Module 3

By

B N Kiran

Asst. Prof

Dept of Information Science and Engineering

Mysuru - 570008

Procedures and Macros

- **Definition:** A procedure is group of instructions that usually performs one task. It is a reusable section of a software program which is stored in memory once but can be used as often as necessary.
- A procedure can be of two types.
- 1) Near Procedure 2) Far Procedure
- **Near Procedure:** A procedure is known as NEAR procedure if is written(defined) in the same code segment which is calling that procedure. Only Instruction Pointer(IP register) contents will be changed in NEAR procedure.
- **FAR procedure :** A procedure is known as FAR procedure if it is written (defined) in the different code segment than the calling segment. In this case both Instruction Pointer(IP) and the Code Segment(CS) register content will be changed.

Directives used for procedure

- **PROC directive:** The PROC directive is used to identify the start of a procedure. The PROC directive follows a name given to the procedure. After that the term FAR and NEAR is used to specify the type of the procedure.
  ENDP Directive: This directive is used along with the name of the procedure to indicate the end of a procedure to the assembler. The PROC and ENDP directive are used to bracket a procedure.

CALL instruction and RET instruction

- **CALL instruction :** The CALL instruction is used to transfer execution to a procedure. It performs two operation. When it executes, first it stores the address of instruction after the CALL instruction on the stack. Second it changes the content of IP register in case of Near call and changes the content of IP register and cs register in case of FAR call. There are two types of calls.
- 1)Near Call or Intra segment call. 2) Far call or Inter Segment call Operation for Near Call : When 8086 executes a near CALL instruction, it decrements the stack pointer by 2 and copies the IP register contents on to the stack.Then it copies address of first instruction of called procedure.
- SP ← SP - 2
- IP → stores onto stack
- IP ← starting address of a procedure.

- **Operation of FAR CALL:** When 8086 executes a far call, it decrements the stack pointer by 2 and copies the contents of CS register to the stack. It the decrements the stack pointer by 2 again and copies the content of IP register to the stack. Finally it loads CS register with base address of segment having procedure and IP with address of first instruction in procedure.

- SP ← SP-2.
- cs contents → stored on stack.
- SP ← sp-2.
- IP contents → stored on stack.
- CS ← Base address of segment having procedure.
- IP ← address of first instruction in procedure.

RET inst

- **RET instruction :** The RET instruction will return execution from a procedure to the next instruction after call in the main program. At the end of every procedure RET instruction must be executed. Operation for Near Procedure : For NEAR procedure ,the return is done by replacing the IP register with a address popped off from stack and then SP will be incremented by 2.
- IP ← Address from top of stack
- SP ← SP+2

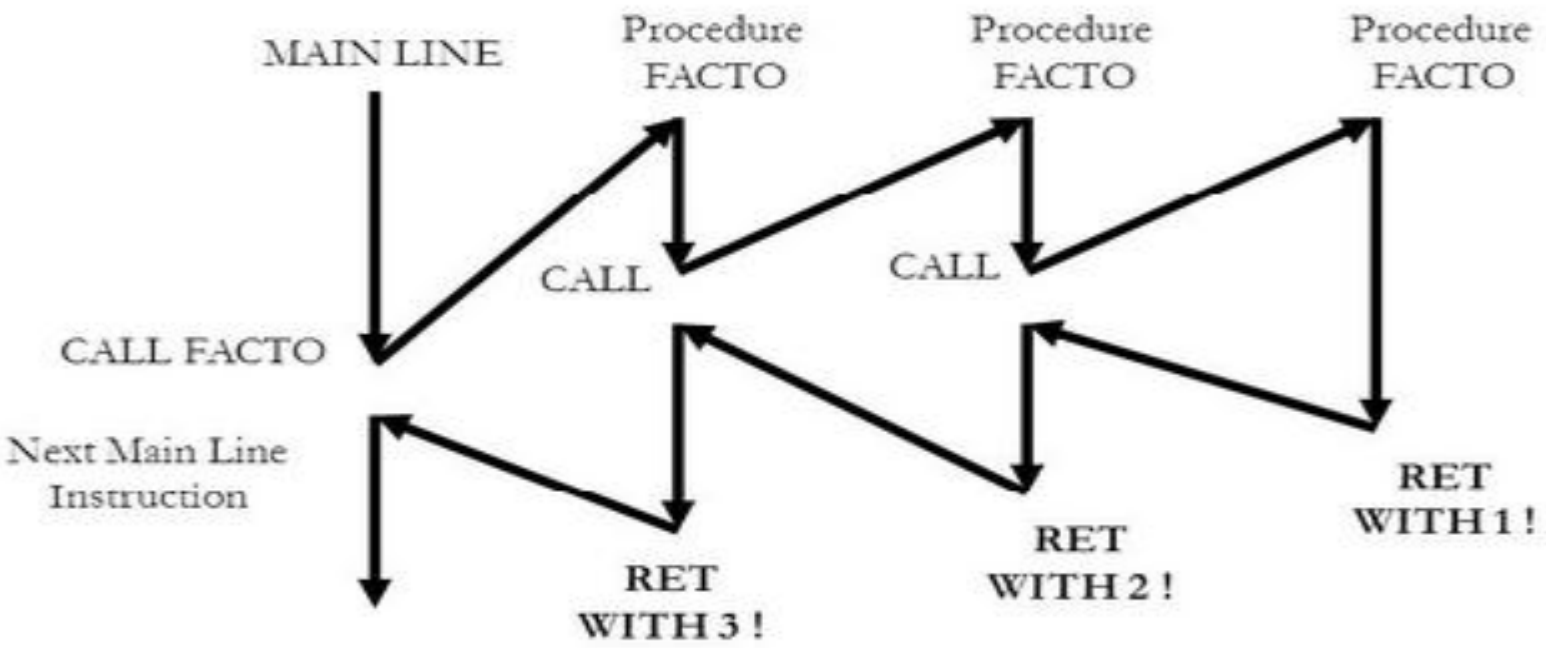
- **Operation for FAR procedure :** IP register is replaced by address popped off from top of stack, then SP will be incremented by 2.The CS register is replaced with a address popped off from top of stack. Again SP will be incremented by 2.
- IP ← Address from top of stack
- SP ← SP+2
- CS ← Address from top of stack
- SP ← SP+2

Difference between FAR CALL and NEAR CALL.

