Microprocessor & Interfaces

(4CS3-04)

UNIT 3

Presented by: Shruti Sharma

Assistant Professor

Department of Computer Science Engineering

AIET, Jaipur

TOPIC:

8085 ASSEMBLY LANGUAGE PROGRAMMING

1) PROGRAM TO ADD OR SUBTRACT TWO 8 BIT NUMBERS

Sl no.	ADDRESS	HEX CODE	LABEL	MNEMONICS	COMMENTS
1	8000	3E, XX		MVI A, XX	Move immediately to [A] the data XX
2	8002	06, YY		MVI B, YY ADD B/SUB B STA F000	Move immediately to [B] the data YY
3	8004	80/(90)			Add(or subtract) [B] with [A]
4	8005	32,00,F0		JNC (Ahead 1)	Store contents of [A] in specified location
5	8008	D2,10,80		MVIA, 01	Jump on no carry to specified location
6	800B	3E,01		JMP(Ahead 2)	Move immediately to [A] the data(01)$_H$
7	800D	C3,12,80		MVIA, 00	Jump unconditionally ahead
8	8010	3E,00		STA F001	Move immediately to [A] the data(00)$_H$
9	8012	32,01,F0		RST1	Store contents of [A] in specified location
10	8015	CF	Ahead 1		Restart
			Ahead 2		

INPUT : XX, YY
1) 2B, 1A
2) 2B, FA

OUTPUT

TRIAL	Addition (XX+YY)	Subtraction (XX-YY)
ADDRESS	F001: F000	F001: F000
1	00 : 45	00 : 25
2	01 : 25	01 : 31

2) PROGRAM TO ADD TWO 16 BIT NUMBERS

Sl no.	ADDRESS	HEX CODE	LABEL	MNEMONICS	COMMENTS
1	8000	AF		XRA A	Clear accumulator and Flag
2	8001	2A, 00, F0		LHLD F000 XCHG	Load memory from specified locations
3	8004	EB		LHLD F100 DAD D	Exchange contents of HL & DE pairs
4	8005	2A,00,F1		SHLD F200	Load memory from specified locations
5	8008	19		JNC (Ahead) INR A	Add HL & DE pairs
6	8009	22,00,F2		STA F202 RST1	
7	800C	D2,10,80			Store memory to specified locations
8	800F	3C			Jump on no carry to ahead address
9	8010	32,02,F2			Increment contents of accumulator by one
10	8013	CF	Ahead		Store contents of [A] in specified location
					Restart

F001: F000 (X)	4A : 2C
F101: F100 (Y)	ED : 5B
F202:F201: F200 (X+Y)	01 : 37 : 87

3) PROGRAM TO SUBTRACT TWO 16 BIT NUMBERS (X>Y)

Sl no.	ADDRESS	HEX CODE	LABEL	MNEMONICS	COMMENTS
1	8000	2A,00,F0		LHLD F000	Load memory from specified locations
2	8003	EB		XCHG	Exchange contents of HL & DE pairs
3	8004	2A,00,F1		LHLD F100	Load memory from specified locations
4	8007	7D		MOV A, L	Move contents of [L] to [A]
5	8008	93		SUB E	Subtract [E] from [A]
6	8009	32,00,F2		STA F200	Store result in specified location
7	800C	7C		MOV A, H	Move contents of [H] to [A]
8	800D	9A		SBB D	Subtract [D] from [A] with borrow
9	800E	32,01,F2		STA F201	Store result in specified location
10	8011	CF		RST 1	Restart

F001: F000 (X)	FA : 2C
F101: F100 (Y)	BD : 5B
F202:F201: F200 (X-Y)	3C : D1

4) PROGRAM TO ADD TWO N BYTE NUMBERS

Sl no.	ADDRESS	HEX CODE	LABEL	MNEMONICS	COMMENTS
1	8000	21,00,F0		LXIH F000	Load HL pair with specified data
2	8003	01,00,F1		LXIB F100 LXID	Load BC pair with specified data Load DE pair
3	8006	11,00,F2		F200 MVI A, (NN)	with specified data Specify the number of bytes
4	8009	3E,(NN)		STA 8100 XRA A	Save the count
5	800B	32,00,81		LDAX B ADC M	Clear accumulator and flags
6	800E	AF 0A	LOOP	STAX D INX H	Load [A] indirectly from address specified by
7	800F	8E		INX B INX D	BC pair
8	8010	12		LDA 8100 DCR	Add memory to [A] with carry
9	8011	23		A	Store contents of [A] indirectly in location
10	8012	03		JNZ (LOOP)	specified by DE pair Update memory
11	8013	13		JNC (Ahead 1)	Update BC register pair Update DE
12	8014	3A,00,81		MVI A, 01	register pair
13	8015	3D C2,0B,80		JMP (Ahead 2)	Load [A] from specified location Decrement
14	8018	D2,24,80 3E,01		MVI A, 00 STAX	count
15	8019	C3,26,80 3E,00		D RST 1	Jump on no zero to perform loop Jump on no
16	801C	12			carry to ahead address
17	801F	CF			Move immediately to [A] the data(01)H
18	8021				Jump unconditionally to ahead address
19	8024				Move immediately to [A] the data(00)H
20	8026				Store contents of [A] indirectly in address specified by DE pair
21	8027				Restart
			Ahead 1		
			Ahead 2		

INPUT(X)	F103 : F102 : F101 : F100	A4 : E6 : F3 : D7
INPUT(Y)	F003 : F002 : F001 : F000	C3 : 54 : A2 : 1B
OUTPUT(X+Y)	F204:F203 : F202 : F201 : F200	01: 68 : 3B: 95: F2

5) PROGRAM FOR BLOCK TRANSFER

Sl no.	ADDRESS	HEX CODE	LABEL	MNEMONICS	COMMENTS
1	8000	16,10		MVI D, (10)H	Set count in [D]
2	8002	21,00,81		LXIH 8100	Load HL pair with specified address
3	8005	01,00,82		LXIB 8200	Load BC pair with specified address
4	8008	7E	LOOP	MOV A,M	Move contents of memory to [A]
5	8009	02		STAX B INX H INX B	Store contents of [A] indirectly to location specified by BC pair
6	800A	23		DCR D	Update memory
7	800B	03		JNZ (LOOP)	Update BC pair Decrement count in [D]
8	800C	15		RST1	Jump on no zero to perform the loop
9	800D	C2,08,80			Restart
10	8010	CF			

INPUT

8100:810:8102:8103:8104:8105:8106:8107:8108:8109:810A:810B:810C:810D:810E:810F
00 : 01: 02 : 03 : 04 : 05 : 06 : 07 : 08 : 09 : 0A : 0B : 0C : 0D : 0E : 0F

OUTPUT

8200:8201:8202:8203:8204:8205:8206:8207:8208:8209:820A:820B:820C:820D:820E:820F
00 : 01: 02 : 03 : 04 : 05 : 06 : 07 : 08 : 09 : 0A : 0B : 0C : 0D : 0E : 0F

6) PROGRAM TO BLOCK TRANSFER IN REVERSE ORDER

Sl no.	ADDRESS	HEX CODE	LABEL	MNEMONICS	COMMENTS
1	8000	16,10		MVI D, $(10)_H$	Set count in [D]
2	8002	21,00,81		LXIH 8100	Load HL pair with specified address
3	8005	01,0F,82		LXIB 820F	Load BC pair with specified address
4	8008	7E	LOOP	MOV A,M	Move contents of memory to [A]
5	8009	02		STAX B	Store contents of [A] indirectly to location specified by BC pair
6	800A	23		INX H	Update memory
7	800B	0B		DCX B	Decrement BC pair by one
8	800C	15		DCR D	Decrement count in [D]
9	800D	C2,08,80		JNZ (LOOP)	Jump on no zero to perform the loop
10	8010	CF		RST1	Restart

INPUT

8100:810:8102:8103:8104:8105:8106:8107:8108:8109:810A:810B:810C:810D:810E:810F
00 : 01: 02 : 03 : 04 : 05 : 06 : 07 : 08 : 09 : 0A : 0B : 0C : 0D : 0E : 0F

OUTPUT

820F:820E:820D:820C:820B:820A:8209:8208:8207:8206:8205:8204:8203:8202:8201:8200
0F : 0E : 0D : 0C : 0B : 0A : 09 : 08 : 07 : 06 : 05 : 04 : 03 : 02 : 01 : 00

7) PROGRAM TO FIND 2`S COMPLEMENT OF 8 BIT/ 16 BIT NUMBER

Sl no.	ADDRESS	HEX CODE	LABEL	MNEMONICS	COMMENTS
1	8000	06,01/(02)		MVI B,01/(02)	Move to [B] (01) for 8-bit & (02) for 16-bit complement of data Load DE pair with specified data
2	8002	11,01,00		LXID 0001	
3	8005	21,00,F0		LXIH F000	Load HL pair with specified data Copy
4	8008	7E		MOV A,M	
5	8009	2F	LOOP	CMA	contents of memory to [A] Complement contents of [A]
6	800A	77		MOV M,A	Copy contents of [A] to memory
7	800B	05		DCR B	Decrement [B] by one
8	800C	CA, 13,80		JZ (Ahead)	Jump on zero to ahead address Update memory
9	800F	23		INX H	
10	8010	C3,08,80		JMP(Loop)	Jump unconditionally to perform loop
11	8013	2A,00,F0		LHLD F000	Load contents of HL pair from specified locations
12	8016	19		DAD D SHLD	Add HL and DE pair of registers
13	8017	22,00,F1 CF	Ahead	F100 RST 1	Store contents of HL pair to specified locations
14	801A				Restart

DATA	INPUT	OUTPUT(2`s complement)
8-Bit	F000 : 7A	F100 : 87
16-Bit	F001 : F000 :: 1A: 26	F101 : F100 :: E5: DA

TOPIC:

TIMER, COUNTER & TIME DELAY

Time Delay

- In real time applications , such as traffic light control, digital clock, process control, it is important to keep a track with time.
- In some applications the time delay is required between execution of two instructions.

