

Digital Computing Platforms (20A02602T)

III B. Tech II Semester (R20)

**Prepared by
Mrs. K.J.POORNIMA, M.Tech
Assistant Professor**

8086

Microprocessor

Table 4.19**History of Intel microprocessors over three decades**

Processor	Year	Feature Size (μm)	Transistors	Frequency (MHz)	Word size	Package
4004	1971	10	2.3k	0.75	4	16-pin DIP
8008	1972	10	3.5k	0.5–0.8	8	18-pin DIP
8080	1974	6	6k	2	8	40-pin DIP
8086	1978	3	29k	5–10	16	40-pin DIP
80286	1982	1.5	134k	6–12	16	68-pin PGA
Intel386	1985	1.5–1.0	275k	16–25	32	100-pin PGA
Intel486	1989	1–0.6	1.2M	25–100	32	168-pin PGA
Pentium	1993	0.8–0.35	3.2–4.5M	60–300	32	296-pin PGA
Pentium Pro	1995	0.6–0.35	5.5M	166–200	32	387-pin MCM PGA
Pentium II	1997	0.35–0.25	7.5M	233–450	32	242-pin SECC
Pentium III	1999	0.25–0.18	9.5–28M	450–1000	32	330-pin SECC2
Pentium 4	2001	0.18–0.13	42–55M	1400–3200	32	478-pin PGA

Microprocessor

Third Generation

During 1978

HMOS technology \Rightarrow Faster speed, Higher packing density

16 bit processors \Rightarrow 40/ 48/ 64 pins
Easier to program

Dynamically relatable programs

Processor has multiply/ divide arithmetic hardware

More powerful interrupt handling capabilities

Flexible I/O port addressing

Intel 8086 (16 bit processor)

First Generation

Between 1971 - 1973

PMOS technology, non compatible with TTL

4 bit processors \Rightarrow 16 pins

8 and 16 bit processors \Rightarrow 40 pins

Due to limitations of pins, signals are multiplexed

Fifth Generation **Pentium**

Fourth Generation

During 1980s

Low power version of HMOS technology (HCMOS)

32 bit processors

Physical memory space 2^{24} bytes = 16 Mb

Virtual memory space 2^{40} bytes = 1 Tb

Floating point hardware

Supports increased number of addressing modes

Intel 80386

Second Generation

During 1973

NMOS technology \Rightarrow Faster speed, Higher density, Compatible with TTL

4 / 8/ 16 bit processors \Rightarrow 40 pins

Ability to address large memory spaces and I/O ports

Greater number of levels of subroutine nesting

Better interrupt handling capabilities

Intel 8085 (8 bit processor)

First 16-bit processor released by INTEL in the year 1978

Originally HMOS, now manufactured using HMOS III technique

Approximately 29,000 transistors, 40 pin DIP, 5V supply

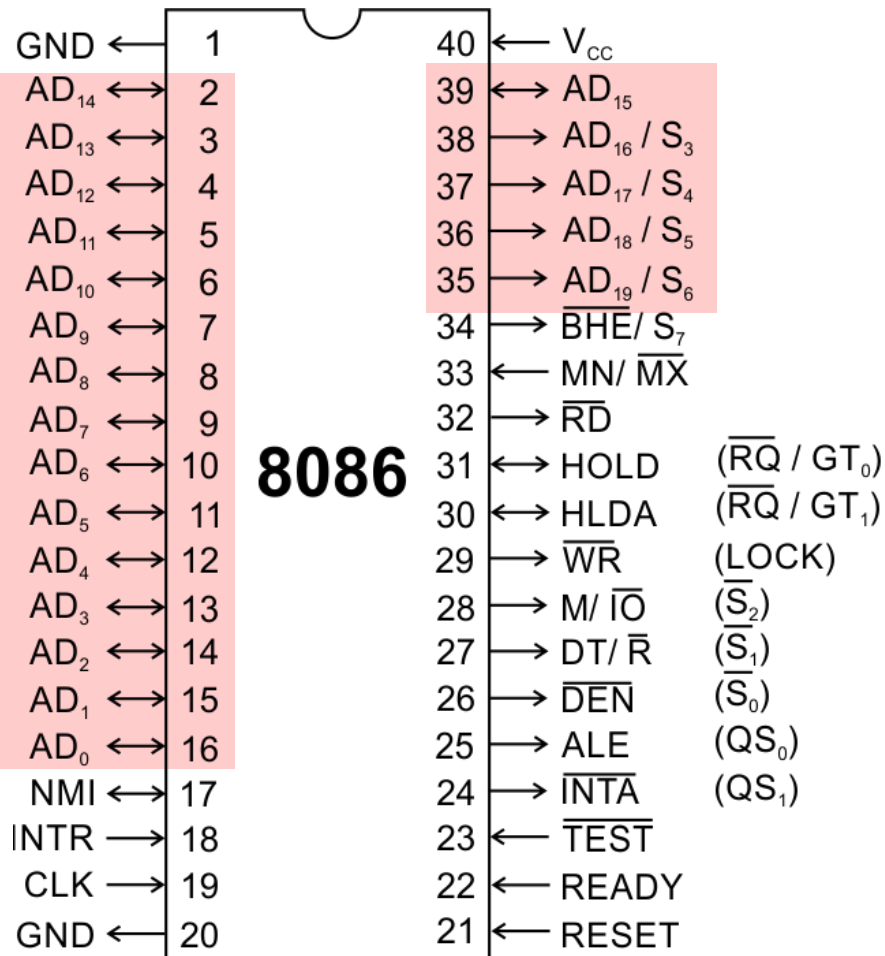
Does not have internal clock; external asymmetric clock source with 33% duty cycle

20-bit address to access memory \Rightarrow can address up to $2^{20} = 1$ megabytes of memory space.

Addressable memory space is organized into two banks of 512 kb each; **Even (or lower) bank and **Odd (or higher) bank**. Address line A_0 is used to select even bank and control signal \overline{BHE} is used to access odd bank**

Uses a separate 16-bit address for I/O mapped devices \Rightarrow can generate $2^{16} = 64$ k addresses.

Operates in two modes: **minimum mode and **maximum mode**, decided by the signal at \overline{MN} and \overline{MX} pins.**



AD₀-AD₁₅ (Bidirectional)

Address/Data bus

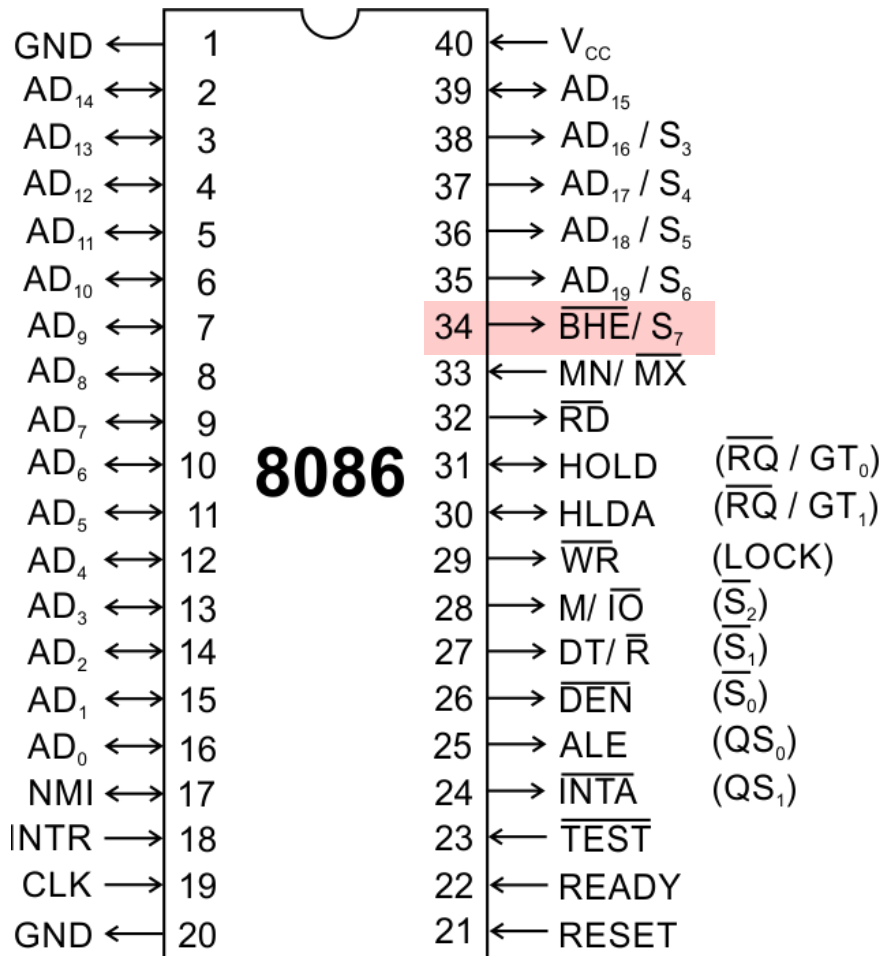
Low order address bus; these are multiplexed with data.

When AD lines are used to transmit memory address the symbol A is used instead of AD, for example A₀-A₁₅.

When data are transmitted over AD lines the symbol D is used in place of AD, for example D₀-D₇, D₈-D₁₅ or D₀-D₁₅.

A₁₆/S₃, A₁₇/S₄, A₁₈/S₅, A₁₉/S₆

High order address bus. These are multiplexed with status signals



BHE (Active Low)/S₇ (Output)

Bus High Enable/Status

It is used to enable data onto the most significant half of data bus, D₈-D₁₅. 8-bit device connected to upper half of the data bus use BHE (Active Low) signal. It is multiplexed with status signal S₇.

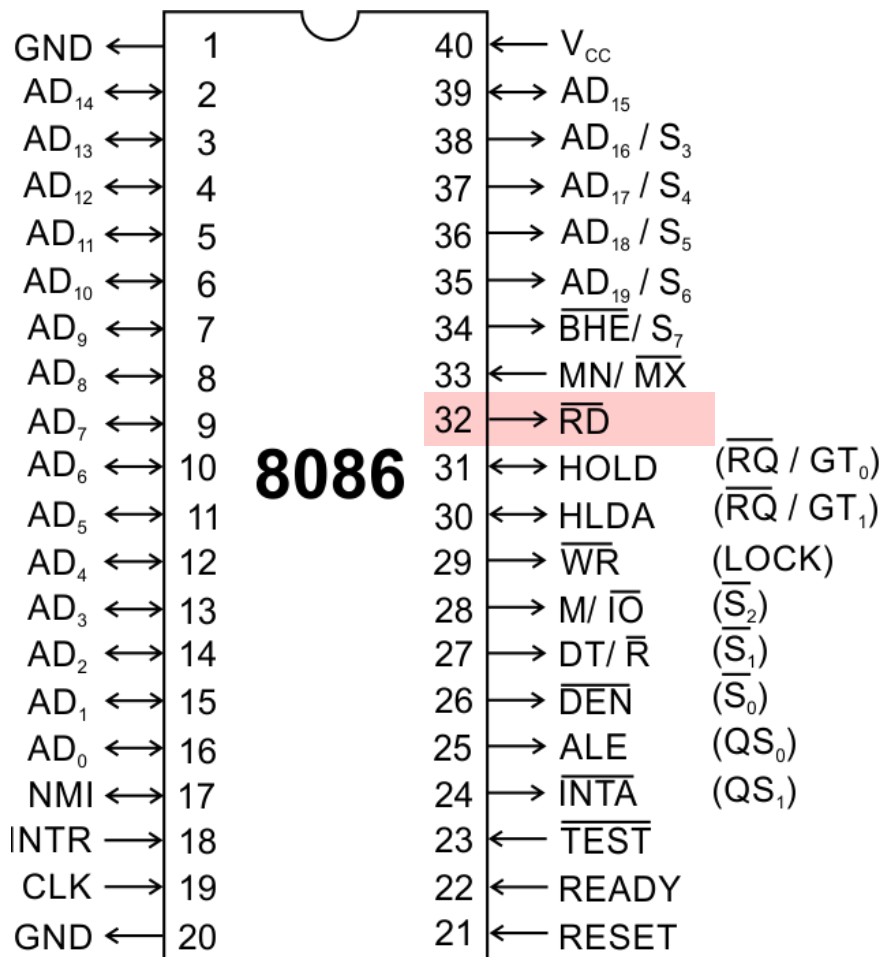
MN / MX

MINIMUM / MAXIMUM

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

RD (Read) (Active Low)

The signal is used for read operation.
It is an output signal.
It is active when low.



TEST

$\overline{\text{TEST}}$ input is tested by the 'WAIT' instruction.

8086 will enter a wait state after execution of the WAIT instruction and will resume execution only when the $\overline{\text{TEST}}$ is made low by an active hardware.

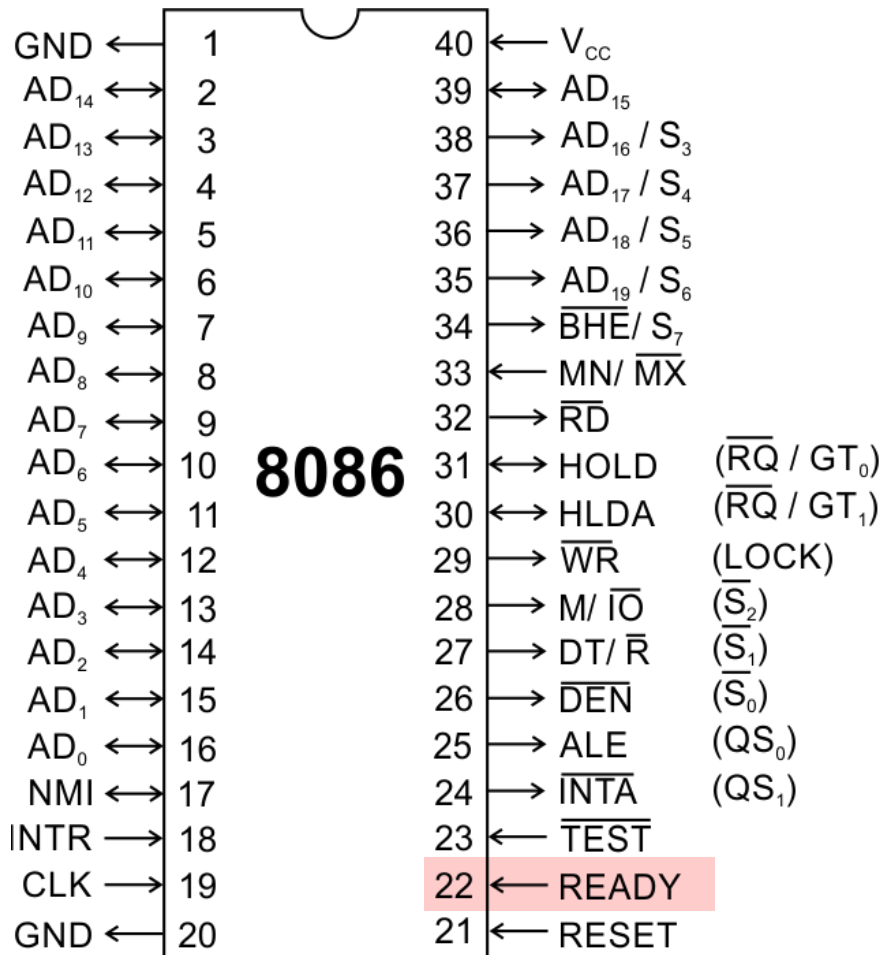
This is used to synchronize an external activity to the processor internal operation.

READY

This is the acknowledgement from the slow device or memory that they have completed the data transfer.

The signal made available by the devices is synchronized by the 8284A clock generator to provide ready input to the **8086**.

The signal is active high.



RESET (Input)

Causes the processor to immediately terminate its present activity.

The signal must be active HIGH for at least four clock cycles.

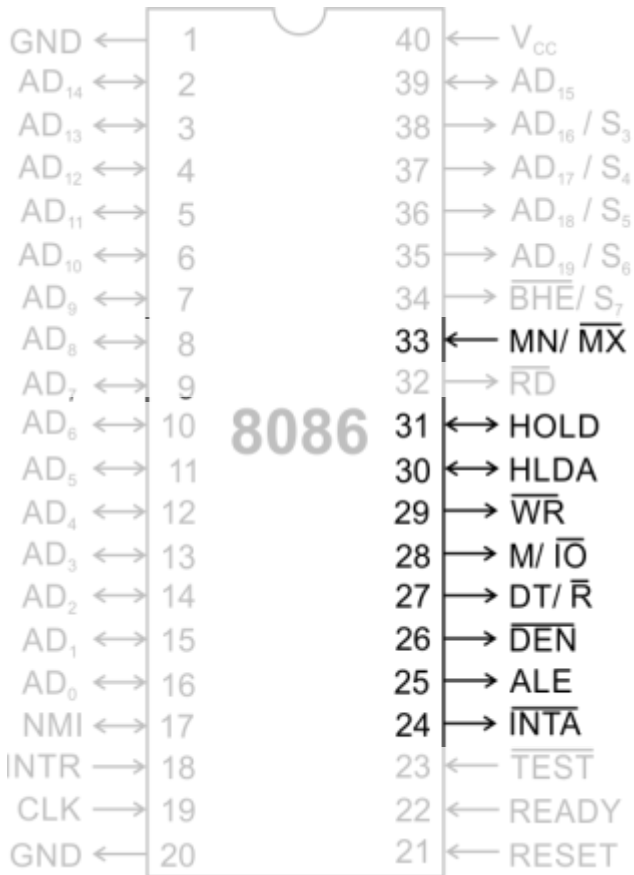
CLK

The clock input provides the basic timing for processor operation and bus control activity. Its an asymmetric square wave with 33% duty cycle.

INTR Interrupt Request

This is a triggered input. This is sampled during the last clock cycles of each instruction to determine the availability of the request. If any interrupt request is pending, the processor enters the interrupt acknowledge cycle.

This signal is active high and internally synchronized.



The 8086 microprocessor can work in two modes of operations : **Minimum mode** and **Maximum mode**.

In the minimum mode of operation the microprocessor do not associate with any co-processors and can not be used for multiprocessor systems.

In the maximum mode the 8086 can work in multi-processor or co-processor configuration.

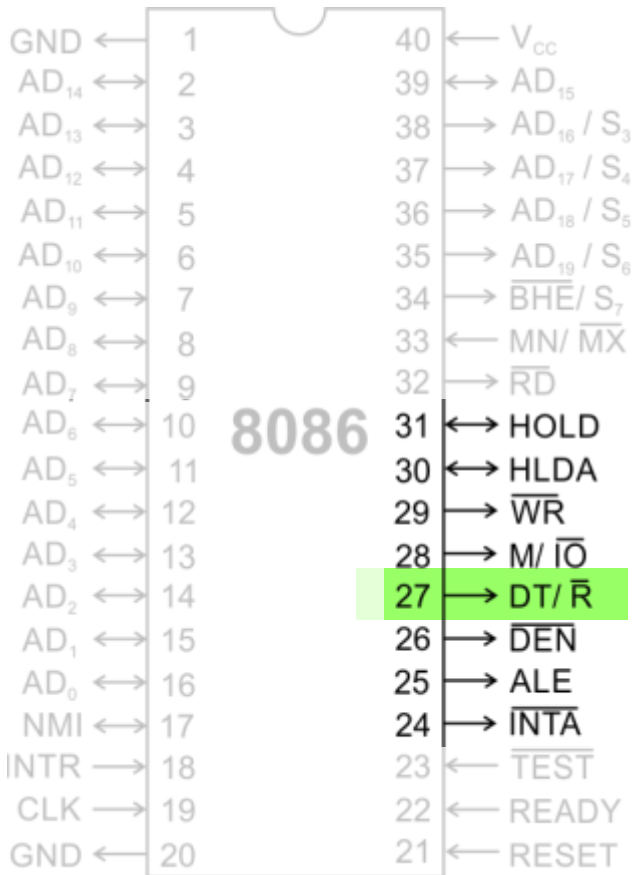
Minimum or maximum mode operations are decided by the pin MN/ MX(Active low).

When this pin is high 8086 operates in minimum mode otherwise it operates in **Maximum mode**.

Pins 24 -31

For minimum mode operation, the $\overline{MN}/\overline{MX}$ is tied to VCC (logic high)

8086 itself generates all the bus control signals



DT/ \overline{R} (**Data Transmit/ Receive**) Output signal from the processor to control the direction of data flow through the data transceivers

\overline{DEN} (**Data Enable**) Output signal from the processor used as out put enable for the transceivers

ALE (**Address Latch Enable**) Used to demultiplex the address and data lines using external latches

$\overline{M}/\overline{IO}$ Used to differentiate memory access and I/O access. For memory reference instructions, it is **high**. For IN and OUT instructions, it is **low**.

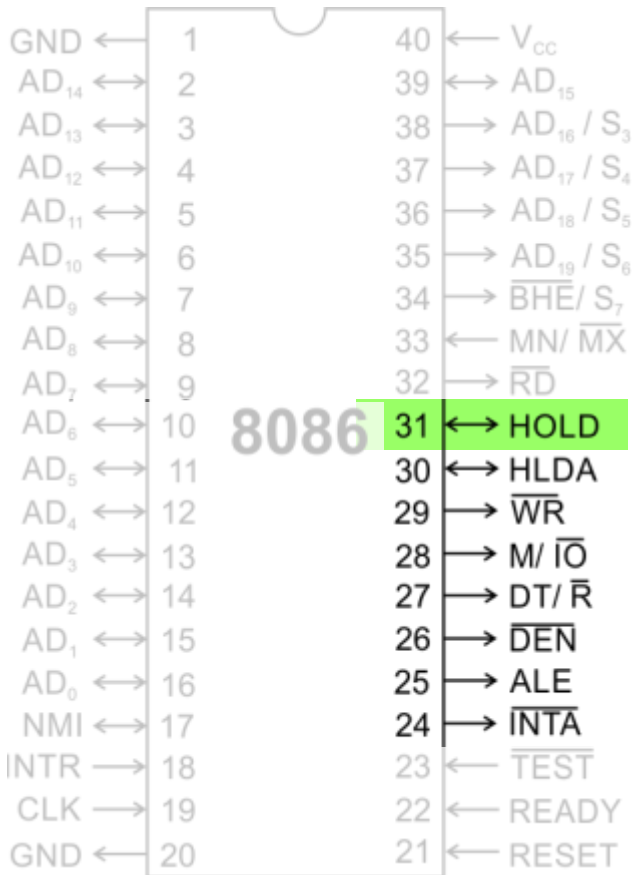
\overline{WR} Write control signal; asserted **low** Whenever processor writes data to memory or I/O port

\overline{INTA} (**Interrupt Acknowledge**) When the interrupt request is accepted by the processor, the output is **low** on this line.