NEAR CALL	FAR CALL
A near call refers a procedure which is in the same code segment.	A Far call refers a procedure which is in different code segment
It is also called Intra-segment call.	It is also called Inter-segment call
A Near Call replaces the old IP with new IP	A FAR replaces CS & IP with new CS & IP
It uses keyword near for calling procedure.	It uses keyword far for calling procedure.
Less stack locations are required	More stack locations are required.

Recursive Procedure

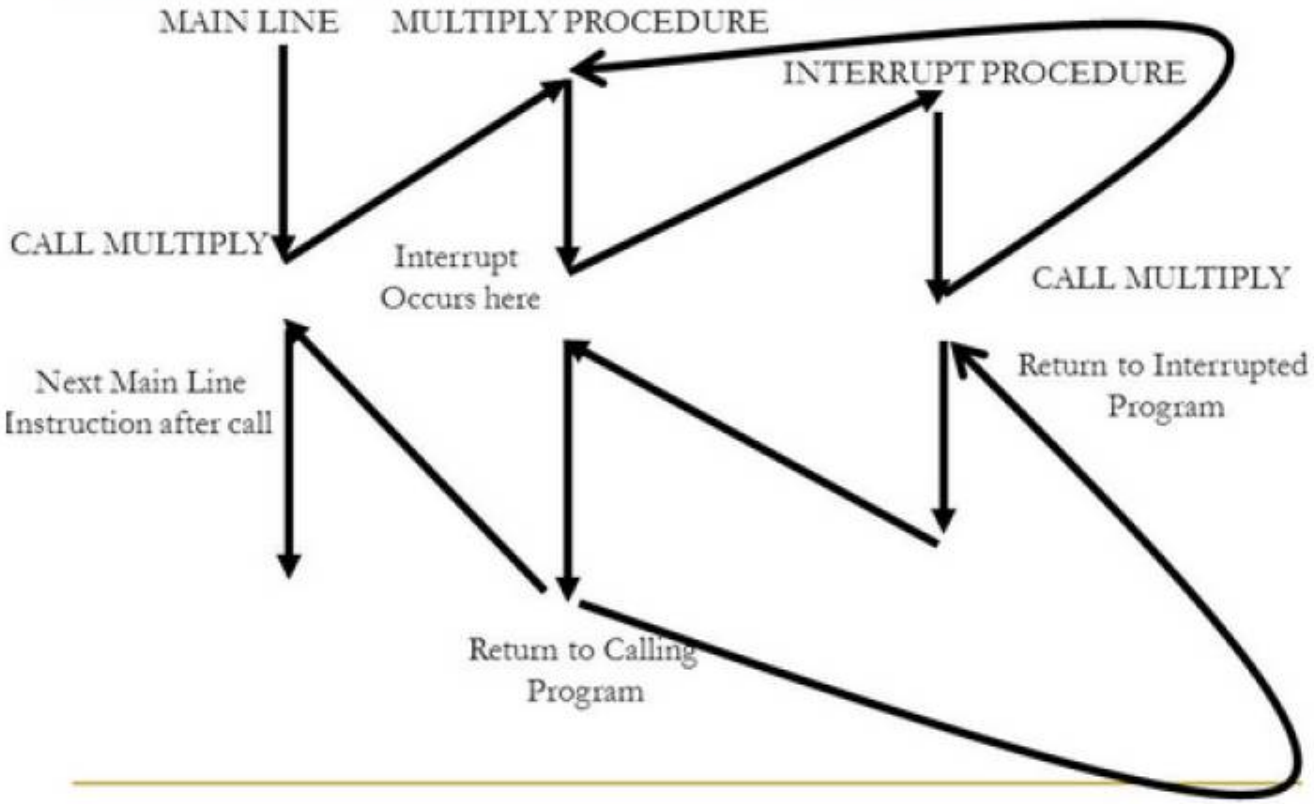
- **Recursive Procedure :** It is a procedure which call itself. Recursive procedures are used to work with complex data structure like trees. If procedure is called with N (recursive depth) then N is decremented by one after each procedure CALL and the procedure is called until n=0.Recursive procedure takes less time to implement a particular task. But it needs certain condition for it’s termination.


MAIN LINE
Procedure FACTO
Procedure FACTO
Procedure FACTO
CALL
CALL
CALL FACTO
Next Main Line Instruction
RET WITH 3 !
RET WITH 2 !
RET WITH 1 !

Re-entrant procedure

- **Re-entrant procedure :** In some situation, it may happen that procedure 1 is called from main program and procedure 2 is called from procedure 1.And again procedure 1 is called from procedure 2. In this situation , program execution flow re-enters in the procedure 1 (first time when procedure 1 was called from main program and second time when procedure 1 was called from procedure 2) .Hence this type of procedure is called as Reentrant procedure.

REENTRANT PROCEDURES



Parameter passing in 8086

- Procedures are written to process data or address variables from the main program.
- To achieve this, it is necessary to pass the information about address, variables or data. This technique is called as parameter passing.
- The four major ways of passing parameters to and from a procedure are:
  - Passing parameters using registers
  - Passing parameters using memory
  - Passing parameters using pointers
  - Passing parameters using stack

- The data to be passed is stored in the registers and these registers are accessed in the procedure to process the data.
- The disadvantage of using registers to pass parameters is that the number of registers limits the number of parameters you can pass.
- E.g. An array of 100 elements can’t be passed to a procedure using registers.

```
.model small
.data
MULTIPLICAND DW 1234H
MULTIPLIER DW 4232H
.code MOV AX, MULTIPLICAND
MOV BX, MULTIPLIER
CALL MULTI
:
:
MULTI PROC NEAR
MUL BX ; Procedure to access data from BX register
RET
MULTI  ENDP
:
:
END
```

- **Passing parameters using memory-**
- In the cases where few parameters have to be passed to and from a procedure, registers are convenient. But, in cases when we need to pass a large number of parameters to procedure, we use memory. This memory may be a dedicated section of general memory or a part of it.

```
.model small
.data
MULTIPLICAND DW 1234H ; Storage for multiplicand value
MULTIPLIER DW 4232H ; Storage for multiplier value
MULTIPLICATION DW ? ; Storage for multiplication result .code
MOV AX, @Data
MOV DS, AX
:
:
CALL MULTI
:
:
MULTI PROC NEAR
MOV AX, MULTIPLICAND
MOV BX, MULTIPLIER
:
: MOV MULTIPLICATION, AX ; Store the multiplication value in named memory location
RET
MULTI ENDP END
```

- **Passing parameter using pointers-**
- A parameter passing method which overcomes the disadvantage of using data item names (i.e. variable names) directly in a procedure is to use registers to pass the procedure pointers to the desired data.

