC	Opcode (Eg: 3E)	**6**
Memory Read	PC + 1	00 (Z)	**3**
Memory Read	PC + 2	20 (W)	**3**
Memory Write	SP – 1	PCH	**3**
Memory Write	SP – 2	PCL	**3**
		Total	**18**

If CF = 0 then condition is false hence,

Machine Cycle	Address Bus	Data Bus	T-States
Opcode Fetch	PC	Opcode (Eg: 3E)	**6**
Memory Read (Idle)	---	---	**3**
		Total	**9**

26) RET

PCL ← [SP] … Memory Read
SP ← SP + 1 … internal operation
PCH ← [SP] … Memory Read
SP ← SP + 1 … internal operation

Machine Cycle	Address Bus	Data Bus	T-States
Opcode Fetch	PC	Opcode (Eg: 3E)	**4**
Memory Read	SP	[SP]	**3**
Memory Read	SP + 1	[SP + 1]	**3**
		Total	**12**

27) RC

If CF = 1 then condition is true hence,

PCL ← [SP] … Memory Read
SP ← SP + 1 … internal operation
PCH ← [SP] … Memory Read
SP ← SP + 1 … internal operation

6T
Opcode Fetch

Machine Cycle	Address Bus	Data Bus	T-States
Opcode Fetch	PC	Opcode (Eg: 3E)	**6**
Memory Read	SP	[SP]	**3**
Memory Read	SP + 1	[SP + 1]	**3**
		Total	**12**

If CF = 0 then condition is false hence,

Machine Cycle	Address Bus	Data Bus	T-States
Opcode Fetch	PC	Opcode (Eg: 3E)	**6**
		Total	**9**

28) RSTn

SP ← SP – 1 … internal operation
[SP] ← PCH … Memory Write
SP ← SP – 1 … internal operation
[SP] ← PCL … Memory Write
PC ← (n x 8) … internal operation

6T
Opcode Fetch

Machine Cycle	Address Bus	Data Bus	T-States
Opcode Fetch	PC	Opcode (Eg: 3E)	**6**
Memory Write	SP – 1	PCH	**3**
Memory Write	SP – 2	PCL	**3**
		Total	**12**

I/O OPERATIONS

29) IN 80H

A ← $[80]_{I/O}$

Machine Cycle	Address Bus	Data Bus	T-States
Opcode Fetch	PC	Opcode (Eg: 3E)	**4**
Memory Read	PC + 1	80	**3**
I/O Read	80	[80]	**3**
		Total	**10**

30) OUT 80H

A → $[80]_{I/O}$

Machine Cycle	Address Bus	Data Bus	T-States
Opcode Fetch	PC	Opcode (Eg: 3E)	**4**
Memory Read	PC + 1	80	**3**
I/O Write	80	A	**3**
		Total	**10**

ADDITIONAL INSTRUCTIONS

31) DAD D

HL ← HL + DE

Machine Cycle	Address Bus	Data Bus	T-States
Opcode Fetch	PC	Opcode (Eg: 3E)	**4**
Bus Idle	---	---	**3**
Bus Idle	---	---	**3**
		Total	**10**

32) HLT

Halt F/F ← 1

Machine Cycle	Address Bus	Data Bus	T-States
Opcode Fetch	PC	Opcode (Eg: 3E)	**4T +1T**
		Total	**5**

33) XTHL

Z ← [SP] … Memory Read
W ← [SP + 1] … Memory Read
[SP + 1] ← H … Memory Write
[SP] ← L … Memory Write
HL ← WZ … internal operation

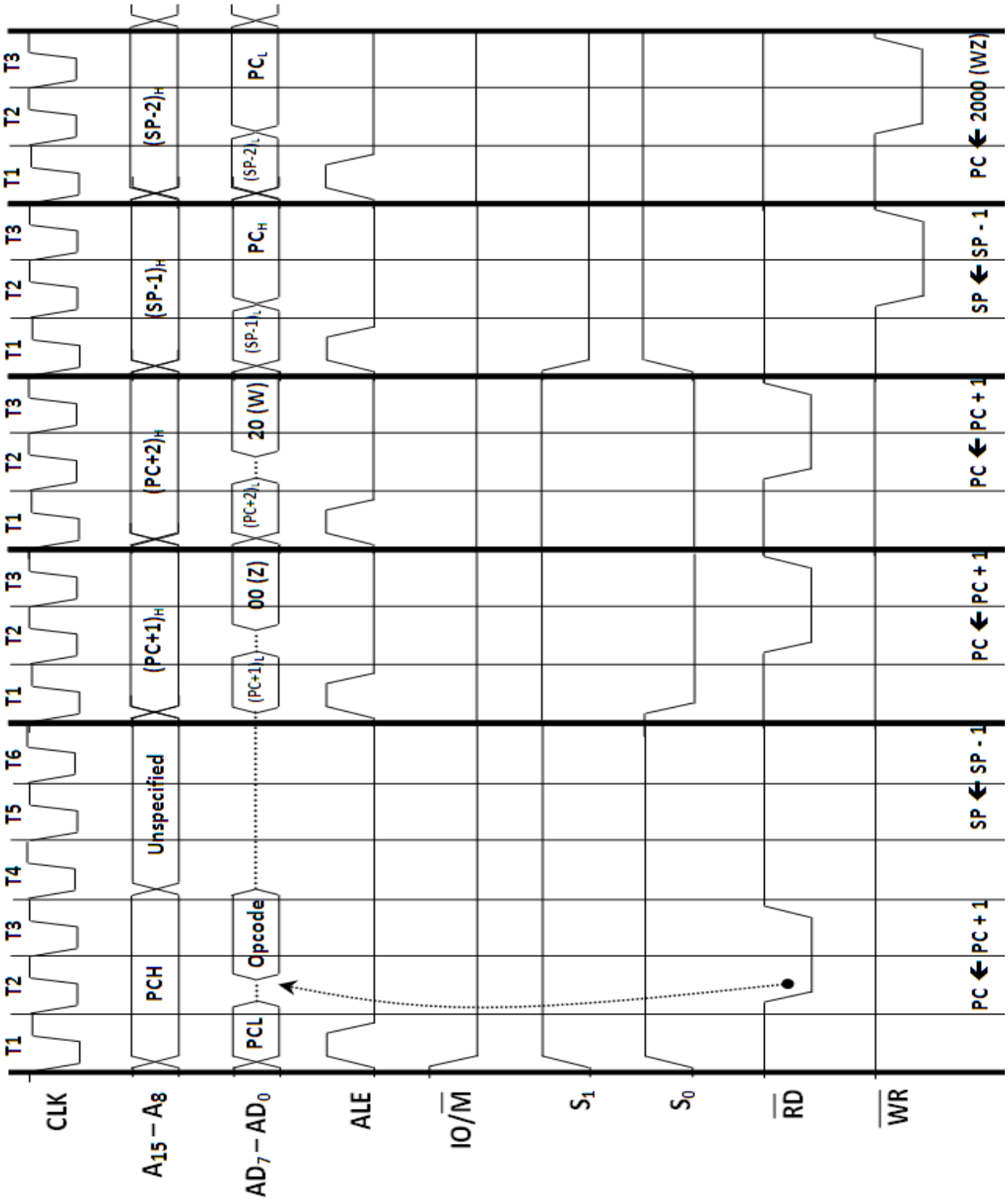
Machine Cycle	Address Bus	Data Bus	T-States
Opcode Fetch	PC	Opcode (Eg: 3E)	**4**
Memory Read	SP	[SP]	**3**
Memory Read	SP + 1	[SP + 1]	**3**
Memory Write	SP + 1	H	**3**
Memory Write	SP	L	**3**
		Total	**16**

34) XCHG

DE ←→ HL … internal operation

Machine Cycle	Address Bus	Data Bus	T-States
Opcode Fetch	PC	Opcode (Eg: 3E)	**4**
		Total	**4**

Timing Diagram for Call 2000 H



Bharat Acharya Education
Learn...
8085 | 8086 | 80386 | Pentium | 8051 | ARM7 | COA

Fees: 1199/- | Duration: 6 months | Activation: Immediate | Certification: Yes
Free: PDFs of Theory explanation, VIVA questions and answers, Multiple-Choice Questions

Start Learning... NOW!
www.BharatAcharyaEducation.com

Order our Books here...

