NARSIMHA REDDY ENGINEERING COLLEGE
UGC AUTONOMOUS INSTITUTION
Maisammaguda (V), Kompally - 500100, Secunderabad, Telangana State, India

Accredited by NBA & NAAC with 'A' Grade
Approved by AICTE
Permanently affiliated to JNTUH

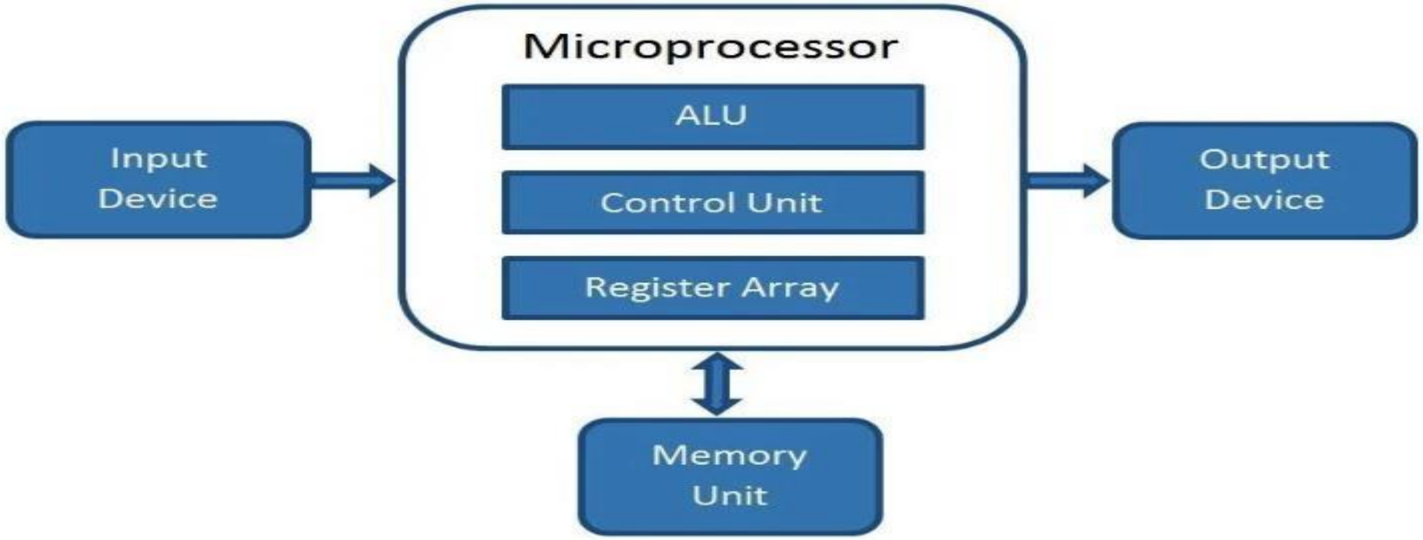
NRCM
your roots to success...

Microprocessor and Microcontrollers(EC3101PC)

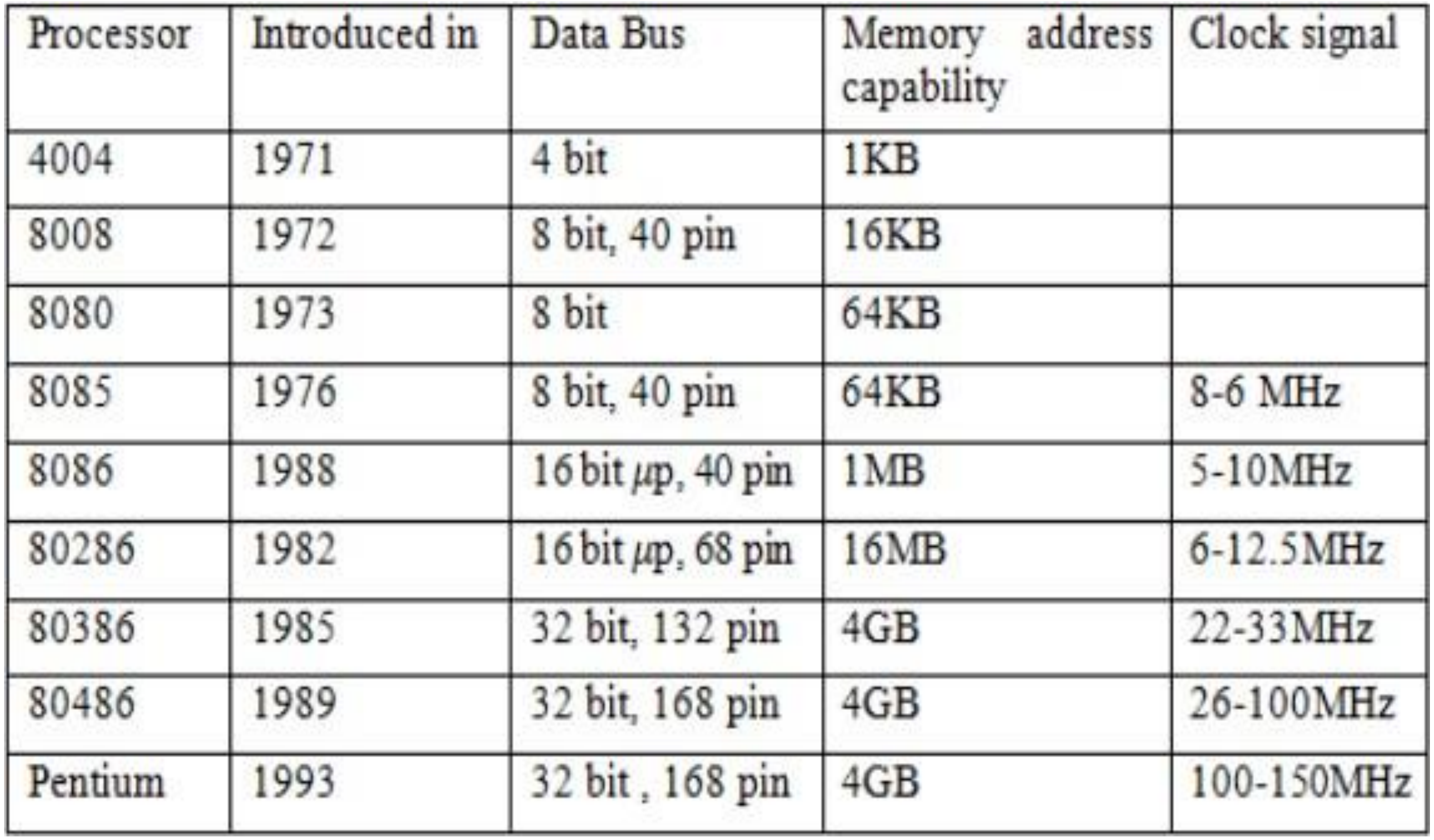
Prepared by
V. Nagalakshmi

UNIT -1
THE 8086 MICROPROCESSORS

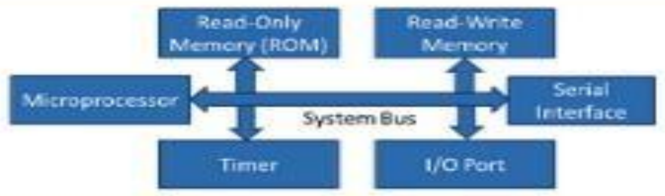
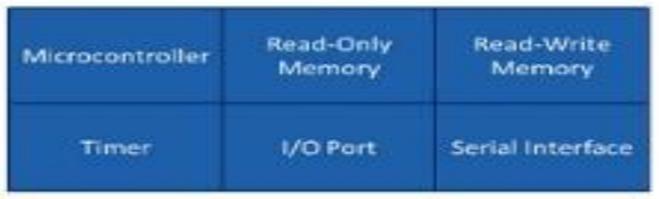
- A **microprocessor** is an electronic component that is used by a computer to do its work. It is a central processing unit on a single integrated circuit chip containing millions of very small components including transistors, resistors, and diodes that work together.



Evolution of Microprocessor



Processor	Introduced in	Data Bus	Memory address capability	Clock signal
4004	1971	4 bit	1KB	
8008	1972	8 bit, 40 pin	16KB	
8080	1973	8 bit	64KB	
8085	1976	8 bit, 40 pin	64KB	8-6 MHz
8086	1988	16 bit μp, 40 pin	1MB	5-10MHz
80286	1982	16 bit μp, 68 pin	16MB	6-12.5MHz
80386	1985	32 bit, 132 pin	4GB	22-33MHz
80486	1989	32 bit, 168 pin	4GB	26-100MHz
Pentium	1993	32 bit , 168 pin	4GB	100-150MHz

Microprocessor	Micro Controller
 Read-Only Memory (ROM) Read-Write Memory Microprocessor System Bus Serial Interface Timer I/O Port	 Microcontroller Read-Only Memory Read-Write Memory Timer I/O Port Serial Interface
Microprocessor is heart of Computer system.	Micro Controller is a heart of embedded system.
It is just a processor. Memory and I/O components have to be connected externally	Micro controller has external processor along with internal memory and i/O components
Since memory and I/O has to be connected externally, the circuit becomes large.	Since memory and I/O are present internally, the circuit is small.
Cannot be used in compact systems and hence inefficient	Can be used in compact systems and hence it is an efficient technique
Cost of the entire system increases	Cost of the entire system is low
Due to external components, the entire power consumption is high. Hence it is not suitable to used with devices running on stored power like batteries.	Since external components are low, total power consumption is less and can be used with devices running on stored power like batteries.
Most of the microprocessors do not have power saving features.	Most of the micro controllers have power saving modes like idle mode and power saving mode. This helps to reduce power consumption even further.
Since memory and I/O components are all external, each instruction will need external operation, hence it is relatively slower.	Since components are internal, most of the operations are internal instruction, hence speed is fast.
Microprocessor have less number of registers, hence more operations are memory based.	Micro controller have more number of registers, hence the programs are easier to write.
Microprocessors are based on von Neumann model/architecture where program and data are stored in same memory module	Micro controllers are based on Harvard architecture where program memory and Data memory are separate
Mainly used in personal computers	Used mainly in washing machine, MP3 players

UNIT 1
THE 8086 MICROPROCESSOR

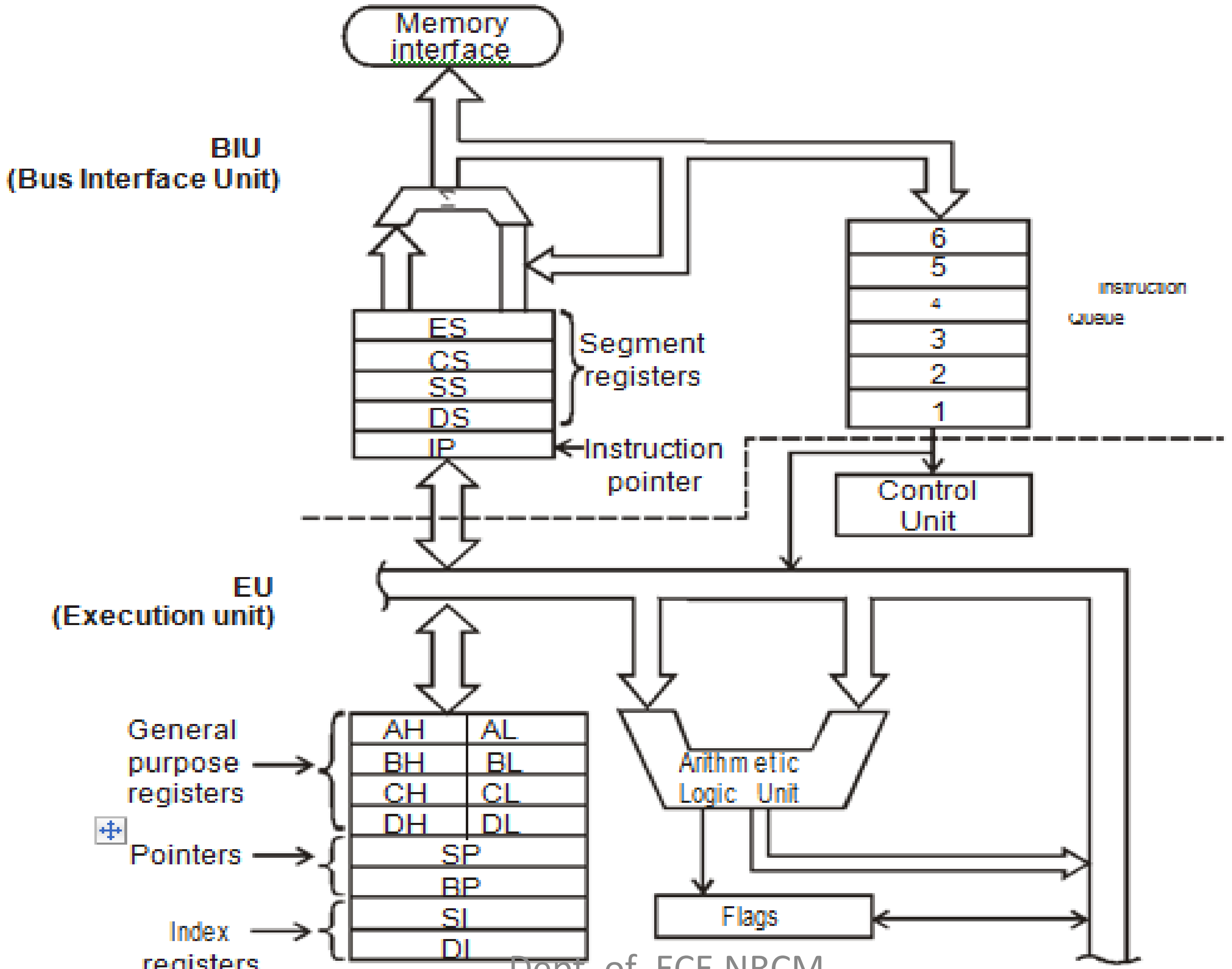
Introduction to 8086 – Microprocessor architecture – Addressing modes - Instruction set and assembler directives – Assembly language programming – Modular Programming - Linking and Relocation - Stacks - Procedures – Macros – Interrupts and interrupt service routines – Byte and String Manipulation.

UNIT 1
THE 8086 MICROPROCESSOR

<u>FEATURES OF 8086</u>

- The 8086 is a 16 bit processor.
- The 8086 has a 16 bit Data bus.
- The 8086 has a 20 bit Address bus.
- Direct addressing capability 1 M Byte of Memory (2^{20})
- It provides fourteen 16-bit register.
- 24 Operand addressing modes.
- Four general-purpose 16-bit registers: AX, BX, CX, DX
- Available in 40pin Plastic Package and Lead Chip.

8086 MICROPROCESSOR ARCHITECTURE



the 8086 processor are partitioned logically into two processing units

- **Bus Interface Unit (BIU)**

  The BIU fetches instructions, reads data from memory and ports, and writes data to memory and I/O ports.

- **Execution Unit (EU)**

  EU receives program instruction codes and data from the BIU, executes these instructions and stores the results either in the general registers or output them through the BIU. EU has no connections to the system buses.

The BIU contains

- Segment registers
- Instruction pointer
- Instruction queue

The EU contains

- ALU
- General purpose registers
- Index registers
- Pointers
- Flag register

General Purpose Registers

All general registers of the 8086 microprocessor can be used for arithmetic and logic operations.

- **Accumulator register (AX)**

  Accumulator can be used for I/O operations and string manipulation.
- **Base register (BX)**

  BX register usually contains a data pointer used for based, based indexed or register indirect addressing.
- **Count register (CX)**

  Count register can be used as a counter in string manipulation and shift/rotate instructions.
- **Data register (DX)**

  Data register can be used as a port number in I/O operations.

Segment Registers:

Most of the registers contain data/instruction offsets within 64 KB memory segment. There are four different 64 KB segments for instructions, stack, data and extra data.

- Code segment (CS)

  The CS register is automatically updated during FAR JUMP, FAR CALL and FAR RET instructions.

- Stack segment (SS)

  SS register can be changed directly using POP instruction.

- Data segment (DS)

  DS register can be changed directly using POP and LDS instructions.

- Extra segment (ES)

  ES register can be changed directly using POP and LES instructions.

Pointer Registers

Stack Pointer (SP)

It is a 16-bit register pointing to program stack.

Base Pointer (BP)

It is a 16-bit register pointing to data in the stack segment. BP register is usually used for based, based indexed or register indirect addressing.

Index Registers

Source Index (SI)

It is a 16-bit register. SI is used for indexed, based indexed and register indirect addressing, as well as a source data address in string manipulation instructions.

Destination Index (DI)

It is a 16-bit register. DI is used for indexed, based indexed and register indirect addressing, as well as a destination data address in string manipulation instructions.

Instruction Pointer (IP)

It is a 16-bit register. The operation is same as the program counter. The IP register is updated by the BIU to point to the address of the next instruction. Programs do not have direct access to the IP, but during execution of a program the IP can be modified or saved and restored from the stack.

Flag register

It is a 16-bit register containing nine 1-bit flags:

- Six status or condition flags (OF, SF, ZF, AF, PF, CF)
- Three control flags (TF, DF, IF)

- **Overflow Flag** (OF) - set if the result is too large positive number, or is too small negative number to fit into destination operand.
- **Sign Flag** (SF) - set if the most significant bit of the result is set.
- **Zero Flag** (ZF) - set if the result is zero.
- **Auxiliary carry Flag** (AF) - set if there was a carry from or borrow to bits 0-3 in the AL register.
- **Parity Flag** (PF) - set if parity (the number of “1” bits) in the low-order byte of the result is even.
- **Carry Flag** (CF) - set if there was a carry from or borrow to the most significant bit during last result calculation.
- **Trap or Single-step Flag** (TF) - if set then single-step interrupt will occur after the next instruction.
- **Direction Flag** (DF) - if set then string manipulation instructions will auto-decrement index registers. If cleared then the index registers will be auto-incremented.
- **Interrupt-enable Flag** (IF) - setting this bit enables maskable interrupts.

AH	AL	Accumulator (AX)
BH	BL	Base (BX)
CH	CL	Count (CX)
DH	DL	Data (DX)

SP	Stack Pointer
BP	Base Pointer
SI	Source Index
DI	Destination Index

CS	Code Segment
DS	Data Segment
SS	Stack Segment
ES	Extra Segment

IP	Instruction Pointer

OF	DF	IF	TF	SF	ZF	AF	PF	CF	Flags

Instruction Queue

The instruction queue is a First-In-First-out (FIFO) group of registers where 6 bytes of instruction code is pre-fetched from memory ahead of time. It is being done to speed-up program execution by overlapping instruction fetch and execution. This mechanism is known as **PIPELINING.**

ALU

It is a 16 bit register. It can add, subtract, increment, decrement, complement, shift numbers and performs AND, OR, XOR operations.

Control unit

The control unit in the EU directs the internal operations like RD , WR , M/IO

Instruction Set

- Data moving instructions.
- Arithmetic instructions - add, subtract, increment, decrement, convert byte/word and compare.
- Logic instructions - AND, OR, exclusive OR, shift/rotate and test.
- String manipulation instructions - load, store, move, compare and scan for byte/ word.
- Control transfer instructions - conditional, unconditional, call subroutine and return from subroutine.
- Input/Output instructions.
- Other instructions - setting/clearing flag bits, stack operations, software interrupts, etc.

Addressing modes

- **Implied** - the data value/data address is implicitly associated with the instruction.
- **Register** - references the data in a register or in a register pair.
- **Immediate** - the data is provided in the instruction.
- **Direct** - the instruction operand specifies the memory address where data is located.
- **Register indirect** - instruction specifies a register containing an address, where data is located. This addressing mode works with SI, DI, BX and BP registers.
- **Based** - 8-bit or 16-bit instruction operand is added to the contents of a base register (BX or BP), the resulting value is a pointer to location where data resides.
- **Indexed** - 8-bit or 16-bit instruction operand is added to the contents of an index register (SI or DI), the resulting value is a pointer to location where data resides.
- **Based Indexed** - the contents of a base register (BX or BP) is added to the contents of an index register (SI or DI), the resulting value is a pointer to location where data resides.
- **Based Indexed with displacement** - 8-bit or 16-bit instruction operand is added to the contents of a base register (BX or BP) and index register (SI or DI), the resulting value is a pointer to location where data resides.

Interrupts

Hardware interrupts

Maskable and non-maskable interrupts

Software interrupts

ADDRESSING MODES

- An addressing mode is the way the 8086 identifies the operands for the instruction. All instructions that access the data use one or more of the addressing modes.
- The memory address of an operand consists of two components

  1.Starting address of the memory segment

  2.Offset
- When an operand is stored in a memory location, how for the operand's memory location is within a memory segment from the starting address of the segment, is called **Offset** or **Effective Address** (EA).
- The 8086 uses 20 bit memory address. The segment register gives 16 MSBs of the starting address of the memory segment. The BIU generates 20 bit starting address of the memory segment by shifting the content of the segment register left by 4 bits. In other words it puts 4 zeros in 4 LSB positions.
- **Memory Address = Starting address of the memory segment + Offset**

The 8086 has the following addressing modes:

- Register Addressing Mode
- Immediate Addressing Mode
- Direct Addressing Mode
- Register Indirect Addressing Mode
- Base Addressing Mode
- Indexed Addressing Mode
- Based Indexed Addressing Mode
- String Addressing Mode
- I/O Port Addressing Mode
- Relative Addressing Mode
- Implied Addressing Mode

Register Addressing Mode

- Both source and destination operands are registers. The operand sizes must match. MOV *destination, source*
- ***Examples:***
- MOV AL, AH
- MOV AX, BX

Immediate Addressing Mode

- The data operand is supplied as part of the instruction. The immediate operand can only be a source.
- ***Examples:***
- MOV CH, 3A H
- MOV DX, 0C1A5 H

Direct Addressing Mode

- One of the operands is a memory location, given by a constant offset.
- In this mode the 16 bit effective address (EA) is taken directly from the displacement field of the instruction.
- ***Examples:***
- MOV AX,[1234 H]
- MOV DL, [3BD2 H],

Register Indirect Addressing Mode

- One of the operands is a memory location, with the offset given by one of the BP, BX, SI, or DI registers.
- ***Example:***
- MOV [BX], CL
- MOV DL, [BX]

Base Addressing Mode

- In this mode EA is obtained by adding a displacement (signed 8 bit or unsigned 16 bit) value to the contents of BX or BP. The segment registers used are DS and SS.
- ***Example:***
- MOV AX, [BP + 200]

Indexed Addressing Mode

- The operand’s offset is the sum of the content of an index register SI or DI and an 8-bit or 16-bit displacement.
- ***Example:***
- MOV AH, [DI]

Based Indexed Addressing Mode

- In this mode, the EA is computed by adding a base register (BX or BP), an index register (SI or DI) and a displacement (unsigned 16 bit or sign extended 8 bit)
- ***Example:***
- MOV AX, [BX + SI + 1234 H]
- MOV CX, [BP][SI] + 4

String Addressing Mode

- The instruction is a string instruction, which uses index registers implicitly to access memory.
- ***Example:***
- MOVS B
- MOVS W

I/O Port Addressing Mode

- The destination or source of the data is an I/O port. Either direct port addressing (including an 8-bit port address) or indirect addressing (DX must contain the port address) may be used.
- ***Examples:***
- IN AX, 50H ; Direct
- OUT DX, AL ; Indirect

Relative Addressing Mode

- In this mode, the operand is specified as a signed 8 bit displacement, relative to PC(Program Counter).
- ***Examples:***
- JMP 0200 H
- JNC START

Implied Addressing M~~od~~e

- Instructions using this mode have no operands.
- ***Examples:***
- CLC, STC, CMC

INSTRUCTION SET

- Intel 8086 has approximately 117 instructions. These instructions are used to transfer data between registers, register to memory, memory to register or register to I/O ports and other instructions are used for data manipulation.
- But in Intel 8086 operations between memory to memory is not permitted. These instructions are classified in to six-groups as follows.

    1.Data Transfer Instructions

    2.Arithmetic Instructions

    3.Bit Manipulation Instructions

    4.String Instructions

    5.Program Execution Transfer Instructions

    6.Processor Control Instructions

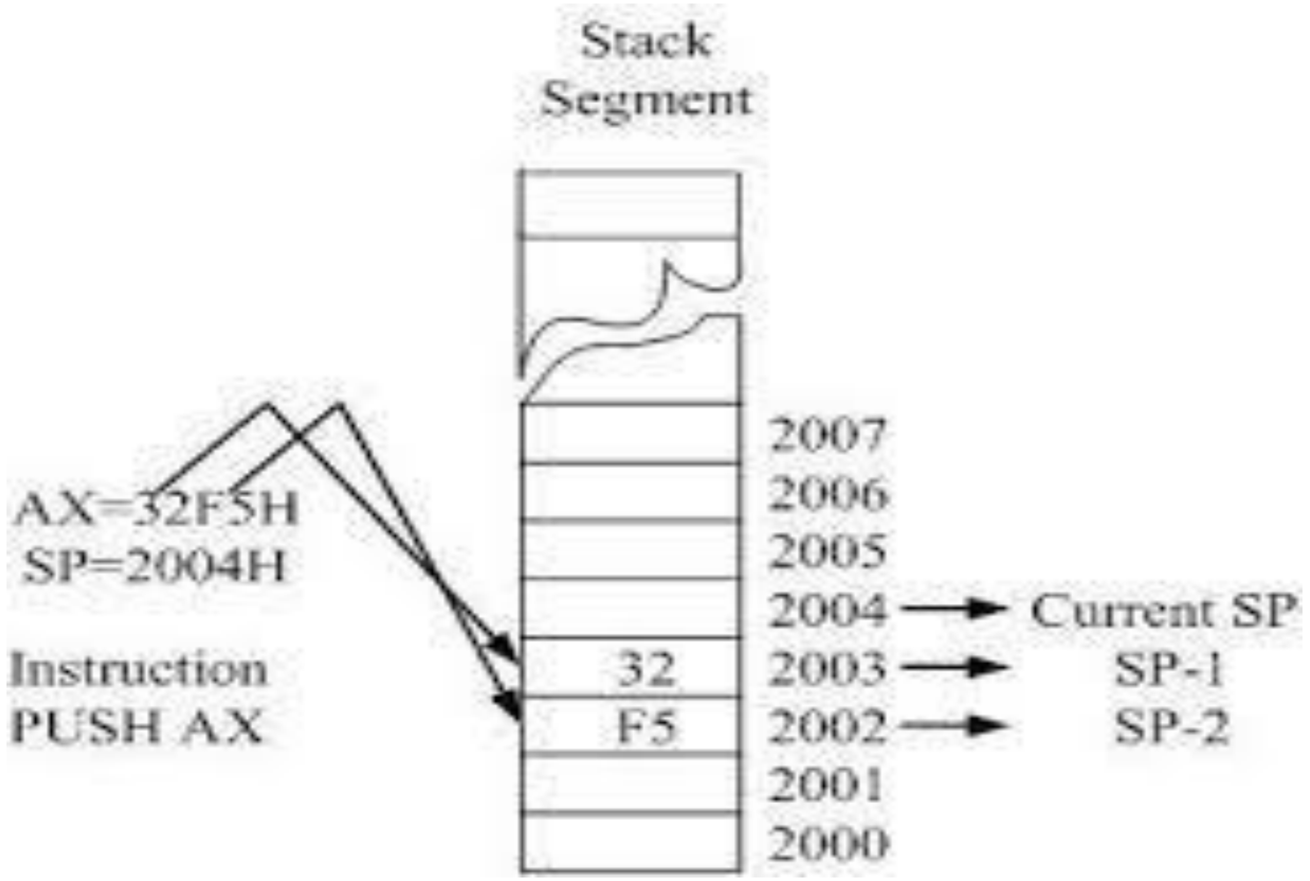
Data Transfer Instructions

1.MOV

- *MOV destination, source*
- This **(Move)** instruction transfers a byte or a word from the source operand to the destination operand.
- (DEST) $\leftarrow$ (SRC)
- DEST = Destination
- SRC = Source
- ***Example :***
- MOV AX, BX
- MOV AX, 2150H
- MOV AL, [1135]

2.PUSH

- *PUSH Source*
- This instruction decrements SP (stack pointer) by 2 and then transfers a word from the source operand to the top of the stack now pointed to by stack pointer.
- $(SP) \leftarrow (SP) - 2$
- $((SP)+1 : (SP)) \leftarrow (SRC)$
- ***Example :***
- PUSH SI
- PUSH BX

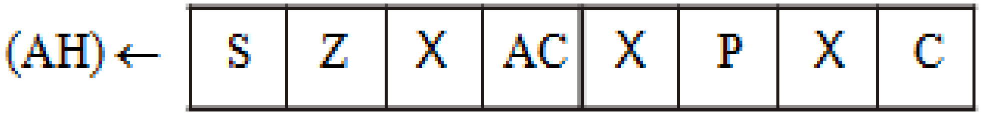

Stack
Segment
AX=32F5H
SP=2004H
Instruction
PUSH AX
32
F5
2007
2006
2005
2004
2003
2002
2001
2000
Current SP
SP-1
SP-2

3.POP

- *POP destination*
- This instruction transfers the word at the current top of stack (pointed to by SP) to the destination operand and then increments SP by 2, pointing to the new top of the stack.
- $(DEST) \leftarrow ((SP)+1:(SP))$
- $(SP) \leftarrow (SP) + 2$
- ***Example :***
- POP DX
- POP DS

LAHF

- *Load Register AH from Flags*
- This instruction copies Sign flag(S), Zero flag (Z), Auxiliary flag (AC), Parity flag (P) and Carry flag (C) of 8086 into bits 7, 6, 4, 2 and 0 respectively, of register AH

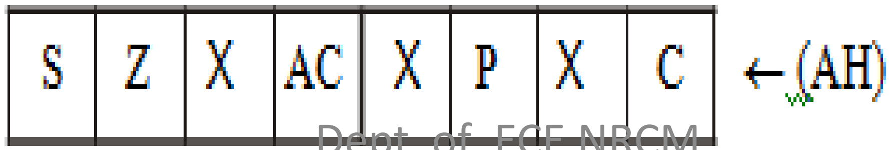


(AH) ←

S	Z	X	AC	X	P	X	C

SAHF

- *Store Register AH into Flags*
- This instruction transfers bits 7, 6, 4, 2 and 0 from register AH into S, Z, AC, P and C flags respectively, thereby replacing the previous values.



S	Z	X	AC	X	P	X	C

← (AH)

XCHG

- *XCHG destination, source*
- This **(Exchange)** instruction switches the contents of the source and destination operands.

$(Temp) \leftarrow (DEST)$

$(DEST) \leftarrow (SRC)$

$(SRC) \leftarrow (Temp)$

Example :

XCHG AX, BX

XCHG BL, AL

XLAT

- *XLAT table*
- This **(Translate)** instruction replaces a byte in the AL register with a byte from a 256-byte, user-coded translation table. XLAT is useful for translating characters from one code to another.
- $AL \leftarrow ((BX) + (AL))$
- ***Example :***
- XLAT ASCII_TAB
- XLAT Table_3

LEA

- *LEA destination, source*
- This **(Load Effective Address)** instruction transfers the offset of the source operand (memory) to the destination operand (16-bit general register).
- (REG)$\leftarrow$ EA
- **Example :**
- LEA BX, [BP] [DI]
- LEA SI, [BX + 02AF H]

LDS

- *LDS destination, source*
- This **(Load pointer using DS)** instruction transfers a 32-bit pointer variable from the source operand (memory operand) to the destination operand and register DS.
- (REG)$\leftarrow$ (EA)
- (DS)$\leftarrow$ (EA+2)
- ***Example :***
- LDS SI, [6AC1H]

LES

- *LES destination, source*
- This **(Load pointer using ES)** instruction transfers a 32-bit pointer variable from the source operand (memory operand) to the destination operand and register ES.
- (REG) $\leftarrow$ (EA)
- (ES) $\leftarrow$ (EA+2)
- ***Example :***
- LES DI, [BX]

IN

- *IN accumulator, port*
- This **(Input)** instruction transfers a byte or a word from an input port to the accumulator (AL or AX).
- (DEST) $\leftarrow$ (SRC)
- ***Example :***
- IN AX, DX
- IN AL, 062H

OUT

- *OUT port, accumulator*
- This **(Output)** instruction transfers a byte or a word from the accumulator (AL or AX) to an output port.
- (DEST) $\leftarrow$ (SRC)
- ***Example :***
- OUT DX, AL
- OUT 31, AX

Arithmetic Instructions

ADD

- ADD destination, source
- This **(Add)** instruction adds the two operands (byte or word) and stores the result in destination operand.
- (DEST) $\leftarrow$ (DEST) + (SRC)
- ***Example :***
- ADD CX, DX
- ADD AX, 1257 H
- ADD BX, [CX]

ADC

- *ADC destination, source*
- This **(Add with carry)** instruction adds the two operands and adds one if carry flag (CF) is set and stores the result in destination operand.
- (DEST) $\leftarrow$ (DEST) + (SRC) + 1
- ***Example :***
- ADC AX, BX
- ADC AL, 8
- ADC CX, [BX]

SUB

- *SUB destination, source*
- This **(Subtract)** instruction subtracts the source operand from the destination operand and the result is stored in destination operand.
- (DEST) $\leftarrow$ (DEST) – (SRC)
- ***Example :***
- SUB AX, 6541 H
- SUB BX, AX
- SUB SI, 5780 H

SBB

- ***SBB destination, source***
- This **(Subtract with Borrow)** instruction subtracts the source from the destination and subtracts 1 if carry flag (CF) is set. The result is stored in destination operand.
- (DEST) $\leftarrow$ (DEST) – (SRC) –1
- ***Example :***
- SBB BX, CX
- SBB AX, 2

CMP

- ***CMP destination, source***
- This **(Compare)** instruction subtracts the source from the destination, but does not store the result.
- (DEST) – (SRC)
- ***Example :***
- CMP AX, 18
- CMP BX, CX

INC

- *INC destination*
- This **(Increment)** instruction adds 1 to the destination operand (byte or word).
- (DEST) ← (DEST) + 1
- ***Example :***
- INC BL
- INC CX

DEC

- *DEC destination*
- This **(Decrement)** instruction subtracts 1 from the destination operand. (DEST) ← (DEST) –1
- ***Example :***
- DEC BL
- DEC AX

NEG

- *NEG destination*
- This **(Negate)** instruction subtracts the destination operand from 0 and stores the result in destination. This forms the 2's complement of the number.
- (DEST) ← 0 – (DEST)
- ***Example :***
- NEG AX
- NEG CL

DAA

- This **(Decimal Adjust for Addition)** instruction converts the binary result of an ADD or ADC instruction in AL to packed BCD format.

DAS

- This **(Decimal Adjust for Subtraction)** instruction converts the binary result of a SUB or SBB instruction in AL to packed BCD format.

AAA

- This **(ASCII Adjust for Addition)** instruction adjusts the binary result of ADD or ADC instruction.
- If bits 0-3 of AL contain a value greater than 9, or if the auxiliary carry flag (AF) is set, the CPU adds 06 to AL and adds 1 to AH. The bits 4-7 of AL are set to zero.
- $(AL) \leftarrow (AL) + 6$
- $(AH) \leftarrow (AH) + 1$
- $(AF) \leftarrow 1$

AAS

- This **(ASCII Adjust for Subtraction)** instruction adjusts the binary result of a SUB or SBB instruction.
- If D_3–D_0 of AL > 9,
- (AL) $\leftarrow$ (AL) – 6
- (AH) $\leftarrow$ (AH) – 1
- (AF) $\leftarrow$ 1

MUL

- *MUL source*
- This **(Multiply)** instruction multiply AL or AX register by register or memory location contents. Both operands are unsigned numbers. If the source is a byte (8 bit), then it is multiplied by register AL and the result is stored in AH and AL.
- If the source operand is a word (16 bit), then it is multiplied by register AX and the result is stored in AX and DX registers.
- If 8 bit data, (AX) $\leftarrow$ (AL) x (SRC)
  If 16 bit data, (AX), (DX) $\leftarrow$ (AX) x (SRC)
  Example :
- MUL 25
- MUL CX

- **IMUL**
- *IMUL Source*
- This **(Integer Multiply)** instruction performs a signed multiplication of the source operand and the accumulator.
- If 8 bit data, (AX)← (AL) x (SRC)
- If 16 bit data, (AX), (DX) ←(AX) x (SRC)
- ***Example :***
- IMUL 250
- IMUL BL

AAM

- This **(ASCII Adjust for Multiplication)** instruction adjusts the binary result of a MUL instruction. AL is divided by 10(0AH) and quotient is stored in AH. The remainder is stored in AL.
- (AH)← (AL/0AH)
- (AL) ←Remainder

DIV

- *DIV Source*
- This **(Division)** instruction performs an unsigned division of the accumulator by the source operand. It allows a 16 bit unsigned number to be divided by an 8 bit unsigned number, or a 32 bit unsigned number to be divided by a 16 bit unsigned number.
- For 8 bit data, AX / source

  (AL) ← Quotient

  (AH) ← Remainder
- For 16 bit data, AX, DX / Source

  (AX) ← Quotient

  (DX) ← Remainder
- ***Example :***
- DIV CX
- DIV 321

IDIV

- *IDIV source*
- This **(Integer Division)** instruction performs a signed division of the accumulator by the source operand.
- For 8 bit data, AX / Source

  $(AL) \leftarrow$ Quotient

  $(AH) \leftarrow$ Remainder
- For 16 bit data, AX, DX / Source

  $(AX) \leftarrow$ Quotient

  $(DX) \leftarrow$ Remainder
- ***Example :***
- IDIV CL
- IDIV AX

AAD

- This **(ASCII Adjust for Division)** instruction adjusts the unpacked BCD dividend in AX before a division operation. AH is multiplied by 10(0AH) and added to AL. AH is set to zero.
- $(AL) \leftarrow (AH \times 0AH) + (AL)$
- $(AH) \leftarrow 0$

CBW

- This **(Convert Byte to Word)** instruction converts a byte to a word. It extends the sign of the byte in register AL through register AH. This instruction can be used for 16 bit IMUL or IDIV instruction.
- IF AL < 80 H, then AH = 00 H
- IF AL > 80 H, then AH = FFH

CWD :

- This **(Convert Word to Double word)** instruction converts a word to a double word.
- It extends the sign of the word in register AX through register DX.
- If AX < 8000 H, then DX = 0000 H
- If AX > 8000 H, then DX = FFFFH

Bit Manipulation Instructions

(i)Logical Instructions: AND, OR, XOR, NOT, TEST

(ii)Shift Instructions: SHL, SAL, SHR, SAR

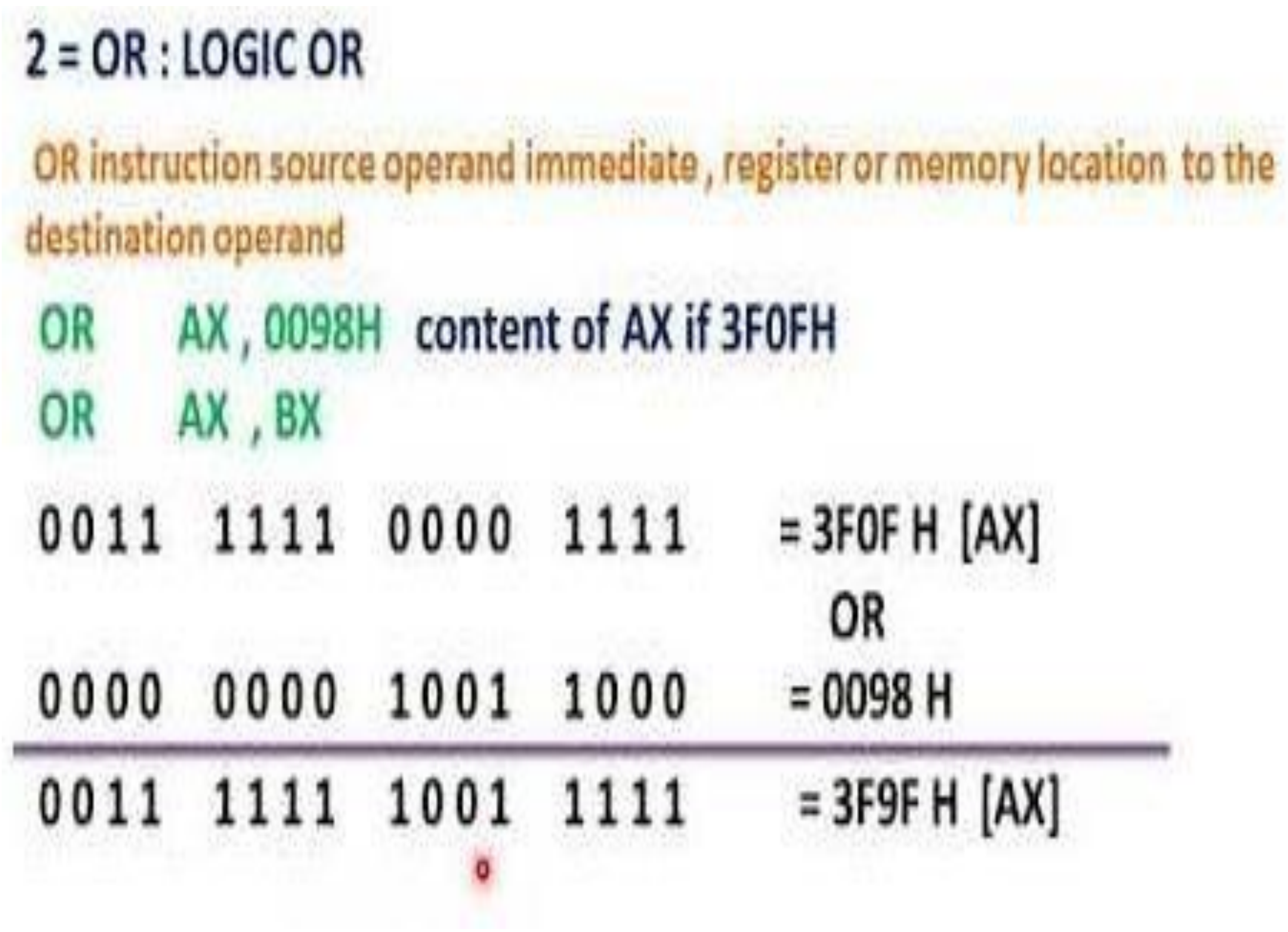
(iii)Rotate Instructions: ROL, ROR, RCL, RCR

AND

- *AND destination, source*
- This **(AND)** instruction performs the logical “AND” of the source operand with the destination operand and the result is stored in destination.
- (DEST) ¬ (DEST) “AND” (SRC)
- ***Example :***
- AND BL, CL
- AND AL, 0011 1100 B

OR

- *OR destination, source*
- This **(OR)** instruction performs the logical “OR” of the source operand with the destination operand and the result is stored in destination.
- (DEST) ¬ (DEST) “OR” (SRC)
- ***Example :***
- OR AX, BX
- OR AL, 0000 1111B



2 = OR : LOGIC OR

OR instruction source operand immediate , register or memory location to the destination operand

```
OR     AX , 0098H   content of AX if 3F0FH
OR     AX  , BX
```

```
0011   1111   0000   1111       = 3F0F H  [AX]
                                    OR
0000   0000   1001   1000       = 0098 H
-----------------------------------------------
0011   1111   1001   1111       = 3F9F H  [AX]
```

XOR

- *XOR destination, source*
- This **(Exclusive OR)** instruction performs the logical “XOR” of the two operands and the result is stored in destination operand.
- (DEST) ¬ (DEST) “XOR” (SRC)
- ***Example :***
- XOR BX, AX
- XOR AL, 1111 1111B

NOT

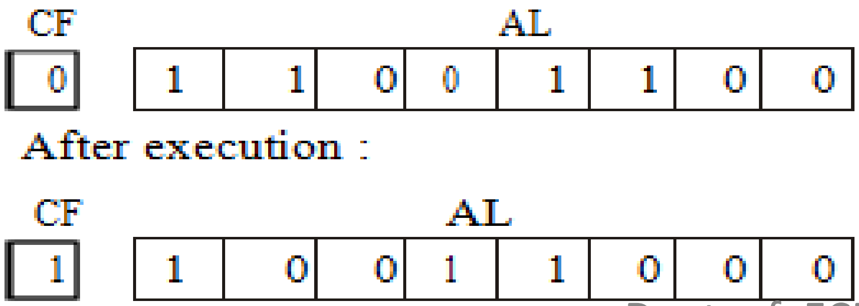
- *NOT destination*
- This **(NOT)** instruction inverts the bits (forms the 1’s complement) of the byte or word.
- (DEST) ¬ 1’s complement of (DEST)
- ***Example :***
- NOT AX

TEST

- *TEST destination, source*
- This **(TEST)** instruction performs the logical “AND” of the two operands and updates the flags but does not store the result.
- (DEST) “AND” (SRC)
- ***Example :***
- TEST AL, 15 H
- TEST SI, DI

SHL

- *SHL destination, count*
- This **(Shift Logical Left)** instruction performs the shift operation. The number of bits to be shifted is represented by a variable count, either 1 or the number contained in the CL register.
- ***Example***
- SHL AL, 1
- Before execution :

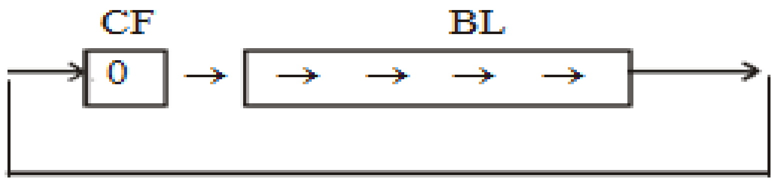


SAL

- *SAL destination, count*
- SAL **(Shift Arithmetric Left)** and SHL (Shift Logical Left) instructions perform the same operation and are physically the same instruction.
- ***Example***
- SAL AL, CL
- SAL AL, 1

SHR

- *SHR destination, count*
- This **(Shift Logical Right)** instruction shifts the bits in the destination operand to the right by the number of bits specified by the count operend, either 1 or the number contained in the CL register.
- ***Example***
- SHR BL, 1
- SHR BL, CL

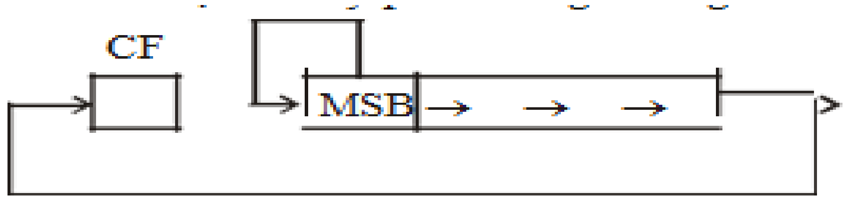


The SHR instruction may be used to divide a number by 2. For example, we can divide 32 by 2,

```
MOV BL, 32    ;  0010  0000  (32)
SHR BL, 1     ;  0001  0000  (16)
SHR BL, 1     ;  0000  1000  (8)
SHR BL, 1     ;  0000  0100  (4)
SHR BL, 1     ;  0000  0010  (2)
```

SAR

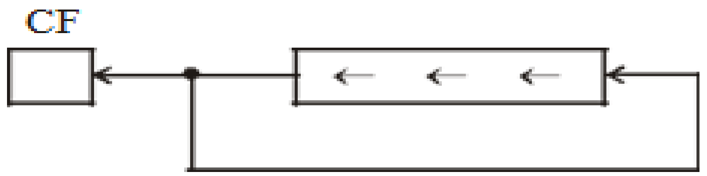
- *SAR destination, count*
- This **(Shift Arithmetic Right)** instruction shifts the bits in the destination operand to the right by the number of bits specified in the count operand. Bits equal to the original high-order (sign) bits are shifted in on the left, thereby preserving the sign of the original value.



ROL

ROL destination, count

This **(Rotate Left)** instruction rotates the bits in the byte/word destination operand to the left by the number of bits specified in the count operand.



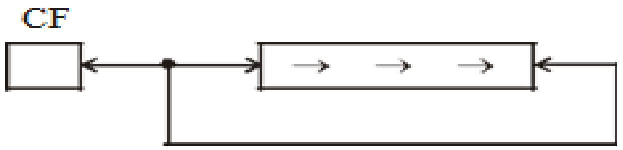
Example :

ROL AL, 1

	CF	AL							
Before execution :	0	1	1	0	0	1	1	0	0
After execution :	1	1	0	0	1	1	0	0	1

ROR

- *ROR destination, count*
- This **(Rotate Right)** instruction rotates the bits in the byte/word destination operand to the right by the number of bits specified in the count operand.



Example :

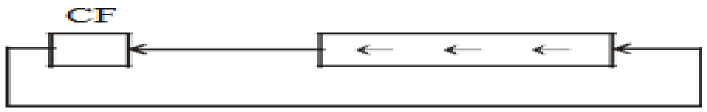
ROR AL, 1

	CF	AL							
Before execution :	0	1	1	0	0	1	1	0	0
After execution :	0	0	1	1	0	0	1	1	0

RCL

RCL destination, count

This **(Rotate through Carry Left)** instruction rotates the contents left through carry by the specified number of bits in count operand.



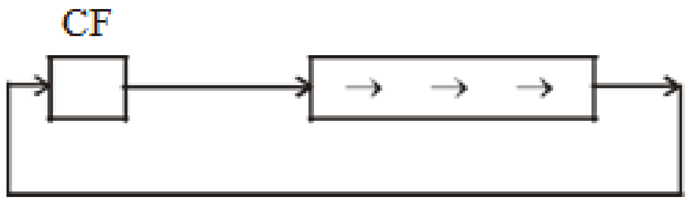
Example :

RCL AL, 1

	CF	AL							
Before execution :	1	0	0	0	0	1	1	1	1
After execution :	0	0	0	0	1	1	1	1	1

RCR

- *RCR destination, count*
- This **(Rotate through Carry Right)** instruction rotates the contents right through carry by the specified number of bits in the count operand.



Example :

RCR AL, 1

	CF	AL							
Before execution :	1	1	1	0	0	0	0	1	0
After execution :	0	1	1	1	0	0	0	0	1

STRING INSTRUCTIONS

REP

- *REP MOVS destination, Source*
- This **(Repeat)** instruction converts any string primitive instruction into a re-executing loop. It specifies a termination condition which causes the string primitive instruction to continue executing until the termination condition is met.
- ***Example :***
- REP MOVS CL, AL
- The other Repeat instructions are :
- REPE - Repeat while Equal
- REPZ - Repeat while zero
- REPNE - Repeat while Not Equal
- REPNZ - Repeat while Not Zero
- The above instructions are used with the CMPS and SCAS instructions.

MOVS

- *MOVS destination - string, source-string*
- This **(Move String)** instruction transfers a byte/word from the source string (addressed by SI) to the destination string (addressed by DI) and updates SI and DI to point to the next string element.
- (DEST) $\leftarrow$ (SRC)
- ***Example :***
- MOVS Buffer 1, Buffer 2

CMPS

- *CMPS destination-string, source-string*
- This **(Compare String)** instruction subtracts the destination byte/word (addressed by DI) from the source byte/word (addressed by SI). It affects the flags but does not affect the operands.
- ***Example :***
- CMPS Buffer 1, Buffer 2

SCAS

- *SCAS destination-string*
- This **(Scan String)** instruction subtracts the destination string element (addressed by DI) from the contents of AL or AX and updates the flags.
- **Example :**
- SCAS Buffer

LODS

- *LODS source-string*
- This **(Load String)** instruction transfers the byte/word string element addressed by SI to register AL or AX and updates SI to point to the next element in the string.
- (DEST) $\leftarrow$ (SRC)
- ***Example :***
- LODSB name
- LODSW name

STOS

- *STOS destination - string*
- This **(Store String)** instruction transfers a byte/word from register AL or AX to the string element addressed by DI and updates DI to point to the next location in the string.
- (DEST) $\leftarrow$ (SRC)
- ***Example :***
- STOS display

Program Transfer Instructions

- (i)Unconditional instructions: CALL, RET, JMP
- (ii)Conditional instructions: JC, JZ, JA.....
- (iii)Iteration control instructions :LOOP, JCXZ
- (iv)Interrupt instructions: INT, INTO, IRET

CALL

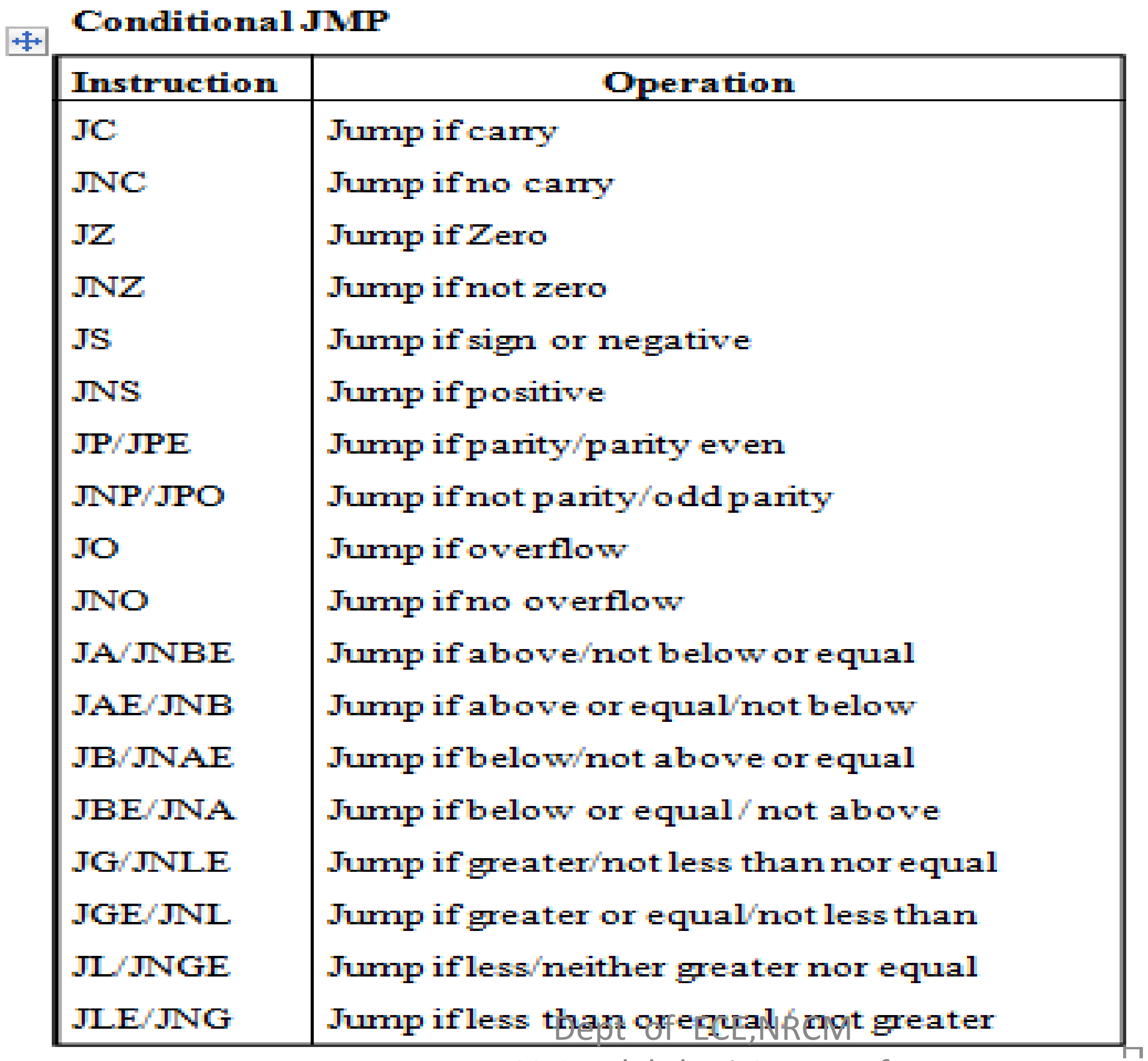
- *CALL procedure - name*
- This **(CALL)** instruction is used to transfer execution to a subprogram or procedure. RET (return) instruction is used to go back to the main program. There are two basic types of CALL : NEAR and FAR
- ***Example :***
- CALL NEAR
- CALL AX

RET

- This **(Return)** instruction will return execution from a procedure to the next instruction after the CALL instruction in the main program.
- ***Example :***
- RET
- RET 6

JMP

- *JMP target*
- This **(Jump)** instruction unconditionally transfers control to the target location. The target operand may be obtained from the instruction itself (direct JMP) or from memory or a register referenced by the instruction (indirect JMP).
- ***Example :***
- JMP BX



Conditional JMP

Instruction	Operation
JC	Jump if carry
JNC	Jump if no carry
JZ	Jump if Zero
JNZ	Jump if not zero
JS	Jump if sign or negative
JNS	Jump if positive
JP/JPE	Jump if parity/parity even
JNP/JPO	Jump if not parity/odd parity
JO	Jump if overflow
JNO	Jump if no overflow
JA/JNBE	Jump if above/not below or equal
JAE/JNB	Jump if above or equal/not below
JB/JNAE	Jump if below/not above or equal
JBE/JNA	Jump if below or equal / not above
JG/JNLE	Jump if greater/not less than nor equal
JGE/JNL	Jump if greater or equal/not less than
JL/JNGE	Jump if less/neither greater nor equal
JLE/JNG	Jump if less than or equal / not greater

LOOP

- *LOOP label*
- This **(Loop if CX not zero)** instruction decrements CX by 1 and transfers control to the target operand if CX is not zero. Otherwise the instruction following LOOP is executed.
- If CX≠0, CX = CX–1
- IP = IP+displacement
- If CX=0, then the next sequential instruction is executed.
- ***Example :***
- LOOP again

Processor Control Instructions

HLT

- This **(Halt)** instruction will cause the 8086 to stop fetching and executing instructions. The 8086 will enter a halt state.

WAIT

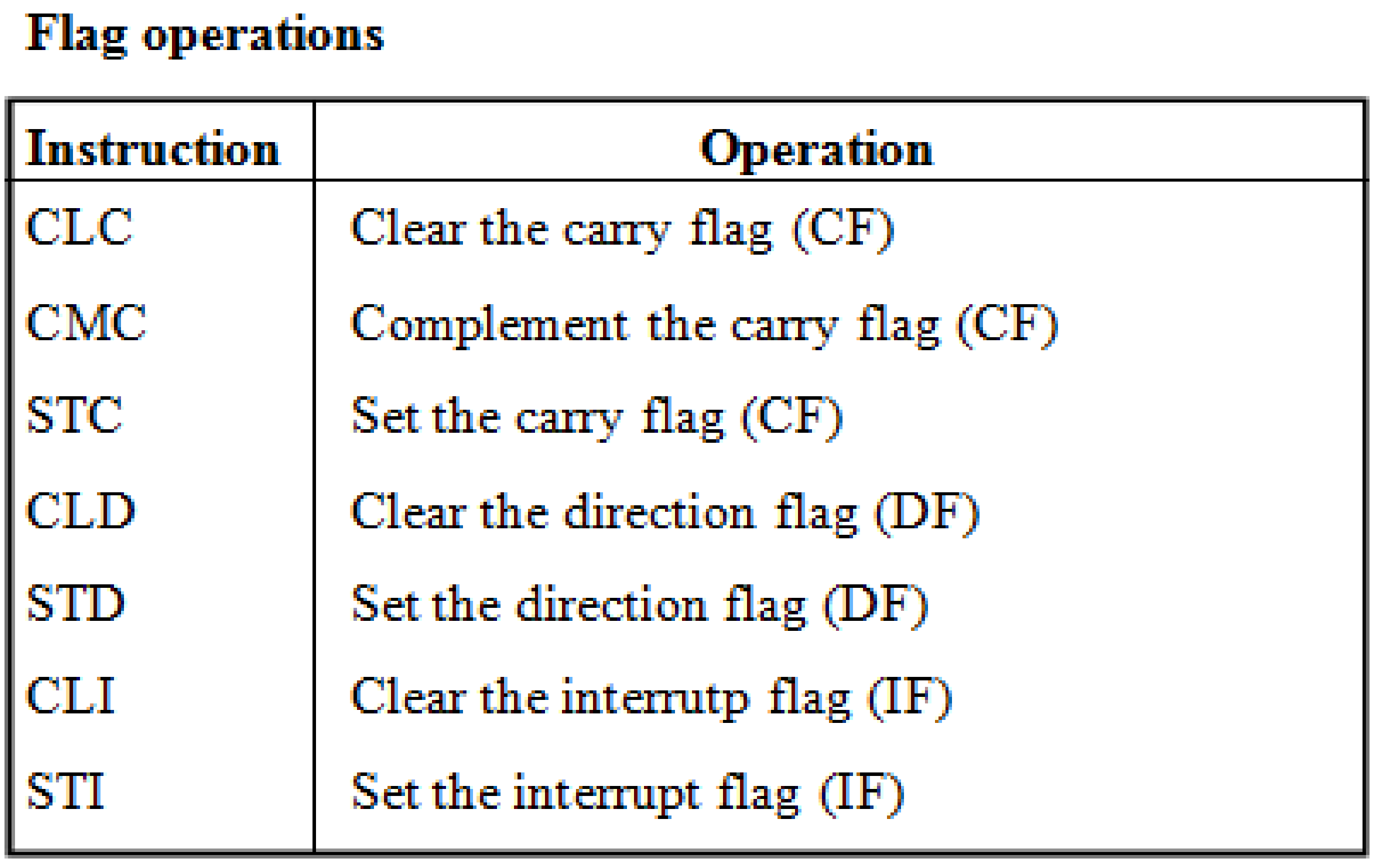
- This **(Wait)** instruction causes the 8086 to enter the wait state while its test line is not active.

ESC

- This **(Escape)** instruction provides a mechanism by which other coprocessors may receive their instructions from the 8086 instruction stream and make use of the 8086 addressing modes. The 8086 does a no operation (NOP) for the ESC instruction other than to access a memory operand and place it on the bus.

NOP

- This **(No operation)** instruction causes the CPU to do nothing. NOP does not affect any flags.