- .model small
- .data MULTIPLICAND DB 12H ; Storage for multiplicand value
- MULTIPLIER DB 42H ; Storage for multiplier value
- MULTIPLICATION DW ? ; Storage for multiplication result
- .code MOV AX, @Data
- MOV DS, AX
- MOV SI, OFFSET MULTIPLICAND
- MOV DI, OFFSET MULTIPLIER
- MOV BX, OFFSET MULTIPLICATION
- CALL MULTI
- :
- :

- MULTI PROC NEAR
- :
- :
- MOV AL, [SI] ; Get multiplicand value pointed by SI in accumulator
- MOV BL, [DI] ; Get multiplier value pointed by DI in BL
- :
- : MOV [BX], AX ; Store result in location pointed out by BX
- RET
- MULTI ENDP END

Passing parameters using stack

- We push them on the stack before the call for the procedure in the main program. The instructions used in the procedure read these parameters from the stack. Whenever stack is used to pass parameters it is important to keep a track of what is pushed on the stack and what is popped off the stack in the main program.

```
.model small
.data
        MULTIPLICAND DW 1234H
        MULTIPLIER DW 4232H
        .code MOV AX, @data
        MOV DS, AX
        :
        :
        PUSH MULTIPLICAND
        PUSH MULTIPLIER
         CALL MULTI
        :
        :
        MULTI PROC NEAR
        PUSH BP MOV BP, SP ; Copies offset of SP into BP
        MOV AX, [BP + 6] ; MULTIPLICAND value is available at ; [BP + 6] and is
            passed to AX
        MUL WORD PTR [BP + 4] ; MULTIPLIER value is passed POP BP
        RET ; Increments SP by 4 to return address
        MULTI ENDP ;
        End procedure END
```

Macros in 8086

- A **Macro** is a set of instructions grouped under a single unit. It is another method for implementing modular programming in the 8086 microprocessors (The first one was using Procedures).
- The **Macro** is different from the Procedure in a way that unlike calling and returning the control as in procedures, the processor generates the code in the program every time whenever and wherever a call to the **Macro** is made.
- A **Macro** can be defined in a program using the following assembler directives: **MACRO** (used after the name of Macro before starting the body of the Macro) and **ENDM** (at the end of the Macro). All the instructions that belong to the Macro lie within these two assembler directives. The following is the syntax for defining a **Macro in the 8086 Microprocessor**:

- Macro_name MACRO [list of parameters]
  Instruction 1
- Instruction 2
- - - - - - - - - -
- - - - - - - - - -
- - - - - - - - - -
- Instruction n
- ENDM

- **Disadvantages of Procedure**
- 1. Linkage associated with them.
-
- 2. It sometimes requires more code to program the linkage than is needed to perform the task. If this is the case, a procedure may not save memory and execution time is considerably increased.
- 3. **Macros** is needed for providing the programming ease of a procedure while avoiding the linkage. Macro is a segment of code that needs to be written only once but whose basic structure can be caused to be repeated several times within a source module by placing a single statement at the point of each reference.
-
- A macro is unlike a procedure in that the machine instructions are repeated each time the macro is referenced. Therefore, no memory is saved, but programming time is conserved (no linkage is required) and some degree of modularity is achieved. The code that is to be repeated is called the prototype code. The prototype code along with the statements for referencing and terminating is called the macro definition.

- Macros can be defined with MACRO and ENDM
- Format

  macro_name MACRO [parameter1, parameter2, ...]

  macro body

  ENDM
- A macro can be invoked using macro_name [argument1, argument2, …]
- Example: Definition Invocation

```
multAX_by_16 MACRO
...
sal AX,4
ENDM

Invocation
mov AX,27
multAX_by_16 …
```

Macros with Parameters

- Macros can be defined with parameters
  - More flexible
  - More useful
- Example

```
mult_by_16 MACRO operand
  sal operand,4
ENDM
```

- To multiply a byte in DL register

```
mult_by_16 DL
```

- To multiply a memory variable count

```
mult_by_16  count
```

Macros with Parameters (cont'd)

- Example: To exchange two memory words
- Wmxchg MACRO operand1, operand2

```
            xchg AX,operand1
            xchg AX,operand2
            xchg AX,operand1
ENDM
```

- Example: To exchange two memory bytes

```
Bmxchg MACRO operand1, operand2
            xchg AL,operand1
            xchg AL,operand2
            xchg AL,operand1
 ENDM
```

Macros vs. Procedures

- Similar to procedures in some respects
  - Both improve programmer productivity
  - Aids development of modular code
  - Can be used when a block of code is repeated in the source program
- Some significant differences as well
  - Parameter passing
  - Types of parameters
  - Invocation mechanism

Macros vs. Procedures (cont'd)

- Parameter passing
  - In macros, similar to a procedure call in a HLL
    mult_by_16 AX
- In procedures, we have to push parameters onto the stack
  - push AX call times16
- Macros can avoid
  - Parameter passing overhead
    - Proportional to the number of parameters passed
  - call/ret overhead

Macros vs. Procedures (cont'd)

- Types of parameters
  - Macros allow more flexibility in the types of parameters that can be passed
    - Result of it being a text substitution mechanism
  - Example

```
shift MACRO op_code,operand,count
op_code operand,count
ENDM
```

  - can be invoked as shift sal,AX,3
  - which results in the following expansion sal AX,3

Macros vs. Procedures (cont'd)

- Invocation mechanism
- Macro invocation
  - Done at assembly time by text substitution *
- Procedure invocation
  - Done at run time
  - Tradeoffs

Type of overhead	Procedure	Macro
Memory space	lower	higher
Execution time	higher	lower
Assembly time	lower	higher

When Are Macros Better?