8086 Microprocessor
Link: https://amzn.to/3qHDpJH

8051 Microcontroller
Link: https://amzn.to/3aFQkXc

Official WhatsApp number:
+91 9136428051

BHARAT ACHARYA EDUCATION
Cell: 98204 08217
BharatSir@hotmail.com

8085 PROGRAMMING

Q 1) **WAP to add the contents of locations 4000H and 4001H. Store the sum at 4002H and the carry at 4003H.**

Soln:

```
        MVI   B, 00H
        LXI   H, 4000H
        MOV   A, M
        INX   H
        ADD   M
        JNC   SKIP
        INC   B
SKIP:   INX   H
        MOV   M, A
        INX   H
        MOV   M, B
        RST1
```

Q 2) **WAP to add two BCD numbers stored in the locations 4000H and 4001H. Store the result at 4001H and 4002H.**

Soln:

```
        MVI   B, 00H
        LXI   H, 4000H
        MOV   A, M
        INX   H
        ADD   M
        DAA
        JNC   SKIP
        INC   B
SKIP:   INX   H
        MOV   M, A
        INX   H
        MOV   M, B
        RST1
```

Q 3) **WAP to add a series of 10 numbers stored from location 4000H. Store the result immediately after the series.**

Soln:

```
        SUB   A
        MOV   B, A
        LXI   H, 4000H
        MVI   C, 0AH

BACK:   ADD   M
        JNC   SKIP
        INC   B
SKIP:   INX   H
        DCR   C
        JNZ   BACK

        MOV   M, A
        INX   H
        MOV   M, B
        RST1
```

Cell: 98204 08217
BharatSir@hotmail.com

Q 4) **WAP to find the largest in a given series of 10 numbers starting from location 4000H. Store the result immediately after the series.**

Soln:

```
      LXI   H, 4000H
      MVI   C, 0AH
      SUB   A
BACK: CMP   M
      JNC   SKIP
      MOV   A, M
SKIP: INX   H
      DCR   C
      JNZ   BACK
      INX   H
      RST1
```

Q 5) **WAP to find the number of +ve, -ve and zeros in a given series of 10 numbers. Store the result immediately after the series.**

Soln:

```
      SUB   A
      MOV   B, A        ; B = No of zeros
      MOV   C, A        ; C = No of +ves
      MOV   D, A        ; D = No of -ves
      LXI   H, 4000H
      MVI   E, 0AH

BACK: CPM   M           ; A - M i.e. 00H - Current number
      JZ    ZERO        ; Current number must be zero
      JC    POSV        ; Current number must be greater than zero
NEGV: INR   D           ; Current number must be less than zero
      JMP   NEXT
POSV: INR   C
      JMP   NEXT
ZERO: INR   B

NEXT: INX   H
      DCR   E
      JNZ   BACK
      RST1
```

Q 6) **SORT ACSENDING a series of 10 numbers starting from location 2100H.**

Soln:

```
      MVI   B, 09H
BCK2: LXI   H, 2100H
      MVI   C, 09H

BCK1: MOV   E, M        ; Current number in E
      INX   H
      MOV   A, M        ; Next number in A
      CMP   E           ; A - E
      JNC   SKIP        ; If next number is greater then don't bother
      MOV   M, E        ; else exchange the two numbers
      DCX   H
      MOV   M, A
      INX   H

SKIP: DCR   C
      JNZ   BCK1
      DCR   B
      JNZ   BCK2
      RST1
```

BHARAT ACHARYA EDUCATION
Cell: 98204 08217
BharatSir@hotmail.com

BLOCK TRANSFER PROGRAMS:

Q 7) WAP to perform BLOCK TRANSFER of 10 bytes from location 2000H to location 3000H.

Soln:

```
      MVI   L, 09H
      LXI   B, 2000H
      LXI   D, 3000H

BACK: LDAX  B
      STAX  D
      INX   B
      INX   D
      DCR   L
      JNZ   BACK
      RST1
```

Q 8) WAP to perform OVERLAPPING BLOCK TRANSFER of 10 bytes from location 2000H to location 2004H.

Soln:

```
      MVI   L, 09H
      LXI   B, 2009H
      LXI   D, 200DH

BACK: LDAX  B
      STAX  D
      DCX   B
      DCX   D
      DCR   L
      JNZ   BACK
      RST1
```

Q 9) WAP to perform INVERTED BLOCK TRANSFER of 10 bytes from location 2000H to location 3000H.

Soln:

```
      MVI   L, 09H
      LXI   B, 2000H
      LXI   D, 3009H

BACK: LDAX  B
      STAX  D
      INX   B
      DCX   D
      DCR   L
      JNZ   BACK
      RST1
```

Cell: 98204 08217
BharatSir@hotmail.com

Q 10) WAP to MULTIPLY two 8-bit numbers stored in location 2000H and 2001H. Store the result at 2002H and 2003H.

Soln:

```
        LXI   H, 0000H
        LXI   D, 0000H

        LDA   2000H       ; Take multiplicand in A
        ADI   00H         ; Check for zero
        JZ    EXIT        ; If zero, then simply exit
        MOV   E, A        ; Else take multiplicand into E

        LDA   2001H       ; take multiplier in A
        ADI   00H         ; Check for zero
        JZ    EXIT        ; If zero, then simply exit
        MOV   C, A        ; Else take multiplier in C as the count

BACK:   DAD   D           ; Add multiplicand to itself
        DCR   C           ; C number of times
        JNZ   BACK

EXIT:   SHLD  2002H       ; Store the result as required
        RST1
```

Q 11) WAP to divide two 8-bit numbers stored at 2000H and 2001H. Store the result at 2002H and 2003H.

Soln:

```
        LXI   H, 2000H
        MOV   B, M        ; Take dividend in B
        INX   H
        MOV   A, M        ; Take divisor in A
        ADI   00H         ; Check for zero
        JZ    EXIT        ; If zero, its an INVALID operand. Simply exit.

        MOV   A, B        ; A gets the dividend
        MOV   B, M        ; B gets the divisor
        MVI   C, 00H      ; C will be the quotient

BACK:   CMP   B           ; A - B
        JC    DONE        ; no further steps as A < B
        SUB   B           ; A ← A - B
        INR   C
        JMP   BACK

DONE:   STA   2002H       ; Store remainder
        MOV   A, C
        STA   2003H       ; Store quotient
EXIT:   RST1
```

Cell: 98204 08217
BharatSir@hotmail.com

Q 12) WAP to generate a delay of 1 msec using 8085 working at 3 MHz.

Soln:

```
DLAY: MVI   B, XXH      ; 7 T-states … … … Count is calculated later
BACK: DCR   B           ; 4 T-states … … … Decrement Count
      JNZ   BACK        ; 10T (true) / 7T (false)
      RET               ; 10T-states
```

T_D = MT + [(Count)$_d$ x NT] – 3T

Here MT = Time outside the loop = 17T
NT = Time inside the loop = 14T

T_D = 17T + [(Count)$_d$ x 14T] – 3T

Required T_D = 1 msec = 10^{-3} sec
Given 1T = 0.333 µsec = 0.333 x 10^{-6} sec

Substituting the above values we get:

10^{-3} = 17x(0.333x10^{-6}) + [(Count)$_d$ x 14x(0.333x10^{-6})] – 3x(0.333x10^{-6})

Dividing by (0.333 x 10^{-6}) we get:

3003 = 17 + [(Count)$_d$ x 14] + 3

2983 = [(Count)$_d$ x 14]

213 = (Count)$_d$

Count = D5H

Similarly any other required delay can be achieved.
In this method, the max-delay achieved is 1.18 msec with count = FFH.
In the above calculations, the value of "1T" will change if operating frequency is anything other than 3 MHz.
If frequency is not given, then you can assume it to be 3 or 5 MHz.

