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**

5UEAKE

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.

Before		Operation	After	
		SP ← SP – 1		
		[SP] ← B	SP	C
		SP ← SP – 1		B
SP	XX	[SP] ← C		XX

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**

Before		Operation	After	
		SP ← SP – 1		
		[SP] ← PC_H	SP	PC_L
		SP ← SP – 1		PC_H
SP	XX	[SP] ← PC_L		XX
		PC ← 2000		

Eg: **RET**

Before		Operation	After	
		PC_L ← [SP]		
SP	PC_L	SP ← SP + 1		
	PC_H	PC_H ← [SP]		
	XX	SP ← SP + 1	SP	XX

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:

```
MVI C,25        ; parameter value 25
                  moved into C
CALL SUB
ADD D
  .
  .
  .
HLT

SUB:  MOV A,C     ; Parameter value accessed from
                    C-Reg
        .
        .
        .
      MOV D,A
      RET
```

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:

```
MVI A,25
STA 2000      ; parameter value 25 moved
              into location 2000
CALL SUB
ADD D
  .
  .
  .
HLT

SUB:  LDA 2000   ; Parameter value
                 ;accessed from ;location
                 2000.
        .
        .
        .
      MOV D,A
      RET
```

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:

```
MVI M,25      ; parameter value 25 ;
              moved into Memory ;
              pointer "M"
CALL SUB
ADD D
  .
  .
  .
HLT

SUB:  MOV A,M    ; Parameter value          ;
                 accessed from location
                 ; 2000.
        .
        .
        .
      MOV D,A
      RET
```

5UEAKE

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

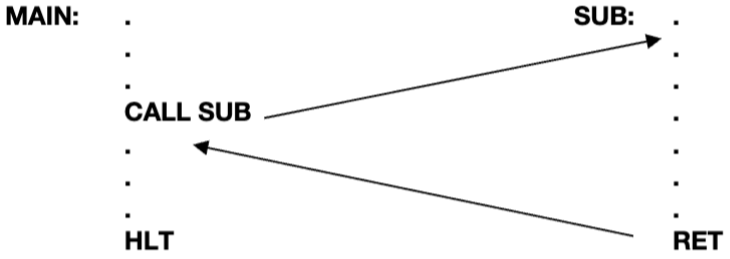
Special Note:

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

TYPES OF SUB-ROUTINES

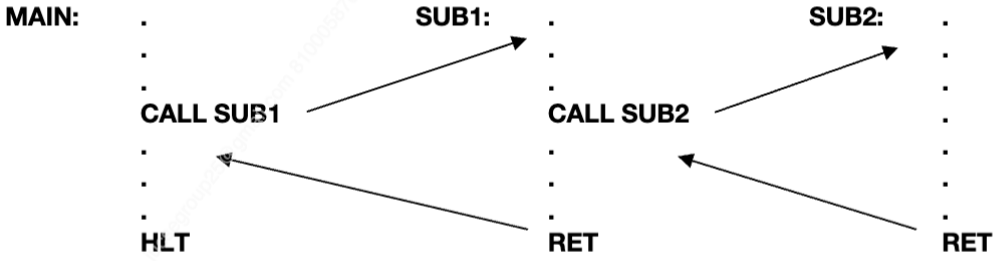
1. Simple Subroutines

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



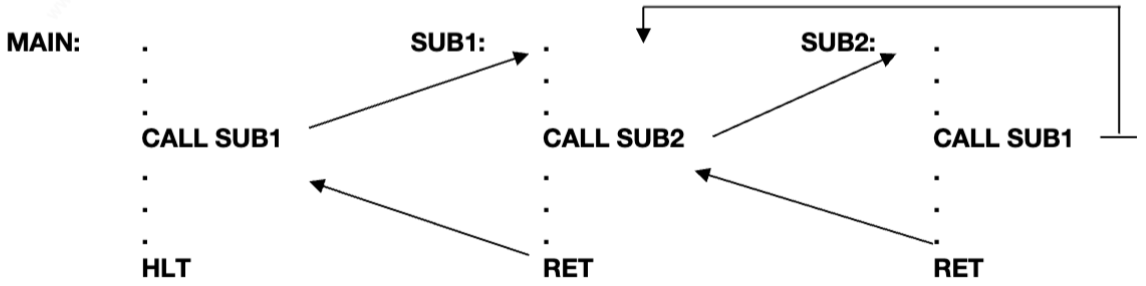
2. Nested Subroutines

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



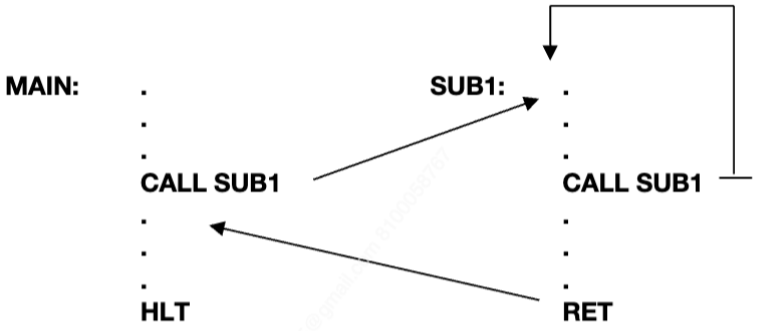
3. Re-entrant Subroutines

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.