- Useful to extend the instruction set by defining macro-instructions

```
times16   PROC
          push   BP
          mov    BP,SP
          push   AX
          mov    AX,[BP+4]
          sal    AX,4
          mov    [BP+4],AX
          pop    AX
          pop    BP
          ret    2
times16   ENDP
```

Invocation

```
push    count
call    times16
pop     count
```

Too much overhead
Use of procedure is impractical

When Are Macros Better? (cont'd)

- Sometimes procedures cannot be used
  - Suppose we want to save and restore BX, CX, DX, SI, DI, and BP registers
  - Cannot use **pusha and popa as they include AX as well**

```
save_regs  MACRO
           push   BP
           push   DI
           push   SI
           push   DX
           push   CX
           push   BX
           ENDM
```

```
restore_regs  MACRO
              pop    BX
              pop    CX
              pop    DX
              pop    SI
              pop    DI
              pop    BP
              ENDM
```

Labels in Macros

- Problem with the following macro definiton

```
to_upper0     MACRO     ch
              cmp       ch,'a'
              jb        done
              cmp       ch,'z'
              ja        done
              sub       ch,32
        done:
              ENDM
```

- If we invoke it more than once, we will have duplicate label **done**

Labels in Macros (cont'd)

- Solution: Use LOCAL directive
- Format: **LOCAL local_label1 [,local_label2,…]**

```
to_upper    MACRO   ch
            LOCAL   done
            cmp     ch,'a'
            jb      done
            cmp     ch,'z'
            ja      done
            sub     ch,32
done:
            ENDM
```

Assembler uses labels

??XXXX

where **XXXX** is between 0 and FFFFH

To avoid conflict, do not use labels that begin with **??**

Comments in Macros

- We don’t want comments in a macro definition to appear every time it is expanded
  - The ;; operator suppresses comments in the expansions
- **;;Converts a lowercase letter to uppercase.**

```
to_upper  MACRO   ch
          LOCAL   done
          ; case conversion macro
          cmp     ch,'a'   ;; check if ch >= 'a'
          jb      done
          cmp     ch,'z'   ;;   and if ch >= 'z'
          ja      done
          sub     ch,32    ;; then ch := ch - 32
done:
          ENDM
```

Comments in Macros (cont'd)

- • Invoking the **to_upper macro by**

```
mov      AL,'b'
to_upper    AL
mov      BL,AL
mov      AH,'l'
to_upper    AH
mov      BH,AH
```

- generates the following macro expansion

Comments in Macros (cont'd)

```
                    17 0000  B0 62                 mov     AL,'b'
                    18                           to_upper   AL
1                   19                           ; case conversion macro
1                   20 0002  3C 61                cmp     AL,'a'
1                   21 0004  72 06                jb      ??0000
1                   22 0006  3C 7A                cmp     AL,'z'
1                   23 0008  77 02                ja      ??0000
1                   24 000A  2C 20                sub     AL,32
1                   25 000C                    ??0000:
                    26 000C  8A D8                mov     BL,AL
                    27 000E  B4 31                mov     AH,'1'
```

Comments in Macros (cont'd)

```
     28                                 to_upper   AH
1    29                                 ; case conversion macro
1    30 0010  80 FC 61                  cmp     AH,'a'
1    31 0013  72 08                     jb      ??0001
1    32 0015  80 FC 7A                  cmp     AH,'z'
1    33 0018  77 03                     ja      ??0001
1    34 001A  80 EC 20                  sub     AH,32
1    35 001D                       ??0001:
     36 001D  8A FC                     mov     BH,AH
```

Macro operators

Five operators

;;	Suppress comment
&	Substitute
< >	Literal-text string
!	Literal-character
%	Expression evaluate

* We have already seen ;; operator
* We will discuss the remaining four operators

Macro operators

Substitute operator (&)

* Substitutes a parameter with the actual argument

- Syntax: **&name**

```
sort2     MACRO  cond, num1, num2
          LOCAL   done
          push    AX
          mov     AX,num1
          cmp     AX,num2
          j&cond  done
          xchg    AX,num2
          mov     num1,AX
     done:
          pop     AX
          ENDM
```

Macro Operators (cont'd)

- To sort two unsigned numbers **value1 and value2, use**

  sort2 ae,value1,value2

  generates the following macro expansion

  push AX
  mov AX,value1
  cmp AX,value2
  jae ??0000
  xchg AX,value2
  mov value1,AX
  ??0000:
  pop AX

- To sort two signed numbers **value1 and value2, use**

  sort2 ge,value1,value2

Macro Operators (cont'd)

- **Literal-text string operator (< >)**
- Treats the enclosed text as a single string literal rather than separate arguments
- Syntax: **<text>**

```
range_error1     MACRO     number,variable,range
err_msg&number     DB    '&variable: out of range',0
range_msg&number   DB    'Correct range is &range',0
```

- Invoking with

```
range_error1      1,<Assignment mark>,<0 to 25>
```

produces

```
err_msg1     DB    'Assignment mark: out of range',0
range_msg1   DB    'Correct range is 0 to 25',0
```

Macro Operators (cont’d)

- **Literal-character operator (!)**
  Treats the character literally without its default meaning
  Syntax: **!character**
  range_error2 MACRO number,variable,range
  err_msg&number DB '&variable: out of range - &range',0
  ENDM
- Invoking with
  range_error2 3,mark,<can!'!'t be !> 100>
  produces
- **err_msg3 DB 'mark: out of range - can''t be > 100',0**
- Without the ! operator, two single quotes will produce a single quote in the output

Macro Operators (cont'd)

- **Expression Evaluate operator (%)**
- Expression is evaluated and its value is used to replace the expression itself

  Syntax: **%expression**

  init_arry MACRO element_size,name,size,init_value

  name D&element_size size DUP (init_value)