Q 13) WAP to generate a delay of 0.5 msec using 8085 working at 3 MHz.

Soln: "Home-work"

Answer: Count = 6AH

Cell: 98204 08217
BharatSir@hotmail.com

Q 14) WAP to generate a SQUARE-WAVE of 1 KHz using SOD pin of 8085.

Soln:

```
BACK: MVI   A, 40H      ; SIM Command = 0100  0000
      SIM
      CALL  DLAY
      MVI   A, C0H      ; SIM Command = 1100  0000
      SIM
      CALL  DLAY
      JMP   BACK
```

For a square wave of 1 KHz, the time period is 1 msec.
Hence the required delay is of 0.5 msec.
Assume 8085 is working at 3 MHZ

```
DLAY: MVI   B, XXH      ; 7 T-states … … … Count is calculated later
BACK: DCR   B           ; 4 T-states … … … Decrement Count
      JNZ   BACK        ; 10T (true) / 7T (false)
      RET               ; 10T-states
```

T_D = MT + [(Count)$_d$ x NT] – 3T
Here MT = Time outside the loop = 17T
NT = Time inside the loop = 14T
T_D = 17T + [(Count)$_d$ x 14T] – 3T
Required T_D = 0.5 msec = 0.5 x 10^{-3} sec
1T = 0.333 µsec = 0.333 x 10^{-6} sec
Substituting the above values we get:
$0.5x10^{-3} = 17x(0.333x10^{-6}) + [(Count)_d \times 14x(0.333x10^{-6})] - 3x(0.333x10^{-6})$
Count = 6AH

Q 15) WAP to transfer the value 35H serially with one start bit "0" and one stop bit "1".

Soln: Serial communication happens bit by bit starting from the LSB.
As per the question, we need to send the start bit (0), then the data and finally the stop bit (1).
Hence a total of 10 bits will move out as follows:

```
  0              1 0 1 0 1 1 0 0           1
Start      8-data bits in reverse order    Stop
```

```
      MVI   A, 40H      ; start bit (0)
      SIM
      MVI   A, C0H      ; send a "1"
      SIM
      MVI   A, 40H      ; send a "0"
      SIM
      MVI   A, C0H      ; send a "1"
      SIM
      MVI   A, 40H      ; send a "0"
      SIM
      MVI   A, C0H
      SIM               ; send a "1"
      SIM               ; send a "1" again
      MVI   A, 40H      ; send a "0"
      SIM
      SIM               ; send a "0" again
      MVI   A, C0H      ; send a "1" as the stop bit
      SIM
      RST1
```

Cell: 98204 08217
BharatSir@hotmail.com

Q 16) WAP to transfer the value 35H serially with one start bit (0) and one stop bit (1) at a Baud rate of 2400. Assume 8085 is working at 3 MHz.

Soln: Baud rate is the rate at which data is send.
BR = 2400 means 2400 bits have to be sent in 1 second.
Hence the delay between sending two bits is of (1/2400) seconds.
$T_D = 0.41667 \times 10^{-3}$ sec

```
DLAY: MVI   B, XXH      ; 7 T-states … … … Count is calculated later
BACK: DCR   B           ; 4 T-states … … … Decrement Count
      JNZ   BACK        ; 10T (true) / 7T (false)
      RET               ; 10T-states
```

$T_D = MT + [(Count)_d \times NT] - 3T$
Here MT = Time outside the loop = 17T
NT = Time inside the loop = 14T
$T_D = 17T + [(Count)_d \times 14T] - 3T$

Required $T_D = 0.41667$ msec $= 0.41667 \times 10^{-3}$ sec
$1T = 0.333$ µsec $= 0.333 \times 10^{-6}$ sec
Substituting the above values we get:
$0.41667 \times 10^{-3} = 17 \times (0.333 \times 10^{-6}) + [(Count)_d \times 14 \times (0.333 \times 10^{-6})] - 3 \times (0.333 \times 10^{-6})$
Count = 58H

A total of 10 bits will move out as follows:

```
  0              1 0 1 0 1 1 0 0           1
Start       8-data bits in reverse order   Stop
```

```
   MVI   A, 40H      ; start bit (0)
   SIM
   CALL  DLAY
   MVI   A, C0H      ; send a "1"
   SIM
   CALL  DLAY
   MVI   A, 40H      ; send a "0"
   SIM
   CALL  DLAY
   MVI   A, C0H      ; send a "1"
   SIM
   CALL  DLAY
   MVI   A, 40H      ; send a "0"
   SIM
   CALL  DLAY
   MVI   A, C0H
   SIM               ; send a "1"
   CALL  DLAY
   SIM               ; send a "1" again
   CALL  DLAY
   MVI   A, 40H      ; send a "0"
   SIM
   CALL  DLAY
   SIM               ; send a "0" again
   CALL  DLAY
   MVI   A, C0H      ; send a "1" as the stop bit
   SIM
   CALL  DLAY
   RST1
```

Cell: 98204 08217
BharatSir@hotmail.com

Q 17) WAP to transfer a RANDOM NUMBER stored at location 2000H serially with a start bit (0) and a stop bit (1).

Soln:

```
        LDA   2000H       ; read number in A
        MOV   B, A        ; store number in B
        MVI   C, 08H      ; count

        MVI   A, 40H      ; send "zero" as the start bit
        SIM

BACK:   MOV   A, B        ; get the number
        RRC               ; get its LSB in Carry flag
        MOV   B, A        ; store rotated number in B for next iteration
        JC    ONE         ; if carry flag is "1" then go to send a "one"
        MVI   A, 40H      ; else send a "zero"
        SIM
        JMP NEXT
ONE:    MVI   A, COH      ; send a "one"
        SIM
NEXT:   DCR C
        JNZ BACK          ; repeat for all 8-bits

        MVI A, C0H        ; send a "one" as the stop bit
        SIM
        RST1
```

Q1: Write a program to multiply 2 8-bit numbers stored at 2000h and 2001h. Store the result at 2002H and 2003H

```
        LXI H, 0000H        ; Result will be in HL, initialized to 0
        LXI D, 0000H        ; DE initialized to 0

        LDA 2000H           ; Take 1st operand
        ADI 00H             ; For Zero check
        JZ Store            ; For Zero check
        MOV E,A             ; DE = 1st operand

        LDA 2001H           ; Take 2nd operand
        ADI 00H             ; For Zero check
        JZ Store            ; For Zero check
        MOV C,A             ; C = 2nd operand

Back:   DAD D               ; Add the 1st operand to HL, HL = HL + DE
        DCR C               ; Decrement the 2nd operand
        JNZ Back            ; Loop till C becomes 0

Store:  SHLD 2002H          ; Store the result

        HLT                 ; End your program
```

Q2: Write a program to divide an 8-bit number stored at 2000h by another stored at 2001h. Store the result at 2002H (Q) and 2003H (R)

```
        LDA 2001H       ; Take divisor in A
        ADI 00H         ; For Zero check
        JZ Exit         ; If Zero, simply exit the program
        MOV C,A         ; C = divisor
        MVI E,00H       ; E = 0 (this will be the quotient)
        LDA 2000H       ; A = Dividend

Back:   CMP C           ; Compare A and C by doing A-C
        JC Next         ; If A < C then move out of the loop
        SUB C           ; A ← A – C, actually do the subtraction
        INR E           ; Increment Quotient
        JMP Back        ; Loop back

Next:   STA 2003H       ; Store the remainder from A to 2003H
        MOV A,E         ; Move Quotient from E to A
        STA 2002H       ; Store the quotient from A to 2002H

Exit:   HLT             ; End your program
```

https://www.bharatacharyaeducation.com
Learn...
8085 | 8086 | 80386 | Pentium |
8051 | ARM7 | COA

Fees: 1199/-
Duration: 6 months
Activation: Immediate
Certification: Yes
Free: PDFs of theory explanation
Free: VIVA questions and answers
Free: PDF of Multiple Choice Questions

Start Learning... NOW!
https://www.bharatacharyaeducation.com

Order my Books here...