5UEAKE

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.

∴ $T_D = 4 \times 0.333 = 1.332$ 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	MT
Loop:	**DCR B**	4T	NT
	JNZ Loop	10/7T	NT
	RET	10T	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}

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

∴ $T_{D\,max} = 3584T$.

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	N T
	ORA C	4T	
	JNZ Loop	10/7T	
	RET	10T	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.

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

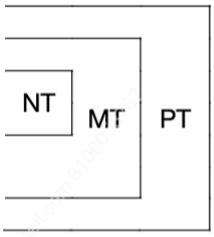
Note

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}

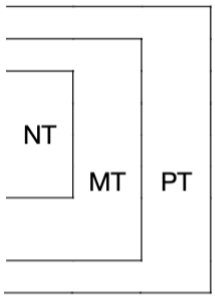
$\therefore T_{D\,max}$ = 914954T.

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

$\therefore$ **$T_{D\,max}$ = 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}

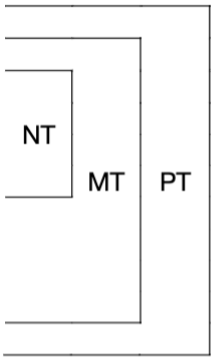
$Count2_{max}$ = 255 {8-bit count in decimal}

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.

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

www.bharatachariyaeducation.com |jaseegroup25@gmail.com 8100959767

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

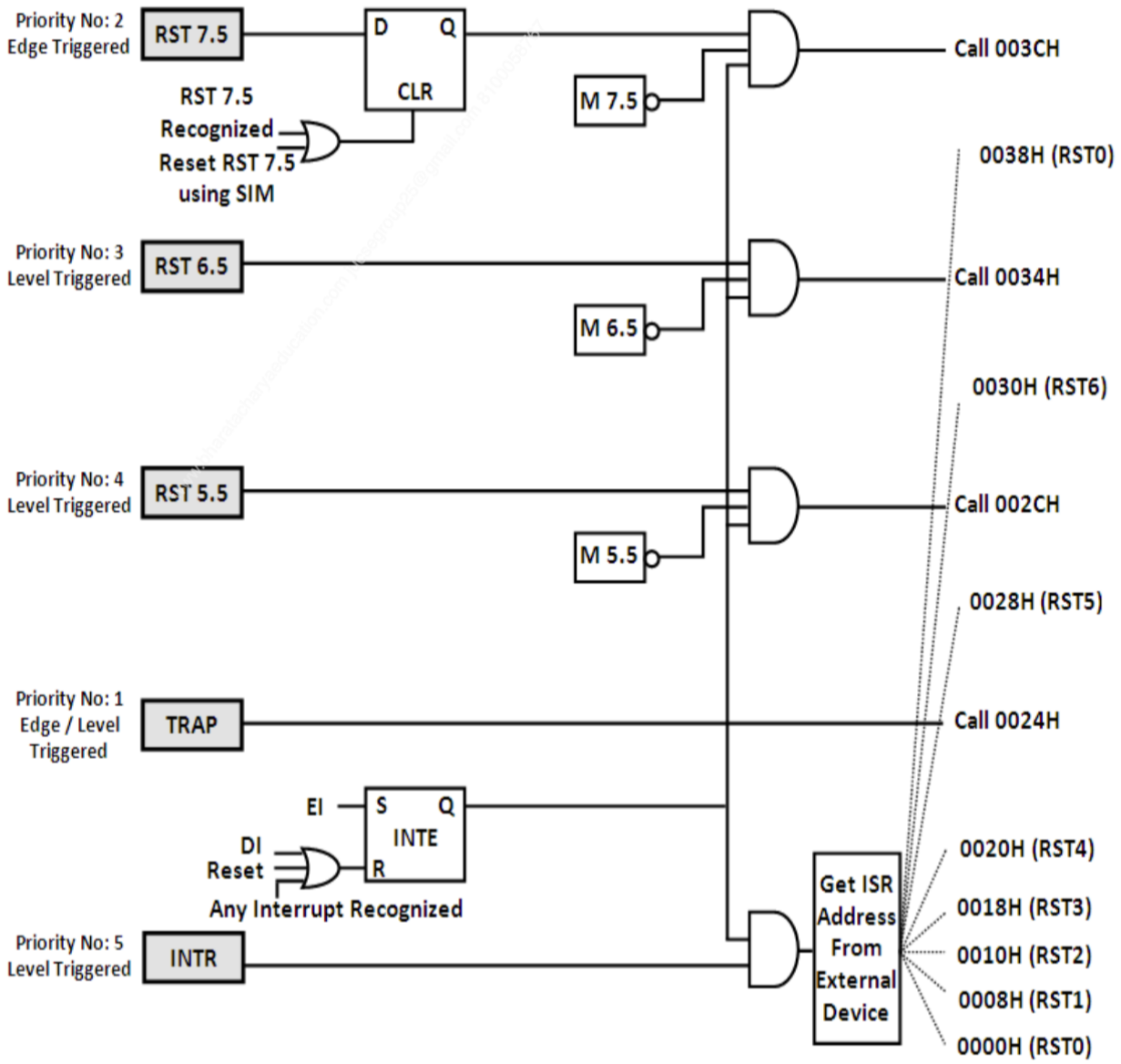
8085 INTERRUPTS

- 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

5UEAKE

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	0000H
RST1	0008H
RST2	0010H
RST3	0018H
RST4	0020H
RST5	0028H
RST6	0030H
RST7	0038H

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...