Time Delay Techniques

There are two techniques to design the time delay.

1. Hardware Technique
2. Software Technique

Software Technique

There are the following methods are used to provide the time delay using software technique.

1. Using NOP Instruction
2. Using Counter
3. Using Nested Loop

NOP Instruction

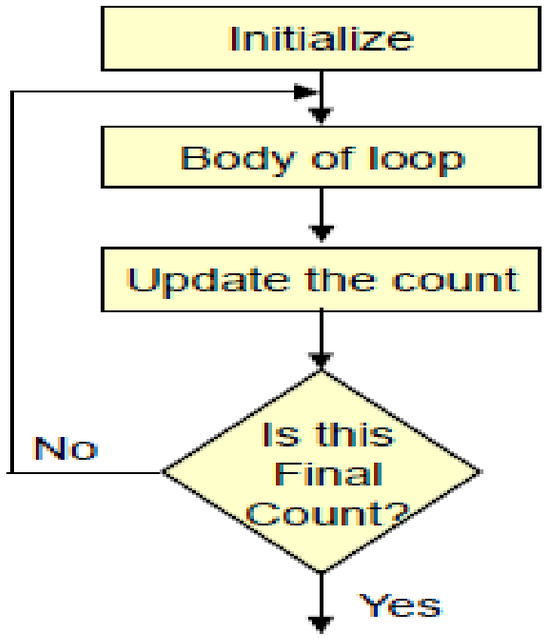
- One of the main usage of NOP instruction is in delay generation.
- The NOP instruction is taking four clock pulses to be fetching, decoding and executing.
- If the 8085 MPU is working on 6MHz clock frequency, then the internal clock frequency is 3MHz. So from that we can easily determine that each clock period is 1/3 of a microsecond. So the NOP will be executed in 1/3 * 4 = 1.333µs.

Counters

- A loop counter is set up by loading a register with a certain value
- Then using the DCR (to decrement) and INR (to increment) the contents of the register are updated.
- A loop is set up with a conditional jump instruction that loops back or depending on whether the count has reached the termination count.

Counters

- The operation of a loop counter can be described using the following flowchart.



Sample ALP for implementing a loop Using DCR instruction

```
        MVI C, 15H
LOOP    DCR C
        JNZ LOOP
```

Using a Register Pair as a Loop Counter

- Using a single register, one can repeat a loop for a maximum count of 255 times.
- It is possible to increase this count by using a register pair for the loop counter instead of the single register. A minor problem arises in how to test for the final count since DCX and INX do not modify the flags.
- However, if the loop is looking for when the count becomes zero, we can use a small trick by Oring the two registers in the pair and then checking the zero flag.

Using a Register Pair as a Loop Counter

The following is an example of a loop set up with a register pair as the loop counter.

```
        LXI B, 1000H
LOOP    DCX B
        MOV A, C
        ORA B
        JNZ LOOP
```

Delays

- Knowing how many T-States an instruction requires, and keeping in mind that a T-State is one clock cycle long, we can calculate the time using the following formula:
- Delay = No. of T-States / Frequency
- For example a “MVI” instruction uses 7 T-States. Therefore, if the Microprocessor is running at 2 MHz, the instruction would require 3.5 µSeconds to complete.

Delay loops

- We can use a loop to produce a certain amount of time delay in a program.
- The following is an example of a delay loop:

```
        MVI C, FFH      7 T-States
LOOP    DCR C           4 T-States
        JNZ LOOP        10 T-States
```

- The first instruction initializes the loop counter and is executed only once requiring only 7 T-States.
- The following two instructions form a loop that requires 14 T-States to execute and is repeated 255 times until C becomes 0.

Delay Loops

- We need to keep in mind though that in the last iteration of the loop, the JNZ instruction will fail and require only 7 T-States rather than the 10.
- Therefore, we must deduct 3 T-States from the total delay to get an accurate delay calculation.
- To calculate the delay, we use the following formula:

$$T_{delay} = T_O + T_L$$

T_{delay} = total delay

T_O = delay outside the loop

T_L = delay of the loop

- TO is the sum of all delays outside the loop.

Delay Loops

- Using these formulas, we can calculate the time delay for the previous example:
- TO= 7 T-States

  Delay of the MVI instruction

- TL= (14 X 255) -3 = 3567 T-States

  14 T-States for the 2 instructions repeated 255 times (FF16= 25510) reduced by the 3 T-States for the final JNZ.

Using a Register Pair as a Loop Counter

- Using a single register, one can repeat a loop for a maximum count of 255 times.
- It is possible to increase this count by using a register pair for the loop counter instead of the single register.
  - A minor problem arises in how to test for the final count since DCX and INX do not modify the flags.
  - However, if the loop is looking for when the count becomes zero, we can use a small trick by Oring the two registers in the pair and then checking the zero flag.

Using a Register Pair as a Loop Counter

- The following is an example of a delay loop set up with a register pair as the loop counter.

	LXI B, 1000H	10 T-States
LOOP	DCX B	6 T-States
	MOV A, C	4 T-States
	ORA B	4 T-States
	JNZ LOOP	10 T-States

Using a Register Pair as a Loop Counter

- Using the same formula from before, we can calculate:
- TO= 10 T-States

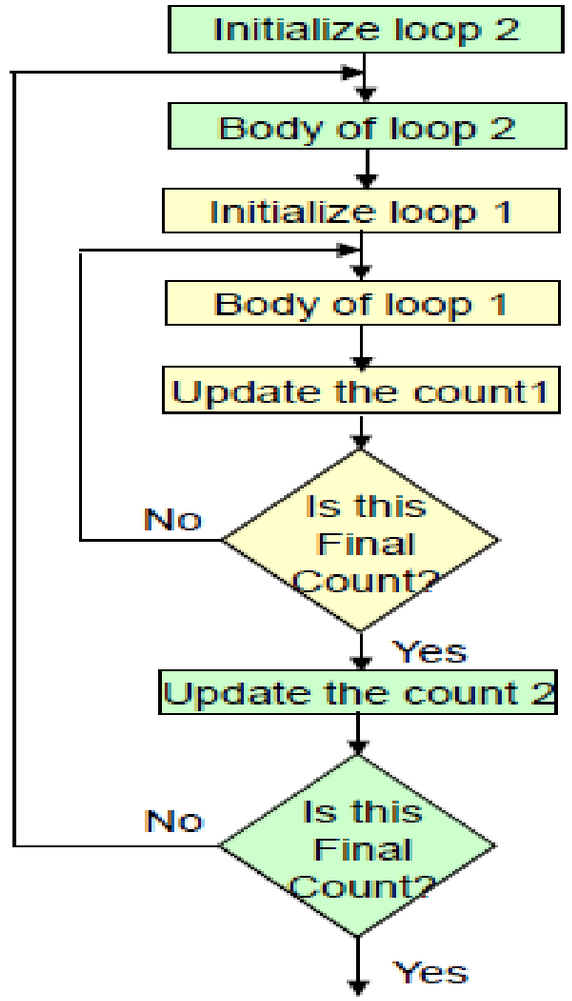
  The delay for the LXI instruction

- TL= (24 X 4096) -3 = 98301 T-States

  24 T-States for the 4 instructions in the loop repeated 4096 times (100016= 409610) reduced by the 3 T-States for the JNZ in the last iteration.

Nested Loops

- Nested loops can be easily setup in Assembly language by using two registers for the two loop counters and updating the right register in the right loop.
- In the figure, the body of loop2 can be before or after loop1.



Nested Loops for Delay

- Instead (or in conjunction with) Register Pairs, a nested loop structure can be used to increase the total delay produced.

```
        MVI B, 10H      7 T-States
LOOP2   MVI C, FFH      7 T-States
LOOP1   DCR C           4 T-States
        JNZ LOOP1       10 T-States
        DCR B           4 T-States
        JNZ LOOP2       10 T-States
```

Delay Calculation of Nested Loops

- The calculation remains the same except that it the formula must be applied recursively to each loop.
  - Start with the inner loop, then plug that delay in the calculation of the outer loop.
- Delay of inner loop
  - T_{O1}= 7 T-States
    - MVI C, FFH instruction
  - T_{L1}= (255 X 14) -3 = 3567 T-States
    - 14 T-States for the DCR C and JNZ instructions repeated 255 times (FF_{16}= 255_{10}) minus 3 for the final JNZ

Delay Calculation of Nested Loops

- Delay of outer loop
  - T_{O2}= 7 T-States
    - MVI B, 10H instruction
  - T_{L1}= (16 X (14 + 3574)) -3 = 57405 T-States1
    - 4 T-States for the DCR B and JNZ instructions and 3574 T-States for loop1 repeated 16 times (10_{16}= 16_{10}) minus 3 for the final JNZ.
  - T_{Delay}= 7 + 57405 = 57412 T-States
- Total Delay
  - T_{Delay}= 57412 X 0.5 µSec = 28.706 mSec

Increasing the delay

- The delay can be further increased by using register pairs for each of the loop counters in the nested loops setup.
- It can also be increased by adding dummy instructions (like NOP) in the body of the loop.