8086 Microprocessor book
Link: https://amzn.to/3qHDpJH

8051 Microcontroller book
Link: https://amzn.to/3aFQkXc

#bharatacharya
#bharatacharyaeducation
#8086 #8051 #8085 #80386 #pentium
#microprocessor #microcontrollers

8085 STACKS AND SUBROUTINES

STACKS

A **stack** is a **part of the Memory** that **operates in LIFO** (Last In First Out) manner.
In 8085 the **Stack** can **located anywhere within the 64 KB memory**.
The **top of the Stack** is **pointed by** the **SP** (Stack Pointer) register.
P.S.: The SP contains the **Address** of the top of the Stack and **not** the **top of the Stack**.
By initializing SP with any suitable value, the programmer can decide where he/she would like to start the stack. Remember SP gets decremented on every PUSH operation. This means the stack grows upwards. Hence it is advisable to start the stack at a lower location so that it does not overwrite on any other useful information.

VIVA Question: Why is LXI SP, FFFFH preferred?

LXI SP, FFFFH is preferred due to two reasons,
- It gives the Stack max space to grow.
- It prevents the Stack from overwriting on programs in the memory.

Instructions related to stacks

- **LXI SP**, 16 bit value Eg LXI SP 4000.
- **SPHL**
- **INX SP**
- **DCX SP**
- **PUSH** Rp Eg PUSH B
- **POP** Rp Eg POP B
- **CALL** 16 bit address Eg CALL 2000
- **RSTn** Eg RST1
- **RET**
- **XTHL**

Note

In the exam, explain any 2 of the above instructions as an introduction to stacks. Preferably PUSH and POP.

USES of STACKS

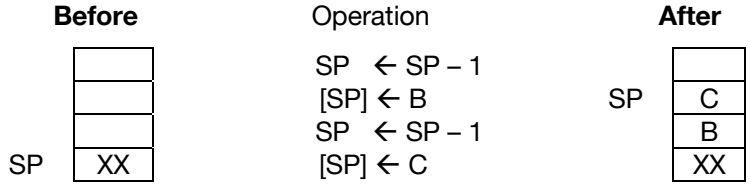
Stacks are used in the following ways:

1) To store data during programs

Stack can be used for **storing data by** the **programmer** as the number of General Purpose registers is very less. The programmer, at any point of time during the program, can store data in the stack and then retrieve it later.

Data can be Pushed into the Stack using Push instruction as below:

Eg : **PUSH B** ; Contents of BC Pair are pushed onto the top of the stack.



Similarly data can also be removed from the stack using the POP operation as:

Eg : **POP B** ; Contents of the top of the stack are popped into BC Pair.

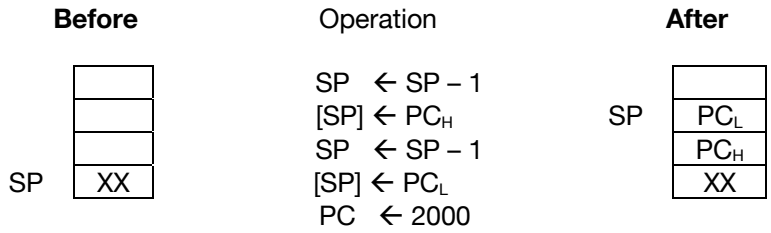
2) To Store return address during CALL instructions

The **µP uses** the Stack **for storing** the **return address** during **CALL** instruction.

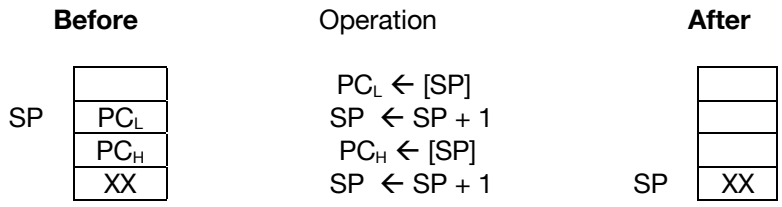
During a CALL instruction, **µP** first pushes the return address (i.e. the current contents of PC) into the stack and then loads the branch address into PC, to branch to the subroutine.

Inside the subroutine, when the **µP** gets a RET instruction, it pops back the return address from the stack, and thus successfully returns to the main program.

Eg: **CALL 2000**



Eg: **RET**



3) To Read/Write the contents of the Flag register

The programmer can Read / Write the contents of the Flag register using the Program Status Word **(PSW)**. The PSW consists of the Accumulator as the higher byte and the Flag register as the lower byte. The PSW is available only for PUSH and POP instructions.

To Read the contents of Flag register

- The programmer pushes PSW into the stack and then pops it into any register pair (BC).
- The contents of the flag register are thus available in the C register (lower register).

Eg:

PUSH PSW

SP	F	;Accumulator and Flag
	A	;Reg Pushed into Stack
	XX	

POP B

		;Contents of Flag Reg
		;come into C-Reg and
SP	XX	;can be read.

To Write into the Flag register

- The programmer loads the appropriate byte into the lower register of a register pair (eg: C reg of BC pair).
- Then the register pair is pushed into the stack.
- These contents are then popped into the PSW, and thus the byte that was originally loaded into C register is now **written** into the Flag register.

Eg:

PUSH B

SP	C	;B and C Registers are
	B	;Pushed into Stack
	XX	

POP PSW

		;Contents of C Reg
		;come into Flag Reg
SP	XX	

Note

This is the **ONLY method** by which the programmer can **write into** the **Flag register**.

4) To Pass parameters to a Sub-Routine

The programmer can use the Stack to pass parameters to Sub-Routines.
Before calling the Sub-Routine the **parameter** is **Pushed** into the Stack and then **inside** the **Sub-Routine** the **parameter** is **Popped from** the **Stack**.

Eg:

```
LXI D 1200 ; Parameter 1020 Pushed        SUB: POP H  ; Return address taken in HL
PUSH D     ; into the Stack                   POP B ; Parameter is accepted into
CALL SUB                                      .      ; BC pair.
ADD B                                         .
    .                                         .
                                              PCHL   ; PC gets the return address
                                                     ; from HL
HLT
```

#Please refer Bharat Sir's video for a detailed explanation of this

Special Note:

If you are learning this by piracy, then you are not my student. You are simply a thief!
#PoorUpbringing

SUBROUTINES

- Subroutines are parts of a program, which can be re-executed by the programmer.
- Subroutines are Called (invoked) using the CALL instruction; control returns to the main program using the RET instruction.
- Subroutines generally perform tasks, which are used regularly by the program such as numerical calculations, Interrupt Service Routines (ISR) etc.

Subroutines are useful in the following ways:

1. Causes **Code Re-Usability** hence reduces the **size** of the Program and also **saves time**.
2. Makes the program easy to **Maintain** as it becomes **Modular.**

Passing Parameters to Subroutines

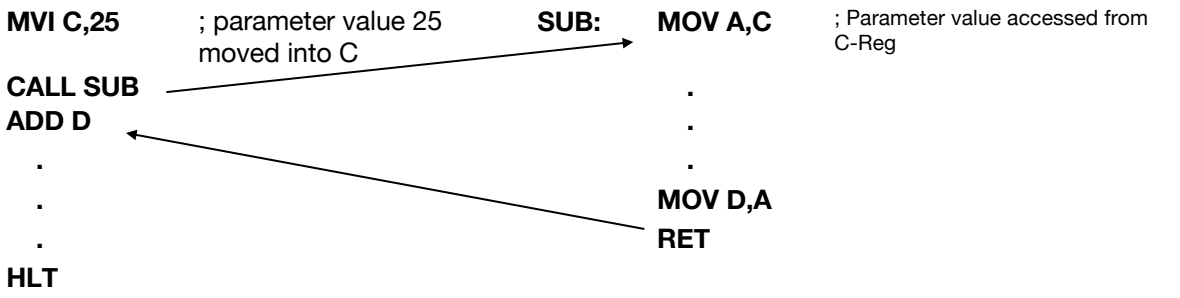
Passing parameters to Subroutines is the procedure of passing some values from the main program to the Subroutines. Passing parameters is very important as it **improves the flexibility of the Subroutine**.