Pins 24 -31

For minimum mode operation, the MN/\overline{MX} is tied to VCC (logic high)

8086 itself generates all the bus control signals

**HOLD**

Input signal to the processor from the bus masters as a request to grant the control of the bus.

Usually used by the DMA controller to get the control of the bus.

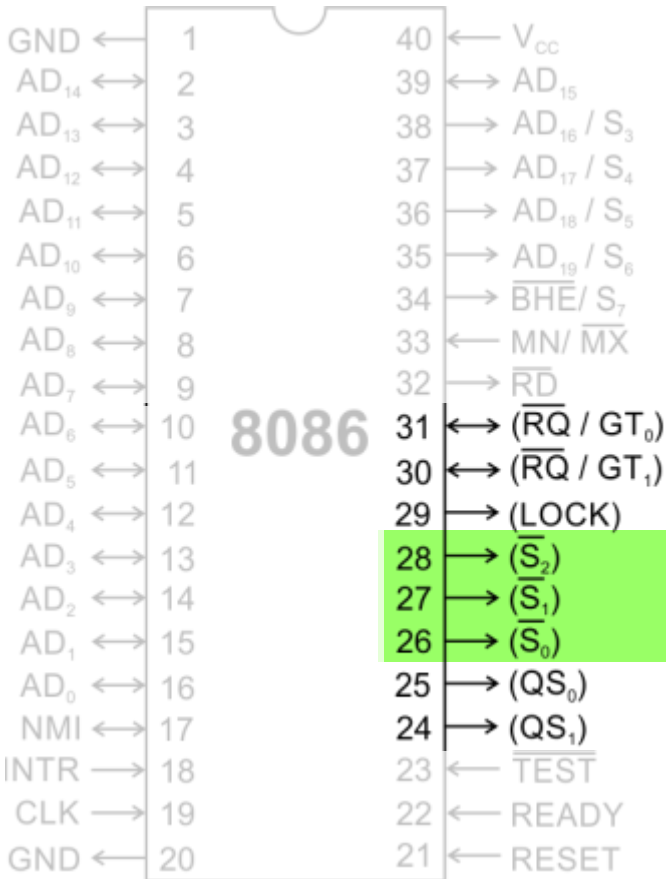
HLDA

(Hold Acknowledge) Acknowledge signal by the processor to the bus master requesting the control of the bus through HOLD.

The acknowledge is asserted high, when the processor accepts HOLD.

During maximum mode operation, the $\overline{MN}/\overline{MX}$ is grounded (logic low)

Pins 24 -31 are reassigned



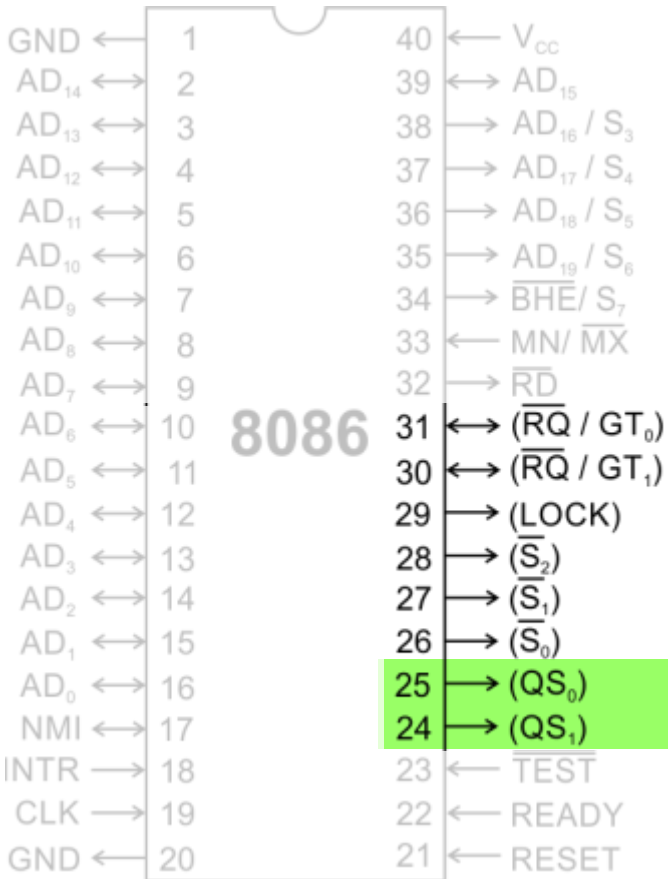
$\overline{S}_0, \overline{S}_1, \overline{S}_2$

Status signals; used by the 8086 bus controller to generate bus timing and control signals. These are decoded as shown.

Status Signal			Machine Cycle
\overline{S}_2	\overline{S}_1	\overline{S}_0	
0	0	0	Interrupt acknowledge
0	0	1	Read I/O port
0	1	0	Write I/O port
0	1	1	Halt
1	0	0	Code access
1	0	1	Read memory
1	1	0	Write memory
1	1	1	Passive/Inactive

During maximum mode operation, the $\overline{MN}/\overline{MX}$ is grounded (logic low)

Pins 24 -31 are reassigned



QS_0, QS_1

(Queue Status) The processor provides the status of queue in these lines.

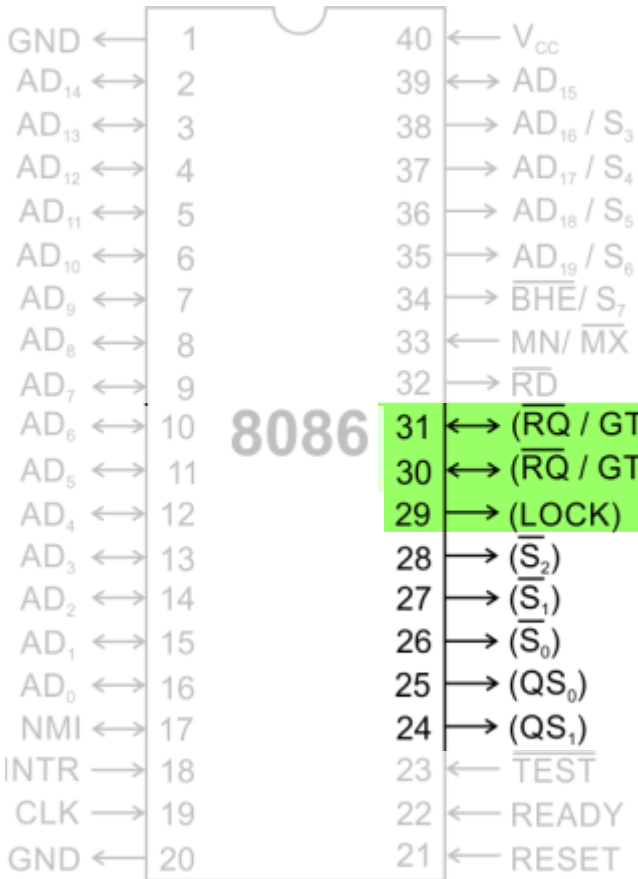
The queue status can be used by external device to track the internal status of the queue in 8086.

The output on QS_0 and QS_1 can be interpreted as shown in the table.

Queue status		Queue operation
QS_1	QS_0	
0	0	No operation
0	1	First byte of an opcode from queue
1	0	Empty the queue
1	1	Subsequent byte from queue

During maximum mode operation, the $\text{MN}/\overline{\text{MX}}$ is grounded (logic low)

Pins 24 -31 are reassigned



$\overline{\text{RQ}}/\text{GT}_0$,
 $\overline{\text{RQ}}/\text{GT}_1$

(Bus Request/ Bus Grant) These requests are used by other local bus masters to force the processor to release the local bus at the end of the processor's current bus cycle.

These pins are bidirectional.

The request on $\overline{\text{GT}}_0$ will have higher priority than $\overline{\text{GT}}_1$

$\overline{\text{LOCK}}$

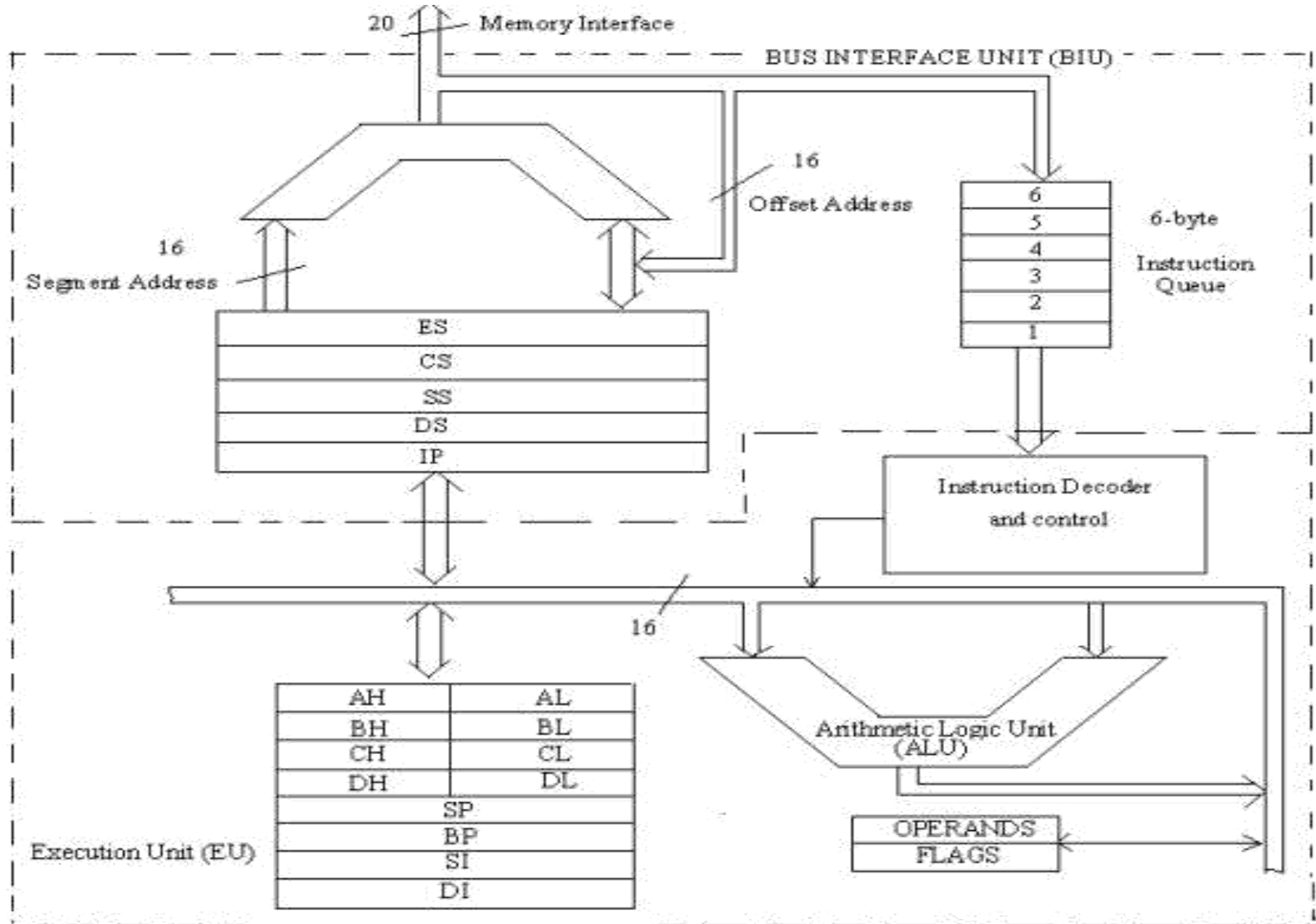
An output signal activated by the LOCK prefix instruction.

Remains active until the completion of the instruction prefixed by LOCK.

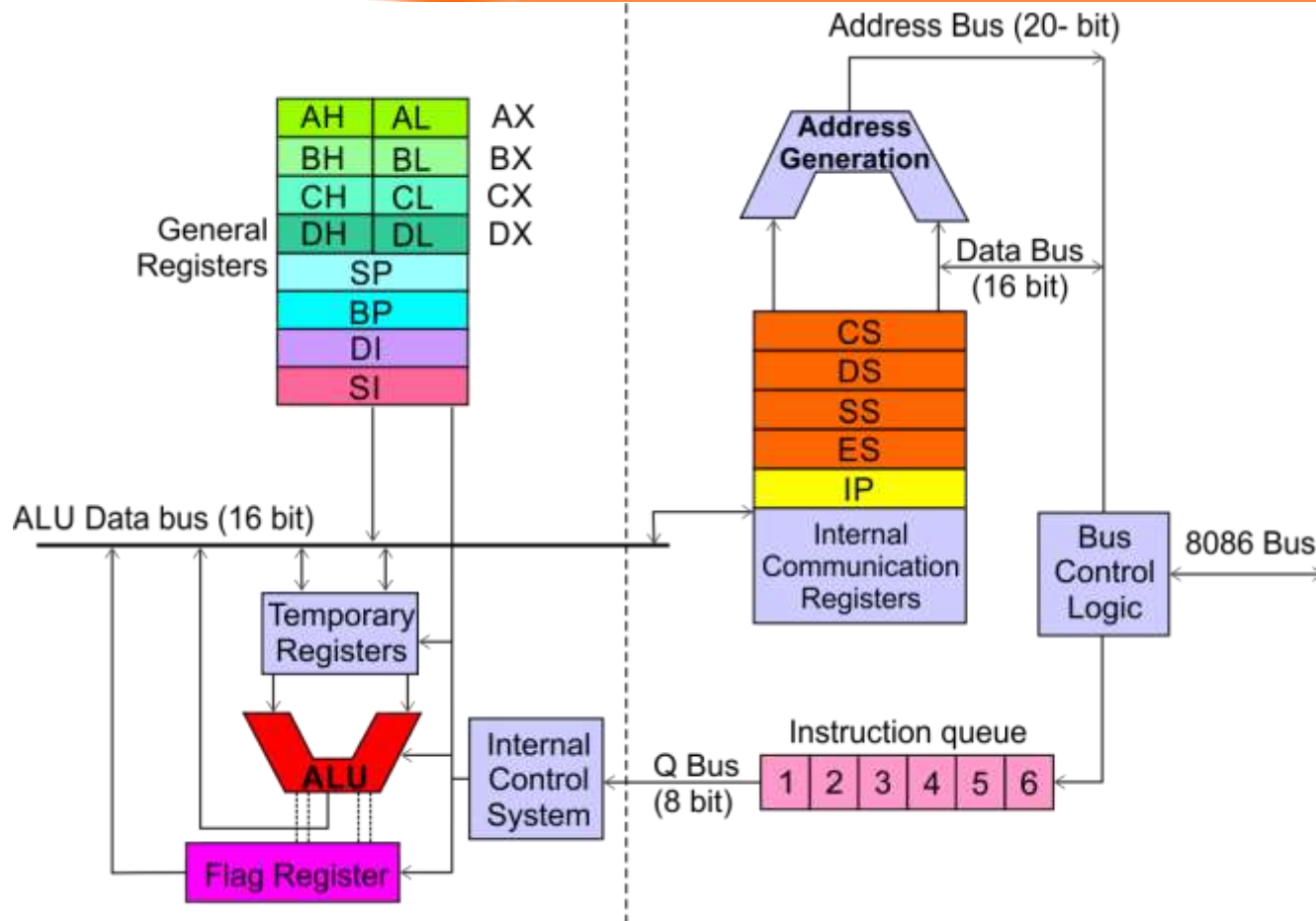
The 8086 output low on the $\overline{\text{LOCK}}$ pin while executing an instruction prefixed by LOCK to prevent other bus masters from gaining control of the system bus.

8086 Microprocessor

Architecture



Architecture



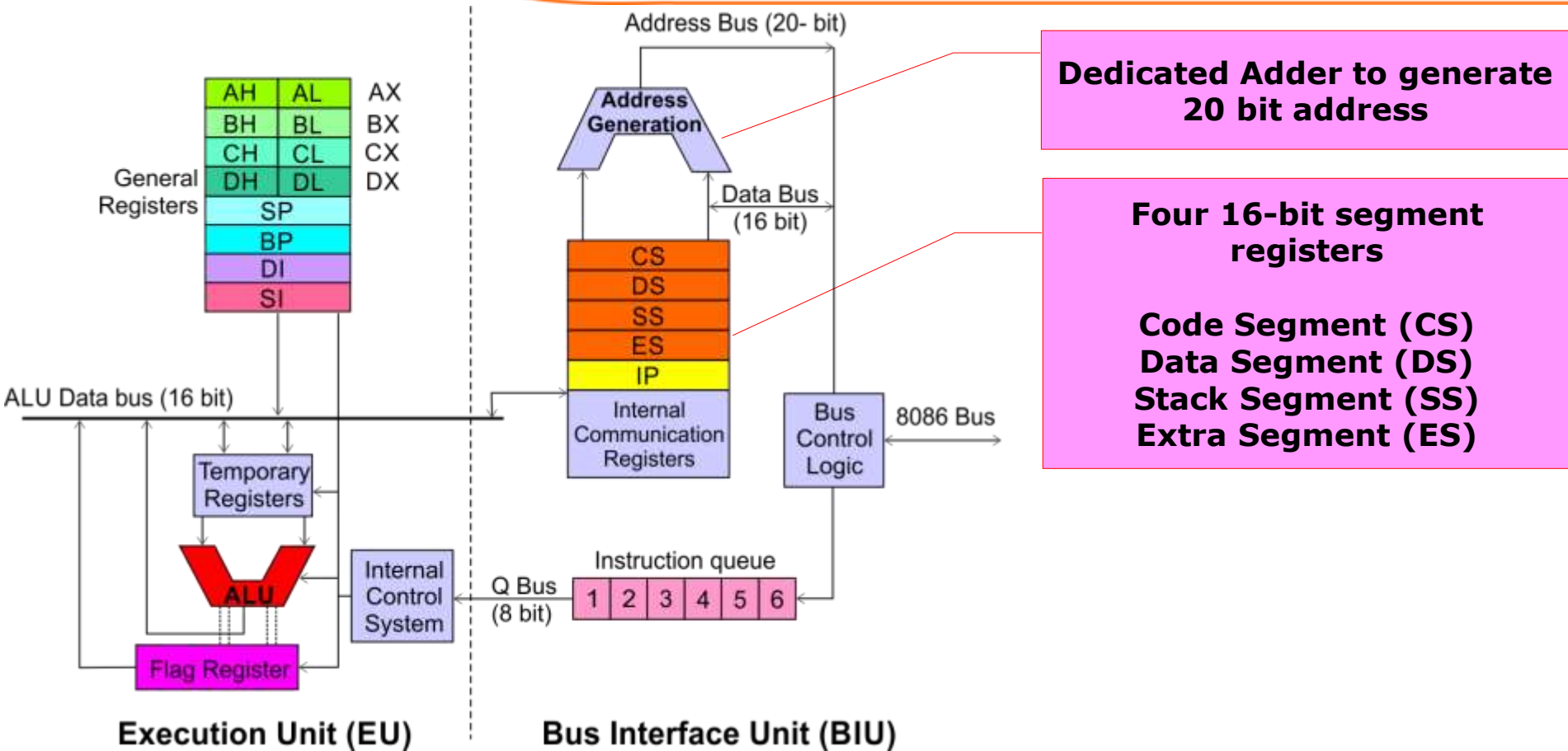
Execution Unit (EU)

EU executes instructions that have already been fetched by the BIU.

BIU and EU functions separately.

Bus Interface Unit (BIU)

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

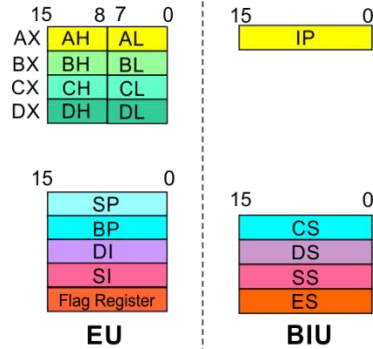


Dedicated Adder to generate 20 bit address

Four 16-bit segment registers

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

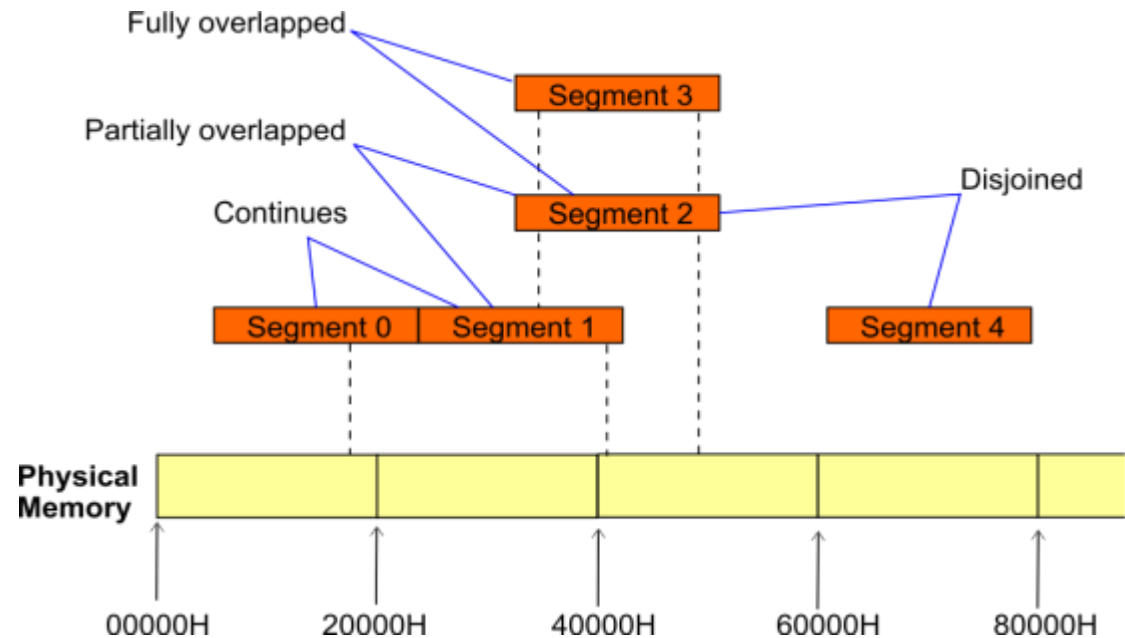
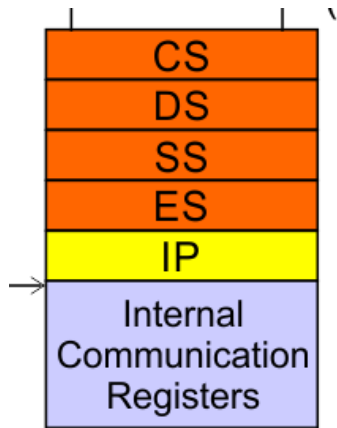
8086 registers categorized into 4 groups



Sl.No.	Type	Register width	Name of register
1	General purpose register	16 bit	AX, BX, CX, DX
		8 bit	AL, AH, BL, BH, CL, CH, DL, DH
2	Pointer register	16 bit	SP, BP
3	Index register	16 bit	SI, DI
4	Instruction Pointer	16 bit	IP
5	Segment register	16 bit	CS, DS, SS, ES
6	Flag (PSW)	16 bit	Flag register

Register	Name of the Register	Special Function
AX	16-bit Accumulator	Stores the 16-bit results of arithmetic and logic operations
AL	8-bit Accumulator	Stores the 8-bit results of arithmetic and logic operations
BX	Base register	Used to hold base value in base addressing mode to access memory data
CX	Count Register	Used to hold the count value in SHIFT, ROTATE and LOOP instructions
DX	Data Register	Used to hold data for multiplication and division operations
SP	Stack Pointer	Used to hold the offset address of top stack memory
BP	Base Pointer	Used to hold the base value in base addressing using SS register to access data from stack memory
SI	Source Index	Used to hold index value of source operand (data) for string instructions
DI	Data Index	Used to hold the index value of destination operand (data) for string operations

Segment Registers



- 8086's 1-megabyte memory is divided into segments of up to 64K bytes each.