TOPIC :

8259 PROGRAMMABLE INTERRUPT CONTROLLER

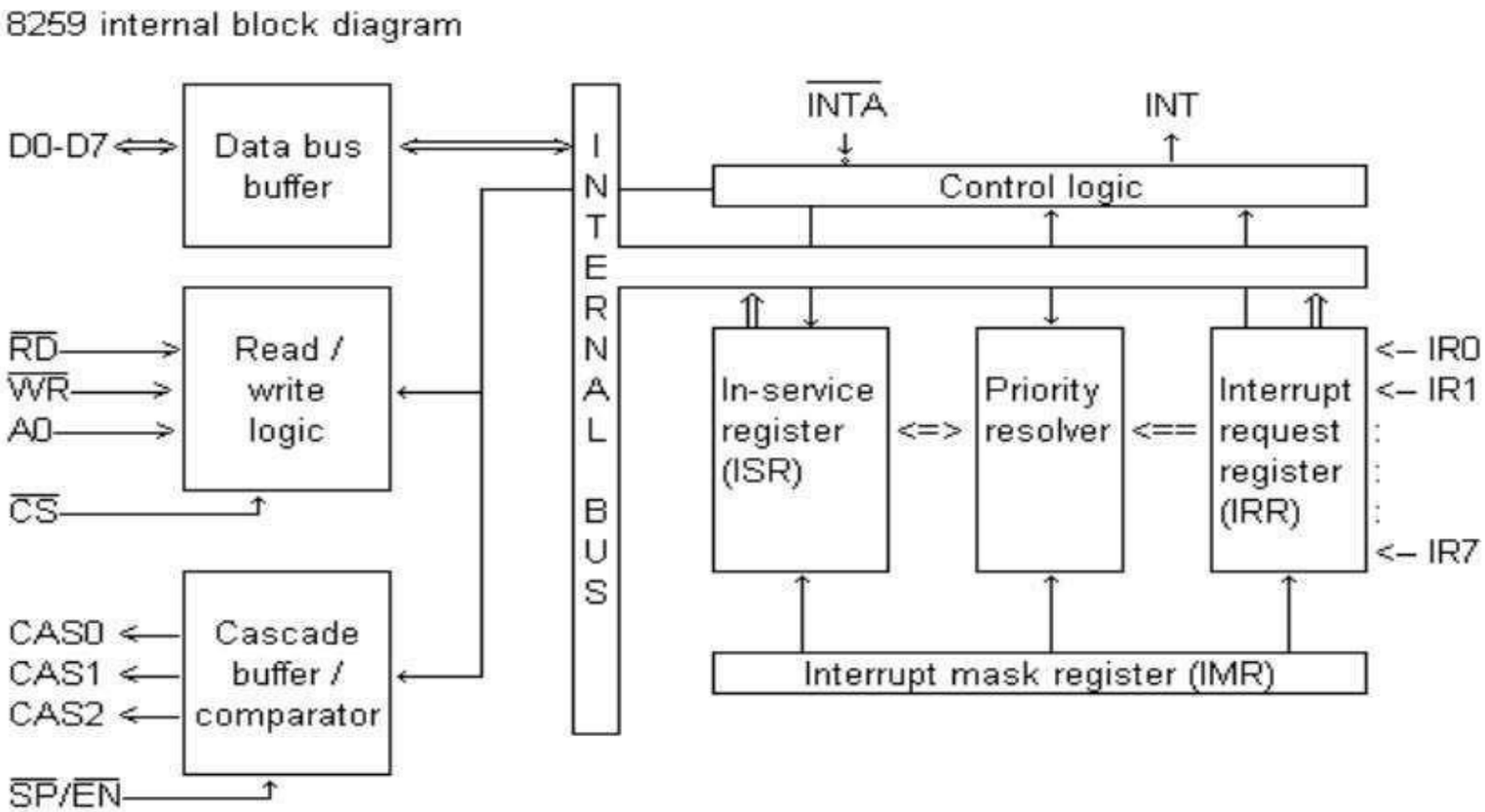
8259 PIC

- The 8259A is a programmable interrupt controller(PIC) specially designed to work with Intel microprocessor 8080, 8085A, 8086, 8088.
- It is a tool for managing the interrupt requests and functions as an overall manager in an Interrupt-Driven system environment.
- It accepts requests from the peripheral equipment, determines which of the incoming requests is of the highest priority, ascertains whether the incoming request has a higher priority value than the level currently being serviced, and issues an interrupt to the CPU based on this determination.
- The PIC, after issuing an Interrupt to the CPU, must somehow input information into the CPU that can ``point'' the Program Counter to the service routine associated with the requesting device.

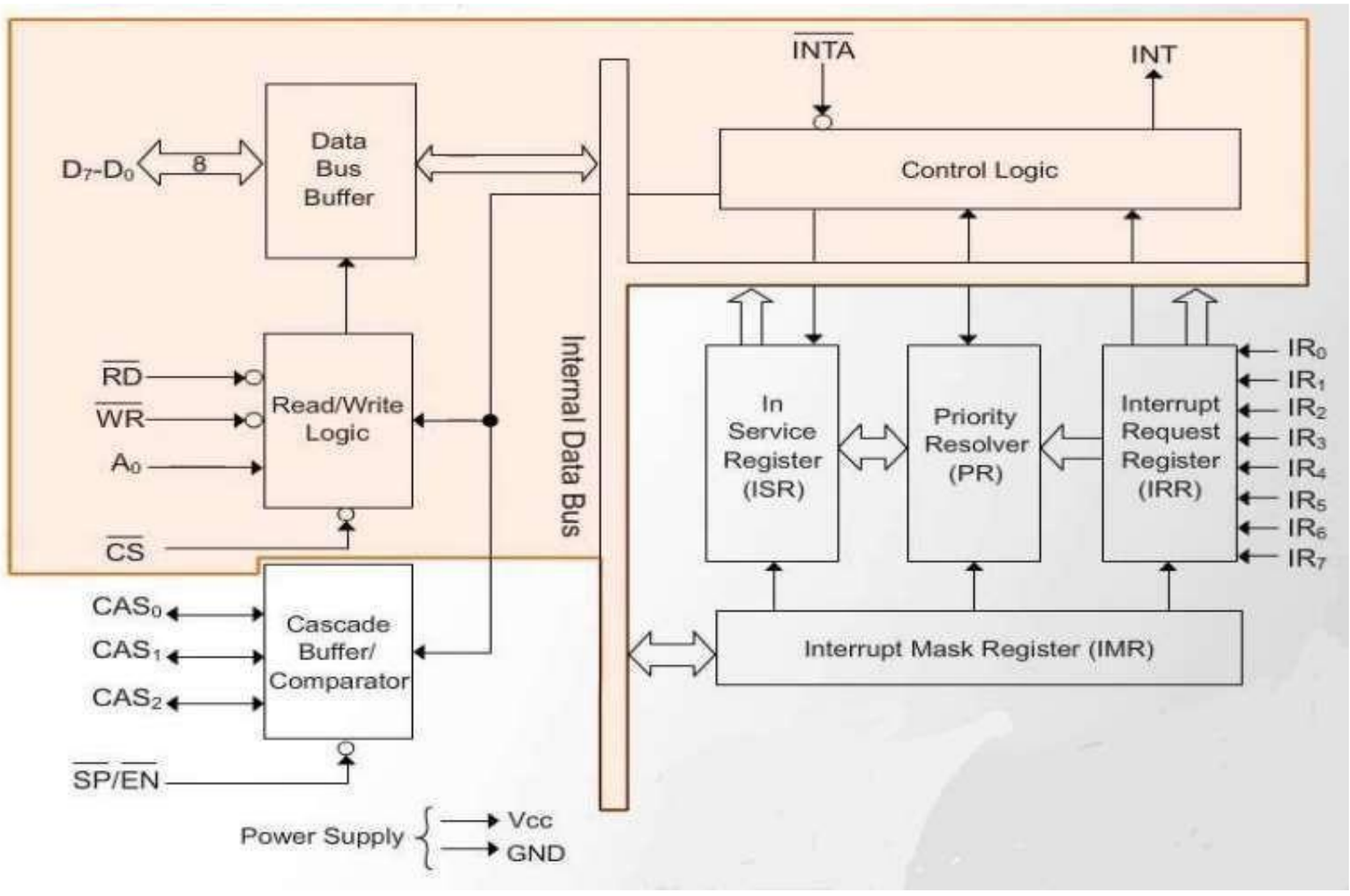
8259 PIC

- 8259 PIC can handle up to 8 vectored priorities interrupts for the processor and it can be cascaded in master-slave configuration to handle 64 levels of interrupts without any additional circuitry.
- It can be programmed either in level triggered or in edge triggered interrupt level and also we can masked individual bits of interrupt request register.
- It has Internal priority resolver which can be programmed into fixed priority mode and rotating priority mode.

Block Diagram of 8259



These Signals are used to interface 8259 with microprocessor



Data bus buffer:

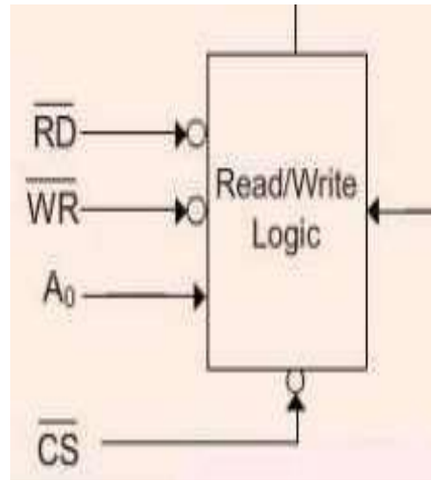
- This, bidirectional 8-bit buffer is used to interface the 8259A to the system data bus.
- Control words and status information from the microprocessor to PIC and from PIC to microprocessor respectively, are transferred through the data bus buffer.
- The processor sends control word to data bus buffer through D0-D7.
- The processor read status word from data bus buffer through D0-D7
- Also, after selection of Interrupt by 8259 microprocessor, it transfer the opcode of the selected Interrupt and address of the Interrupt service sub routine to the other connected microprocessor.