In 8085 there are 4 methods of passing parameters to Subroutines.

1) Using Registers

The parameter value to be passed is moved /loaded into the register before calling the SubRoutine. The SubRoutine accesses the value from the same register.

Eg:

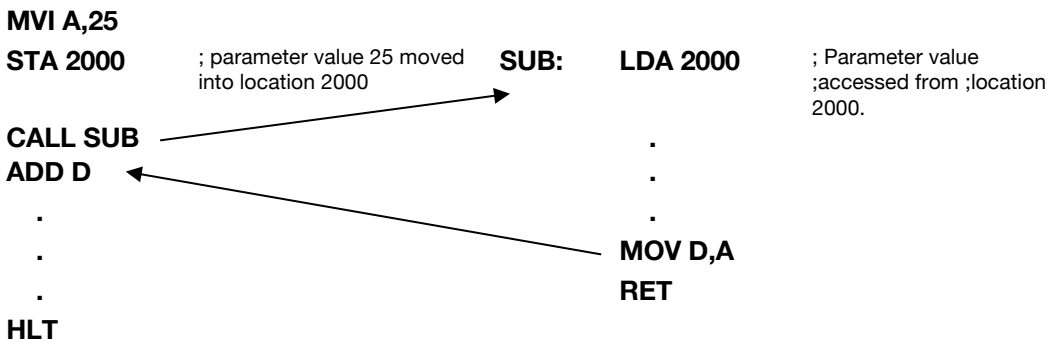


This is the **simplest method** of passing parameters.
The only main **drawback** is that it **occupies** the **general-purpose registers** available to the programmer whish are already very few. **Also** the **number** of parameters passed is **restricted by** the number of **registers available**.

2) Using Memory

The parameter value to be passed is stored into the Memory Location before calling the SubRoutine.
The SubRoutine accesses the value from the same memory location.

Eg:

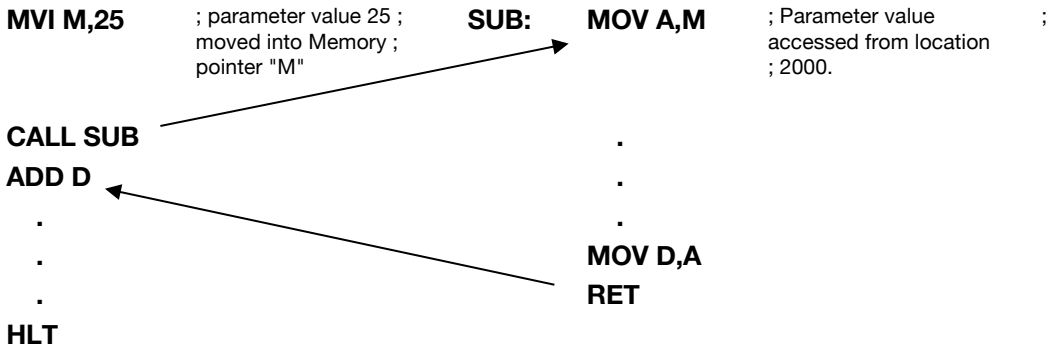


If **more parameters** are to be passed, then **consecutive memory locations** can be used (which are **plenty**), thus **eliminating** the **drawbacks** of the previous method.
The **only drawback** here is that the programmer needs to **remember** the **memory locations** associated with each subroutine. When the number of subroutines is large, this can be troublesome.

3) Using Memory Pointer "M"

The parameter value to be passed is moved into the Memory pointer **"M"** before calling the Subroutine.
☺ In case of doubts, contact Bharat Sir: Connect with us on our Official WhatsApp number: +919136428051
The HL pair at this point of time may contain any random value.
The SubRoutine accesses the value from M.
Care has to be taken that after the value is put into M the value of **HL pair** should not change until the SubRoutine accesses the value of M.

Eg:



Thus the memory is used but without remembering the memory location.
i.e. In the above example, we still **do not know** the value of **HL** pair, and thus the **address** of the memory location used (thus avoiding the previous drawback).
But, when **more** than one values have to be passed, this method becomes **troublesome**.

4) Using Stack ※

The **parameter** value **to be passed** is **Pushed** into the Stack before calling the SubRoutine.
The SubRoutine **Pops** the **passed parameter** from the Stack.
When the **µP** calls the SubRoutine it **pushes** the **return address** into the stack.
This address appears above the passed parameter in the stack.
Due to the **LIFO** property of the stack we cannot access the passed parameter unless we pop the return address.
Hence inside the SubRoutine **firstly**, the **return address** is **popped into** the **HL** pair.
Then the **passed parameter** is **popped** into the BC pair.
Now in this case the **RET** instruction **cannot be used** to come back to the main program as the top of the stack no longer contains the return address (as we had taken it out in the HL pair).
Thus the return address is obtained from the HL pair using the **PCHL** instruction which **causes** the control to **return to the main program**.

Eg:

```
LXI D 1200 ; Parameter 1020 Pushed
PUSH D     ; into the Stack
CALL SUB
ADD B
   .
   .
   .
HLT;

SUB:POP H;   Return address taken in HL
POP B ;      Parameter is accepted into
   .         BC pair.
   .
   .
PCHL ;       PC gets the return address
;            from HL
```

#Please refer Bharat Sir's video for a detailed explanation of this

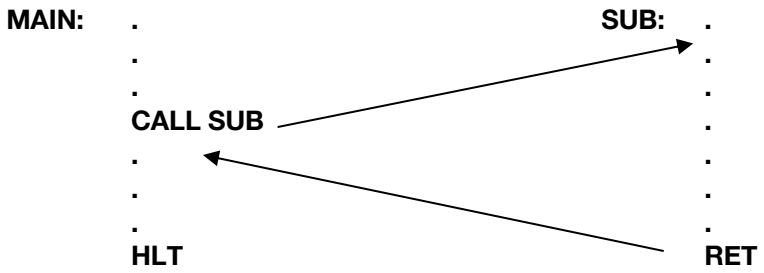
Special Note:

If you are learning this by piracy, then you are not my student. You are simply a thief!
#PoorUpbringing

TYPES OF SUB-ROUTINES

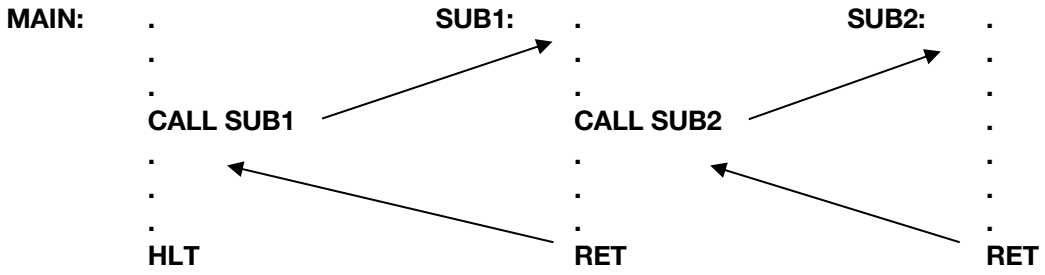
1. Simple Subroutines

When a SubRoutine does not call another SubRoutine it is called a Simple SubRoutine.



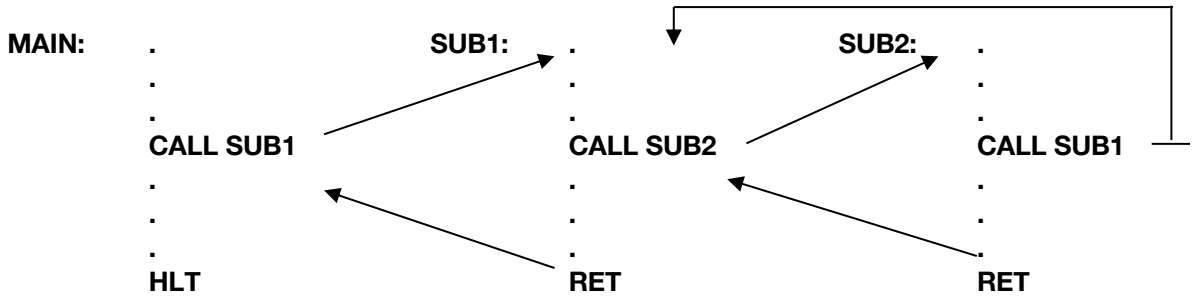
2. Nested Subroutines

When a SubRoutine calls another SubRoutine it is called a Nested SubRoutine.