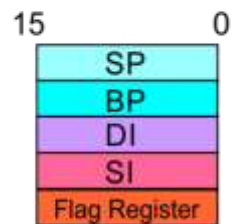
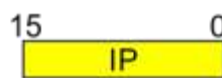
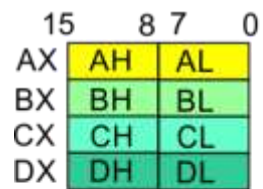
- The 8086 can directly address four segments (256 K bytes within the 1 M byte of memory) at a particular time.

- Programs obtain access to code and data in the segments by changing the segment register content to point to the desired segments.

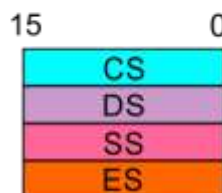
Segment Registers

Code Segment Register

- 16-bit
- CS contains the base or start of the current code segment; IP contains the distance or offset from this address to the next instruction byte to be fetched.
- BIU computes the 20-bit physical address by logically shifting the contents of CS 4-bits to the left and then adding the 16-bit contents of IP.
- That is, all instructions of a program are relative to the contents of the CS register multiplied by 16 and then offset is added provided by the IP.



EU

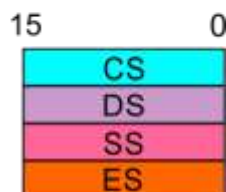
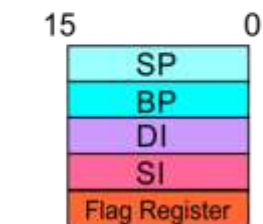
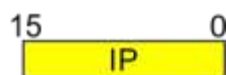
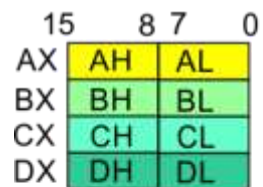


BIU

Segment Registers

Data Segment Register

- 16-bit
- Points to the current data segment; operands for most instructions are fetched from this segment.
- The 16-bit contents of the Source Index (SI) or Destination Index (DI) or a 16-bit displacement are used as offset for computing the 20-bit physical address.



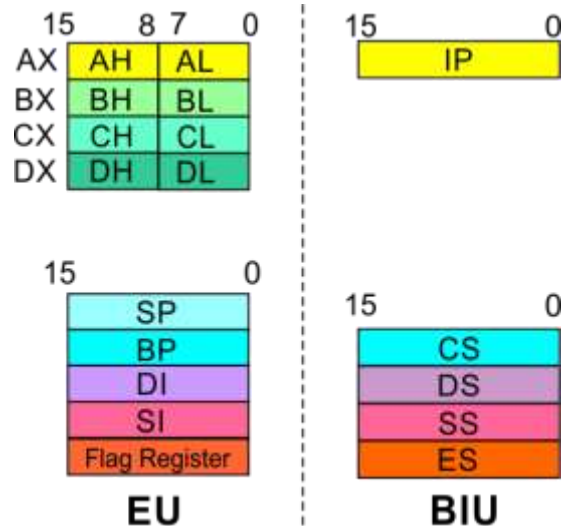
EU

BIU

Segment Registers

Stack Segment Register

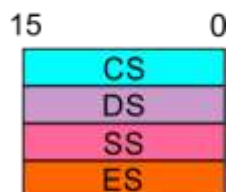
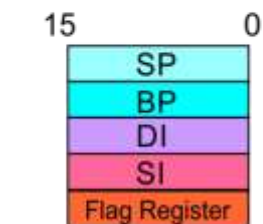
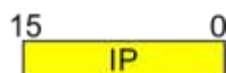
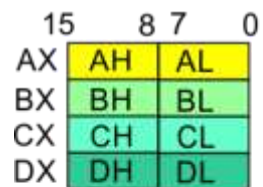
- 16-bit
- Points to the current stack.
- The 20-bit physical stack address is calculated from the Stack Segment (SS) and the Stack Pointer (SP) for stack instructions such as **PUSH** and **POP**.
- In based addressing mode, the 20-bit physical stack address is calculated from the Stack segment (SS) and the Base Pointer (BP).



Segment Registers

Extra Segment Register

- 16-bit
- Points to the extra segment in which data (in excess of 64K pointed to by the DS) is stored.
- String instructions use the ES and DI to determine the 20-bit physical address for the destination.



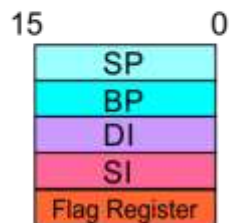
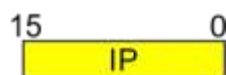
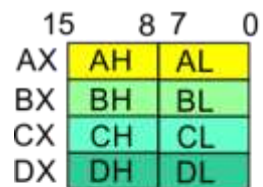
EU

BIU

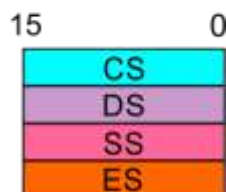
Segment Registers

Instruction Pointer

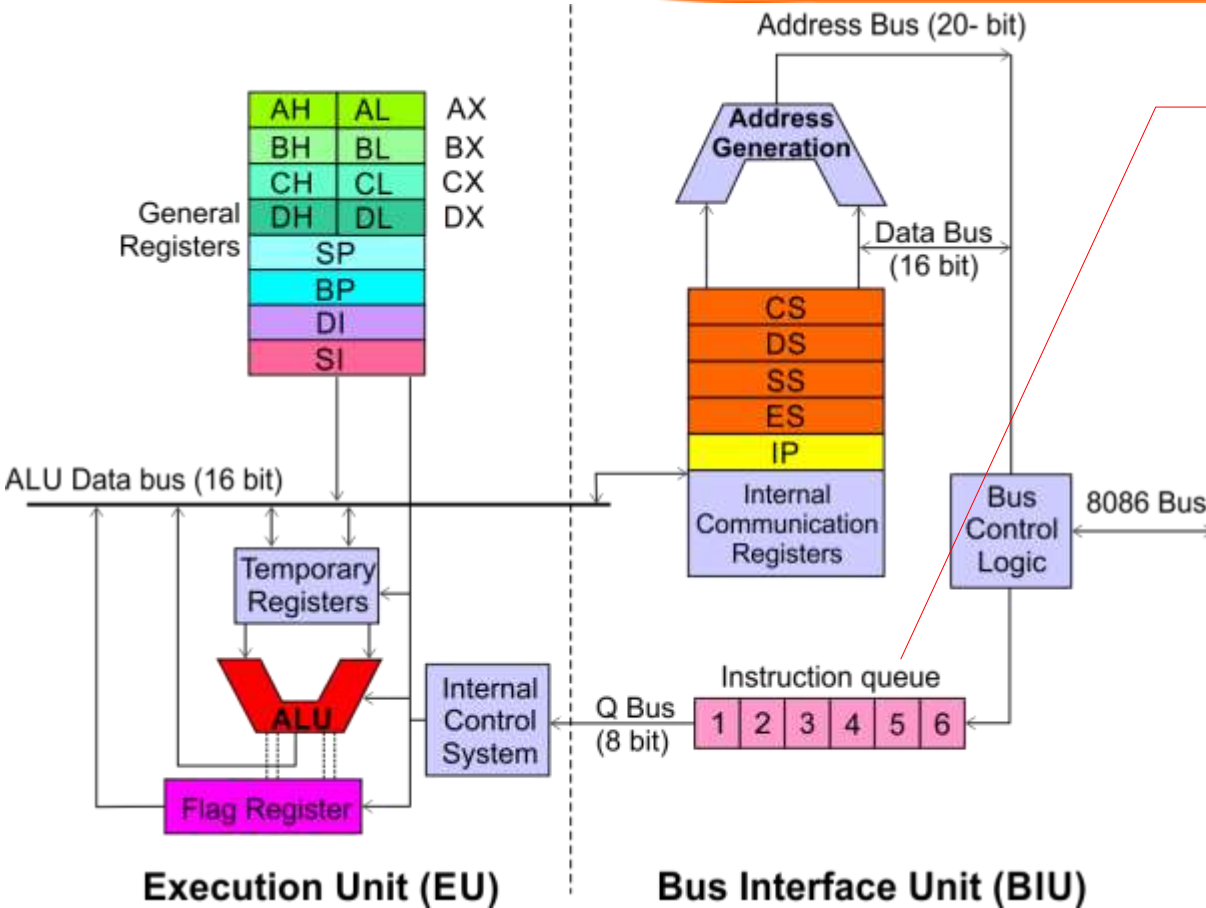
- 16-bit
- Always points to the next instruction to be executed within the currently executing code segment.
- So, this register contains the 16-bit offset address pointing to the next instruction code within the 64Kb of the code segment area.
- Its content is automatically incremented as the execution of the next instruction takes place.



EU



BIU



Instruction queue

- A group of First-In-First-Out (FIFO) in which up to 6 bytes of instruction code are pre fetched from the memory ahead of time.
- This is done in order to speed up the execution by overlapping instruction fetch with execution.
- This mechanism is known as pipelining.

EU decodes and executes instructions.

A decoder in the EU control system translates instructions.

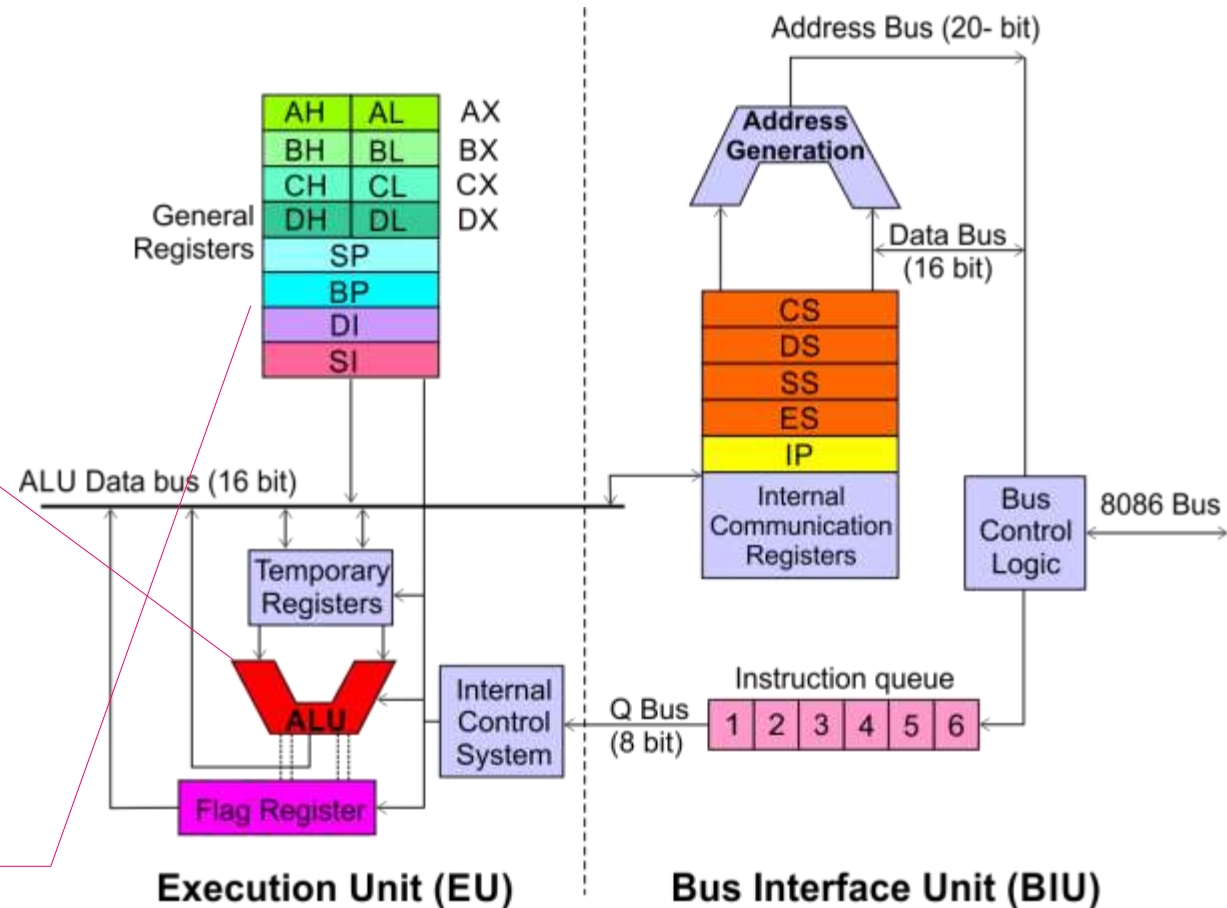
16-bit ALU for performing arithmetic and logic operation

Four general purpose registers (AX, BX, CX, DX);

Pointer registers (Stack Pointer, Base Pointer);

and

Index registers (Source Index, Destination Index) each of 16-bits



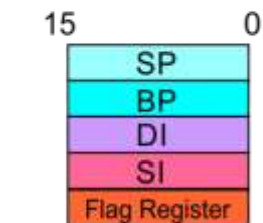
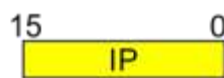
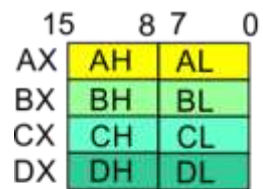
Some of the 16 bit registers can be used as two 8 bit registers as :

**AX can be used as AH and AL
 BX can be used as BH and BL
 CX can be used as CH and CL
 DX can be used as DH and DL**

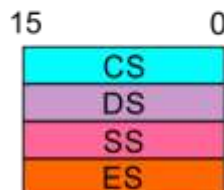
EU Registers

Accumulator Register (AX)

- Consists of two 8-bit registers AL and AH, which can be combined together and used as a 16-bit register AX.
- AL in this case contains the low order byte of the word, and AH contains the high-order byte.
- The I/O instructions use the AX or AL for inputting / outputting 16 or 8 bit data to or from an I/O port.
- Multiplication and Division instructions also use the AX or AL.



EU

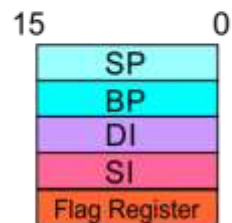
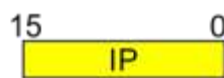
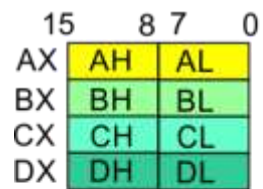


BIU

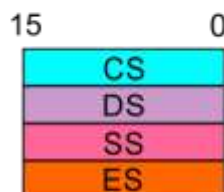
EU Registers

Base Register (BX)

- Consists of two 8-bit registers BL and BH, which can be combined together and used as a 16-bit register BX.
- BL in this case contains the low-order byte of the word, and BH contains the high-order byte.
- This is the only general purpose register whose contents can be used for addressing the 8086 memory.
- All memory references utilizing this register content for addressing use DS as the default segment register.



EU



BIU

EU Registers

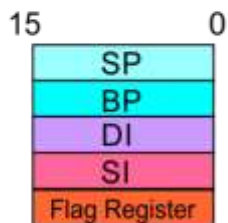
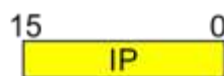
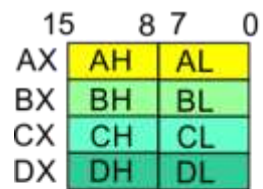
Counter Register (CX)

- Consists of two 8-bit registers CL and CH, which can be combined together and used as a 16-bit register CX.
- When combined, CL register contains the low order byte of the word, and CH contains the high-order byte.
- Instructions such as **SHIFT**, **ROTATE** and **LOOP** use the contents of CX as a counter.

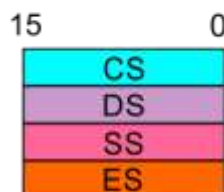
Example:

The instruction **LOOP START** automatically decrements CX by 1 without affecting flags and will check if [CX] = 0.

If it is zero, 8086 executes the next instruction; otherwise the 8086 branches to the label **START**.



EU

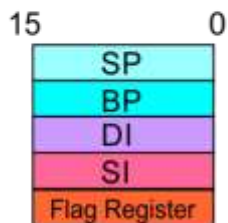
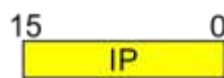
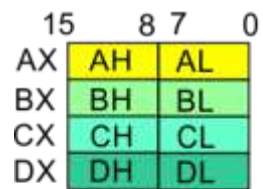


BIU

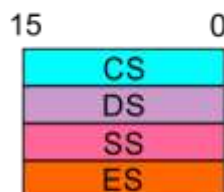
EU Registers

Data Register (DX)

- Consists of two 8-bit registers DL and DH, which can be combined together and used as a 16-bit register DX.
- When combined, DL register contains the low order byte of the word, and DH contains the high-order byte.
- Used to hold the high 16-bit result (data) in 16 X 16 multiplication or the high 16-bit dividend (data) before a $32 \div 16$ division and the 16-bit remainder after division.



EU

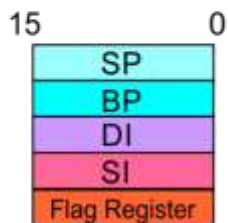
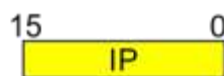
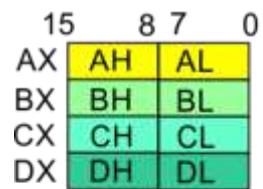


BIU

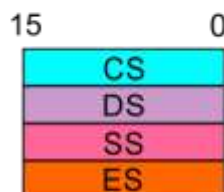
EU Registers

Stack Pointer (SP) and Base Pointer (BP)

- SP and BP are used to access data in the stack segment.
- SP is used as an offset from the current SS during execution of instructions that involve the stack segment in the external memory.
- SP contents are automatically updated (incremented/decremented) due to execution of a POP or PUSH instruction.
- BP contains an offset address in the current SS, which is used by instructions utilizing the based addressing mode.



EU

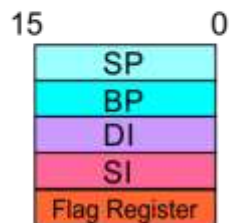
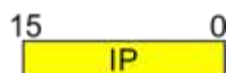
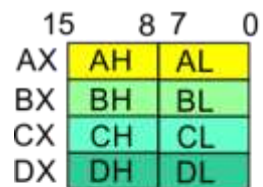


BIU

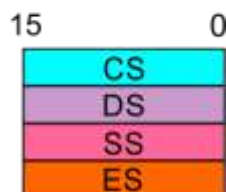
EU Registers

Source Index (SI) and Destination Index (DI)

- Used in indexed addressing.
- Instructions that process data strings use the SI and DI registers together with DS and ES respectively in order to distinguish between the source and destination addresses.



EU

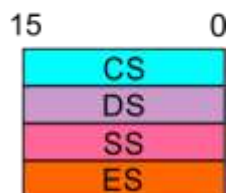
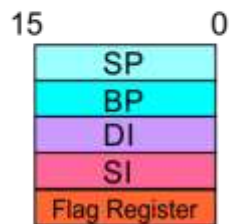
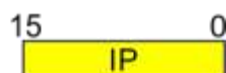
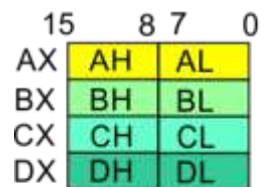


BIU

EU Registers

Source Index (SI) and Destination Index (DI)

- Used in indexed addressing.
- Instructions that process data strings use the SI and DI registers together with DS and ES respectively in order to distinguish between the source and destination addresses.



EU

BIU

Flag Register

Sign Flag

This flag is set, when the result of any computation is negative

Auxiliary Carry Flag

This is set, if there is a carry from the lowest nibble, i.e, bit three during addition, or borrow for the lowest nibble, i.e, bit three, during subtraction.

Carry Flag

This flag is set, when there is a carry out of MSB in case of addition or a borrow in case of subtraction.

Zero Flag

This flag is set, if the result of the computation or comparison performed by an instruction is zero

Parity Flag

This flag is set to 1, if the lower byte of the result contains even number of 1's ; for odd number of 1's set to zero.