Flag operations

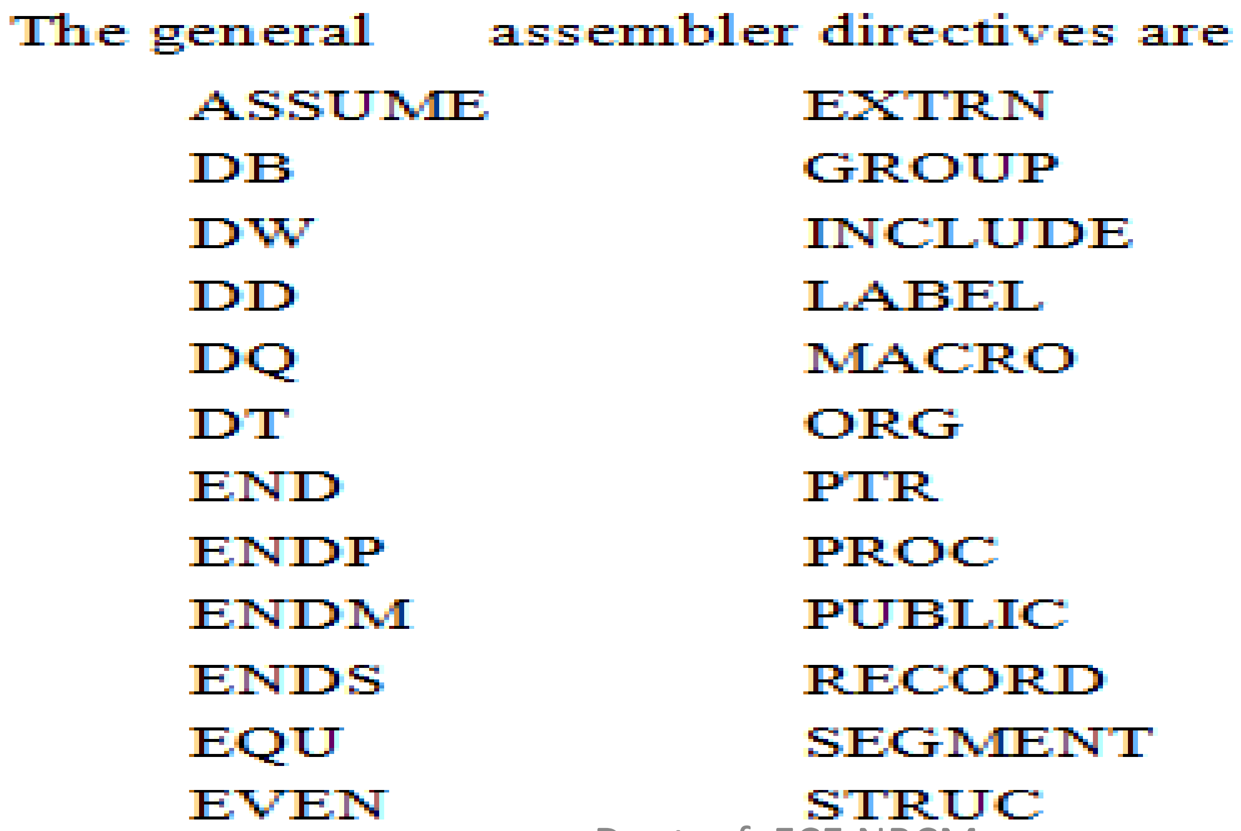
Instruction	Operation
CLC	Clear the carry flag (CF)
CMC	Complement the carry flag (CF)
STC	Set the carry flag (CF)
CLD	Clear the direction flag (DF)
STD	Set the direction flag (DF)
CLI	Clear the interrutp flag (IF)
STI	Set the interrupt flag (IF)

ASSEMBLER DIRECTIVES

- An assembler is a program which translates an assembly language program into machine language program.
- An assembler directive is a statement to give direction to the assembler to perform the task of assembly process.
- The assembler directives control organization of the program and provide necessary information to the assembler to understand assembly language programs to generate machine codes.
- An assembler supports directives to define data, to organize segments, to control procedures, to define macros etc.
- An assembly language program consists of two types of statements: Instructions and Directives.

Some assembler directives are,

- Borland Turbo Assembler (TASM)
- IBM Macro Assembler (MASM)
- Intel 8086 Macro Assembler (ASM)
- Microsoft Macro Assembler



The general assembler directives are

ASSUME	EXTRN
DB	GROUP
DW	INCLUDE
DD	LABEL
DQ	MACRO
DT	ORG
END	PTR
ENDP	PROC
ENDM	PUBLIC
ENDS	RECORD
EQU	SEGMENT
EVEN	STRUC

ASSUME

- The ASSUME directive enables error-checking for register values.
- It is used to inform the assembler the names of the logical segments, which are to be assigned to the different segments used in an assembly language program
- Format:
- **ASSUME** *segregister*:*name* [[, *segregister*:*name*]]...
- **ASSUME** *dataregister*:*type* [[, *dataregister*:*type*]]...
- **ASSUME** *register*:**ERROR** [[, *register*:**ERROR**]]...
- **ASSUME** [[*register*:]] **NOTHING** [[, *register*:**NOTHING**]]...

DB (Define Byte)

- It can be used to define data like **BYTE**.
- <u>Format</u>:
- *Name of the Variable DB Initial values*
- <u>Example</u>:
- WEIGHTS DB 18, 68, 45

DW (Define Word)

- It can be used to define data like **WORD** (2 bytes).
- <u>Format</u>:
- *Name of the Variable DW Initial values*
- <u>Example</u>:
- SUM DW 4589

DD (Define Double Word)

- It can be used to define data like **DWORD** (4 bytes).
- Format:
- *Name of the Variable DD Initial values*
- Example:
- NUMBER DD 12345678

DQ (Define Quad Word)

- It can be used to define data like **QWORD** (8 bytes).
- Format:
- *Name of the Variable DQ Initial values*
- Example:
- TABLE DQ 1234567812345678

DT (Define Ten Bytes)

- It can be used to define data like **TBYTE** (10 bytes).
- Format:
- *Name of the Variable   DT   Initial values*
- Example:
- AMOUNT   DT   12345678123456781234

END (End of program)

- It marks the end of a program module and, optionally, sets the program entry point to *address*.
- Format:
- **END** *[[address]]*
- Example:
- END label

ENDP (End Procedure)

- It marks the end of procedure.
- *name* previously begun with PROC.
- Format:
- *name***ENDP**
- Example:

```
CONTROL PROC FAR
  .
  .
  .
 CONTROL ENDP
```

- **ENDM (End Macro)**
- It terminates a macro or repeat block.
- Format:
- **ENDM**
- Example:

```
 CODE          MACRO
  .
  .
  .
  ENDM
```

- **ENDS (End of Segment)**
- It marks the end of segment, structure, or union *name* previously begun with SEGMENT, STRUCT, UNION, or a simplified segment directive.
- Format:
- *name* **ENDS**
- Example:

```
CODE SEGMENT
  .
  .
  .
 CODEENDS
```

EQU (Equate)

- It assigns numeric value of *expression or text* to *name*. The *name* cannot be redefined later.
- Format:
- *name* **EQU** *expression*
- *name* **EQU** **<***text***>**
- Example:
- CLEAR_CARRY EQU CLC

- **EVEN (Align on Even memory Address)**
- Format:
- EVEN
- Example:
- SALES DB 9
- EVEN
- DATA_ARRAY DW 100 DUP (?)

INCLUDE

- This directive inserts source code from the source file given by *filename* into the current source file during assembly. The *filename* must be enclosed in angle brackets if it includes a backslash, semicolon, greater-than symbol, less-than symbol, single quotation mark, or double quotation mark.
- Format:
- **INCLUDE** *filename*
- Example:
- INCLUDE C: \ MICRO \ ASSEM.LEV
- The above directive informs assembler to include all statements mentioned in the file, ASSEM.LEV from the directory C: \ MICRO.

MACRO

- A sequence of instructions to which a name is assigned is called a macro. The name of a macro is used in assembly language programming. Macros and subroutines are similar. Macros are used for short sequences of instructions, where as subroutines for longer ones. Macros execute faster than subroutines. A subroutine requires CALL and RET instructions whereas macros do not.
- <u>Format</u>:
- *name* **MACRO** [optional arguments]
- *statements* **ENDM**

ASSEMBLY LANGUAGE PROGRAMMING

Program

A computer can only do what the programmer asks to do. To perform a particular task the programmer prepares a sequence of instructions, called a program.

Programming languages

- Microcomputer programming languages can typically be divided into three main types:
    1. Machine language
    2. Assembly language
    3. High-level language

Machine language

- A program written in the form of 0s and 1s is called a machine language program. In the machine language program there is a specific binary code for each instruction.
- A microprocessor has a unique set of machine language instructions defined by its manufacturer.
- For example, the Intel 8085 uses the code $1000\ 1110_2$ for its addition instruction while the Motorola 6800 uses the code $1011\ 1001_2$.

The machine language program has the following demerits:

- It is very difficult to understand or debug a program.
- Program writing is difficult.
- Programs are long.
- More errors occur in writing the program.
- Since each bit has to be entered individually the entry of a program is very slow.

Assembly language

- Assembly language programming is writing machine instructions in mnemonic form, using an assembler to convert these mnemonics into actual processor instructions and associated data.

The advantages of assembly language programming

1.The computation time is
less. 2.It is faster to produce
result.

The disadvantages of assembly language programming

- many instructions are required to achieve small tasks
- source programs tend to be large and difficult to follow

High-level language

- High level language programs composed of English-language-type statements rectify all deficiencies of machine and assembly language programming. The high level languages are FORTRON, COBAL, BASIC, C, C++, Pascal, Visual Basic etc.

The high level language program has the following demerits:

- One has to learn the special rules for writing programs in a particular high level language.
- Low speed.
- A **compiler** has to be provided to convert a high level language program into a machine language program. The compiler is costly.

Assembly language program

- Assembly language statements are written one per line.
- A machine code program thus consists of a sequence of assembly language statements, where each statement contains a mnemonic.
- Each line of an assembly language program is split into four fields, as below:

    1. Label field
    2. Mnemonic or Opcode field
    3. Operand field
    4. Comment field

As an example, a typical program for block transfer of data written in 8086 assembly language is given here.

LABEL	OPCODE	OPERAND	COMMENTS
	CLD		Clear direction flag DF
	MOV	SI, 0200	Source address in SI
	MOV	DI, 0302	Destination address in DI
	MOV	CX, [SI]	Count in CX
	INC	SI	Increment SI
	INC	SI	Increment SI
BACK:	MOV	SB	Move byte
	LOOP	BACK	Jump to BACK until CX = 0
	INT		Interrupt program

LABEL

- The label field is optional. A label is an identifier.
- A label can be used to refer to a memory location the value of a piece of data the address of a program, sub-routine, code portion etc.

```
START: LDAA #24H
JMP START
```

- Here, the label START is equal to the address of the instruction LDAA #24H. The label is used in the program as a reference. This would result in the processor jumping to the location (address) associated with the label START, thus executing the instruction LDAA #24H immediately after the JMP instruction.

OPCODE

- Each instruction consists of an opcode (Mnemonic) and possible one or more operands. In the above instruction

  JMP START

- The opcode is JMP and the operand is the address of the label START.

Mnemonics are used because they

- are more meaningful than hex or binary values
- reduce the chances of making an error
- are easier to remember than bit values

OPERAND

- The operand field consists of additional information or data that the opcode requires. In certain types of addressing modes, the operand is used to specify
- constants or labels
- immediate data
- data contained in another accumulator or register
- an address

Examples of operands are

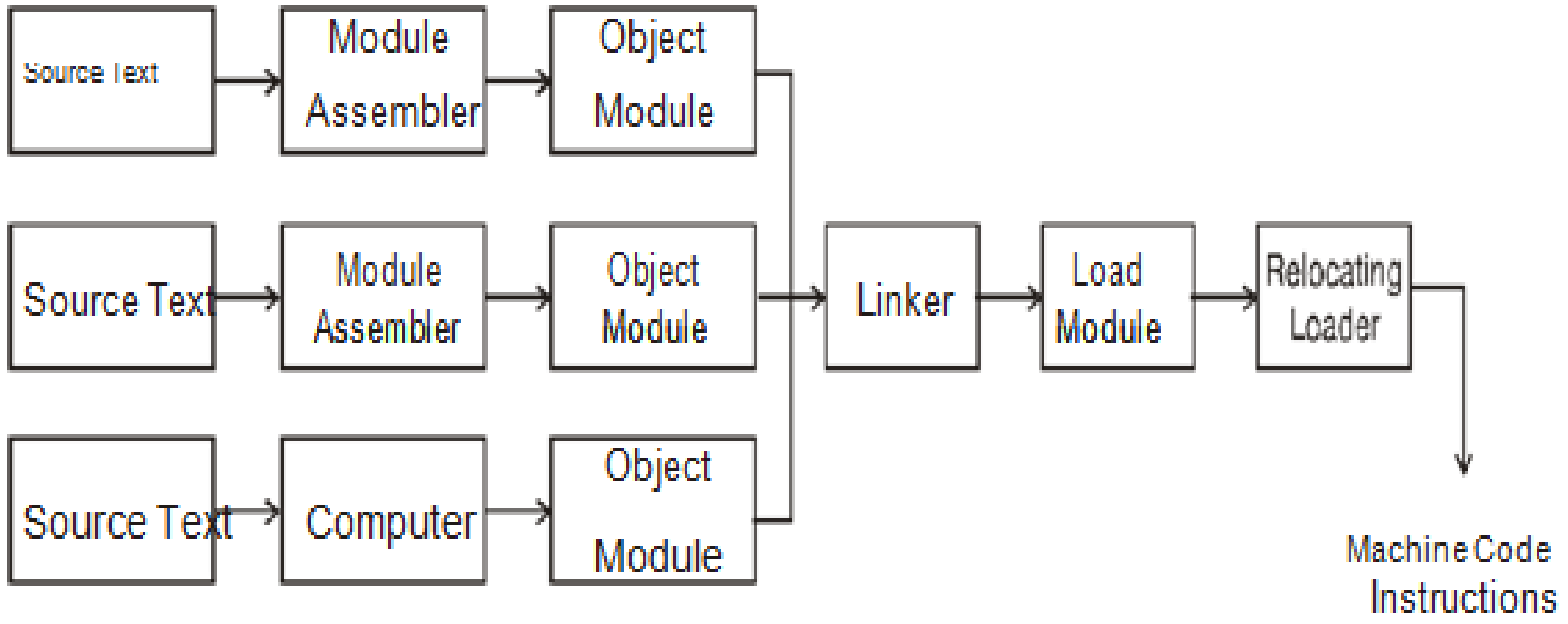
- JNZ STEP1
- MOV AX, 5000 H
- MOV AX, BX
- MOV AX, [3000 H]

COMMENTS

- The comment field is optional, and is used by the programmer to explain how the coded program works. Comments are preceded by a semi-colon. The assembler, when generating instructions from the source file, ignores all comments.

Assembly Language Program - Development Tools

- Editor
- Assembler
- Linker
- Locator
- Loader
- Debugger
- Emulator



Editor:

- An editor is a program which allows creating a file containing the assembly language statements for the program.

Assembler:

- An assembler is a program which translates an assembly language program into machine language program.

Linker:

- A linker is a program which links smaller programs together to form a large program. It is used to join several object files into one large object file. It also links the subroutines with the main program.

Locator:

- A locator is a program which assigns specific memory addresses for the machine codes of the program, which is to be loaded into the memory.

Loader:

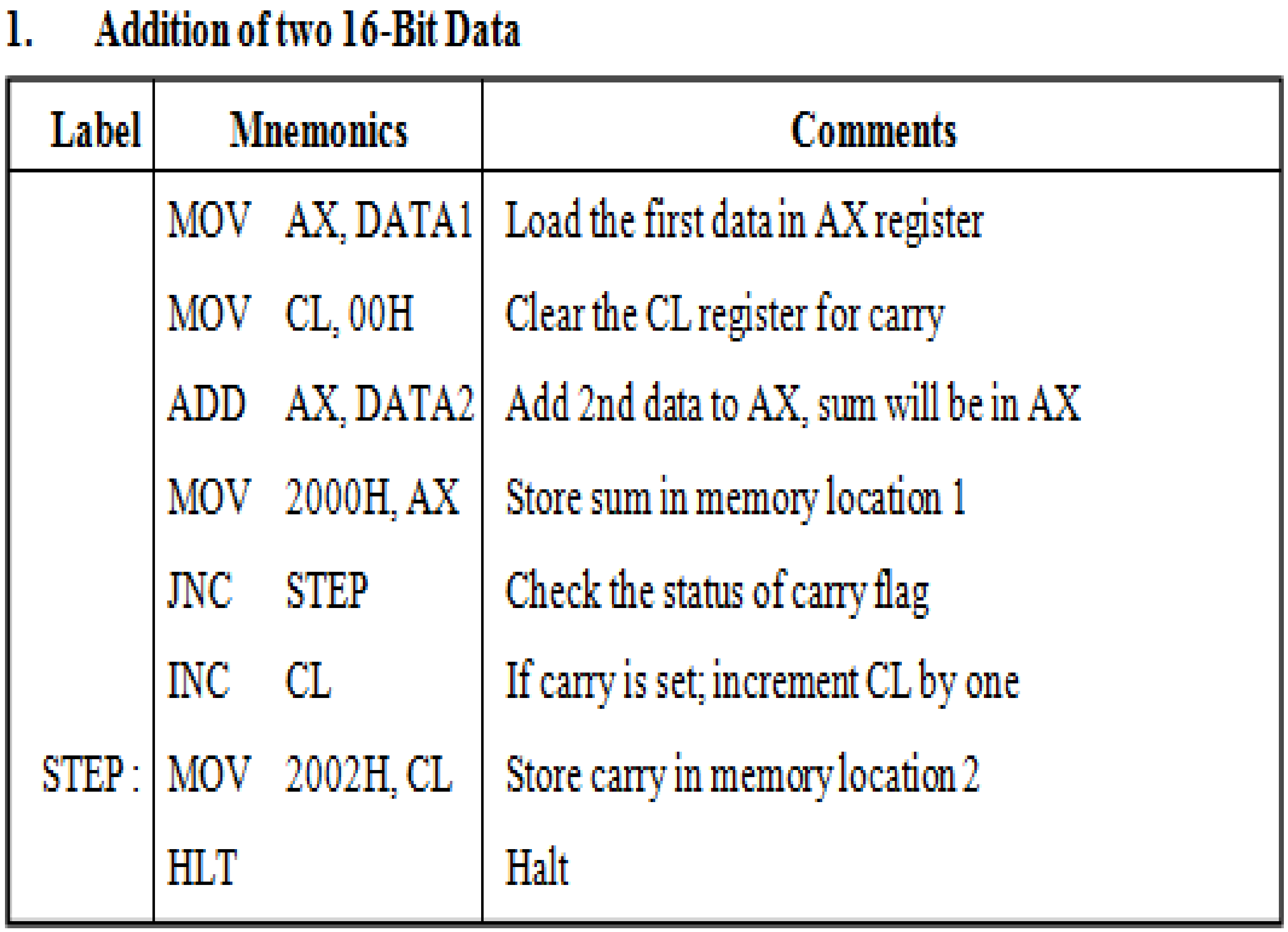
- A loader is a program which loads object code into system memory. It can accept programs in absolute or relocatable format.

Debugger:

- A debugger is a program which allows user to test and debug programs.

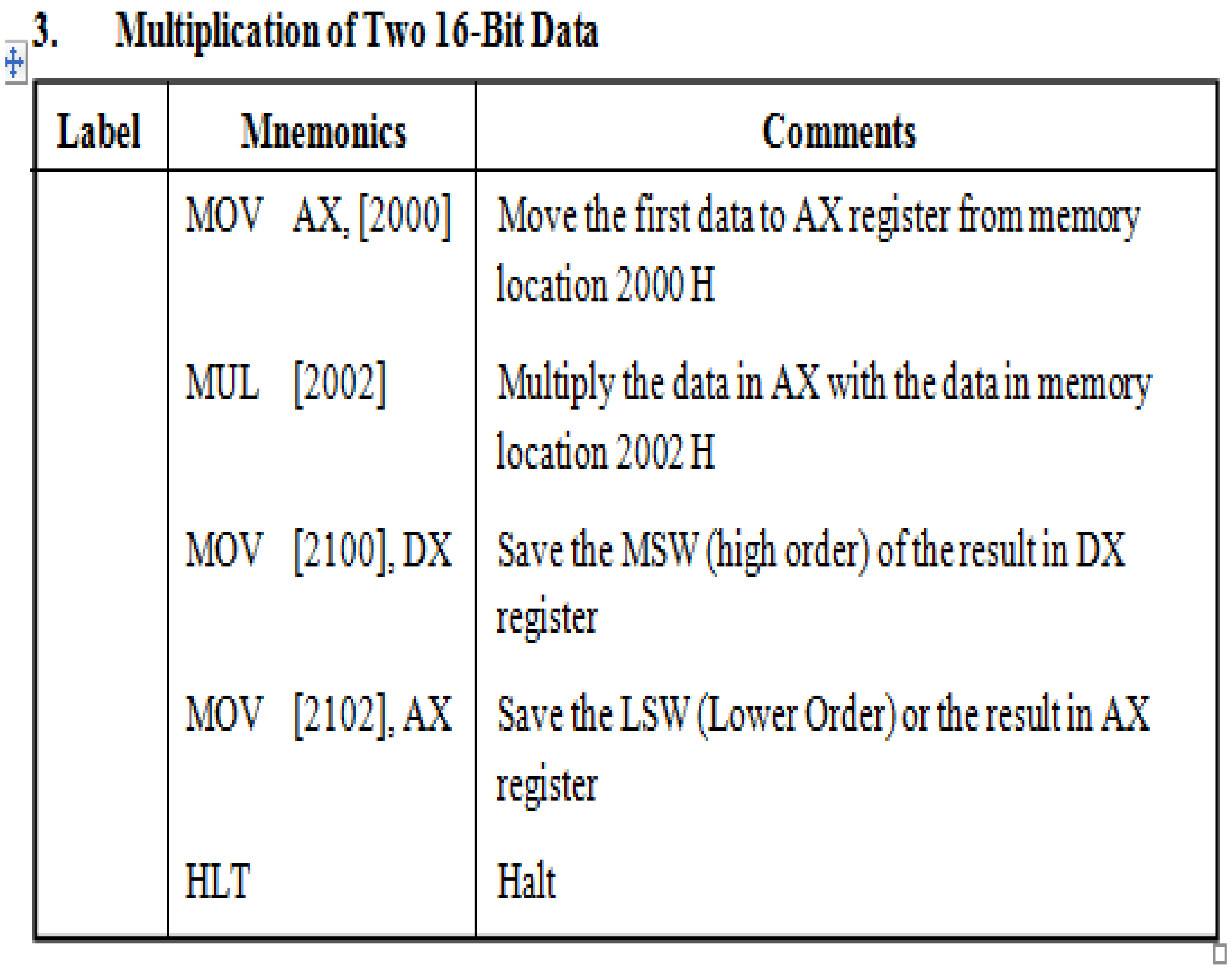
Emulator:

- An emulator is a mixture of software and hardware. It is usually used to test and debug the software and hardware of an external system.



1. **Addition of two 16-Bit Data**

Label	Mnemonics	Comments
	MOV AX, DATA1	Load the first data in AX register
	MOV CL, 00H	Clear the CL register for carry
	ADD AX, DATA2	Add 2nd data to AX, sum will be in AX
	MOV 2000H, AX	Store sum in memory location 1
	JNC STEP	Check the status of carry flag
	INC CL	If carry is set; increment CL by one
STEP :	MOV 2002H, CL	Store carry in memory location 2
	HLT	Halt



3. Multiplication of Two 16-Bit Data

Label	Mnemonics	Comments
	MOV AX, [2000]	Move the first data to AX register from memory location 2000 H
	MUL [2002]	Multiply the data in AX with the data in memory location 2002 H
	MOV [2100], DX	Save the MSW (high order) of the result in DX register
	MOV [2102], AX	Save the LSW (Lower Order) or the result in AX register
	HLT	Halt

MODULAR PROGRAMMING

- Modular programming is subdividing the complex program into separate subprograms such as functions and subroutines.
- Similar functions are grouped in the same unit of programming code and separate functions are developed as separate units of code so that the code can be reused by other applications.
- For example, if a program needs initial and boundary conditions, use subroutines to set them.
- Then if someone else wants to compute a different solution using the program, only these subroutines need to be changed. This is very easier than having to read through a program line by line, trying to figure out what each line is supposed to do and whether it needs to be changed.

- Subprograms make the actual program shorter, hence easier to read and understand. Further, the arguments show exactly what information a subprogram is using. That makes it easier to figure out whether it needs to be changed when modifying the program.

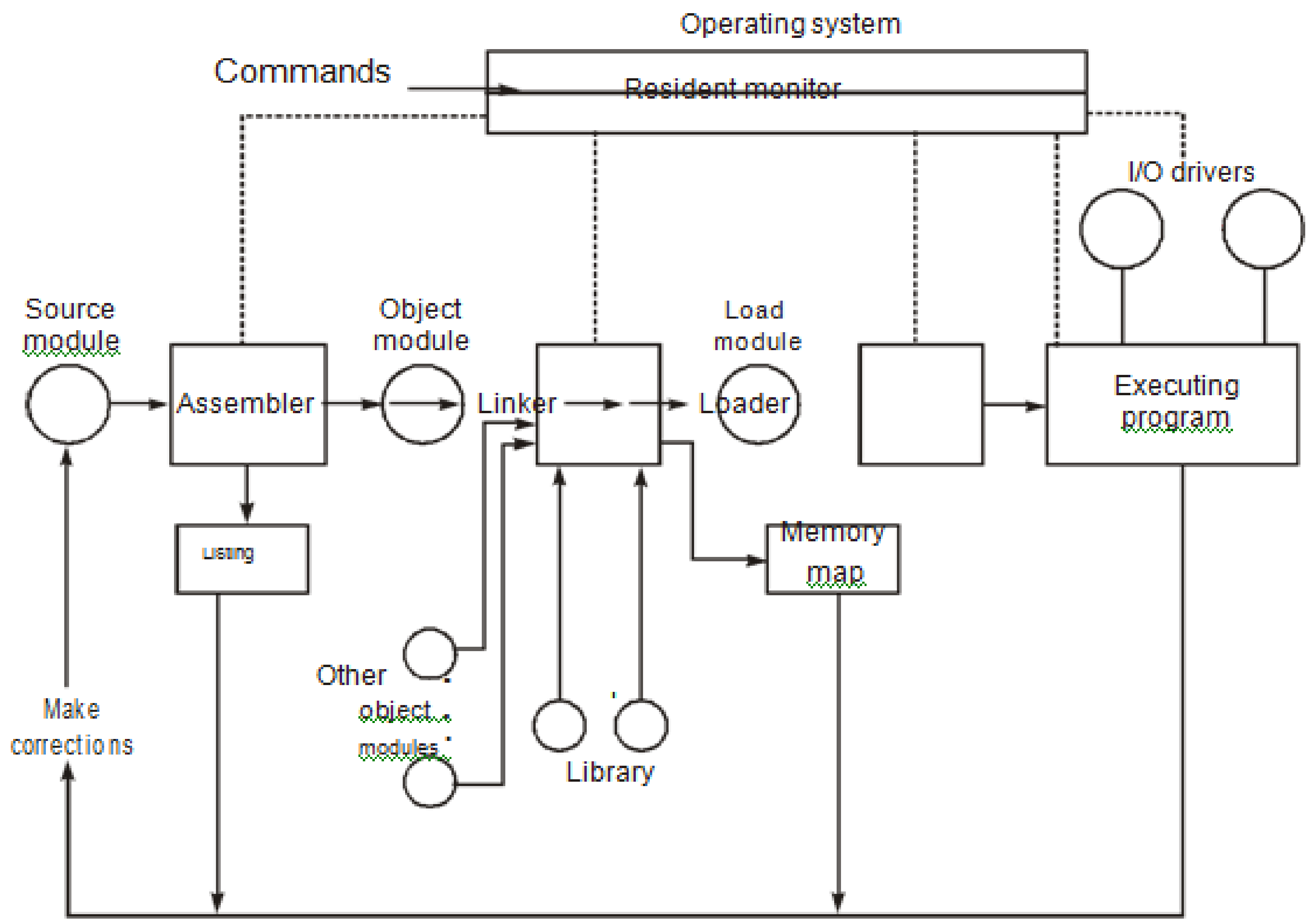
ALPs are developed by essentially the same procedure as high-level language programs by,

- Exactly stating what the program is to do.
- Splitting the overall problem into tasks.
- Defining exactly what each task must do and how it is to communicate with the other tasks.
- Putting the tasks into assembler language modules and connecting the modules together to form the program.
- Debugging and testing the program.
- Documenting the program.

The benefits of using modular programming are,

- Modular programming allows many programmers to collaborate on the same application.
- Same code can be used in many applications.
- Code is short, simple and easy to understand.
- Code is stored across multiple files.
- A single procedure can be developed for reuse, eliminating the need to retype the code many times.
- Errors can easily be identified, as they are localized to a subroutine or function.

LINKING AND RELLOCATION



The process combines the following.

- Find the object modules to be linked.
- Construct the load module by assigning the positions of all of all the segments in all of the object modules being linked.
- Fill in all offset that could not be determined by the assembler.
- Fill in all segment address.
- Load the program for execution.

Segment combination

- In addition to the linker commands, the assembler provides a means of regulating the way segments in different object modules are organized by the linker. Segments with same name are joined together by using the modifiers attached to the SEGMENT directives. SEGMENT directive may have the form:
- **Segment name SEGMENT Combination-type**

PROCEDURES & MACROS

- A single instruction that expands automatically into a set of instructions to perform a particular task.
- A **macro** (which stands for "macroinstruction") is a programmable pattern which translates a certain sequence of input into a preset sequence of output. **Macros** can be used to make tasks less repetitive by representing a complicated sequence of keystrokes, mouse movements, commands, or other types of input.

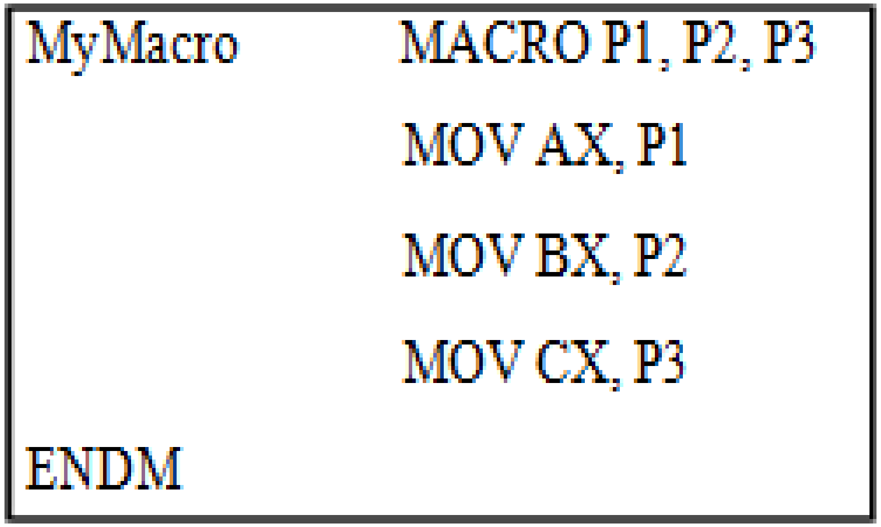
Macro definition:

name MACRO [parameters,...]

statements >

ENDM

Example:



```
MyMacro      MACRO P1, P2, P3
             MOV AX, P1
             MOV BX, P2
             MOV CX, P3
ENDM
```

Advantages of macros

- Repeated small groups of instructions replaced by one macro
- Errors in macros are fixed only once, in the definition
- Duplication of effort is reduced
- In effect, new higher level instructions can be created
- Programming is made easier, less error prone
- Generally quicker in execution than subroutines

Disadvantages of macros

- In large programs, produce greater code size than procedures

When to use Macros

- To replace small groups of instructions not worthy of subroutines
- To create a higher instruction set for specific applications
- To create compatibility with other computers
- To replace code portions which are repeated often throughout the program

Procedure (PROC)

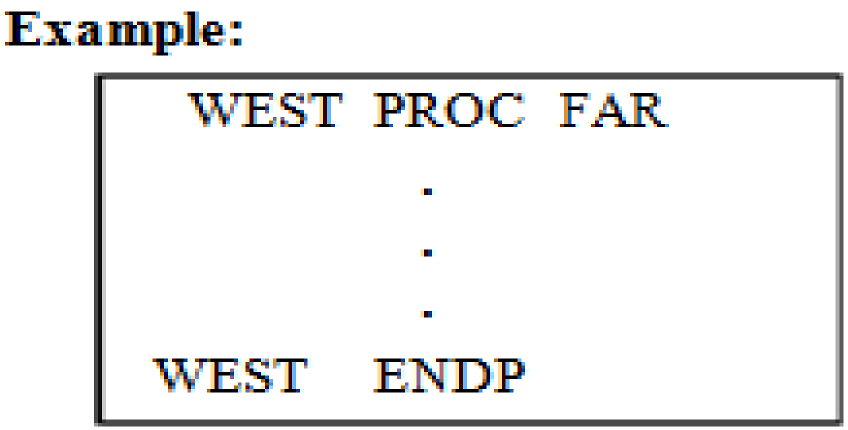
- This directive marks the start and end of a procedure block called *label*. The statements in the block can be called with the **CALL** instruction.

PROC definition:

label **PROC** [[near / far]]

<Procedure instructions>

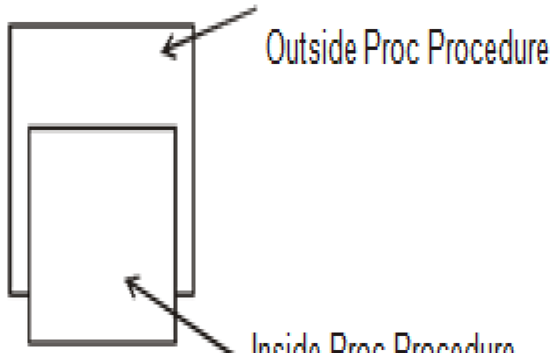
label **ENDP**



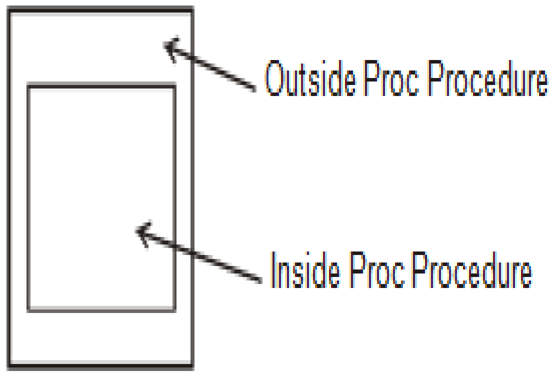
Example:

```
    WEST  PROC  FAR
               .
               .
               .
  WEST   ENDP
```

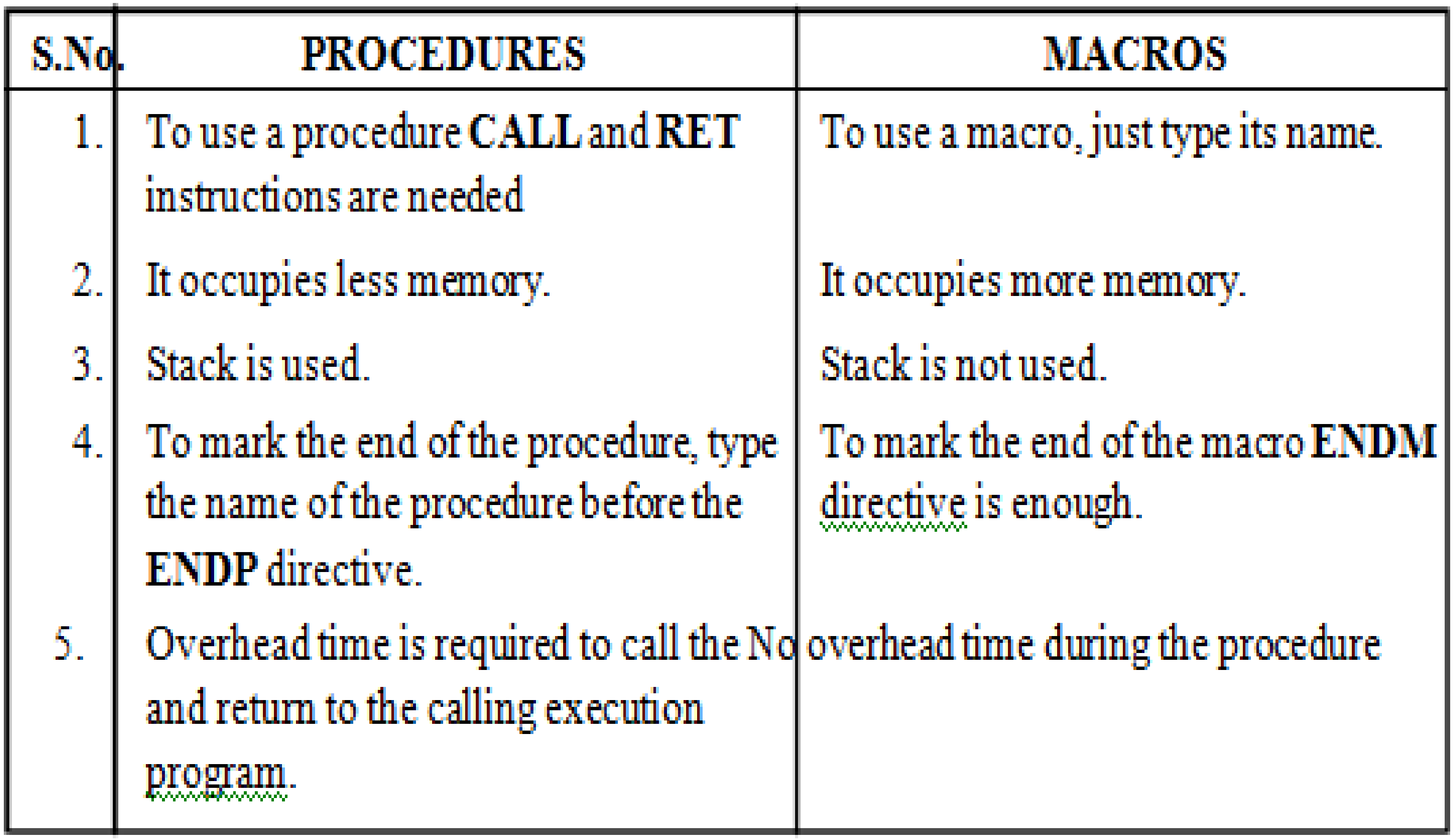
Overlapping Proc



Nested Proc



Differences between Macros and Procedures



S.No.	PROCEDURES	MACROS
1.	To use a procedure **CALL** and **RET** instructions are needed	To use a macro, just type its name.
2.	It occupies less memory.	It occupies more memory.
3.	Stack is used.	Stack is not used.
4.	To mark the end of the procedure, type the name of the procedure before the **ENDP** directive.	To mark the end of the macro **ENDM** directive is enough.
5.	Overhead time is required to call the and return to the calling execution program.	No overhead time during the procedure

INTERRUPTS AND INTERRUPT SERVICE ROUTINES

Interrupts

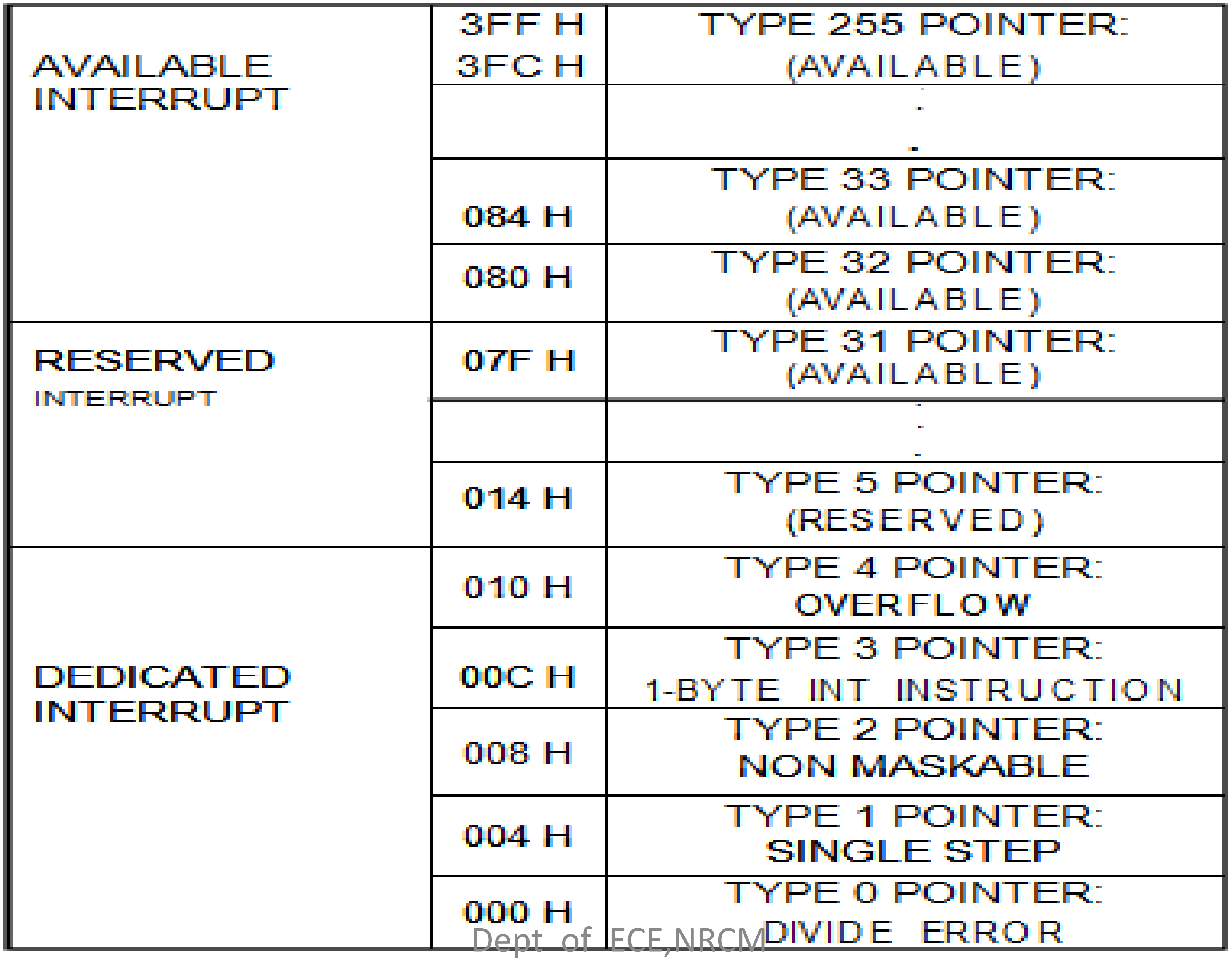
- A signal to the processor to halt its current operation and immediately transfer control to an interrupt service routine is called as interrupt. Interrupts are triggered either by hardware, as when the keyboard detects a key press, or by software, as when a program executes the INT instruction.

- Interrupts can be seen as a number of functions. These functions make the programming much easier, instead of writing a code to print a character, simply call the interrupt and it will do everything.
- There are also interrupt functions that work with disk drive and other hardware. They are called as **software interrupts**.
- Interrupts are also triggered by different hardware, these are called **hardware interrupts**.
- To make a software interrupt there is an INT instruction, it has very simple syntax: INT *value*.
- Where value can be a number between 0 to 255 (or 00 to FF H).

Interrupt Service Routines (ISRs)

- ISR is a routine that receives processor control when a specific interrupt occurs.
- The 8086 will directly call the service routine for 256 vectored interrupts without any software processing. This is in contrast to non vectored interrupts that transfer control directly to a single interrupt service routine, regardless of the interrupt source.

Interrupt vector table:



AVAILABLE INTERRUPT	3FF H 3FC H	TYPE 255 POINTER: (AVAILABLE)
		. .
	084 H	TYPE 33 POINTER: (AVAILABLE)
	080 H	TYPE 32 POINTER: (AVAILABLE)
RESERVED INTERRUPT	07F H	TYPE 31 POINTER: (AVAILABLE)
		. . .
	014 H	TYPE 5 POINTER: (RESERVED)
DEDICATED INTERRUPT	010 H	TYPE 4 POINTER: OVERFLOW
	00C H	TYPE 3 POINTER: 1-BYTE INT INSTRUCTION
	008 H	TYPE 2 POINTER: NON MASKABLE
	004 H	TYPE 1 POINTER: SINGLE STEP
	000 H	TYPE 0 POINTER: DIVIDE ERROR

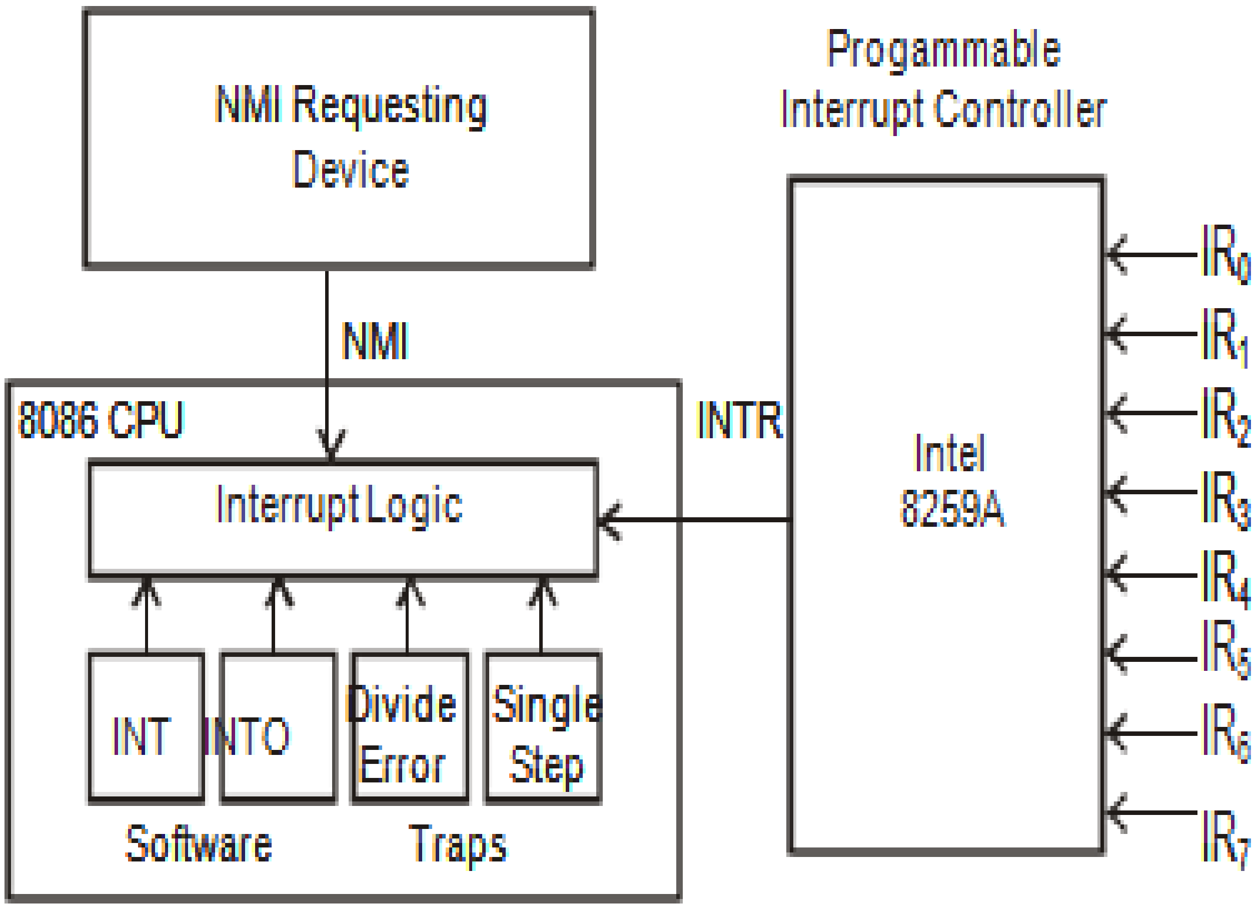
When an interrupt occurs, regardless of source, the 8086 does the following:

- The CPU pushes the flags register onto the stack.
- The CPU pushes a far return address (segment:offset) onto the stack, segment value first.
- The CPU determines the cause of the interrupt (i.e., the interrupt number) and fetches the four byte interrupt vector from address 0 **:** vector x 4 (0:0, 0:4, 0:8 etc)
- The CPU transfers control to the routine specified by the interrupt vector table entry.

After the completion of these steps, the interrupt service routine takes control. When the interrupt service routine wants to return control, it must execute an IRET (interrupt return) instruction. The interrupt return pops the far return address and the flags off the stack

Types of Interrupts

- Hardware Interrupt - External uses INTR and NMI
- Software Interrupt - Internal - from INT or INTO
- Processor Interrupt - Traps and 10 Software Interrupts
- E*xternal* - generated outside the CPU by other hardware (INTR, NMI)
- *Internal* - generated within CPU as a result of an instruction or operation (INT, INTO, Divide Error and Single Step)


NMI Requesting Device
NMI
8086 CPU
Interrupt Logic
INT
INTO
Divide Error
Single Step
Software
Traps
INTR
Progammable Interrupt Controller
Intel 8259A
IR0
IR1
IR2
IR3
IR4
IR5
IR6
IR7

Dedicated Interrupts

- **Divide Error Interrupt (Type 0)**

  This interrupt occurs automatically following the execution of DIV or IDIV instructions when the quotient exceeds the maximum value that the division instructions allow.

- **Single Step Interrupt (Type 1)**

  This interrupt occurs automatically after execution of each instruction when the Trap Flag (TF) is set to 1.It is used to execute programs one instruction at a time, after which an interrupt is requested. Following the ISR, the next instruction is executed and another single stepping interrupt request occurs.

- **Non Maskable Interrupt (Type 2)**

  It is the highest priority hardware interrupt that triggers on the positive edge.

  This interrupt occurs automatically when it receives a low-to-high transition on its NMI input pin.

  This interrupt cannot be disabled or masked. It is used to save program data or processor status in case of system power failure.

- **Breakpoint Interrupt (Type 3)**

  This interrupt is used to set break points in software debugging programs.

- **Overflow Interrupt (Type 4)**

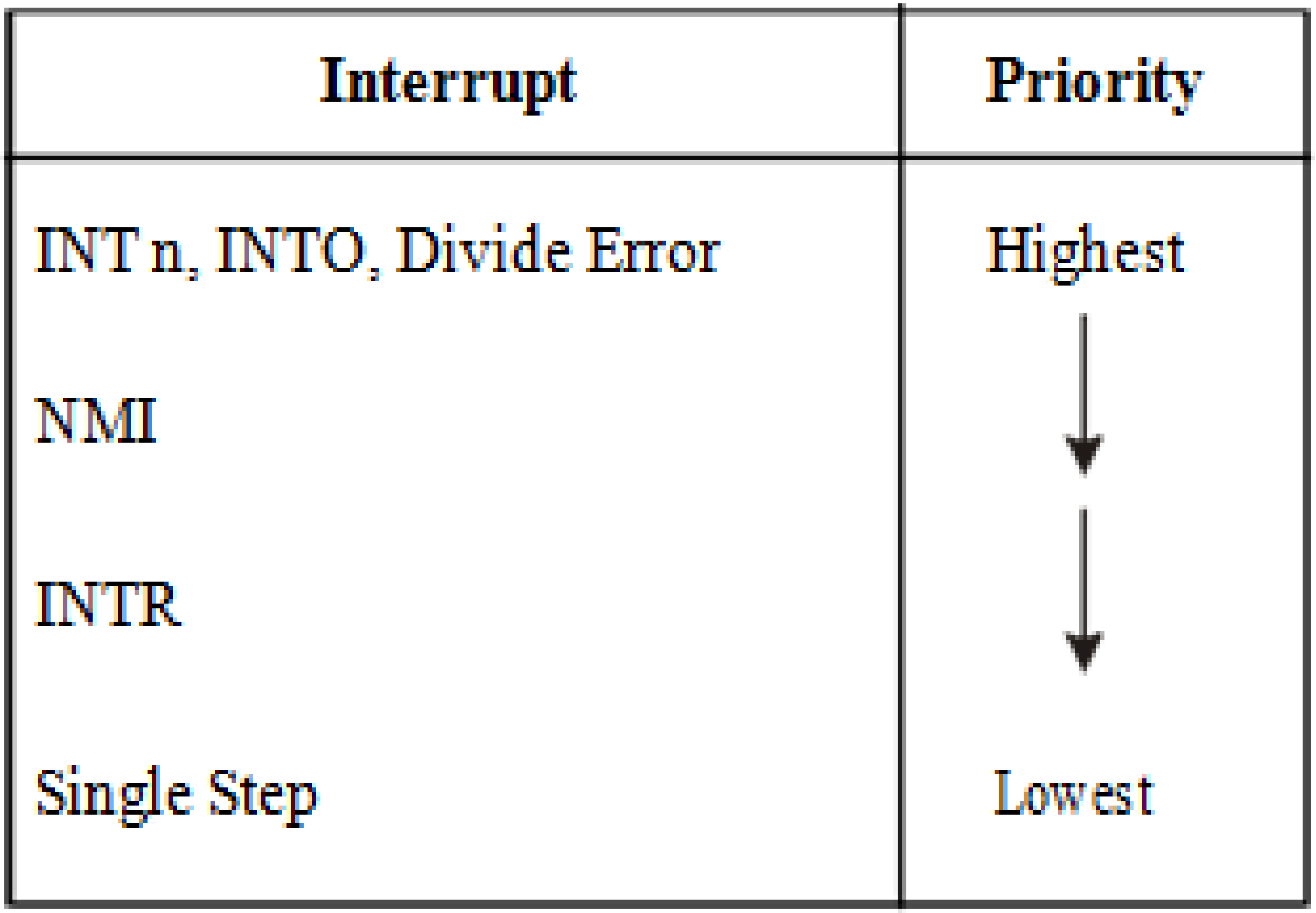
Software Interrupts (INT n)

- The software interrupts are non maskable interrupts. They are higher priority than hardware interrupts.

Hardware Interrupts

- INTR and NMI are called hardware interrupts. INTR is maskable and NMI is non-maskable interrupts.

Interrupt Priority



Interrupt	Priority
INT n, INTO, Divide Error	Highest
NMI	↓
INTR	↓
Single Step	Lowest

Byte And String Manipulation

- The 8086 microprocessor is equipped with special instructions to handle string operations.
- By string we mean a series of data words or bytes that reside in consecutive memory locations.
- The string instructions of the 8086 permit a programmer to implement operations such as to move data from one block of memory to a block elsewhere in memory.

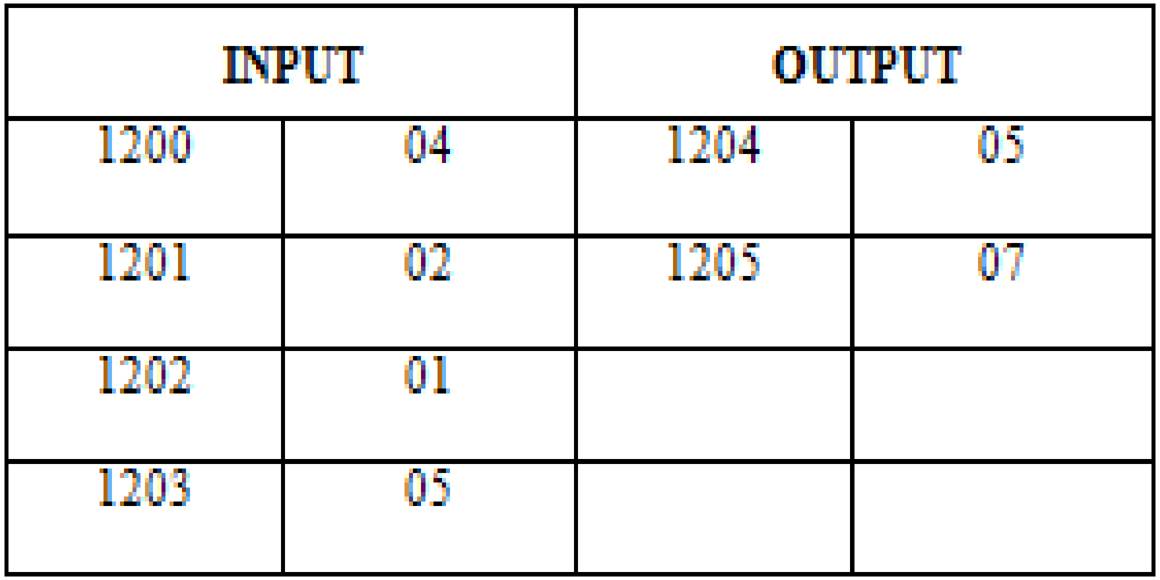
- A second type of operation that is easily performed is to scan a string and data elements stored in memory looking for a specific value.
- Other examples are to compare the elements and two strings together in order to determine whether they are the same or different.
- **Move String** : MOV SB, MOV SW: An element of the string specified by the source index (SI) register with respect to the current data segment (DS) register is moved to the location specified by the destination index (DI) register with respect to the current extra segment (ES) register.

- The move can be performed on a byte (MOV SB) or a word (MOV SW) of data. After the move is complete, the contents of both SI & DI are automatically incremented or decremented by 1 for a byte move and by 2 for a word move.
- Address pointers SI and DI increment or decrement depends on how the direction flag DF is set.