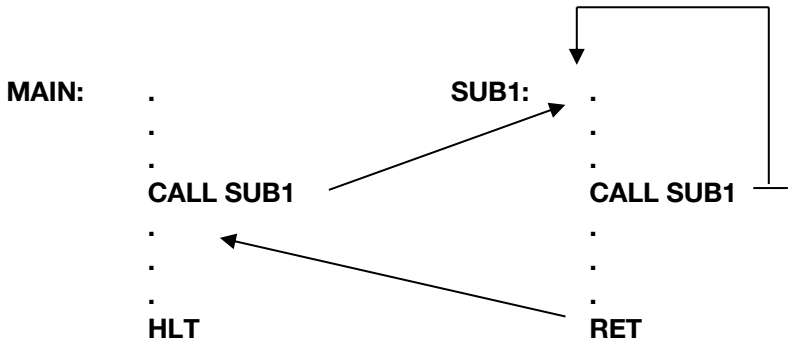
3. Re-entrant Subroutines

When a SubRoutine is re-entered by another SubRoutine it is called a Re-entrant SubRoutine.



4. Recursive Subroutines

When a SubRoutine calls itself, it is called a Recursive SubRoutine.



Bharat Acharya Education
Learn...
8085 | 8086 | 80386 | Pentium | 8051 | ARM7 | COA

Fees: 1199/- | Duration: 6 months | Activation: Immediate | Certification: Yes
Free: PDFs of Theory explanation, VIVA questions and answers, Multiple-Choice Questions

Start Learning... NOW!
www.BharatAcharyaEducation.com

Order our Books here...

8086 Microprocessor
Link: https://amzn.to/3qHDpJH

8051 Microcontroller
Link: https://amzn.to/3aFQkXc

Official WhatsApp number:
+91 9136428051

8085 DELAYS AND DELAY ROUTINES

DELAYS

A delay is a time gap between two events. Delays are very useful in computer programs. Something as simple as a traffic light needs delays after every state.

Delays in a computer system can be generated in two ways:
1) Hardware delays
2) Software delays

	HARDWARE DELAYS	SOFTWARE DELAYS
1	Caused by external hardware (Timer IC like 8254)	Caused by a software delay routine (program)
2	μP is not involved in causing the delay and hence is free for other applications.	μP is busy as it is executing the delay routine so cannot be used otherwise.
3	Multiple delay routines are possible	Multiple delay routines are not possible
4	Flexibility is low as h/w is involved.	Flexibility is high as delay caused by s/w.
5	External h/w required so circuit becomes more complex	External h/w not required so circuit is simple and less expensive
6	Circuit is More Expensive	Circuit is Less Expensive.

Note

In this section we will focus on Software delay routines. After a few lectures, we will learn Timer chips like 8253/54. There we will focus on Hardware delays.

SOFTWARE DELAY ROUTINES

Software delays are produced in programs by one of the various software **"Delay Routines"**.

As the amount of delay required varies from application-to-application different types of delay routines are present, as follows:

1) **Using NOP Instruction**

2) **Using One 8-bit register**

3) **Using One 16-bit register**

Note

The above three methods are the most standard ways of producing delays.
We can also produce delays using Nested loops or any other user defined method.
Though not advisable to be used in college exams, they too are interesting to learn.

Nested delay routines: (Not standard)

4) Using Two 8-bit registers

5) Using One 8-bit register and One 16-bit register

6) Using Two 16-bit registers

1) Using NOP Instruction

Eg: NOP;

1-byte instruction

Opcode fetch --- 4 T-states.

T_D = 4T.

T = 1/(Clk freq)

∴ assuming 8085 working at 3 MHz, T = 1/(3 MHz) = 0.333 μ sec.

∴ T_D = 4 × 0.333 = 1.332 μ sec.

∴ T_D = 1.332 μ sec.

This is the maximum delay that can be achieved by writing a NOP instruction.

2) Using One 8-bit register

Delay:	**MVI B**, 8-bit count	7T
Loop:	**DCR B**	4T
	JNZ Loop	10/7T
	RET	10T

NT | MT

T_D = MT + [(Count)$_d$ × NT] – 3T

NT = No of T-states inside the loop {here NT = 10T + 4T = **14T**}

MT = No of T-states outside the loop {here MT = 7T + 10T = **17T**}

$Count_{max}$ = 255 {8-bit count in decimal}

∴ $T_{D\,max}$ = 17T + [255 × 14T] – 3T

∴ $T_{D\,max}$ = 3584T.

Assuming 8085 working at 3 MHz i.e. T = 333 n sec.

∴ $T_{D\,max}$ = 1.18 m sec.

Very Important:

8-bit delays are extremely useful for the further topics where we need delays to generate square waves, perform serial communications etc.

Special Note:

If you are learning this by piracy, then you are not my student. You are simply a thief!
#PoorUpbringing

3) Using One 16-bit register

Delay:	**LXI B**, 16-bit count	10T
Loop:	**DCX B**	6T
	MOV A, B	4T
	ORA C	4T
	JNZ Loop	10/7T
	RET	10T

N T | M T

$\mathbf{T_D = MT + [(Count)_d \times NT] - 3T}$

Here **NT** = 6T + 4T + 4T + 10T = **24T**

MT = 10T + 10T = **20T**

$Count_{max}$ = 65535 {16-bit count in decimal}

$\therefore T_{D\,max} = 20T + [65535 \times 24T] - 3T$

$\therefore T_{D\,max} = 1572057T.$

Assuming 8085 working at 3 MHz i.e. T = 333 n sec.

$\therefore \mathbf{T_{D\,max} = 0.525\ sec.}$

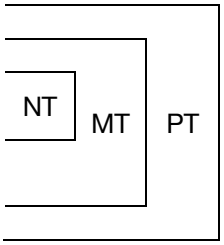
Note

In the above method, you have learnt how to decrement a 16-bit register paid and check for zero. Remember this trick for other programs as well – Bharat Acharya.

Non-standard but interesting delay methods to learn just for knowledge…

4) Using Two 8-bit registers

Delay:	**MVI C**, Count2	7T
Loop2:	**MVI B**, Count1	7T
Loop1:	**DCR B**	4T
	JNZ Loop1	10/7T
	DCR C	4T
	JNZ Loop2	10/7T
	RET	10T



$\mathbf{T_D = PT + [(Count2)_d \times T_{loop1}] - 3T}$

$\mathbf{T_{loop1} = MT + [(Count1)_d \times NT] - 3T}$

NT = No of T-states inside loop1 {here NT = 4T + 10T = **14T**}.

MT = No of T-states outside loop1 but inside loop2 {MT = 7T + 4T + 10T= **17T**}

PT = No of T-states outside loop2 {here PT = 10T + 7T = **17T**}.

$Count1_{max}$ = 255 {8-bit count in decimal}

$Count2_{max}$ = 255 {8-bit count in decimal}

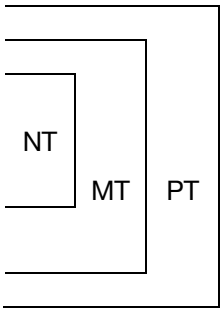
∴ $T_{D\,max}$ = 914954T.

Assuming 8085 working at 3 MHz i.e. T = 333 n sec.

∴ $\mathbf{T_{D\,max}}$ **= 0.304 sec.**

5) Using One 8-bit register and One 16-bit register

Delay:	**MVI D**, Count2	7T
Loop2:	**LXI B**, Count1	10T
Loop1:	**DCX B**	6T
	MOV A, C	4T
	ORA B	4T
	JNZ Loop1	10/7T
	DCR D	4T
	JNZ Loop2	10/7T
	RET	10T



$\mathbf{T_D = PT + [(Count2)_d \times T_{loop1}] - 3T}$

$\mathbf{T_{loop1} = MT + [(Count1)_d \times NT] - 3T}$

Here **NT** = 6T + 4T + 4T + 10T = **24T**.