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0

**Over flow Flag**

This flag is set, if an overflow occurs, i.e, if the result of a signed operation is large enough to accommodate in a destination register. The result is of more than 7-bits in size in case of 8-bit signed operation and more than 15-bits in size in case of 16-bit sign operations, then the overflow will be set.

Tarp Flag

If this flag is set, the processor enters the single step execution mode by generating internal interrupts after the execution of each instruction

Direction Flag

This is used by string manipulation instructions. If this flag bit is '0', the string is processed beginning from the lowest address to the highest address, i.e., auto incrementing mode. Otherwise, the string is processed from the highest address towards the lowest address, i.e., auto decrementing mode.

Interrupt Flag

Causes the 8086 to recognize external mask interrupts; clearing IF disables these interrupts.

ADDRESSING MODES & Instruction set

```
;PROGRAM TO ADD TWO 16-BIT DATA (METHOD-1)
```

```
DATA SEGMENT ;Assembler directive
    ORG 1104H ;Assembler directive
    SUM DW 0 ;Assembler directive
    CARRY DB 0 ;Assembler directive
```

```
DATA ENDS ;Assembler directive
```

```
CODE SEGMENT ;Assembler directive
    ASSUME CS:CODE ;Assembler directive
    ASSUME DS:DATA ;Assembler directive
    ORG 1000H ;Assembler directive
```

```
MOV AX,205AH ;Load the first data in AX register
MOV BX,40EDH ;Load the second data in BX register
MOV CL,00H ;Clear the CL register for carry
ADD AX,BX ;Add the two data, sum will be in AX
MOV SUM,AX ;Store the sum in memory location (1104H)
JNC AHEAD ;Check the status of carry flag
INC CL ;If carry flag is set,increment CL by one
AHEAD: MOV CARRY,CL ;Store the carry in memory location (1106H)
HLT
```

```
CODE ENDS ;Assembler directive
END ;Assembler directive
```

Program

A set of instructions written to solve a problem.

Instruction

Directions which a microprocessor follows to execute a task or part of a task.

Computer language

High Level

Low Level

Machine Language

Assembly Language

- Binary bits

- English Alphabets
- 'Mnemonics'
- Assembler
Mnemonics → Machine Language

ADDRESSING MODES

Addressing Modes

- Every instruction of a program has to operate on a data.
- The different ways in which a source operand is denoted in an instruction are known as addressing modes.

1. Register Addressing

2. Immediate Addressing

Group I : Addressing modes for register and immediate data

3. Direct Addressing

4. Register Indirect Addressing

5. Based Addressing

6. Indexed Addressing

7. Based Index Addressing

8. String Addressing

Group II : Addressing modes for memory data

9. Direct I/O port Addressing

10. Indirect I/O port Addressing

Group III : Addressing modes for I/O ports

11. Relative Addressing

Group IV : Relative Addressing mode

12. Implied Addressing

Group V : Implied Addressing mode

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing

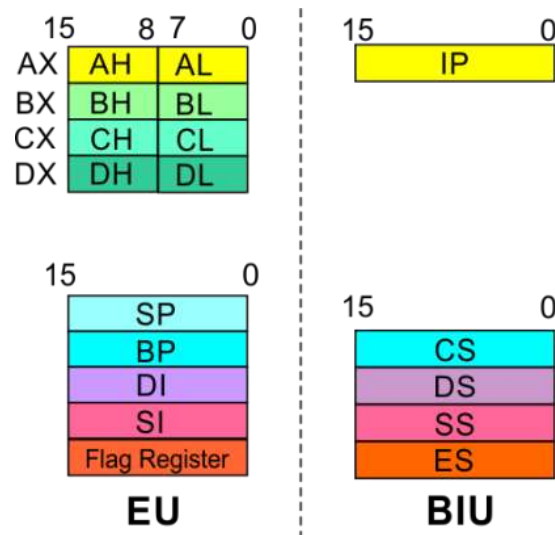
The instruction will specify the name of the register which holds the data to be operated by the instruction.

Example:

MOV CL, DH

The content of 8-bit register DH is moved to another 8-bit register CL

$(CL) \leftarrow (DH)$



1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing

In immediate addressing mode, an 8-bit or 16-bit data is specified as part of the instruction

Example:

MOV DL, 08H

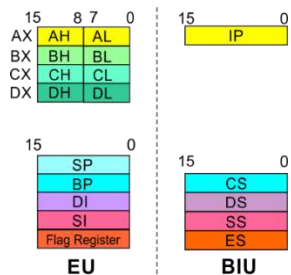
The 8-bit data (08_H) given in the instruction is moved to DL

(DL) ← 08_H

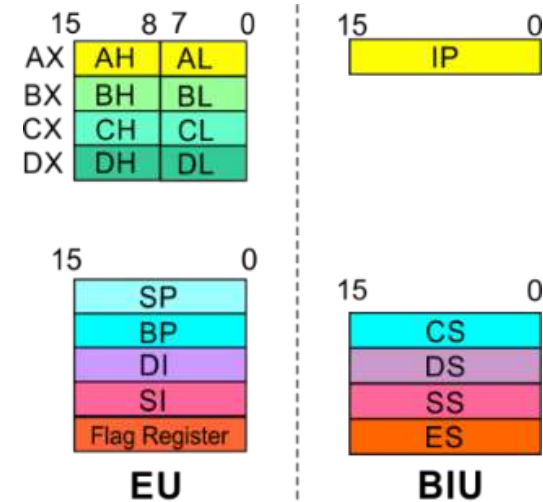
MOV AX, 0A9FH

The 16-bit data (0A9F_H) given in the instruction is moved to AX register

(AX) ← 0A9F_H



- 20 Address lines \Rightarrow 8086 can address up to $2^{20} = 1\text{M}$ bytes of memory
- However, the largest register is only 16 bits
- Physical Address will have to be calculated
Physical Address : Actual address of a byte in memory. i.e. the value which goes out onto the address bus.
- Memory Address represented in the form –
Seg : Offset (Eg - 89AB:F012)
- Each time the processor wants to access memory, it takes the contents of a segment register, shifts it one hexadecimal place to the left (same as multiplying by 16_{10}), then add the required offset to form the 20-bit address



16 bytes of contiguous memory

89AB : F012 \rightarrow 89AB \rightarrow 89AB0 (Paragraph to byte $\rightarrow 89AB \times 10 = 89AB0$)
 F012 \rightarrow 0F012 (Offset is already in byte unit)
 + -----
 98AC2 (The absolute address)

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing

Here, the effective address of the memory location at which the data operand is stored is given in the instruction.

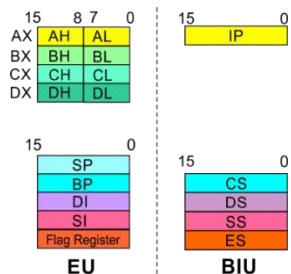
The effective address is just a 16-bit number written directly in the instruction.

Example:

```
MOV BX, [1354H]
MOV BL, [0400H]
```

The square brackets around the 1354_H denotes the contents of the memory location. When executed, this instruction will copy the contents of the memory location into BX register.

This addressing mode is called direct because the displacement of the operand from the segment base is specified directly in the instruction.



1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing

In Register indirect addressing, name of the register which holds the effective address (EA) will be specified in the instruction.

Registers used to hold EA are any of the following registers:

BX, BP, DI and SI.

Content of the DS register is used for base address calculation.

Example:

MOV CX, [BX]

Operations:

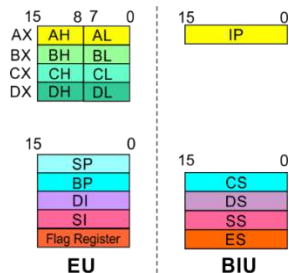
$$\begin{aligned} EA &= (BX) \\ BA &= (DS) \times 16_{10} \\ MA &= BA + EA \end{aligned}$$

$$(CX) \leftarrow (MA) \text{ or,}$$

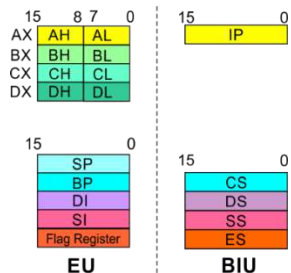
$$(CL) \leftarrow (MA)$$

$$(CH) \leftarrow (MA + 1)$$

Note : Register/ memory enclosed in brackets refer to content of register/ memory



1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing



In Based Addressing, **BX** or **BP** is used to hold the base value for effective address and a **signed 8-bit** or **unsigned 16-bit** displacement will be specified in the instruction.

In case of 8-bit displacement, it is **sign extended** to 16-bit before adding to the base value.

When **BX** holds the base value of EA, 20-bit physical address is calculated from **BX** and **DS**.

When **BP** holds the base value of EA, **BP** and **SS** is used.

Example:

MOV AX, [BX + 08H]

Operations:

$0008_H \leftarrow 08_H$ (Sign extended)

$EA = (BX) + 0008_H$

$BA = (DS) \times 16_{10}$

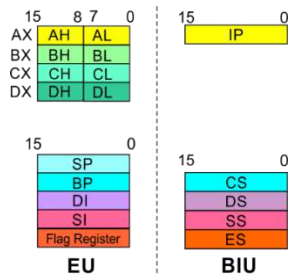
$MA = BA + EA$

$(AX) \leftarrow (MA)$ or,

$(AL) \leftarrow (MA)$

$(AH) \leftarrow (MA + 1)$

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing



SI or DI register is used to hold an index value for memory data and a signed 8-bit or unsigned 16-bit displacement will be specified in the instruction.

Displacement is added to the index value in SI or DI register to obtain the EA.

In case of 8-bit displacement, it is sign extended to 16-bit before adding to the base value.

Example:

MOV CX, [SI + 0A2H]

Operations:

$FFA2_H \leftarrow A2_H$ (Sign extended)

$EA = (SI) + FFA2_H$

$BA = (DS) \times 16_{10}$

$MA = BA + EA$

$(CX) \leftarrow (MA)$ or,

$(CL) \leftarrow (MA)$

$(CH) \leftarrow (MA + 1)$

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing

In Based Index Addressing, the effective address is computed from the sum of a base register (BX or BP), an index register (SI or DI) and a displacement.

Example:

MOV DX, [BX + SI + 0AH]

Operations:

$000A_H \leftarrow 0A_H$ (Sign extended)

$EA = (BX) + (SI) + 000A_H$

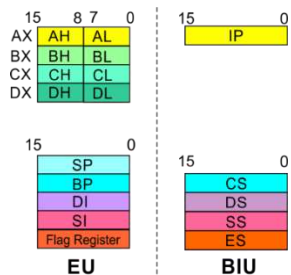
$BA = (DS) \times 16_{10}$

$MA = BA + EA$

$(DX) \leftarrow (MA)$ or,

$(DL) \leftarrow (MA)$

$(DH) \leftarrow (MA + 1)$



1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing

Note : Effective address of the Extra segment register

Employed in string operations to operate on string data.

The effective address (EA) of source data is stored in **SI register** and the EA of destination is stored in **DI register**.

Segment register for calculating base address of source data is **DS** and that of the destination data is **ES**

Example: MOVSB

Operations:

Calculation of source memory location:

$$EA = (SI) \quad BA = (DS) \times 16_{10} \quad MA = BA + EA$$

Calculation of destination memory location:

$$EA_E = (DI) \quad BA_E = (ES) \times 16_{10} \quad MA_E = BA_E + EA_E$$

$$(MAE) \leftarrow (MA)$$

If $DF = 1$, then $(SI) \leftarrow (SI) - 1$ and $(DI) = (DI) - 1$

If $DF = 0$, then $(SI) \leftarrow (SI) + 1$ and $(DI) = (DI) + 1$

1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing

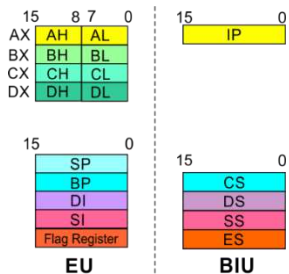
These addressing modes are used to access data from standard I/O mapped devices or ports.

In **direct port addressing mode**, an 8-bit port address is directly specified in the instruction.

Example: `IN AL, [09H]`

Operations: $PORT_{addr} = 09_H$
 $(AL) \leftarrow (PORT)$

Content of port with address 09_H is moved to AL register



1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing

In this addressing mode, the effective address of a program instruction is specified relative to Instruction Pointer (IP) by an 8-bit signed displacement.

Example: **JZ 0AH**

Operations:

$000A_H \leftarrow 0A_H$ (sign extend)

If $ZF = 1$, then

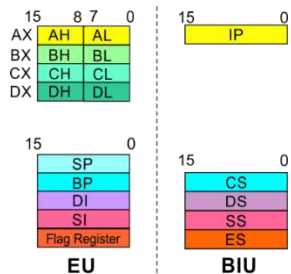
$EA = (IP) + 000A_H$

$BA = (CS) \times 16_{10}$

$MA = BA + EA$

If $ZF = 1$, then the program control jumps to new address calculated above.

If $ZF = 0$, then next instruction of the program is executed.

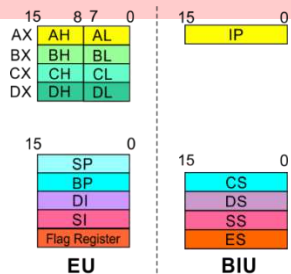


1. Register Addressing
2. Immediate Addressing
3. Direct Addressing
4. Register Indirect Addressing
5. Based Addressing
6. Indexed Addressing
7. Based Index Addressing
8. String Addressing
9. Direct I/O port Addressing
10. Indirect I/O port Addressing
11. Relative Addressing
12. Implied Addressing

Instructions using this mode have no operands. The instruction itself will specify the data to be operated by the instruction.

Example: CLC

This clears the carry flag to zero.



INSTRUCTION SET

8086 supports 6 types of instructions.

- 1. Data Transfer Instructions**
- 2. Arithmetic Instructions**
- 3. Logical Instructions**
- 4. String manipulation Instructions**
- 5. Process Control Instructions**
- 6. Control Transfer Instructions**

1. Data Transfer Instructions

Instructions that are used to transfer data/ address in to registers, memory locations and I/O ports.

Generally involve two operands: Source operand and Destination operand of the same size.

Source: Register or a memory location or an immediate data
Destination : Register or a memory location.

The size should be a either a byte or a word.

A 8-bit data can only be moved to 8-bit register/ memory and a 16-bit data can be moved to 16-bit register/ memory.

1. Data Transfer Instructions

Mnemonics: **MOV, XCHG, PUSH, POP, IN, OUT ...****MOV reg2/ mem, reg1/ mem**

MOV reg2, reg1
 MOV mem, reg1
 MOV reg2, mem

(reg2) ← (reg1)
 (mem) ← (reg1)
 (reg2) ← (mem)

MOV reg/ mem, data

MOV reg, data
 MOV mem, data

(reg) ← data
 (mem) ← data

XCHG reg2/ mem, reg1

XCHG reg2, reg1
 XCHG mem, reg1

(reg2) ↔ (reg1)
 (mem) ↔ (reg1)

1. Data Transfer Instructions

Mnemonics: **MOV, XCHG, PUSH, POP, IN, OUT ...****PUSH reg16/ mem****PUSH reg16**

$$(\text{SP}) \leftarrow (\text{SP}) - 2$$

$$\text{MA}_S = (\text{SS}) \times 16_{10} + \text{SP}$$

$$(\text{MA}_S; \text{MA}_S + 1) \leftarrow (\text{reg16})$$
PUSH mem

$$(\text{SP}) \leftarrow (\text{SP}) - 2$$

$$\text{MA}_S = (\text{SS}) \times 16_{10} + \text{SP}$$

$$(\text{MA}_S; \text{MA}_S + 1) \leftarrow (\text{mem})$$
POP reg16/ mem**POP reg16**

$$\text{MA}_S = (\text{SS}) \times 16_{10} + \text{SP}$$

$$(\text{reg16}) \leftarrow (\text{MA}_S; \text{MA}_S + 1)$$

$$(\text{SP}) \leftarrow (\text{SP}) + 2$$
POP mem

$$\text{MA}_S = (\text{SS}) \times 16_{10} + \text{SP}$$

$$(\text{mem}) \leftarrow (\text{MA}_S; \text{MA}_S + 1)$$

$$(\text{SP}) \leftarrow (\text{SP}) + 2$$

1. Data Transfer Instructions

Mnemonics: **MOV, XCHG, PUSH, POP, IN, OUT ...**

IN A, [DX]		OUT [DX], A	
IN AL, [DX]	$\text{PORT}_{\text{addr}} = (\text{DX})$ $(\text{AL}) \leftarrow (\text{PORT})$	OUT [DX], AL	$\text{PORT}_{\text{addr}} = (\text{DX})$ $(\text{PORT}) \leftarrow (\text{AL})$
IN AX, [DX]	$\text{PORT}_{\text{addr}} = (\text{DX})$ $(\text{AX}) \leftarrow (\text{PORT})$	OUT [DX], AX	$\text{PORT}_{\text{addr}} = (\text{DX})$ $(\text{PORT}) \leftarrow (\text{AX})$
IN A, addr8		OUT addr8, A	
IN AL, addr8	$(\text{AL}) \leftarrow (\text{addr8})$	OUT addr8, AL	$(\text{addr8}) \leftarrow (\text{AL})$
IN AX, addr8	$(\text{AX}) \leftarrow (\text{addr8})$	OUT addr8, AX	$(\text{addr8}) \leftarrow (\text{AX})$

2. Arithmetic Instructions

Mnemonics: **ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP...**

ADD reg2/ mem, reg1/mem

ADC reg2, reg1
 ADC reg2, mem
 ADC mem, reg1

$(\text{reg2}) \leftarrow (\text{reg1}) + (\text{reg2})$
 $(\text{reg2}) \leftarrow (\text{reg2}) + (\text{mem})$
 $(\text{mem}) \leftarrow (\text{mem}) + (\text{reg1})$

ADD reg/mem, data

ADD reg, data
 ADD mem, data

$(\text{reg}) \leftarrow (\text{reg}) + \text{data}$
 $(\text{mem}) \leftarrow (\text{mem}) + \text{data}$

ADD A, data

ADD AL, data8
 ADD AX, data16

$(\text{AL}) \leftarrow (\text{AL}) + \text{data8}$
 $(\text{AX}) \leftarrow (\text{AX}) + \text{data16}$

2. Arithmetic Instructions

Mnemonics: **ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP...**

ADC reg2/ mem, reg1/mem

ADC reg2, reg1
ADC reg2, mem
ADC mem, reg1

$(\text{reg2}) \leftarrow (\text{reg1}) + (\text{reg2}) + \text{CF}$
 $(\text{reg2}) \leftarrow (\text{reg2}) + (\text{mem}) + \text{CF}$
 $(\text{mem}) \leftarrow (\text{mem}) + (\text{reg1}) + \text{CF}$

ADC reg/mem, data

ADC reg, data
ADC mem, data

$(\text{reg}) \leftarrow (\text{reg}) + \text{data} + \text{CF}$
 $(\text{mem}) \leftarrow (\text{mem}) + \text{data} + \text{CF}$

ADDC A, data

ADD AL, data8
ADD AX, data16

$(\text{AL}) \leftarrow (\text{AL}) + \text{data8} + \text{CF}$
 $(\text{AX}) \leftarrow (\text{AX}) + \text{data16} + \text{CF}$

2. Arithmetic Instructions

Mnemonics: **ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP...**

SUB reg2/ mem, reg1/mem

SUB reg2, reg1
SUB reg2, mem
SUB mem, reg1

(reg2) ← (reg1) - (reg2)
(reg2) ← (reg2) - (mem)
(mem) ← (mem) - (reg1)

SUB reg/mem, data

SUB reg, data
SUB mem, data

(reg) ← (reg) - data
(mem) ← (mem) - data

SUB A, data

SUB AL, data8
SUB AX, data16

(AL) ← (AL) - data8
(AX) ← (AX) - data16

2. Arithmetic Instructions

Mnemonics: **ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP...**

SBB reg2/ mem, reg1/mem

SBB reg2, reg1
SBB reg2, mem
SBB mem, reg1

$(\text{reg2}) \leftarrow (\text{reg1}) - (\text{reg2}) - \text{CF}$
 $(\text{reg2}) \leftarrow (\text{reg2}) - (\text{mem}) - \text{CF}$
 $(\text{mem}) \leftarrow (\text{mem}) - (\text{reg1}) - \text{CF}$

SBB reg/mem, data

SBB reg, data
SBB mem, data

$(\text{reg}) \leftarrow (\text{reg}) - \text{data} - \text{CF}$
 $(\text{mem}) \leftarrow (\text{mem}) - \text{data} - \text{CF}$

SBB A, data

SBB AL, data8
SBB AX, data16

$(\text{AL}) \leftarrow (\text{AL}) - \text{data8} - \text{CF}$
 $(\text{AX}) \leftarrow (\text{AX}) - \text{data16} - \text{CF}$

2. Arithmetic Instructions

Mnemonics: **ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP...**

INC reg/ mem**INC reg8** $(\text{reg8}) \leftarrow (\text{reg8}) + 1$ **INC reg16** $(\text{reg16}) \leftarrow (\text{reg16}) + 1$ **INC mem** $(\text{mem}) \leftarrow (\text{mem}) + 1$ **DEC reg/ mem****DEC reg8** $(\text{reg8}) \leftarrow (\text{reg8}) - 1$ **DEC reg16** $(\text{reg16}) \leftarrow (\text{reg16}) - 1$ **DEC mem** $(\text{mem}) \leftarrow (\text{mem}) - 1$

2. Arithmetic Instructions

Mnemonics: **ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP...**

<p>MUL reg/ mem</p> <p>MUL reg</p> <p>MUL mem</p>	<p><u>For byte</u> : $(AX) \leftarrow (AL) \times (\text{reg8})$ <u>For word</u> : $(DX)(AX) \leftarrow (AX) \times (\text{reg16})$</p> <p><u>For byte</u> : $(AX) \leftarrow (AL) \times (\text{mem8})$ <u>For word</u> : $(DX)(AX) \leftarrow (AX) \times (\text{mem16})$</p>
<p>IMUL reg/ mem</p> <p>IMUL reg</p> <p>IMUL mem</p>	<p><u>For byte</u> : $(AX) \leftarrow (AL) \times (\text{reg8})$ <u>For word</u> : $(DX)(AX) \leftarrow (AX) \times (\text{reg16})$</p> <p><u>For byte</u> : $(AX) \leftarrow (AX) \times (\text{mem8})$ <u>For word</u> : $(DX)(AX) \leftarrow (AX) \times (\text{mem16})$</p>