- **Load and store strings :** (LOD SB/LOD SW and STO SB/STO SW) LOD SB: Loads a byte from a string in memory into AL. The address in SI is used relative to DS to determine the address of the memory location of the string element. (AL) <= [(DS) + (SI)] (SI) <= (SI) + 1
- LOD SW : The word string element at the physical address derived from DS and SI is to be loaded into AX. SI is automatically incremented by 2. (AX) <= [(DS) + (SI)] (SI) <= (SI) + 2
- STO SB : Stores a byte from AL into a string location in memory. This time the contents of ES and DI are used to form the address of the storage location in memory [(ES) + (DI)] <= (AL) (DI) <=(DI) + 1
- STO SW : [(ES) + (DI)] <= (AX) (DI) <= (DI) + 2

1.A. 16 BIT ADDITION USING 8086

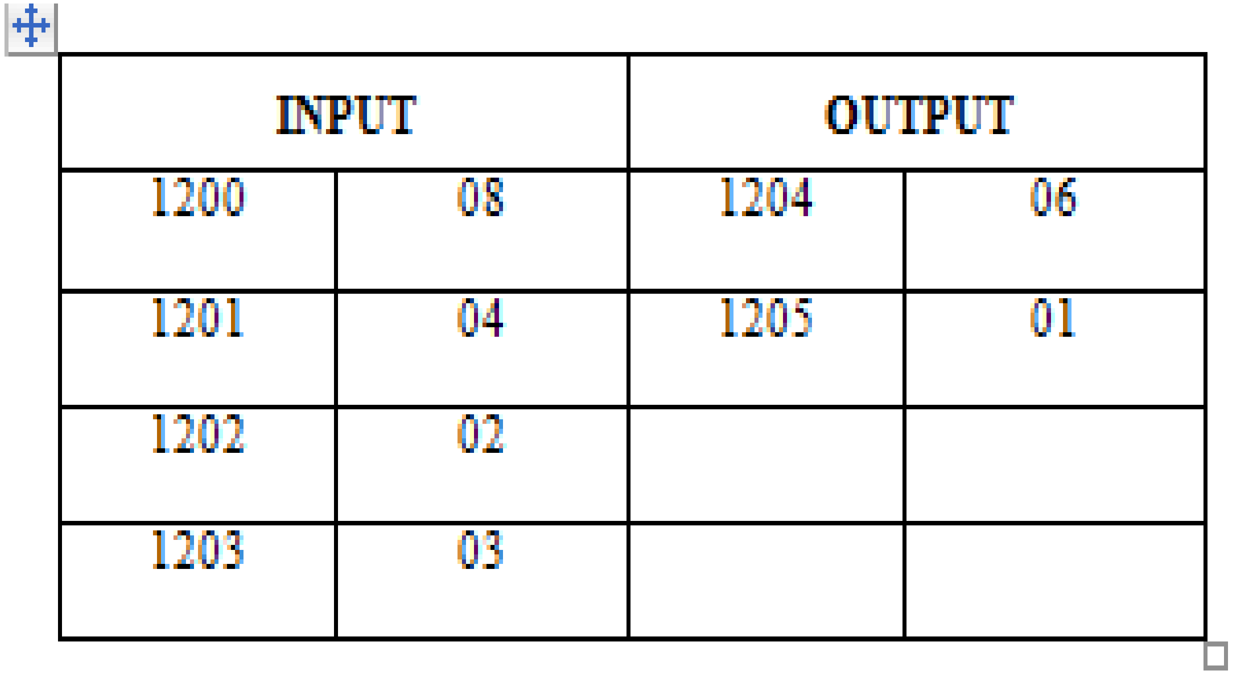
ADDRESS	LABEL	MNEMONICS	OPCODE	COMMENTS
1000		MOV AX,[1200]H	A1	Clear C register
			00	
			12	
1003		ADD AX,[1202]H	03	Move the immediate data 1 to accumulator
			06	
			02	
			12	
1007		MOV[1204]H,AX	A3	Move the immediate data 2 to B register
			04	
			12	
100A		HLT	F4	End the program



INPUT		OUTPUT	
1200	04	1204	05
1201	02	1205	07
1202	01		
1203	05		

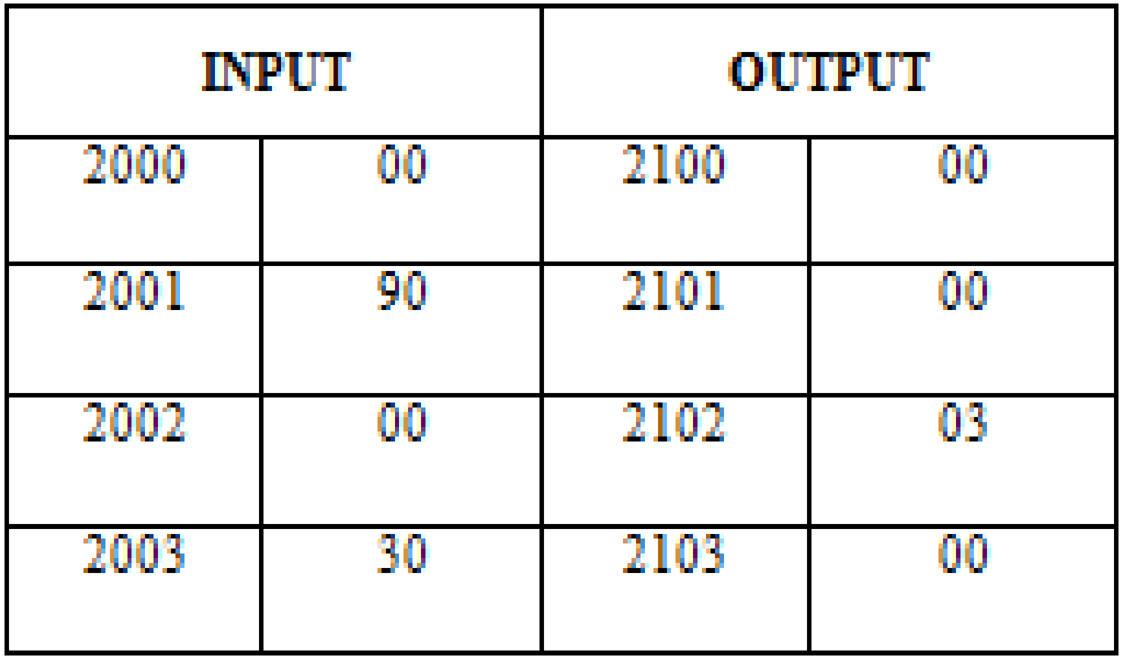
1.B. 16BIT SUBTRACTION USING 8086

ADDRESS	LABEL	MNEMONICS	OPCODE	COMMENTS
1000		MOV AX,[1200]H	A1	Clear C register
			00	
			12	
1003		SUB AX,[1202]H	2B	Move the immediate data 1 to accumulator
			06	
			02	
			12	
1007		MOV[1204]H, AX	A3	Move the immediate data 2 to B register
			04	
			12	
100A		HLT	F4	End the program



INPUT		OUTPUT	
1200	08	1204	06
1201	04	1205	01
1202	02		
1203	03		

INPUT		OUTPUT	
2000	02	2100	06
2001	03	2101	00
2002	03	2102	09
2003	03	2103	00

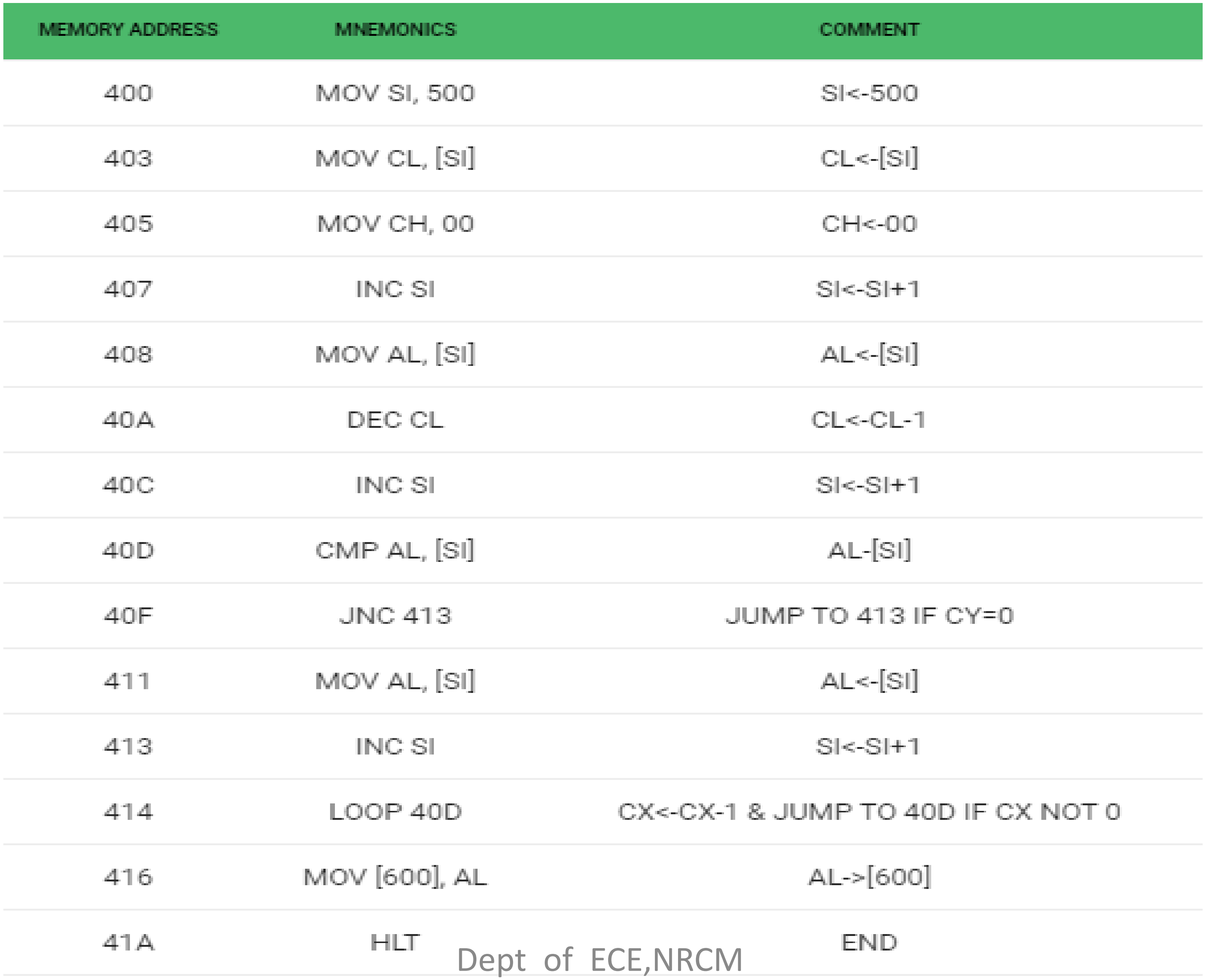


INPUT		OUTPUT	
2000	00	2100	00
2001	90	2101	00
2002	00	2102	03
2003	30	2103	00

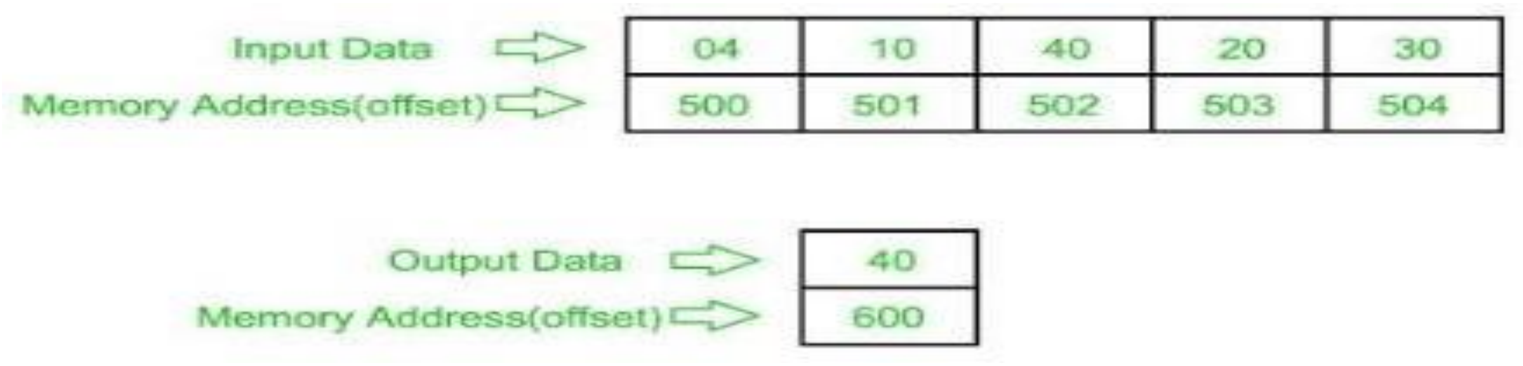
8086 program to determine largest number in an array of n numbers

Algorithm –

- Load data from offset 500 to register CL and set register CH to 00 (for count).
- Load first number(value) from next offset (i.e 501) to register AL and decrease count by 1.
- Now compare value of register AL from data(value) at next offset, if that data is greater than value of register AL then update value of register AL to that data else no change, and increase offset value for next comparison and decrease count by 1 and continue this till count (value of register CX) becomes 0.
- Store the result (value of register AL) to memory address 2000 : 600.



MEMORY ADDRESS	MNEMONICS	COMMENT
400	MOV SI, 500	SI<-500
403	MOV CL, [SI]	CL<-[SI]
405	MOV CH, 00	CH<-00
407	INC SI	SI<-SI+1
408	MOV AL, [SI]	AL<-[SI]
40A	DEC CL	CL<-CL-1
40C	INC SI	SI<-SI+1
40D	CMP AL, [SI]	AL-[SI]
40F	JNC 413	JUMP TO 413 IF CY=0
411	MOV AL, [SI]	AL<-[SI]
413	INC SI	SI<-SI+1
414	LOOP 40D	CX<-CX-1 & JUMP TO 40D IF CX NOT 0
416	MOV [600], AL	AL->[600]
41A	HLT	END



Input Data	04	10	40	20	30
Memory Address(offset)	500	501	502	503	504

Output Data	40
Memory Address(offset)	600

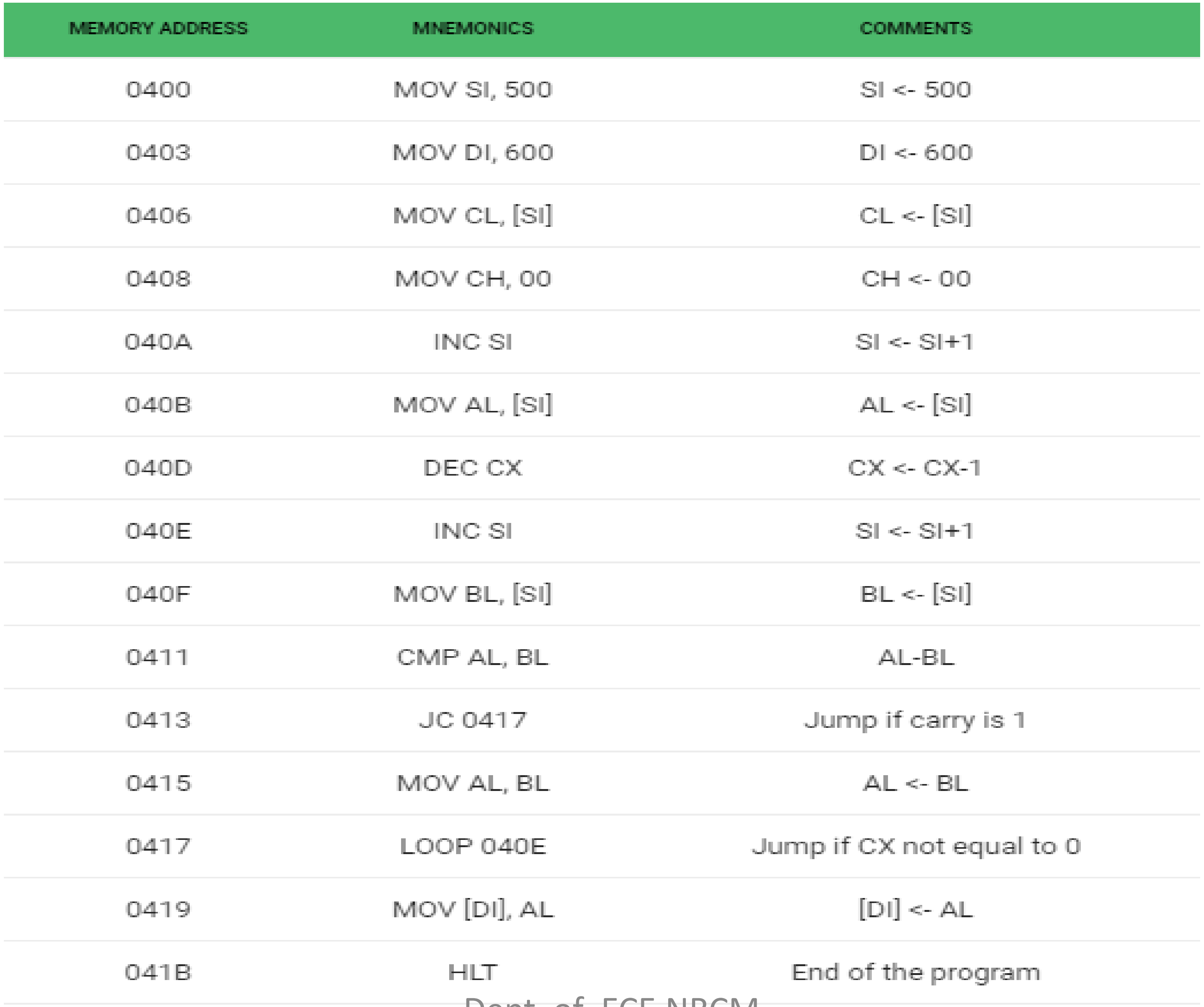
Explanation –

- **MOV SI, 500** : set the value of SI to 500
- **MOV CL, [SI]** : load data from offset SI to register CL
- **MOV CH, 00** : set value of register CH to 00
- **INC SI** : increase value of SI by 1.
- **MOV AL, [SI]** : load value from offset SI to register AL
- **DEC CL** : decrease value of register CL by 1
- **INC SI** : increase value of SI by 1
- **CMP AL, [SI]** : compares value of register AL and [SI] (AL-[SI])
- **JNC 413** : jump to address 413 if carry not generated
- **MOV AL, [SI]** : transfer data at offset SI to register AL
- **INC SI** : increase value of SI by 1
- **LOOP 40C** : decrease value of register CX by 1 and jump to address 40D if value of register CX is not zero
- **MOV [600], AL** : store the value of register AL to offset 600
- **HLT** : stop

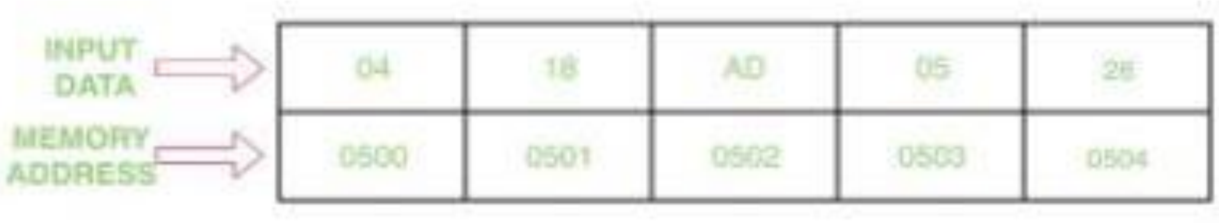
8086 program to find the min value in a given array

Algorithm –

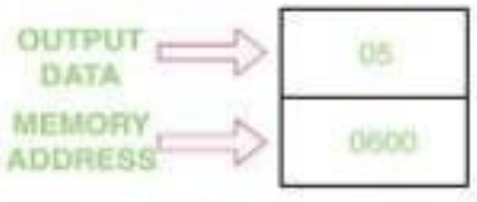
- Assign value 500 in SI and 600 in DI
- Move the contents of [SI] in CL and increment SI by 1
- Assign the value 00 H to CH
- Move the content of [SI] in AL
- Decrease the value of CX by 1
- Increase the value of SI by 1
- Move the contents of [SI] in BL
- Compare the value of BL with AL
- Jump to step 11 if carry flag is set
- Move the contents of BL in AL
- Jump to step 6 until the value of CX becomes 0, and decrease CX by 1
- Move the contents of AL in [DI]
- Halt the program



MEMORY ADDRESS	MNEMONICS	COMMENTS
0400	MOV SI, 500	SI <- 500
0403	MOV DI, 600	DI <- 600
0406	MOV CL, [SI]	CL <- [SI]
0408	MOV CH, 00	CH <- 00
040A	INC SI	SI <- SI+1
040B	MOV AL, [SI]	AL <- [SI]
040D	DEC CX	CX <- CX-1
040E	INC SI	SI <- SI+1
040F	MOV BL, [SI]	BL <- [SI]
0411	CMP AL, BL	AL-BL
0413	JC 0417	Jump if carry is 1
0415	MOV AL, BL	AL <- BL
0417	LOOP 040E	Jump if CX not equal to 0
0419	MOV [DI], AL	[DI] <- AL
041B	HLT	End of the program

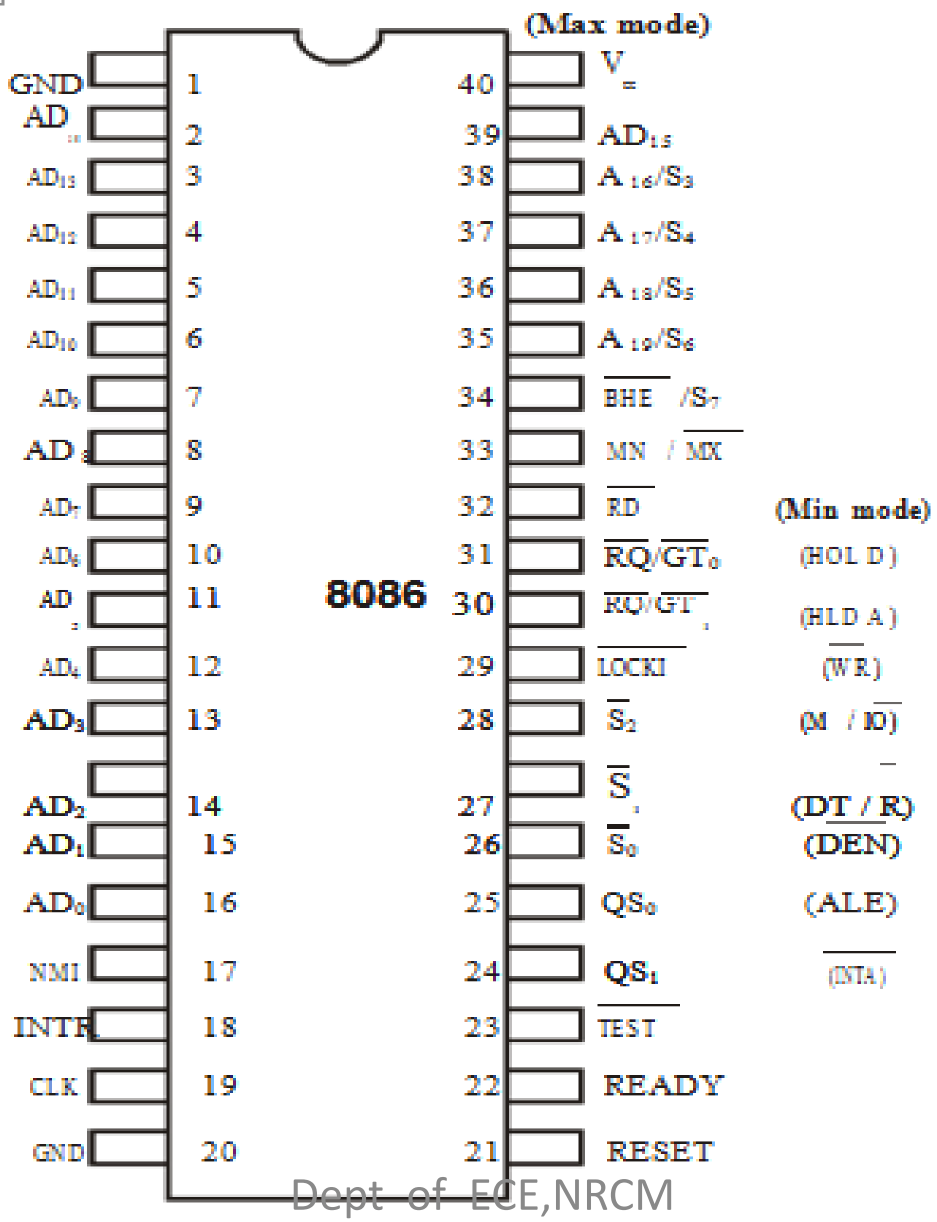


INPUT DATA	04	18	AD	05	2B
MEMORY ADDRESS	0500	0501	0502	0503	0504

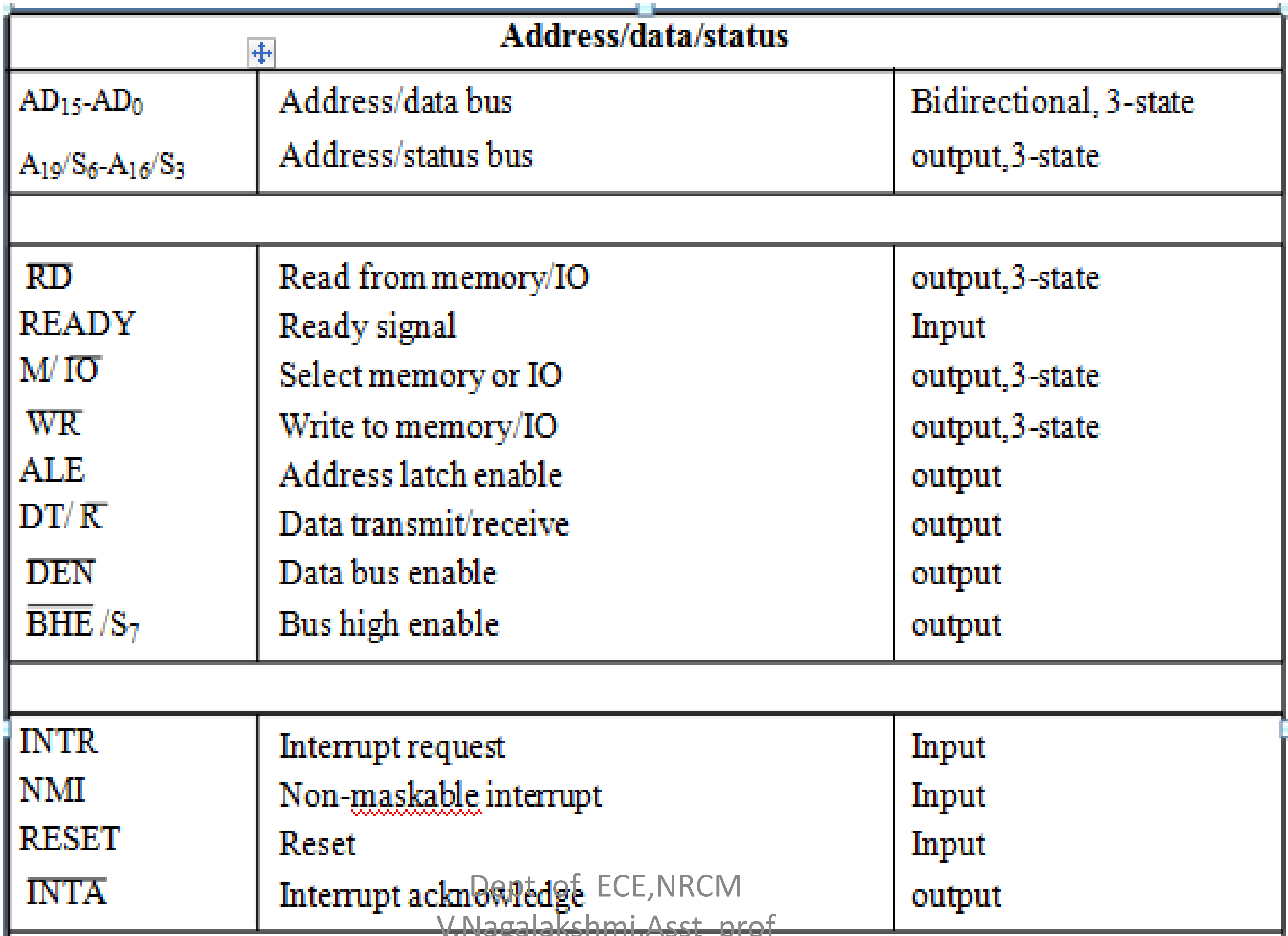


OUTPUT DATA	05
MEMORY ADDRESS	0600

PIN DIAGRAM


(Max mode)
GND 1
AD14 2
AD13 3
AD12 4
AD11 5
AD10 6
AD9 7
AD8 8
AD7 9
AD6 10
AD5 11
AD4 12
AD3 13
AD2 14
AD1 15
AD0 16
NMI 17
INTR 18
CLK 19
GND 20
8086
40 VCC
39 AD15
38 A16/S3
37 A17/S4
36 A18/S5
35 A19/S6
34 BHE /S7
33 MN / MX
32 RD
31 RQ/GT0 (HOLD)
30 RQ/GT1 (HLDA)
29 LOCK (WR)
28 S2 (M / IO)
27 S1 (DT / R)
26 S0 (DEN)
25 QS0 (ALE)
24 QS1 (INTA)
23 TEST
22 READY
21 RESET
(Min mode)

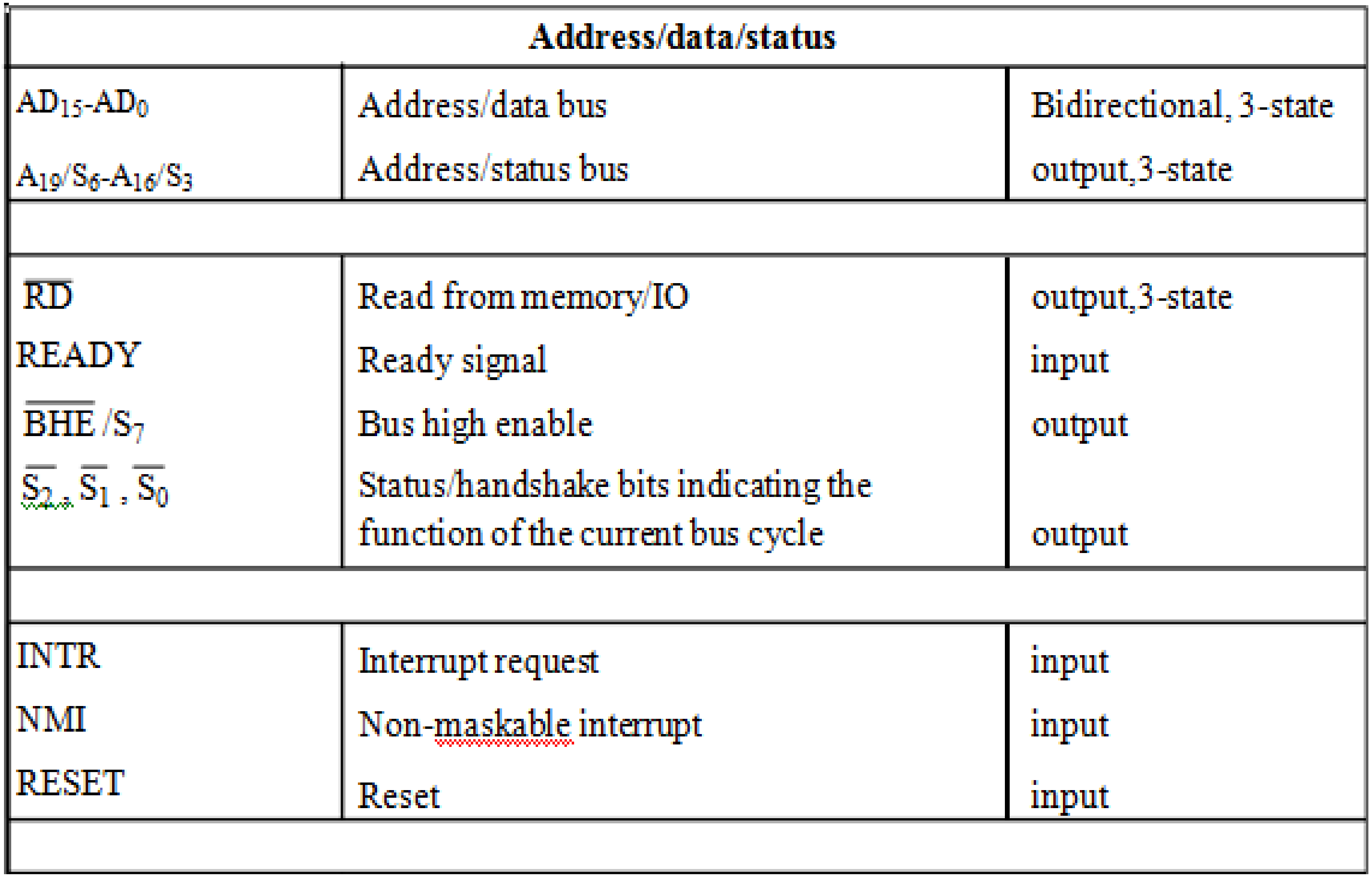
MINIMUM MODE SIGNALS



Address/data/status		
AD_{15}-AD_0	Address/data bus	Bidirectional, 3-state
A_{19}/S_6-A_{16}/S_3	Address/status bus	output,3-state
$\overline{RD}$	Read from memory/IO	output,3-state
READY	Ready signal	Input
$M/\overline{IO}$	Select memory or IO	output,3-state
$\overline{WR}$	Write to memory/IO	output,3-state
ALE	Address latch enable	output
$DT/\overline{R}$	Data transmit/receive	output
$\overline{DEN}$	Data bus enable	output
$\overline{BHE}/S_7$	Bus high enable	output
INTR	Interrupt request	Input
NMI	Non-maskable interrupt	Input
RESET	Reset	Input
$\overline{INTA}$	Interrupt acknowledge	output

HOLD HLDA	Hold request Hold acknowledge	Input output
$\overline{TEST}$ MN/ $\overline{MX}$ CLK V_{cc} GND	Test pin tested by WAIT instruction Minimum/maximum mode, 5V Clock pin for basic timing signal Power supply, +5 V Ground connection, 0V	Input Input Input

MAXIMUM MODE SIGNALS



Address/data/status		
AD_{15}-AD_0 A_{19}/S_6-A_{16}/S_3	Address/data bus Address/status bus	Bidirectional, 3-state output,3-state
$\overline{RD}$ READY $\overline{BHE}/S_7$ $\overline{S_2}$, $\overline{S_1}$, $\overline{S_0}$	Read from memory/IO Ready signal Bus high enable Status/handshake bits indicating the function of the current bus cycle	output,3-state input output output
INTR NMI RESET	Interrupt request Non-maskable interrupt Reset	input input input

$\overline{RQ}/\overline{GT}_1$, $\overline{RQ}/\overline{GT}_0$ $\overline{LOCK}$	Request/grant pins for bus access Used to lock the bus, activated by LOCK prefix on any instruction	bidirectional output
QS_1,QS_0 $\overline{TEST}$ MN/ $\overline{MX}$ CLK V_{cc} GND	Queue status Test pin tested by WAIT instruction Minimum/maximum mode, 0V Clock pin for basic timing signal Power supply, +5 V Ground connection, 0V	output input input input

Address / Data Bus (AD_{15}–AD_0)

- The multiplexed Address/ Data bus acts as address bus during the first part of machine cycle (T1) and data bus for the remaining part of the machine cycle.

Address/Status ($A_{19}/S_6, A_{18}/S_5, A_{17}/S_4, A_{16}/S_3$)

- During T1 these are the four most significant address lines for memory operations.
- During I/O operations these lines are LOW.

S_4	S_3	Function
0	0	ES, Extra segment
0	1	SS, Stack Segment
1	0	CS, Code segment
1	1	DS, Data segment

$\overline{BHE}$	A_0	Characteristics
0	0	Whole word
0	1	Upper byte from/to odd address
1	0	Lower byte from/to even address
1	1	None

Read(RD)

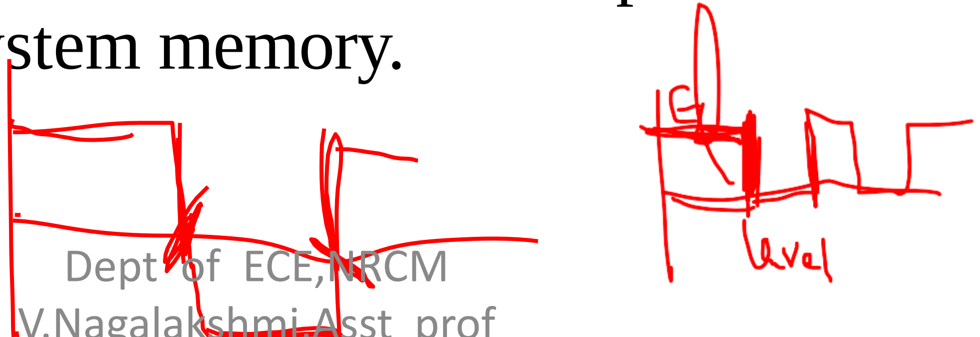
- This signal is used to read data from memory or I/O device which reside on the 8086 local bus.

Ready

- If this signal is low the 8086 enters into WAIT state.
- The READY signal from memory/ IO is synchronized by the 8284A clock generator to form READY.
- This signal is active HIGH.

Interrupt Request (INTR)

- It is a level triggered maskable interrupt request.
- A subroutine is vectored via an interrupt vector lookup table located in system memory.



TEST

- This input is examined by the “Wait” instruction.
- If the TEST input is LOW execution continues,
- otherwise the processor waits in an ``Idle” state.

Non-Maskable Interrupt (NMI)

- It is an edge triggered input which causes a type 2 interrupt.
- NMI is not maskable internally by software.

Reset

- This signal is used to reset the 8086.
- It causes the processor to immediately terminate its present activity.
- The signal must be active HIGH for at least four clock cycles.
- It restarts execution when RESET returns LOW.

Clock (CLK)

- This signal provides the basic timing for the processor and bus controller.
- The clock frequency may be 5 MHz or 8 MHz or 10 MHz depending on the version of 8086.

V_{CC}

- It is a +5V power supply pin.

Ground (GND)

- Two pins (1 and 20) are connected to ground ie, 0 V power supply.

Minimum/Maximum (MN/ MX)

- This pin indicates what mode the processor is to operate in.

MEMORY / IO (M/ IO)

- It is used to distinguish a memory access from an I/O access. M = HIGH, I/O = LOW.

WRITE(WR)

- It indicates that the processor is performing a write memory or write I/O cycle, depending on the state of the M/ IO signal.
- **Interrupt Acknowledge (INTA)** This signal indicates recognition of an interrupt request. It is used as a read strobe for interrupt acknowledge cycles.

Address Latch Enable (ALE)

- This signal is used to demultiplex the AD_0-AD_{15} into A_0-A_{15} and D_0-D_{15}. It is a HIGH pulse active during T1 of any bus cycle.

Data Enable(DEN)

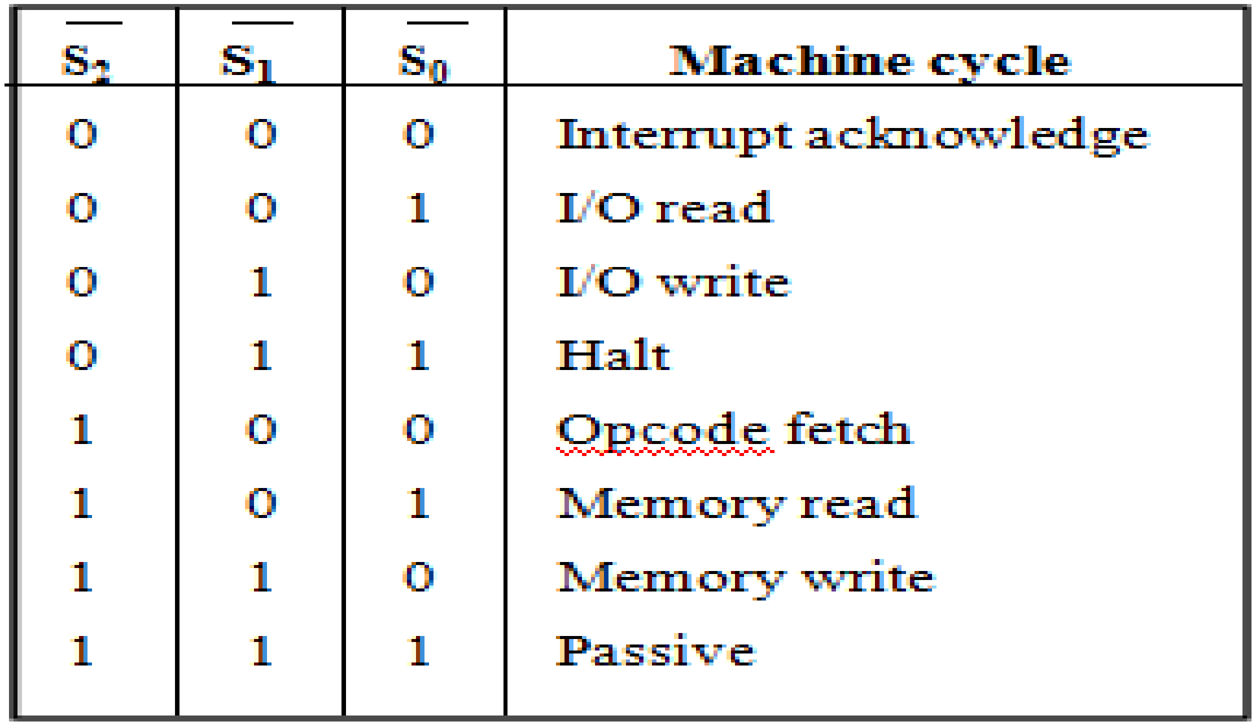
This signal informs the transceivers (8286/8287) that the 8086 is ready to send or receive data.

Hold

- This signal indicates that another master (DMA or processor) is requesting the host 8086 to handover the system bus.

Hold Acknowledge (HLDA)

- On receiving HOLD signal 8086 outputs HLDA signal HIGH as an acknowledgement.



$\overline{S_2}$	$\overline{S_1}$	$\overline{S_0}$	Machine cycle
0	0	0	Interrupt acknowledge
0	0	1	I/O read
0	1	0	I/O write
0	1	1	Halt
1	0	0	Opcode fetch
1	0	1	Memory read
1	1	0	Memory write
1	1	1	Passive

Request/Grant (RQ / GT_0 , RQ / GT_1)

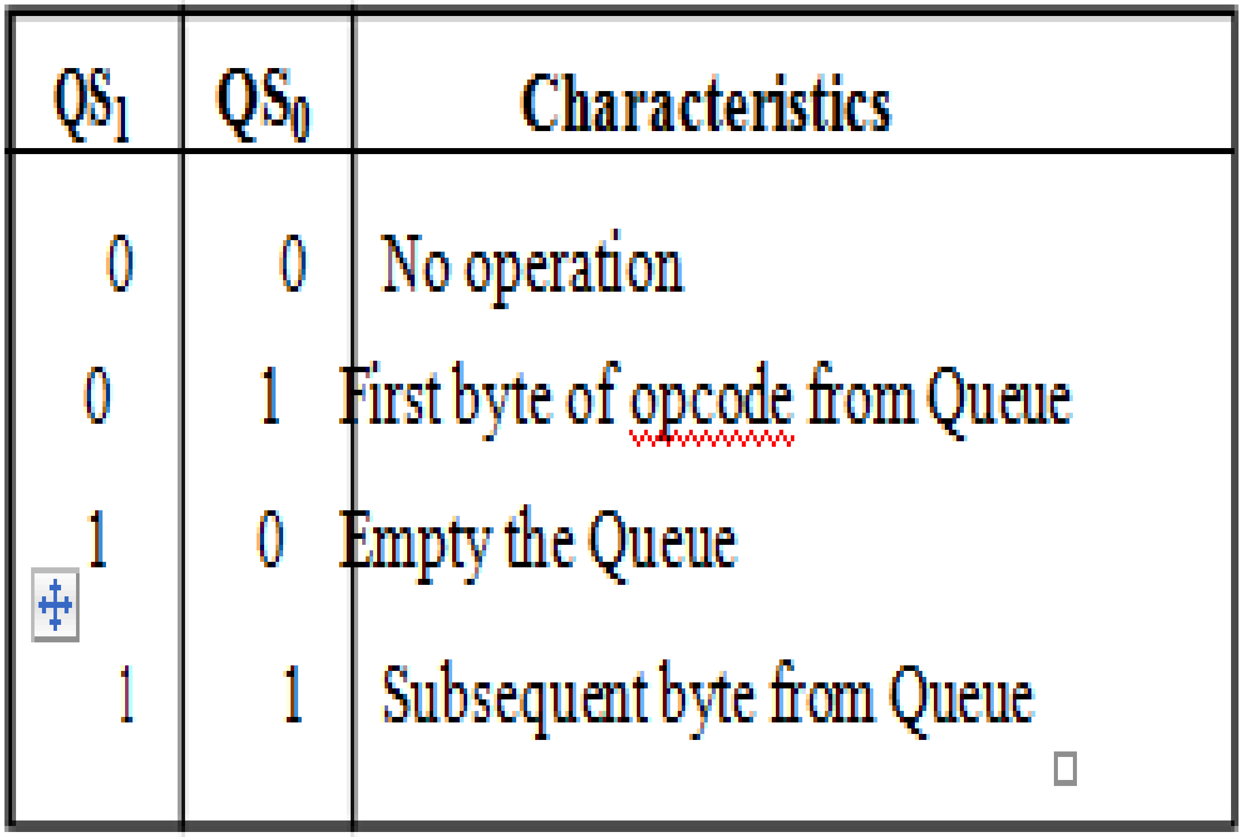
- These pins are used by other local bus masters to force RQ / GT_1 the processor to release the local bus at the end of the processor’s current bus cycle

LOCK

- This signal indicates that other system bus masters are not to gain control of the system bus while LOCK is active LOW.
- The LOCK signal is activated by the “LOCK” prefix instruction and remains active until the completion of the next instruction.

QUEUEue Status (QS_1, QS_0)

- The queue status is valid during the CLK cycle after which the queue operation is performed.



QS_1	QS_0	Characteristics
0	0	No operation
0	1	First byte of opcode from Queue
1	0	Empty the Queue
1	1	Subsequent byte from Queue

SYSTEM BUS STRUCTURE

- System bus is a single computer bus that connects the major components of a computer system.
- It consists of data bus, address bus and control bus.
- To communicate with external world, microprocessor make use of buses.

DATA BUS

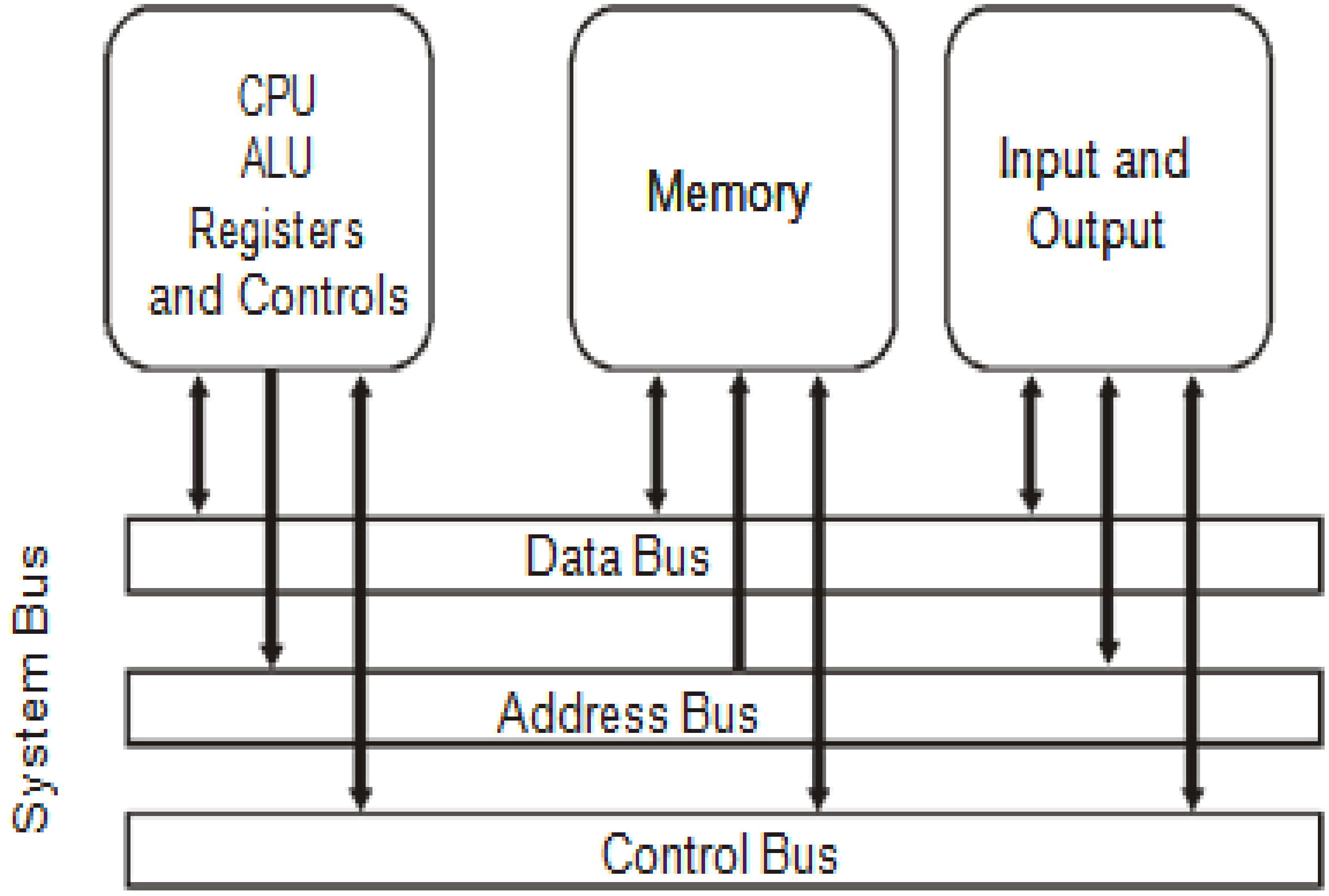
- It is used for the exchange of data between the processor, memory and peripherals.
- It is bi-directional so that it allows data flow in both directions.
- The width of the data bus can differ for every microprocessor.
- When the microprocessor issues the address of the instruction, it gets back the instruction through the data bus.

ADDRESS BUS

- The address bus contains the connections between the microprocessor and memory or output devices
- It is unidirectional.
- The width of the address bus corresponds to the maximum addressing capacity

CONTROL BUS

- The control bus carries the signals relating to the control and coordination of the various activities across the computer, which can be sent from the control unit within the CPU.
- Microprocessor uses control bus to process data, that is what to do with the selected memory location.


CPU
ALU
Registers
and Controls
Memory
Input and
Output
Data Bus
Address Bus
Control Bus
System Bus

MIN-MAX MODE OF OPERATION

Intel 8086 has two modes of operation. They are:

- Minimum mode
- Maximum mode

- When only 8086 microprocessor is to be used in a microcomputer system, the 8086 is used in the **minimum mode** of operation.
- In this mode, the microprocessor issues the control signals required by memory or I/O devices.
- In a multiprocessor system it operates in the **maximum mode**. In this mode, the control signals are issued by Intel 8288 bus controller.

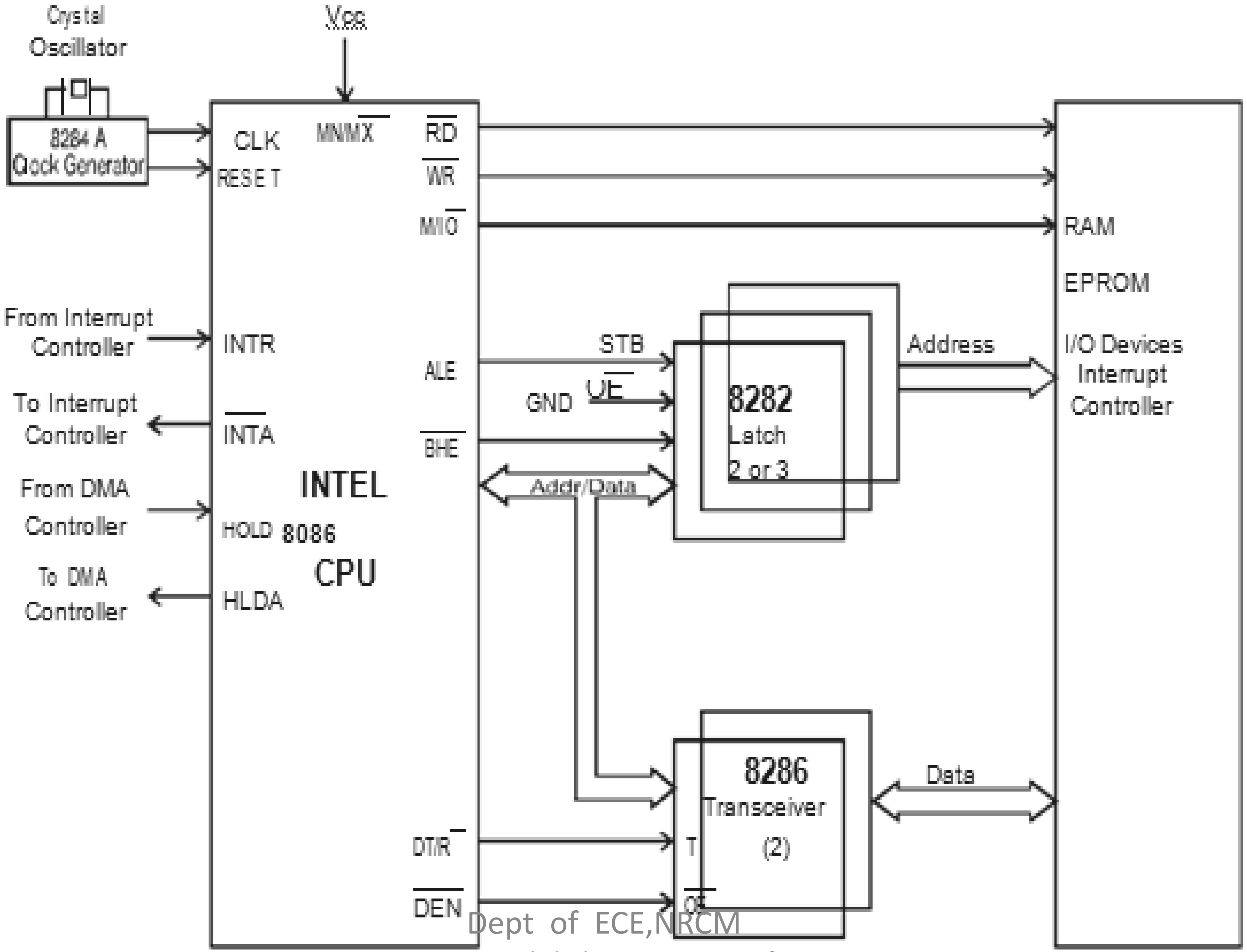
- The pin MN/ MX (33) decides the operating mode of 8086.
- When MN/ MX = 0, maximum mode of operation.

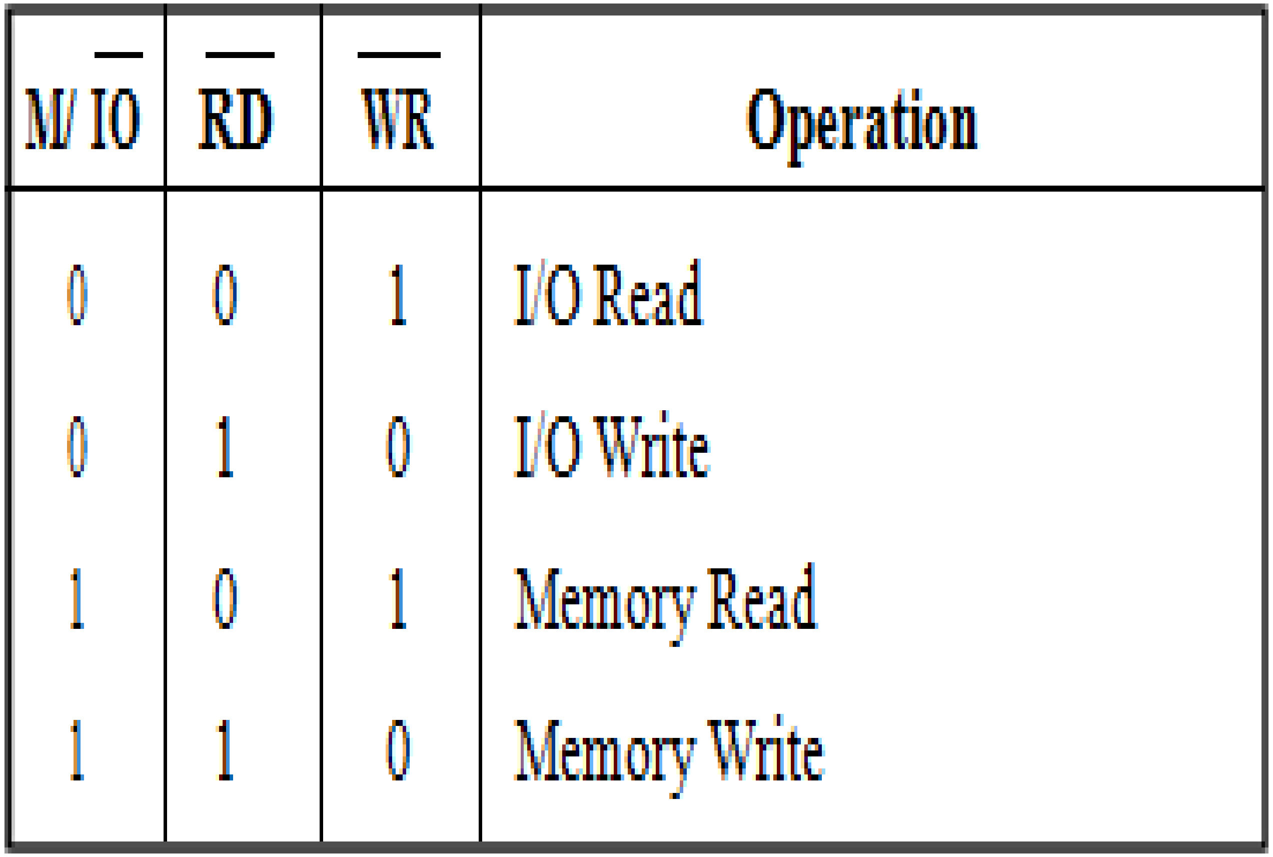
  = 1, minimum mode of operation.
- Pins 24 to 31 have different functions for minimum mode and maximum mode.

Minimum Mode

- For minimum mode of operation MN/ MX is connected to **V**$_{CC}$ (+5 volts).