  ENDM

  Assuming **NUM_STUDENTS EQU 47**

  NUM_TESTS EQU 7

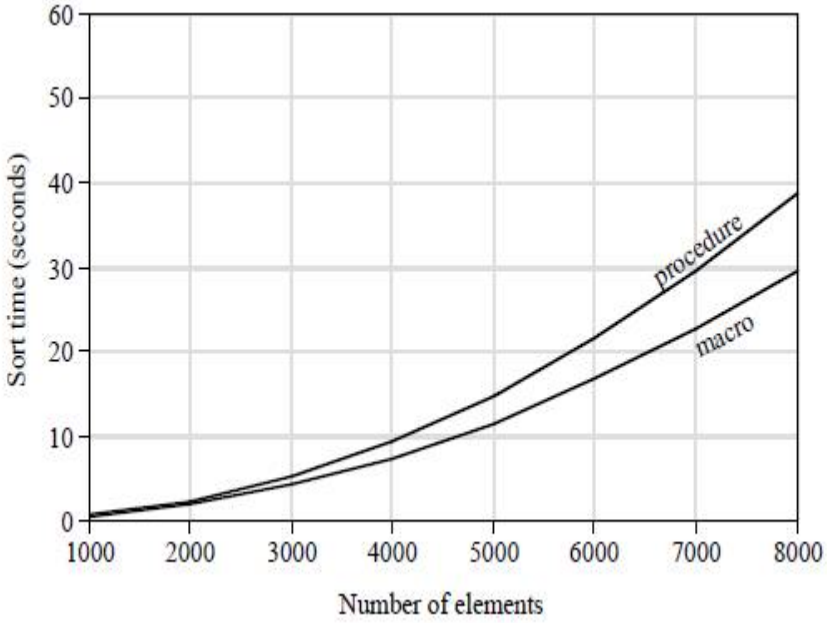
  Invoking with

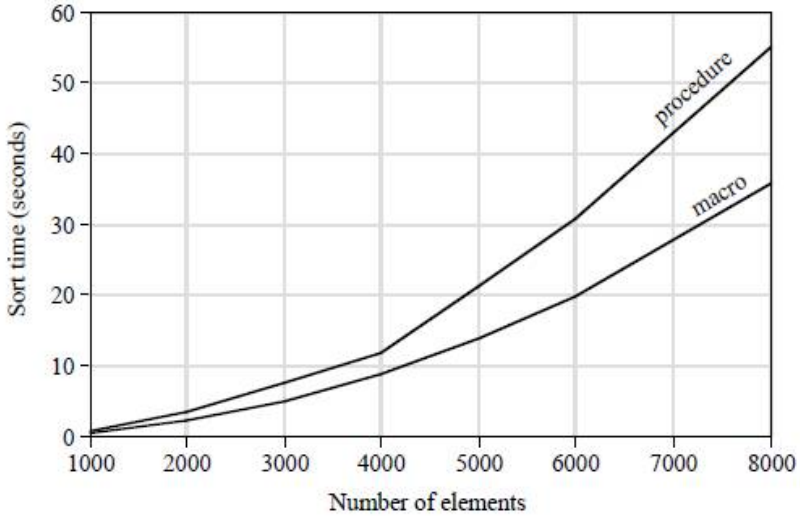
  init_array W,marks,%NUM_STUDENTS*NUM_TESTS,-1

  produces

  marks DW 329 DUP (-1)

Performance



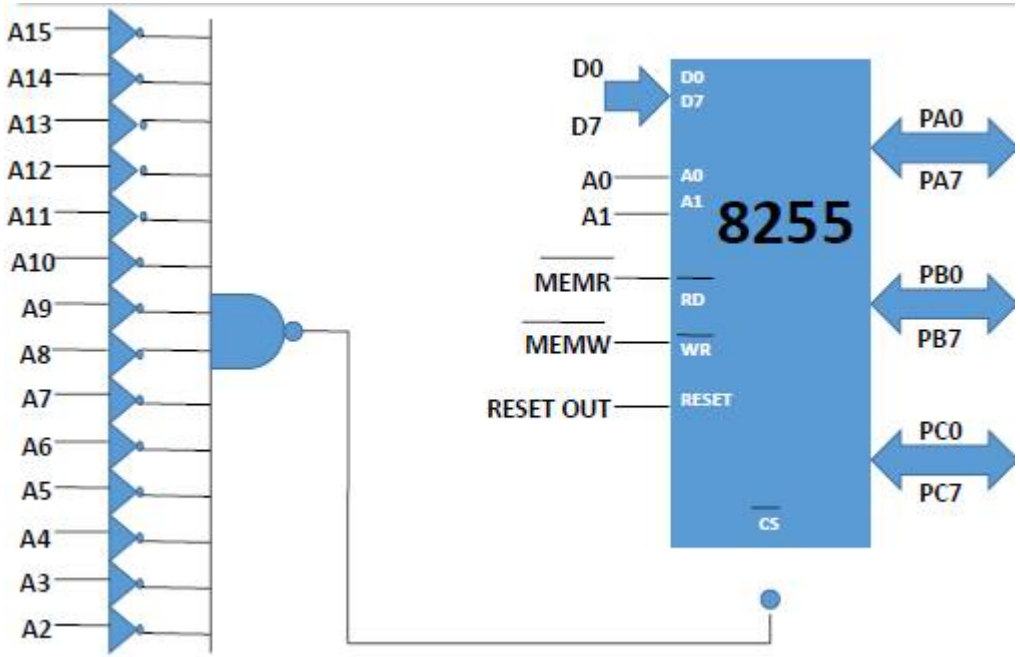


Accessing hardware

- Memory mapped I/O
- I/O mapped I/O

Memory mapped I/O

- Device address is of 16 Bit. means A_0 to A_{15} lines are used to generate device address.
- $\overline{\text{MEMR}}$ and $\overline{\text{MEMW}}$ control signals are used to control read and write I/O operations.
- Data transfer is between Any register and I/O device.
- Maximum number of I/O devices are 65536.
- Decoding 16 bit address may requires more hardware.
- For e.g. MOV R M, ADD M,CMP M etc.


A15
A14
A13
A12
A11
A10
A9
A8
A7
A6
A5
A4
A3
A2
D0
D7
A0
A1
MEMR
MEMW
RESET OUT
D0
D7
A0
A1
8255
RD
WR
RESET
CS
PA0
PA7
PB0
PB7
PC0
PC7

I/O mapped I/O

- **Device address is of 8 Bit. means A0 to A7 or A8 to A15 lines are used to generate device address.**
- **IOR and IOW control signals are used to control read and write I/O operations.**
- **Data transfer is between Accumulator and I/O device.**
- **Maximum number of I/O devices are 256.**
- **Decoding 16 bit address may requires less hardware.**
- **For e.g. IN, OUT etc.**

8255A - Programmable Peripheral Interface

- The 8255A is a general purpose programmable I/O device designed to transfer the data from I/O to interrupt I/O under certain conditions as required. It can be used with almost any microprocessor.
- It consists of three 8-bit bidirectional I/O ports (24I/O lines) which can be configured as per the requirement.