2. Arithmetic Instructions

Mnemonics: **ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP...**

DIV reg/ mem**DIV reg****For 16-bit :- 8-bit :**

(AL) ← (AX) :- (reg8) Quotient

(AH) ← (AX) MOD(reg8) Remainder

For 32-bit :- 16-bit :

(AX) ← (DX)(AX) :- (reg16) Quotient

(DX) ← (DX)(AX) MOD(reg16) Remainder

DIV mem**For 16-bit :- 8-bit :**

(AL) ← (AX) :- (mem8) Quotient

(AH) ← (AX) MOD(mem8) Remainder

For 32-bit :- 16-bit :

(AX) ← (DX)(AX) :- (mem16) Quotient

(DX) ← (DX)(AX) MOD(mem16) Remainder

2. Arithmetic Instructions

Mnemonics: **ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP...**

IDIV reg/ mem**IDIV reg****For 16-bit :- 8-bit :**

(AL) ← (AX) :- (reg8) Quotient

(AH) ← (AX) MOD(reg8) Remainder

For 32-bit :- 16-bit :

(AX) ← (DX)(AX) :- (reg16) Quotient

(DX) ← (DX)(AX) MOD(reg16) Remainder

IDIV mem**For 16-bit :- 8-bit :**

(AL) ← (AX) :- (mem8) Quotient

(AH) ← (AX) MOD(mem8) Remainder

For 32-bit :- 16-bit :

(AX) ← (DX)(AX) :- (mem16) Quotient

(DX) ← (DX)(AX) MOD(mem16) Remainder

2. Arithmetic Instructions

Mnemonics: **ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP...**

CMP reg2/mem, reg1/ mem**CMP reg2, reg1****Modify flags \leftarrow (reg2) - (reg1)****If (reg2) > (reg1) then CF=0, ZF=0, SF=0****If (reg2) < (reg1) then CF=1, ZF=0, SF=1****If (reg2) = (reg1) then CF=0, ZF=1, SF=0****CMP reg2, mem****Modify flags \leftarrow (reg2) - (mem)****If (reg2) > (mem) then CF=0, ZF=0, SF=0****If (reg2) < (mem) then CF=1, ZF=0, SF=1****If (reg2) = (mem) then CF=0, ZF=1, SF=0****CMP mem, reg1****Modify flags \leftarrow (mem) - (reg1)****If (mem) > (reg1) then CF=0, ZF=0, SF=0****If (mem) < (reg1) then CF=1, ZF=0, SF=1****If (mem) = (reg1) then CF=0, ZF=1, SF=0**

2. Arithmetic Instructions

Mnemonics: **ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP...**

CMP reg/mem, data

CMP reg, data

Modify flags \leftarrow (reg) - (data)

If (reg) > data then CF=0, ZF=0, SF=0

If (reg) < data then CF=1, ZF=0, SF=1

If (reg) = data then CF=0, ZF=1, SF=0

CMP mem, data

Modify flags \leftarrow (mem) - (mem)

If (mem) > data then CF=0, ZF=0, SF=0

If (mem) < data then CF=1, ZF=0, SF=1

If (mem) = data then CF=0, ZF=1, SF=0

2. Arithmetic Instructions

Mnemonics: **ADD, ADC, SUB, SBB, INC, DEC, MUL, DIV, CMP...**

CMP A, data

CMP AL, data8

Modify flags \leftarrow (AL) - data8

If (AL) > data8 then CF=0, ZF=0, SF=0

If (AL) < data8 then CF=1, ZF=0, SF=1

If (AL) = data8 then CF=0, ZF=1, SF=0

CMP AX, data16

Modify flags \leftarrow (AX) - data16

If (AX) > data16 then CF=0, ZF=0, SF=0

If (mem) < data16 then CF=1, ZF=0, SF=1

If (mem) = data16 then CF=0, ZF=1, SF=0

3. Logical Instructions

Mnemonics: **AND, OR, XOR, TEST, SHR, SHL, RCR, RCL ...**

AND A, data AND AL, data8	$(AL) \leftarrow (AL) \& \text{data8}$
AND AX, data16	$(AX) \leftarrow (AX) \& \text{data16}$
AND reg/mem, data AND reg, data	$(\text{reg}) \leftarrow (\text{reg}) \& \text{data}$
AND mem, data	$(\text{mem}) \leftarrow (\text{mem}) \& \text{data}$

3. Logical Instructions

Mnemonics: **AND, OR, XOR, TEST, SHR, SHL, RCR, RCL ...**

OR reg2/mem, reg1/mem OR reg2, reg1	$(reg2) \leftarrow (reg2) (reg1)$
OR reg2, mem	$(reg2) \leftarrow (reg2) (mem)$
OR mem, reg1	$(mem) \leftarrow (mem) (reg1)$
OR reg/mem, data OR reg, data OR mem, data	$(reg) \leftarrow (reg) data$ $(mem) \leftarrow (mem) data$
OR A, data OR AL, data8 OR AX, data16	$(AL) \leftarrow (AL) data8$ $(AX) \leftarrow (AX) data16$

3. Logical Instructions

Mnemonics: **AND, OR, XOR, TEST, SHR, SHL, RCR, RCL ...**

XOR reg2/mem, reg1/mem	
XOR reg2, reg1	$(reg2) \leftarrow (reg2) \wedge (reg1)$
XOR reg2, mem	$(reg2) \leftarrow (reg2) \wedge (mem)$
XOR mem, reg1	$(mem) \leftarrow (mem) \wedge (reg1)$

XOR reg/mem, data	
XOR reg, data	$(reg) \leftarrow (reg) \wedge data$
XOR mem, data	$(mem) \leftarrow (mem) \wedge data$

XOR A, data	
XOR AL, data8	$(AL) \leftarrow (AL) \wedge data8$
XOR AX, data16	$(AX) \leftarrow (AX) \wedge data16$

3. Logical Instructions

Mnemonics: **AND, OR, XOR, TEST, SHR, SHL, RCR, RCL ...**

TEST reg2/mem, reg1/mem TEST reg2, reg1 TEST reg2, mem TEST mem, reg1	Modify flags \leftarrow (reg2) & (reg1) Modify flags \leftarrow (reg2) & (mem) Modify flags \leftarrow (mem) & (reg1)
TEST reg/mem, data TEST reg, data TEST mem, data	Modify flags \leftarrow (reg) & data Modify flags \leftarrow (mem) & data
TEST A, data TEST AL, data8 TEST AX, data16	Modify flags \leftarrow (AL) & data8 Modify flags \leftarrow (AX) & data16

3. Logical Instructions

Mnemonics: **AND, OR, XOR, TEST, SHR, SHL, RCR, RCL ...**

SHR reg/mem

SHR reg

i) SHR reg, 1

ii) SHR reg, CL

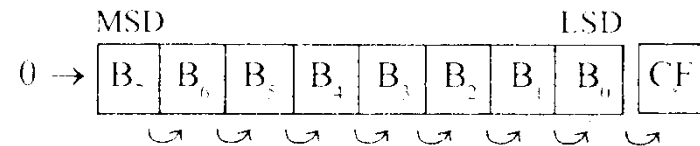
SHR mem

i) SHR mem, 1

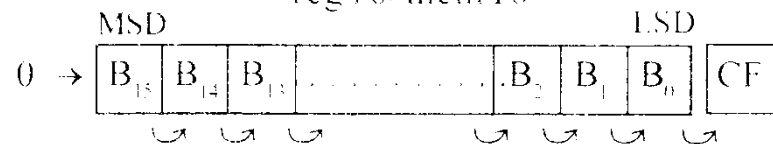
ii) SHR mem, CL

$$CF \leftarrow B_{LSD} ; B_n \leftarrow B_{n+1} ; B_{MSD} \leftarrow 0$$

reg 8 / mem 8



reg 16 / mem 16



3. Logical Instructions

Mnemonics: **AND, OR, XOR, TEST, SHR, SHL, RCR, RCL ...**

SHL reg/mem or SAL reg/mem

SHL reg or SAL reg

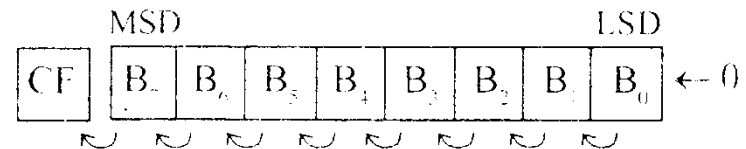
- i) SHL reg, 1 or SAL reg, 1
- ii) SHL reg, CL or SAL reg, CL

SHL mem or SAL mem

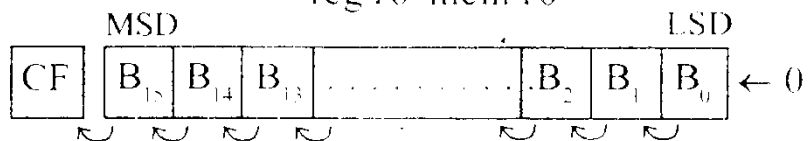
- i) SHL mem, 1 or SAL mem, 1
- ii) SHL mem, CL or SAL mem, CL

$$CF \leftarrow B_{\text{MSD}} ; B_{n+1} \leftarrow B_n ; B_{\text{LSD}} \leftarrow 0$$

reg 8 / mem 8



reg 16 / mem 16



3. Logical Instructions

Mnemonics: **AND, OR, XOR, TEST, SHR, SHL, RCR, RCL ...**

RCR reg/mem

RCR reg

i) RCR reg, 1

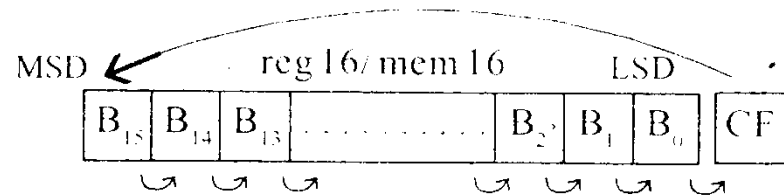
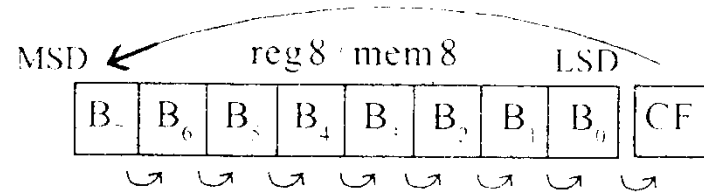
ii) RCR reg, CL

RCR mem

i) RCR mem, 1

ii) RCR mem, CL

$$B_n \leftarrow B_{n-1} ; B_{\text{MSD}} \leftarrow \text{CF} ; \text{CF} \leftarrow B_{\text{LSD}}$$



3. Logical Instructions

Mnemonics: **AND, OR, XOR, TEST, SHR, SHL, RCR, RCL ...**

ROL reg/mem

ROL reg

i) ROL reg, 1

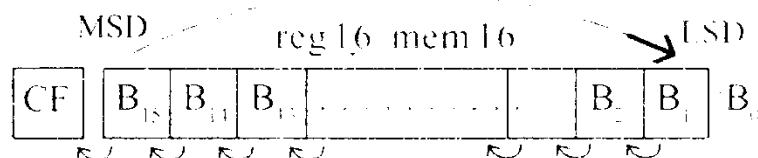
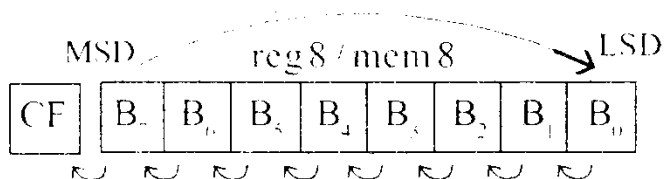
ii) ROL reg, CL

ROL mem

i) ROL mem, 1

ii) ROL mem, CL

$$B_{n+1} \leftarrow B_n ; CF \leftarrow B_{MSD} ; B_{LSD} \leftarrow B_{MSD}$$



4. String Manipulation Instructions

- ❑ **String** : Sequence of bytes or words
- ❑ 8086 instruction set includes instruction for string movement, comparison, scan, load and store.
- ❑ **REP instruction prefix** : used to repeat execution of string instructions
- ❑ **String instructions end with S or SB or SW.**
S represents string, **SB** string byte and **SW** string word.
- ❑ **Offset or effective address of the source operand is stored in SI register and that of the destination operand is stored in DI register.**
- ❑ **Depending on the status of DF, SI and DI registers are automatically updated.**
- ❑ **DF = 0 ⇒ SI and DI are incremented by 1 for byte and 2 for word.**
- ❑ **DF = 1 ⇒ SI and DI are decremented by 1 for byte and 2 for word.**

4. String Manipulation Instructions

Mnemonics: **REP, MOVS, CMPS, SCAS, LODS, STOS**

REP

REPZ/ REPE

(Repeat CMPS or SCAS until
ZF = 0)

While $CX \neq 0$ and $ZF = 1$, repeat execution of
string instruction and
 $(CX) \leftarrow (CX) - 1$

REPNZ/ REPNE

(Repeat CMPS or SCAS until
ZF = 1)

While $CX \neq 0$ and $ZF = 0$, repeat execution of
string instruction and
 $(CX) \leftarrow (CX) - 1$

4. String Manipulation Instructions

Mnemonics: **REP, MOVS, CMPS, SCAS, LODS, STOS**

MOVS

MOVSB

$$MA = (DS) \times 16_{10} + (SI)$$

$$MA_E = (ES) \times 16_{10} + (DI)$$

$$(MA_E) \leftarrow (MA)$$

If DF = 0, then $(DI) \leftarrow (DI) + 1$; $(SI) \leftarrow (SI) + 1$

If DF = 1, then $(DI) \leftarrow (DI) - 1$; $(SI) \leftarrow (SI) - 1$

MOVSW

$$MA = (DS) \times 16_{10} + (SI)$$

$$MA_E = (ES) \times 16_{10} + (DI)$$

$$(MA_E ; MA_E + 1) \leftarrow (MA ; MA + 1)$$

If DF = 0, then $(DI) \leftarrow (DI) + 2$; $(SI) \leftarrow (SI) + 2$

If DF = 1, then $(DI) \leftarrow (DI) - 2$; $(SI) \leftarrow (SI) - 2$

4. String Manipulation Instructions

Mnemonics: **REP, MOVS, CMPS, SCAS, LODS, STOS**

Compare two string byte or string word

CMPS

CMPSB

$$MA = (DS) \times 16_{10} + (SI)$$

$$MA_E = (ES) \times 16_{10} + (DI)$$

Modify flags $\leftarrow (MA) - (MA_E)$

If $(MA) > (MA_E)$, then CF = 0; ZF = 0; SF = 0

If $(MA) < (MA_E)$, then CF = 1; ZF = 0; SF = 1

If $(MA) = (MA_E)$, then CF = 0; ZF = 1; SF = 0

CMPSW

For byte operation

If DF = 0, then $(DI) \leftarrow (DI) + 1$; $(SI) \leftarrow (SI) + 1$

If DF = 1, then $(DI) \leftarrow (DI) - 1$; $(SI) \leftarrow (SI) - 1$

For word operation

If DF = 0, then $(DI) \leftarrow (DI) + 2$; $(SI) \leftarrow (SI) + 2$

If DF = 1, then $(DI) \leftarrow (DI) - 2$; $(SI) \leftarrow (SI) - 2$

4. String Manipulation Instructions

Mnemonics: **REP, MOVS, CMPS, SCAS, LODS, STOS**

Scan (compare) a string byte or word with accumulator

SCAS

SCASB

$MA_E = (ES) \times 16_{10} + (DI)$
 Modify flags $\leftarrow (AL) - (MA_E)$

If $(AL) > (MA_E)$, then $CF = 0$; $ZF = 0$; $SF = 0$

If $(AL) < (MA_E)$, then $CF = 1$; $ZF = 0$; $SF = 1$

If $(AL) = (MA_E)$, then $CF = 0$; $ZF = 1$; $SF = 0$

If $DF = 0$, then $(DI) \leftarrow (DI) + 1$

If $DF = 1$, then $(DI) \leftarrow (DI) - 1$

SCASW

$MA_E = (ES) \times 16_{10} + (DI)$
 Modify flags $\leftarrow (AX) - (MA_E)$

If $(AX) > (MA_E ; MA_E + 1)$, then $CF = 0$; $ZF = 0$; $SF = 0$

If $(AX) < (MA_E ; MA_E + 1)$, then $CF = 1$; $ZF = 0$; $SF = 1$

If $(AX) = (MA_E ; MA_E + 1)$, then $CF = 0$; $ZF = 1$; $SF = 0$

If $DF = 0$, then $(DI) \leftarrow (DI) + 2$

If $DF = 1$, then $(DI) \leftarrow (DI) - 2$

4. String Manipulation Instructions

Mnemonics: **REP, MOVS, CMPS, SCAS, LODS, STOS**

Load string byte in to AL or string word in to AX

LODS

LODSB

$MA = (DS) \times 16_{10} + (SI)$
 $(AL) \leftarrow (MA)$

If $DF = 0$, then $(SI) \leftarrow (SI) + 1$

If $DF = 1$, then $(SI) \leftarrow (SI) - 1$

LODSW

$MA = (DS) \times 16_{10} + (SI)$
 $(AX) \leftarrow (MA ; MA + 1)$

If $DF = 0$, then $(SI) \leftarrow (SI) + 2$

If $DF = 1$, then $(SI) \leftarrow (SI) - 2$

4. String Manipulation Instructions

Mnemonics: **REP, MOVS, CMPS, SCAS, LODS, STOS**

Store byte from AL or word from AX in to string

STOS

STOSB

$MA_E = (ES) \times 16_{10} + (DI)$
 $(MA_E) \leftarrow (AL)$

If $DF = 0$, then $(DI) \leftarrow (DI) + 1$

If $DF = 1$, then $(DI) \leftarrow (DI) - 1$

STOSW

$MA_E = (ES) \times 16_{10} + (DI)$
 $(MA_E ; MA_E + 1) \leftarrow (AX)$

If $DF = 0$, then $(DI) \leftarrow (DI) + 2$

If $DF = 1$, then $(DI) \leftarrow (DI) - 2$

5. Processor Control Instructions

Mnemonics	Explanation
STC	Set CF $\leftarrow 1$
CLC	Clear CF $\leftarrow 0$
CMC	Complement carry CF $\leftarrow CF'$
STD	Set direction flag DF $\leftarrow 1$
CLD	Clear direction flag DF $\leftarrow 0$
STI	Set interrupt enable flag IF $\leftarrow 1$
CLI	Clear interrupt enable flag IF $\leftarrow 0$
NOP	No operation
HLT	Halt after interrupt is set
WAIT	Wait for TEST pin active
ESC opcode mem/ reg	Used to pass instruction to a coprocessor which shares the address and data bus with the 8086
LOCK	Lock bus during next instruction

6. Control Transfer Instructions

- Transfer the control to a specific destination or target instruction
- Do not affect flags

□ 8086 Unconditional transfers

Mnemonics	Explanation
CALL reg/ mem/ disp16	Call subroutine
RET	Return from subroutine
JMP reg/ mem/ disp8/ disp16	Unconditional jump

6. Control Transfer Instructions

- ❑ **8086 signed conditional branch instructions**
 - ❑ **8086 unsigned conditional branch instructions**
- **Checks flags**
 - **If conditions are true, the program control is transferred to the new memory location in the same segment by modifying the content of IP**

6. Control Transfer Instructions

❑ 8086 signed conditional branch instructions

Name	Alternate name
JE disp8 Jump if equal	JZ disp8 Jump if result is 0
JNE disp8 Jump if not equal	JNZ disp8 Jump if not zero
JG disp8 Jump if greater	JNLE disp8 Jump if not less or equal
JGE disp8 Jump if greater than or equal	JNL disp8 Jump if not less
JL disp8 Jump if less than	JNGE disp8 Jump if not greater than or equal
JLE disp8 Jump if less than or equal	JNG disp8 Jump if not greater

❑ 8086 unsigned conditional branch instructions

Name	Alternate name
JE disp8 Jump if equal	JZ disp8 Jump if result is 0
JNE disp8 Jump if not equal	JNZ disp8 Jump if not zero
JA disp8 Jump if above	JNBE disp8 Jump if not below or equal
JAE disp8 Jump if above or equal	JNB disp8 Jump if not below
JB disp8 Jump if below	JNAE disp8 Jump if not above or equal
JBE disp8 Jump if below or equal	JNA disp8 Jump if not above

6. Control Transfer Instructions

- ❑ 8086 conditional branch instructions affecting individual flags

Mnemonics	Explanation
JC disp8	Jump if CF = 1
JNC disp8	Jump if CF = 0
JP disp8	Jump if PF = 1
JNP disp8	Jump if PF = 0
JO disp8	Jump if OF = 1
JNO disp8	Jump if OF = 0
JS disp8	Jump if SF = 1
JNS disp8	Jump if SF = 0
JZ disp8	Jump if result is zero, i.e, Z = 1
JNZ disp8	Jump if result is not zero, i.e, Z = 1

Assembler directives

- **Instructions to the Assembler regarding the program being executed.**
- **Control the generation of machine codes and organization of the program; but no machine codes are generated for assembler directives.**
- **Also called 'pseudo instructions'**
- **Used to :**
 - **specify the start and end of a program**
 - **attach value to variables**
 - **allocate storage locations to input/ output data**
 - **define start and end of segments, procedures, macros etc..**

DB**DW****SEGMENT
ENDS****ASSUME****ORG
END
EVEN
EQU****PROC
FAR
NEAR
ENDP****SHORT****MACRO
ENDM**

- **Define Byte**
- **Define a byte type (8-bit) variable**
- **Reserves specific amount of memory locations to each variable**
- **Range : $00_H - FF_H$ for unsigned value;
 $00_H - 7F_H$ for positive value and
 $80_H - FF_H$ for negative value**
- **General form : **variable DB value/ values****

Example:

```
LIST DB 7FH, 42H, 35H
```

Three consecutive memory locations are reserved for the variable LIST and each data specified in the instruction are stored as initial value in the reserved memory location

DB

DW

SEGMENT
ENDS

ASSUME

ORG
END
EVEN
EQUPROC
FAR
NEAR
ENDP

SHORT

MACRO
ENDM

- Define Word
- Define a word type (16-bit) variable
- Reserves two consecutive memory locations to each variable
- Range : $0000_H - FFFF_H$ for unsigned value;
 $0000_H - 7FFF_H$ for positive value and
 $8000_H - FFFF_H$ for negative value
- General form : **variable DW value/ values**

Example:

```
ALIST DW 6512H, 0F251H, 0CDE2H
```

Six consecutive memory locations are reserved for the variable ALIST and each 16-bit data specified in the instruction is stored in two consecutive memory location.