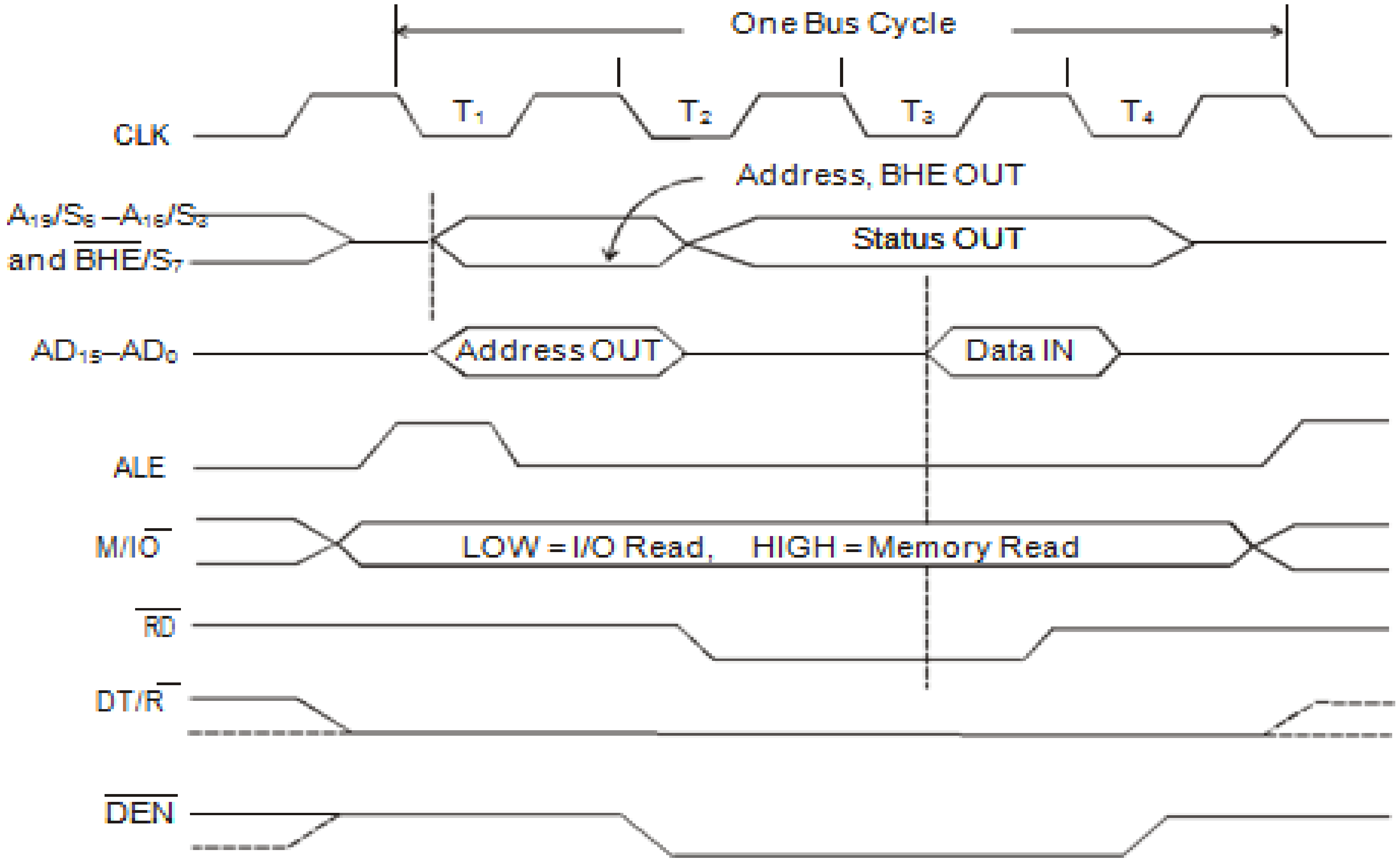
- All control signals for controlling memory and I/O devices are generated inside the 8086 microprocessor.
- In this mode , peripheral devices can be used with the microprocessor without any special consideration


Crystal Oscillator
Vcc
8284 A Clock Generator
CLK
RESET
MN/$\overline{MX}$
$\overline{RD}$
$\overline{WR}$
M/$\overline{IO}$
RAM
EPROM
I/O Devices
Interrupt Controller
From Interrupt Controller
INTR
To Interrupt Controller
$\overline{INTA}$
From DMA Controller
HOLD
To DMA Controller
HLDA
INTEL 8086 CPU
ALE
STB
GND
$\overline{OE}$
$\overline{BHE}$
Addr/Data
8282 Latch 2 or 3
Address
8286 Transceiver (2)
Data
DT/$\overline{R}$
T
$\overline{DEN}$
$\overline{OE}$

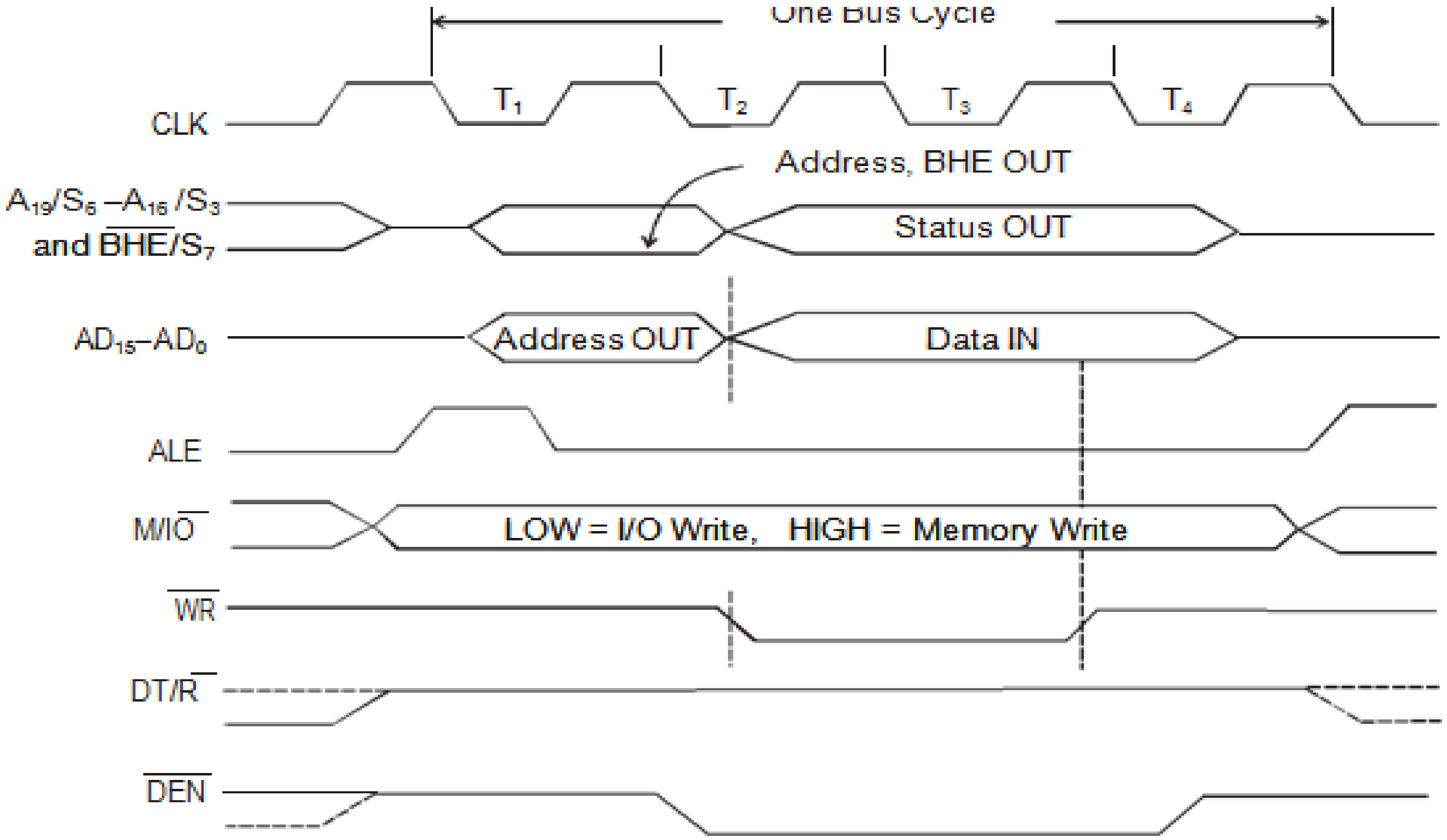


$M/\overline{IO}$	$\overline{RD}$	$\overline{WR}$	Operation
0	0	1	I/O Read
0	1	0	I/O Write
1	0	1	Memory Read
1	1	0	Memory Write

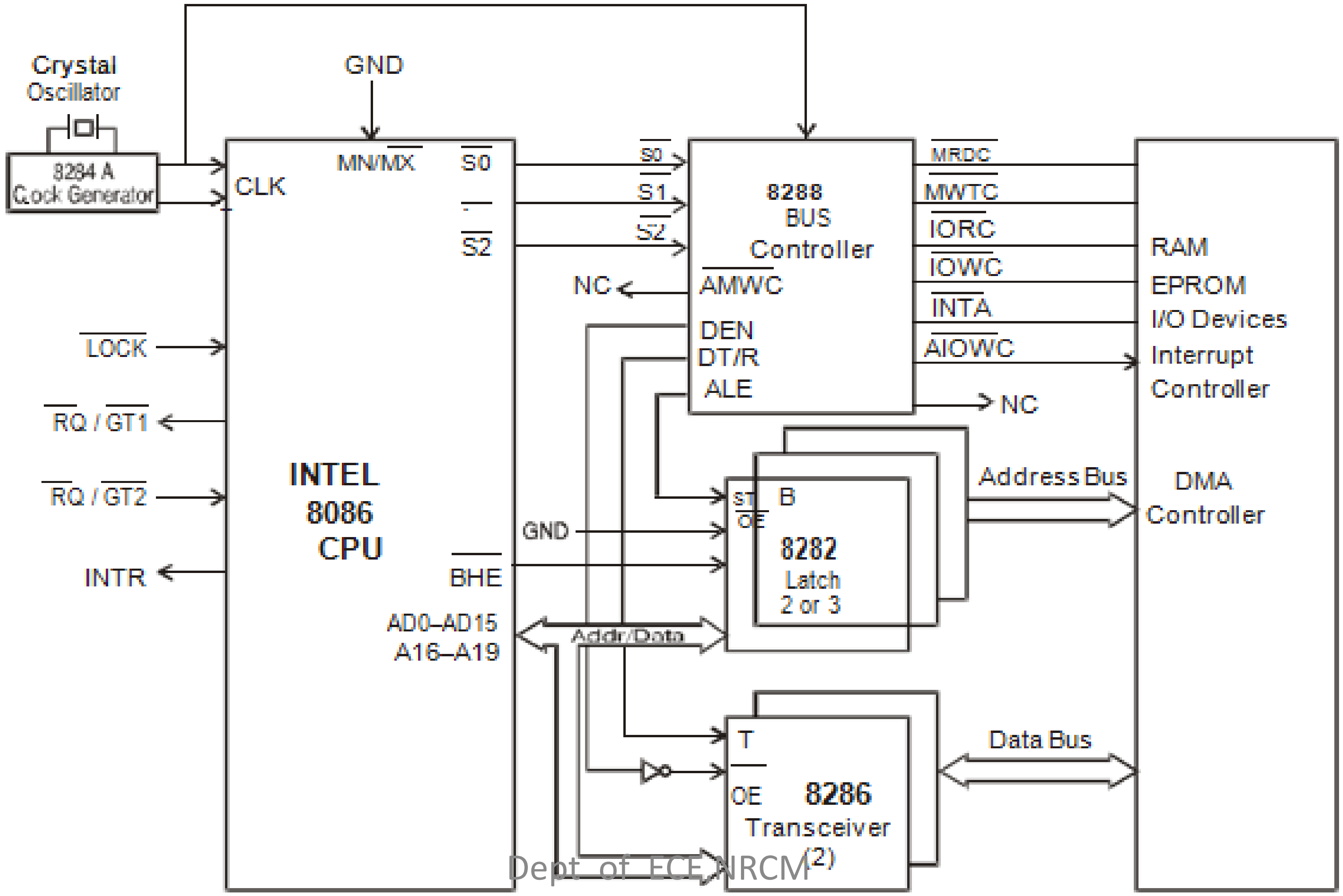
READ CYCLE


One Bus Cycle
CLK
T1
T2
T3
T4
Address, BHE OUT
A19/S6 – A16/S3 and BHE/S7
Status OUT
AD15–AD0
Address OUT
Data IN
ALE
M/IO
LOW = I/O Read, HIGH = Memory Read
RD
DT/R
DEN

WRITE CYCLE


One Bus Cycle
T1
T2
T3
T4
CLK
Address, BHE OUT
A19/S6 –A16 /S3
and BHE/S7
Status OUT
AD15–AD0
Address OUT
Data IN
ALE
M/IO
LOW = I/O Write, HIGH = Memory Write
WR
DT/R
DEN

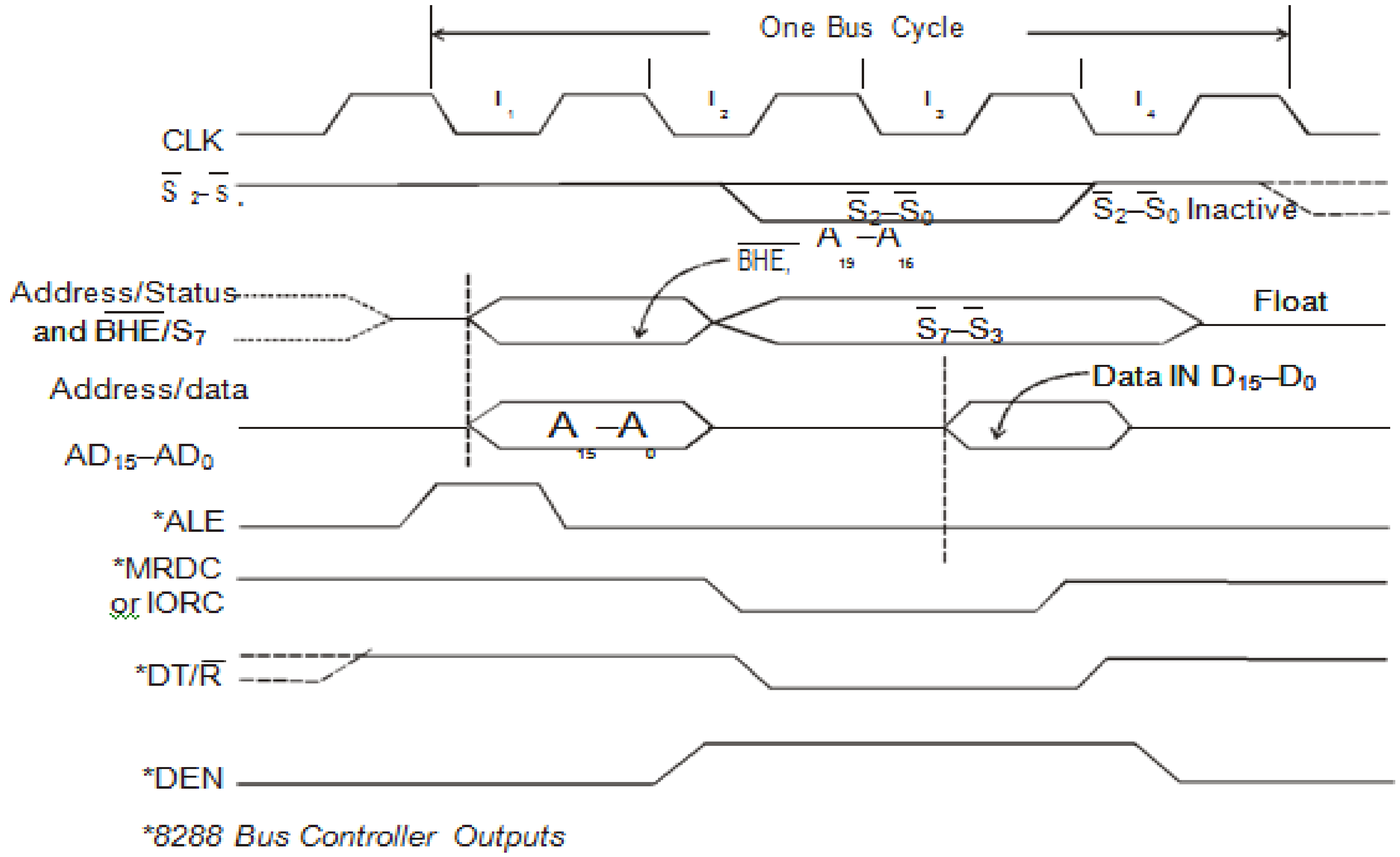
Maximum mode operation’



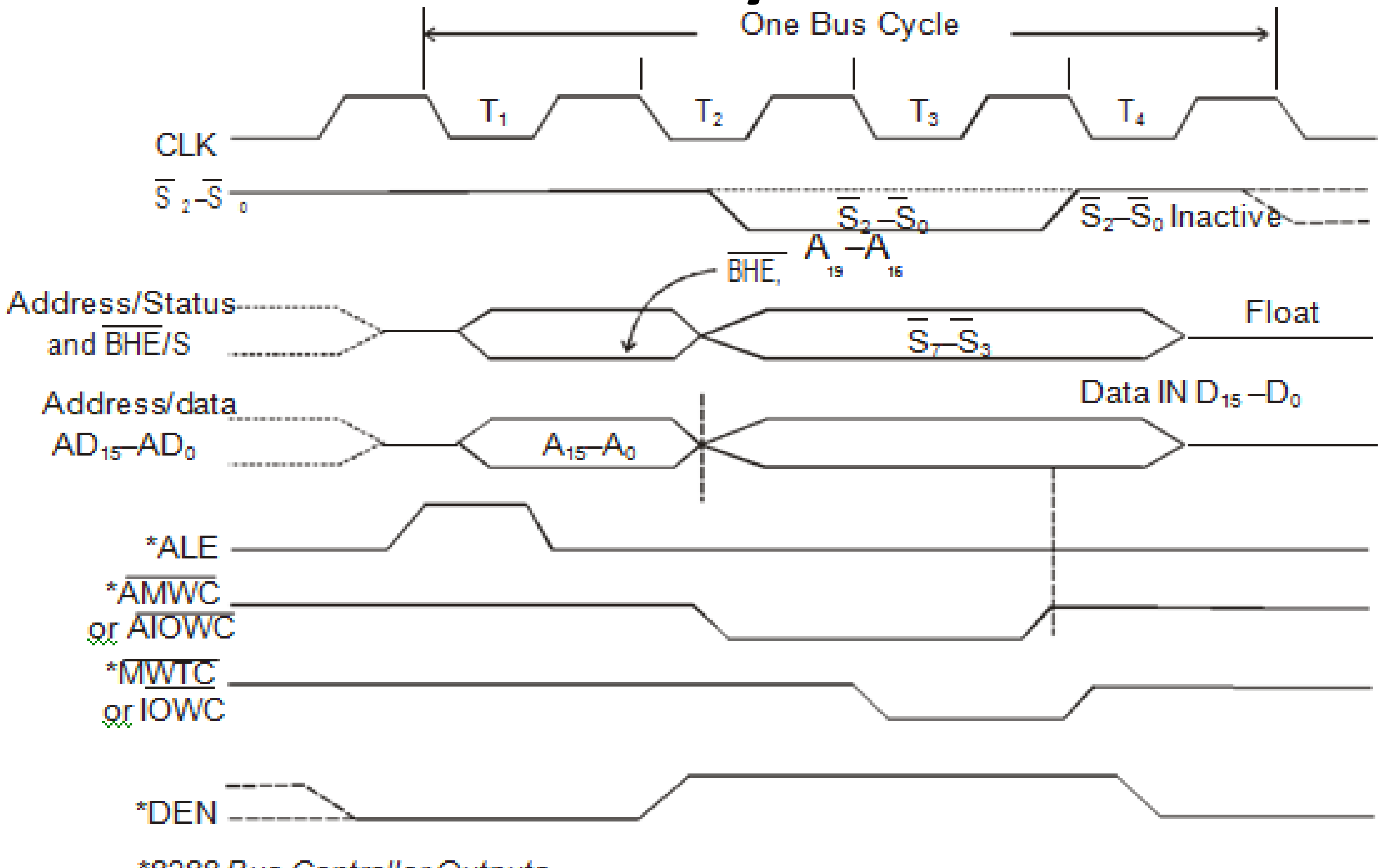
- In maximum mode 8086 based system, an external Bus Controller (Intel 8288) has to be employed to generate the bus control signals.
- The important signals are :
- MRDC - Memory Read Command
  MWTC - Memory Write Command
  IORC - I/O Read Command
  IOWC - I/O Write Command
  AMWC - Advanced Memory Write Command
  AIOWC - Advanced I/O Write Command

- Three numbers of 8 bit latches (Intel 8282) are employed to demultiplex the address lines.
- The latches are enabled by using the ALE signal generated by the bus controller.
- Two numbers of octal bus transceivers (Intel 8286) are used as data transceivers.
- The signals DEN and DT/ R are generated by the bus controller are used as enable and direction control respectively.
- The clock generator (Intel 8284) is used to generate clock, reset and ready signals for 8086.
- A quartz crystal of frequency 15 MHz is connected to 8284.

Read cycle


One Bus Cycle
T1
T2
T3
T4
CLK
$\overline{S}_2$–$\overline{S}_0$
$\overline{S}_2$–$\overline{S}_0$
$\overline{S}_2$–$\overline{S}_0$ Inactive
$\overline{BHE}$, A_{19}–A_{16}
Address/Status and $\overline{BHE}$/S_7
$\overline{S}_7$–$\overline{S}_3$
Float
Address/data AD_{15}–AD_0
A_{15}–A_0
Data IN D_{15}–D_0
*ALE
*MRDC or IORC
*DT/$\overline{R}$
*DEN
*8288 Bus Controller Outputs

Write cycle



*8288 Bus Controller Outputs

SYSTEM DESIGN USING 8086

The specification of the system includes the following:

- I/O devices
- Memory requirement
- System clock frequency
- Peripheral devices required
- Application

I/O devices

Input devices : 8279 – keyboard and display controller

The popular output devices are,

- LED display
- LCD
- Printer
- Floppy disk / CD
- CRT terminal

Memory requirement

- The memory of the system is splitted between EPROM and RAM.
- The popular EPROM used in 8086 based system are 2708 (1K x 8), 2716 (2K x 8), 2732 (4K x 8), 2764 (8K x 8) and 27256 (32K x 8).
- The popular static RAM used in 8086 based system are 6208 (1K x 8), 6216 (2K x 8), 6232 (4K x 8), 6264 (8K x 8) and 62256 (32 K x 8).

System clock frequency

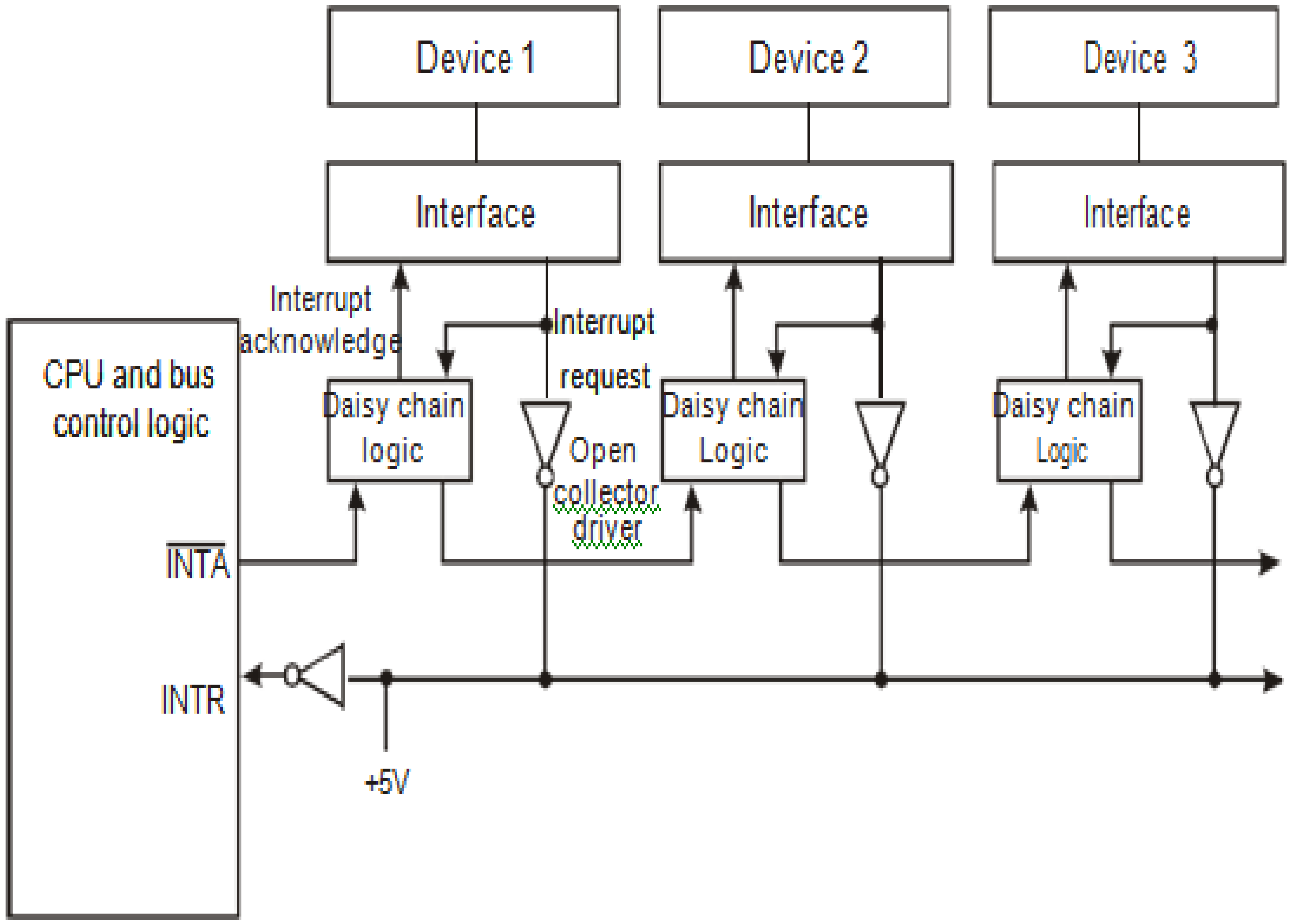
- The 8086 does not have an internal clock circuit. Hence clock has to be supplied from an external device.
- The Intel 8284 clock generator is employed to generate the clock.
- An external quartz crystal has to be connected to 8284 to generate the clock signal.

Peripheral devices

- Intel 8253 - Programmable Interval Timer
- Intel 8251 - USART
- Intel 8255 - Programmable Peripheral Interface
- Intel 8279 - Keyboard / Display controller
- Intel 8257 - DMA controller
- ADC, DAC etc.

Application

- The specifications of the microprocessor itself depends on the applications for the proposed system and the nature of work.
- The I/O device, memory, peripheral device are all depends on the nature of work to be performed by the system.


Device 1
Device 2
Device 3
Interface
Interface
Interface
Interrupt acknowledge
Interrupt request
Daisy chain logic
Daisy chain Logic
Daisy chain Logic
Open collector driver
CPU and bus control logic
$\overline{\text{INTA}}$
INTR
+5V

Interrupt priority management hardware

- By designing a programmable interrupt priority management circuit and bus control logic.
- The duty is placed on the requesting device to request the interrupt and identify itself.
- The identity could be a branching address .
- If the device just supplies an identification number, this can be used in conjunction with a lookup table to determine the address of the required service routine.

Direct Memory Access Block Transfer

- A DMA controller allows devices to transfer data to or from the system's memory without the intervention of the processor.

- Components connected to the system bus is given control of the bus.

- This component is said to be the master during that cycle and the component it is communicating with is said to be the slave.

- Taking control of the bus for a bus cycle is called **cycle stealing.**
- The interface sends the DMA controller a request for DMA service.
- A Bus request is made to the HOLD pin (active High) on the 8086 microprocessor and the controller gains control of the bus.

- A Bus grant is returned to the DMA controller from the Hold Acknowledge (HLDA) pin (active High) on the 8086 microprocessor.
- The DMA controller places contents of the address register onto the address bus.
- The controller sends the interface a DMA acknowledgment, which tells the interface to put data on the data bus.

- The data byte is transferred to the memory location indicated by the address bus.
- The interface latches the data.
- The Bus request is dropped, the HOLD pin goes Low, and the controller relinquishes the bus.

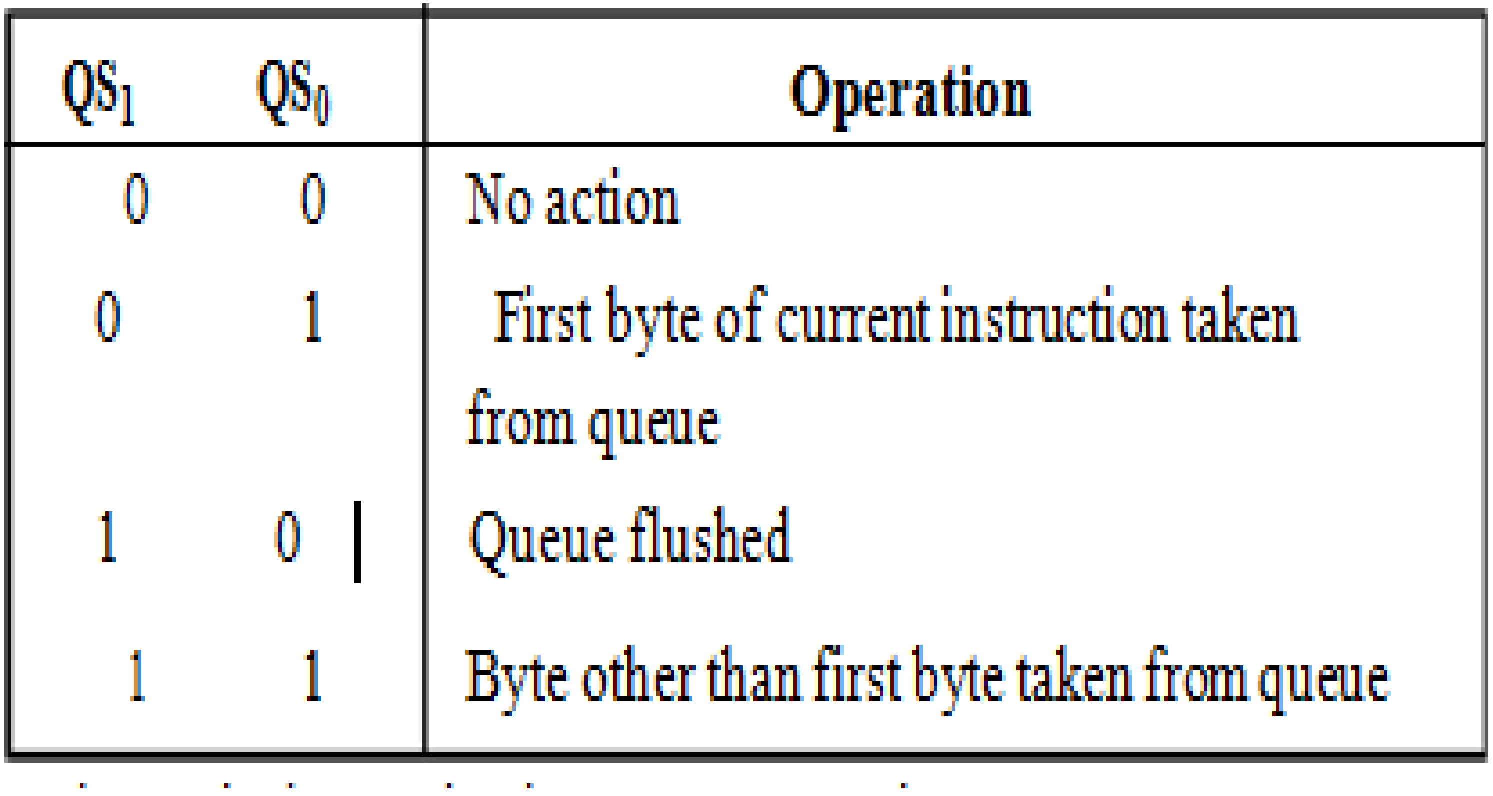
- The Bus grant from the 8086 microprocessor is dropped and the HLDA pin goes Low.
- The address register is incremented by 1.
- The byte count is decremented by 1.
- If the byte count is non-zero, return to step 1, otherwise stop.

MULTIPROGRAMMING

- Multiprogramming can execute several jobs concurrently by switching the attention of the CPU back and forth among them.

- Multiprogramming enable the CPU to be utilized more efficiently. If the operating system can quickly switch the CPU to another task

- The **8086 fetches** the instructions.
- The **coprocessor monitors** the instruction sequence and captures its **own** instructions.
- The **ESC** is decoded by the CPU and coprocessor simultaneously.
- The CPU computes the 20 bit address of memory operand and does a dummy read. The coprocessor captures the address of the data and obtains control of the bus to load or store as needed.
- The **coprocessor** sends **BUSY (high)** to the TEST pin.
- The CPU goes to the next instruction and if this is an 8086 instruction, the CPU and coprocessor execute in parallel.
- If another coprocessor instruction occurs, the 8086 must wait until BUSY goes low ie, TEST pin become active. To implement this, a WAIT instruction is put in front of most 8087 instructions by the Assembler.
- The WAIT instruction does the operations ie, wait until the TEST pin is active.

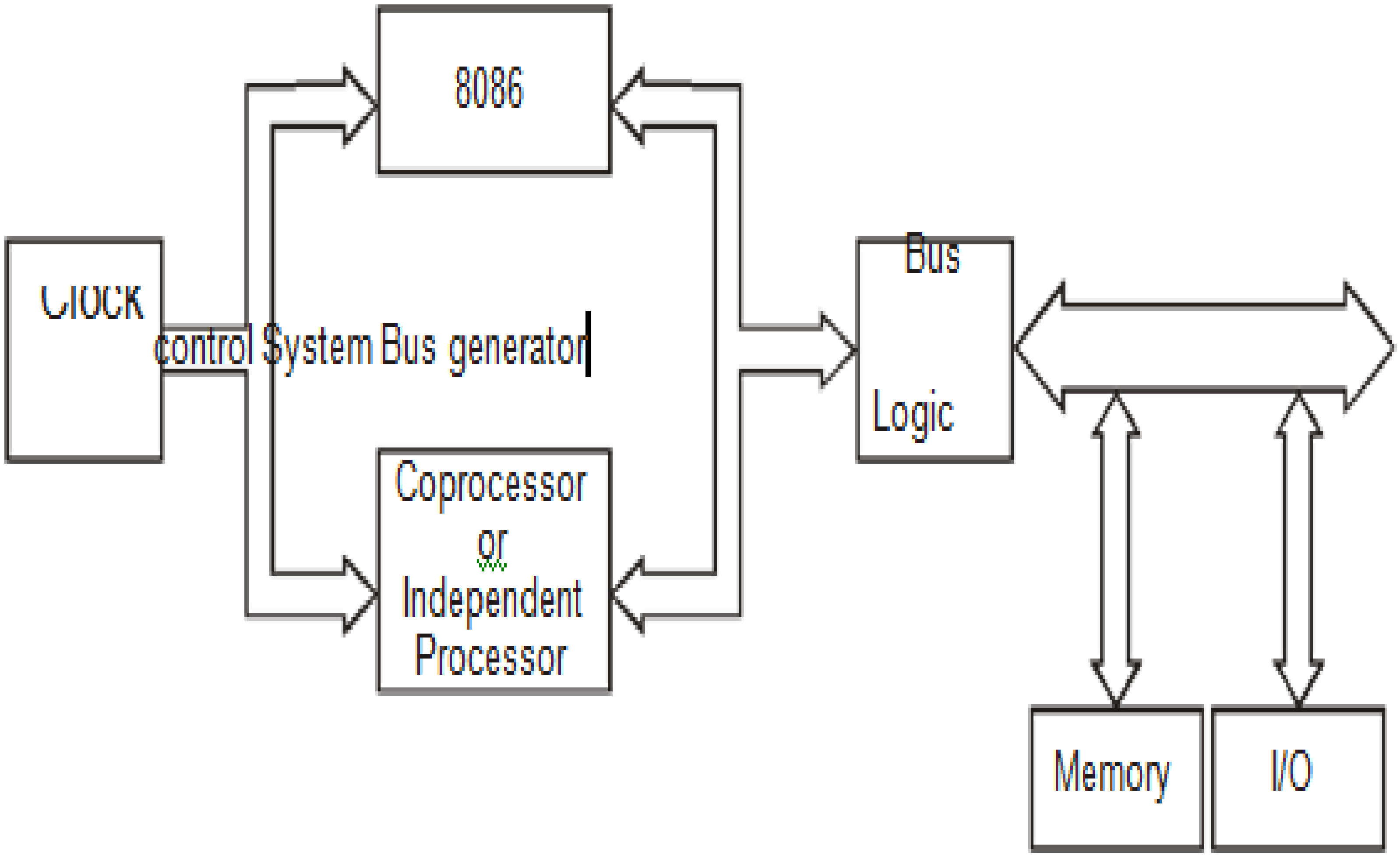


QS_1	QS_0	Operation
0	0	No action
0	1	First byte of current instruction taken from queue
1	0	Queue flushed
1	1	Byte other than first byte taken from queue

CLOSELY COUPLED CONFIGURATION

Share :

- Memory
- I/O system
- Bus and Bus control logic
- Clock generator


8086
Clock
control System Bus generator
Bus
Logic
Coprocessor
or
Independent
Processor
Memory
I/O

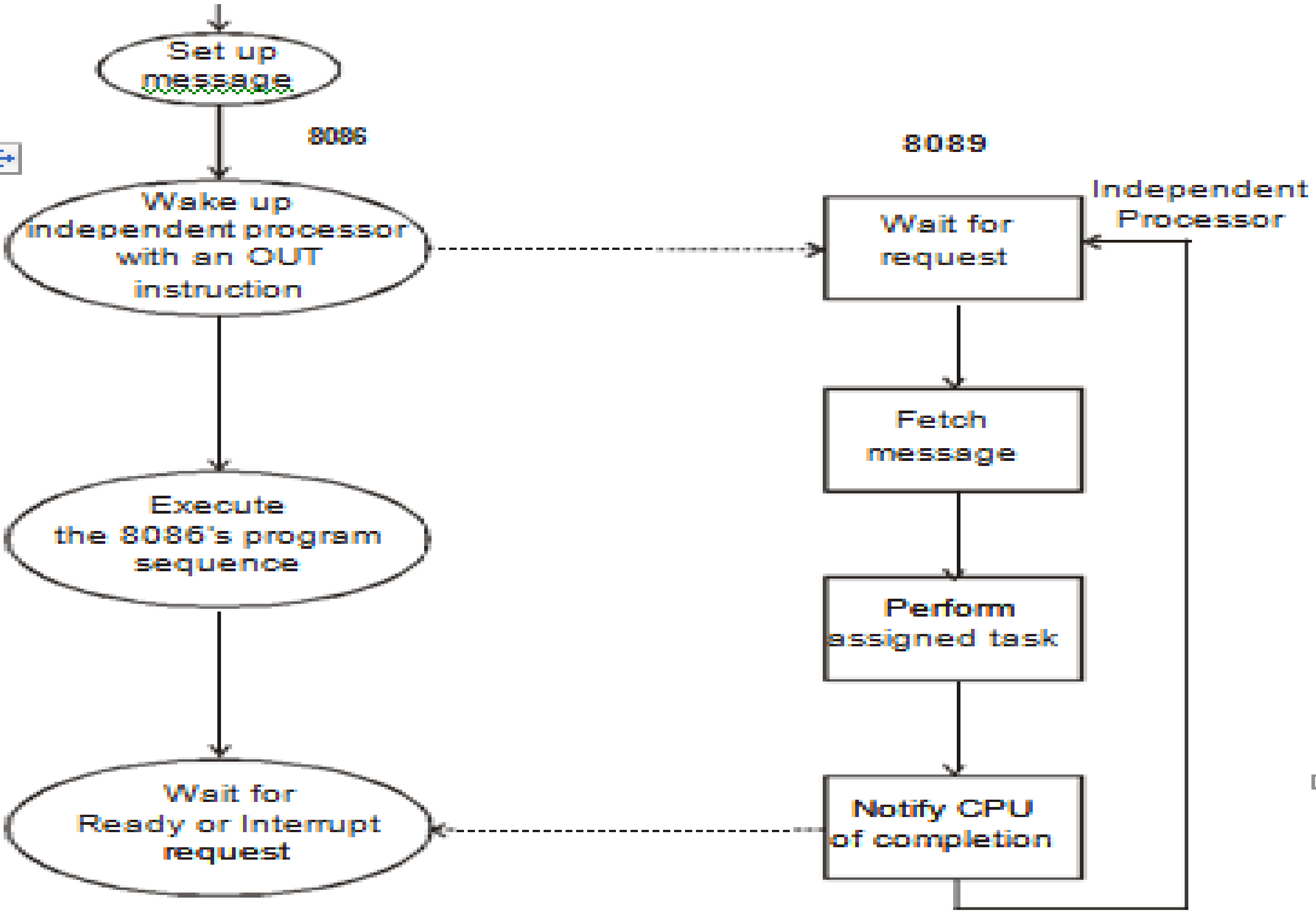
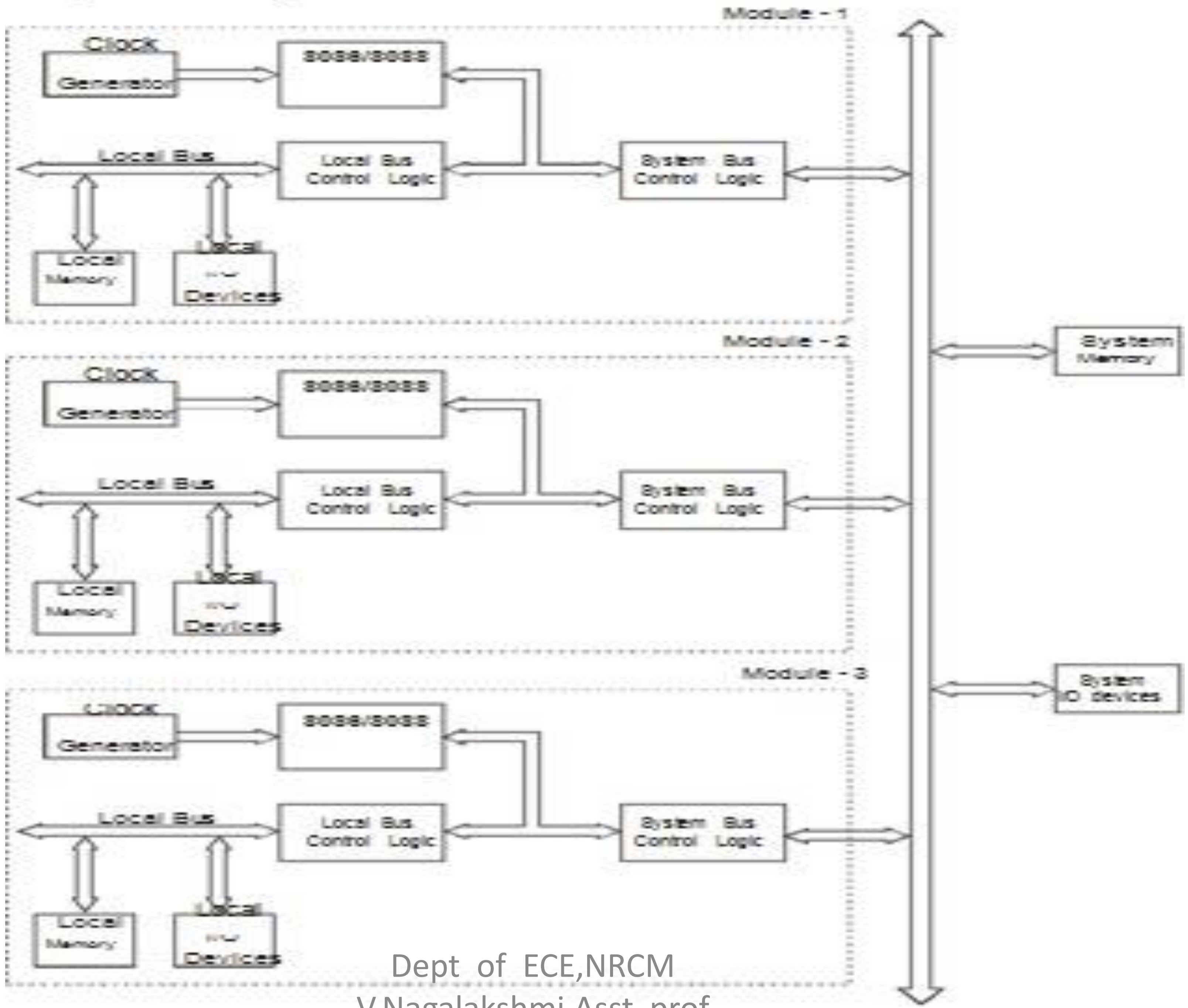

Set up message
8086
8089
Wake up independent processor with an OUT instruction
Wait for request
Independent Processor
Fetch message
Execute the 8086's program sequence
Perform assigned task
Wait for Ready or Interrupt request
Notify CPU of completion

Fig. 2.19. Interaction between 8086 and 8089

LOOSELY COUPLED CONFIGURATION

- In loosely coupled configuration a number of modules of 8086 can be interfaced through a common system bus to work as a multiprocessor.
- Each module has an independent microprocessor based system with its own clock source, and its own memory and I/O devices interfaced through a local bus.
- Each module can also be a closely coupled configuration of a processor or coprocessor.


Module - 1
Clock Generator
8086/8088
Local Bus
Local Bus Control Logic
System Bus Control Logic
Local Memory
Local I/O Devices
Module - 2
Clock Generator
8086/8088
Local Bus
Local Bus Control Logic
System Bus Control Logic
Local Memory
Local I/O Devices
Module - 3
Clock Generator
8086/8088
Local Bus
Local Bus Control Logic
System Bus Control Logic
Local Memory
Local I/O Devices
System Memory
System IO devices

Advantages

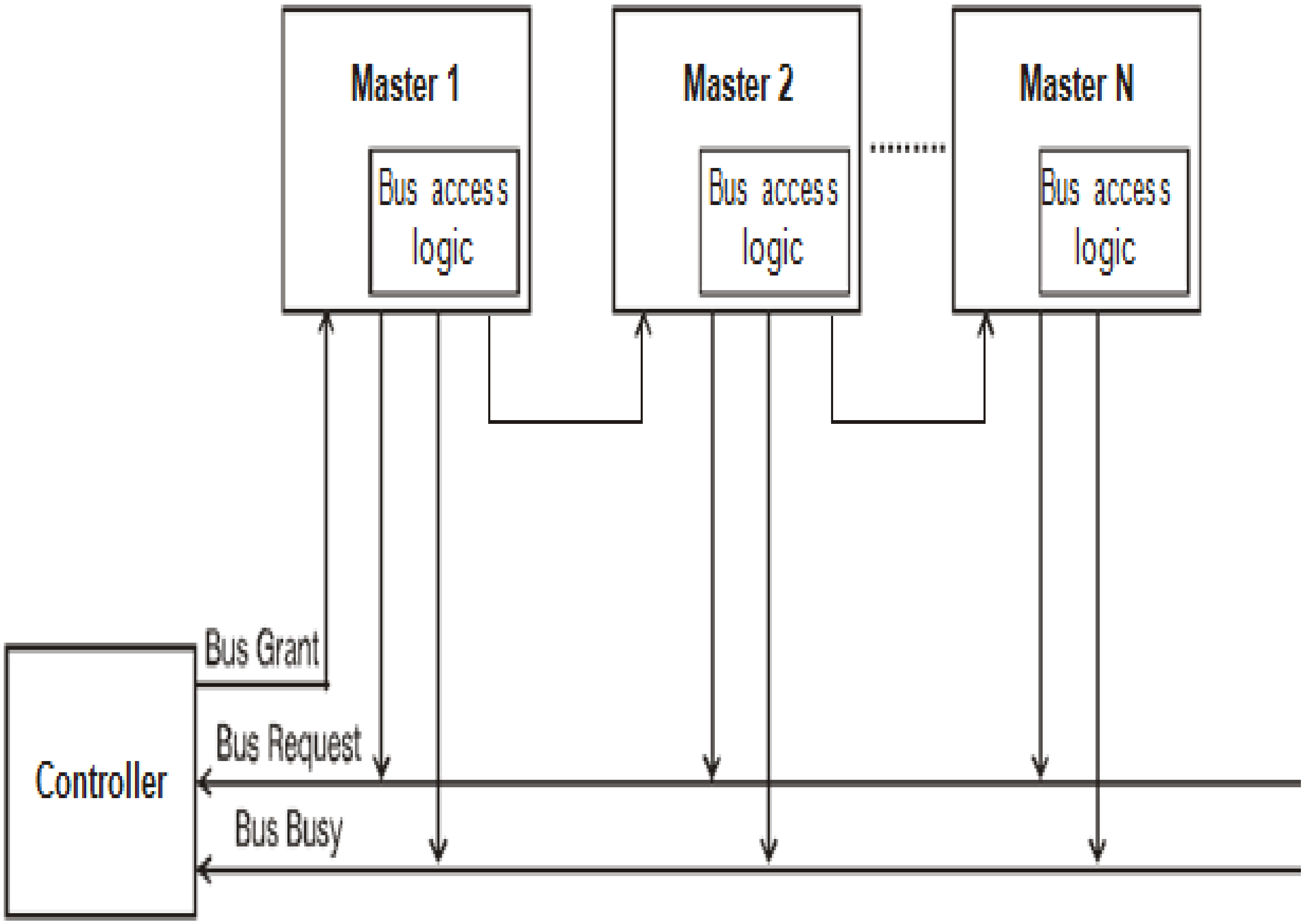
- Better system throughput by having more than one processor.
- The system can be expanded in modular form.
- A failure in one module normally does not affect the breakdown of the entire system and faulty module can be easily detected and replaced.

Bus allocation schemes

- Daisy chaining
- Polling method
- Independent Priority

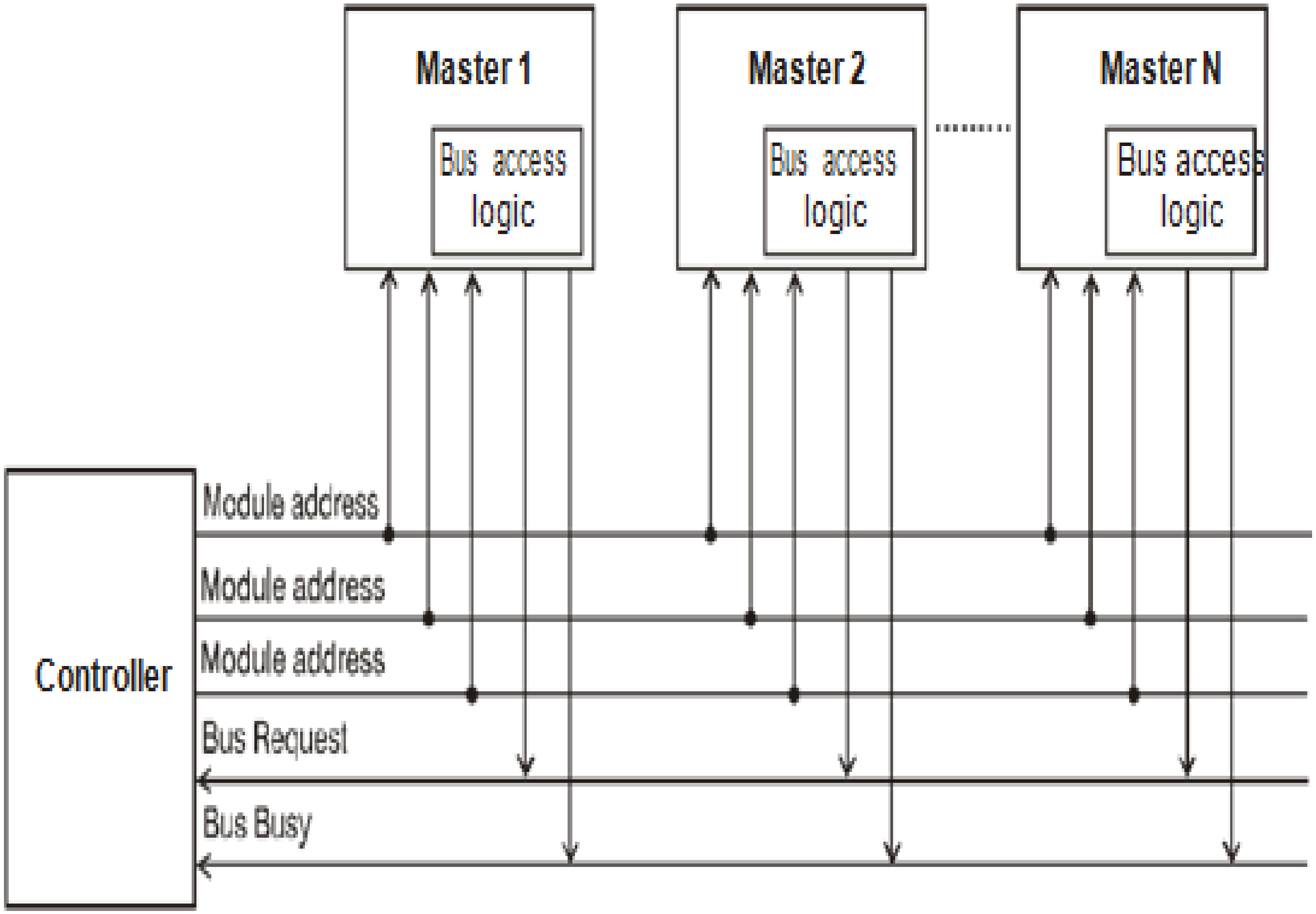
DAISY CHAINING METHOD

- In daisy chaining method all masters make use of the same line for bus request.
- In response to a bus request, the controller sends a bus grant if the bus is free.
- The bus grant signal serially propagates through each master until it encounters the first one that is requesting access to the bus.
- This master blocks the propagation of the bus grant signal, activates the busy line and gains control of the bus.


Master 1
Bus access logic
Master 2
Bus access logic
Master N
Bus access logic
Bus Grant
Bus Request
Bus Busy
Controller

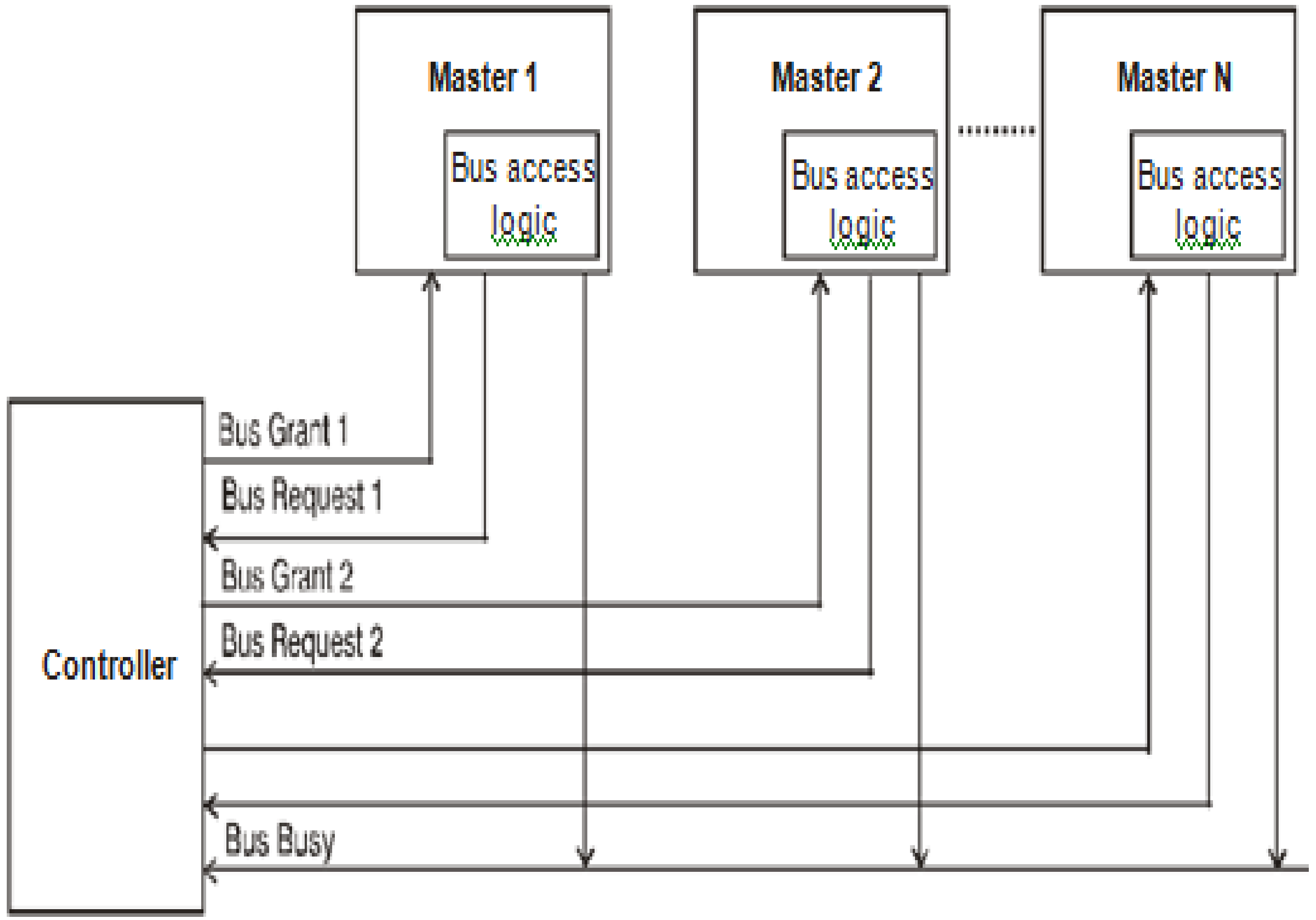
POLLING

- In polling method, the controller sends address of device to grant bus access.
- The number of address lines required is depend on the number of masters connected in the system.
- In response to a bus request, controller generates a sequence of master addresses.
- When the requesting master recognizes the address, it activates the busy line and begins to use the bus.
- The priority can be changed by altering the polling sequence stored in the controller.
- Another one advantage of this method is, if one module fails entire system does not fail.

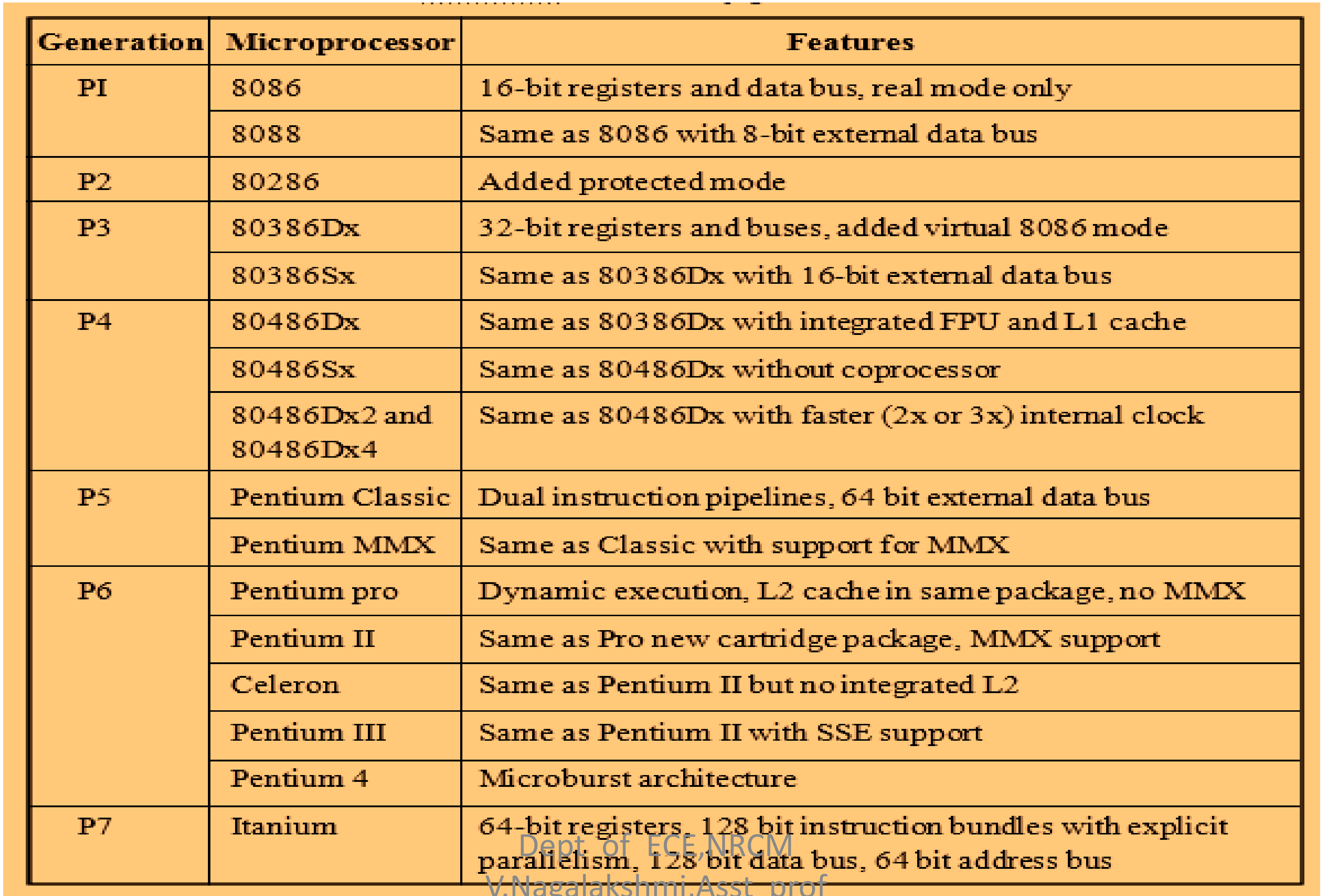

Master 1
Bus access logic
Master 2
Bus access logic
Master N
Bus access logic
Controller
Module address
Module address
Module address
Bus Request
Bus Busy

Independent priority

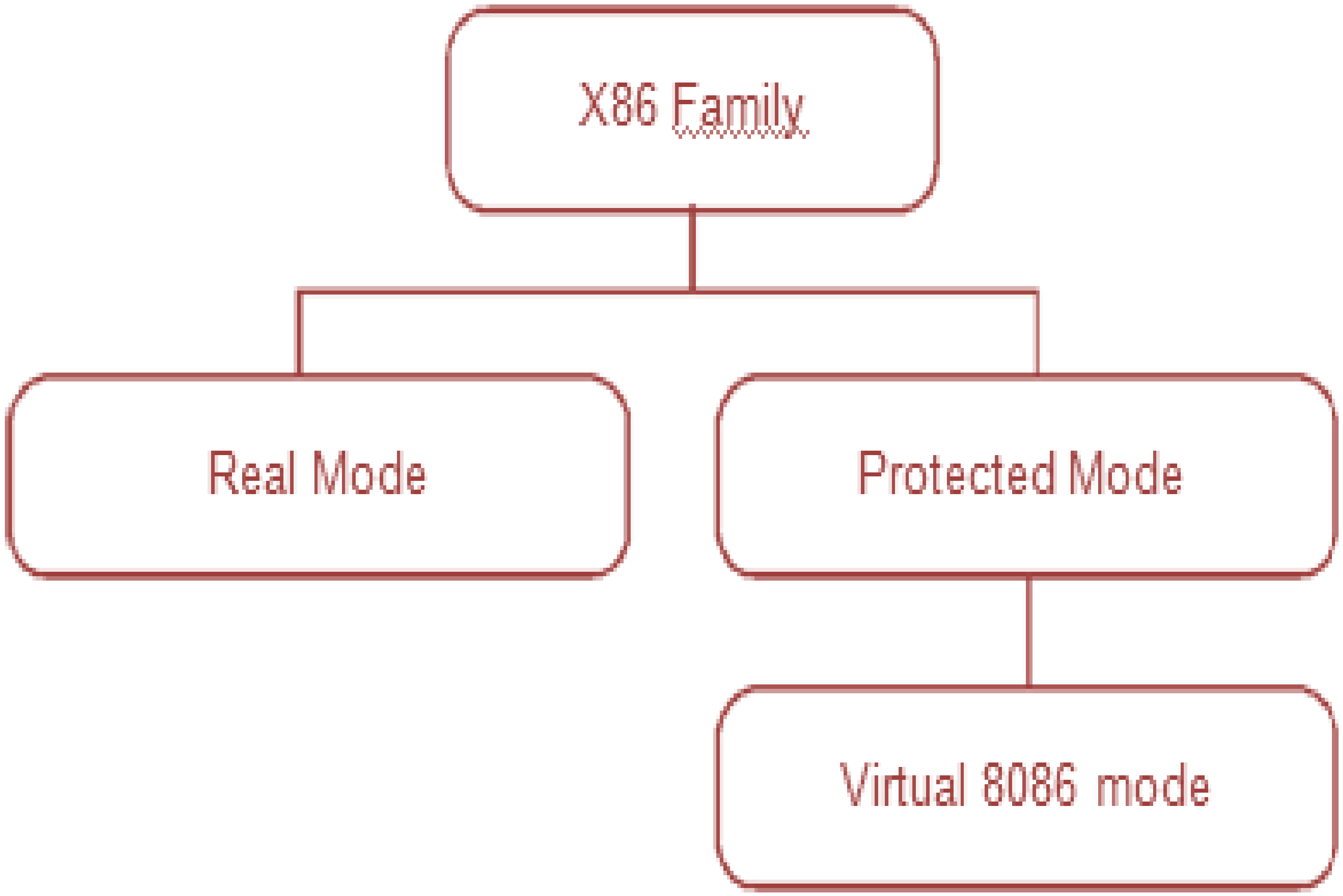
- Each master has a separate pair of bus request (BRQ) and bus grant (BGR) lines and each pair has a priority assigned to it.
- The built in priority decoder within the controller selects the highest priority request and asserts the corresponding bus grant signal.


Master 1
Bus access logic
Master 2
Bus access logic
Master N
Bus access logic
Controller
Bus Grant 1
Bus Request 1
Bus Grant 2
Bus Request 2
Bus Busy

ADVANCED PROCESSOR



Generation	Microprocessor	Features
PI	8086	16-bit registers and data bus, real mode only
	8088	Same as 8086 with 8-bit external data bus
P2	80286	Added protected mode
P3	80386Dx	32-bit registers and buses, added virtual 8086 mode
	80386Sx	Same as 80386Dx with 16-bit external data bus
P4	80486Dx	Same as 80386Dx with integrated FPU and L1 cache
	80486Sx	Same as 80486Dx without coprocessor
	80486Dx2 and 80486Dx4	Same as 80486Dx with faster (2x or 3x) internal clock
P5	Pentium Classic	Dual instruction pipelines, 64 bit external data bus
	Pentium MMX	Same as Classic with support for MMX
P6	Pentium pro	Dynamic execution, L2 cache in same package, no MMX
	Pentium II	Same as Pro new cartridge package, MMX support
	Celeron	Same as Pentium II but no integrated L2
	Pentium III	Same as Pentium II with SSE support
	Pentium 4	Microburst architecture
P7	Itanium	64-bit registers, 128 bit instruction bundles with explicit parallelism, 128 bit data bus, 64 bit address bus


X86 Family
Real Mode
Protected Mode
Virtual 8086 mode

- In **real mode**, the advanced processors, including the Pentium, **simply operate like very fast 8086**, with the associated **1 MB memory limit.**
- Real mode operation is automatically selected upon power-up.
- Pentium-based PC that boots up into **DOS** is operating in real mode.

- In **protected mode**, the full **4 GB** of memory is available to the processor.
- It supports for multitasking, virtual memory addressing, memory management, protection and control over the internal data and instruction cache.
- The **Windows operating system** runs in protected mode to take advantage of these improvements.

PENTIUM PROCESSOR

- The term ‘’Pentium processor’ refers to a family of microprocessors that share a common architecture and instruction set.
- The first Pentium processors were introduced in 1993.
- It runs at a clock frequency of either 60 or 66 MHz and has 3.1 million transistors.