Read/Write & Control Logic

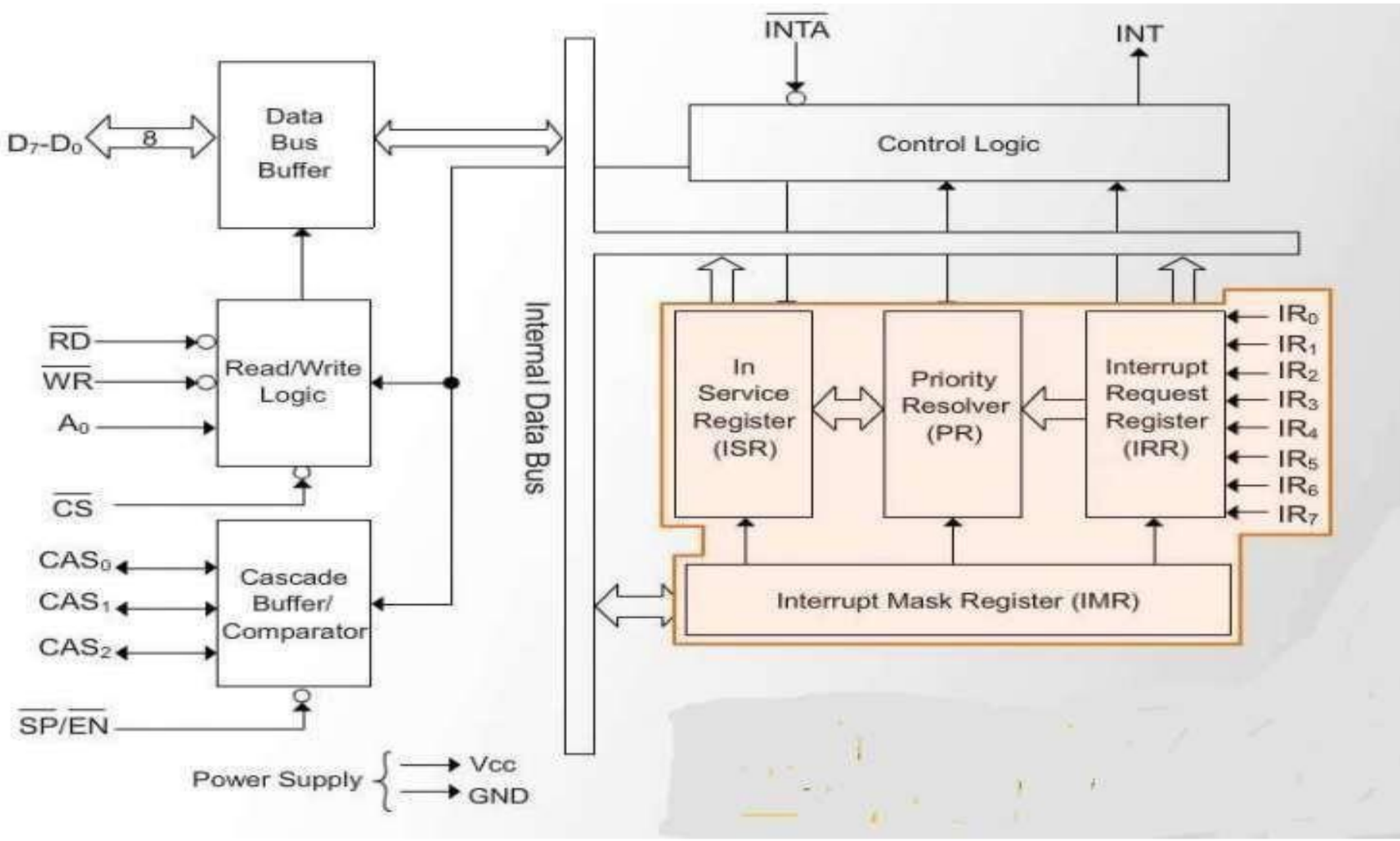
- The function of this block is to accept output commands sent from the CPU.
- It contains **the initialization command word (ICW)** registers and **operation command word (OCW)** registers which store the various control formats for device operation.
- ICW and OCW are two types of command words of 8259, where ICW is used to initialize 8259 where as OCW are issued to control the operation of 8259.
- This function block also allows the status of 8259A to be transferred to the data bus.
- **Control Logic**
  - The function of control logic is to control all the internal operations, generates an INT signal to 8085 and receives INTA signal from 8085.

Pins Connected with Read/Write & Control Logic

- RD, WR, CS and A0 are the control signals used in Read/Write and Control Logic.
- The processor uses the RD (low), WR (low) and A0 to read or write 8259.
- A LOW on WR input enables the CPU to write control words (ICWs and OCWs) to the 8259A.
- A LOW on RD input enables the 8259A to send the status of the Interrupt Request Register (IRR), In Service Register (ISR), the Interrupt Mask Register (IMR), or the Interrupt level onto the Data Bus.
- A0 input signal is used in conjunction with WR and RD signals to write commands into the various command registers, as well as reading the various status registers of the chip. This line can be tied directly to one of the address lines

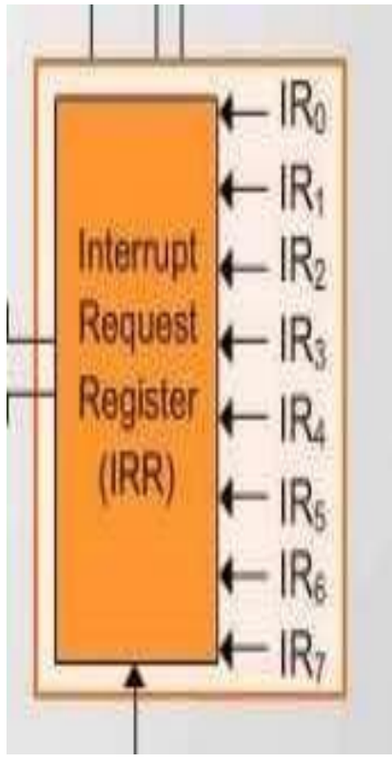


This Sub System of 8259 handles all the external Interrupt requests



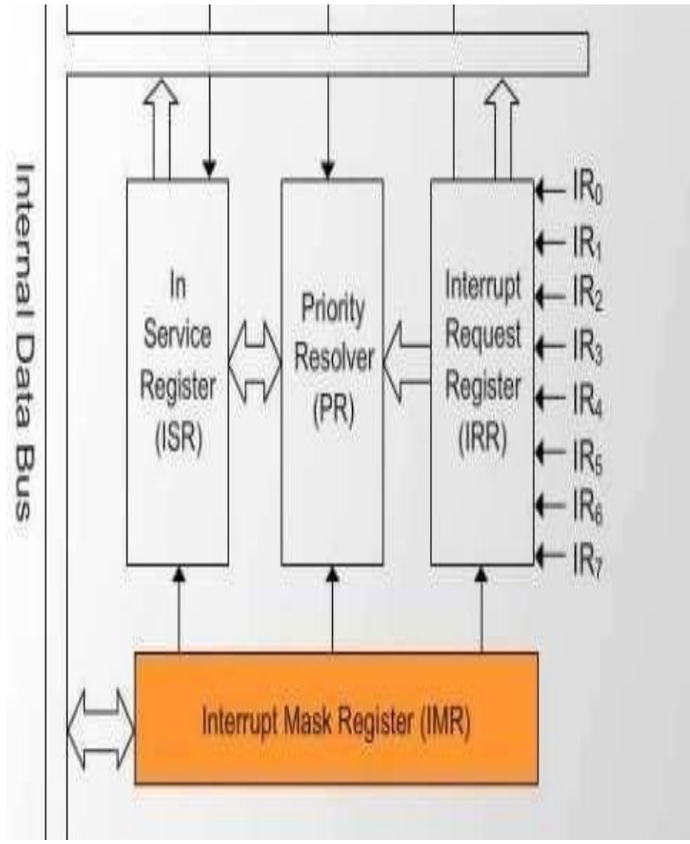
Interrupt Request Register (IRR):

- This block accepts and stores the actual interrupt.
  requests from external interrupting devices on IR_0 – IR_7 lines.
- Interrupt request register (IRR) stores all the incoming interrupt inputs that are requesting service.
- It is an 8-bit register – one bit for each interrupt request.
- Basically, it keeps track of which interrupt inputs are asking for service and also store the pending interrupt requests.