MT = MT = 10T + 4T + 10T = **24T**.

PT = 7T + 10T = **17T**.

$Count1_{max}$ = 65535 {16-bit count in decimal}

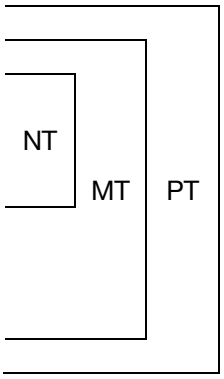
$Count2_{max}$ = 255 {8-bit count in decimal}

Assuming 8085 working at 3 MHz i.e. T = 333 n sec.

$\therefore \mathbf{T_{D\,max} = 133.69\ sec.}$

6) Using Two 16-bit registers

Delay:	**LXI B**, Count2	10T
Loop2:	**LXI D**, Count1	10T
Loop1:	**DCX D**	6T
	MOV A, E	4T
	ORA D	4T
	JNZ Loop1	10/7T
	DCX B	6T
	MOV A, C	4T
	ORA B	4T
	JNZ Loop2	10/7T
	RET	10T



$\mathbf{T_D = PT + [(Count2)_d \times T_{loop1}] - 3T}$

$\mathbf{T_{loop1} = MT + [(Count1)_d \times NT] - 3T}$

Here **NT** = 6T + 4T + 4T + 10T = **24T**.

MT = MT = 10T + 6T + 4T + 4T + 10T = **34T**.

PT = 10T + 10T = **20T**.

$Count1_{max}$ = 65535 {16-bit count in decimal}

$Count2_{max}$ = 65535 {16-bit count in decimal}

Assuming 8085 working at 3 MHz i.e. T = 333 n sec.

$\therefore$ **$T_{D\ max}$ = 9 hours, 32 min, 24 sec.**

Delay Method	Max Delay
NOP	**1.332 μ sec**
One 8-bit register	**1.18 m sec**
One 16-bit register	**.525 sec**
Two 8-bit registers	.304 sec
One 8-bit / one 16-bit reg	133.69 sec
Two 16-bit register	9 hrs, 32 min, 24 sec

Note

All these calculations are w.r.t. 8085 operating at 3 MHz.
In case in the exam, the frequency is different then calculate 1T = 1/(Clk. freq.), and then calculate the appropriate delay.
☺ In case of doubts, contact us on our Official WhatsApp number: +919136428051

Bharat Acharya Education
Learn...
8085 | 8086 | 80386 | Pentium | 8051 | ARM7 | COA

Fees: 1199/- | Duration: 6 months | Activation: Immediate | Certification: Yes
Free: PDFs of Theory explanation, VIVA questions and answers, Multiple-Choice Questions

Start Learning... NOW!
www.BharatAcharyaEducation.com

Order our Books here...

8086 Microprocessor
Link: https://amzn.to/3qHDpJH

8051 Microcontroller
Link: https://amzn.to/3aFQkXc

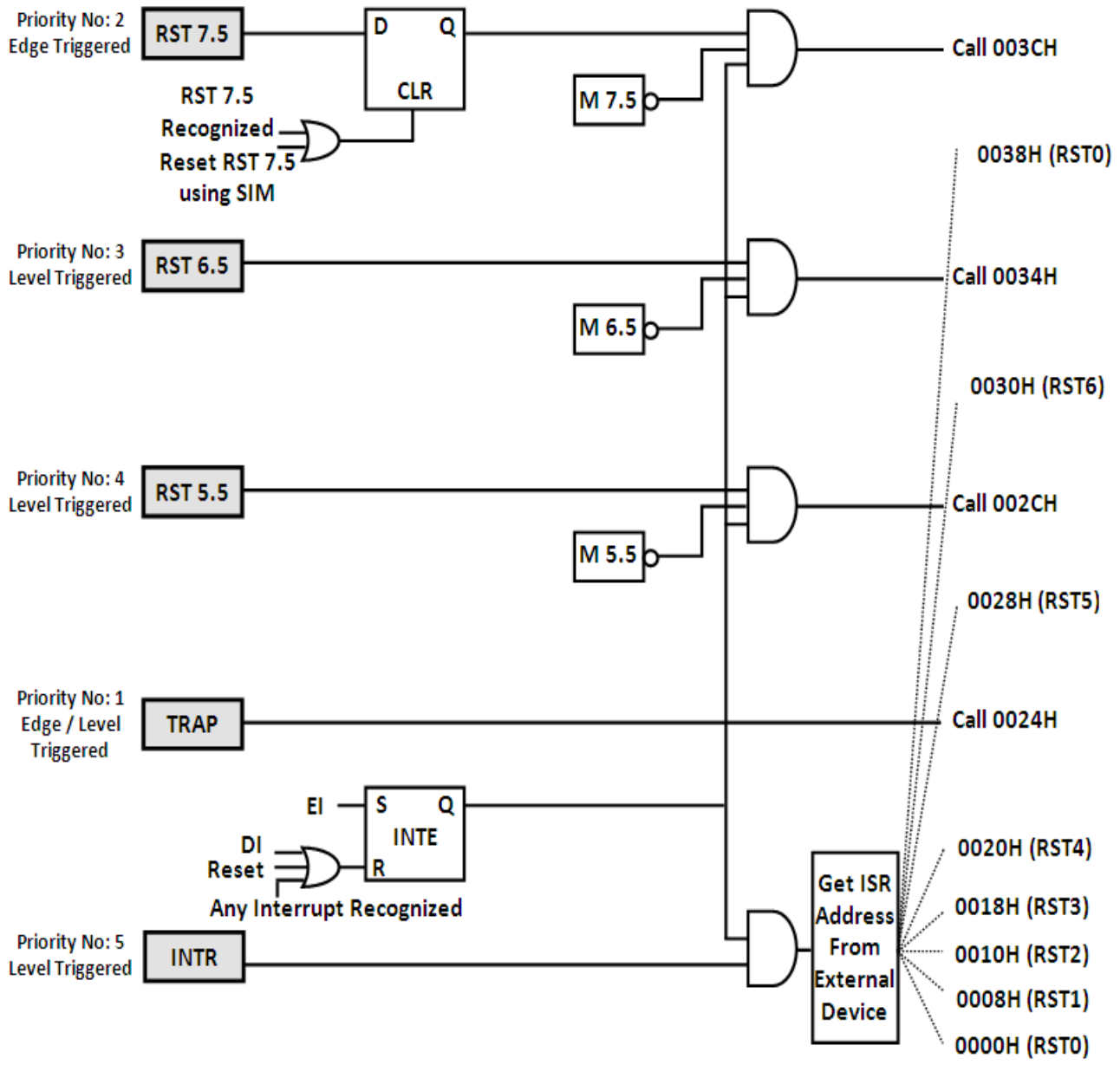
Official WhatsApp number:
+91 9136428051

8085 INTERRUPTS

- An interrupt is a **special condition** that makes the µP execute an ISR.
- **µP services** the interrupt **by executing** a subroutine called as the **Interrupt Service Routine**.
- The **µP checks** for interrupts **during every instruction**.
- When an **interrupt occurs**, the **µP first finishes the current instruction**.
- It then **Pushes** the address of the next instruction **(contents of PC) on the STACK**.
- It **resets** the **INTE flip-flop** so that **no more interrupts are recognized**.
- Thereafter the program control transfers to the **address of** the Interrupt Service Routine (**ISR**) and the **µP** thus **executes the ISR**.