Assemble Directives

DB

DW

SEGMENT
ENDS

ASSUME

ORG
END
EVEN
EQU

PROC
FAR
NEAR
ENDP

SHORT

MACRO
ENDM

- **SEGMENT** : Used to indicate the beginning of a code/ data/ stack segment
- **ENDS** : Used to indicate the end of a code/ data/ stack segment
- **General form:**

Segnam SEGMENT

...
...
...
...
...
...

Segnam ENDS

Program code
or
Data Defining Statements

User defined name of
the segment

Assemble Directives

DB

DW

SEGMENT
ENDS

 ASSUME
ORG
END
EVEN
EQUPROC
FAR
NEAR
ENDP


SHORT

MACRO
ENDM

- Informs the assembler the name of the program/ data segment that should be used for a specific segment.

- General form:

ASSUME segreg : segnam, .. , segreg : segnam


 Segment Register


 User defined name of
the segment

Example:

```
ASSUME CS: ACODE, DS:ADATA
```

Tells the compiler that the instructions of the program are stored in the segment ACODE and data are stored in the segment ADATA

Assemble Directives

DB

DW

**SEGMENT
ENDS**

ASSUME

**ORG
END
EVEN
EQU**

**PROC
FAR
NEAR
ENDP**

SHORT

**MACRO
ENDM**

- **ORG** (Origin) is used to assign the starting address (Effective address) for a program/ data segment
- **END** is used to terminate a program; statements after END will be ignored
- **EVEN** : Informs the assembler to store program/ data segment starting from an even address
- **EQU** (Equate) is used to attach a value to a variable

Examples:

ORG 1000H	Informs the assembler that the statements following ORG 1000H should be stored in memory starting with effective address 1000 _H
LOOP EQU 10FEH	Value of variable LOOP is 10FE _H
_SDATA SEGMENT ORG 1200H A DB 4CH EVEN B DW 1052H _SDATA ENDS	In this data segment, effective address of memory location assigned to A will be 1200 _H and that of B will be 1202 _H and 1203 _H .

Assemble Directives

DB

DW

**SEGMENT
ENDS**

ASSUME

**ORG
END
EVEN
EQU**

**PROC
ENDP
FAR
NEAR**

SHORT

**MACRO
ENDM**

- **PROC** Indicates the beginning of a procedure
- **ENDP** End of procedure
- **FAR** Intersegment call
- **NEAR** Intrasegment call
- **General form**

procname PROC[NEAR/ FAR]

...
...
...

RET

procname ENDP

} **Program statements of the
procedure**

} **Last statement of the
procedure**

**User defined name of
the procedure**

DB

DW

SEGMENT
ENDS

ASSUME

ORG
END
EVEN
EQUPROC
ENDP
FAR
NEAR

SHORT

MACRO
ENDM

Examples:

ADD64 PROC NEAR

...

...

...

RET
ADD64 ENDP

The subroutine/ procedure named ADD64 is declared as NEAR and so the assembler will code the CALL and RET instructions involved in this procedure as near call and return

CONVERT PROC FAR

...

...

...

RET
CONVERT ENDP

The subroutine/ procedure named CONVERT is declared as FAR and so the assembler will code the CALL and RET instructions involved in this procedure as far call and return

Assemble Directives

DB

- Reserves one memory location for 8-bit signed displacement in jump instructions

DW

**SEGMENT
ENDS**

Example:

**JMP SHORT
AHEAD**

The directive will reserve one memory location for 8-bit displacement named AHEAD

**ORG
END
EVEN
EQU**

**PROC
ENDP
FAR
NEAR**

SHORT

**MACRO
ENDM**

Assemble Directives

DB

DW

SEGMENT
ENDS

ASSUME

ORG
END
EVEN
EQUPROC
ENDP
FAR
NEAR

SHORT

MACRO
ENDM

- **MACRO** Indicate the beginning of a macro

- **ENDM** End of a macro

- **General form:**

macroname **MACRO**[Arg1, Arg2 ...]

...
...
...



Program
statements in
the macro

macroname **ENDM**

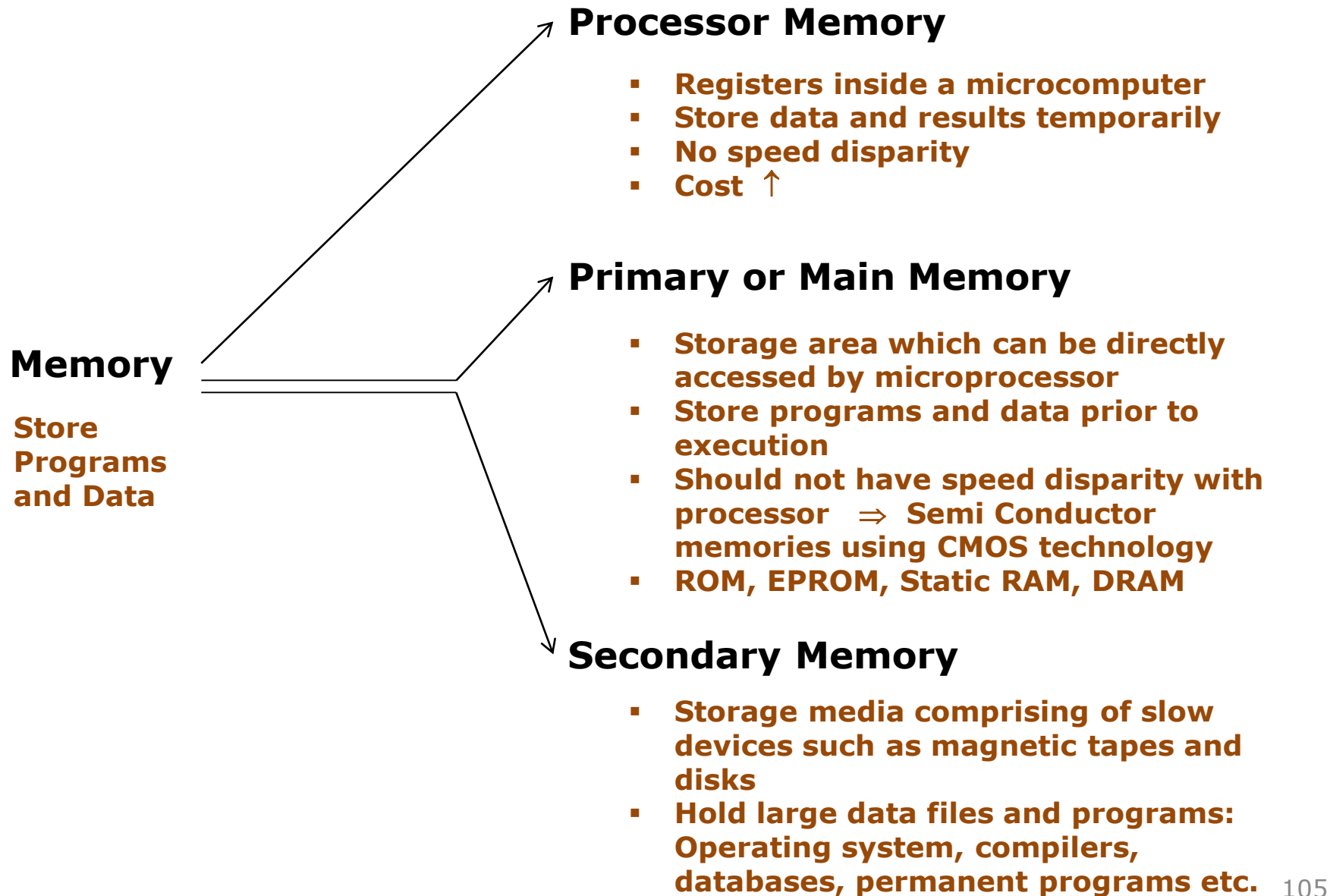
User defined name of
the macro

Unit II

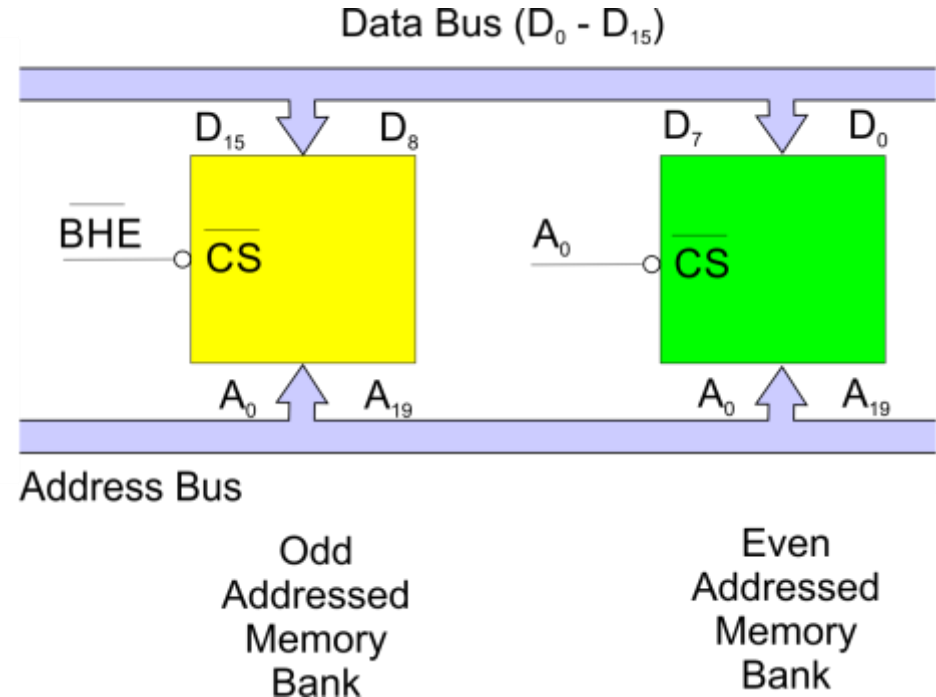
8086 Microprocessor (19 Hrs)

The Intel 8086 - architecture - MN/MX modes - 8086 addressing modes - instruction set- instruction format - assembler directives and operators - Programming with 8086 - interfacing memory and I/O ports - Comparison of 8086 and 8088 - Coprocessors - Intel 8087 - Familiarisation with Debug utility.

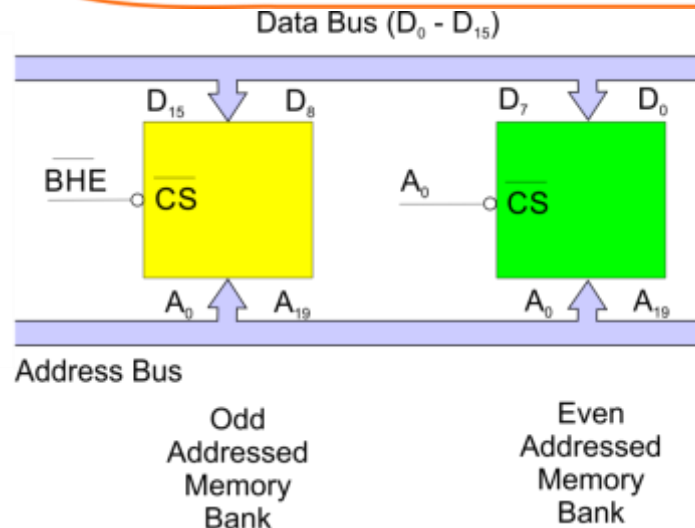
Interfacing memory and i/o ports



- Memory IC's : Byte oriented
- 8086 : 16-bit
- Word : Stored by two consecutive memory locations; for LSB and MSB
- Address of word : Address of LSB
- **Bank 0** : $A_0 = 0 \Rightarrow$ Even addressed memory bank
- **Bank 1** : $\overline{BHE} = 0 \Rightarrow$ Odd addressed memory bank



Memory organization in 8086



	Operation	\overline{BHE}	A ₀	Data Lines Used
1	Read/ Write byte at an even address	1	0	D ₇ - D ₀
2	Read/ Write byte at an odd address	0	1	D ₁₅ - D ₈
3	Read/ Write word at an even address	0	0	D ₁₅ - D ₀
4	Read/ Write word at an odd address	0	1	D ₁₅ - D ₀ in first operation byte from odd bank is transferred
		1	0	D ₇ - D ₀ in first operation byte from odd bank is transferred

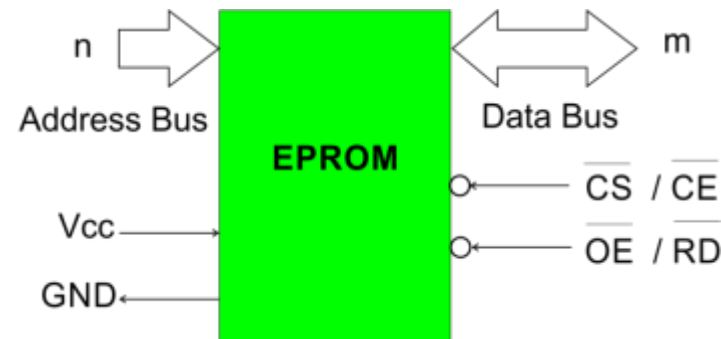
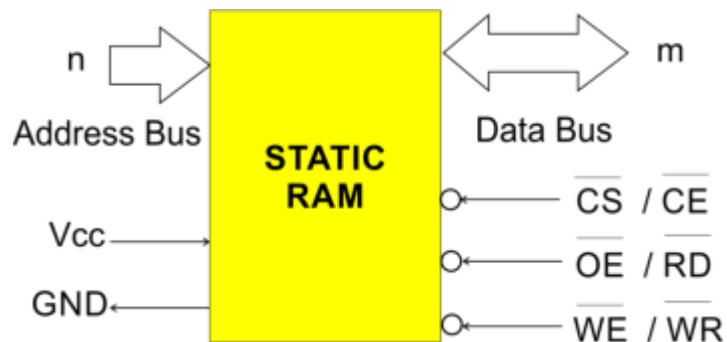
- Available memory space = EPROM + RAM
- Allot equal address space in odd and even bank for both EPROM and RAM
- Can be implemented in two IC's (one for even and other for odd) or in multiple IC's

- **Memory interface** \Rightarrow **Read from and write in to a set of semiconductor memory IC chip**
- **EPROM** \Rightarrow **Read operations**
- **RAM** \Rightarrow **Read and Write**

In order to perform read/ write operations,

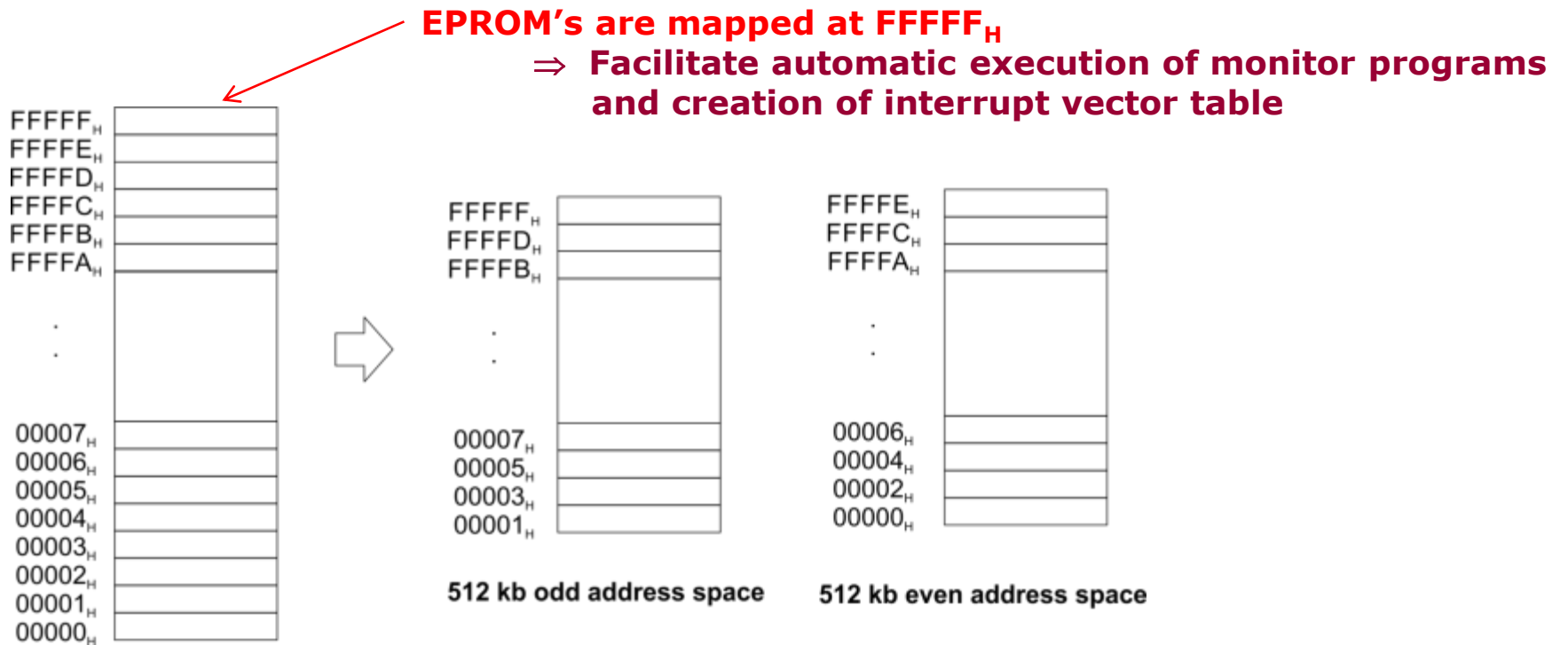
- **Memory access time $<$ read / write time of the processor**
- **Chip Select (CS) signal has to be generated**
- **Control signals for read / write operations**
- **Allot address for each memory location**

■ Typical Semiconductor IC Chip



No of Address pins	Memory capacity			Range of address in hexa
	In Decimal	In kilo	In hexa	
20	$2^{20} = 10,48,576$	1024 k = 1M	100000	00000 to FFFFF

■ Memory map of 8086



RAM are mapped at the beginning; 00000H is allotted to RAM

Monitor Programs

- ⇒ **Programming 8279 for keyboard scanning and display refreshing**
- ⇒ **Programming peripheral IC's 8259, 8257, 8255, 8251, 8254 etc**
- ⇒ **Initialization of stack**
- ⇒ **Display a message on display (output)**
- ⇒ **Initializing interrupt vector table**

Note :	8279	Programmable keyboard/ display controller
	8257	DMA controller
	8259	Programmable interrupt controller
	8255	Programmable peripheral interface

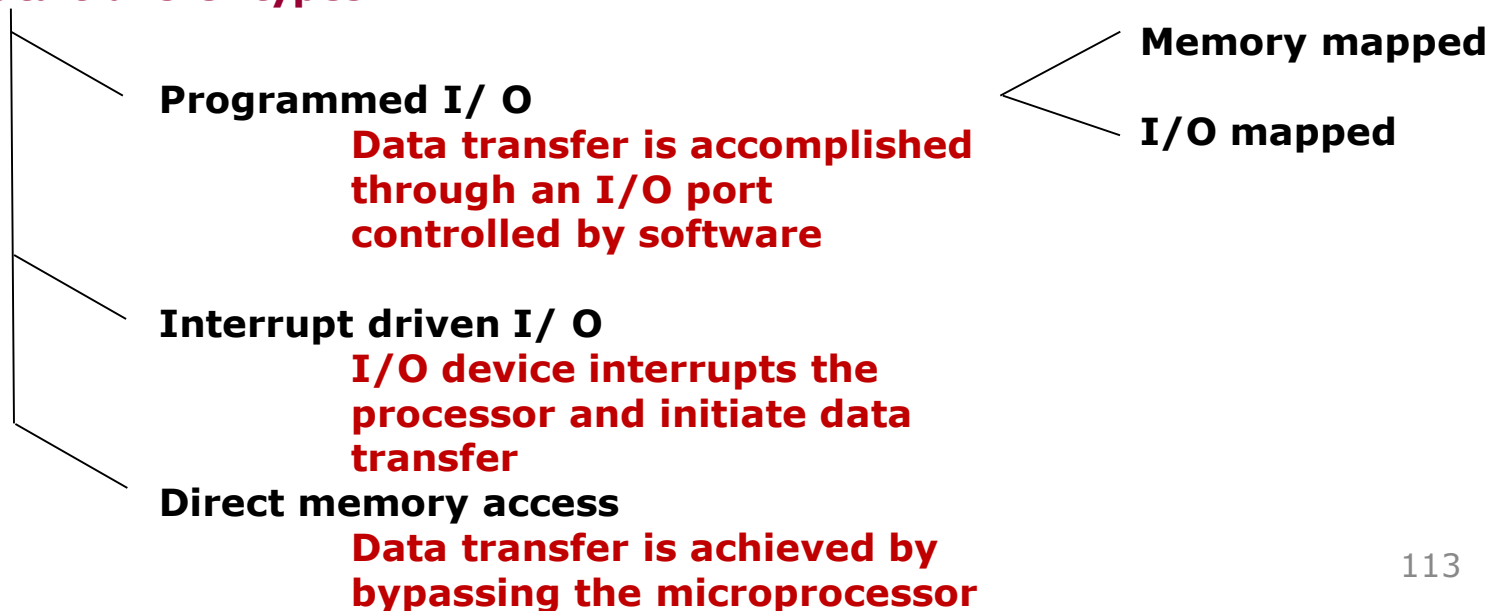
I/O devices

⇒ For communication between microprocessor and outside world

⇒ Keyboards, CRT displays, Printers, Compact Discs etc.