8255A - Programmable Peripheral Interface contd …

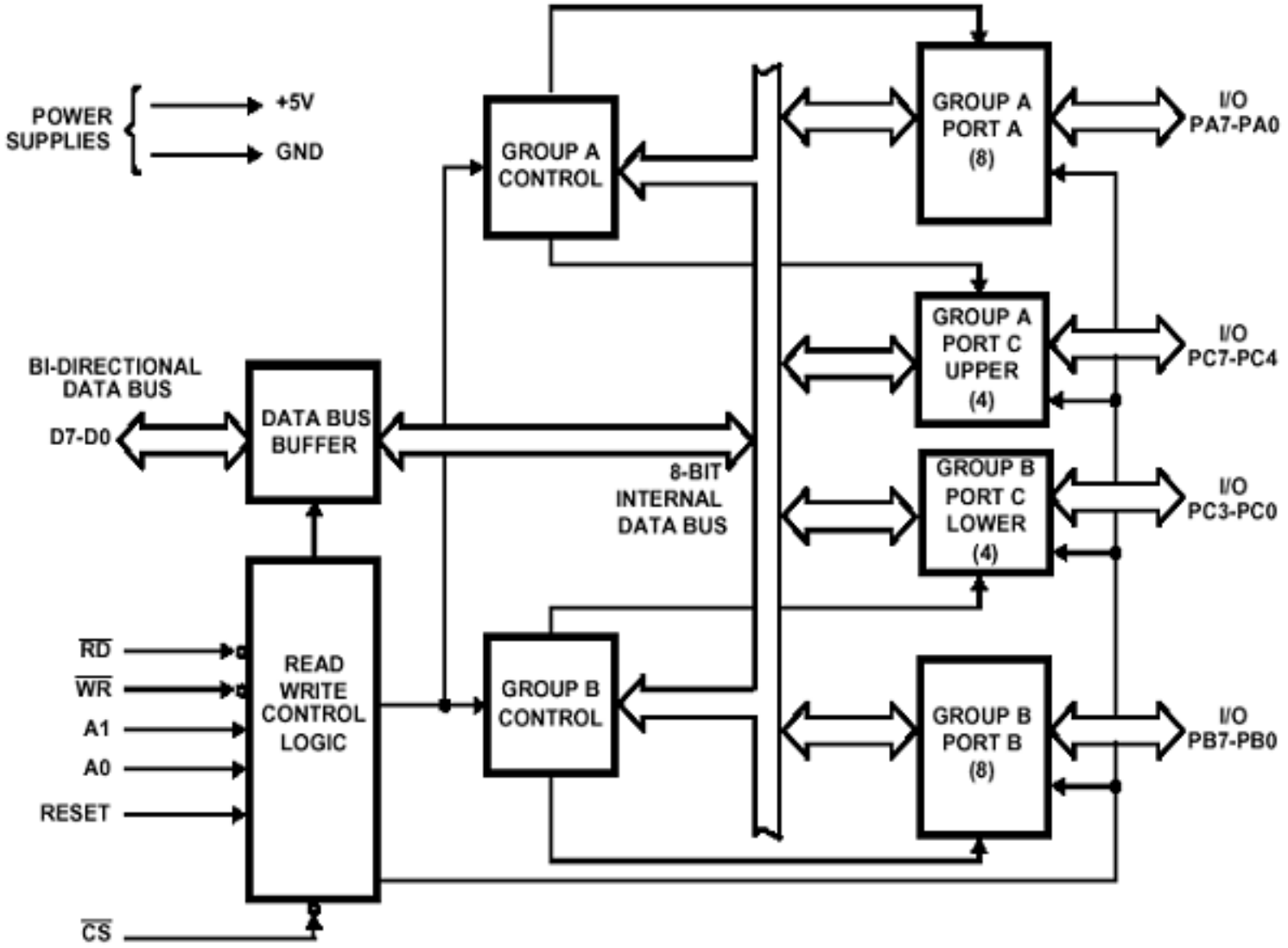
- Ports of 8255A
  - 8255A has three ports, i.e., PORT A, PORT B, and PORT C.
  - **Port A** contains one 8-bit output latch/buffer and one 8-bit input buffer.
  - **Port B** is similar to PORT A.
  - **Port C** can be split into two parts, i.e. PORT C lower (PC0-PC3) and PORT C upper (PC7-PC4) by the control word.
- These three ports are further divided into two groups, i.e. Group A includes PORT A and upper PORT C. Group B includes PORT B and lower PORT C. These two groups can be programmed in three different modes, i.e. the first mode is named as mode 0, the second mode is named as Mode 1 and the third mode is named as Mode 2.

8255A - Programmable Peripheral Interface contd …

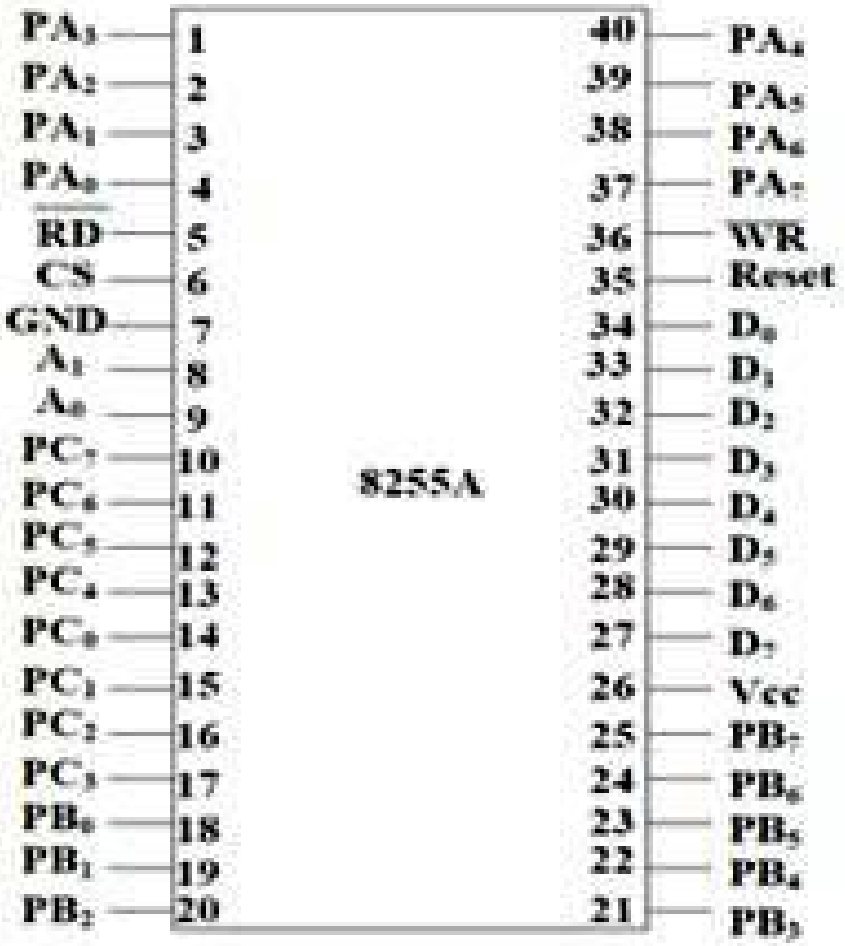
- Operating Modes
- 8255A has three different operating modes −
- **Mode 0** − In this mode, Port A and B is used as two 8-bit ports and Port C as two 4-bit ports. Each port can be programmed in either input mode or output mode where outputs are latched and inputs are not latched. Ports do not have interrupt capability.
- **Mode 1** − In this mode, Port A and B is used as 8-bit I/O ports. They can be configured as either input or output ports. Each port uses three lines from port C as handshake signals. Inputs and outputs are latched.
- **Mode 2** − In this mode, Port A can be configured as the bidirectional port and Port B either in Mode 0 or Mode 1. Port A uses five signals from Port C as handshake signals for data transfer. The remaining three signals from Port C can be used either as simple I/O or as handshake for port B.