	SOFTWARE INTERRUPTS	HARDWARE INTERRUPTS
1	They are caused by writing an instruction.	They occur as signals on external pins.
2	8085 has 8 software interrupt instructions called RST N, where N can be any value from 0…7. Hence we have RST 0 … RST 7.	8085 has 5 hardware interrupt pins: TRAP RST 7.5 RST 6.5 RST 5.5 INTR
3	Software interrupts cannot be masked or disabled.	All Hardware interrupts can be disabled except TRAP.
4	All Software interrupts have the same priority.	Hardware interrupts have the following priority order: TRAP … 1st (Highest) RST 7.5 … 2nd RST 6.5 … 3rd RST 5.5 … 4th INTR … 5th(Lowest)
5	All Software interrupts are Vectored	All Hardware interrupts are Vectored except INTR.
6	The Vector addresses are as follows: RST 0 … 0000 H RST 1 … 0008 H RST 2 … 0010 H RST 3 … 0018 H RST 4 … 0020 H RST 5 … 0028 H RST 6 … 0030 H RST 7 … 0038 H	The Vector addresses are as follows: TRAP … 0024 H RST 7.5 … 003C H RST 6.5 … 0034 H RST 5.5 … 002C H INTR … obtain ISR address from device

INTERRUPT STRUCTURE OF 8085



Software Interrupts:
Interrupts that are **initiated by writing an Instruction** (software) are called as software interrupts.
8085 has 8 software interrupts:
RSTn where n = 0,1,2, … 7 i.e. RST0, RST1 … upto RST7.
– This instruction causes a **service routine** to be Called from the **address (n*8)**.
– Hence, if RST1 occurs then the program control moves to location 0008 (1*8 = 0008).
The respective addresses for software interrupts are given below.

S/W Interrupt	ISR Address
RST0	00**0**0H
RST1	00**0**8H
RST2	00**1**0H
RST3	00**1**8H
RST4	00**2**0H
RST5	00**2**8H
RST6	00**3**0H
RST7	00**3**8H

Hardware Interrupts:
Interrupts that are initiated through a hardware pin are called as hardware interrupts.
8085 supports the following hardware interrupts:
- TRAP
- RST 7.5
- RST 6.5
- RST 5.5
- INTR

Vectored Interrupts:
- Interrupts that **have** a **FIXED Address** for their ISR are called as Vectored Interrupts.
- **Eg: TRAP** is a vectored interrupt. Its vector address is 0024H.

Non-Vectored Interrupts:
- Interrupts that **have** a **Variable Address** for their ISR are called as Non-Vectored Interrupts.
- **Eg: INTR** is a Non-Vectored Interrupt.

Methods of preventing an interrupt from occurring.
- **MASK Individual Bits** through **SIM** Instruction
- **Disable all** Interrupts through **DI** Instruction

MASKING:
- We can prevent an interrupt from occurring by MASKING its individual bit through **SIM Instruction**.
- If an interrupt is masked it will not be serviced.
- One of the **main advantages** of masking as opposed to disabling interrupts is that by masking we can **selectively disable a particular interrupt** while keeping other interrupts active**,** whereas through DI instruction all interrupts are disabled.
- ONLY **RST 7.5, RST 6.5 and RST 5.5** can be masked by this method.

DISABLING INTERRUPTS:

- Interrupts can be disabled through the **DI** Instruction.
- This instruction resets the INTE Flip Flop and hence none of the interrupts can occur (**Except TRAP**).
- I.e. INTE F/F ← 0.
- Once disabled, these interrupts can be **re-enabled** through **EI** instruction, which sets the INTE Flip Flop.
- I.e. INTE F/F ← 1.

Hardware Interrupts (In detail)

TRAP

- TRAP has the **highest priority.**
- It is **Edge as well as Level triggered** hence the signal must go High and also Remain high for some time for it to be recognized. This prevents any noise signal from being accepted.
- It is a Non-Maskable Interrupt i.e. it can **neither be masked nor be disabled.**
- It is a vectored interrupt and has a **vector address of 0024H.**

RST 7.5

- RST 7.5 has the **priority lower than TRAP.**
- It is **Edge triggered.**
- It is a **Maskable Interrupt** i.e. it can be masked through the **SIM Instruction**.
- It can also be disabled though the **DI Instruction**.
- It is a vectored interrupt and has a **vector address of 003CH.**
- RST 7.5 can also be **reset** through the **R 7.5** bit in the **SIM** Instruction irrespective of whether it is Masked or not.

RST 6.5

- RST 6.5 has the **priority lower than RST 7.5.**
- It is **Level triggered.**
- It is a **Maskable Interrupt** i.e. it can be masked through the **SIM Instruction**.
- It can also be disabled though the **DI Instruction**.
- It is a vectored interrupt and has a **vector address of 0034H.**

RST 5.5

- RST 5.5 has the **priority lower than RST 6.5.**
- It is **Level triggered.**
- It is a **Maskable Interrupt** i.e. it can be masked through the **SIM Instruction**.
- It can also be disabled though the **DI Instruction**.
- It is a vectored interrupt and has a **vector address of 002CH.**

INTR

- INTR has the **priority lower than RST 5.5.**
- It is **Level triggered.**
- It can only be disabled though the **DI Instruction**.
- **It cannot be masked through the SIM Instruction**.
- It is a **Non-Vectored** interrupt.

Response to INTR:

- When INTR occurs the µP, in response, issues the **first INTA** cycle.
- The **External Hardware sends** an **opcode**, which can be of **RSTn** Instruction **or** of **CALL** instruction.

a) **If** opcode of **RSTn** is sent by the external hardware
   - The µP calculates the address of the ISR as **n*8.**

b) **If** opcode of **CALL** is sent:
   - As Call is a **3-Byte Instruction** the µP send **2 more INTA** signals
   - In response to the **2nd and the 3rd INTA** cycle the external hardware returns the **lower and the higher byte of the address** of the ISR respectively.

This is how the address is determined when INTR occurs.

INTA : (Interrupt Acknowledge)

- This is an **active low acknowledge** signal going out of 8085.
- This signal is **given in response to** an interrupt on **INTR ONLY**.
- After the **first INTA** is given, the interrupting **peripheral sends an opcode**.
- **If** the **Opcode** is **of RSTn**, then the **ISR address is calculated as n × 8.**
- **If** the **Opcode** is **of CALL**, then **two more INTA signals** are given and, the lower byte, and then the higher byte of the ISR address are sent by the peripheral.
- *#Please refer Bharat Sir's video at www.BharatAcharyaEducation.com for understanding this..*

EI and DI Instructions

INTE F/F : (Interrupt Enable Flip Flop)

- This flip-flop decides if interrupts are enabled in the µP i.e. **if it is set, all interrupts are enabled**.
- It is **set by** the **EI Instruction**.
- It is **reset** in the following 3 ways:

i. **If µP is reset**.
- If **DI instruction** is executed.
- If **any other interrupt is recognized** by the µP. In this case the INTE F/F is later set in the ISR by EI.

☺ In case of doubts, contact Bharat Sir: - 98204 08217.

The INTE F/F affects all interrupts **EXCEPT TRAP**, as it cannot be disabled.

Note

On reset by default interrupts are disabled.
If we don't write EI in our program hardware interrupts will not be serviced except TRAP.

Interrupt	Priority	Triggering	Maskable by SIM	Disabled by DI	Vectored	Vector Address
TRAP	**1**	**Edge / Level**	**No**	**No**	Yes	0024 H
RST 7.5	2	**Edge**	Yes	Yes	Yes	003C H
RST 6.5	3	Level	Yes	Yes	Yes	0034 H
RST 5.5	4	Level	Yes	Yes	Yes	002C H
INTR	5	Level	**No**	Yes	**No**	**Get ISR Address from External Hardware**

Special Note:

If you are learning this by piracy, then you are not my student. You are simply a thief! #PoorUpbringing

Bharat Acharya Education
Learn...
8085 | 8086 | 80386 | Pentium | 8051 | ARM7 | COA

Fees: 1199/- | Duration: 6 months | Activation: Immediate | Certification: Yes
Free: PDFs of Theory explanation, VIVA questions and answers, Multiple-Choice Questions

Start Learning... NOW!
www.BharatAcharyaEducation.com

Order our Books here...

8086 Microprocessor
Link: https://amzn.to/3qHDpJH

8051 Microcontroller
Link: https://amzn.to/3aFQkXc

Official WhatsApp number:
+91 9136428051