The features of Pentium architecture are

- Improved instruction execution time
- Bus cycle pipelining
- Address parity .
- Internal parity checking
- Functional redundancy checking

FEATURES

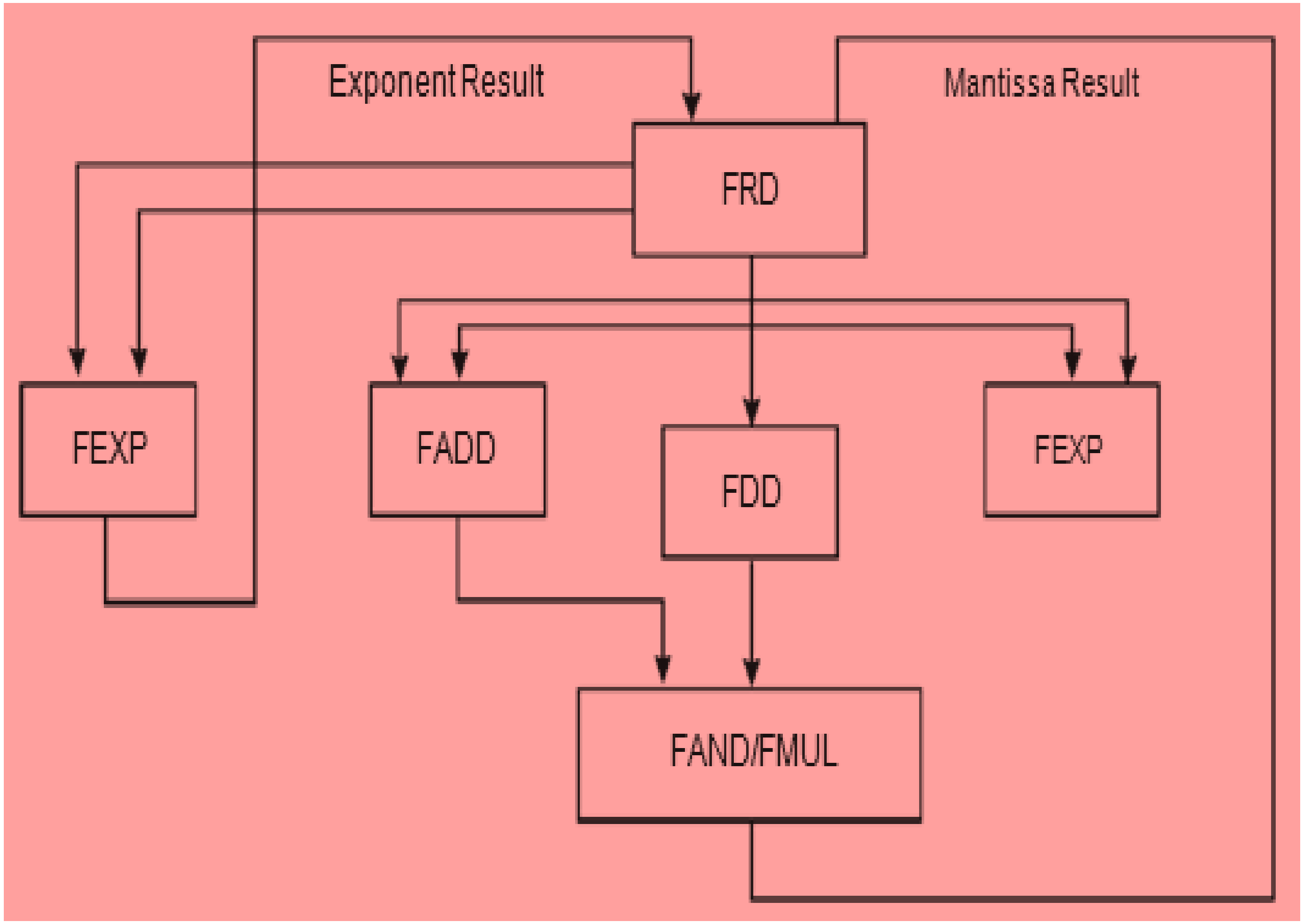
- ***Wider (64-bit) Data Bus:*** With its 64-bit-wide external data bus the Pentium processor can handle up to twice the data load of the Intel486 processor at the same clock frequency.
- ***Superscalar Architecture:*** Dual Instruction Pipeline
- ***Dynamic Branch Prediction Logic:*** The Pentium processor fetches the branch target instruction before it executes the branch instruction.
- ***Enhanced Floating Point Unit:*** The Pentium processor executes individual instructions faster through execution pipelining, which allows multiple floating point instructions to be executed at the same time.
- ***Dedicated Instruction and Data Cache:*** The Pentium processor has two separate 8 KB caches on chip-one for instructions and one for data.
- ***Write-Back MESI Protocol in Data Cache:*** When data is modified; only the data in the cache is changed.

STAGES OF PENTIUM PROCESSOR

- ***Pre-fetch/Fetch :*** Instructions are fetched from the instruction cache and aligned in pre-fetch buffers for decoding.
- ***Decode*1 *:*** Instructions are decoded into the Pentium's internal instruction format. Branch prediction also takes place at this stage.
- ***Decode*2 *:*** Same as above, and microcode ROM kicks in here, if necessary. Also, address computations take place at this stage.
- ***Execute :*** The integer hardware executes the instruction.
- ***Write-back :*** The results of the computation are written back to the register file.

FLOATING POINT UNIT

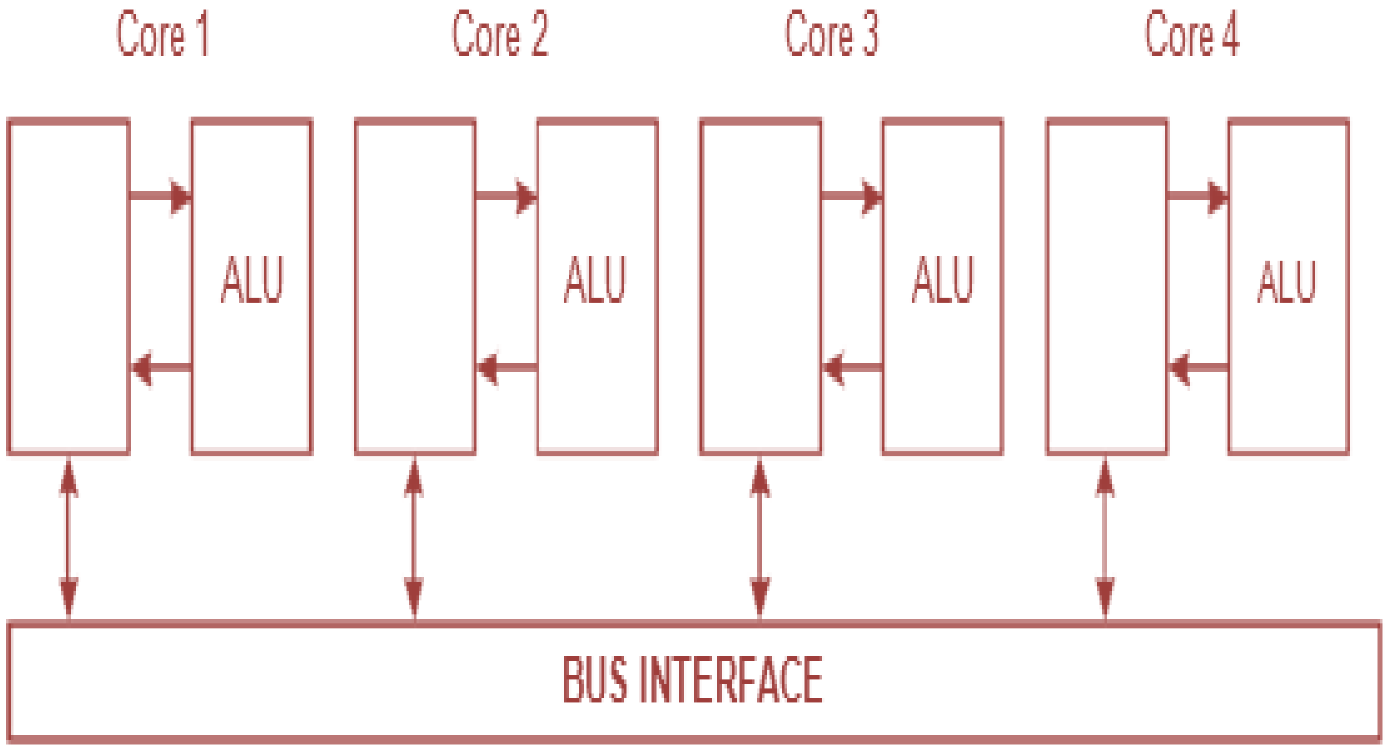
- There are 8 general-purpose 80-bit floating point registers.
- Floating point unit has 8 stages of pipelining. First five are similar to integer unit.
- Since the possibility of error is more in floating point unit (FPU) than in integer unit


Exponent Result
Mantissa Result
FRD
FEXP
FADD
FDD
FEXP
FAND/FMUL

Multi-core processor

- A multi-core processor is a single chip that contains more than one microprocessor core.
- Each core can simultaneously execute processor instructions in parallel.
- This effectively multiplies the processor's potential performance by the number of cores.

- Because the cores are physically close to each other, they can communicate with each other much faster than separate processors in a multiprocessor system.
- It improves overall system performance.


Core 1
Core 2
Core 3
Core 4
ALU
ALU
ALU
ALU
BUS INTERFACE

UNIT- 3

I/O INTERFACING

Memory Interfacing

- While executing a program, the microprocessor needs to access memory frequently to read instruction code and data stored in memory; the interfacing circuit enables that access.
- Memory has some signal requirements to write into and read from its registers.
- Similary, the microprocessor initiates a set of signals when it wants to read from and write into memory.

I/O INTERFACING

- The Input/Output devices such as keyboards and displays are the communication channels to the outside world.
- Latches and buffers are used for I/O interfacing. They once hardwired, perform only one function (either as input device if it is buffer and as output device if it is a latch). Thus limiting their capabilities.
- To improve the overall system performance the Intel has designed various programmable I/O devices.

- Some of the peripheral devices developed by Intel for 8085/8086/8088 based system are:
- 8255 - Parallel Communication Interface
- 8251 - Serial Communication Interface
- 8254 - Programmable Timer
- 8279 - Keyboard / Display Controller
- 8257 - DMA Controller
- 8259 - Programmable Interrupt Controller

- The microprocessor can communicate with external world or other systems using two types of communication interfaces. They are:
- Serial Communication Interface
- Parallel Communication Interface.

Serial Communication Interface

- The serial communication interface gets a byte of data from the microprocessor and sends it bit by bit to the other system serially or it receives data bit by bit serially from the external system.
- Then it converts the data into bytes and sends to the microprocessor.

Parallel Communication Interface

- A parallel communication interface gets a byte from the microprocessor and sends all the bits in that byte simultaneously (parallel) to the external system and vice-versa.

SERIAL COMMUNICATION INTERFACE

- The primary difference between parallel I/O and serial I/O is the number of lines used for data transfer; the parallel I/O uses the entire data bus and serial I/O uses one data line.
- In serial I/O transmission the microprocessor selects the peripheral through chip select (CS) and uses the control signals read to receive data and write to transmit data.
- The address decoding can be either I/O-mapped I/O or memory-mapped I/O.

- Serial data transmission is classified as
- Simplex
- Half duplex
- Full duplex

Simplex

- The data are transmitted in only one direction. There is no possibility of data transfer in the other direction.
- Example : Transmission from a computer to the printer.

Half duplex

- The data are transmitted in both directions, but not simultaneously.
- Example : Walky - Talky

Full duplex

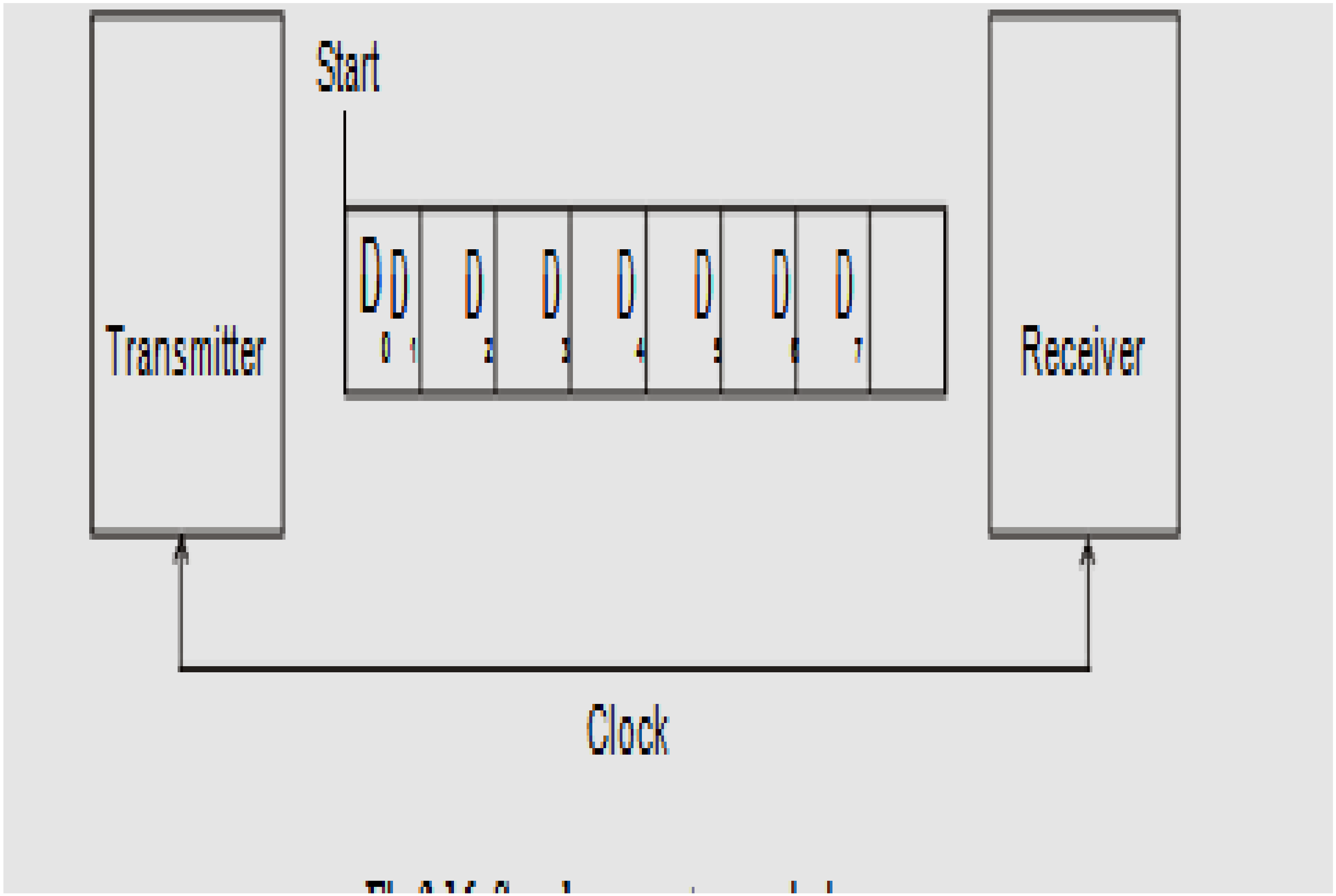
- The data are transmitted in both directions simultaneously.
- Example : Telephone

The data in the serial communication may be sent in two formats:

- Asynchronous
- Synchronous

Synchronous Transmission

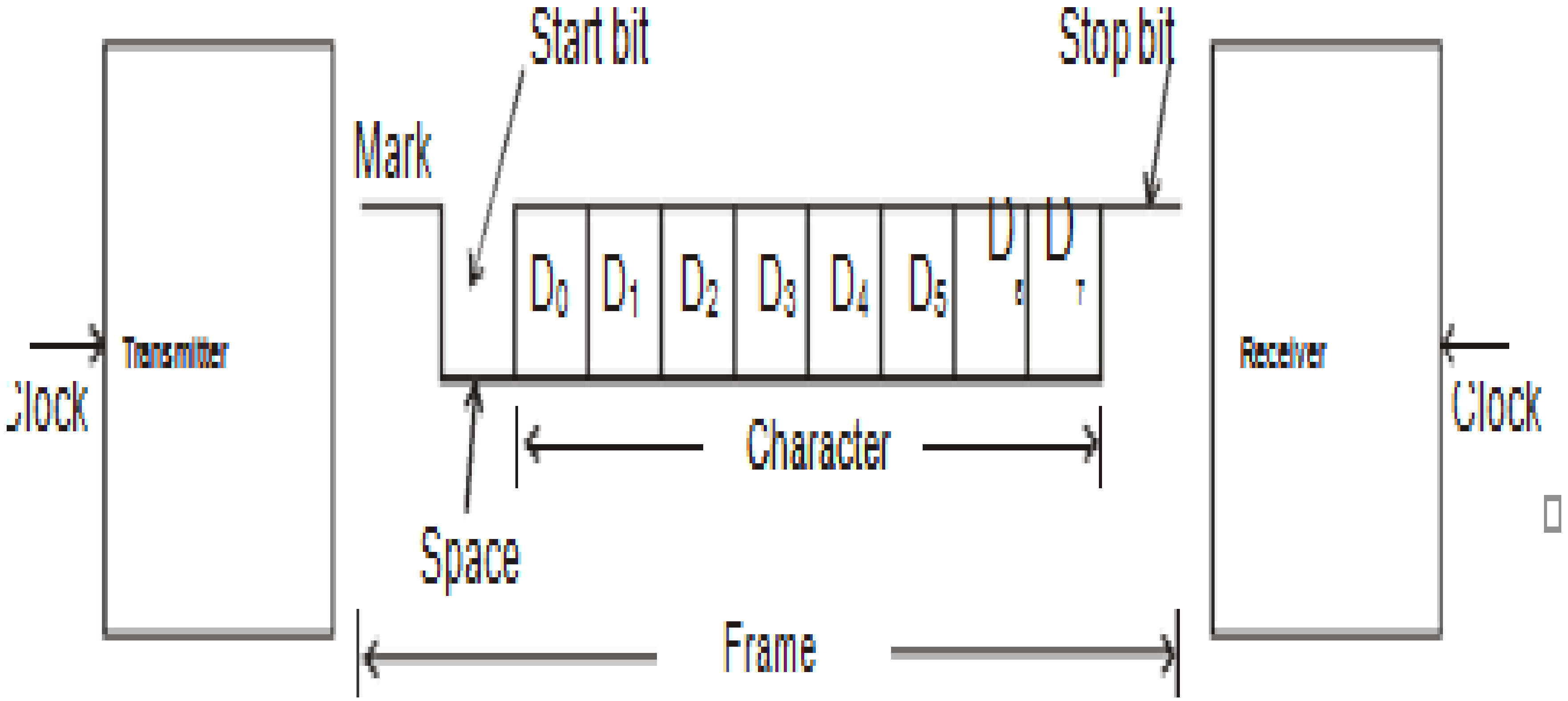
- In synchronous transmission, a receiver and transmitter work in same speed and could be synchronized.
- Both will use a common clock and start at the same time


Start
Transmitter
D_0 D_1 D_2 D_3 D_4 D_5 D_6 D_7
Receiver
Clock

[illegible]

Asynchronous transmission

- The asynchronous transmission is character-oriented. Each character carries the information of the Start and Stop bits
- When no data are being transmitted, a receiver stays high at logic 1, called **Mark** and logic 0 is called **Space**.
- Transmission begins with one start bit (Low), followed by 7 or 8 bits to represent a character and 1 or 2 Stop bits (high).
- A start bit, character and stop bits are called as **Frame**.


Start bit
Stop bit
Mark
D0
D1
D2
D3
D4
D5
D6
D7
Transmitter
Receiver
Clock
Clock
Character
Space
Frame

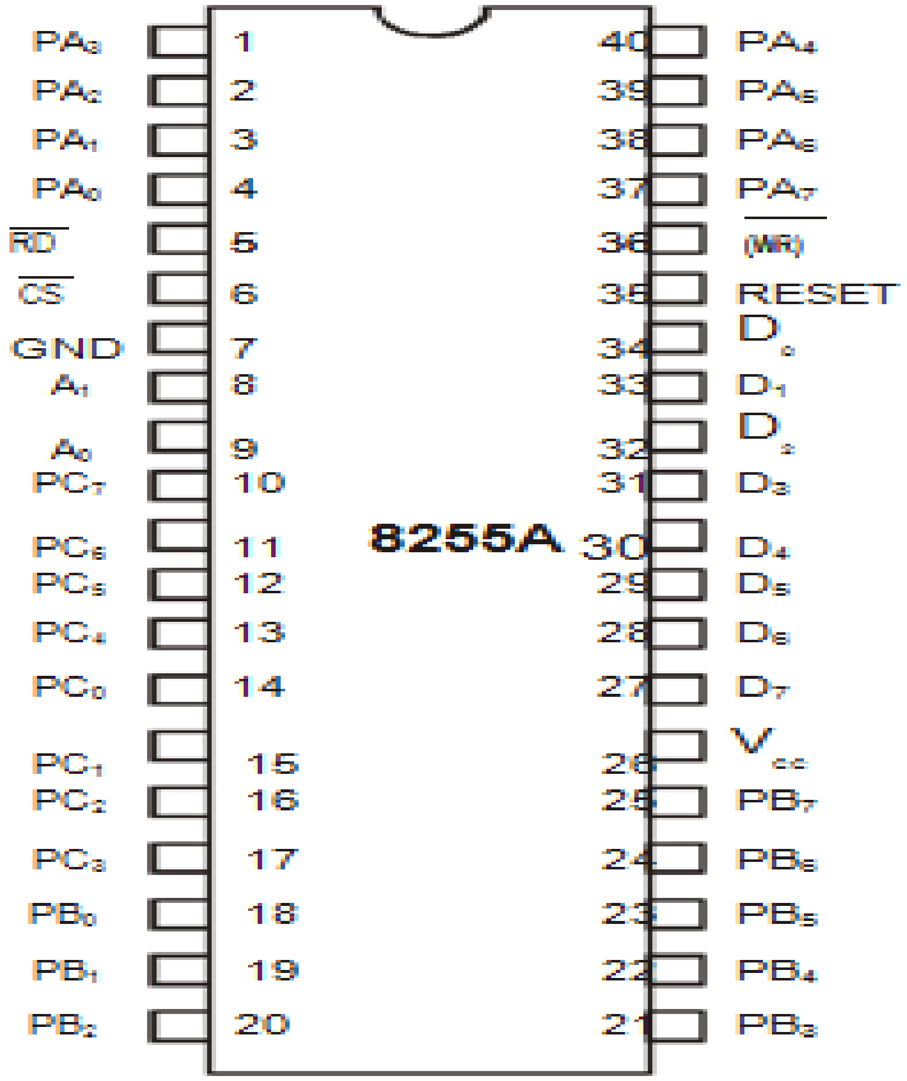
PARALLEL COMMUNICATION INTERFACE OR (8255 A - Programmable Peripheral Interface)

- It has a 3-state bi-directional 8-bit buffer which interfaces the 8255A to the sys-tem data bus.
- It has 24 programmable I/O Pins.
- It reduces the external logic normally needed to interface peripheral devices.
- It has two 8 bit ports: Port A, Port B, and two 4 bit ports: C_{UPPER} and $C_{LOWER.}$
- Available in 40-Pin DIP and 44-Pin PLCC.

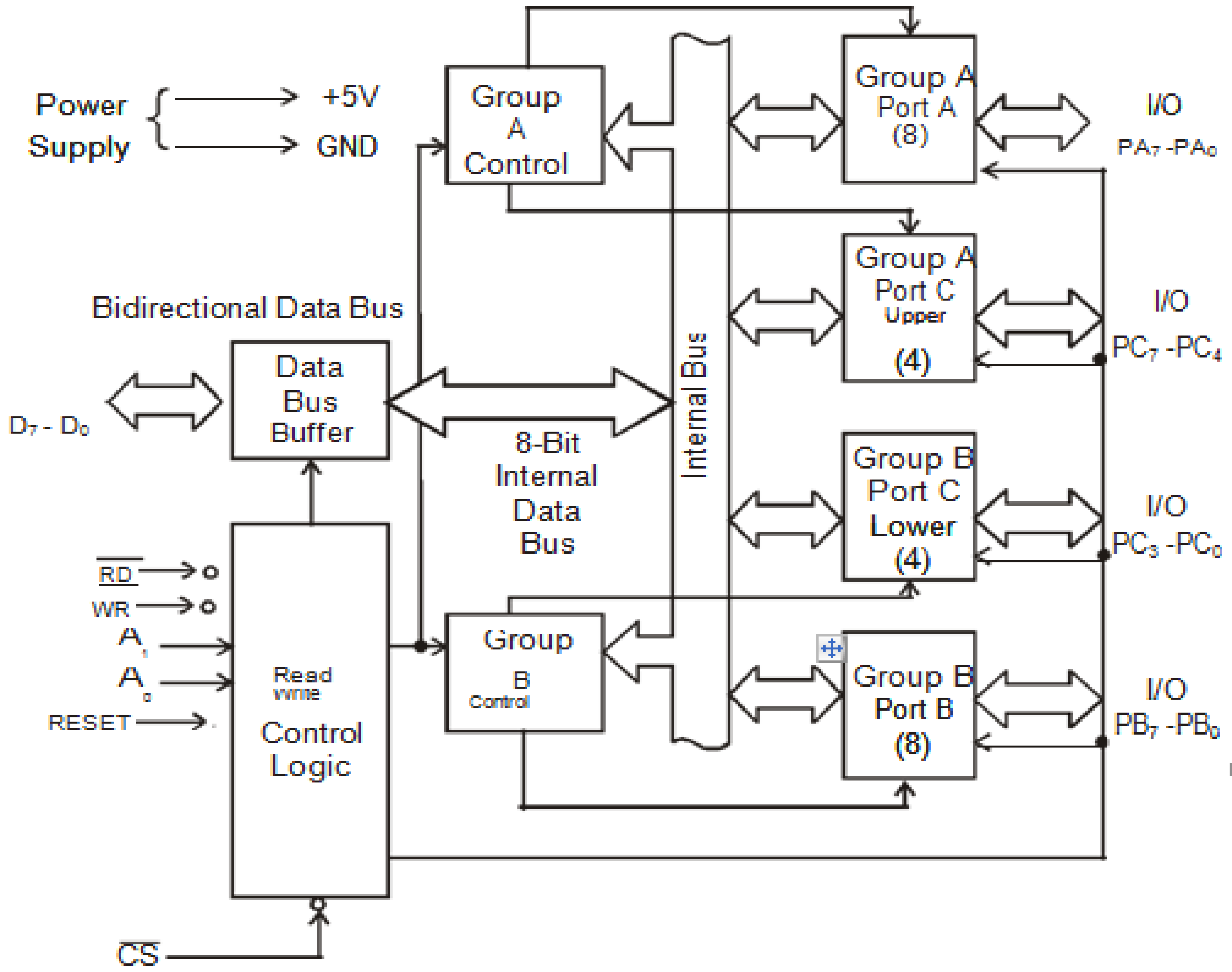
OPERATING MODES

- It can be operated in two basic modes:
  - Bit Set/Reset Mode
  - I/O Mode
- I/O mode is further divided into 3 modes:
  - Simple I/O mode (Mode 0)
  - Strobed I/O mode (Mode 1)
  - Bidirectional Data Transfer mode (Mode 2)

Pin diagram of 8255A



- The 8255 consists of Four sections namely
- Data Bus Buffer
- Read/Write Control Logic
- Group A Control
- Group B Control

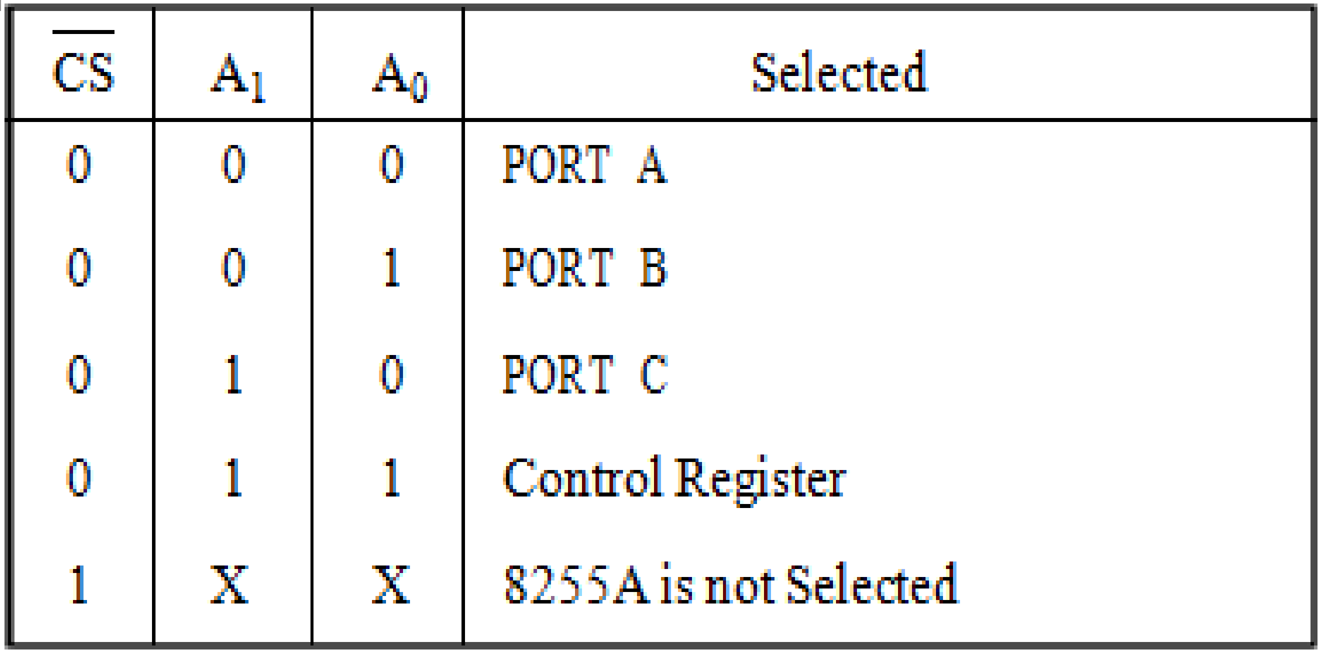

Power Supply
+5V
GND
Group A Control
Group A Port A (8)
I/O PA7 -PA0
Group A Port C Upper (4)
I/O PC7 -PC4
Bidirectional Data Bus
Data Bus Buffer
D7 - D0
8-Bit Internal Data Bus
Internal Bus
Group B Port C Lower (4)
I/O PC3 -PC0
RD
WR
A1
A0
RESET
Read Write Control Logic
Group B Control
Group B Port B (8)
I/O PB7 -PB0
CS

DATA BUS BUFFER

- Used to interface the internal data bus of 8255A to the system data bus of 8085.
- Using IN or OUT instructions, CPU can read or write the data from/to the data bus buffer.
- It can also be used to transfer control words and status information between CPU and 8255A.

Read/Write Control Logic

- This block controls the Chip Selection (CS), Read (RD) and Write (WR) operations.
- It consists of A_0 and A_1 signals which are generally connected to the CPU address lines A_0 and A_1 respectively.
- When CS (Chip Select) signal goes LOW, different values of $\mathbf{A}_0$ and A_1 select one of the I/O ports or control register



$\overline{CS}$	A_1	A_0	Selected
0	0	0	PORT A
0	0	1	PORT B
0	1	0	PORT C
0	1	1	Control Register
1	X	X	8255A is not Selected

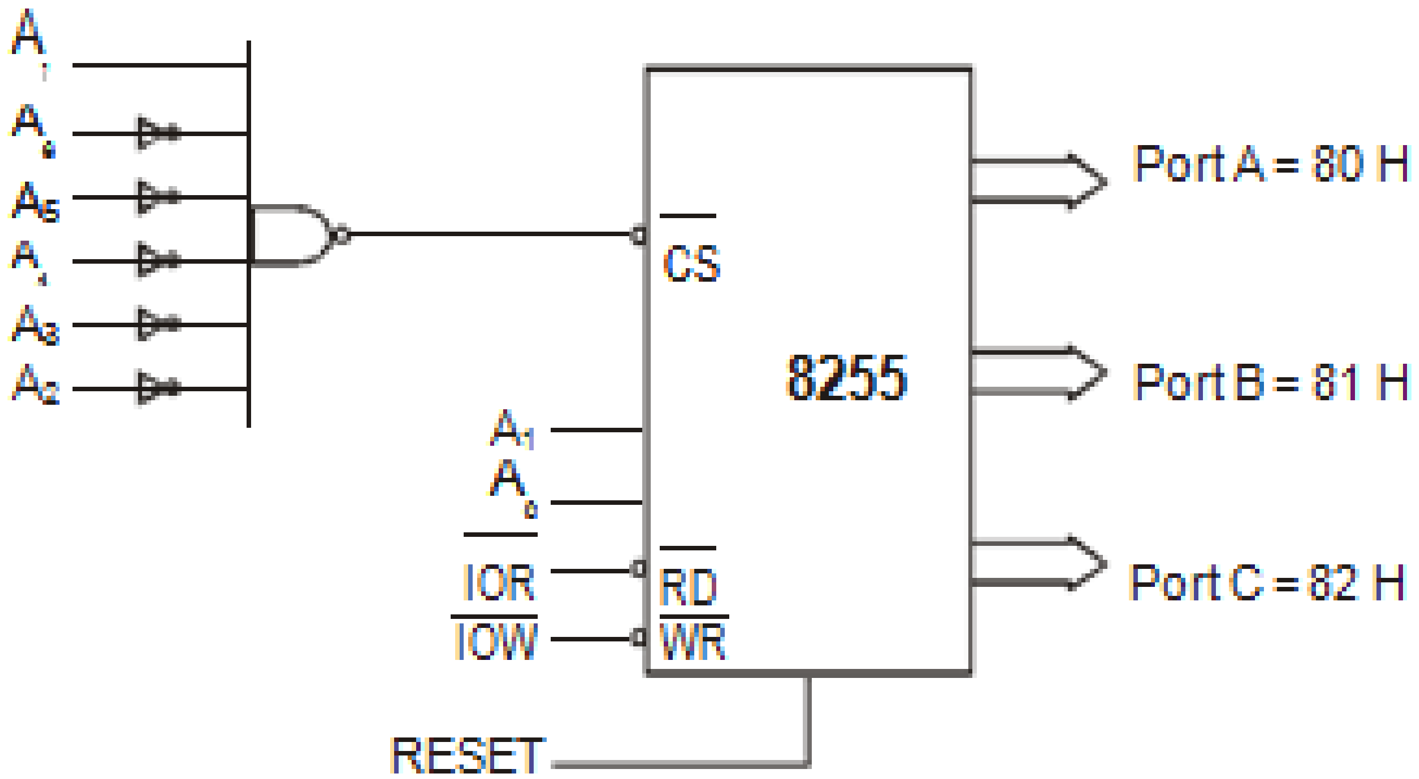
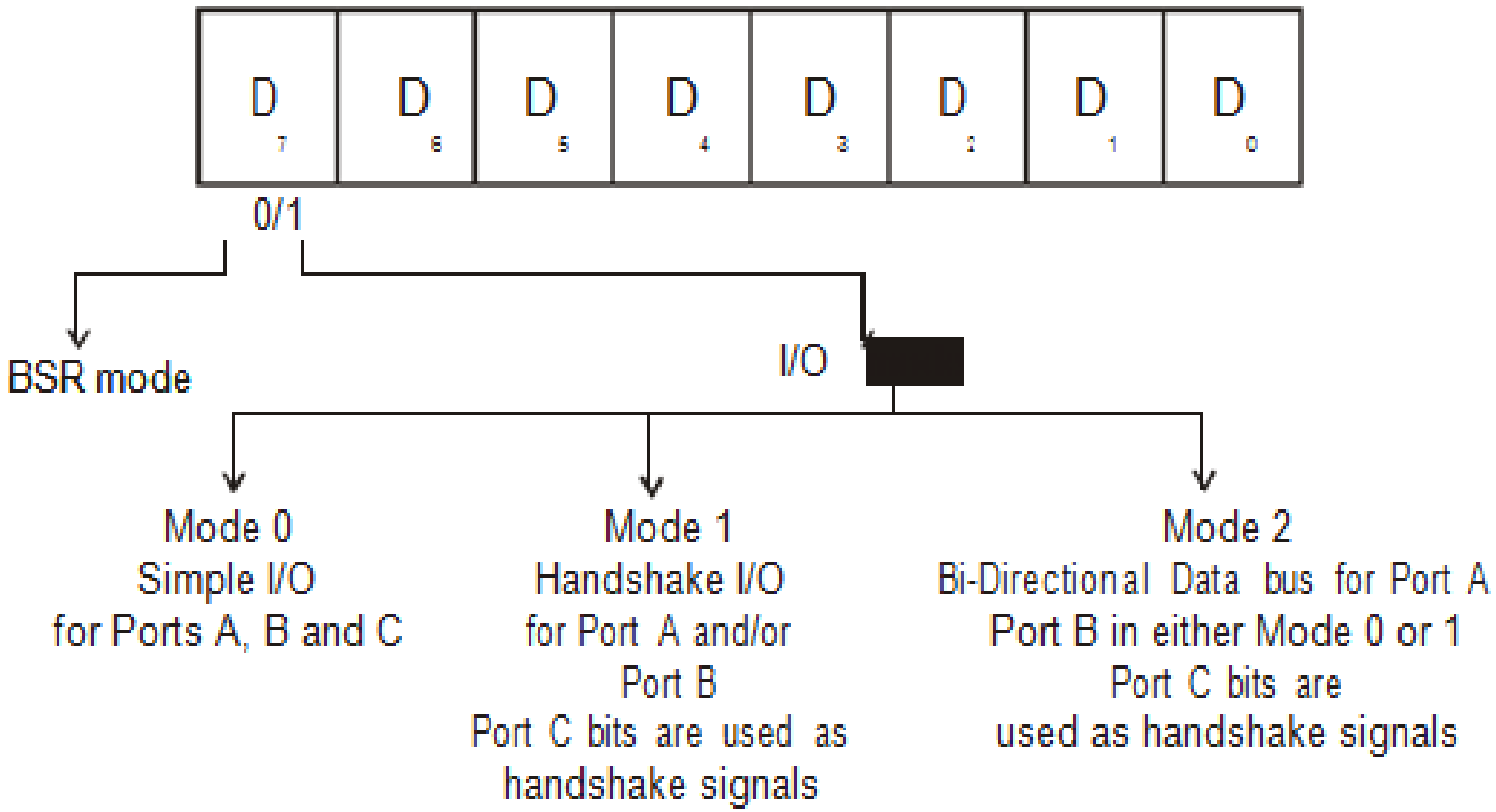


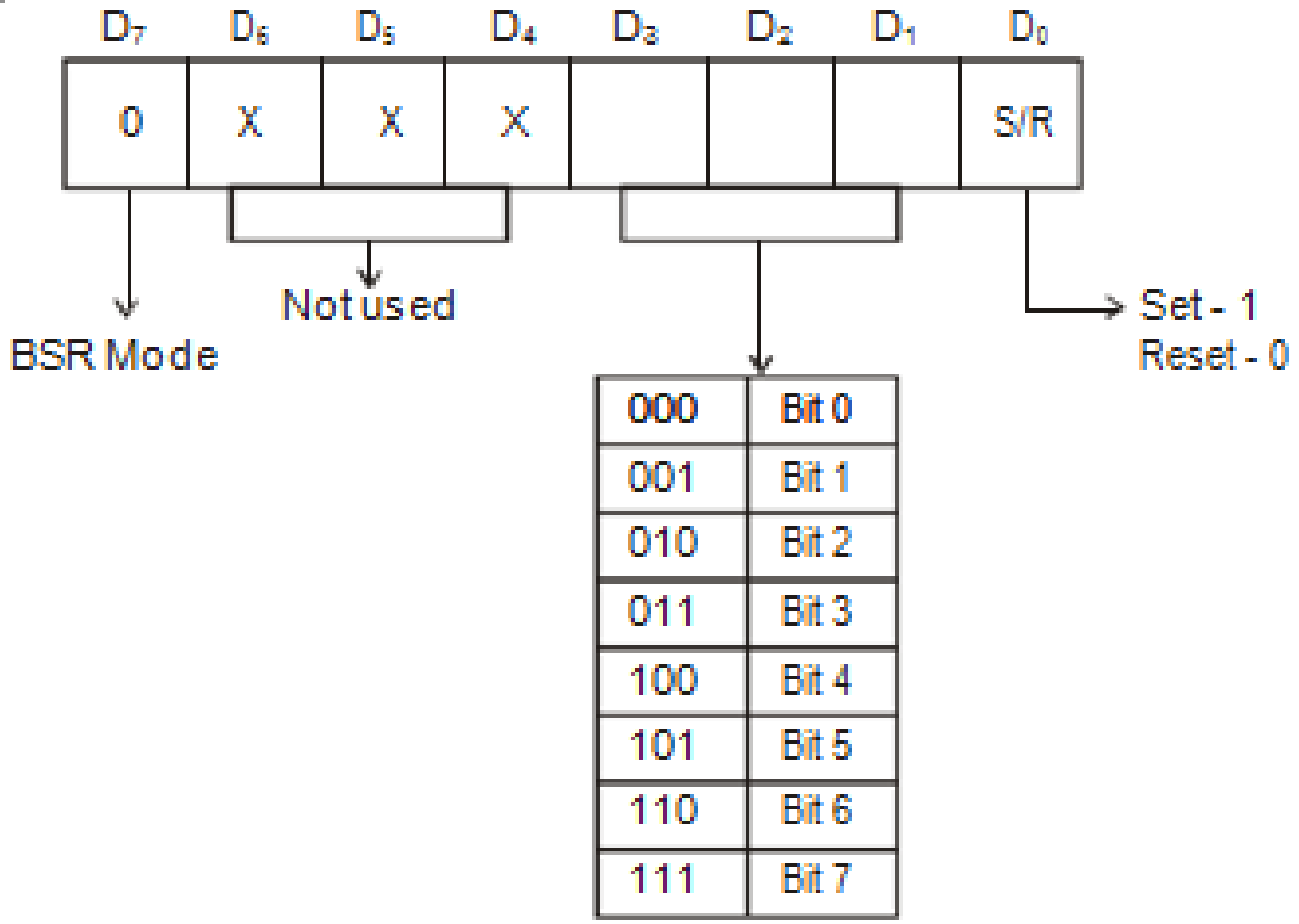
Fig.3.3. Chip Select Logic

- Group A : Port A and Most Significant Bits (MSB) of Port C (PC_4 – PC_7)
- Group B : Port B and Least Significant Bits (LSB) of Port C (PC_0 – PC_3)
- **Port A:** One 8-bit data output latch/buffer and one 8-bit input latch buffer.
- **Port B:** One 8-bit data input/output latch/buffer.
- **Port C:** One 8-bit data output latch/buffer and one 8-bit data input buffer. This port can be divided into two 4-bit ports and it can be used for the control signal outputs and status signal inputs in conjunction with ports A and B.


D7
D6
D5
D4
D3
D2
D1
D0
0/1
BSR mode
I/O
Mode 0
Simple I/O
for Ports A, B and C
Mode 1
Handshake I/O
for Port A and/or
Port B
Port C bits are used as
handshake signals
Mode 2
Bi-Directional Data bus for Port A
Port B in either Mode 0 or 1
Port C bits are
used as handshake signals

BSR (Bit Set/Reset) Mode

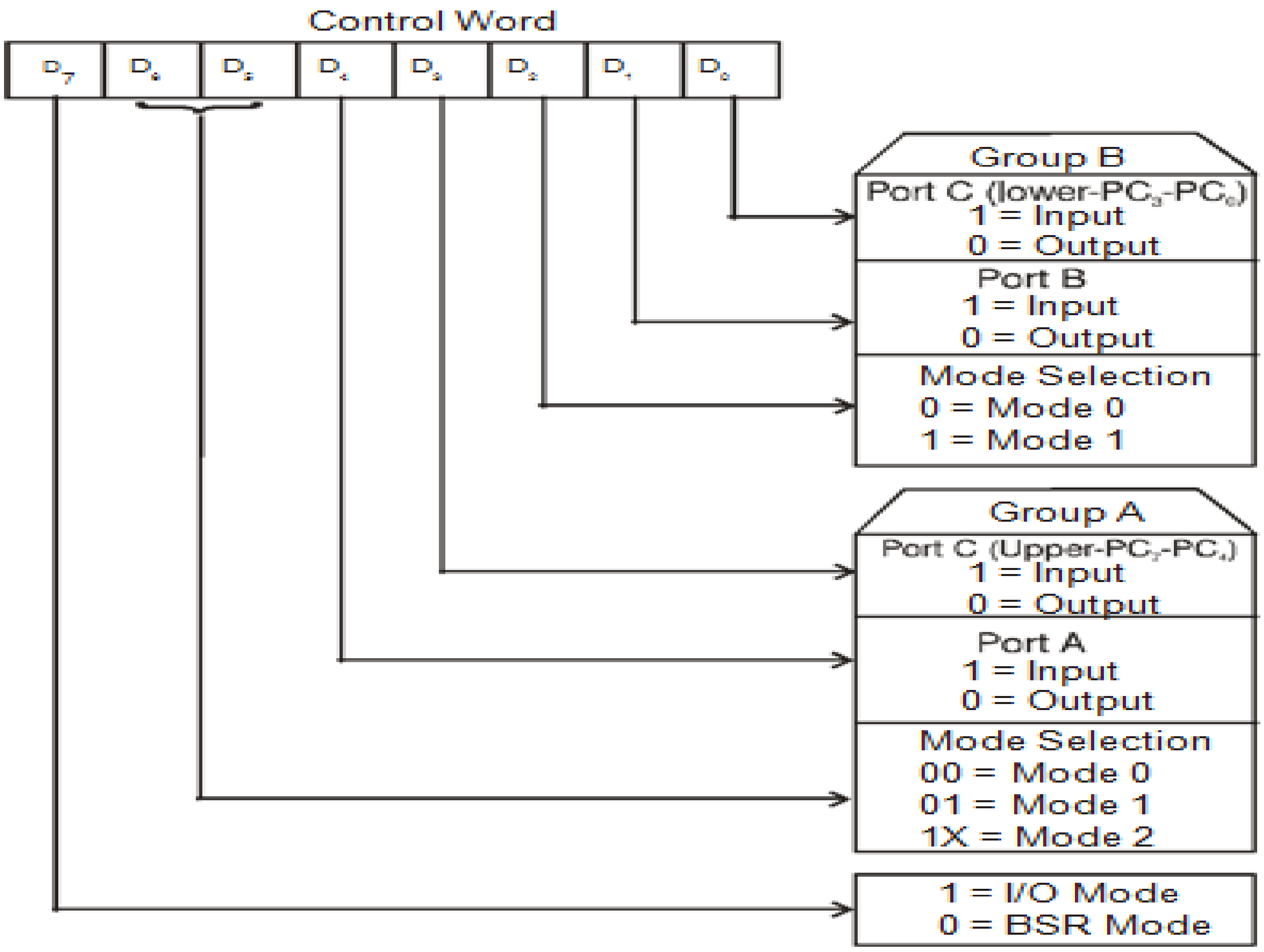
- This mode is applicable only for Port C.
- A control word with bit $D_7 = 0$ is recognized as BSR control word.
- This control word can set or reset a single bit in the Port C.



000	Bit 0
001	Bit 1
010	Bit 2
011	Bit 3
100	Bit 4
101	Bit 5
110	Bit 6
111	Bit 7

The I/O mode is divided into three modes Mode 0, Mode 1 and Mode 2 as given below.

- Mode 0 – Basic I/O Mode
- Mode 1 – Strobed I/O Mode
- Mode 2 – Bi-directional data transfer mode



Control Word
D7
D6
D5
D4
D3
D2
D1
D0
Group B
Port C (lower-PC3-PC0)
1 = Input
0 = Output
Port B
1 = Input
0 = Output
Mode Selection
0 = Mode 0
1 = Mode 1
Group A
Port C (Upper-PC7-PC4)
1 = Input
0 = Output
Port A
1 = Input
0 = Output
Mode Selection
00 = Mode 0
01 = Mode 1
1X = Mode 2
1 = I/O Mode
0 = BSR Mode

Mode 0 – Basic I/O mode

- The features of Mode 0 are :
- Two 8-bit ports (Port A, Port B) and two 4-bit ports (Port C_U, Port C_L). Any port can be input or output.
- Outputs are latched.
- Inputs are not latched.

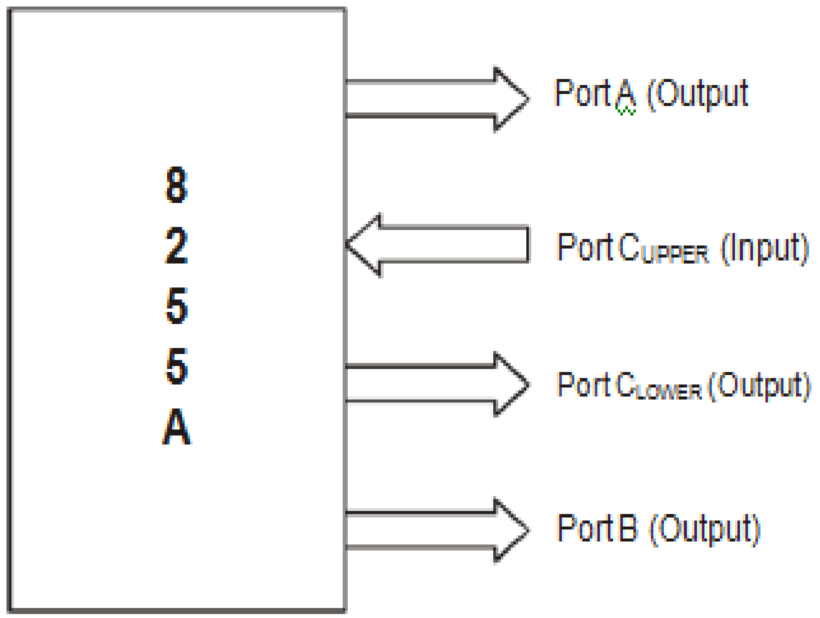


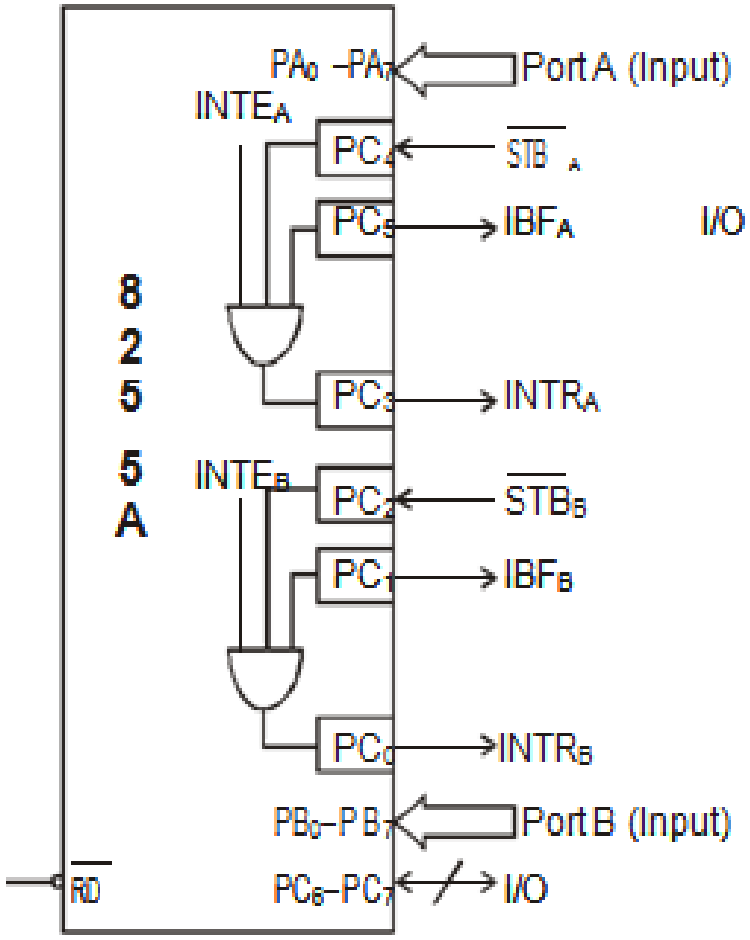
Fig.3.6. Ports in Mode 0

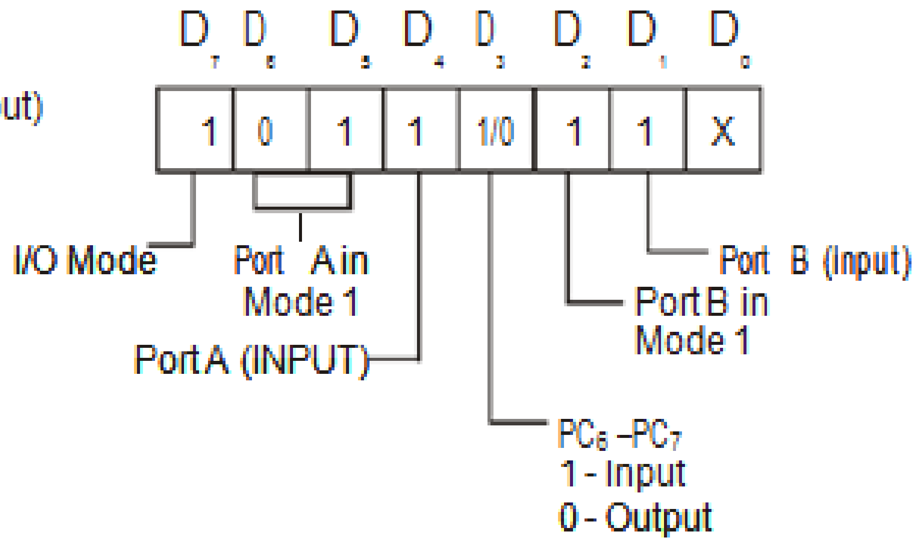
Mode 1 - Strobed Input/Output

- In this mode, handshake signals are exchanged between the microprocessor and peripherals prior to data transfer

The features of mode 1 are :

- Two Groups (Group A and Group B).
- Each group contains one 8-bit data port and one 4-bit control/data port. The 8-bit data port can be either input or output
- The 4-bit port is used for control and status of the 8-bit data port.
- If Port A is in mode 1 (input), then PC_3, PC_4, PC_5 are used as control signals. If Port B is in mode 1 (input), then PC_0, PC_1, PC_2 are used as control signals.
- Both inputs and outputs are latched.





Control Word

D7	D6	D5	D4	D3	D2	D1	D0
I/O	I/O	IBFA	INTEA	INTRA	INTEB	IBFB	INTRB

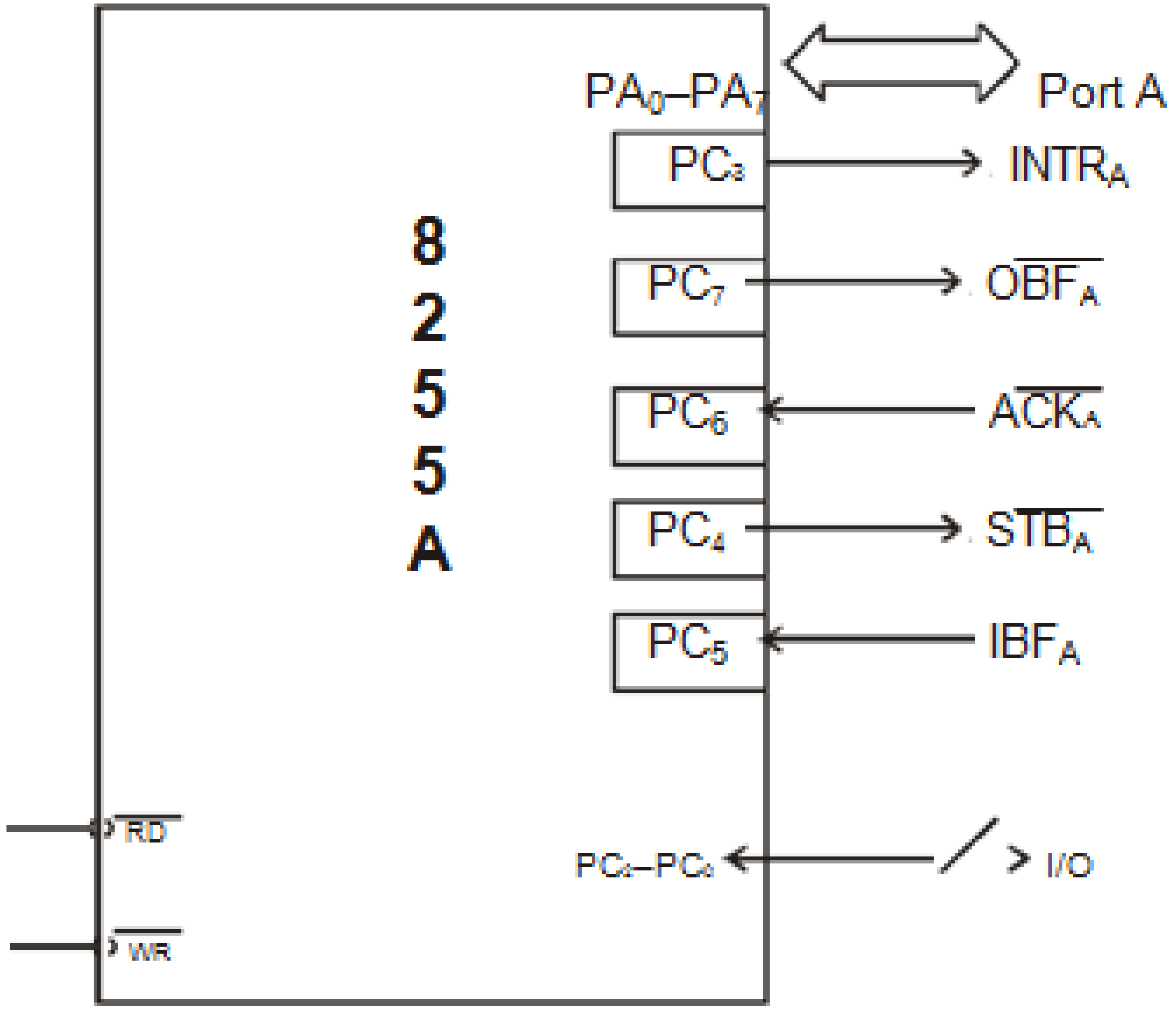
Status Word for Mode 1 (Input)

- **STB** (Strobe Input) – A "low" signal on this pin indicates that the peripheral device has transmitted a byte of data.
- The 8255A in response to STB , generates IBF and INTR.
- **IBF** (Input Buffer Full) – A "high" signal issued by 8255A is an acknowledge to indicate that the input latch has received the data byte. This is reset when the CPU reads the data.

- **INTR** (Interrupt Request) – This is an output signal, used to interrupt the CPU. This will be in active state when STB , IBF and INTE (internal Flip-Flop) are all at logic 1. This will be reset by the falling edge of RD signal.
- **INTE** (Interrupt Enable) – This is an Internal Flip-Flop used to enable or disable the generation of INTR signal. There are two Flip-Flops $INTE_A$ and $INTE_B$ are set/reset using the BSR mode.

Mode 2 – Bi-directional Data Transfer Mode

- This mode provides a means for communicating with a peripheral device or structure on a single 8-bit bus for both transmitting and receiving data (bidirectional bus I/O).
- The features of Mode 2 are :
- Used in Group A only.
- Port A only acts as bi-directional bus port
- Port C (PC_3-PC_7) is used for handshaking purpose.



8255A
PA0–PA7
Port A
PC3
INTRA
PC7
OBFA
PC6
ACKA
PC4
STBA
PC5
IBFA
RD
WR
PC2–PC0
I/O

INTR (Interrupt Request):

- A high on this output can be used to interrupt the CPU for input or output operations.

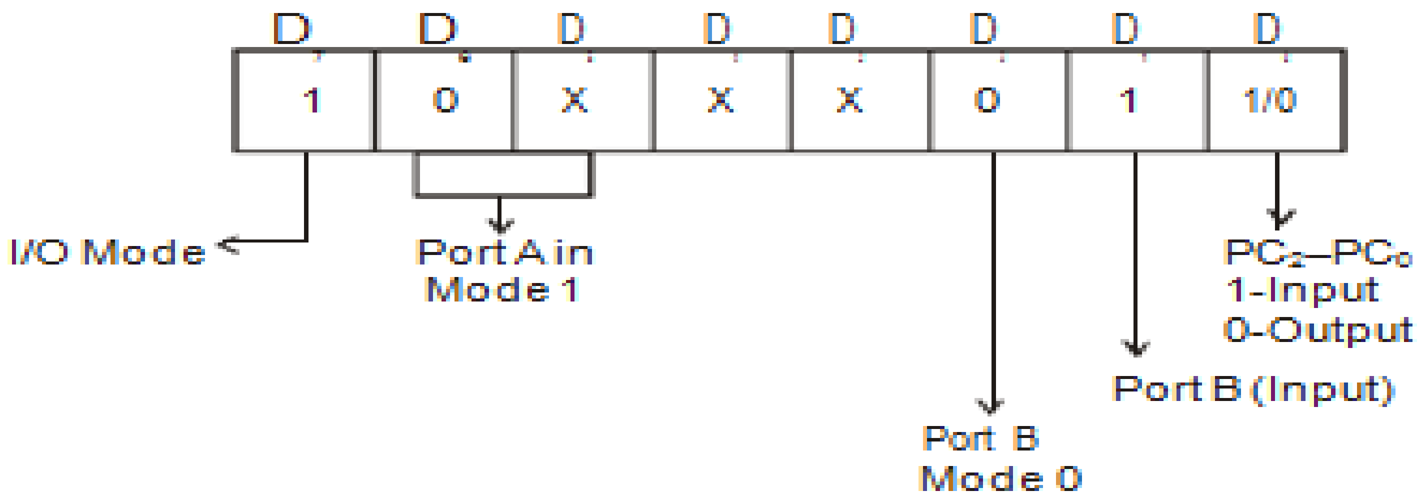
OBF(Output Buffer Full):

This signal will go LOW to indicate that the CPU has written data out to Port A.

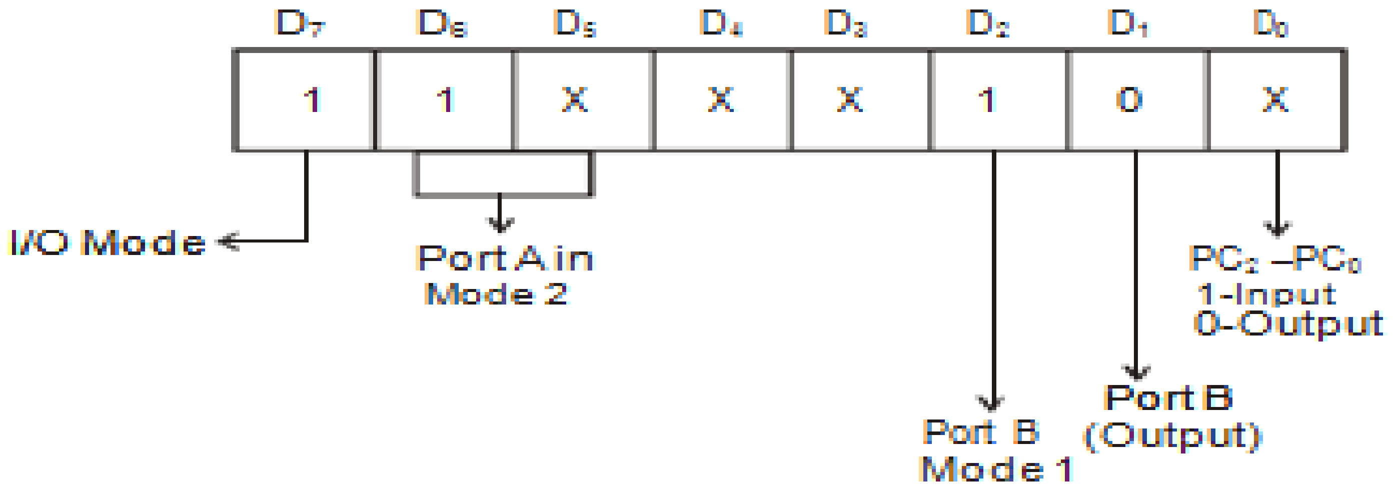
ACK(Acknowledge):

A LOW on this input enables the tri-state output buffer of Port A to send out the data.