- Features of 8255A
  - The prominent features of 8255A are as follows –
  - It consists of 3 8-bit IO ports i.e. PA, PB, and PC.
  - Address/data bus must be externally demux'd.
  - It is TTL compatible.
  - It has improved DC driving capability.

Architecture of 8255



Pin diagram of 8255



- **PA0 – PA7 –** Pins of port A
- **PB0 – PB7 –** Pins of port B
- **PC0 – PC7 –** Pins of port C
- **D0 – D7 –** Data pins for the transfer of data
- **RESET –** Reset input
- **RD' –** Read input
- **WR' –** Write input
- **CS' –** Chip select
- **A1 and A0 –** Address pins

- Data Bus Buffer
  - It is a tri-state 8-bit buffer, which is used to interface the microprocessor to the system data bus. Data is transmitted or received by the buffer as per the instructions by the CPU. Control words and status information is also transferred using this bus.
- Read/Write Control Logic
  - This block is responsible for controlling the internal / external transfer of data / control / status word. It accepts the input from the CPU address and control buses, and in turn issues command to both the control groups.

- CS
- It stands for Chip Select. A LOW on this input selects the chip and enables the communication between the 8255A and the CPU. It is connected to the decoded address, and A_0 & A_1 are connected to the microprocessor address lines.
- Their result depends on the following conditions −

CS	A1	A0	Result
0	0	0	PORT A
0	0	1	PORT B
0	1	0	PORT C
0	1	1	Control Register
1	x	x	No Selection

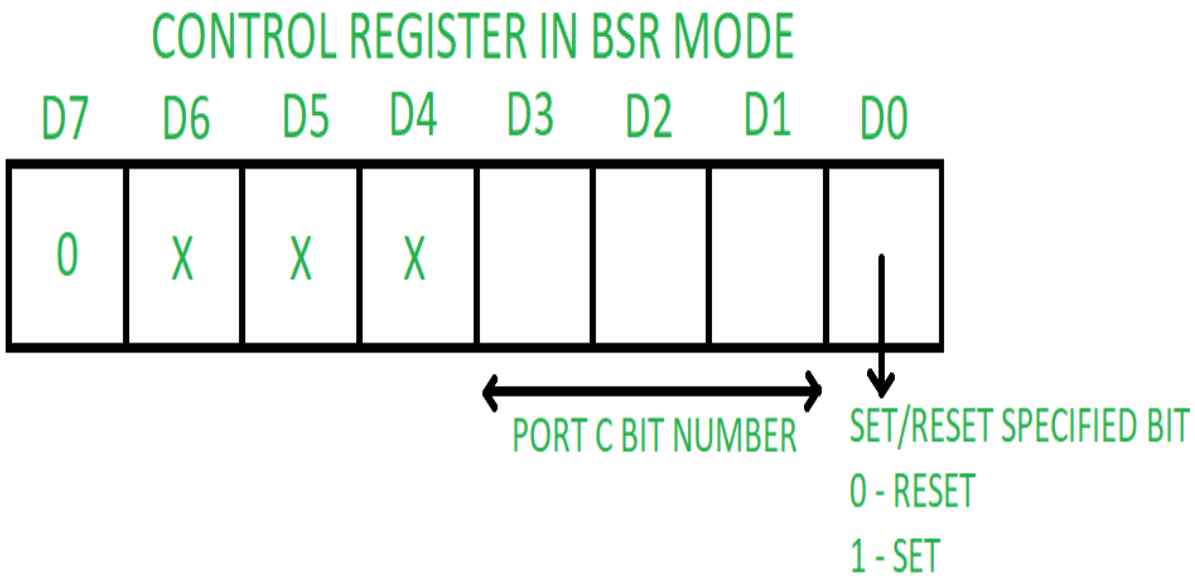
- WR
  - It stands for write. This control signal enables the write operation. When this signal goes low, the microprocessor writes into a selected I/O port or control register.
- RESET
  - This is an active high signal. It clears the control register and sets all ports in the input mode.
- RD
  - It stands for Read. This control signal enables the Read operation. When the signal is low, the microprocessor reads the data from the selected I/O port of the 8255.
- A_0 and A_1
  - These input signals work with RD, WR, and one of the control signal. Following is the table showing their various signals with their result.

A_1	A_0	RD	WR	CS	Result
0	0	0	1	0	**Input Operation** PORT A → Data Bus
0	1	0	1	0	PORT B → Data Bus
1	0	0	1	0	PORT C → Data Bus
0	0	1	0	0	**Output Operation** Data Bus → PORT A
0	1	1	0	0	Data Bus → PORT A
1	0	1	0	0	Data Bus → PORT B
1	1	1	0	0	Data Bus → PORT D

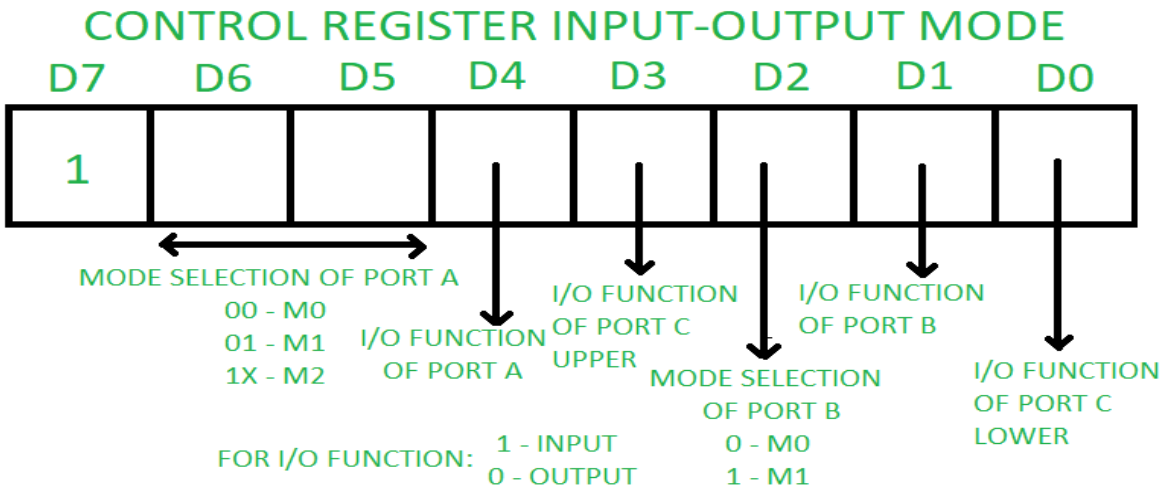
Operating modes –

1.Bit set reset (BSR) mode –

If MSB of control word (D7) is 0, PPI works in BSR mode. In this mode only port C bits are used for set or reset.