⇒ Data transfer types



Memory mapping	I/O mapping
<p>20 bit address are provided for I/O devices</p>	<p>8-bit or 16-bit addresses are provided for I/O devices</p>
<p>The I/O ports or peripherals can be treated like memory locations and so all instructions related to memory can be used for data transmission between I/O device and processor</p>	<p>Only IN and OUT instructions can be used for data transfer between I/O device and processor</p>
<p>Data can be moved from any register to ports and vice versa</p>	<p>Data transfer takes place only between accumulator and ports</p>
<p>When memory mapping is used for I/O devices, full memory address space cannot be used for addressing memory.</p>	<p>Full memory space can be used for addressing memory.</p> <p>⇒ Suitable for systems which require large memory capacity</p>
<p>⇒ Useful only for small systems where memory requirement is less</p>	
<p>For accessing the memory mapped devices, the processor executes memory read or write cycle.</p>	<p>For accessing the I/O mapped devices, the processor executes I/O read or write cycle.</p>
<p>⇒ M / \overline{IO} is asserted high</p>	<p>⇒ M / \overline{IO} is asserted low</p>

DEPT & SEM : EEE & III/II SEM

SUBJECT NAME: DIGITAL COMPUTE PLATFORMS

COURSE CODE: 19A02601T

UNIT : II

PREPARED BY : Mrs C. MUNIKANTHA

OVER VIEW

ASSEMBLY LANGUAGE PROGRAMMING & I/O INTERFACE

Assembler directives

macros –

simple programs involving logical

branch instructions

sorting

evaluating arithmetic expressions

string manipulations

8255 PPI

various modes of operation

A/D - D/A converter interfacing

Memory interfacing to 8086

interrupt structure of 8086

vector interrupt table

interrupt service routine

interfacing interrupt controller 8259

Need of DMA

serial communication standards

serial data transfer schemes

ASSEMBLER DIRECTIVE:

ASSEMBLER DIRECTIVE:

Assembler directives are non executable instructions which are pseudo instructions which helps assembler to execute a program.

Few Examples for assembler directives are as follows

Org 2000h

Start:

End start

Assume DS: Data

ASSEMBLER DIRECTIVE:

The SEGMENT directive is used to indicate the start of a logical segment. Preceding the SEGMENT directive is the name you want to give the segment. For example, the statement `CODE SEGMENT` indicates to the assembler the start of a logical segment called CODE. The SEGMENT and ENDS directive are used to “bracket” a logical segment containing code or data.

ASSEMBLER DIRECTIVE:

Additional terms are often added to a SEGMENT directive statement to indicate some special way in which we want the assembler to treat the segment. The statement `CODE SEGMENT WORD` tells the assembler that we want the content of this segment located on the next available word (even address) when segments are combined and given absolute addresses.

Without this `WORD` addition, the segment will be located on the next available paragraph (16-byte) address, which might waste as much as 15 bytes of memory. The statement `CODE SEGMENT PUBLIC` tells the assembler that the segment may be put together with other segments named `CODE` from other assembly modules when the modules are linked together.

ASSEMBLER DIRECTIVE:

ENDS (End Segment)

This directive is used with the name of a segment to indicate the end of that logical segment.

CODE SEGMENT	Start of logical segment containing code instruction statements
CODE ENDS	End of segment named CODE

END (End Procedure)

The END directive is put after the last statement of a program to tell the assembler that this is the end of the program module. The assembler will ignore any statements after an END directive, so you should make sure to use only one END directive at the very end of your program module. A carriage return is required after the END directive

ASSEMBLER DIRECTIVE:

ASSUME

The ASSUME directive is used to tell the assembler the name of the logical segment it should use for a specified segment. The statement ASSUME CS: CODE, for example, tells the assembler that the instructions for a program are in a logical segment named CODE.

The statement ASSUME DS: DATA tells the assembler that for any program instruction, which refers to the data segment, it should use the logical segment called DATA.

DB (Define Byte)

The DB directive is used to declare a byte type variable, or to set aside one or more storage locations of type byte in memory.

ASSEMBLER DIRECTIVE:

PRICES DB 49H, 98H, 29H Declare array of 3 bytes named PRICE and initialize them with specified values.

NAMES DB "THOMAS" Declare array of 6 bytes and initialize with ASCII codes for the letters in THOMAS.

RESULT DB 100 DUP (?) Set aside 100 bytes of storage in memory and give it the name RESULT. But leave the 100 bytes un-initialized.

PRESSURE DB 20H DUP (0) Set aside 20H bytes of storage in memory, give it the name PRESSURE and put 0 in all 20H locations.

ASSEMBLER DIRECTIVE:

DD (Define Double Word)

The DD directive is used to declare a variable of type double word or to reserve memory locations, which can be accessed as type double word.

The statement `ARRAY DD 25629261H`, for example, will define a double word named `ARRAY` and initialize the double word with the specified value when the program is loaded into memory to be run. The low word, `9261H`, will be put in memory at a lower address than the high word.

DQ (Define Quadword)

The DQ directive is used to tell the assembler to declare a variable 4 words in length or to reserve 4 words of storage in memory. The statement `BIG_NUMBER DQ 243598740192A92BH`, for example, will declare a variable named `BIG_NUMBER` and initialize the 4 words set aside with the specified number when the program is loaded into memory to be run.

ASSEMBLER DIRECTIVE:

DT (Define Ten Bytes)

The DT directive is used to tell the assembler to declare a variable, which is 10 bytes in length or to reserve 10 bytes of storage in memory. The statement `PACKED_BCD DT 11223344556677889900` will declare an array named `PACKED_BCD`, which is 10 bytes in length. It will initialize the 10 bytes with the values 11, 22, 33, 44, 55, 66, 77, 88, 99, and 00 when the program is loaded into memory to be run. The statement `RESULT DT 20H DUP (0)` will declare an array of 20H blocks of 10 bytes each and initialize all 320 bytes to 00 when the program is loaded into memory to be run.

ASSEMBLER DIRECTIVE:

DW (Define Word)

The DW directive is used to tell the assembler to define a variable of type word or to reserve storage locations of type word in memory. The statement `MULTIPLIER DW 437AH`, for example, declares a variable of type word named `MULTIPLIER`, and initialized with the value `437AH` when the program is loaded into memory to be run.

`WORDS DW 1234H, 3456H` Declare an array of 2 words and initialize them with the specified values.

`STORAGE DW 100 DUP (0)` Reserve an array of 100 words of memory and initialize all 100 words with `0000`. Array is named as `STORAGE`.

`STORAGE DW 100 DUP (?)` Reserve 100 word of storage in memory and give it the name `STORAGE`, but leave the words uninitialized.

EQU (Equate)

EQU is used to give a name to some value or symbol. Each time the assembler finds the given name in the program, it replaces the name with the value or symbol you equated with that name. Suppose, for example, you write the statement

FACTOR EQU 03H at the start of your program, and later in the program you write the instruction statement **ADD AL, FACTOR**. When the assembler codes this instruction statement, it will code it as if you had written the instruction **ADD AL, 03H**.

```
CONTROL EQU 11000110 B MOV AL, CONTROL
```

```
DECIMAL_ADJUST EQU DAA ADD AL, BL DECIMAL_ADJUST
```

Replacement

Assignment

Create clearer mnemonic for DAA Add BCD numbers

Keep result in BCD format

ASSEMBLER DIRECTIVE:

OFFSET

OFFSET is an operator, which tells the assembler to determine the offset or displacement of a named data item (variable), a procedure from the start of the segment, which contains it. When the assembler reads the statement `MOV BX, OFFSET PRICES`, for example, it will determine the offset of the variable `PRICES` from the start of the segment in which `PRICES` is defined and will load this value into `BX`.

ASSEMBLER DIRECTIVE:

PTR (POINTER)

The PTR operator is used to assign a specific type to a variable or a label. It is necessary to do this in any instruction where the type of the operand is not clear. When the assembler reads the instruction `INC [BX]`, for example, it will not know whether to increment the byte pointed to by BX. We use the PTR operator to clarify how we want the assembler to code the instruction. The statement `INC BYTE PTR [BX]` tells the assembler that we want to increment the byte pointed to by BX. The statement `INC WORD PTR [BX]` tells the assembler that we want to increment the word pointed to by BX. The PTR operator assigns the type specified before PTR to the variable specified after PTR.

We can also use the PTR operator to clarify our intentions when we use indirect Jump instructions. The statement `JMP [BX]`, for example, does not tell the assembler whether to code the instruction for a near jump. If we want to do a near jump, we write the instruction as `JMP WORD PTR [BX]`. If we want to do a far jump, we write the instruction as `JMP DWORD PTR [BX]`.

ASSEMBLER DIRECTIVE:

EVEN (Align On Even Memory Address)

As an assembler assembles a section of data declaration or instruction statements, it uses a location counter to keep track of how many bytes it is from the start of a segment at any time. The EVEN directive tells the assembler to increment the location counter to the next even address, if it is not already at an even address. A NOP instruction will be inserted in the location incremented over.

DATA SEGMENT

SALES DB 9 DUP (?) Location counter will point to 0009 after this instruction.

EVEN Increment location counter to 000AH

INVENTORY DW 100 DUP (0) Array of 100 words starting on even address for
quicker read DATA ENDS

PROC (Procedure)

The PROC directive is used to identify the start of a procedure. The PROC directive follows a name you give the procedure. After the PROC directive, the term *near* or the term *far* is used to specify the type of the procedure. The statement DIVIDE PROC FAR, for example, identifies the start of a procedure named DIVIDE and tells the assembler that the procedure is far (in a segment with different name from the one that contains the instructions which calls the procedure). The PROC directive is used with the ENDP directive to “bracket”

ENDP (End Procedure)

The directive is used along with the name of the procedure to indicate the end of a procedure to the assembler. The directive, together with the procedure directive, PROC, is used to “bracket” a procedure

Procedure

SQUARE_ROOT PROC Start of procedure.

SQUARE_ROOT ENDP End of procedure.

Macros

A **Macro** is a set of instructions grouped under a single unit

The **Macro** is different from the Procedure in a way that unlike calling and returning the control as in procedures, the processor generates the code in the program every time whenever and wherever a call to the **Macro** is made.

A **Macro** can be defined in a program using the following assembler directives: **MACRO** and **ENDM**.

All the instructions that belong to the Macro lie within these two assembler directives. The following is the syntax for defining a **Macro in the 8086 Microprocessor**

Macro_name MACRO [list of parameters]

Instruction 1

Instruction 2

Instruction n

ENDM

Macros

A **Macro** is a set of instructions grouped under a single unit

The **Macro** is different from the Procedure in a way that unlike calling and returning the control as in procedures, the processor generates the code in the program every time whenever and wherever a call to the **Macro** is made.

A **Macro** can be defined in a program using the following assembler directives: **MACRO** and **ENDM**.

. All the instructions that belong to the Macro lie within these two assembler directives. The following is the syntax for defining a **Macro in the 8086 Microprocessor**

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

Largest Number

MOV SI, 5000

MOV AL, [SI]

MOV CL, [SI]

SVEC: INC SI

MOV CH, 00

LOOP SVEW

INC SI

MOV [6000], AL

MOV AL, [SI]

HLT

DEC CL

INC SI

SVEW : CMP AL,[SI]

JNC SVEC

ASCENDING ORDER

MOV SI, 5000H
MOV CL, [SI]
DEC CL
EEE: MOV SI, 5000H
MOV CH, [SI]
DEC CH
INC SI
VEMU: MOV AL, [SI]
INC SI
CMP AL, [SI]
JC SVEW

XCHG AL, [SI]
DEC SI
XCHG AL, [SI]
INC SI
SVEW: DEC CH
JNZ VEMU
DEC CL
JNZ EEE
HLT

Smallest Number

MOV SI, 5000

MOV AL, [SI]

MOV CL, [SI]

SVEC: INC SI

MOV CH, 00

LOOP SVEW

INC SI

MOV [6000], AL

MOV AL, [SI]

HLT

DEC CL

INC SI

SVEW : CMP AL,[SI]

JC SVEC

DESCENDING ORDER

MOV SI, 5000H
MOV CL, [SI]
DEC CL
EEE: MOV SI, 5000H
MOV CH, [SI]
DEC CH
INC SI
SVEC: MOV AL, [SI]
INC SI
CMP AL, [SI]
JNC SVEW

XCHG AL, [SI]
DEC SI
XCHG AL, [SI]
INC SI
SVEW: DEC CH
JNZ SVEC
DEC CL
JNZ EEE
HLT

Factorial

MOV CX, [5000]

MOV AX, 0001

MOV DX, 0000

SVEW: MUL CX

LOOP SVEW

MOV [6000], AX

MOV [6002], DX

HLT

Fibonacci sequence

MOV AL, 00H

MOV SI, 5000H

MOV [SI], AL

ADD SI, 01H

ADD AL, 01H

MOV [SI], AL

SUB CX, 0002H

L1: MOV AL, [SI-1]

ADD AL, [SI]

ADD SI, 01H

MOV [SI], AL

LOOP L1

HLT

Fibonacci sequence

MOV AL, 00H

MOV SI, 5000H

MOV [SI], AL

ADD SI, 01H

ADD AL, 01H

MOV [SI], AL

SUB CX, 0002H

L1: MOV AL, [SI-1]

ADD AL, [SI]