- Otherwise, the output buffer will be in the high impedance state.



Mode 2 - Input Configuration



Mode 2 - Output Configuration

DIGITAL TO ANALOG CONVERTERS (DAC)

- The digital to analog converters (DAC) convert binary numbers into their analog equivalent voltages or currents. Several techniques are employed for digital to analog conversion.
- Weighted resistor network
- R-2R ladder network
- Current output D/A converter

APPLICATIONS

- Digitally controlled gains
- Motor speed control
- Programmable gain amplifiers
- Digital voltmeters
- Panel meters, etc.

- ***Resolution:*** It is a change in analog output for one LSB change in digital input.

$$(1/2^{n})*V_{ref}$$

- 1/256*5 V=39.06 mV (since *n*=8 for 8-bit DAC)
- ***Settling time:*** It is the time required for the DAC to settle for a full scale code change.

DAC 0800 8-bit Digital to Analog converter

- DAC0800 is a monolithic 8-bit DAC manufactured by National semiconductor.
- It has settling time around 100ms.
- It can operate on a range of power supply voltage i.e. from 4.5V to +18V.
- Usually the supply V+ is 5V or +12V. The V- pin can be kept at a minimum of –12V.
- Resolution of the DAC is 39.06mV

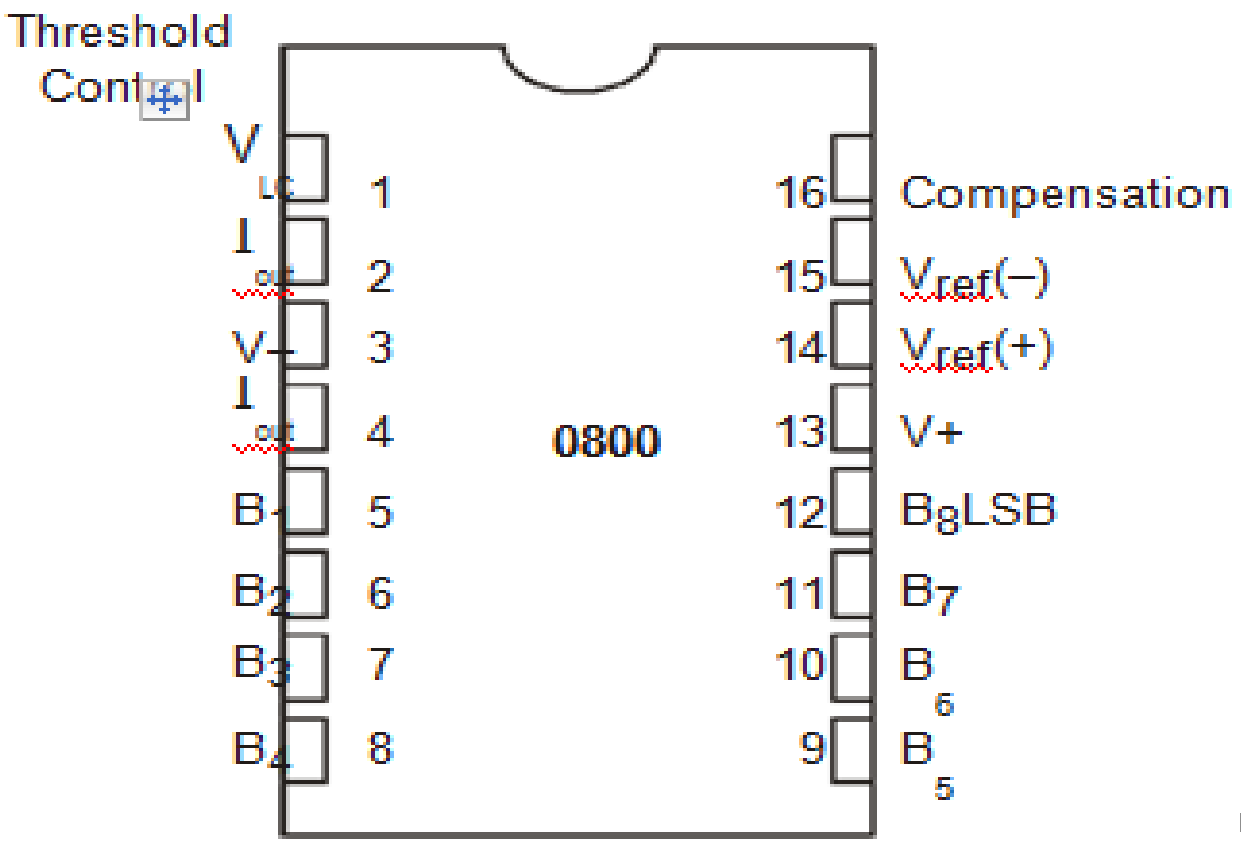
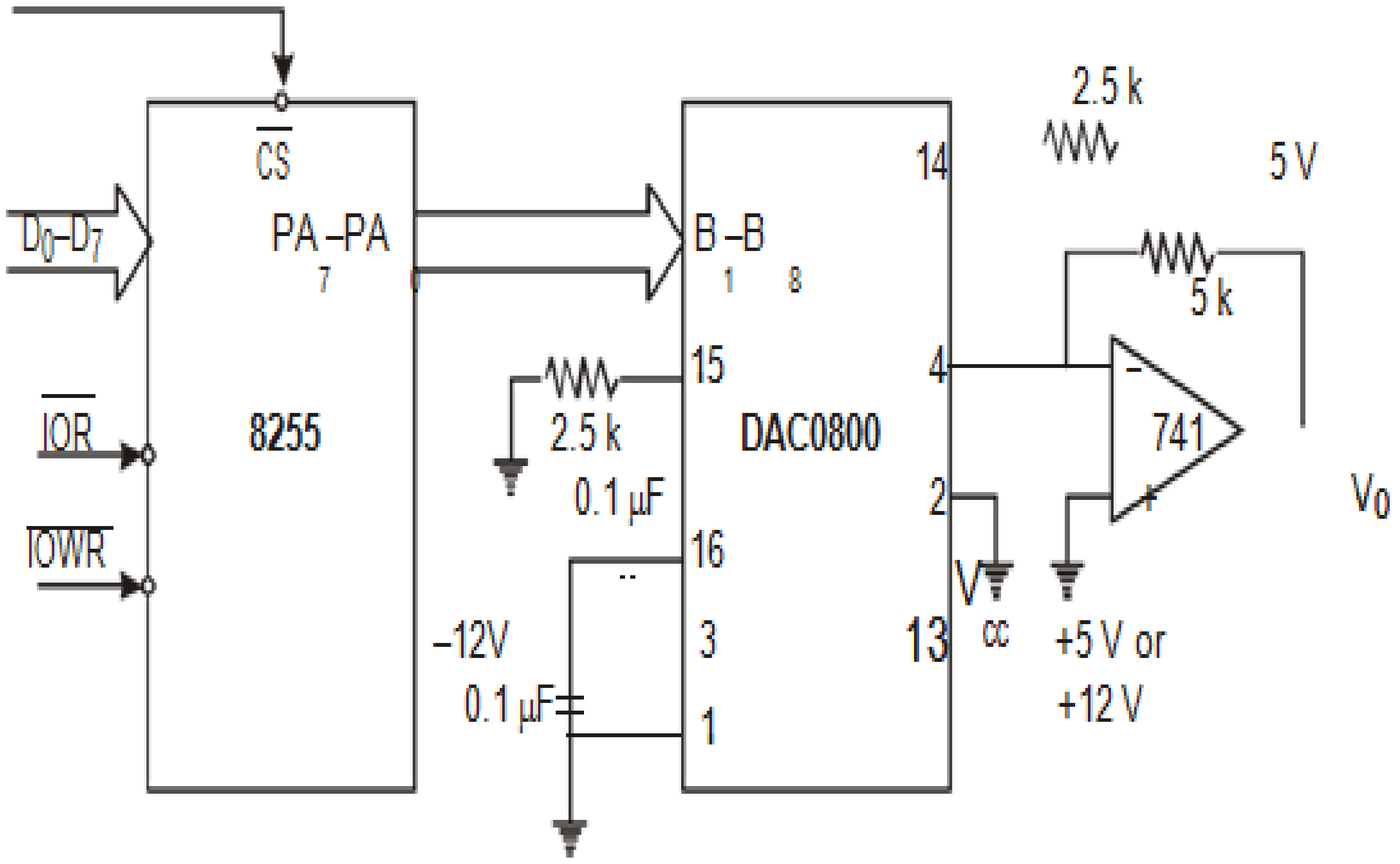


Fig.3.18. Pin Diagram of DAC 0800

Interfacing of DAC0800 with 8086

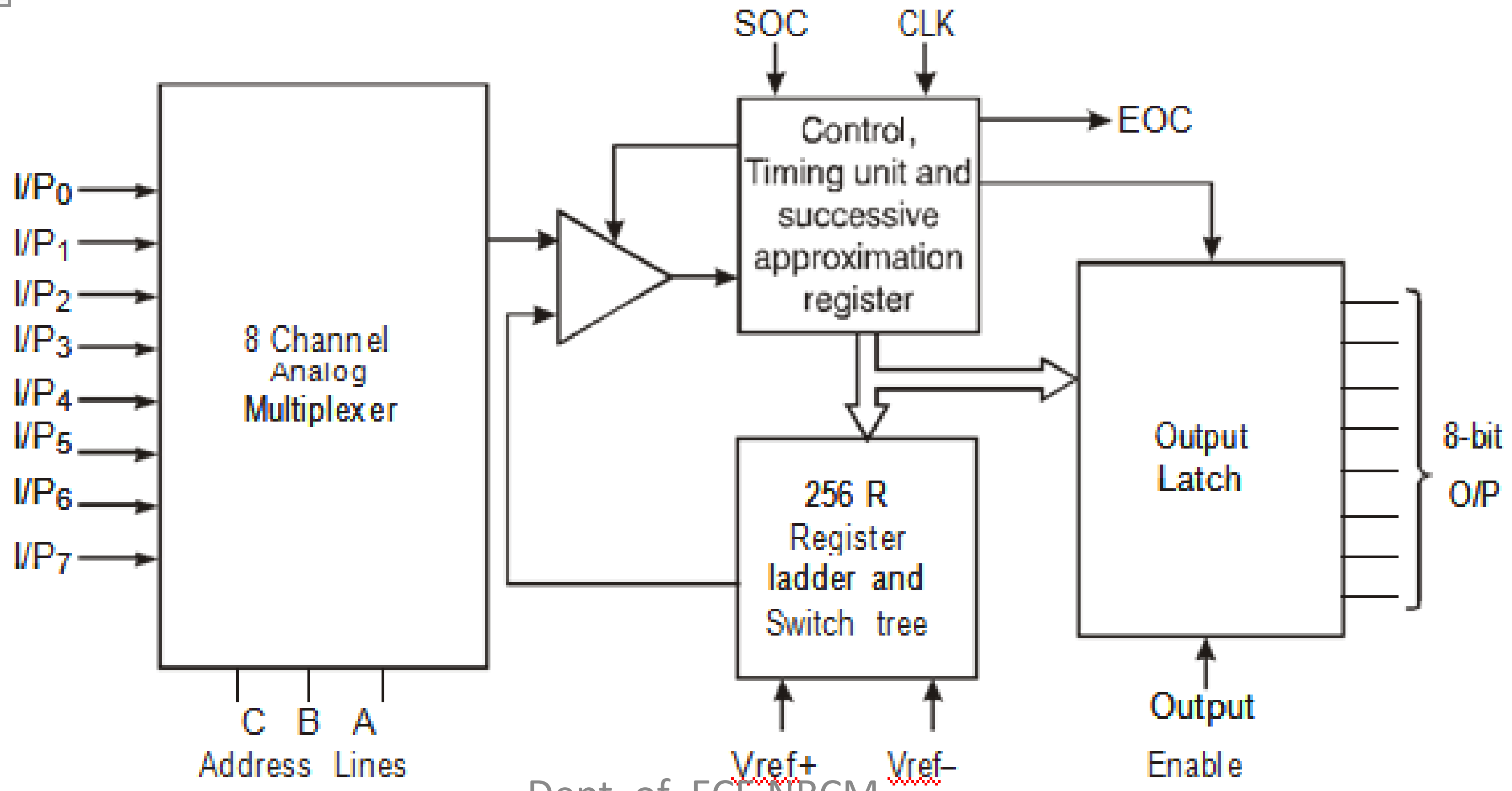


- The V_{ref+} should be tied to +5 V to generate a wave of +5V amplitude.
- The required frequency of the output is 500 Hz, i.e. the period is 2 ms.
- Assuming the wave to be generated
- is symmetric, the waveform will rise for 1 ms and fall for 1 ms.
- This will be repeated continuously.

```
ASSUME   CS : CODE
CODE     SEGMENT
START :  MOV AL,80 H        ; Initialise 8255 ports
         OUT CWR, AL        ; suitably.
         MOV AL, 00H        ; Start rising ramp from
BACK :   OUT Port A, AL     ; 0V by sending 00H to DAC.
         INC AL             ; Increment ramp till 5V
         CMP AL, FFH        ; compare with FFH
         JB BACK            ; If it is FFH then
BACK1 :  OUT Port A, AL     ; Output it and start the falling
         DEC AL             ; ramp by decrementing the
         CMP AL, 00         ; counter till it reaches
         JA BACK1           ; zero. Then start again
         JMP BACK           ; for the next cycle.
CODE     ENDS
         END START
```

ANALOG TO DIGITAL INTERFACE

- **ADC 0808/0809**


SOC
CLK
EOC
I/P0
I/P1
I/P2
I/P3
I/P4
I/P5
I/P6
I/P7
8 Channel Analog Multiplexer
Control, Timing unit and successive approximation register
256 R Register ladder and Switch tree
Output Latch
8-bit O/P
C B A
Address Lines
Vref+
Vref–
Output Enable

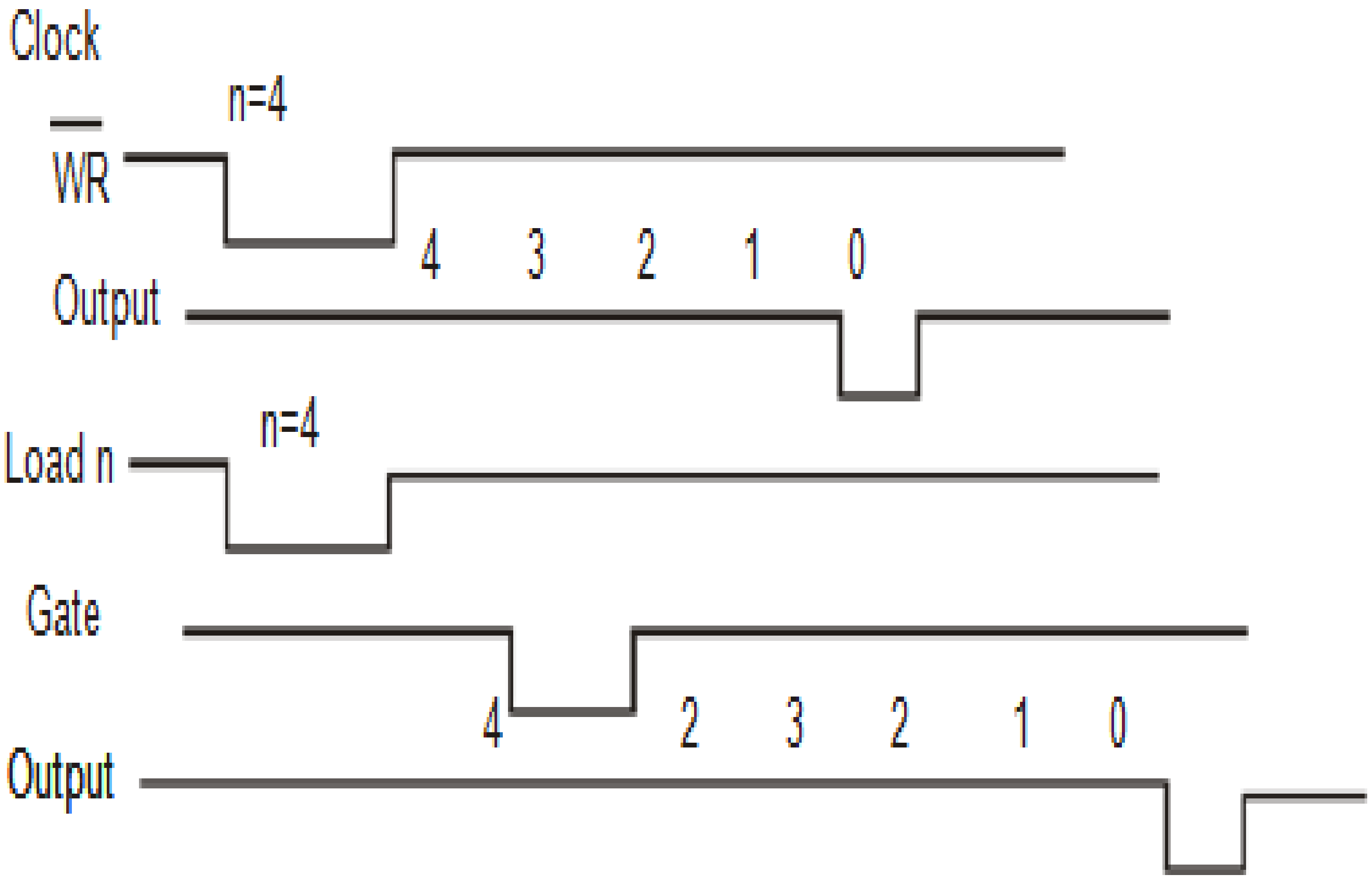
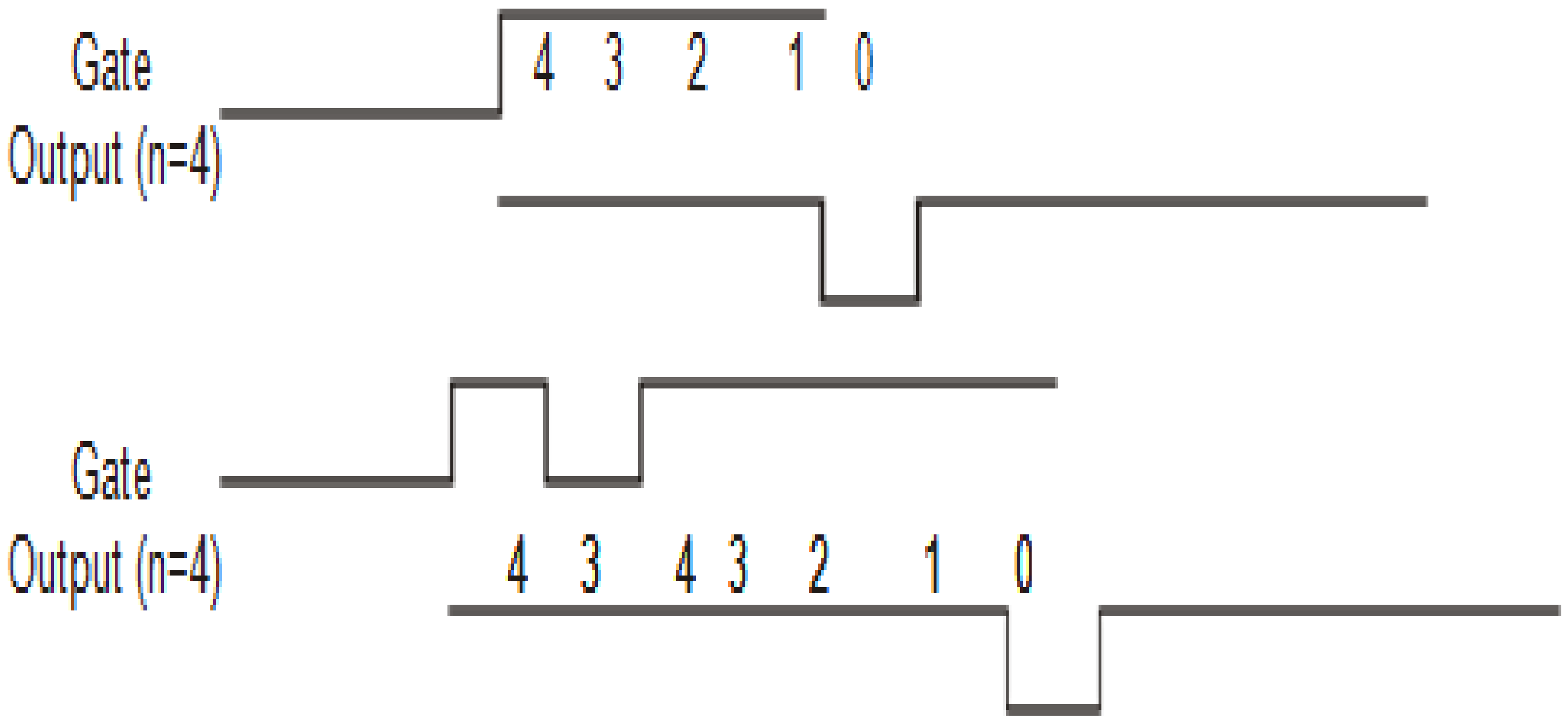


Fig 3.30 Mode 4


Gate
Output (n=4)
4 3 2 1 0
Gate
Output (n=4)
4 3 4 3 2 1 0

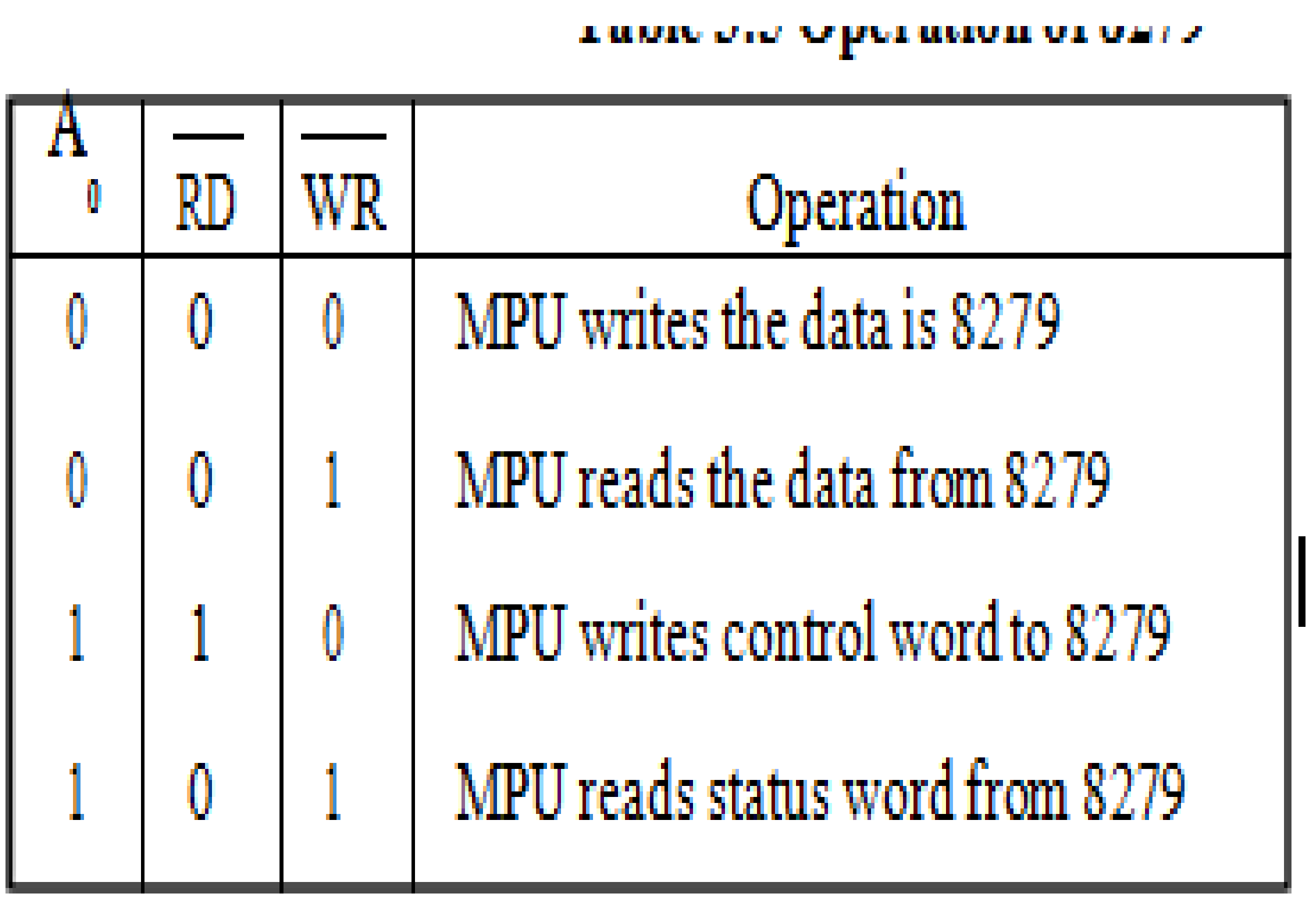


Table [illegible] Operation of 8279

A_0	$\overline{RD}$	$\overline{WR}$	Operation
0	0	0	MPU writes the data is 8279
0	0	1	MPU reads the data from 8279
1	1	0	MPU writes control word to 8279
1	0	1	MPU reads status word from 8279

Keyboard Section

- This section has keyboard debounce and control, 8x8 FIFO/Sensor RAM, 8 Return lines (RL_0 – RL_7) and CNTL/STB and shift lines.
- In the keyboard debounce and control unit, keys are automatically debounced and the keyboard can be operated in two modes.
- Two key lock out
- N – key roll over

FEATURES

High integration of functionality :

- Microcontrollers are called as single chip computers because they have on - chip memory and I/O circuitry and other circuitries that enable them to function as small stand - alone computers without other supporting circuitry.

Advantages of microcontrollers

- The overall system cost is low, as the peripherals are integrated in a single chip.
- The product is of small size as compared to the microprocessor based system and is very handy.
- The system is more reliable.
- The system is easy to troubleshoot and maintain.
- If required additional RAM, ROM and I/O ports may be interfaced

ARCHITECTURE OF 8051

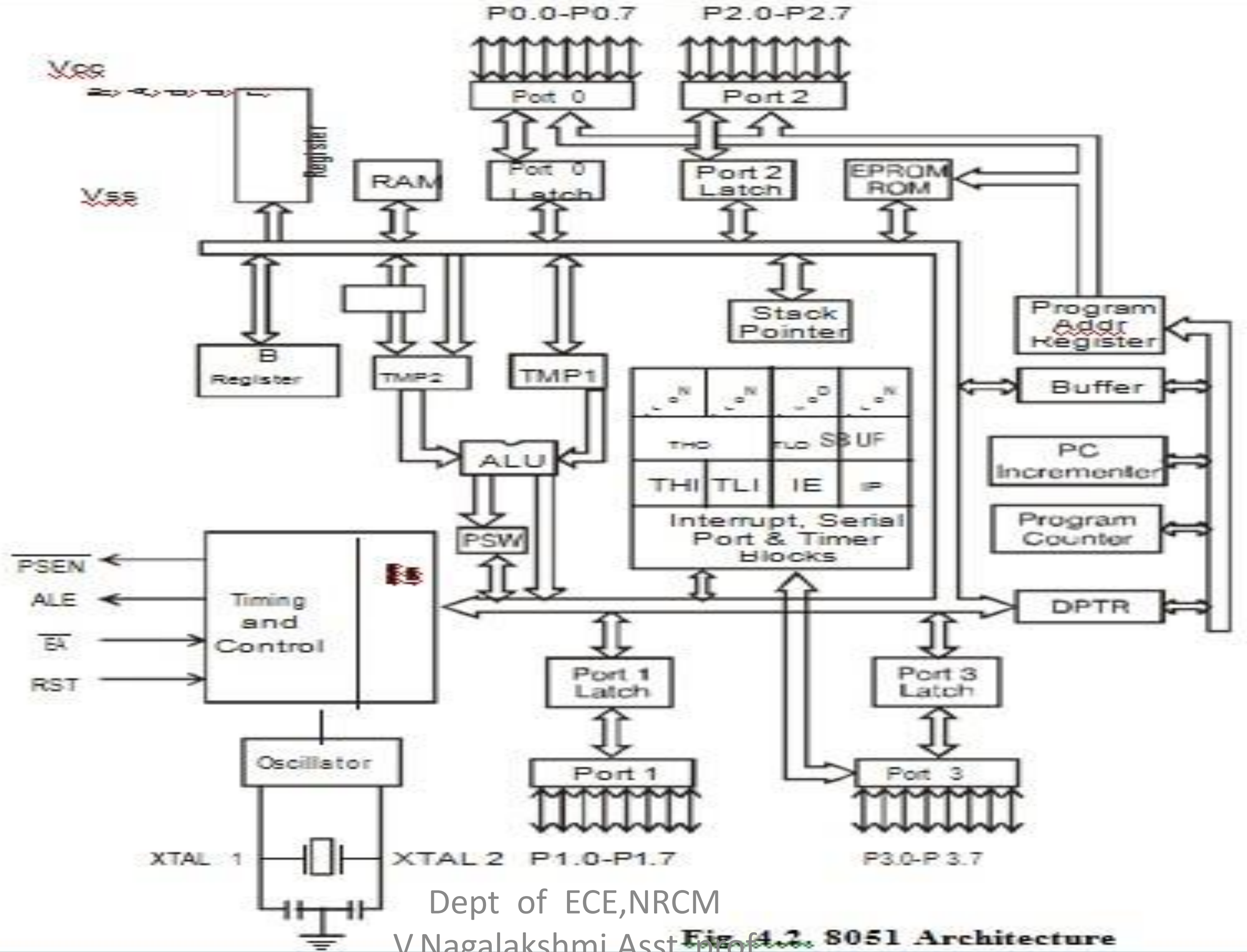


Fig. 4.2. 8051 Architecture

The features of the 8051 are :

- 8 bit CPU with registers A (the accumulator) and B
- 16 bit Program Counter (PC) and Data Pointer (DPTR)
- 8 bit Program Status Word (PSW)
- 64K Program memory address space
- 64K Data memory address space
- 128 bytes of on chip data memory
- 32 I/O pins for four 8 bit ports : Port 0, Port 1, Port 2, Port 3
- Two 16 bit timers / counters : T_0 and T_1
- Full duplex UART : SBUF
- Two external and three internal interrupt sources
- On chip clock oscillator.

Central processing unit

- The CPU is the brain of the microcontrollers reading user’s programs and executing the expected task as per instructions stored there in. It’s primary elements are an Accumulator (ACC), B register (B), Stack pointer (SP), Program counter (PC), Program status word (PSW), Data pointer register (DPTR) and few more 8 bit registers.

Accumulator

- The accumulator performs arithmetic and logic functions on 8 bit input variables.
- Arithmetic operations include basic addition, subtraction, multiplication and division.
- Logical operations are AND, OR XOR as well as rotate, clear, complement etc.
- Apart from all the above, accumulator is responsible for conditional branching decisions and provides a temporary place in a data transfer operations within the device.

B Register

- B register is used in multiply and divide operations.
- During execution B register either keeps one of the two inputs and then retains a portion of the result.
- For other instructions it is used as general purpose register.

Stack Pointer

- Stack Pointer (SP) is an 8 bit register.
- This pointer keeps track of memory space where the important register information are stored when the program flow gets into executing a subroutine.
- The stack portion may be placed in anywhere in the onchip RAM.
- But normally SP is initialized to 07H after a device reset and grows up from the location 08H.
- The SP is automatically incremented or decremented for all PUSH or POP instructions and for all subroutine calls and returns.

Program Counter

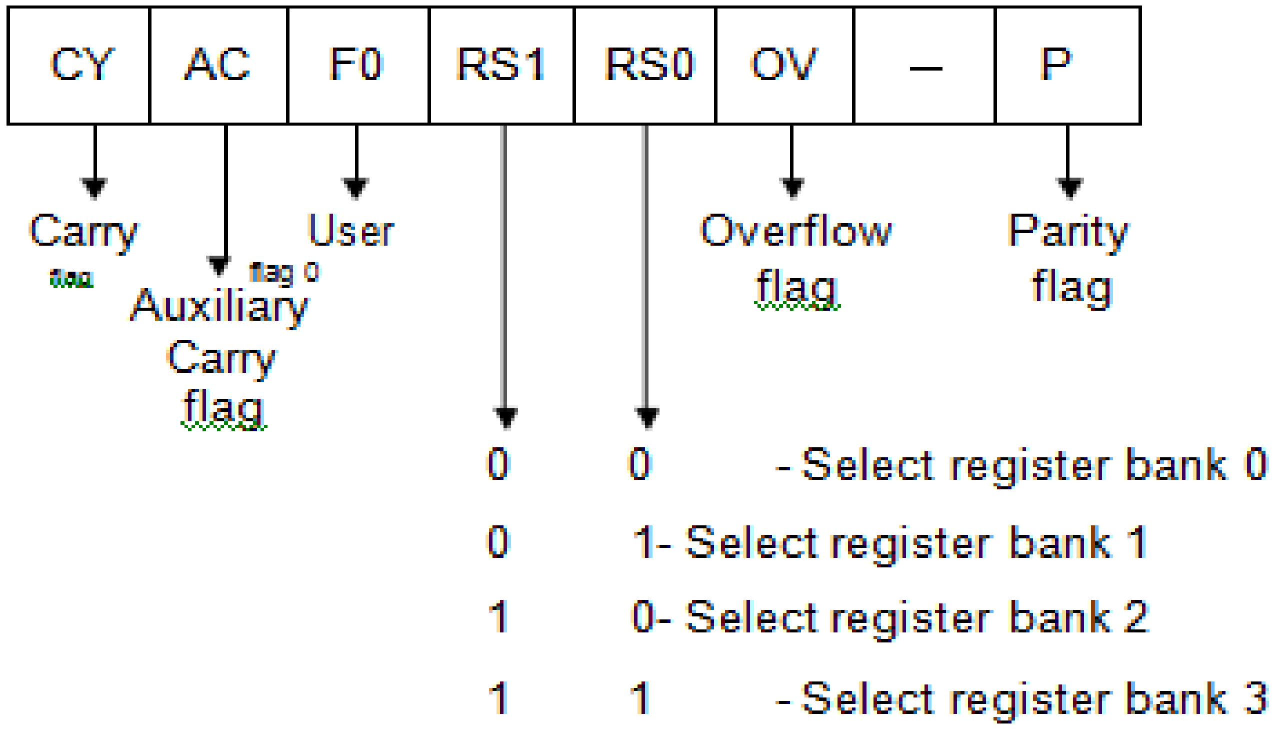
- The Program Counter (PC) is the 16 bit register giving address of next instruction to be executed during program execution.
- It always points to the program memory space.

Data Pointer Register

- The Data Pointer Register (DPTR) is the 16 bit addressing register that can be used to fetch any 8 bit data from the data memory space.
- When it is not being used for this purpose, it can be used as two eight bit registers, DPH and DPL.

Program Status Word

- The Program Status Word (PSW) keeps the current status of the arithmetic and logic operations in different bits.
- The 8051 has four math flags that respond automatically to the outcomes of arithmetic and logic operations and 3 general purpose user flags that can be set 1 or cleared to 0 by the programmer as desired.
- The math flags are carry (C), auxiliary carry (AC), overflow (OV) and parity (P).
- User flags are named flag 0 (F0), Register bank select bits RS0 and RS1.


CY
AC
F0
RS1
RS0
OV
–
P
Carry flag
Auxiliary Carry flag
User flag 0
Overflow flag
Parity flag
0 0 - Select register bank 0
0 1- Select register bank 1
1 0- Select register bank 2
1 1 - Select register bank 3

Input / Output Ports

- 8051 has 32 I/O pins configured as 4 eight bit parallel ports (P0, P1, P2 and P3).
- Each pin can be used as an input or as an output under the software control.
- These I/O pins can be accessed directly by memory instructions during program execution to get require flexibility.

Timers / Counters

- 8051 has two 16 bit Timers / Counters, T0 and T1 capable of working in different modes.
- Each consists of a ‘HIGH’ byte and a ‘LOW’ byte which can be accessed under software.
- There is a mode control register (TMOD) and a control register (TCON) to configure these timers / counters in number of ways.
- These timers are used to measure time intervals, determine pulse widths or initiate events with one microsecond resolution upto a maximum 65ms.

Serial Port

- The 8051 has a high speed full duplex serial port which is software configurable in 4 basic modes :
- Shift register mode
- Standard UART mode
- Multiprocessor mode
- 9 bit UART mode

Interrupts

- The 8051 has five interrupt sources : One from the serial port (RI / TI) when a transmission or reception operation is executed : two from the timers (TF0, TF1) when overflow occurs and two come from the two input pins INT0, INT1.
- Each interrupt may be independently enabled or disabled to allow polling on same sources and each may be classified as high or low priority.
- These operations are selected by Interrupt Enable (IE) and Interrupt Priority (IP) registers.

Oscillator and Clock

- The 8051 generates the clock pulses by which all internal operations are synchronized.
- Pins XTAL 1 and XTAL 2 are provided for connecting a resonant network to form an oscillator.
- A quartz crystal is used for oscillator.
- The crystal frequency is the basic internal clock frequency of the microcontroller.

SPECIAL FUNCTION REGISTERS (SFRS)

- The address of the Special Function Registers are above 80H, since the addresses 00H to 7FH are the addresses of RAM memory.
- The SFRs have addresses between 80H and FFH.
- But all the address space of 80H to FFH is not used by the SFRs.
- The unused locations are reserved and must not be used by the programmer.

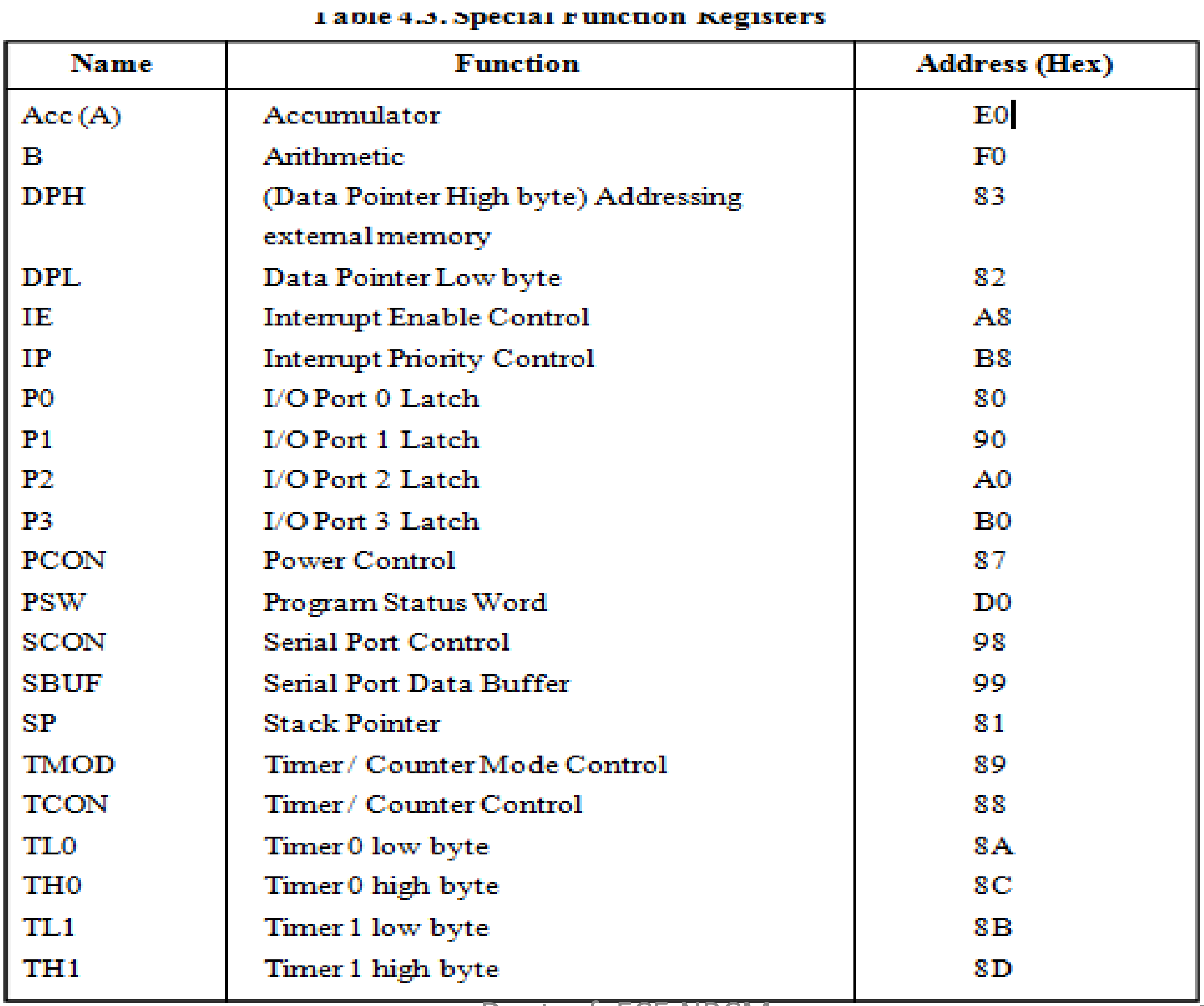


Table 4.3. Special Function Registers

Name	Function	Address (Hex)
Acc (A)	Accumulator	E0
B	Arithmetic	F0
DPH	(Data Pointer High byte) Addressing external memory	83
DPL	Data Pointer Low byte	82
IE	Interrupt Enable Control	A8
IP	Interrupt Priority Control	B8
P0	I/O Port 0 Latch	80
P1	I/O Port 1 Latch	90
P2	I/O Port 2 Latch	A0
P3	I/O Port 3 Latch	B0
PCON	Power Control	87
PSW	Program Status Word	D0
SCON	Serial Port Control	98
SBUF	Serial Port Data Buffer	99
SP	Stack Pointer	81
TMOD	Timer / Counter Mode Control	89
TCON	Timer / Counter Control	88
TL0	Timer 0 low byte	8A
TH0	Timer 0 high byte	8C
TL1	Timer 1 low byte	8B
TH1	Timer 1 high byte	8D

ADDRESSING MODES

- Immediate addressing mode
- Register addressing mode
- Direct addressing mode
- Register indirect addressing mode
- Indexed addressing mode

Immediate Addressing Mode

- When a source operand is a constant rather than a variable, then the constant can be embedded into the instruction itself.
- This kind of instructions take two bytes and first one specifies the opcode and second byte gives the required constant.
- The operand comes immediately after the opcode. The mnemonic for immediate data is the pound sign (#).
- This addressing mode can be used to load information into any of the registers including DPTR register.

Examples :

MOV A, # 18H

$$\boxed{A} \leftarrow 18H$$

MOV B, # 65H

$$\boxed{B} \leftarrow 65H$$

MOV DPTR, #2040H

$$DPL \leftarrow 40H$$

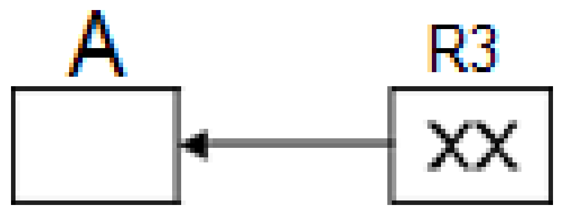
$$DPH \leftarrow 20H$$

Register Addressing Mode

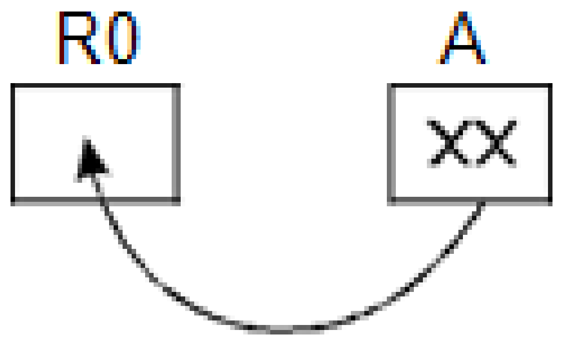
- Register addressing accesses the eight working registers (R_0 - R_7) of the selected register bank.
- The least significant three bits of the instruction opcode indicate which register is to be used for the operation.
- One of the four banks of registers is to be predefined in the PSW before using register addressing instruction.
- ACC, B and DPTR can also be addressed in this mode.

Examples :

MOV A, R3



MOV R0, A



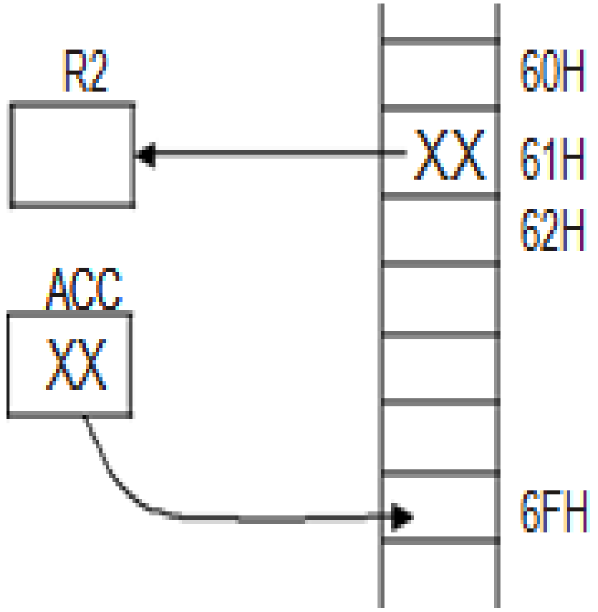
Direct Addressing Mode

- In the direct addressing mode, all 128 bytes of internal RAM and the SFRs may be addressed directly using the single - byte address assigned to each RAM location and each SFR.
- Internal RAM uses address from 00H to 7FH to address each byte.

Examples

MOV R2, 61H

MOV 6F H, A



Register Indirect Addressing Mode

- In this mode a register is used as a pointer to the data.
- If the data is inside the CPU, only registers R0 and R1 are used for this purpose.
- When R0 and R1 hold the addresses of RAM locations, they must be preceded by the "@" sign.

Examples

MOV @ R1, A	:	Move contents of A into RAM location whose address is held by R1.
MOV B, @ R0	:	Move contents of RAM location whose address is held by R0 into B.

Indexed Addressing Mode

- Only the program memory can be accessed by this mode.
- This mode is intended for reading lookup tables in the program memory.
- A 16 bit base register (DPTR or PC) points to the base of the lookup tables and accumulator carries the constant indicating table entry number.
- The address of the exact location of the table is formed by adding the accumulator data to the base pointer.

Example

MOVC A, @A + DPTR

- The contents of A are added to the DPTR to form the 16 bit address of the needed data. ‘C’ means code.

I/OPORTS

Port 0 (P0.0 - 0.7)

- Port 0 is used for both address and data bus (AD_0 – AD_7).
- When the microcontroller chip is connected to an external memory, Port 0 provides both address and data.
- ALE pin indicates if Port 0 has address or data.
- When ALE = 0, Port 0 provides data (D_0 – D_7)

  = 1, Port 0 provides address (A_0 – A_7)
- ALE is used for demultiplexing address and data with the help of a latch

Port 1 (P1.0 - P1.7)

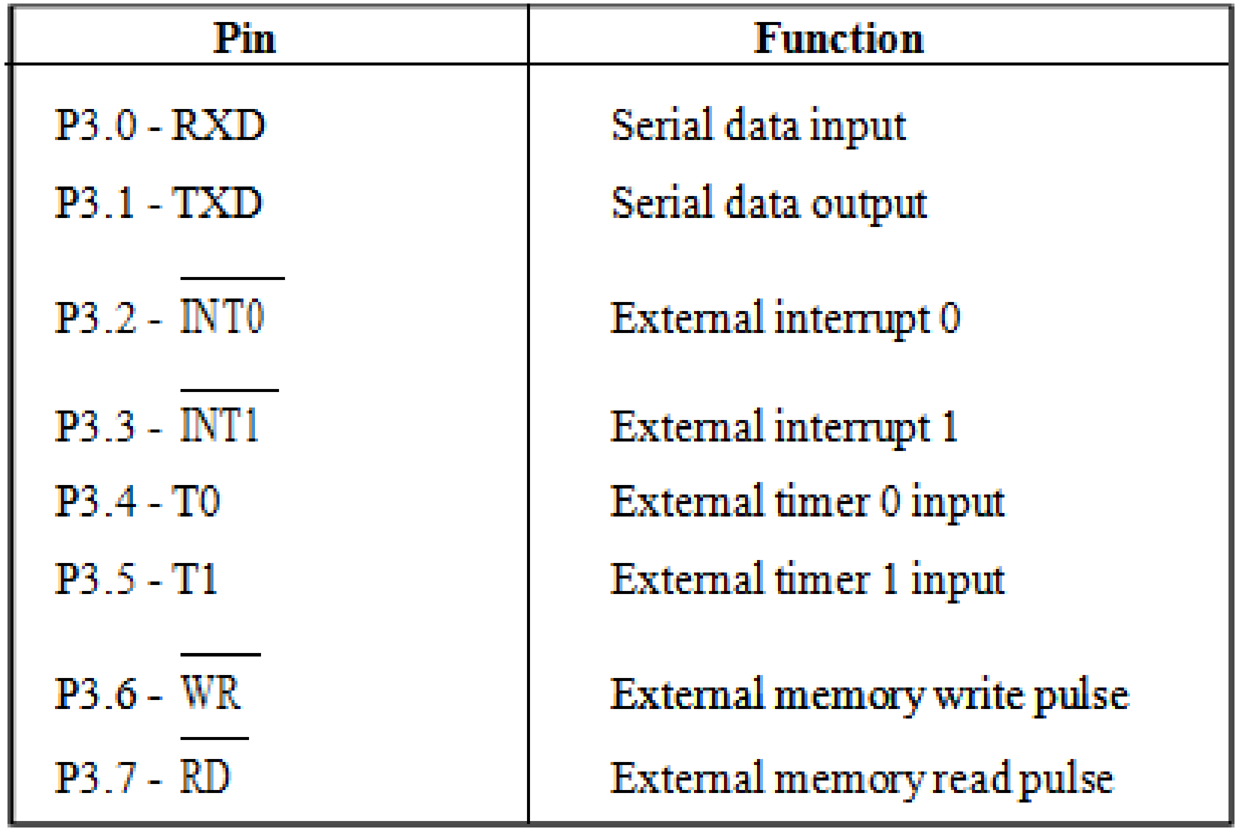
- Port 1 pins are used as input or output.
- To make port 1 as an input port, write 1 to all its 8 bits.
- To make port 1 as output port, write 0 to all its 8 bits.
- Thus port 1 pins have no dual functions.

Port 2 (P2.0 - P2.7)

- Port 2 pins are used as input / output pins similar in operation to port 1.
- The alternate use of port 2 is to supply a high order address byte (A_8 – A_{15}) when the microcontroller is connected to external memory

Port 3 (P3.0 - P3.7)

- Port 3 pins are used as input or output



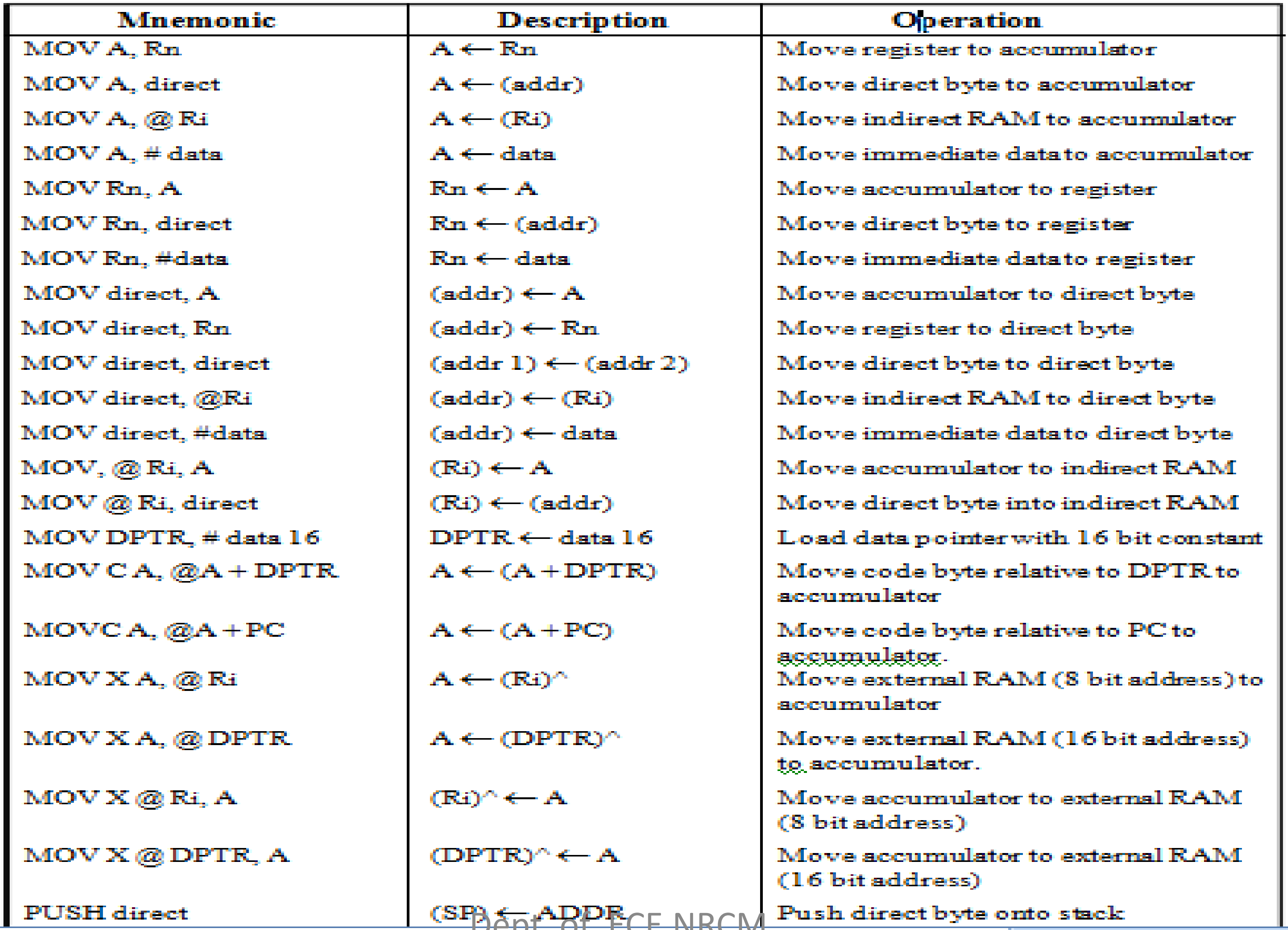
Pin	Function
P3.0 - RXD	Serial data input
P3.1 - TXD	Serial data output
P3.2 - $\overline{\text{INT0}}$	External interrupt 0
P3.3 - $\overline{\text{INT1}}$	External interrupt 1
P3.4 - T0	External timer 0 input
P3.5 - T1	External timer 1 input
P3.6 - $\overline{\text{WR}}$	External memory write pulse
P3.7 - $\overline{\text{RD}}$	External memory read pulse

INSTRUCTION SET

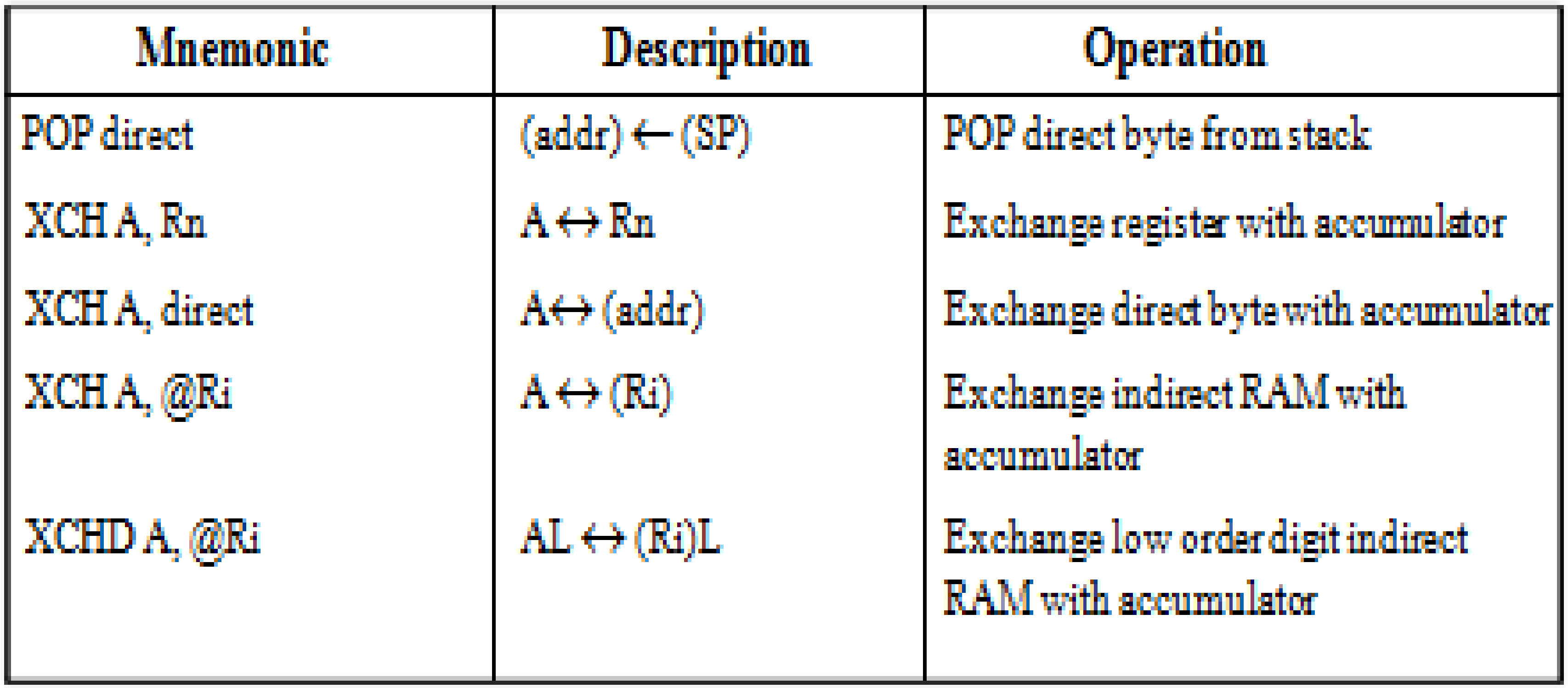
- An instruction is a command given to the computer to perform a specified operation on given data.
- The instruction set is the collection of instructions that the microcontroller is designed to execute.
- The programmer can write the program in assembly language using these instructions.