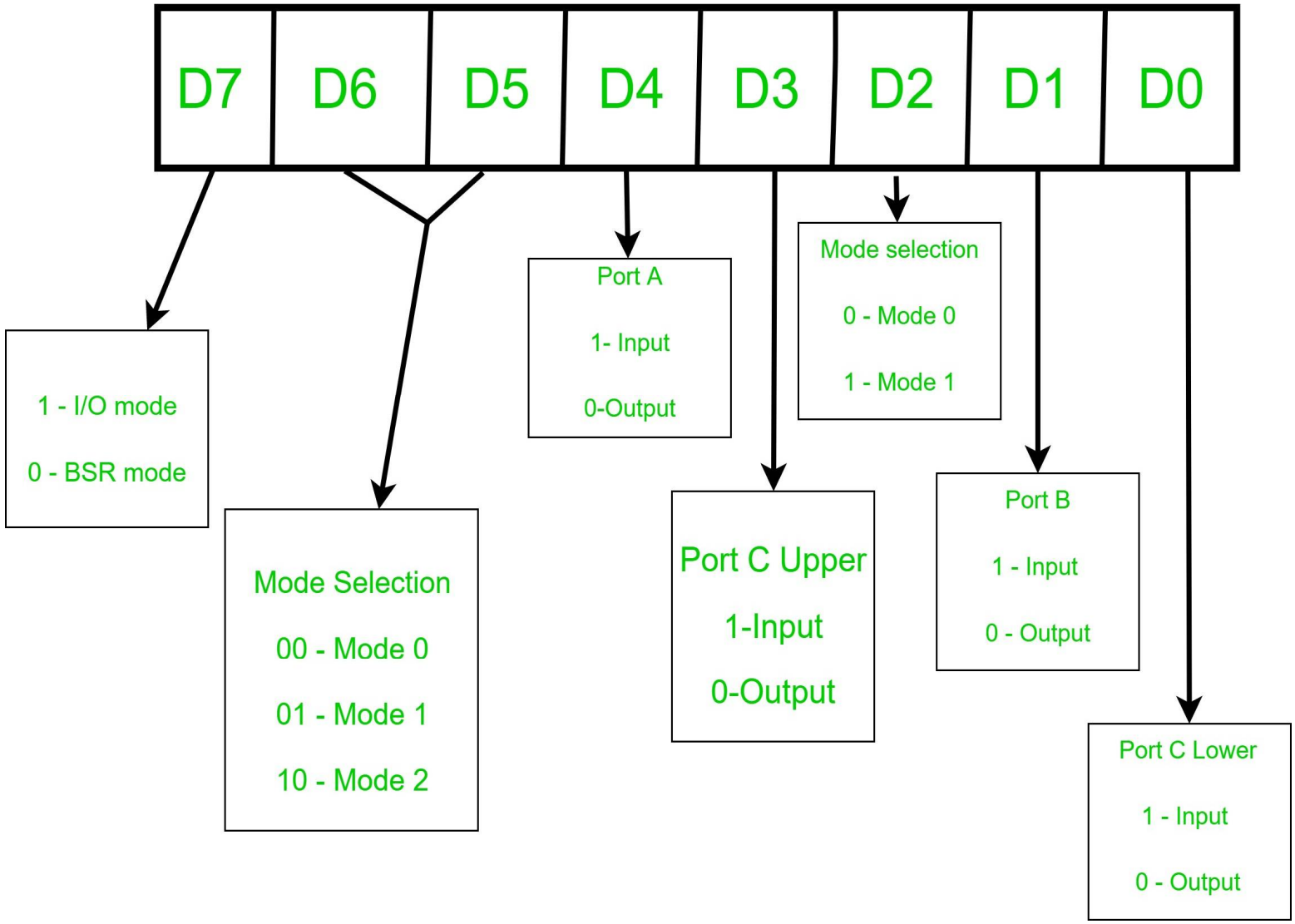
Input-Output mode –
If MSB of control word (D7) is 1, PPI works in input-output mode. This is further divided into three modes:



- **Mode 0 –**In this mode all the three ports (port A, B, C) can work as simple input function or simple output function. In this mode there is no interrupt handling capacity.
- **Mode 1 –** Handshake I/O mode or strobbed I/O mode. In this mode either port A or port B can work as simple input port or simple output port, and port C bits are used for handshake signals before actual data transmission. It has interrupt handling capacity and input and output are latched.Example: A CPU wants to transfer data to a printer. In this case since speed of processor is very fast as compared to relatively slow printer, so before actual data transfer it will send handshake signals to the printer for synchronization of the speed of the CPU and the peripherals.

- **Mode 2 –** Bi-directional data bus mode. In this mode only port A works, and port B can work either in mode 0 or mode 1. 6 bits port C are used as handshake signals. It also has interrupt handling capacity.

Control Word Format



- 80 - (1000 0000, 1=I/o, 00=mode 0, 0 =PA o/p, 0=PCU o/p, 0= mode 0, 0= PB o/p, 0=PCL o/p)
- 90 - (1001 0000, I/o, mode 0,PA i/p, PB, PC o/p)
- 91 - (1001 0001, I/o, mode 0,PA i/p, PCl I/p)
- 92 (1001 0010, I/o, mode 0,PA , PB i/p)
- 85 (1000 0101 , I/o, mode 0, mode 0, PCL i/p)
- 55 (0101 0101, BSR, mode 2, PA i/p, PCU o/p, mode 1, PB o/p, PCL i/p)
- 82 (1000 0010, i/o, PB i/p rest are o/p)

- Addresses
- PA 1001
- PB 1002
- PC 1003
- CW 1004