Interrupt Mask Register (IMR):

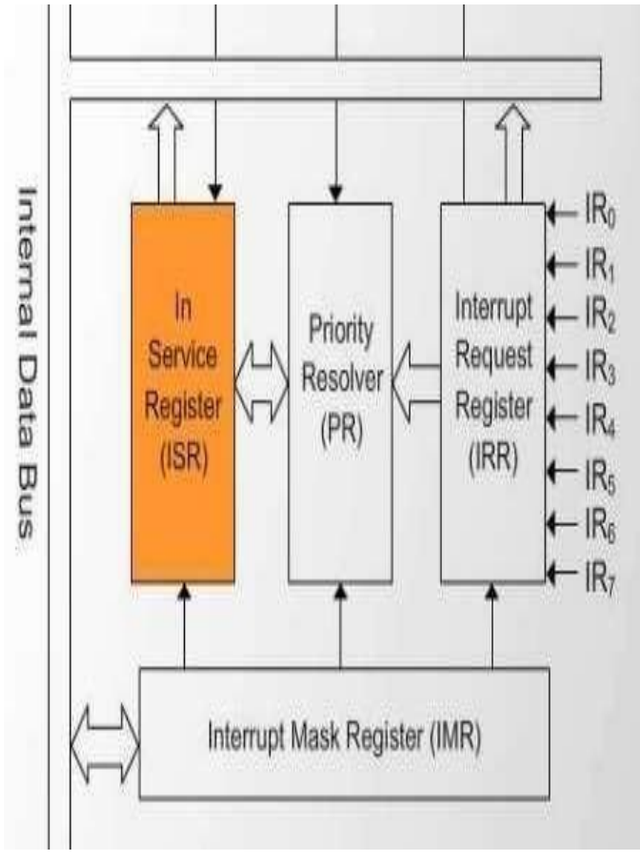
- This logic block masks interrupt lines based on programming by the processor and prevents masked interrupt lines from interrupting the processor.
- The IMR is used to disable (Mask) or enable (Unmask) individual interrupt request inputs.



- The IMR operates on the IRR. Masking of higher priority input will not affect the interrupt request lines of lower priority.
- To unmask any interrupt the corresponding bit is set '0'

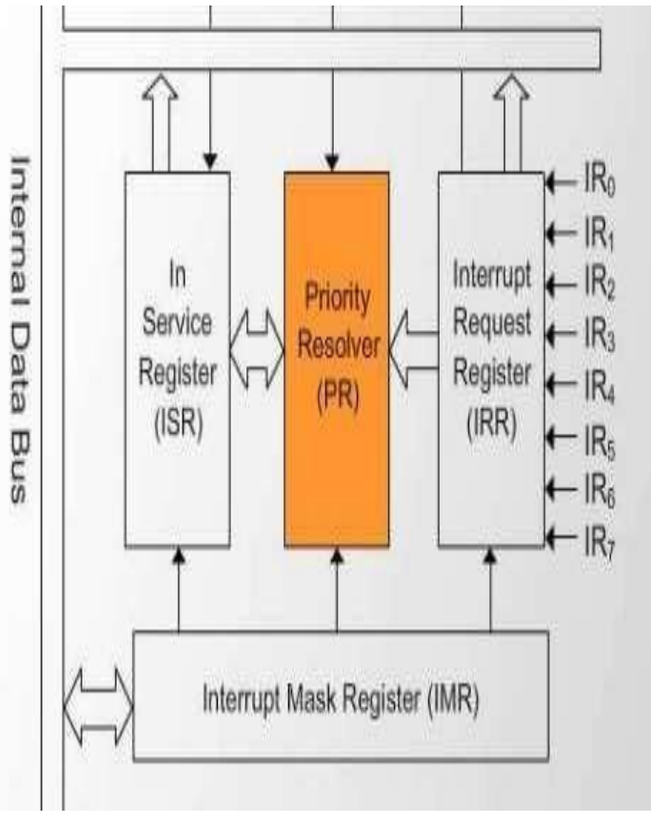
In-service Register (ISR):

- The in-service register keeps track of which interrupt inputs are currently being serviced.
- For each input that is currently being serviced the corresponding bit of in-service register (ISR) will be set.
- In 8259A, during the service of an interrupt request, if another higher priority interrupt becomes active, it will be acknowledged and the control will be transferred from lower priority interrupt service subroutine (ISS) to higher priority ISS.
- Thus, more than one bit of ISR will be set indicating the number of interrupts being serviced.
- Each of these 3-registers can be read as status register.



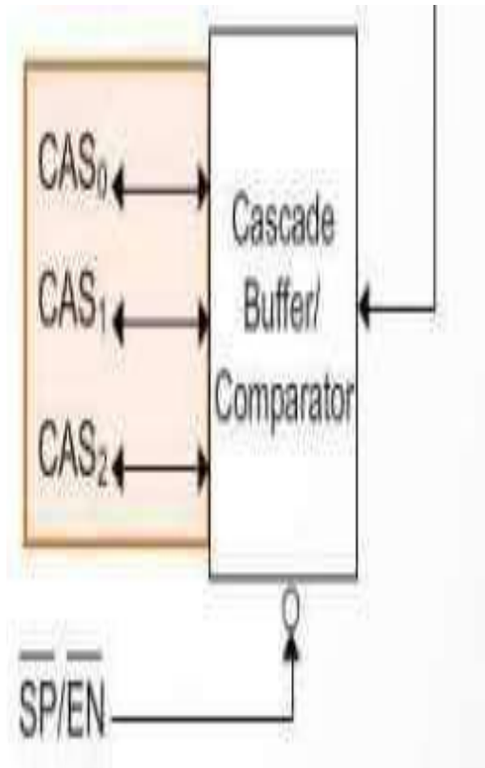
Priority Resolver:

- This logic block determines the priorities of the incoming interrupts set in the IRR.
- It takes the information from IRR, IMR and ISR to determine whether the new interrupt request is having highest priority or not.
- If the new interrupt request is having the highest priority, it is selected and processed.
- The corresponding bit of ISR will be set during interrupt acknowledge machine cycle



Cascade Buffer/Comparator:

- This logic allows for cascading multiple 8259 controllers in a Master/ Slave configuration.
- This function block stores and compares the IDs of all 8259A's in the system.
- The associated 3-I/O lines (CAS2-CAS0) are outputs when 8259A is used as a master and are inputs when 8259A is used as a slave.
- As a master, the 8259A sends the ID of the interrupting slave device onto the CAS2-0 lines.
- The slave 8259As compare this ID with their own programmed ID.



How does the Master-Slave concept work?

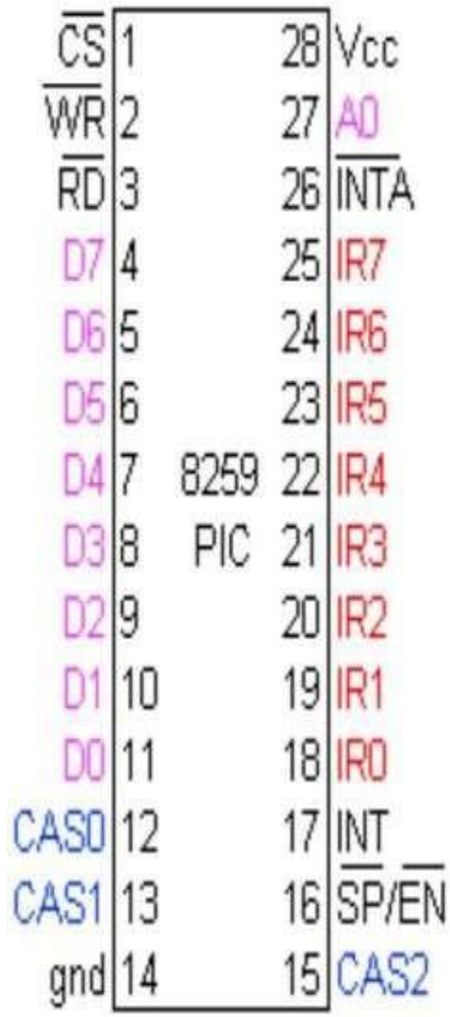
- The slave 8259 creates an interrupt trigger on master as soon as it receives an interrupt trigger at one of its eight input lines.
- The master in turn interrupts the processor.
- The processor read Interrupts Service Routine address.
- The master 8259 informs the corresponding slave 8259 to release the ISR address onto the Data Bus.
- The processor reads this information and executes the appropriate Interrupt service Routine.

INTERCONNECTING OF MASTER /SLAVE PICs AND CPU

- Each PIC scheme provides to receive only up to 8 IR signals. If required more than 8 IR signals then used multiple PIC schemes from which one is master and others are slave. At this case PIC schemes are used in cascading mode.
- In cascading mode INT outs of Slave are connected into nonuse IR line of Master.
- INTR input of CPU can be receives common interrupt request signal only from INT output of single Master.
- Number of selected interrupt vector can be transferred from only Master PIC

PIN Diagram of 8259

1. **D7- D0** is connected to microprocessor data bus D7-D0 (AD7-AD0).
2. **IR7- IR0,** Interrupt Request inputs are used to request an interrupt and to connect to a slave in a system with multiple 8259As
3. **WR** - the write input connects to write strobe signal of microprocessor.
4. **RD** - the read input connects to the IORC signal
5. **INT** - the interrupt output connects to the INTR pin on the microprocessor from the master, and is connected to a master IR pin on a slave.
6. **INTA** - the interrupt acknowledge is an input that connects to the INTA signal on the system. In a system with a master and slaves, only the master INTA signal is connected.
7. **A0** - this address input selects different command words within the 8259A.
8. **CS** - chip select enables the 8259A for programming and control.
9. **SP/EN** - Slave Program/Enable Buffer is a dual-function pin.
   - When the 8259A is in buffered mode, this pin is an output that controls the data bus transceivers in a large microprocessor-based system.
   - When the 8259A is not in buffered mode, this pin programs the device as a master (1) or a slave (0).
10. **CAS2-CAS0**, the cascade lines are used as outputs from the master to the slaves for cascading multiple 8259As in a system.