- Data transfer group
- Arithmetic group
- Logical group
- Boolean variable manipulation
- Program branching

Data Transfer Instructions

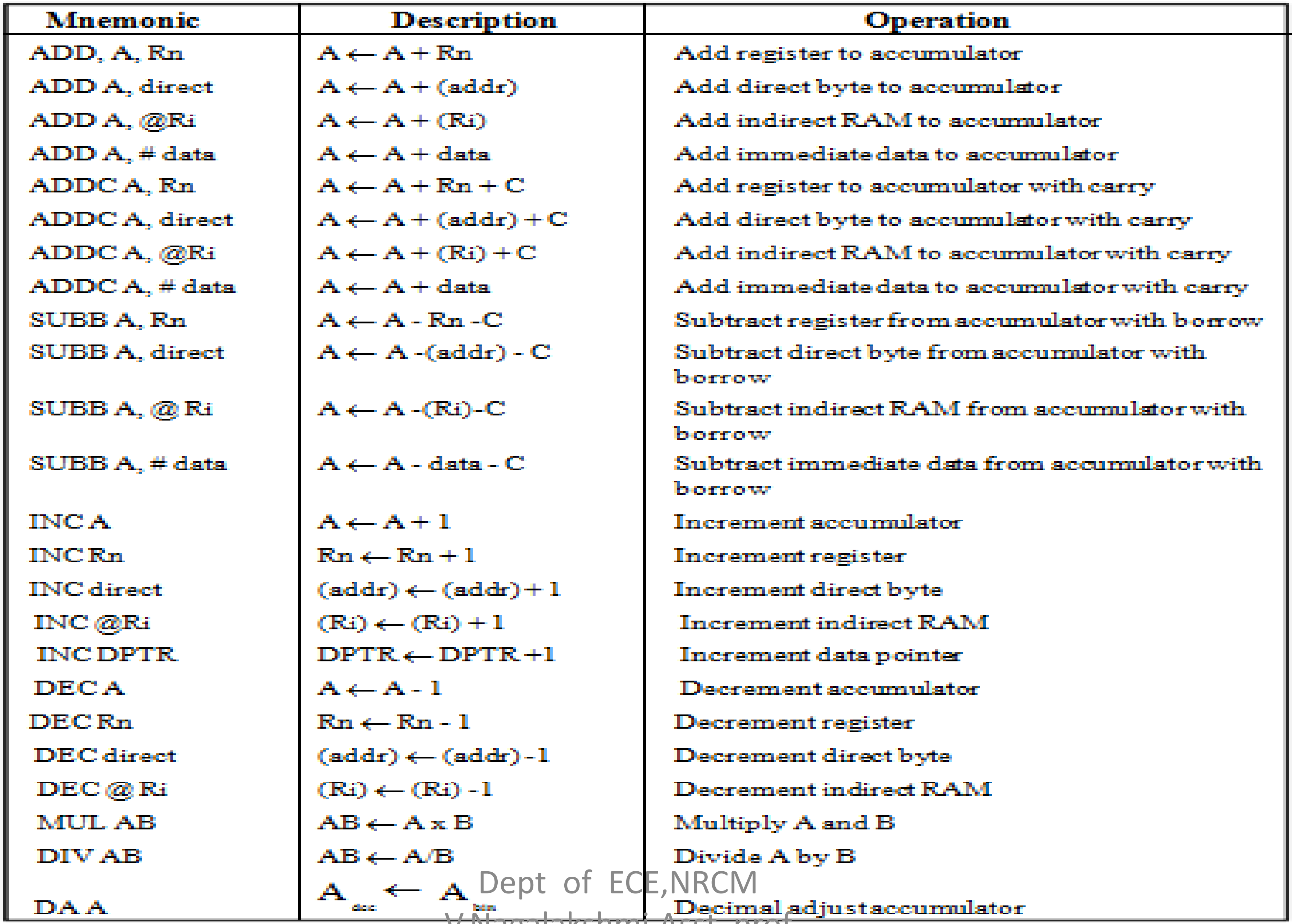


Mnemonic	Description	Operation
MOV A, Rn	A ← Rn	Move register to accumulator
MOV A, direct	A ← (addr)	Move direct byte to accumulator
MOV A, @ Ri	A ← (Ri)	Move indirect RAM to accumulator
MOV A, # data	A ← data	Move immediate data to accumulator
MOV Rn, A	Rn ← A	Move accumulator to register
MOV Rn, direct	Rn ← (addr)	Move direct byte to register
MOV Rn, #data	Rn ← data	Move immediate data to register
MOV direct, A	(addr) ← A	Move accumulator to direct byte
MOV direct, Rn	(addr) ← Rn	Move register to direct byte
MOV direct, direct	(addr 1) ← (addr 2)	Move direct byte to direct byte
MOV direct, @Ri	(addr) ← (Ri)	Move indirect RAM to direct byte
MOV direct, #data	(addr) ← data	Move immediate data to direct byte
MOV, @ Ri, A	(Ri) ← A	Move accumulator to indirect RAM
MOV @ Ri, direct	(Ri) ← (addr)	Move direct byte into indirect RAM
MOV DPTR, # data 16	DPTR ← data 16	Load data pointer with 16 bit constant
MOV C A, @A + DPTR	A ← (A + DPTR)	Move code byte relative to DPTR to accumulator
MOVC A, @A + PC	A ← (A + PC)	Move code byte relative to PC to accumulator.
MOV X A, @ Ri	A ← (Ri)^	Move external RAM (8 bit address) to accumulator
MOV X A, @ DPTR	A ← (DPTR)^	Move external RAM (16 bit address) to accumulator.
MOV X @ Ri, A	(Ri)^ ← A	Move accumulator to external RAM (8 bit address)
MOV X @ DPTR, A	(DPTR)^ ← A	Move accumulator to external RAM (16 bit address)
PUSH direct	(SP) ← ADDR	Push direct byte onto stack



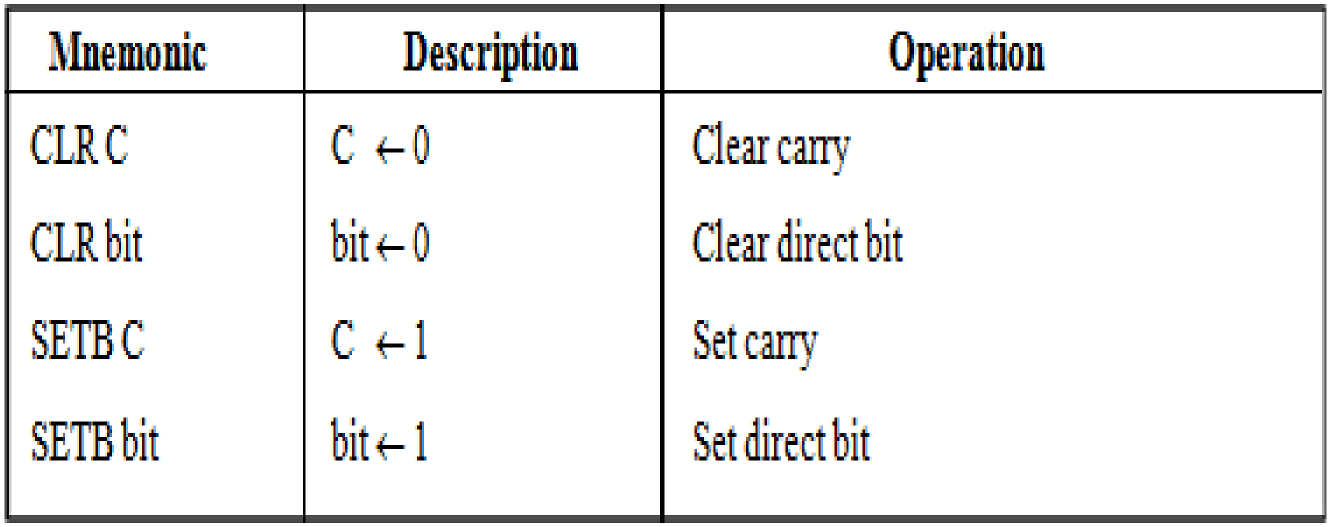
Mnemonic	Description	Operation
POP direct	(addr) ← (SP)	POP direct byte from stack
XCH A, Rn	A ↔ Rn	Exchange register with accumulator
XCH A, direct	A↔ (addr)	Exchange direct byte with accumulator
XCH A, @Ri	A ↔ (Ri)	Exchange indirect RAM with accumulator
XCHD A, @Ri	AL ↔ (Ri)L	Exchange low order digit indirect RAM with accumulator

ARITHMETIC INSTRUCTIONS

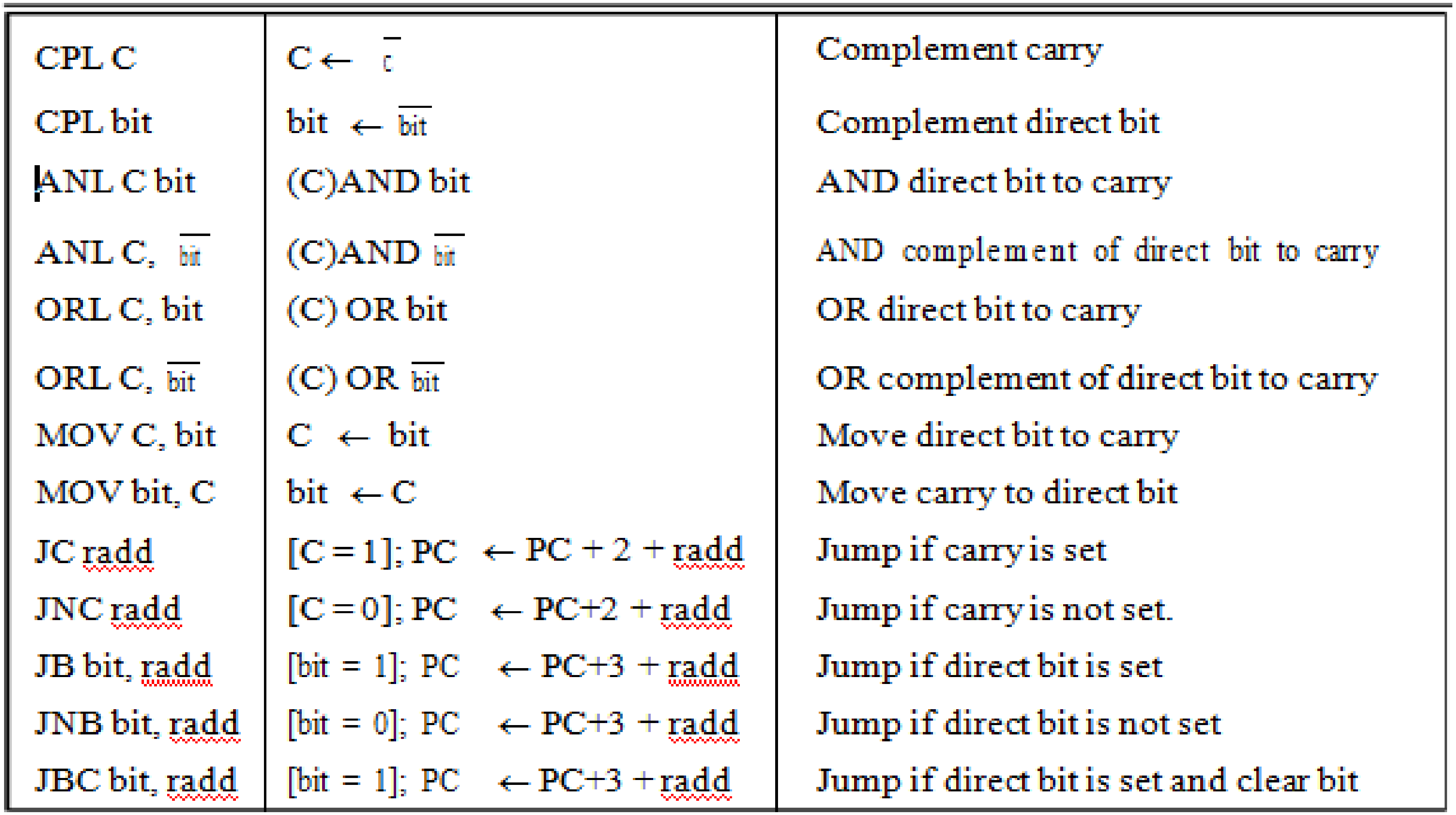


Mnemonic	Description	Operation
ADD, A, Rn	A ← A + Rn	Add register to accumulator
ADD A, direct	A ← A + (addr)	Add direct byte to accumulator
ADD A, @Ri	A ← A + (Ri)	Add indirect RAM to accumulator
ADD A, # data	A ← A + data	Add immediate data to accumulator
ADDC A, Rn	A ← A + Rn + C	Add register to accumulator with carry
ADDC A, direct	A ← A + (addr) + C	Add direct byte to accumulator with carry
ADDC A, @Ri	A ← A + (Ri) + C	Add indirect RAM to accumulator with carry
ADDC A, # data	A ← A + data	Add immediate data to accumulator with carry
SUBB A, Rn	A ← A - Rn -C	Subtract register from accumulator with borrow
SUBB A, direct	A ← A -(addr) - C	Subtract direct byte from accumulator with borrow
SUBB A, @ Ri	A ← A -(Ri)-C	Subtract indirect RAM from accumulator with borrow
SUBB A, # data	A ← A - data - C	Subtract immediate data from accumulator with borrow
INC A	A ← A + 1	Increment accumulator
INC Rn	Rn ← Rn + 1	Increment register
INC direct	(addr) ← (addr) + 1	Increment direct byte
INC @Ri	(Ri) ← (Ri) + 1	Increment indirect RAM
INC DPTR	DPTR ← DPTR +1	Increment data pointer
DEC A	A ← A - 1	Decrement accumulator
DEC Rn	Rn ← Rn - 1	Decrement register
DEC direct	(addr) ← (addr) -1	Decrement direct byte
DEC @ Ri	(Ri) ← (Ri) -1	Decrement indirect RAM
MUL AB	AB ← A x B	Multiply A and B
DIV AB	AB ← A/B	Divide A by B
DA A	$A_{dec} \leftarrow A_{bin}$	Decimal adjust accumulator

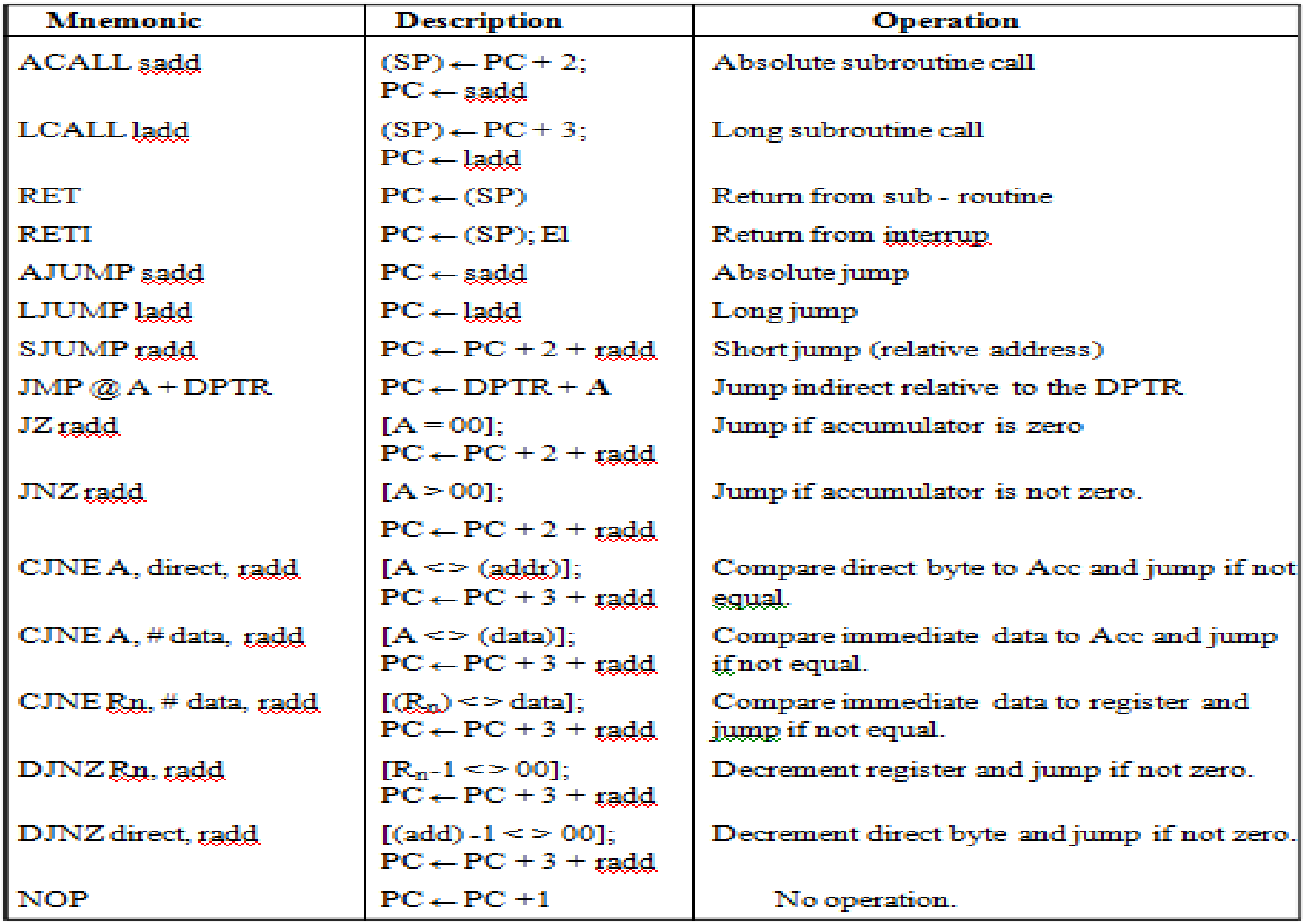
Mnemonic	Description	Operation
ANL., A, Rn	(A) AND (Rn)	AND register to accumulator
ANL A, direct	(A) AND (addr)	AND direct byte to accumulator
ANL A, @Ri	(A) AND ((Ri))	AND indirect RAM to accumulator
ANL A, #data	(A) AND data	AND immediate data to accumulator
ANL direct, A	(addr) AND (A)	AND accumulator to direct byte
ANL direct, #data	(addr) AND data	AND immediate data to direct byte
ORL A, Rn	(A) OR (Rn)	OR register to accumulator
ORL A, direct	(A) OR (addr)	OR direct byte to accumulator
ORL A, @Ri	(A) OR ((Ri))	OR indirect RAM to accumulator
ORL A, #data	(A) OR data	OR immediate data to accumulator
ORL direct, A	(addr) OR (A)	OR accumulator to direct byte
ORL direct, # data	(addr) OR data	OR immediate data to direct byte
XRL A, Rn	(A) XOR (Rn)	Ex - OR register to accumulator
XRL A, direct	(A) XOR (addr)	EX - OR direct byte to accumulator
XRL A, @Ri	(A) XOR ((Ri))	EX - OR indirect RAM to accumulator
XRL A, #data	(A) XOR data	EX - OR immediate data to accumulator
XRL direct, A	(addr) XOR (A)	EX - OR accumulator to direct byte
XRL direct, #data	(addr) XOR data	EX - OR immediate data to direct byte
RL A	$A_0 \leftarrow A_7 \leftarrow A_6 \ldots \leftarrow A_1 \leftarrow A_0$	Rotate accumulator left
RLC A	$C \leftarrow A_7 \leftarrow A_6 \ldots \leftarrow A_0 \leftarrow C$	Rotate accumulator left through carry
RR A	$A_0 \rightarrow A_7 \rightarrow A_6 \ldots \rightarrow A_1 \rightarrow A_0$	Rotate accumulator right
RRC A	$C \rightarrow A_7 \rightarrow A_6 \ldots \rightarrow A_0 \rightarrow C$	Rotate accumulator right through carry
CLR A	$A \leftarrow 00$	Clear accumulator
CPL A	$A \leftarrow \bar{A}$	Complement accumulator
SWAP A	$A_L \leftrightarrow A_H$	Swap nibbles within the accumulator.



Mnemonic	Description	Operation
CLR C	C ← 0	Clear carry
CLR bit	bit ← 0	Clear direct bit
SETB C	C ← 1	Set carry
SETB bit	bit ← 1	Set direct bit



CPL C	C ← $\overline{C}$	Complement carry
CPL bit	bit ← $\overline{\text{bit}}$	Complement direct bit
ANL C bit	(C)AND bit	AND direct bit to carry
ANL C, $\overline{\text{bit}}$	(C)AND $\overline{\text{bit}}$	AND complement of direct bit to carry
ORL C, bit	(C) OR bit	OR direct bit to carry
ORL C, $\overline{\text{bit}}$	(C) OR $\overline{\text{bit}}$	OR complement of direct bit to carry
MOV C, bit	C ← bit	Move direct bit to carry
MOV bit, C	bit ← C	Move carry to direct bit
JC radd	[C = 1]; PC ← PC + 2 + radd	Jump if carry is set
JNC radd	[C = 0]; PC ← PC+2 + radd	Jump if carry is not set.
JB bit, radd	[bit = 1]; PC ← PC+3 + radd	Jump if direct bit is set
JNB bit, radd	[bit = 0]; PC ← PC+3 + radd	Jump if direct bit is not set
JBC bit, radd	[bit = 1]; PC ← PC+3 + radd	Jump if direct bit is set and clear bit



Mnemonic	Description	Operation
ACALL sadd	(SP) $\leftarrow$ PC + 2; PC $\leftarrow$ sadd	Absolute subroutine call
LCALL ladd	(SP) $\leftarrow$ PC + 3; PC $\leftarrow$ ladd	Long subroutine call
RET	PC $\leftarrow$ (SP)	Return from sub - routine
RETI	PC $\leftarrow$ (SP); EI	Return from interrup
AJUMP sadd	PC $\leftarrow$ sadd	Absolute jump
LJUMP ladd	PC $\leftarrow$ ladd	Long jump
SJUMP radd	PC $\leftarrow$ PC + 2 + radd	Short jump (relative address)
JMP @ A + DPTR	PC $\leftarrow$ DPTR + A	Jump indirect relative to the DPTR
JZ radd	[A = 00]; PC $\leftarrow$ PC + 2 + radd	Jump if accumulator is zero
JNZ radd	[A > 00]; PC $\leftarrow$ PC + 2 + radd	Jump if accumulator is not zero.
CJNE A, direct, radd	[A <> (addr)]; PC $\leftarrow$ PC + 3 + radd	Compare direct byte to Acc and jump if not equal.
CJNE A, # data, radd	[A <> (data)]; PC $\leftarrow$ PC + 3 + radd	Compare immediate data to Acc and jump if not equal.
CJNE Rn, # data, radd	[(R_n) <> data]; PC $\leftarrow$ PC + 3 + radd	Compare immediate data to register and jump if not equal.
DJNZ Rn, radd	[R_n-1 <> 00]; PC $\leftarrow$ PC + 3 + radd	Decrement register and jump if not zero.
DJNZ direct, radd	[(add) -1 <> 00]; PC $\leftarrow$ PC + 3 + radd	Decrement direct byte and jump if not zero.
NOP	PC $\leftarrow$ PC +1	No operation.

UNIT – 3

INTERFACING MICROCONTROLLER

PROGRAMMING 8051 TIMERS

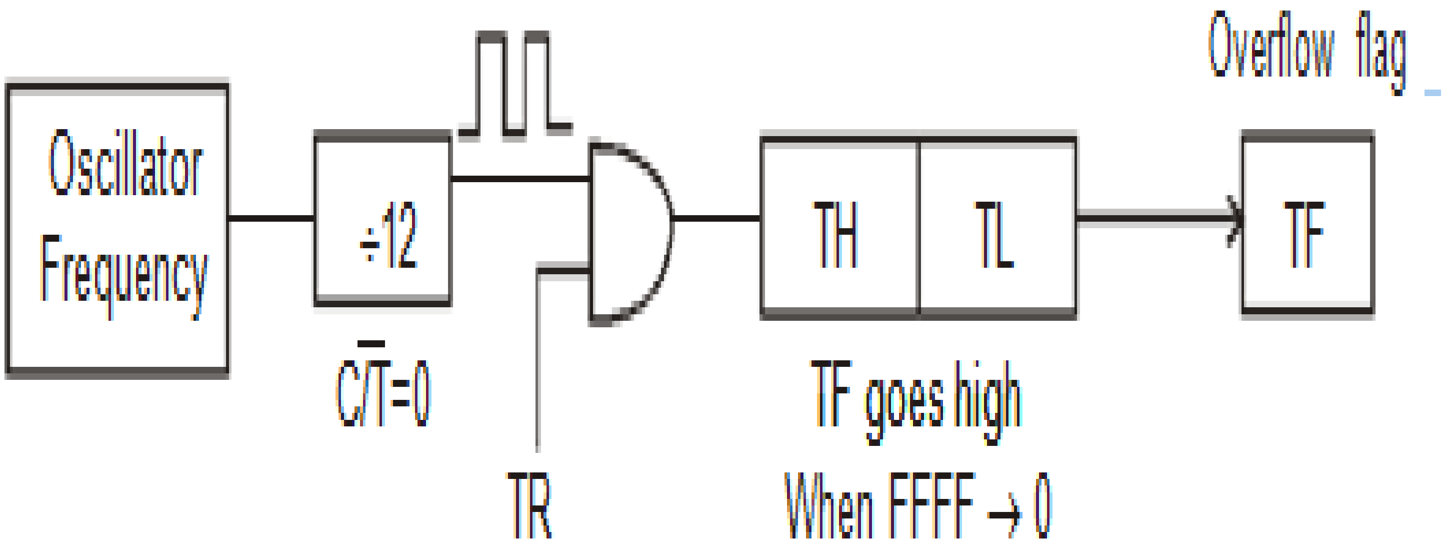
Mode 1 Programming

Operations of mode 1:

- It allows values of **0000 H to FFFF H to be loaded** into the timer's registers TL and TH.
- After TH and TL are loaded with a 16 - bit initial value, the timer must be started.
- This is done by "SET B TR0" for Timer 0 and "SET B TR1" for Timer 1.

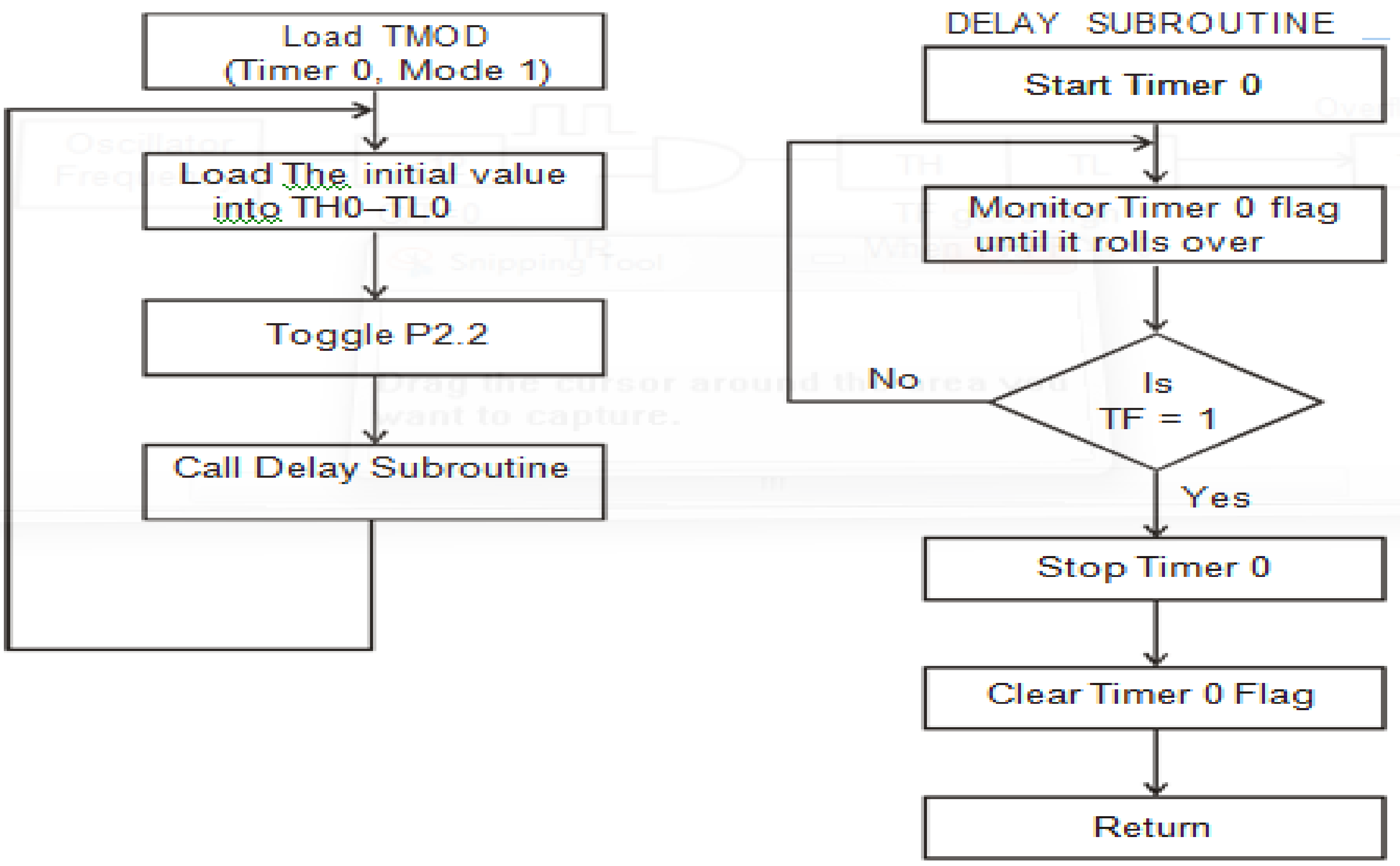
- After the timer is started, it starts to count up. It counts up until it reaches its limit of FFFF H. When it rolls over from FFFF H to 0000H, it sets high a flag bit called TF (Timer Flag). This timer flag can be monitored. When this timer flag is raised, one option would be to stop the timer with the instructions “CLR TR0” or “CLR TR1” for Timer 0 and Timer 1 respectively.
- After the timer reaches its limit and rolls over to repeat the process the registers TH and TL must be reloaded with the original value and TF must be reset to 0

TIMER FOR MODE 1

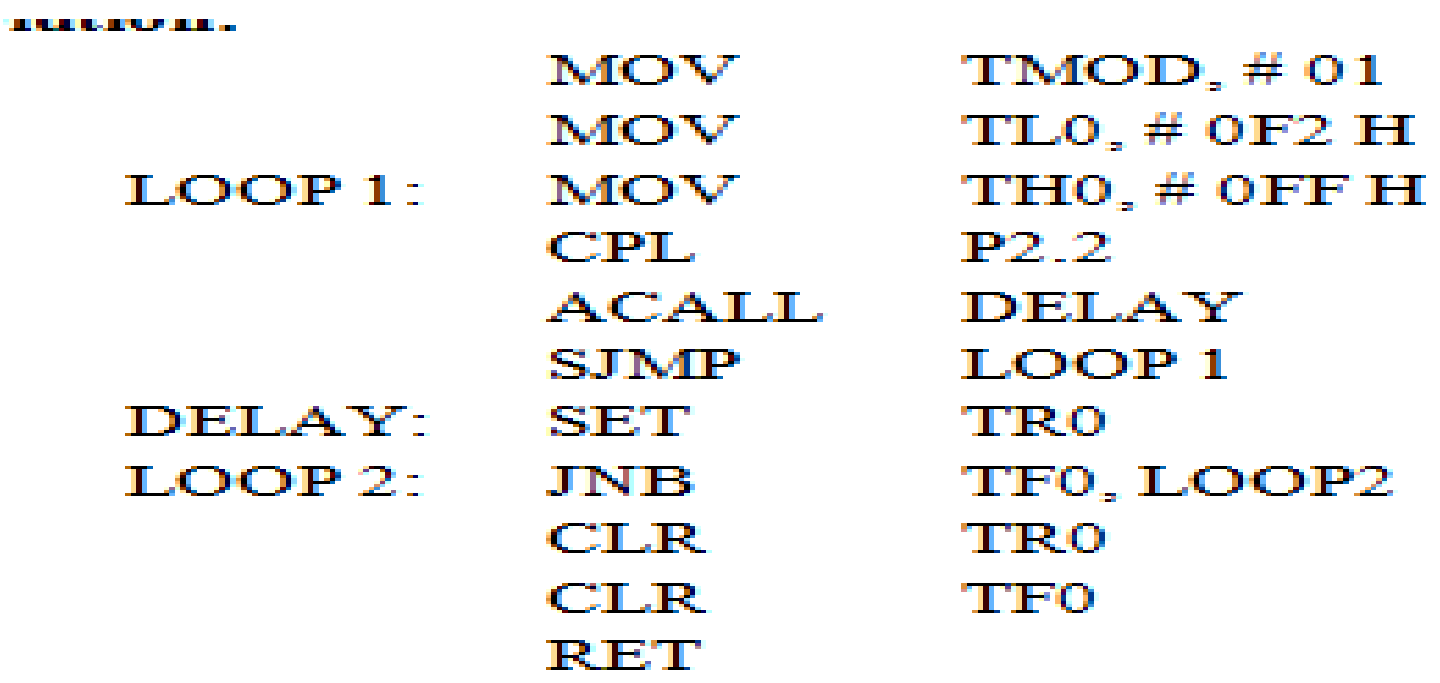


PROCEDURE

- Load the TMOD value register indicating which timer (Timer 0 or Timer 1) is to be used and which timer mode (0 or 1) is selected.
- Load registers TL and TH with initial count values.
- Start the Timer.
- Keep monitoring the timer flag (TF). When TF becomes high get out of the loop.
- Stop the timer.
- Clear the TF flag for the next round.


Load TMOD (Timer 0, Mode 1)
Load The initial value into TH0–TL0
Toggle P2.2
Call Delay Subroutine
DELAY SUBROUTINE
Start Timer 0
Monitor Timer 0 flag until it rolls over
Is TF = 1
No
Yes
Stop Timer 0
Clear Timer 0 Flag
Return

PROGRAM



```
lution:
                    MOV         TMOD, # 01
                    MOV         TL0, # 0F2 H
     LOOP 1:        MOV         TH0, # 0FF H
                    CPL         P2.2
                    ACALL       DELAY
                    SJMP        LOOP 1
     DELAY:         SET         TR0
     LOOP 2:        JNB         TF0, LOOP2
                    CLR         TR0
                    CLR         TF0
                    RET
```

TMOD register :

Timer 1				Timer 0				
GATE	C/T	M1	M0	GATE	C/T	M1	M0	
0	0	0	0	0	0	0	1	=01 H

Timer 0, Mode 1 –16 bit Timer mode.

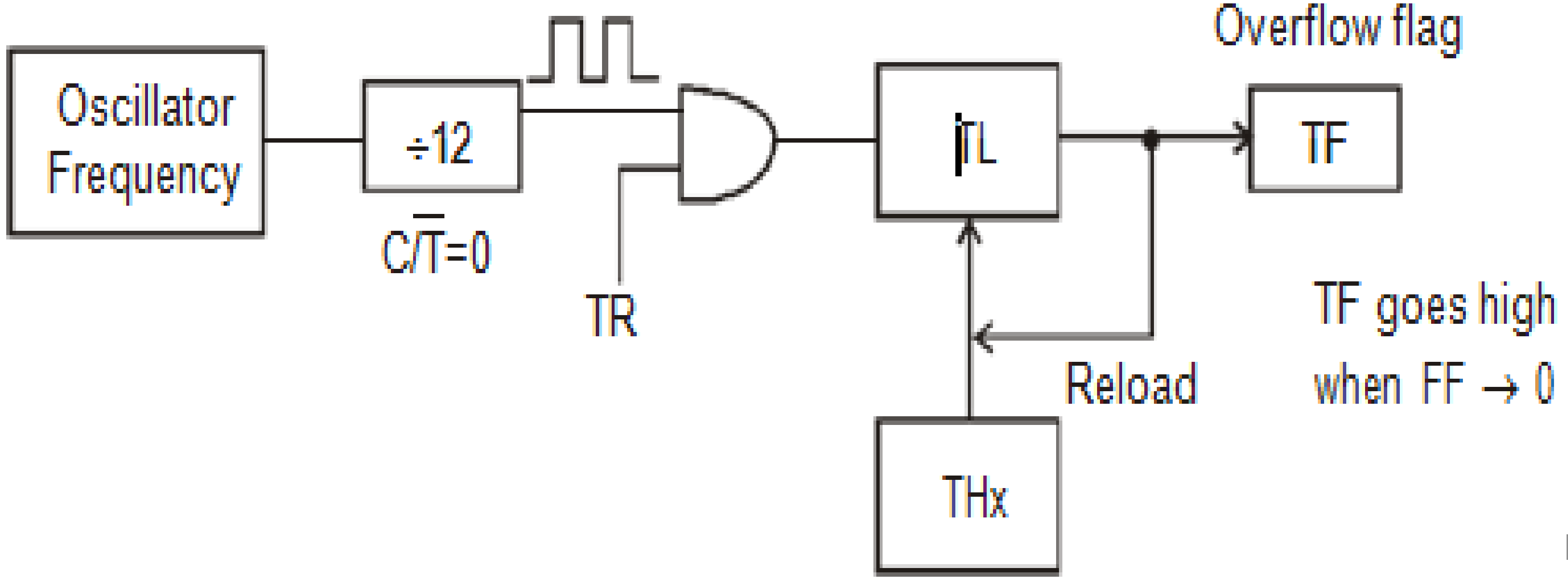
Program :

```
            CLR       P1.4
            MOV       TMOD,    #01 H
LOOP 1:     MOV       TL0,     #0B0 H
            MOV       TH0,     #3C H
            SET B     P1.4
            SET B     TR0
LOOP2:      JNB       TF0, LOOP2
            CLR       TR0
            CLR       TF0
            CLR       P1.4
            LJMP      LOOP1
```

Mode 2 Programming

Operations of Mode 2:

- Mode 2 allows only values of 00 H to FF H to be loaded into the timer's register TH.
- After TH is loaded with the 8 bit value, the 8051 gives a copy of it to TL. Then the timer must be started. This is done by "SET B TR0" for Timer 0 and "SET B TR 1" for Timer 1.
- After the timer is started, it started it starts to count up by incrementing the TL register. It counts up until it reaches its limit of FFH. When it rolls over from FFH to 00H, it sets high the timer flag (TF) TF0 is raised for Timer 0 and TF 1 is raised for Timer 1.
- When the TL register rolls from FF H to 00 H and TF is set to 1, TL is reloaded automatically with the original value kept by the TH register. To repeat the process clear TF (anti - reloading).


Oscillator Frequency
÷12
$C/\overline{T}=0$
TR
TL
Overflow flag
TF
Reload
THx
TF goes high when FF → 0

PROCEDURE

- Load the TMOD value register indicating which timer (Timer 0 or 1) is to be used and select the timer mode 2.
- Load the TH registers with the initial count value.
- Start the timer.
- Keep monitoring the timer flag (TF) with “JNB TFx” instruction. When TF becomes high get out of the loop.
- Clear the TF flag
- Go back to step 4, since Mode 2 is auto - reload.

TMOD register :

Timer 1				Timer 0			
GATE	C/T	M1	M0	GATE	C/T	M1	M0
0	0	1	0	0	0	0	0

=20 H

Timer 1, Mode 2 – Auto reload

Program :

```
            MOV     TMOD, # 20H
            MOV     TH1, #6
            SETB    TR1
LOOP:       JNB     JF1, LOOP
            CPL     P1.3
            CLR     TF 1
            SJMP    LOOP
```

COUNTER PROGRAMMING

- When C/T = 1, the timer is used as a counter and gets its pulses from outside the 8051. The counter counts up as pulses are fed from pins T0 (Timer 0 input) and T1 (Timer 1 input). These two pins belong to port 3. For Timer 0, when C/T = 1 pin 3.4 provides the clock pulse and counter counts up for each clock pulse coming from that pin.
- Similarly for Timer 1, when C/T = 1 each clock pulse coming in from pin 3.5 makes the counter countup.

  P3.4 - T0 - Timer/Counter 0 external input

  P3.5 - T1 - Timer/Counter 1 external input
- In counter mode, the TMOD, TH and TL registers are the same as for the timer. Counter programming also same as timer programming.

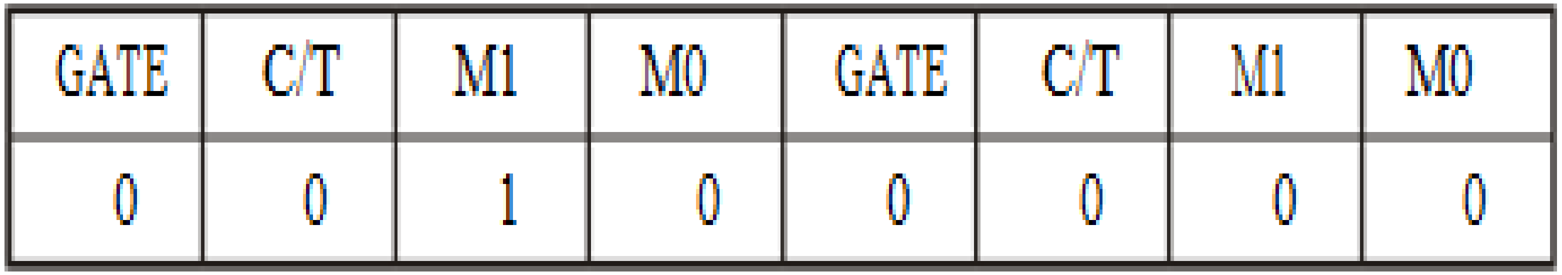
```
         MOV     TMOD, # 0110 0000 B   ;  Counter 1, Mode 2, C/T =1
         MOV     TH 1, # 00 H          ;  Clear TH 1
         SET B   P3.5                  ;  Make T1 input
LOOP 1:  SET B   TR 1                  ;  Start the counter
LOOP 2:  MOV     A, TL 1               ; Get copy of count TL 1
         MOV     P2, A                 ;  Display it on Port 2
         JNB     TF 1, LOOP 2          ; Goto Loop 2 if TF = 0
         CLR     TR1                   ;  Stop the counter 1
         CLR     TF 1                  ;  Make TF = 0
         SJM P   LOOP 1                ;  Jump to Loop 1.
```

SERIAL PORT PROGRAMMING

Programming the 8051 to transfer data serially

- The TMOD register is loaded with the value 20H, indicating the use of Timer 1 in mode 2

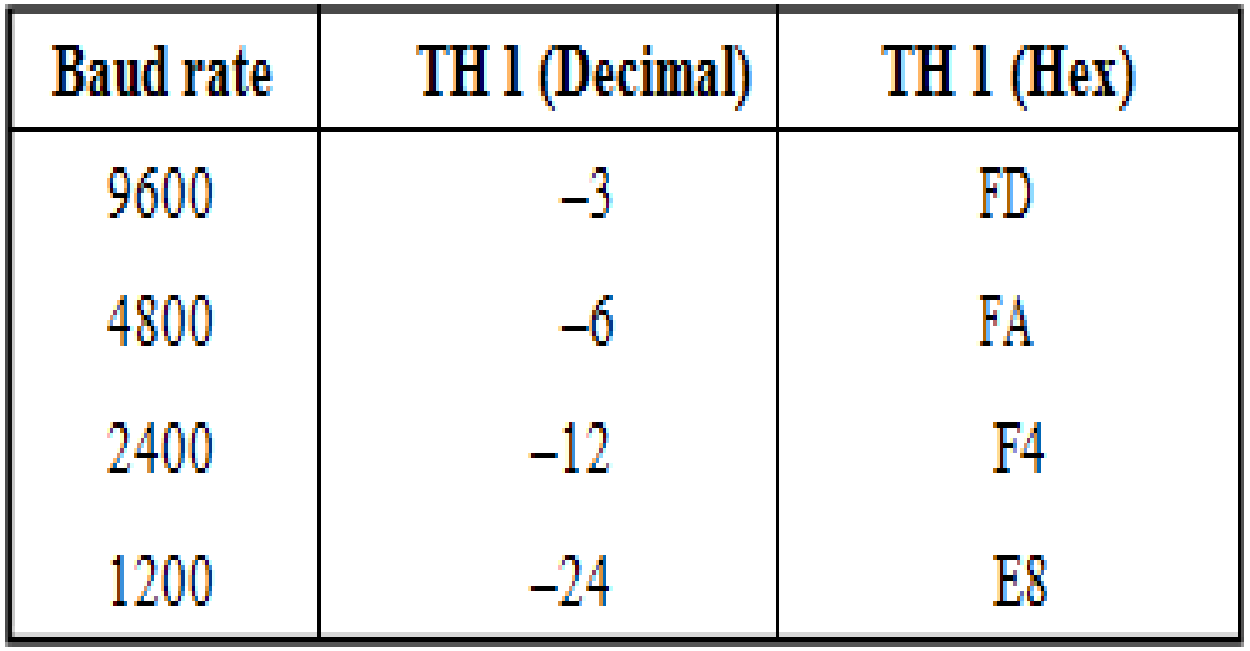
TMOD Register



GATE	C/T	M1	M0	GATE	C/T	M1	M0
0	0	1	0	0	0	0	0

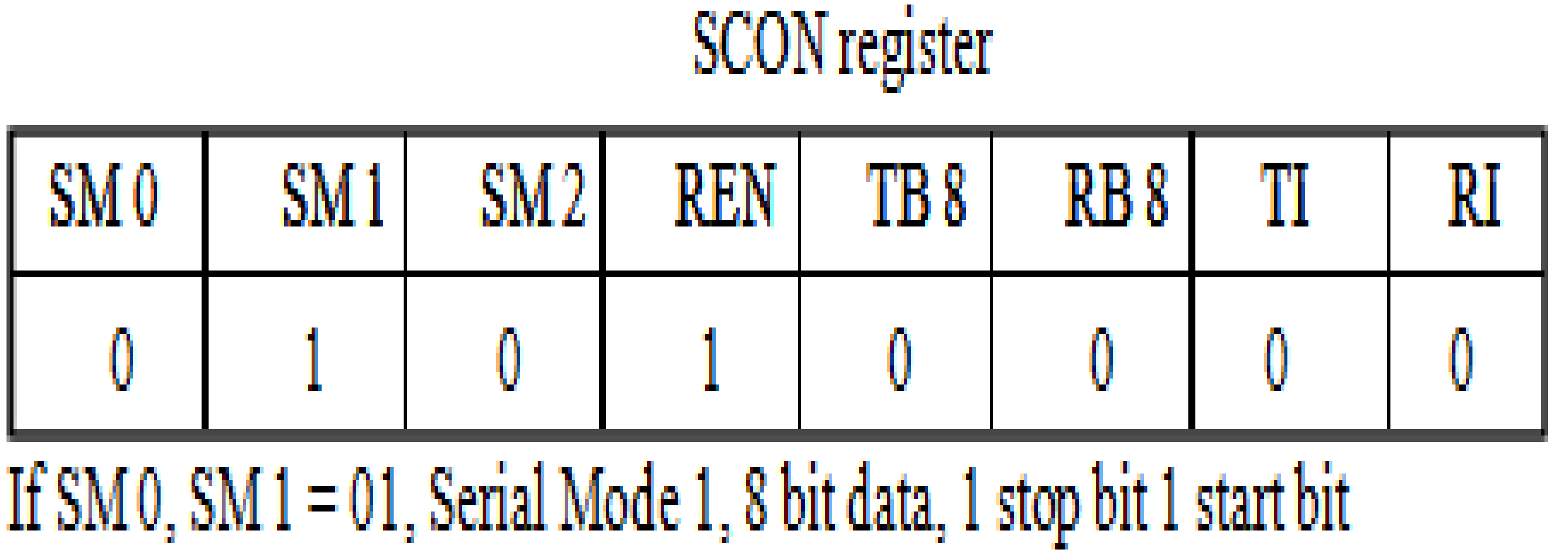
If M_1 M_0 = 10, 8 bit Auto - reload counter

- The TH 1 is loaded with one of the values in Table to set the baud rate for serial data transfer.



Baud rate	TH 1 (Decimal)	TH 1 (Hex)
9600	–3	FD
4800	–6	FA
2400	–12	F4
1200	–24	E8

- The SCON register is loaded with the value 50 H, indicating serial mode 1, where 8-bit data is framed with start and stop bits.



SCON register

SM 0	SM 1	SM 2	REN	TB 8	RB 8	TI	RI
0	1	0	1	0	0	0	0

If SM 0, SM 1 = 01, Serial Mode 1, 8 bit data, 1 stop bit 1 start bit

- TR 1 is set to start Timer 1.
- TI is cleared by the “CLR TI” instruction.
- The character byte to be transferred serially is written into the SBUF registers.
- The TI flag bit is monitored with the use of the instruction “JNB TI, XX ” to see if the character has been transferred completely.
- To transfer next character, go to step 5.

Program

Write an ALP to transfer letter ‘E’ serially at 4800 baud continuously.

Solution:

```
          MOV     TMOD, # 20 H     ;    Timer 1, Mode 2 (Auto-reload)
          MOV     TH 1 #-6         ;    4800 baud rate
          MOV     SCON, # 50 H     ;  8-bit, 1 stop, 1 start, REN enabled
          SET B   TR 1             ;    Start Timer 1
LOOP 1:   MOV     SBUF, # ‘E’      ;    Letter ‘E’ to be transferred
LOOP 2:   JNB     TI, LOOP2        ;  Wait for the last bit
          CLR     TI               ;    Clear TI for next character
          SJMP    LOOP 1           ;    Go to Loop 1 for sending ‘E’
```

Programming the 8051 to receive data serially

- The first 4 steps are as same in programming to transfer data serially.
- RI is cleared with “CLR RI “ instruction.
- The RI flag bit is monitored with the use of the instruction “JNB RI, XX” to see if the character has been received yet.
- When RI is raised, SBUF has the byte. Its contents are moved into a safe place.
- To receive the next character, go to step 5.

Program

Write an ALP to receive bytes of data serially and put them in Port 2. Set the baud rate at 2400, 8 bit data and 1 stop bit.

Solution:

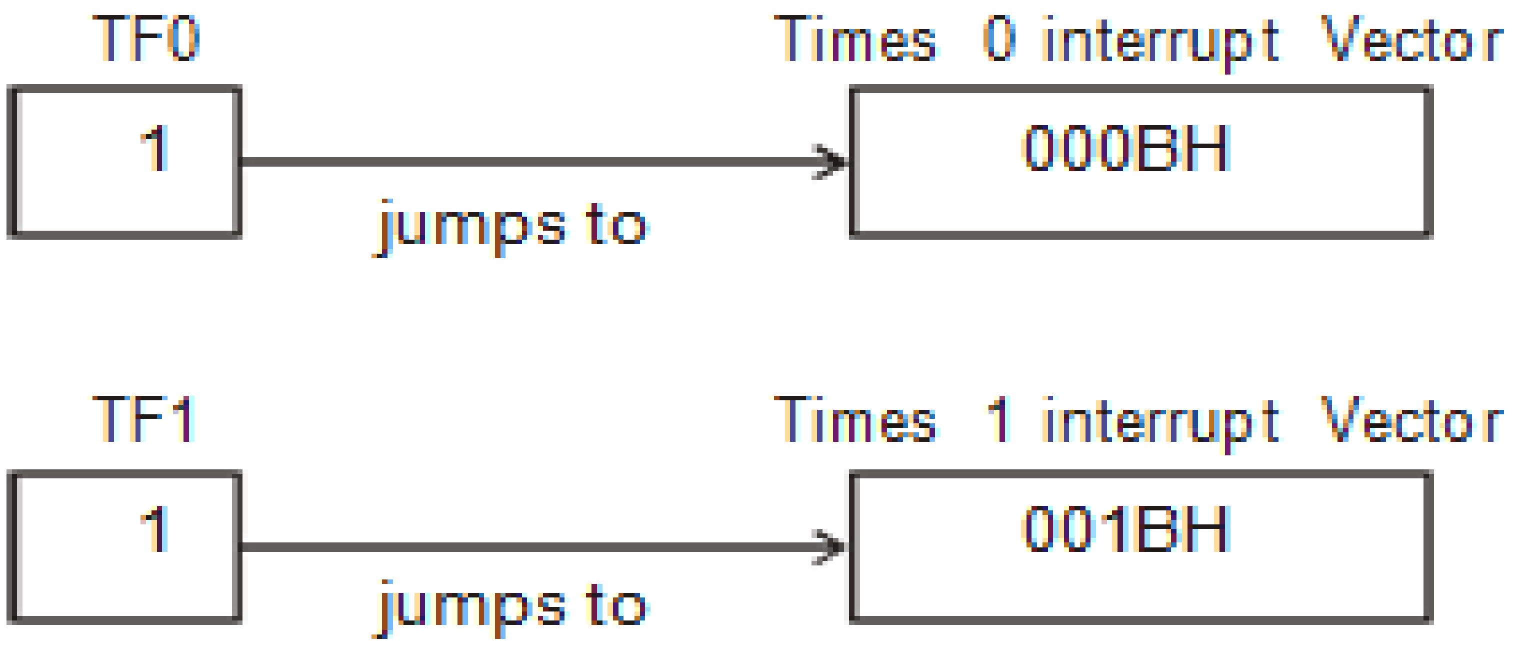
```
            MOV     TMOD, # 20 H    ;   Timer 1, mode 2
            MOV     TH 1, # F4 H    ;  For 2400 baud TH1=–12 (F4 H)
            MOV     SCON, # 50 H    ;  8-bit, 1 stop, REN enabled
            SET B   TR 1            ;   Start Timer 1
LOOP 1;  JNB        RI, LOOP 1      ;   Wait for character to come in
            MOV     A, SBUF         ;   Save incoming byte in A
            MOV     P2, A           ;   Send to Port 2
            CLR     RI              ;  Get ready to receive next byte
            SJMP    LOOP 1          ;   Go to Loop 1 , to keep getting data.
```

INTERRUPT PROGRAMMING

- An interrupt is an internal or external event that interrupts the microcontroller to inform it that a device needs its service. Every interrupt has a program associated with it called the interrupt service routine (ISR).
- The 8051 has 6 interrupts:
- Reset
- Timer interrupts :Timer 0 interrupt and Timer 1 interrupt
- External hardware interrupts : INT 0 INT 1
- Serial communication interrupt
- The 8051 can be programmed to enable or disable an interrupt and the interrupt priority can be altered. Register IE is responsible for enabling and disabling the interrupts.

Programming Timer Interrupts

- The timer flag (TF) is raised when the timer rolls over. In polling TF, we have to wait until the TF is raised.
- In problem with polling method is that the microcontroller is tied down while waiting for TF to be raised and cannot do anything else.
- Using interrupts solves this problem and avoids tying down the microcontroller.
- If the timer interrupt in the IE register is enabled, whenever the timer rolls over, TF is raised and the microcontroller is interrupted in whatever it is doing and jumps to the interrupt vector table to service the ISR.
- In this way the microcontroller can do other things until it is notified that the timer has rolled over.

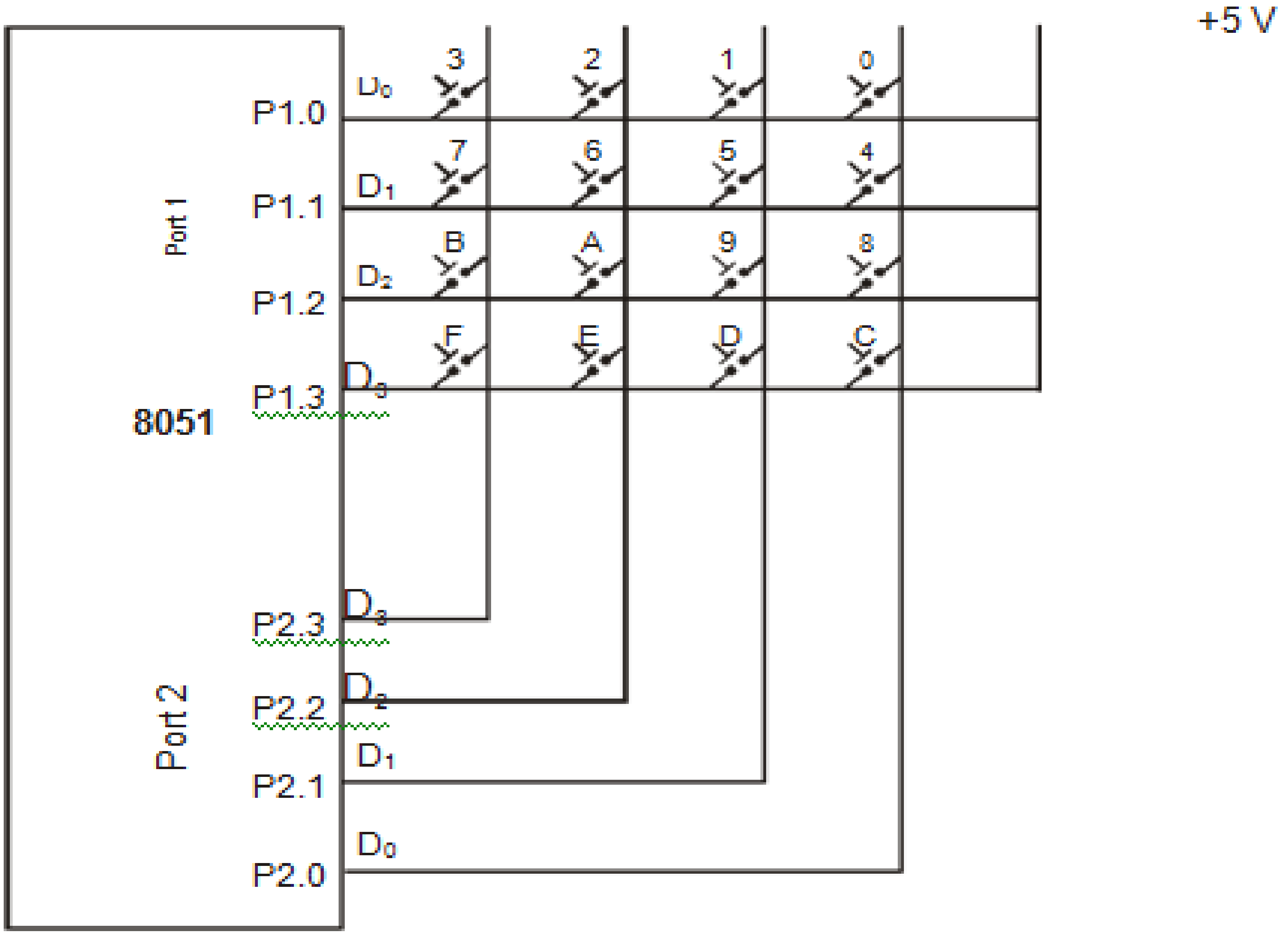

TF0
1
jumps to
Times 0 interrupt Vector
000BH
TF1
1
jumps to
Times 1 interrupt Vector
001BH

Programming External Hardware Interrupts

- The 8051 has two external hardware interrupts INT 0 and INT 1.
- Upon activation of these interrupts through Port pins P3.2 and P3.3, the 8051 gets interrupted in whatever it is doing and jumps to the interrupt vector table to perform the interrupt service routine (ISR).
- There are two types of activation for the external hardware interrupts: Level triggered and Edge triggered.

KEYBOARD INTERFACING

- The rows are connected to an output port and the columns are connected to an input port.
- When a key is pressed, a row and a column make a contact, otherwise there is no connection between rows and columns.
- If all the rows are grounded and a key is pressed, one of the columns will have 0 since the key pressed provides the path to ground.
- If no key has been pressed, reading the input port will yield 1s for all columns since they are connected to Vcc.


+5 V
8051
Port 1
Port 2
P1.0
P1.1
P1.2
P1.3
P2.3
P2.2
P2.1
P2.0
D0
D1
D2
D3
D3
D2
D1
D0
3
2
1
0
7
6
5
4
B
A
9
8
F
E
D
C

- If any key is pressed, the columns are scanned again and again until one of them has a 0 on it.
- After the key press detection, it waits 20 milli seconds for the bounce and then scans the columns again.
- After 20 ms delay, the key is still pressed, it goes to detect which row it belongs to. To detect the row it grounds one row at a time, reading the columns each time.
- If all columns are high, the pressed key cannot belong to that row. Therefore it grounds the next row and continues until it finds the row the key press belongs to.

- After finding the row, it sets up the starting address for the look-up table holding the ASCII codes for that row and goes to the next stage to identify the key.
- Now it rotates the column bits, one bit at a time into the carry flag and checks if it is low.
- When carry flag is zero, it pulls out the ASCII code for that key from look-up table; otherwise it increments the pointer to point to the next element of the look-up table.

Program

Write 8051 ALP to interface 4x4 matrix keyboard.

Solution :

```
ROW_1 :     MOV DPTR, #KEY1
             SJMP FIND
ROW_2 :     MOV DPTR, #KEY2
             SJMP FIND
ROW_3 :     MOV DPTR, #KEY3
FIND :      RRC A
             JNC MATCH
             INC DPTR
             SJMP FIND
MATCH :     CLR A
             MOV CA, @A +DPTR
             MOV P0, A
             MOV P1, #00H
L3 :        MOV A, P2
             ANL A, #0F H
             CJNE A, #0FH, L3
             CALL DELAY
             SJMP L2
```

ASCII-Look up table for each row

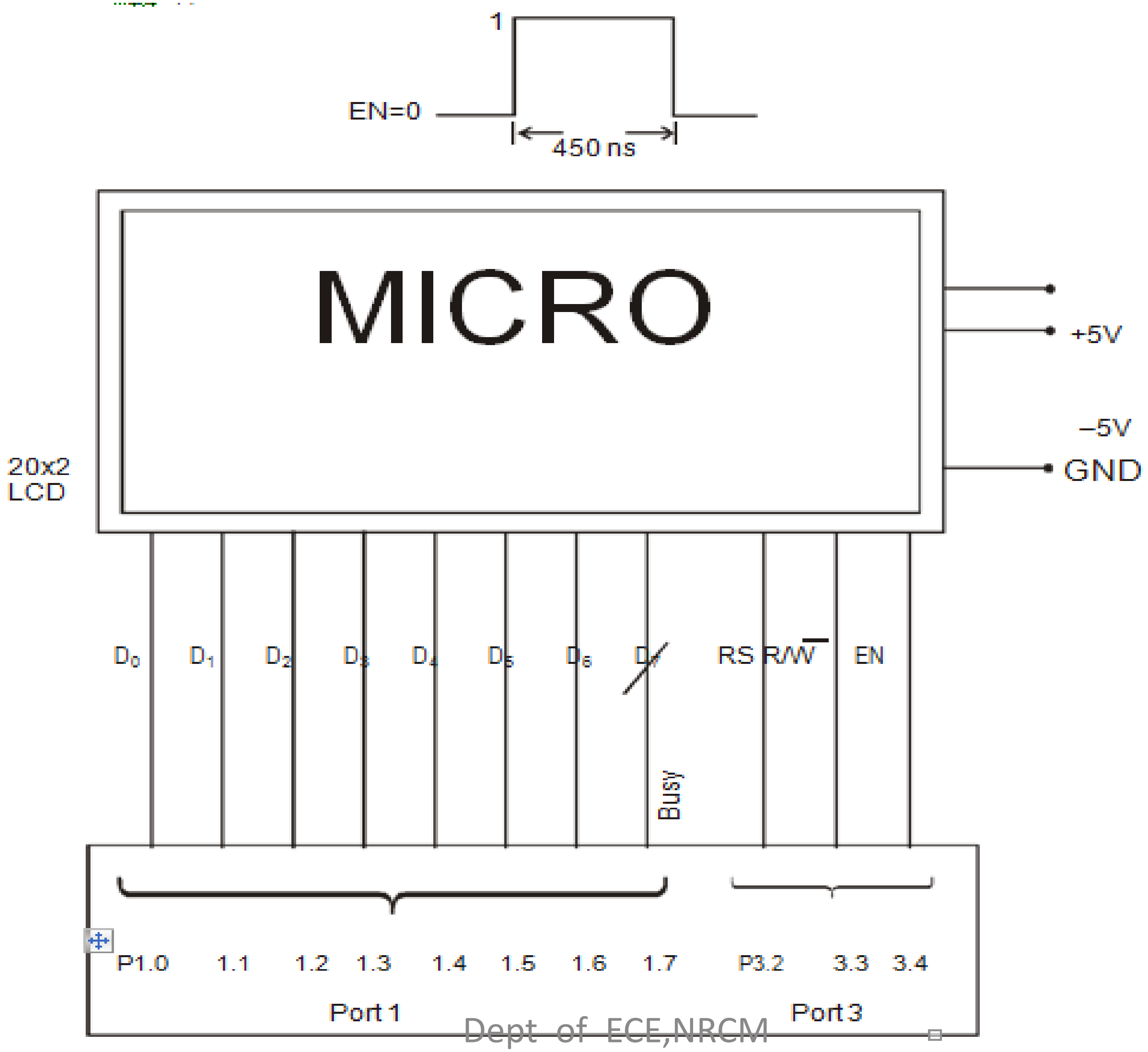
```
        ORG               3000H

      KEY 0 :       DB '0' : '1' : '2' : '3'
      KEY 1 :       DB '4' : '5' : '6' : '7'
      KEY 2 :       DB '8' : '9' : 'A' : 'B'
      KEY3 :        DB 'C' : 'D' : 'E' : 'F'
            END
```

Program for Keyboard

```
                MOV P2, #0FF H
                MOV P1, #00H
      L2:       MOV A, P2
                ANL A, #0F H
                CJNE A, #0F H, OVER
                SJMP L2
      OVER:     ACALL DELAY
                MOV A, P2
                ANL A, #0F H
                CJNE A, #0FH, OVER1
                SJMP L2
      OVER1 :   MOV P1, #0FEH
                MOV A, P2
                ANL A, #0FH
                CJNE A, #0FH, ROW_0
                MOV P1, #0FD H
                MOV A, P2
                ANL A, #0F H
                CJNE A, #0FH, ROW_1
                MOV P1, # 0FB H
                MOV A, P2
                ANL A, #0F H
                CJNE A, #0F H, ROW_2
                MOV P1, # 0F7 H
                MOV A, P2
                ANL A, #0F H
                CJNE A, # 0FH, ROW_3
```

LCD INTERFACING

- The various types of LCD displays are, 16x2, 20x1, 20x2, 20x4, 40x2 and 40x4 LCDs. 16x2 LCD means that it having two lines, 16 characters per line.
- The 8 bit data pins (D_0–D_7) are used to send information tot he LCD or read the contents of the LCD's internal registers.
- The data lines are connected to Port 1. Register Select (RS),
- Read/Write (R/W) and Enable (EN) plans are connected to Port 3.