5UEAKE

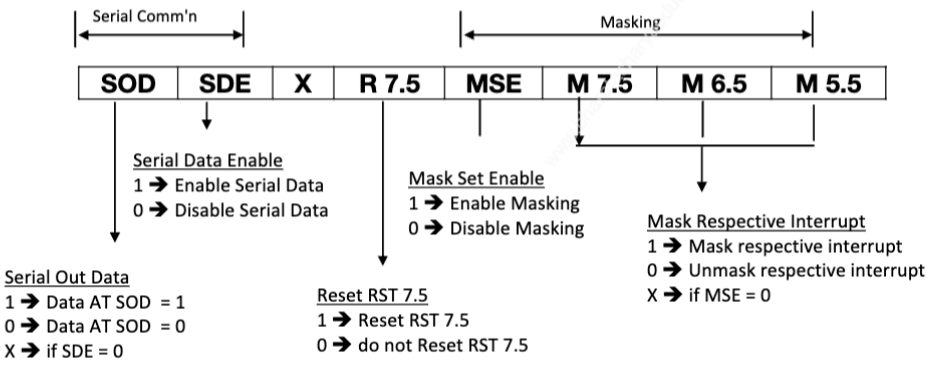
8086 Microprocessor
Link: https://amzn.to/3qHDpJH

SIM & RIM Instructions

SIM: Set Interrupt Mask

SIM is a multipurpose instruction.
It is used for the following…

- To **Mask** or Un-Mask the **RST7.5, RST6.5 and RST5.5** interrupts.
- To send the data out serially (bit - by - bit) through the **SOD** line of the µP.
- To **reset RST7.5** interrupt irrespective of whether it is masked or not.



Method of execution:

- The appropriate byte is formed and **loaded into** the **Accumulator**.
- Then the **SIM** Instruction is **executed**.
- The µP reads the contents of the accumulator in the above order.

Note

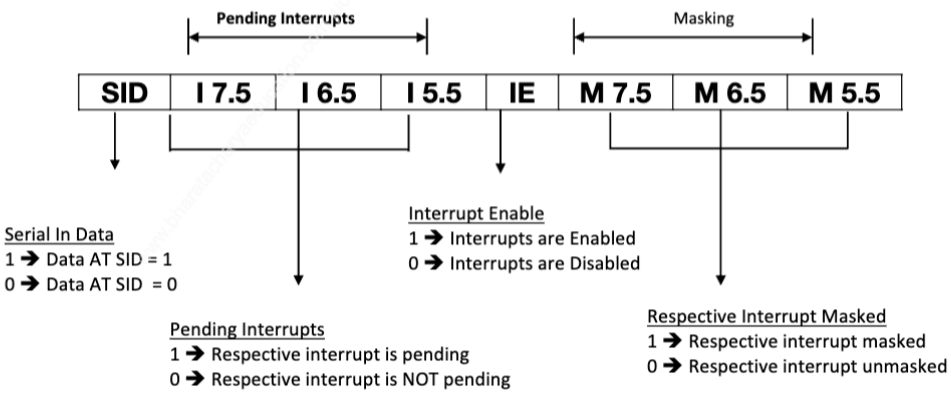
If We have disabled interrupts using DI, then whatever masking we do in SIM is of no use. First, we must enable interrupts using EI instruction. Only Then the masking pattern we give in SIM will come into effect.

5UEAKE

RIM: Read Interrupt Mask

This instruction is used for the following purposes:

- To **read the Interrupt Mask** of the µP.
- To accept data serially through the **SID** pin.
- To see the **"Pending Interrupts"** of the µP.



Pending Interrupts:

Pending interrupts are those interrupts, which are waiting to be serviced.
An interrupt becomes pending as a higher priority interrupt is currently being serviced.
RIM Instruction indicates the Pending Status of RST7.5, RST6.5 and RST5.5.

Method of execution:

- Then the **RIM** Instruction is **executed**.
- The **µP loads** the appropriate byte into the **Accumulator.**
- The programmer reads the contents of the Accumulator.

Special Note:

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

SQUARE WAVE PROGRAM USING SIM

Write a program 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 = 0.5×10^{-3} sec
1T = 0.333 μsec = 0.333×10^{-6} sec

Substituting the above values we get:
$0.5\times10^{-3} = 17\times(0.333\times10^{-6}) + [(Count)_d \times 14\times(0.333\times10^{-6})] - 3\times(0.333\times10^{-6})$
Count = 6AH

Special Note:

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