8259A Interrupt Operation

- To implement interrupt, the interrupt enable flip-flop in the microprocessor should be enabled by writing the EI instruction and the 8259A should be initialized by writing control words in the control register. The 8259A requires two types of control words:
  a. Initialization Command Words (ICWS)
  b. Operational Command Words (OCWs)
- ICWs are used to set up the proper conditions and specify RST vector address. The OCWs are used to perform functions such as masking interrupts, setting up status-read operations etc.
- **Step-1:** The IRR of 8259A stores the request.
- **Step-2:** The priority resolver checks 3 registers-
  - The **IRR** for interrupt requests.
  - **IMR** for masking bits and
  - The **ISR** for interrupt request being served
- It resolves the priority and sets the INT high when appropriate.

8259A Interrupt Operation

- **Step-3:** The MPU acknowledges the interrupt by sending signals in INTA
- **Step-4**: After the $INTA$ is received, the appropriate bit in the ISR is set to indicate which interrupt level is being served and the corresponding bit in the IRR is reset to indicate that the request for the CALL instruction is placed on the data bus.
- **Step-5:** When MPU decodes the CALL instruction, it places two more $INTA$ signals on the data bus.
- **Step-6:** When the 8259A receives the second $INTA$, it places the low-order byte of the CALL address on the data bus. At the 3rd $INTA$, it places the high order byte on the data bus. The CALL address is the vector memory location for the interrupt, this address is placed in the control register during the initialization.
- **Step-7:** During the 3rd $INTA$ pulse, the ISR bit is reset either automatically (Automatic-End-of Interrupt-AEOI) or by a command word that must be issued at the end of the service routine (End of Interrupt-EOI).This option is determined by the initialization command word (ICW).
- **Step-8:** The program sequence is transferred to the memory location specified by the CALL instruction

TOPIC :

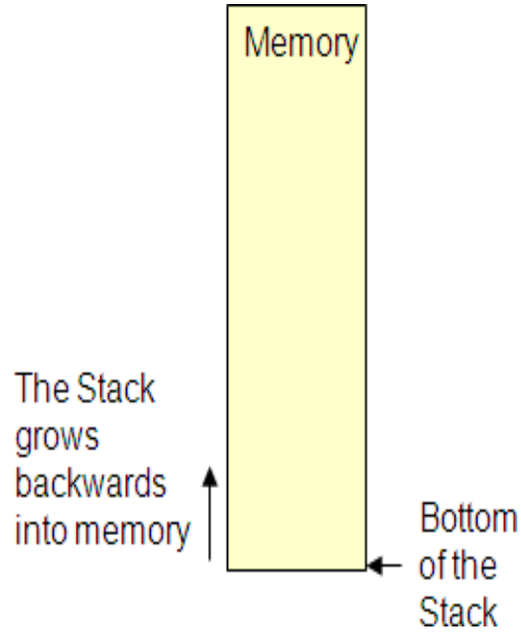
STACK, MACROS & SUBROUTINES

Stack

- The stack is an area of memory identified by the programmer for **temporary storage** of information.

- The stack is a **LIFO (Last In First Out.)** structure.

- The stack normally **grows backwards** into memory.
- In other words, the programmer defines the bottom of the stack and the stack grows up into reducing address range.



Stack

- Given that the stack grows backwards into memory, it is customary to place the bottom of the stack at the **end of memory** to keep it as far away from user programs as possible.

- In the 8085, the stack is defined by setting the SP (Stack Pointer) register.

LXI SP, FFFFH

- This sets the Stack Pointer to location FFFFH (end of memory for the 8085).

Saving Information on STACK

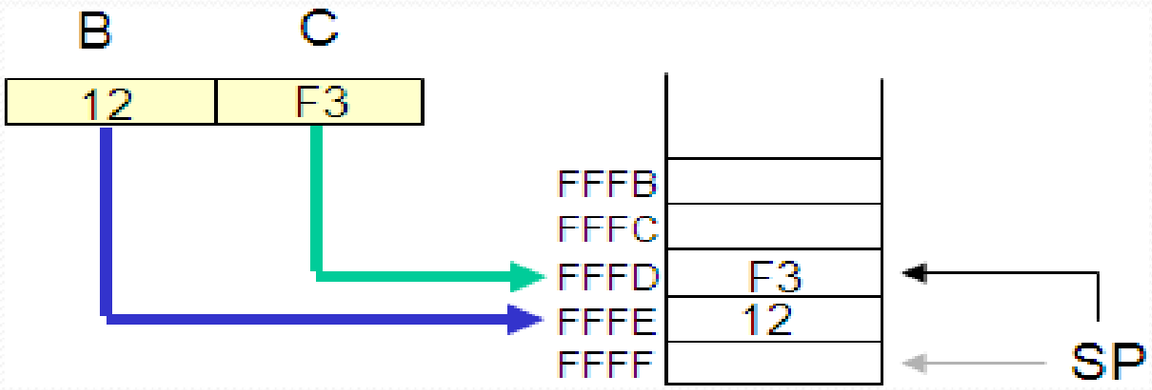
- Information is saved on the stack by PUSHing it on.
- It is retrieved from the stack by POPing it off.

- The 8085 provides two instructions: PUSH and POP for storing information on the stack and retrieving it back.

- Both PUSH and POP work with register pairs ONLY.

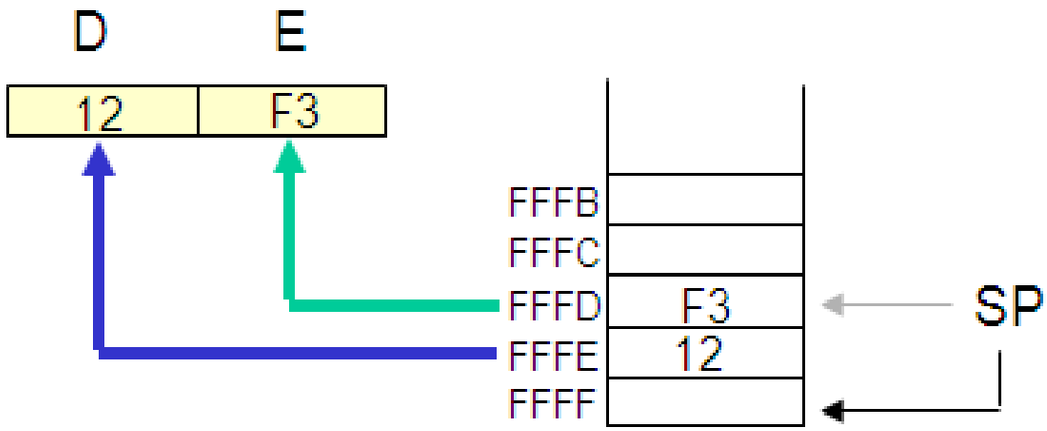
The PUSH Instruction

- PUSH B/D/H/PSW
  - Decrement SP
  - Copy the contents of register B to the memory location pointed to by SP
  - Decrement SP
  - Copy the contents of register C to the memory location pointed to by SP



The POP Instruction

- POP B/D/H/PSW
  - Copy the contents of the memory location pointed to by the SP to register E
  - Increment SP
  - Copy the contents of the memory location pointed to by the SP to register D
  - Increment SP



Operation of the Stack

- During pushing, the stack operates in a “decrement then store” style.
- The stack pointer is decremented first, then the information is placed on the stack.

- During poping, the stack operates in a “use then increment” style.
- The information is retrieved from the top of the the stack and then the pointer is incremented.

- The SP pointer always points to “the top of the stack”.

LIFO

- The order of PUSHs and POPs must be opposite of each other in order to retrieve information back into its original location.

```
PUSH B
PUSH D
...
POP D
POP B
```

- **Reversing the order of the POP instructions will result in the exchange of the contents of BC and DE.**

The PSW Register Pair

- The 8085 recognizes one additional register pair called the PSW (Program Status Word).

- This register pair is made up of the Accumulator and the Flags registers.

- It is possible to push the PSW onto the stack, do whatever operations are needed, then POP it off of the stack.

- The result is that the contents of the Accumulator and the status of the Flags are returned to what they were before the operations were executed.

Cautions with PUSH and POP

- PUSH and POP should be used in opposite order.
- There has to be as many POP's as there are PUSH's.
- If not, the RET statement will pick up the wrong information from the top of the stack and the program will fail.
- It is not advisable to place PUSH or POP inside a loop.

Subroutines

- A subroutine is a group of instructions that will be used repeatedly in different locations of the program.

- Rather than repeat the same instructions several times, they can be grouped into a subroutine that is called from the different locations.

- In Assembly language, a subroutine can exist anywhere in the code.
- However, it is customary to place subroutines separately from the main program.