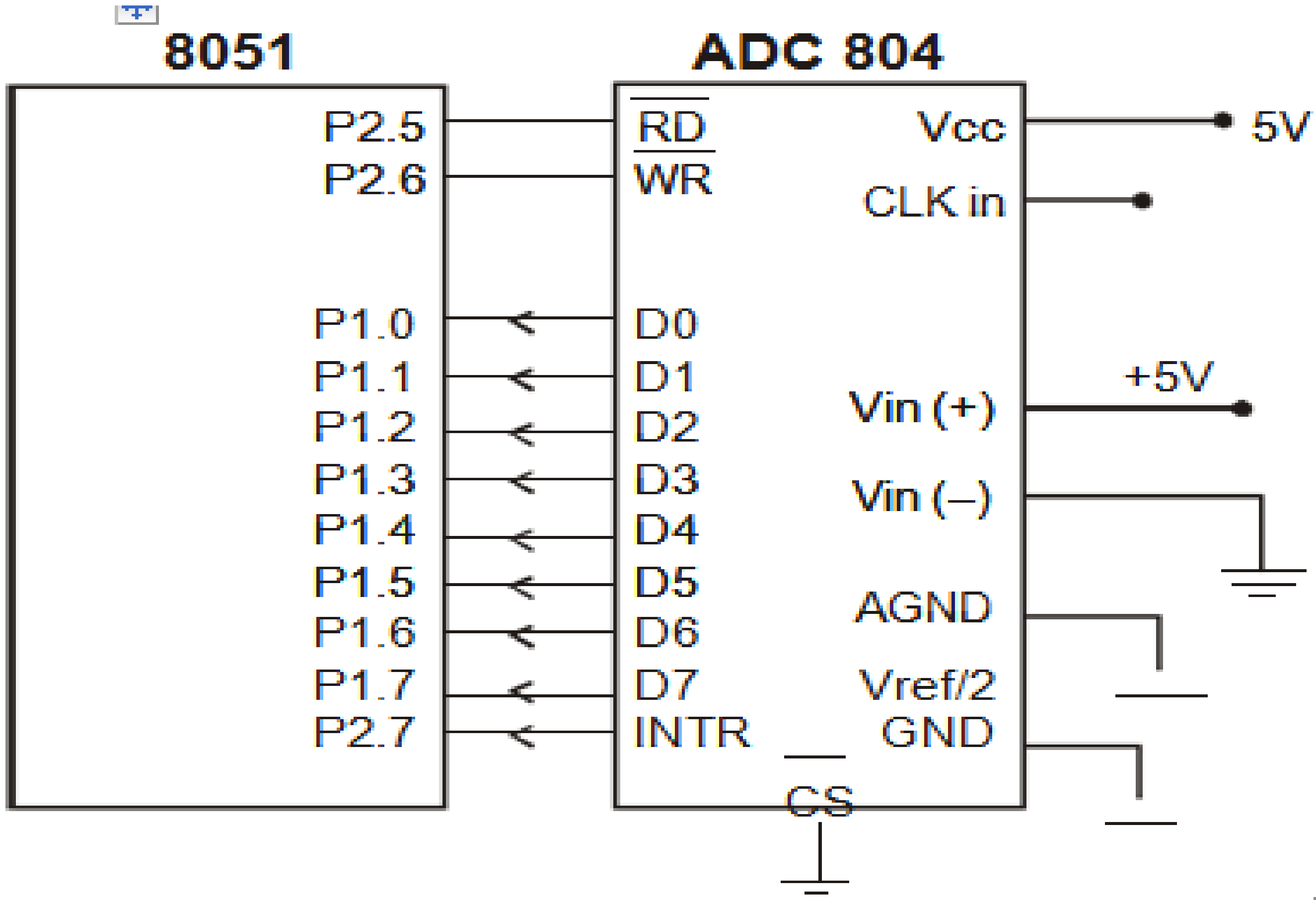

1
EN=0
450 ns
MICRO
+5V
–5V
GND
20x2
LCD
D0
D1
D2
D3
D4
D5
D6
D7
Busy
RS
R/W
EN
P1.0
1.1
1.2
1.3
1.4
1.5
1.6
1.7
P3.2
3.3
3.4
Port 1
Port 3

- There are two important registers are available inside the LCD. They are (i) instruction command register, (ii) data register.
- The RS pin is used to select the register. If RS=0, the instruction command code register is selected, allowing the user to send a command. If RS=1, the data register is selected, allowing the user to send data to be displayed on the LCD.
- R/W pin is used to write information to the LCD or read information from it. EN (enable) pin is used to latch information presented to its data pins.
- When data is supplied to data pins, a high-to-low pulse must be applied to EN pin in order for the LCD to latch in the data present at the data pins.
- This pulse must be a minimum of 450 ns.
- If RS=0 and R/W = 0
  When busy flag (D_7)=1, the LCD is busy and will not accept any new information.
- When busy flag (D_7) = 0, the LCD is ready to receive new information.

ADC interfacing

- ADCs are used to convert the analog signals to digital numbers so that the microcontroller can read them.
- ADC [like ADC 0804 IC] works with +5 volts and has a resolution of 8 bits.
- Conversion time is defined as the time taken to convert the analog input to digital (binary) number. The conversion time varies depending upon the clock signals; it cannot be faster than 110 μs .
- Analog input is given to the pins V_{in} (+) and V_{in} (-).
- V_{in} (-) is connected to ground.
- Digital output pins are D_0 - D_7. D_7 is the MSB and D_0 is the LSB.
- There are two pins for ground, analog ground and digital ground. Analog ground is connected to the ground of the analog V_{in} and digital ground is connected to the ground of the V_{CC} pin.

- The following steps are followed for data conversion :
- Make chip select (CS) = 0 and send a low - to - high pulse to pin WR to start the conversion.
- Keep monitoring the INTR pin. If INTR is low, the conversion is finished and go to the next step. If INTR is high, keep polling until it goes low.
- After the INTR has become low, we make CS = 0 and send a high- to-low pulse to the RD pin to get the data out.

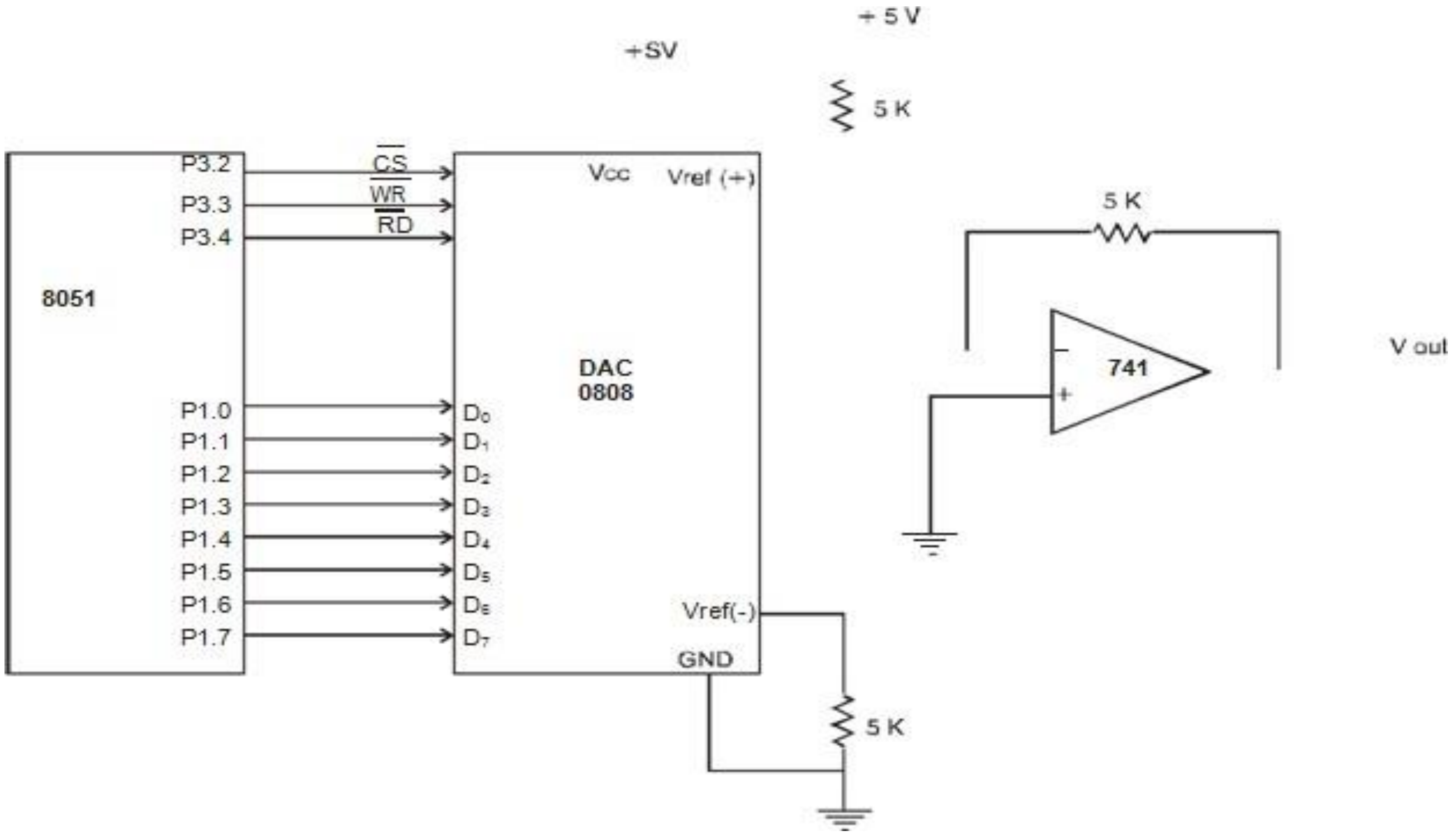

8051
ADC 804
P2.5
P2.6
P1.0
P1.1
P1.2
P1.3
P1.4
P1.5
P1.6
P1.7
P2.7
RD
WR
D0
D1
D2
D3
D4
D5
D6
D7
INTR
CS
Vcc
5V
CLK in
Vin (+)
+5V
Vin (–)
AGND
Vref/2
GND

The program presents the concept to monitor the INTR pins and bring an analog input into register A. Then call a hex - to - ASCII conversion and data display subroutines continuously.

- P2.6 = WR (start conversion needs to low - to - high pulse)
- P2.7 = INTR, when low, end - of - conversion
- P2.5 = RD (a high-to-low will read the data from ADC chip)
- P1.0 - P1.7 = D_0 - D_7 of ADC 804

```
        MOV     P1, # 0FF H       ; make P1 = input
BACK :  CLR     P2.6              ;WR = 0
        SET B   P2.6              ;WR = 1 Low - to - high to start conversion.
HERE :  JB      P2.7, HERE        ; Wait for end of conversion
        CLR     P2.5              ;  Conversion finished, enable RD
        MOV     A, P1             ;  read the data
        A CALL  CONVERSION        ;  hex - to - ASCII conversion
        A CALL  DATA_DISPLAY      ;  display the data
        SET B   P2.5              ;  make RD = 1 for next round
        SJMP    BACK
```

DAC INTERFACING



- The digital - to - analog converter (DAC) is used to convert digital pulses to analog signals.

The methods of creating a DAC are:

- Binary weighted
- R/2R ladder.
- Mostly R/2R method with DAC 0808 (MC 1408) is used since it can achieve a much higher degree of precision. Port 1 furnishes the digital byte to be converted to an analog Voltage and port 3 controls the conversion process.
- In DAC 0808, the digital inputs are converted to current. The total courrent provided by the I_{out} pin is a function of the binary numbers at the $D_0 – D_7$ inputs of DAC and the reference current I_{ref}.

$$I_{out} = I_{ref}\left(\frac{D_7}{2}+\frac{D_6}{4}+\frac{D_5}{8}+\frac{D_4}{16}+\frac{D_3}{32}+\frac{D_2}{64}+\frac{D_1}{128}+\frac{D_0}{256}\right)$$

Where $I_{ref} = 2$ mA.

SENSOR INTERFACING

Sensor :

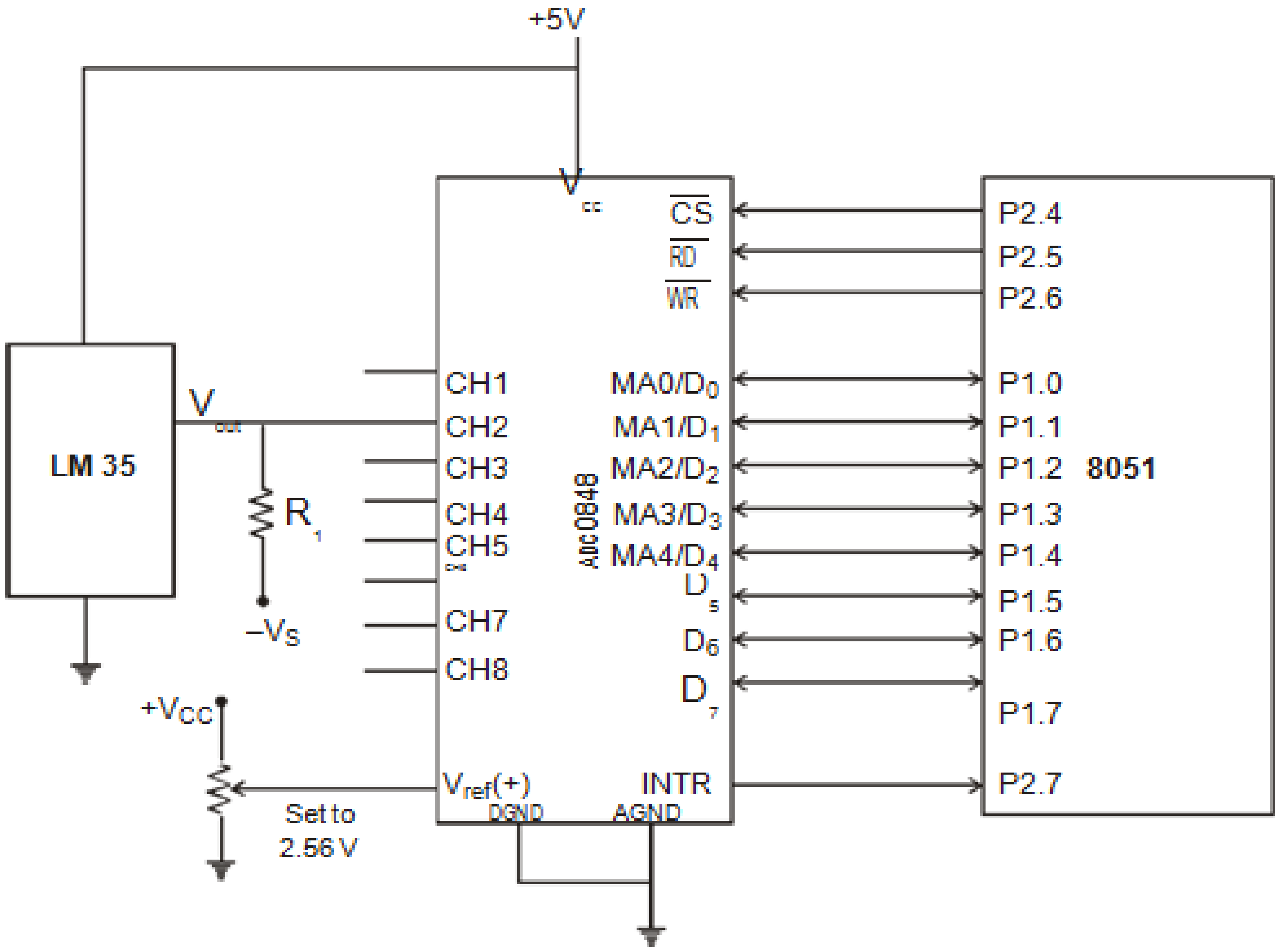
- Sensor converts the physical Pressure, Temperature or other variable to a proportional voltage or current.

Types of Sensors :

- Light Sensor
- Temperature Sensor
- Pressure Sensor
- Force Sensor
- Flow Sensor

Temperature Sensor

- There are many types of temperature sensors. Now we discuss about Semiconductor Temperature Sensor (LM 35). The LM35 series sensors are precision integrated circuit temperature sensor whose output voltage is proportional to the Celsius (centigrade) temperature.
- It outputs 10 mV for each degree of centigrade temperature. If the output is connected to a negative reference voltage V_S, the sensor will give a meaningful output for a temperature range of -55^0C to $+150^0$C. The output voltage can be amplified or filtered for a particular application.



+5V
Vcc
CS
RD
WR
P2.4
P2.5
P2.6
LM 35
Vout
R1
–Vs
CH1
CH2
CH3
CH4
CH5
CH6
CH7
CH8
ADC0848
MA0/D0
MA1/D1
MA2/D2
MA3/D3
MA4/D4
D5
D6
D7
P1.0
P1.1
P1.2
8051
P1.3
P1.4
P1.5
P1.6
P1.7
+Vcc
Vref(+)
INTR
P2.7
DGND
AGND
Set to 2.56 V

EXTERNAL MEMORY INTERFACING

- When the data is located in the code space of 8051, MOVC instruction is used to get the data, where ‘C’ stands for code.
- When the data memory space must be implemented externally, MOVX instruction is used, where ‘X’ stands for external.

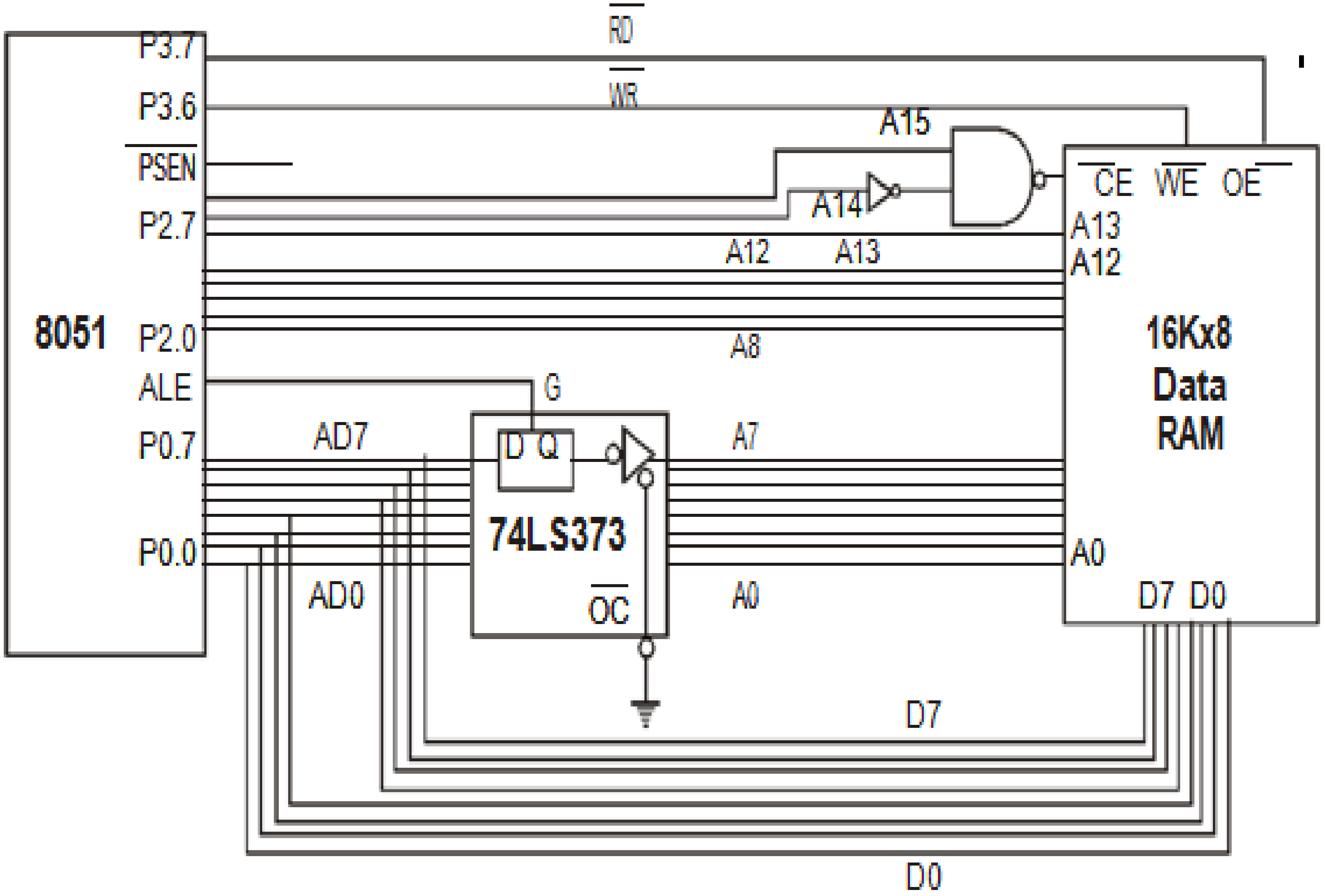
External data RAM interfacing

- To connect the 8051 to an external SRAM, we must use both RD (P3.7) and WR (P3.6).
- In writing data to external data RAM, the instruction “MOVX @DPTR, A” is used, where the contents of register A are written to external RAM whose address is pointed to by the DPTR register.

Program:

Write a program to read 200 bytes of data from Port 1 and save the data in external RAM starting at RAM location 5000H.

```
RAMDATA  EQU    5000H
COUNT    EQU    200
         MOV    DPTR, # RAMDATA ; pointer to external NV-RAM
         MOV    R3, #COUNT      ; counter
AGAIN :  MOV    A, P1           ; read data from P1
         MOVX   @DPTR, A        ; save it external NV-RAM
         ACALL  DELAY           ; wait before next sample
         INC    DPTR            ; next data location
         DJNZ   R3, AGAIN       ; until all are read
HERE :   SJMP   HERE            ; stay here when finished
```


RD
WR
A15
P3.7
P3.6
PSEN
P2.7
A14
A12
A13
CE
WE
OE
A13
A12
8051
P2.0
A8
16Kx8
Data
RAM
ALE
G
P0.7
AD7
D Q
A7
74LS373
P0.0
AD0
OC
A0
A0
D7 D0
D7
D0

STEPPER MOTOR INTERFACING

- A stepper motor is a widely used device that translates electrical pulses into mechanical movement. In applications such as disk drives, dot matrix printers and robotics the stepper motor is used for position control. Every stepper motor has a permanent magnet rotor surrounded by four stator windings, that are paired with a center-tapped common.
- The center tap allows a change of current direction in each of two coils when a winding is grounded, thereby resulting in a polarity change of the stator. The stepper motor shaft runs in a fixed repeatable increment which allows one to move it to a precise position.
- This repeatable fixed movement is possible as a result of basic magnetic theory where poles of the same polarity repel and opposite poles attract. The direction of the rotation is dictated by the stator poles. The stator poles are determined by the current sent through the wire coils.
- As the direction of the current is changed, the polarity is also changed causing the reverse motion of the rotor As the sequence of power is applied to each stator winding, the rotor will rotate. There are several used sequences where each has a different degree of precision.

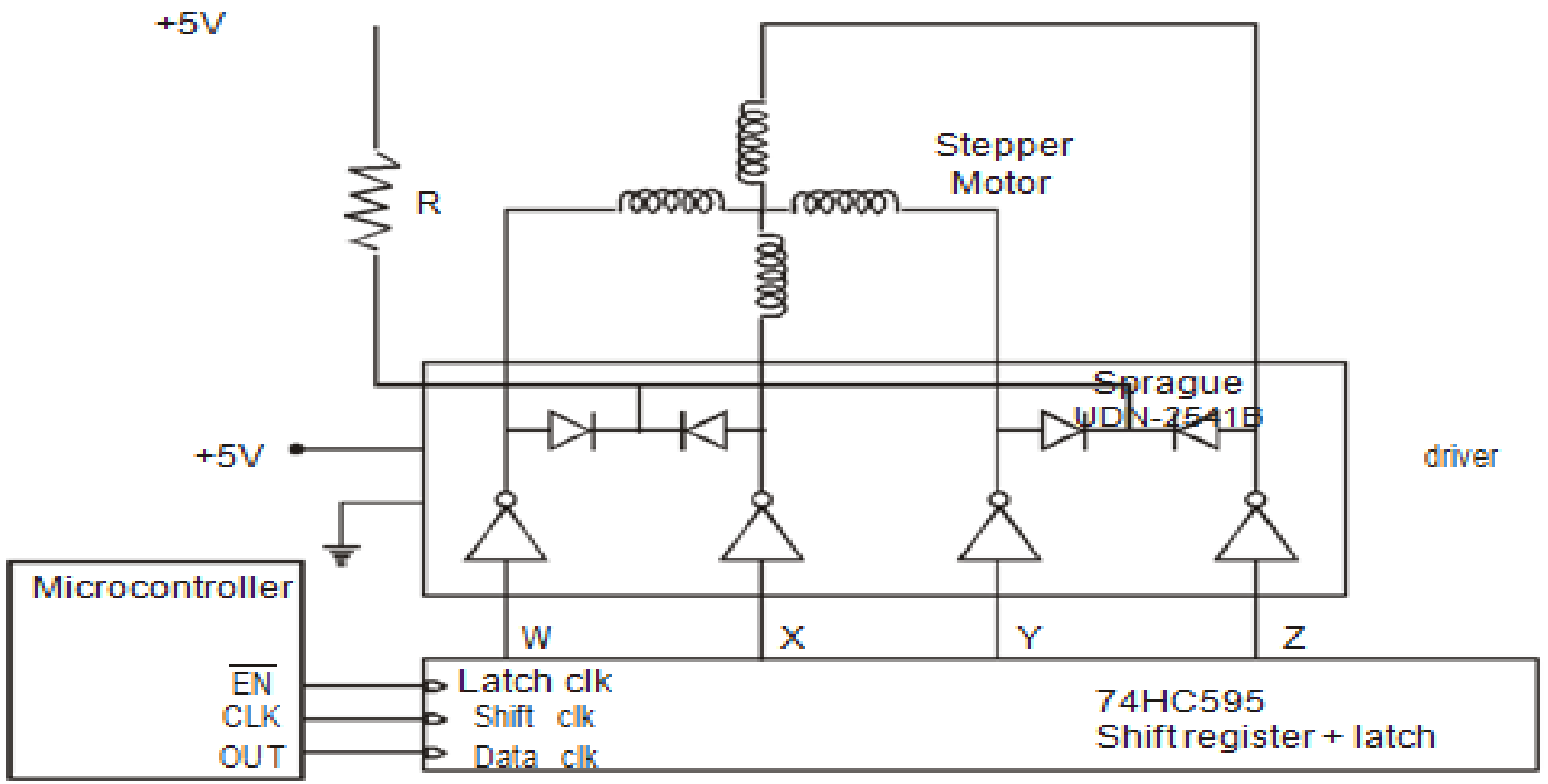


Fig 5.20 Drive circuitry for a stepper motor

The movement of the stepper motor with a single step is depends on the internal construction of the motor, in particular the number of teeth on the stator and the rotor. The step angle is the minimum degree of rotation associated with a single step. Various motors have different step angles. Table 5.3 shows some step angles for various motors.

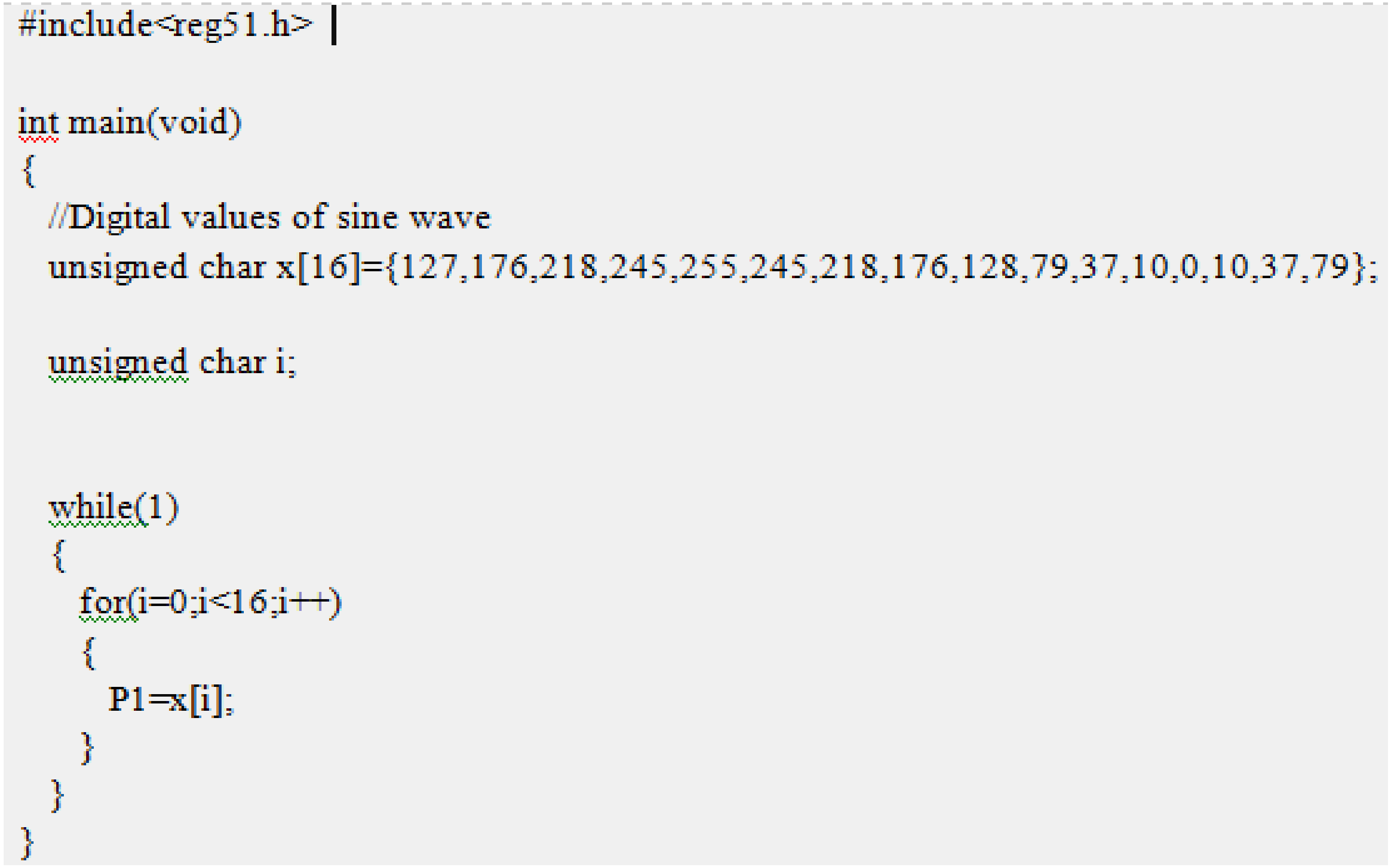
Steps per revolution = Total number of steps needed to rotate one complete rotation or 360 degrees.

$$\text{Steps per second} = \frac{\text{RPM} \times \text{Steps per revolution}}{60}$$

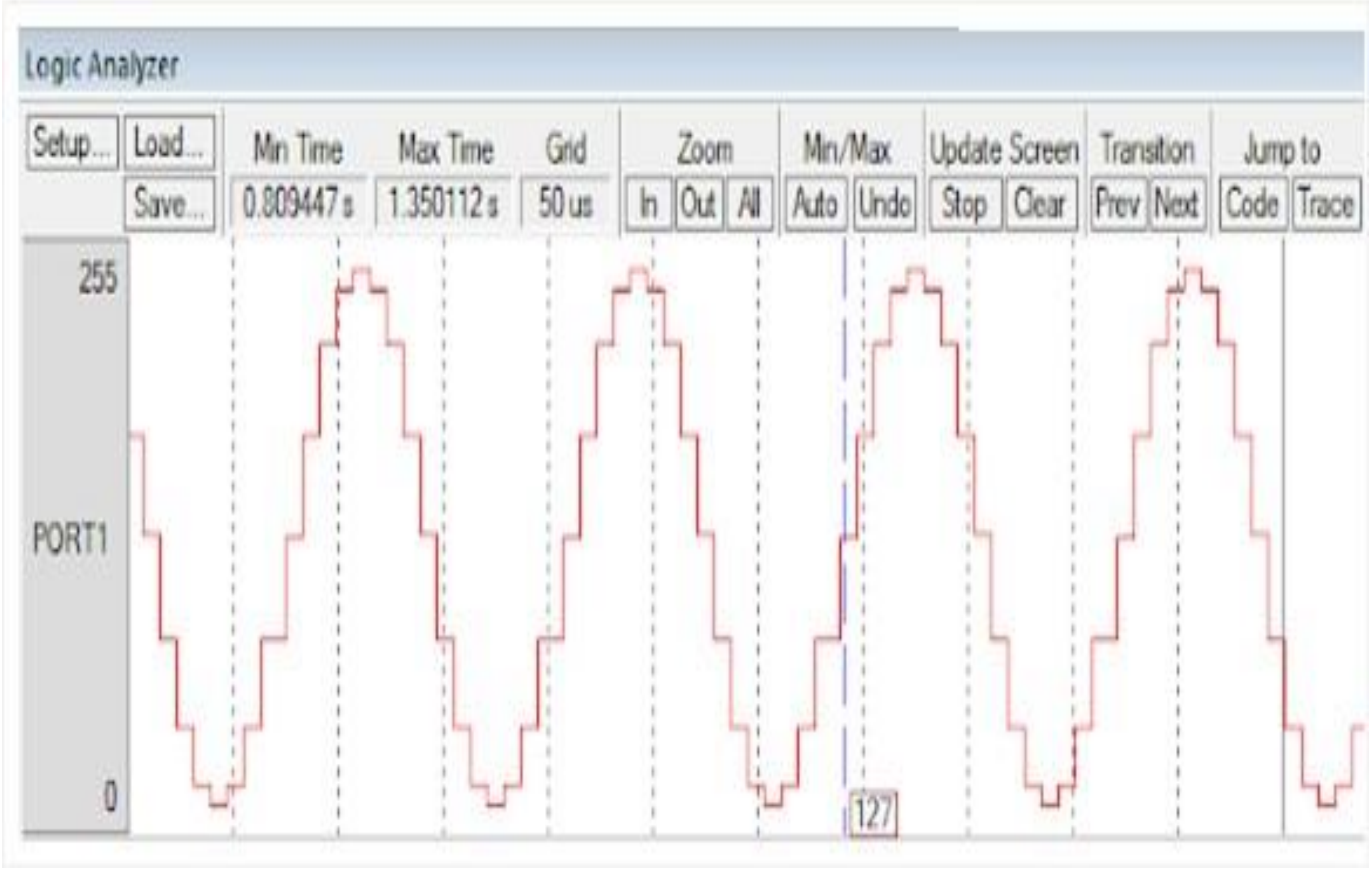
WAVEFORM GENERATION:

- Steps to generate sine wave on 8051 microcontroller.
- Generate digital values of sine wave on a port that is 8 bit binary value.
- Convert that digital value into analog value to take that 8 bit output on 1 pin.
- Generated sine wave is in steps hence to obtain a pure sine wave, pass it through low pass filter. Thus by remove high frequency part, obtain smoother sine wave.
- First, generate digital values for sine wave. For this example take 16 points in 1 cycle. Thus 1 value will hold for 1/16th of 360 degree. Hence use sine(360 * (i/16)) where i runs from 0 to 15.
- This will cover 16 equally spaced points in one cycle. Place this cycle in while (1) loop so that will get continuous sine wave.
- In a cycle of sine wave, half cycle is positive and remaining half cycle is negative. Since microcontroller cannot have negative voltage, will shift sine wave to half of maximum value.
- As maximum value is 255 for 8 bits, half of it is 127.5.Thus digital value to be assigned to port is 127.5 + 127.5 * sine(360*(i/16)) where i runs from 0 to 15. Here minimum value is 127.5 - 127.5 = 0 and maximum value is 127.5 + 127.5 = 255
- Hence sine wave will be between 0 and 255 and which can be assigned to port. Since most of the values will come in fraction, have to round figure to assign integer value.

Program:



```
#include<reg51.h>

int main(void)
{
  //Digital values of sine wave
  unsigned char x[16]={127,176,218,245,255,245,218,176,128,79,37,10,0,10,37,79};

  unsigned char i;



  while(1)
  {
    for(i=0;i<16;i++)
    {
      P1=x[i];
    }
  }
}
```


Logic Analyzer
Setup...
Load...
Save...
Min Time
0.809447 s
Max Time
1.350112 s
Grid
50 us
Zoom
In
Out
All
Min/Max
Auto
Undo
Update Screen
Stop
Clear
Transition
Prev
Next
Jump to
Code
Trace
255
PORT1
0
127

COMPARISON OF MICROPROCESSOR, MICROCONTROLLER, PIC AND ARM PROCESSORS

Microprocessor

- Microprocessor has only a CPU inside them in one or few Integrated Circuits. Like microcontrollers it does not have RAM, ROM and other peripherals. They are dependent on external circuits of peripherals to work. But microprocessors are not made for specific task but they are required where tasks are complex and tricky like development of software’s, games and other applications that require high memory and where input and output are not defined. It may be called heart of a computer system. Some examples of microprocessor are Pentium, I3, and I5 etc.

Microcontroller

- A micro-controller can be comparable to a little stand alone computer; it is an extremely powerful device, which is able of executing a series of pre-programmed tasks and interacting with extra hardware devices. Being packed in a tiny integrated circuit (IC) whose size and weight is regularly negligible, it is becoming the perfect controller for as robots or any machines required some type of intelligent automation.
- A single microcontroller can be enough to manage a small mobile robot, an automatic washer machine or a security system. Several microcontrollers contains a memory to store the program to be executed, and a lot of input/output lines that can be a used to act jointly with other devices, like reading the state of a sensor or controlling a motor.8051 microcontroller is an 8-bit family of microcontroller is developed by the Intel in the year 1981.

PIC Microcontroller

- Peripheral Interface Controller (PIC) is microcontroller developed by a Microchip, PIC microcontroller is fast and simple to implement program when we contrast other microcontrollers like 8051. The ease of programming and simple to interfacing with other peripherals PIC become successful microcontroller. Microcontroller is an integrated chip which is consists of RAM, ROM, CPU, TIMER and COUNTERS.
- The PIC is a microcontroller which as well consists of RAM, ROM, CPU, timer, counter, ADC (analog to digital converters), DAC (digital to analog converter). PIC Microcontroller also support the protocols like CAN, SPI, UART for an interfacing with additional peripherals. PIC mostly used to modify Harvard architecture and also supports RISC (Reduced Instruction Set Computer) by the above requirement RISC and Harvard we can simply that PIC is faster than the 8051 based controllers which is prepared up of Von-Newman architecture.

ARM Processor

- An ARM processor is also one of a family of CPUs based on the RISC (Reduced Instruction Set Computer) architecture developed by Advanced RISC Machines (ARM). An ARM makes at 32-bit and 64-bit RISC multi-core processors. RISC processors are designed to perform a smaller number of types of computer instructions so that they can operate at a higher speed, performing extra millions of instructions per second (MIPS).
- By stripping out unnecessary instructions and optimizing pathways, RISC processors give outstanding performance at a part of the power demand of CISC (complex instruction set computing) procedure. ARM processors are widely used in customer electronic devices such as smart phones, tablets, multimedia players and other mobile devices, such as wearables. Because of their reduced to instruction set, they need fewer transistors, which enable a smaller die size of the integrated circuitry(IC).

ARM PROCESSORS
UNIT 4

Contents

Introducing ARM

- Modes of operation

•ARM processor has 7 modes of operation.

•Switching between modes can be done manually through modifying the mode bits in the CPSR register.

•Most application programs execute in user mode

•Non user modes (called privileged modes) are entered to serve interrupts or exceptions

•The system mode is special mode for accessing protected resources. It don‘t use registers used by exception hanlders, so it can‘t be corrupted by any exception handler error!!!

Introducing ARM

- Modes of operation

Switching between these modes requires saving/loading register values

Processor Mode	**Description**
User (***usr***)	Normal program execution mode
FIQ (***fiq***)	Fast data processing mode
IRQ (***irq***)	For general purpose interrupts
Supervisor (***svc***)	A protected mode for the operating system
Abort (***abt***)	When data or instruction fetch is aborted
Undefined (***und***)	For undefined instructions
System (***sys***)	Privileged mode for OS Tasks

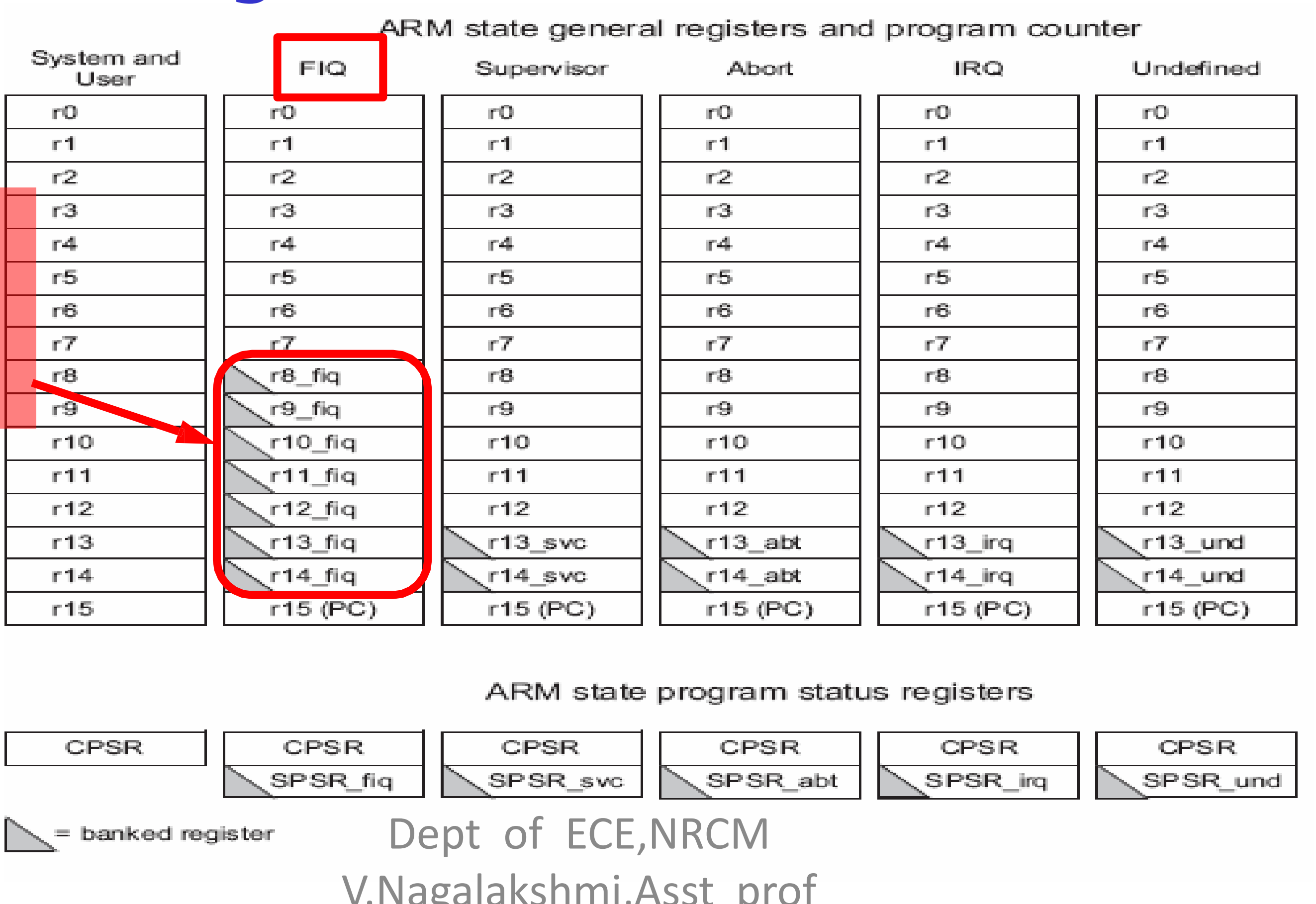
Introducing ARM

- ARM register set
  - •ARM processor has 37 32-bit registers.
  - •31 registers are general purpose registers.
  - •6 registers are control registers
  - •Registers are named from R0 to R16 with some registers banked in different modes
  - •R13 is the stack pointer ***SP*** (Banked)
  - •R14 is subroutine link register ***LR*** (Banked)
  - •R15 is progrm counter ***PC***
  - •R16 is current program status register ***CPSR*** (Banked)

Introducing ARM

- ARM register set

More banked registers, so context switching is faster



ARM state general registers and program counter

System and User	FIQ	Supervisor	Abort	IRQ	Undefined
r0	r0	r0	r0	r0	r0
r1	r1	r1	r1	r1	r1
r2	r2	r2	r2	r2	r2
r3	r3	r3	r3	r3	r3
r4	r4	r4	r4	r4	r4
r5	r5	r5	r5	r5	r5
r6	r6	r6	r6	r6	r6
r7	r7	r7	r7	r7	r7
r8	▨ r8_fiq	r8	r8	r8	r8
r9	▨ r9_fiq	r9	r9	r9	r9
r10	▨ r10_fiq	r10	r10	r10	r10
r11	▨ r11_fiq	r11	r11	r11	r11
r12	▨ r12_fiq	r12	r12	r12	r12
r13	▨ r13_fiq	▨ r13_svc	▨ r13_abt	▨ r13_irq	▨ r13_und
r14	▨ r14_fiq	▨ r14_svc	▨ r14_abt	▨ r14_irq	▨ r14_und
r15	r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)	r15 (PC)

ARM state program status registers

System and User	FIQ	Supervisor	Abort	IRQ	Undefined
CPSR	CPSR	CPSR	CPSR	CPSR	CPSR
	▨ SPSR_fiq	▨ SPSR_svc	▨ SPSR_abt	▨ SPSR_irq	▨ SPSR_und

▨ = banked register

Contents

Exceptions

- What is an exception?

  An exception is any condition that needs to halt normal execution of the instructions

- Examples

  •Resetting ARM core
  •Failure of fetching instructions
  •HWI
  •SWI

Exceptions

- Exceptions and modes

Each exception causes the ARM core to enter a specific mode.

Exception	**Mode**	**Purpose**
Fast Interrupt Request	FIQ	Fast interrupt handling
Interrupt Request	IRQ	Normal interrupt handling
SWI and RESET	SVC	Protected mode for OS
Pre-fetch or data abort	ABT	Memory protection handling
Undefined Instruction	UND	SW emulation of HW coprocessors

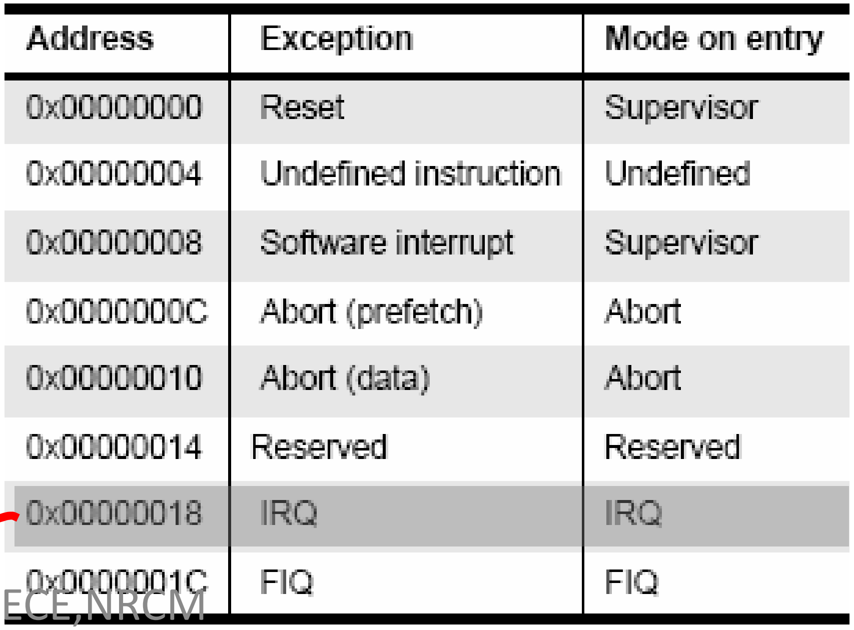
Exceptions

- Vector table

It is a table of addresses that the ARM core branches to when an exception is raised and there is always branching instructions that direct the core to the ISR.

At this place in memory, we find a branching instruction

ldr pc, [pc, #_IRQ_handler_offset]



Address	Exception	Mode on entry
0x00000000	Reset	Supervisor
0x00000004	Undefined instruction	Undefined
0x00000008	Software interrupt	Supervisor
0x0000000C	Abort (prefetch)	Abort
0x00000010	Abort (data)	Abort
0x00000014	Reserved	Reserved
0x00000018	IRQ	IRQ
0x0000001C	FIQ	FIQ

Exceptions

- Exception priorities

decide which of the currently raised exceptions is more important

Decide if the exception handler itself can be interrupted during execution or not?

Both are caused by an instruction entering the execution stage of the ARM instruction pipeline

Exception	Priority	I bit	F bit
Reset	1	1	1
Data Abort	2	1	-
FIQ	3	1	1
IRQ	4	1	-
Prefetch abort	5	1	-
SWI	6	1	-
Undefined instruction	6	1	-

Exceptions

- Link Register Offset

This register is used to return the ***PC*** to the appropriate place in the interrupted task since this is not always the old ***PC*** value.It is modified depending on the type of exception.

The ***PC*** has advanced beyond the instruction causing the exception. Upon exit of the prefetch abort exception handler, software must re-load the PC back one instruction from the ***PC*** saved at the time of the exception.

Exception	Returning Address
Reset	None
Data Abort	LR-8
FIQ, IRQ, prefetch Abort	LR-4
SWI, Undefined Instruction	LR

Exceptions

- Entering exception handler
  1. Save the address of the next instruction in the appropriate Link Register ***LR***.
  2. Copy ***CPSR*** to the ***SPSR*** of new mode.
  3. Change the mode by modifying bits in ***CPSR***.
  4. Fetch next instruction from the vector table.
- Leaving exception handler
  1. Move the Link Register ***LR*** (minus an offset) to the ***PC***.
  2. Copy ***SPSR*** back to ***CPSR,*** this will automatically changes the mode back to the previous one.
  3. Clear the interrupt disable flags (if they were set).

Contents

- Introducing ARM
- Exceptions
- Interrupts
- Interrupt handling schemes
- Summary

Interrupts

- Assigning interrupts

It is up to the system designer who can decide which HW peripheral can produce which interrupt.

But system designers have adopted a standard design for assigning interrupts:

•SWI are used to call privileged OS routines.
•IRQ are assigned to general purpose interrupts like periodic timers.
•FIQ is reserved for one single interrupt source that requires fast response time.

Interrupts

- Interrupt latency

It is the interval of time from an external interrupt signal being raised to the first fetch of an instruction of the ISR of the raised interrupt signal.

System architects try to achieve two main goals:

•To handle multiple interrupts simultaneously.
•To minimize the interrupt latency.

And this can be done by 2 methods:

•allow nested interrupt handling
•give priorities to different interrupt sources

Interrupts

- Enabling and disabling Interrupt

This is done by modifying the ***CPSR***, this is done using only 3 ARM instruction:

MRS	To read *CPSR*
MSR	To store in *CPSR*
BIC	Bit clear instruction
ORR	OR instruction

Enabling an IRQ/FIQ Interrupt:

```
MRS     r1, cpsr
BIC     r1, r1, #0x80/0x40
MSR     cpsr_c, r1
```

Disabling an IRQ/FIQ Interrupt:

```
MRS     r1, cpsr
ORR     r1, r1, #0x80/0x40
MSR     cpsr_c, r1
```

Interrupts

- Interrupt stack

Stacks are needed extensively for context switching between different modes when interrupts are raised.

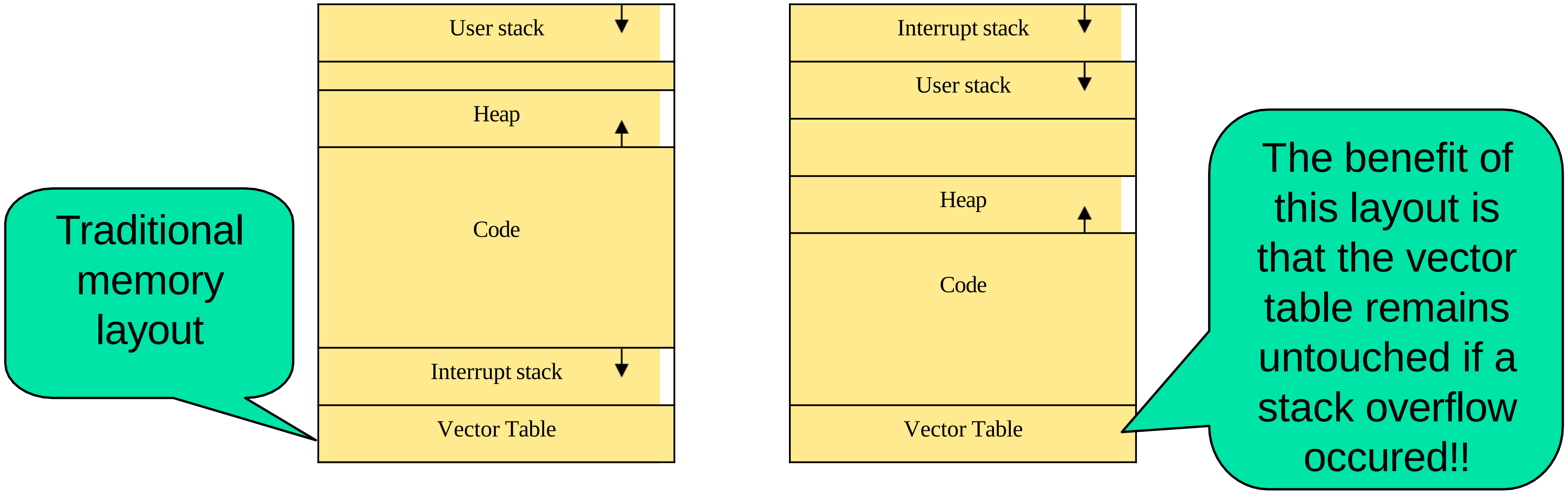
The design of the exception stack depends on two factors:
•OS Requirements.
•Target hardware.

A good stack design tries to avoid stack overflow because it cause instability in embedded systems.

Interrupts

■ Interrupt stack

Two design decisions need to be made for the stacks:
•The location
•The size


User stack
Heap
Code
Interrupt stack
Vector Table
Interrupt stack
User stack
Heap
Code
Vector Table
Traditional memory layout
The benefit of this layout is that the vector table remains untouched if a stack overflow occured!!

Contents

Interrupt handling schemes

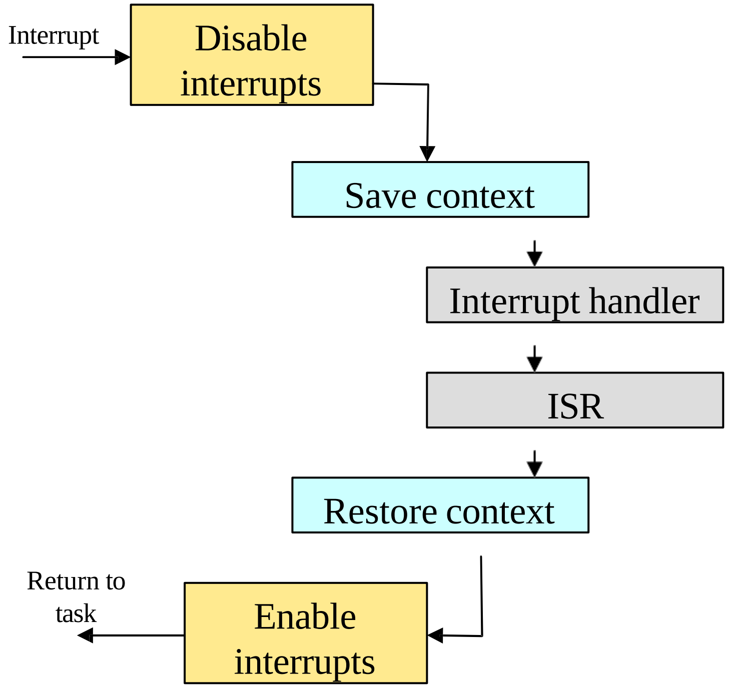
▪ Non-nested interrupt handling scheme

•This is the simplest interrupt handler.

•Interrupts are disabled until control is returned back to the interrupted task.

•One interrupt can be served at a time.

•Not suitable for complex embedded systems.



Interrupt handling schemes

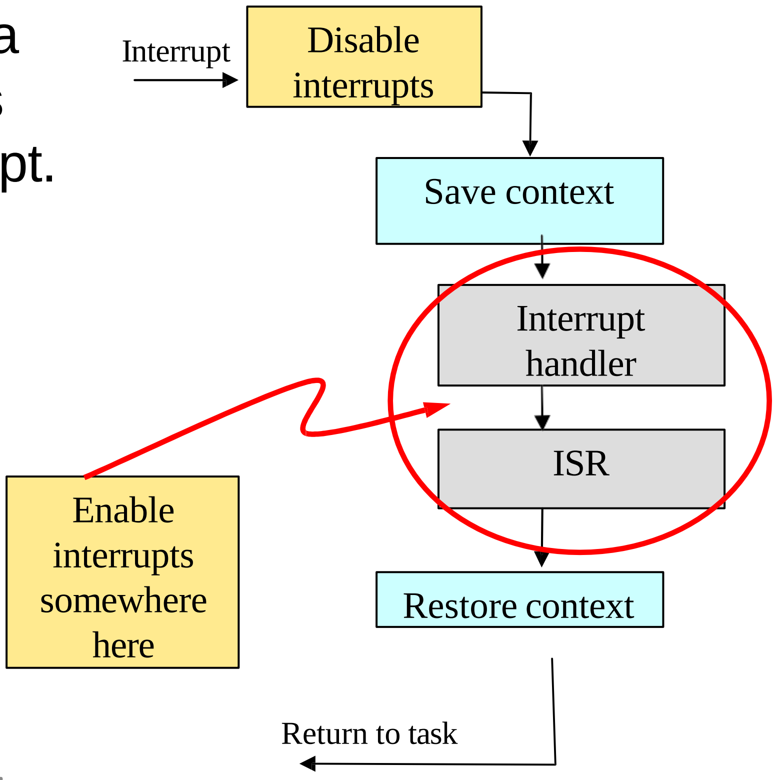
Nested interrupt handling scheme(1)

•Handling more than one interrupt at a time is possible by enabling interrupts before fully serving the current interrupt.

•Latency is improved.

•System is more complex.

•No difference between interrupts by priorities, so normal interrupts can block critical interrupts.



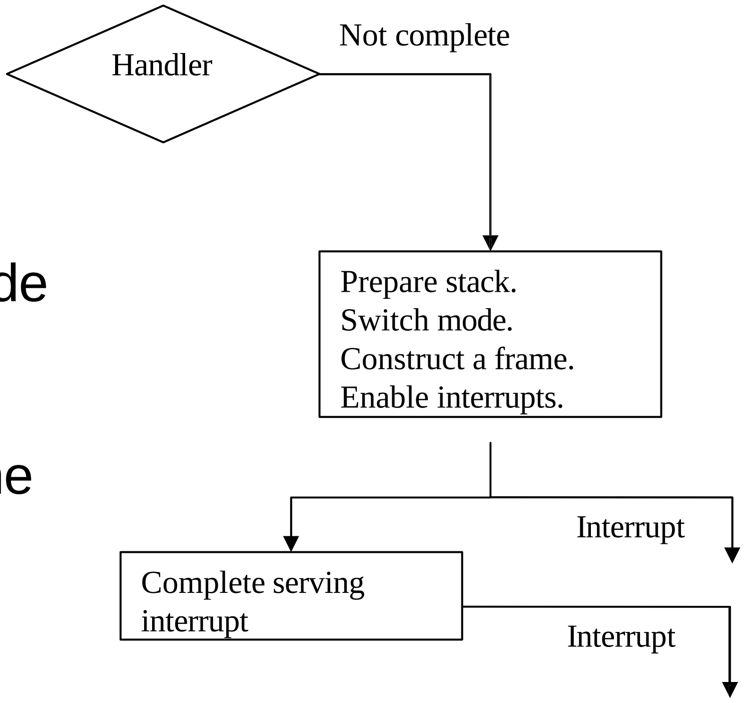
Interrupt handling schemes

- Nested interrupt handling scheme(2)

•The handler tests a flag that is updated by the ISR

•Re enabling interrupts requires switching out of current interrupt mode to either SVC or system mode.

•Context switch involves emptying the IRQ stack into reserved blocks of memory on SVC stack called stack frames.



Interrupt handling schemes

- Prioritized simple interrupt handling

- associate a priority level with a particular interrupt source.
- Handling prioritization can be done by means of software or hardware.
- When an interrupt signal is raised, a fixed amount of comparisons is done.
    - So the interrupt latency is deterministic.
    - But this could be considered a disadvantage!!

Interrupt handling schemes

- Other schemes

There are some other schemes, which are actually modifications to the previous schemes as follows:

•"Re-entrant interrupt handler": re-enable interrupts earlier and support priorities, so the latency is reduced.

•"Prioritized standard interrupt handler": arranges priorities in a special way to reduce the time needed to decide on which interrupt will be handled.

•"Prioritized grouped interrupt handler": groups some interrupts into subset which has a priority level, this is good for large amount of interrupt sources.

Contents

- Introducing ARM
- Exceptions
- Interrupts
- Interrupt handling schemes
- Summary