Subroutines

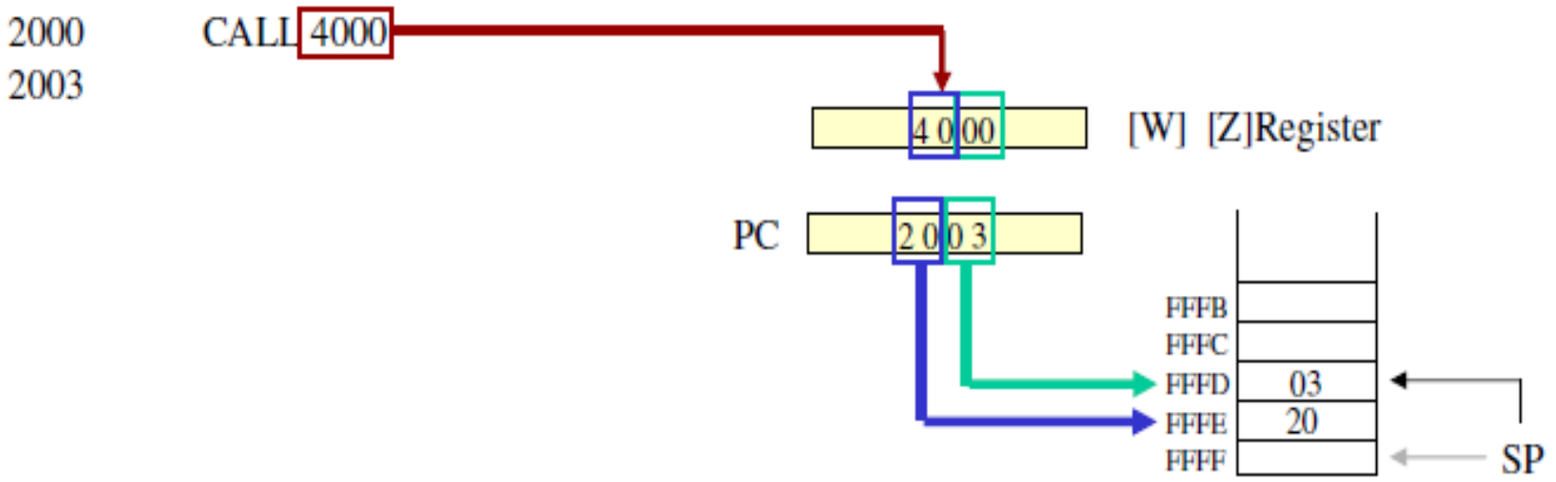
- The 8085 has two instructions for dealing with subroutines.

- The **CALL** instruction is used to redirect program execution to the subroutine.

- The **RTE** instruction is used to return the execution to the calling routine.

The CALL Instruction

- **CALL 4000H**
  - 3-byte instruction.
  - Push the address of the instruction immediately following the CALL onto the stack and decrement the stack pointer register by two.
  - Load the program counter with the 16-bit address supplied with the CALL instruction.
  - Jump Unconditionally to memory location.

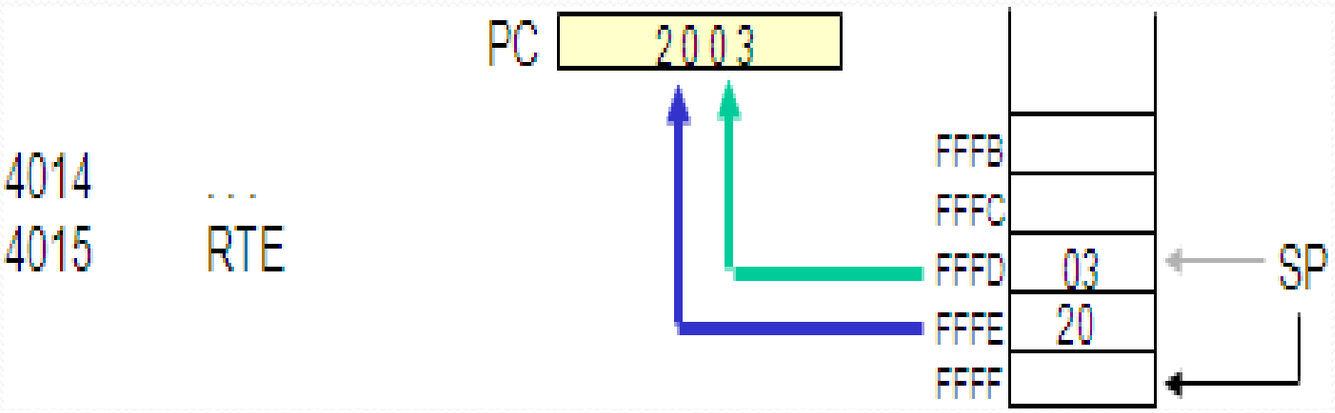


The CALL Instruction

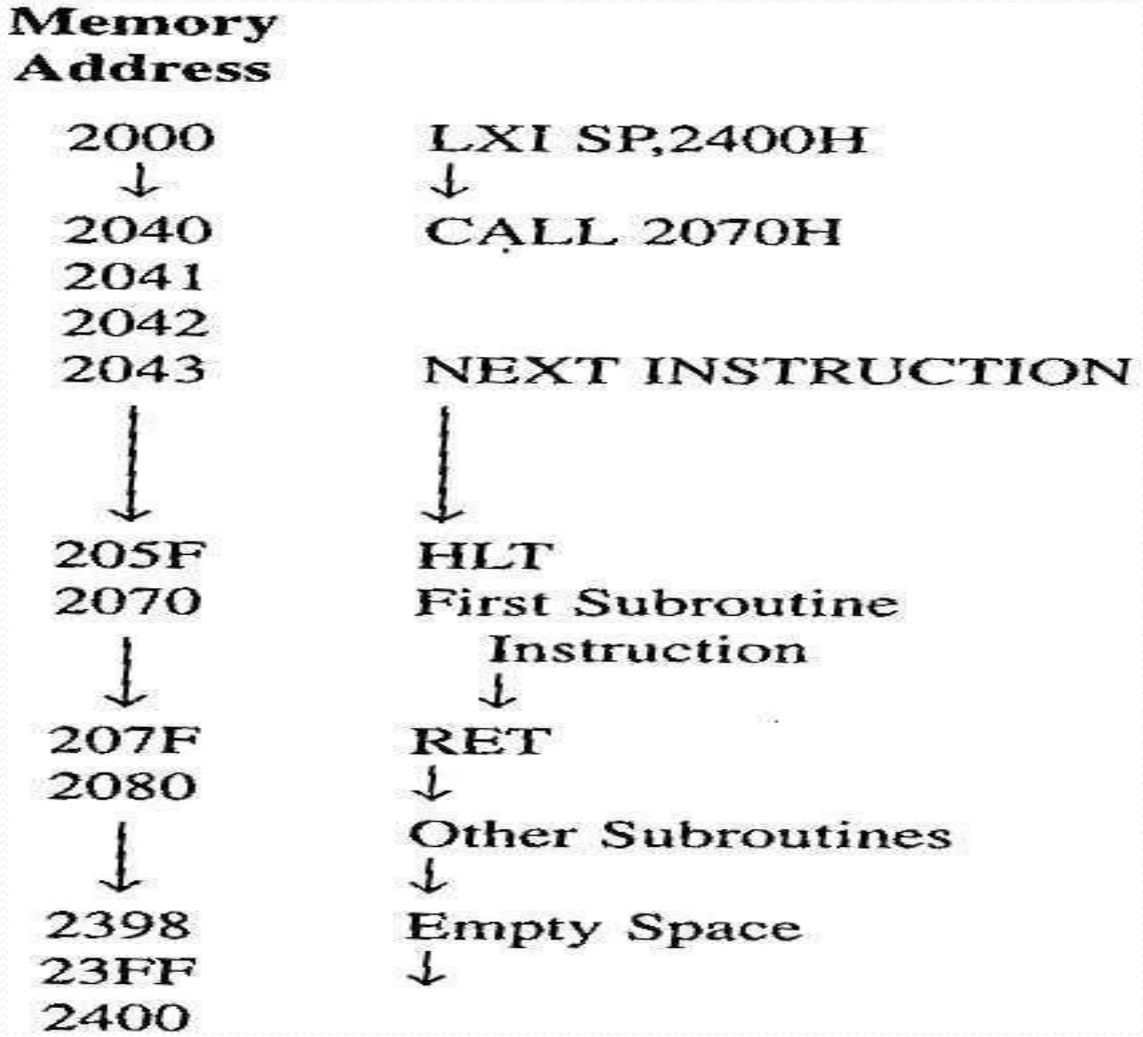
- MP Reads the subroutine address from the next two memory location and stores the higher order 8bit of the address in the W register and stores the lower order 8bit of the address in the Z register
- Push the address of the instruction immediately following the CALL onto the stack [Return address]
- Loads the program counter with the 16-bit address supplied with the CALL instruction from WZ register.

The RTE Instruction

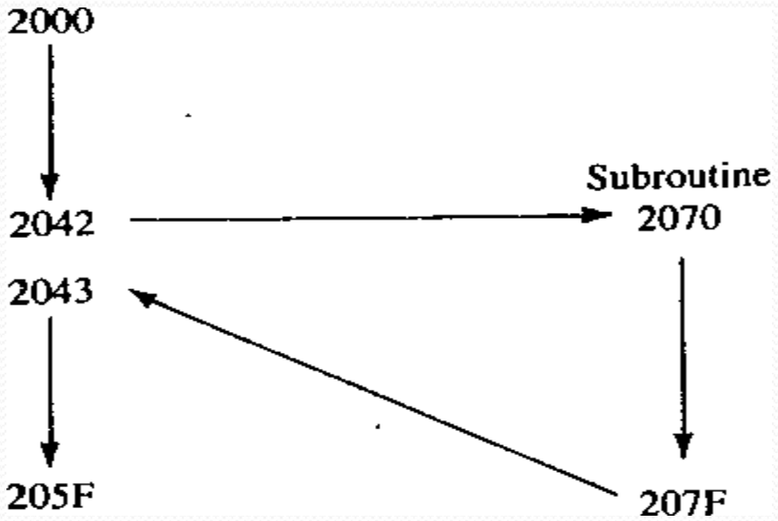
- **RTE**
  - 1-byte instruction
  - Retrieve the return address from the top of the stack and increments stack pointer register by two.
  - Load the program counter with the return address.
  - Unconditionally returns from a subroutine.



Illustrates the exchange of information between stack and Program Counter



Program Execution



Memory Address	Machine Code	Mnemonics	Comments
2040	CD	CALL 2070H	;Call subroutine located at the memory
2041	70		; location 2070H
2042	20		
2043	NEXT	INSTRUCTION	

CALL Execution

- **Instruction requires five machine cycles and eighteen T-states:** Call instruction is fetched, 16-bit address is read during M2 and M3 and stored temporarily in W/Z registers. In next two cycles content of program counter are stored on the stack (address from where microprocessor continue it execution of p

r

Instruction: CALL 2070H

Machine Cycles	Stack Pointer (SP) 2400	Address Bus (AB)	Program Counter (PCH) (PCL)	Data Bus (DB)	Internal Registers (W) (Z)
M_1 Opcode Fetch	23FF (SP–1)	2040	20 41	CD Opcode	—
M_2 Memory Read		2041	20 42	70 Operand	→ 70
M_3 Memory Read	23FF	2042	20 43	20 Operand	→ 20
M_4 Memory Write	23FE (SP–2)	23FF	20 43	20 (PCH)	
M_5 Memory Write	23FE	23FE	20 43	43 (PCL)	(20) (70)
M_1 Opcode Fetch of Next Instruction		20 70 → (W)(Z)	2071		(2070) → (W)(Z)

Memory Address	Code (H)
2040	CD
2041	70
2042	20

Data Transfer During the Execution of the CALL Instruction

RET Execution

- **Program execution sequence is transferred to the memory location 2043H location.M1 is normal fetch cycle during M2 contents of stack pointer are placed on address bus so 43H data is fetched and stored on Z register and SP is upgraded. Similarly for M3. Program sequence is transfered to2043H by placing contents of**

Memory Address	Code (H)
207F	C9

Contents of Stack Memory	
23FE	43
23FF	20

Machine Cycles	Stack Pointer (23FE)	Address Bus (AB)	Program Counter	Data Bus (DB)	Internal Registers (W) (Z)
M_1 Opcode Fetch	23FE	207F	2080	C9 Opcode	
M_2 Memory Read	23FF	23FE		43 (Stack) →	→ 43
M_3 Memory Read	2400	23FF		20 (Stack–1) →	→ 20
M_1 Opcode Fetch of Next Instruction		2043 (W) (Z) ←	2044		2043 (W) (Z) ←

Data Transfer During the Execution of the RET Instruction

Passing Data to a **Subroutine**

- In Assembly Language data is passed to a subroutine through registers.
- The data is stored in one of the registers by the calling program and the subroutine uses the value from the register.

- The other possibility is to use agreed upon memory locations.
- The calling program stores the data in the memory location and the subroutine retrieves the data from the location and uses it.

RST Instruction & RETURN INSTRUCTIONS

RESTART, CONDITIONAL CALL & RETURN INSTRUCTIONS

The conditional Call and Return instructions are based on four data conditions(flags): Carry, Zero, Sign, and Parity.

In case of a conditional Call the program is transferred to the subroutine if condition is met

In case of a conditional Return instruction, the sequence returns to the main program if the condition is met

Conditional CALL

CC	Call subroutine if Carry flag is set (CY = 1)
CNC	Call subroutine if Carry flag is reset (CY = 0)
CZ	Call subroutine if Zero flag is set (Z = 1)
CNZ	Call subroutine if Zero flag is reset (Z = 0)
CM	Call subroutine if Sign flag is set (S = 1, negative number)
CP	Call subroutine if Sign flag is reset (S = 0, positive number)

RESTART, CONDITIONAL CALL & RETURN INSTRUCTIONS

CPE	Call subroutine if Parity flag is set (P = 1, even parity)
CPO	Call subroutine if Parity flag is reset (P = 0, odd parity)

Conditional RETURN

RC	Return if Carry flag is set (CY = 1)
RNC	Return if Carry flag is reset (CY = 0)
RZ	Return if Zero flag is set (Z = 1)
RNZ	Return if Zero flag is reset (Z = 0)
RM	Return if Sign flag is set (S = 1, negative number)
RP	Return if Sign flag is reset (S = 0, positive number)
RPE	Return if Parity flag is set (P = 1, even parity)
RPO	Return if Parity flag is reset (P = 0, odd parity)

A Proper Subroutine

- According to Software Engineering practices, a proper subroutine:
- Is only entered with a CALL and exited with an RTE
  - **Has a single entry point**
  - **Do not use a CALL statement to jump into different points of the same subroutine.**

Writing Subroutines

Write a Program that will display FF and 11 repeatedly on the seven segment display. Write a 'delay' subroutine and Call it as necessary.

C000: LXI SP, FFFF
C003: MVI A, FF
C005: OUT 00
C007: CALL C014
C00A: MVI A, 11

C00C: OUT 00
C00E: CALL 1420
C011: JMP C003

Writing Subroutines

DELAY: C014: MVIB, FF

C016: MVIC, FF

C018: DCR C

C019: JNZ C018

C01C: DCR B

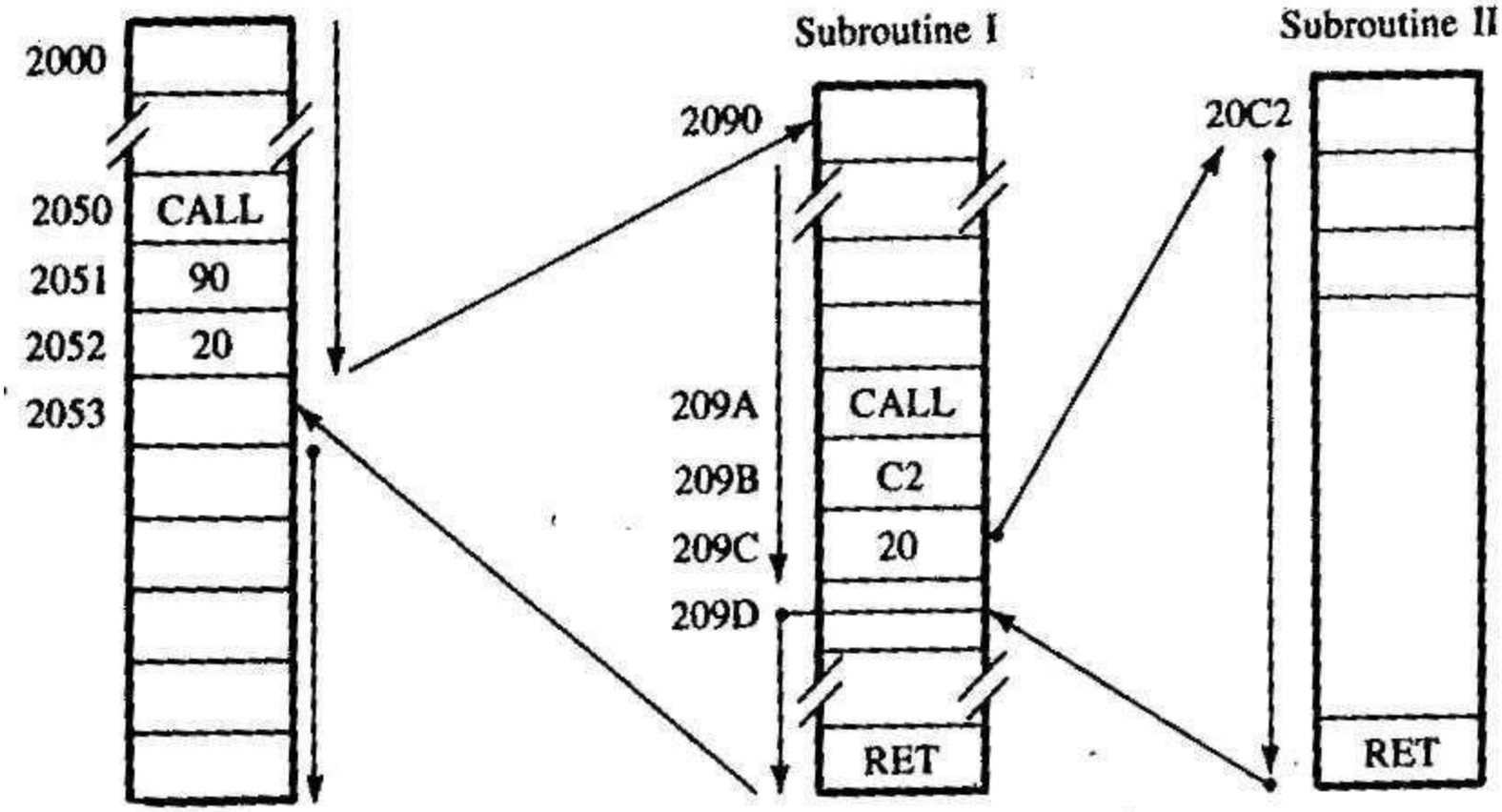
C01D: JNZ C016

C020: RET

Nesting Subroutines

The programming technique of a subroutine calling another subroutine

This process is limited only by the number of available stack locations.



Macros

- Macro is a group of instructions. The Macros in Microprocessor assembler generates the code in the program each time where the macro is ‘called’. Macros can be defined by MACRO and ENDM assembler directives. Creating macro is very similar to creating a new opcode that can be used in the program.

- It is important to note that macro sequences execute faster than subroutines because there are no CALL and RET instructions to execute. The assembler places the macro instructions in the program each time when it is invoked. This procedure is known as **Macro expansion**.

Thank You