ADD SI, 01H

MOV [SI], AL

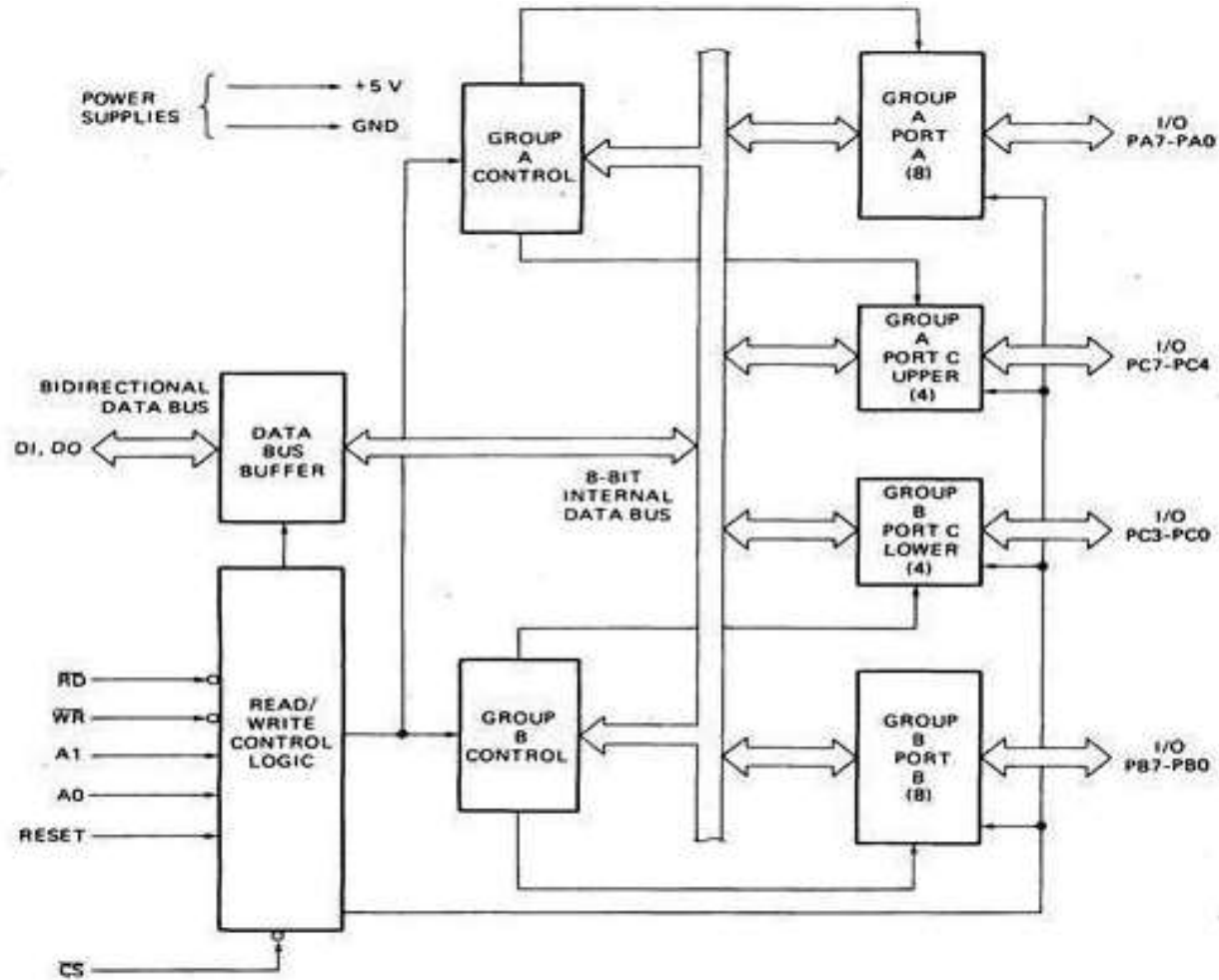
LOOP L1

HLT

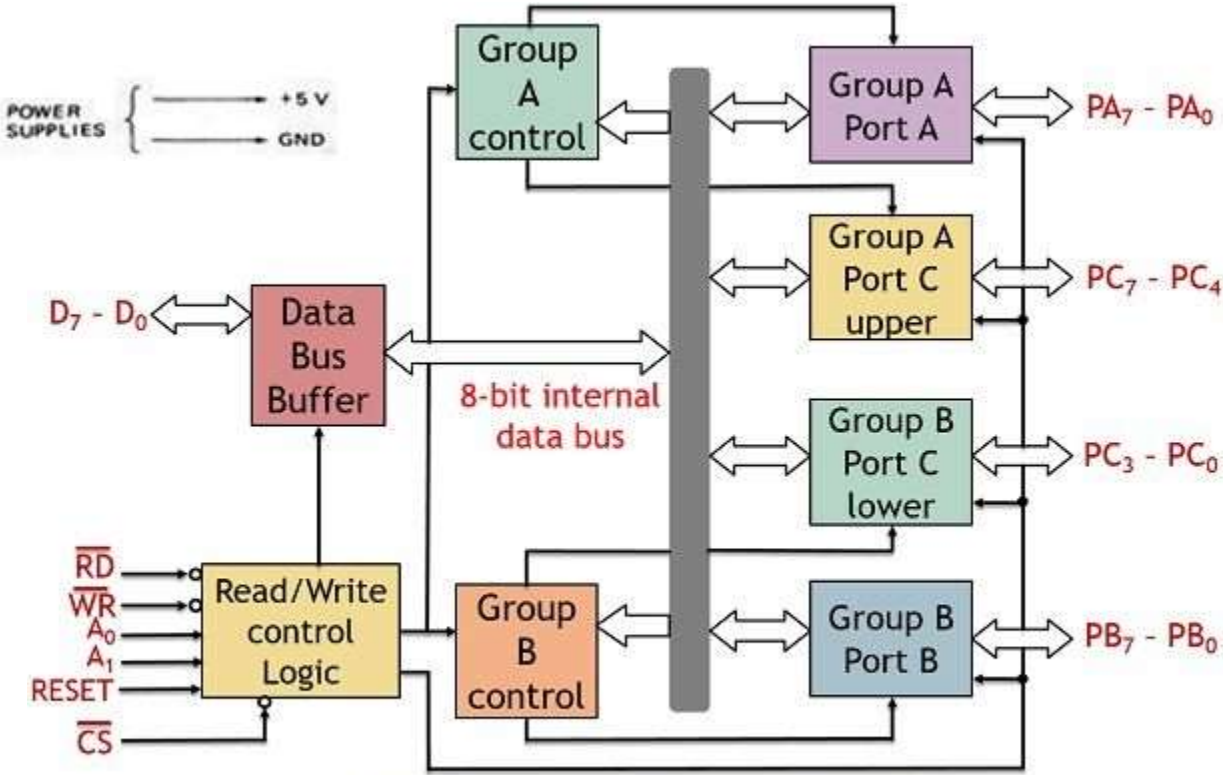
STRING MANIPULATION – MOVE

	MOV CL,05H
	MOV SI,1100H
	MOV DI,1200H
	CLD
L1	MOVSB
	LOOP L1
	HLT

8255 PPI



8255 PPI

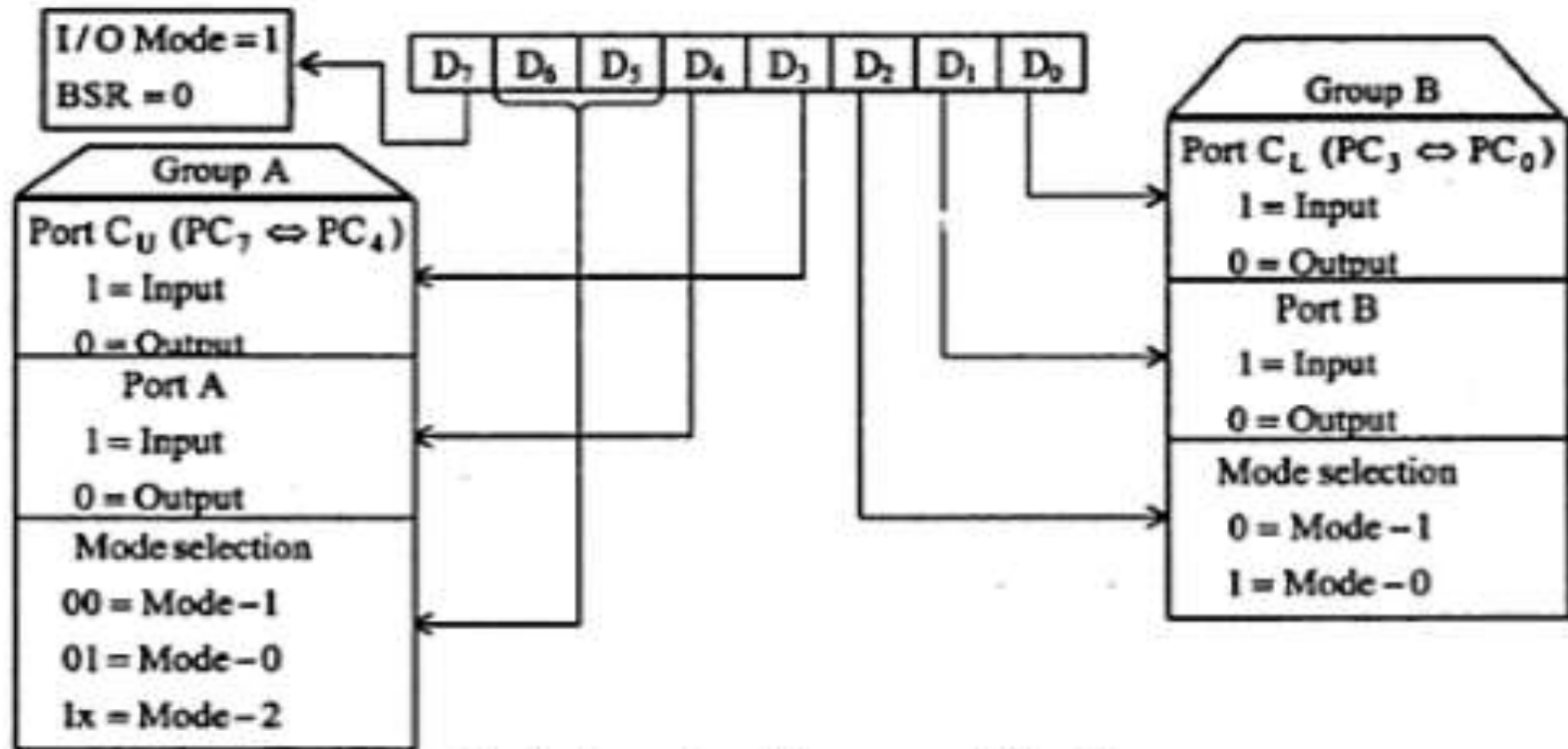


Architecture of 8255 PPI

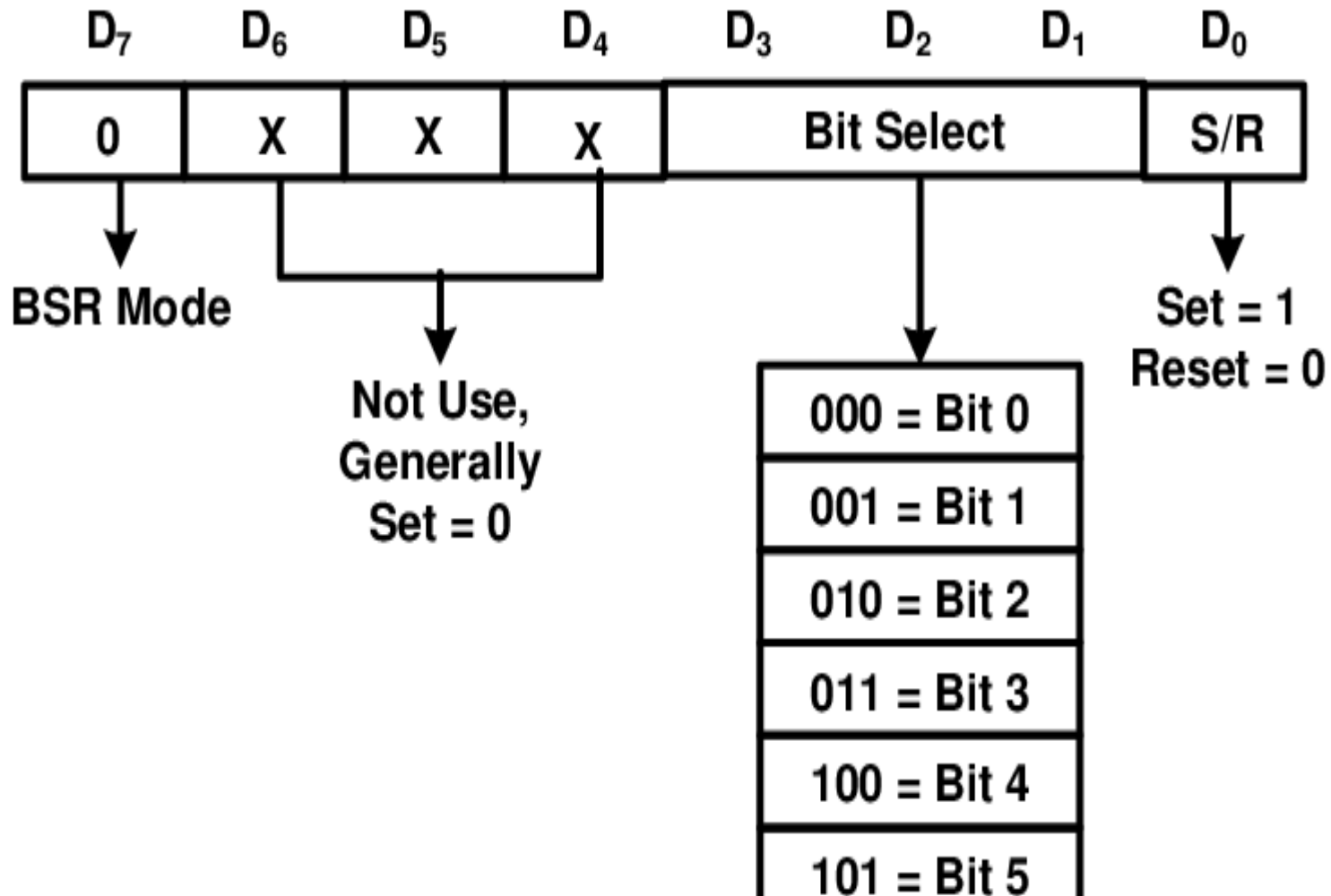
Register selection

S'	A1	A0	Selection	Address
0	0	0	PORT A	80 H
0	0	1	PORT B	81 H
0	1	0	PORT C	82 H
0	1	1	Control Register	83 H
1	X	X	No Selection	X

PPI 8255 CONTROL WORD

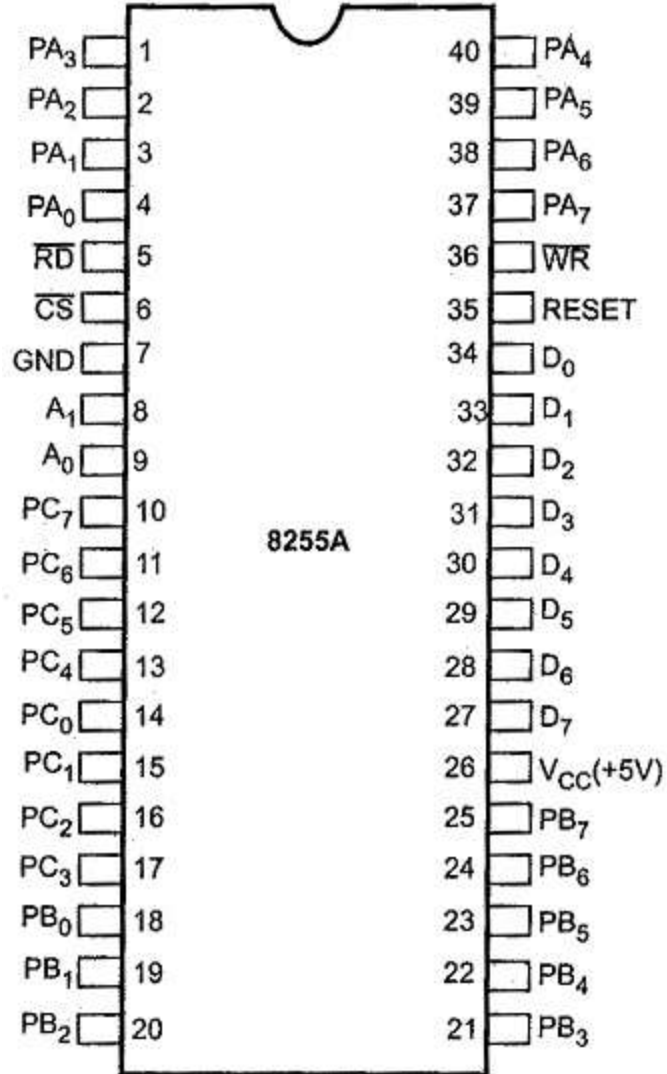


Control word with group definition



CWR FORMATE

8255 PPI PIN DIAGRAM



8255 FEATURES

- The port A lines are identified by symbols PA0-PA7 while the port C lines are identified as PC4-PC7.
- Similarly, Group B contains an 8-bit port B, containing lines PB0-PB7 and a 4-bit port C with lower bits PC0- PC3.
- The port C upper and port C lower can be used in combination as an 8-bit port C.
- Both the port C is assigned the same address.
- Thus one may have either three 8-bit I/O ports or two 8-bit and two 4-bit ports from 8255.
- All of these ports can function independently either as input or as output ports.
- This can be achieved by programming the bits of an internal register of 8255 called as control word register (CWR).

Modes of 8255

Modes of Operation of 8255

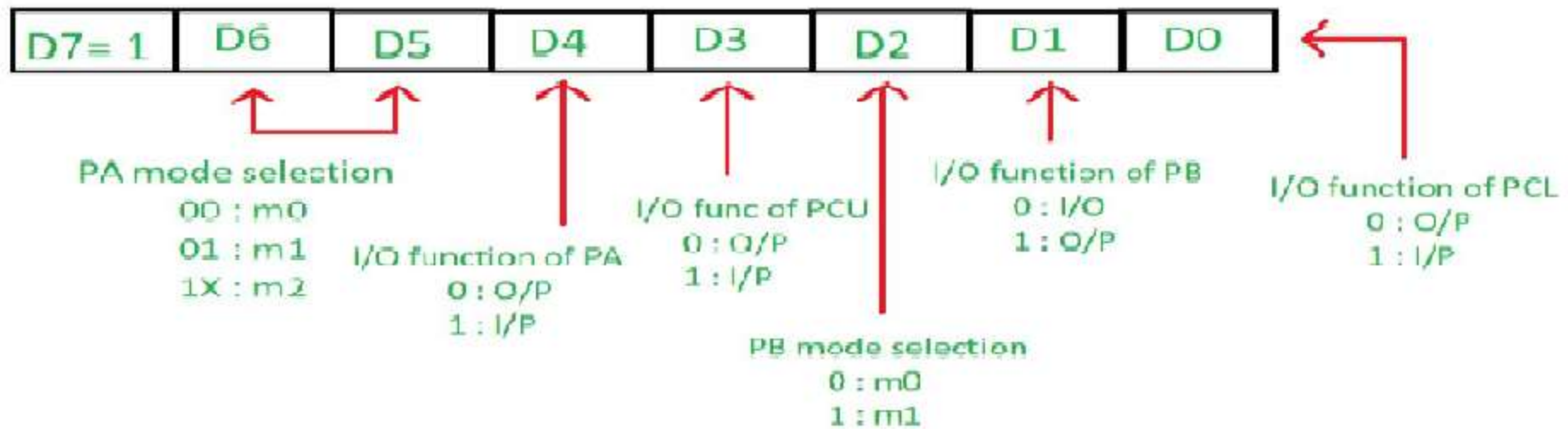
These are two basic modes of operation of 8255.

I/O mode and Bit Set-Reset mode (BSR).

In **I/O mode**, the 8255 ports work as programmable I/O ports, while in BSR mode only port C (PC0-PC7) can be used to set or reset its individual port bits.

Under the I/O mode of operation, further there are three modes of operation of 8255, so as to support different types of applications, mode 0, mode 1 and mode 2.

BSR Mode: In this mode any of the 8-bits of port C can be set or reset depending on D0 of the control word. The bit to be set or reset is selected by bit select flags D3, D2 and D1 of the CWR as given in table.

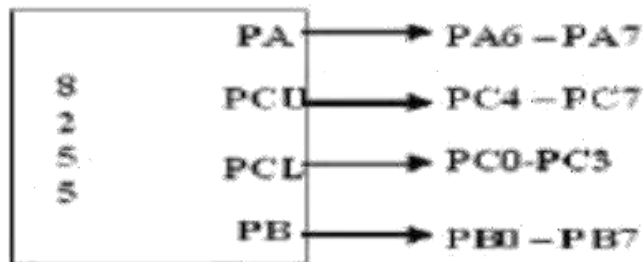


I/O Modes:

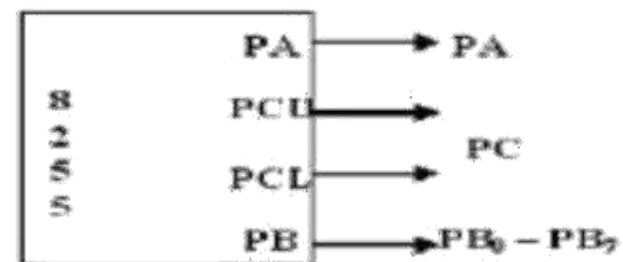
Mode 0 (Basic I/O mode): This mode is also called as basic input/output Mode. This mode provides simple input and output capabilities using each of the three ports. Data can be simply read from and written to the input and output ports respectively, after appropriate initialization.

D ₃	D ₂	D ₁	Selected bits of port C
0	0	0	D ₀
0	0	1	D ₁
0	1	0	D ₂
0	1	1	D ₃
1	0	0	D ₄
1	0	1	D ₅
1	1	0	D ₆
1	1	1	D ₇

BSR Mode : CWR Format



All Output



Port A and Port C acting as O/P. Port B acting as I/P

Mode 1

Mode 1: (Strobed input/output mode) In this mode the handshaking control the input and output action of the specified port. Port C lines PC0-PC2, provide strobe or handshake lines for port B.

This group which includes port B and PC0-PC2 is called as group B for Strobed data input/output. Port C lines PC3-PC5 provides strobe lines for port A.

This group including port A and PC3-PC5 from group A. Thus port C is utilized for generating handshake signals.

The salient features of mode 1 are listed as follows

- Two groups – group A and group B are available for strobed data transfer.
- Each group contains one 8-bit data I/O port and one 4-bit control/data port.
- The 8-bit data port can be either used as input and output port. The inputs and outputs both are latched.

Out of 8-bit port C, PC0-PC2 are used to generate control signals for port B and PC3-PC5 are used to generate control signals for port A. the lines PC6, PC7 may be used as independent data lines.

Mode 2 (Strobed bidirectional I/O): This mode of operation of 8255 is also called as strobed bidirectional I/O.

- This mode of operation provides 8255 with additional features for communicating with a peripheral device on an 8-bit data bus.
- Handshaking signals are provided to maintain proper data flow and synchronization between the data transmitter and receiver.
- The interrupt generation and other functions are similar to mode 1.
- In this mode, 8255 is a bidirectional 8-bit port with handshake signals. The Rd and WR signals decide whether the 8255 is going to operate as an input port or output port.
- The Salient features of Mode 2 of 8255 are listed as follows:

The single 8-bit port in group A is available.

The 8-bit port is bidirectional and additionally a 5-bit control port is available.

Three I/O lines are available at port C. (PC2 – PC0)

Inputs and outputs are both latched.

Interfacing ADC Port A

Interfacing ADC Port A acts as a 8-bit input data port to receive the digital data output from the ADC.

The 8255 control word is written as follows:

D7 D6 D5 D4 D3 D2 D1 D0

1 0 0 1 1 0 0 0

Interfacing ADC using ALP

Interfacing ADC The required ALP is as follows:

```
MOV AL, 98h ; initializes 8255.
```

```
OUT CWR, AL ;
```

```
MOV AL, 02h ;Select I/P2 as analog input
```

```
OUT Port B, AL ;
```

```
MOV AL, 00h ;Give start of conversion
```

```
OUT Port C, AL ; pulse to the ADC
```

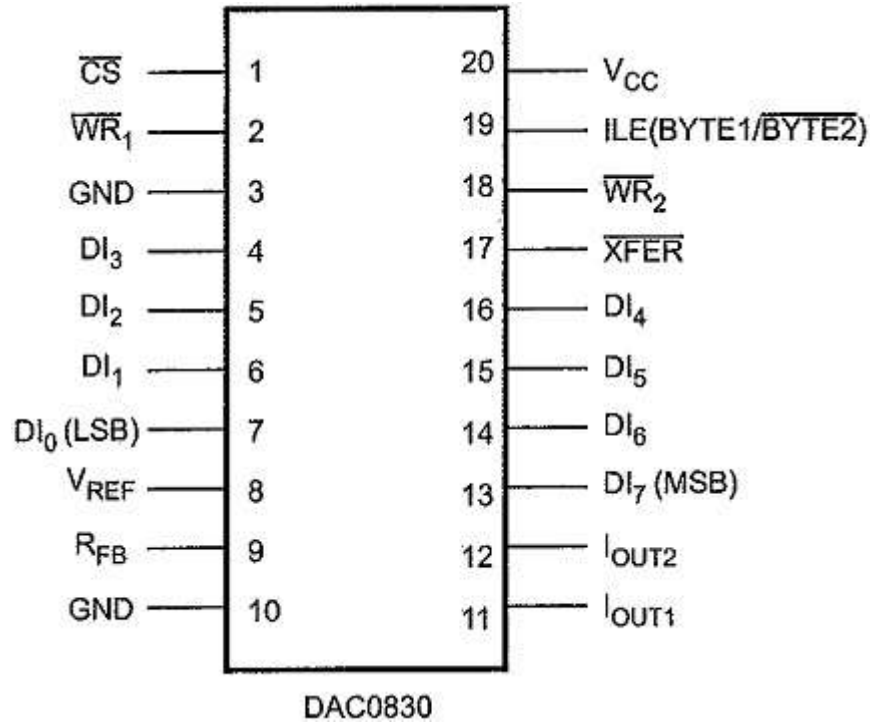
```
MOV AL, 01h
```

```
OUT Port C, AL
```

```
MOV AL, 00h
```

```
OUT Port C, AL
```

DAC INTERFACING WITH 8086



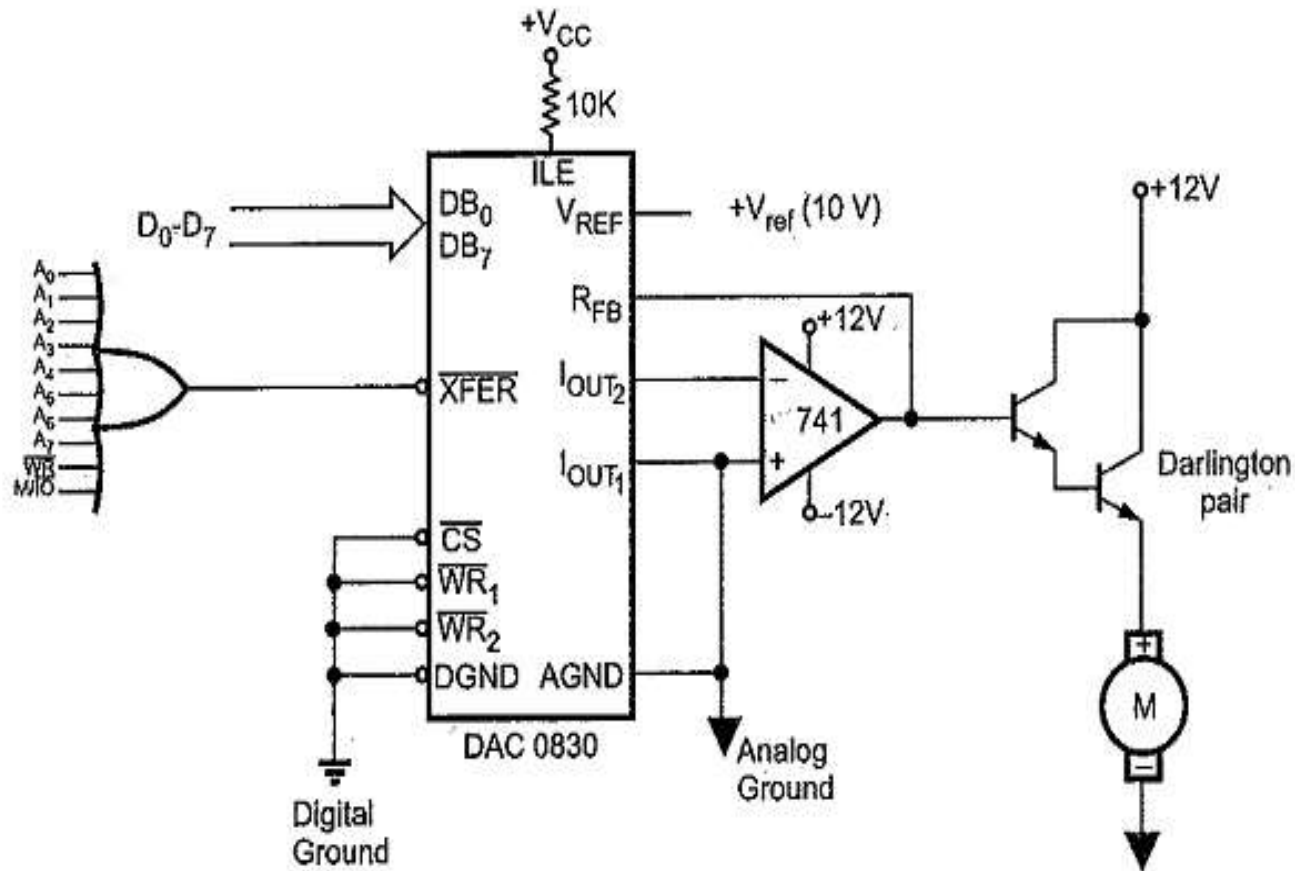
DAC INTERFACING WITH 8086

Control Signals (All control signals level actuated)	
\overline{CS} :	Chip Select (active low). The \overline{CS} in combination with ILE will enable \overline{WR}_1 .
ILE:	Input Latch Enable (active high). The ILE in combination with \overline{CS} enables \overline{WR}_1 .
\overline{WR}_1 :	Write 1. The active low \overline{WR}_1 is used to load the digital input data bits (DI) into the input latch. The data in the input latch is latched when \overline{WR}_1 is high. To update the input latch- \overline{CS} and \overline{WR}_1 must be low while ILE is high.
\overline{WR}_2 :	Write 2 (active low). This signal, in combination with \overline{XFER} , causes the 8-bit data which is available in the input latch to transfer to the DAC register.
\overline{XFER} :	Transfer control signal (active low). The \overline{XFER} will enable \overline{WR}_2 .

DAC INTERFACING WITH 8086

Other Pin Functions	
DI_0 - DI_7 :	Digital Inputs. DI_0 is the least significant bit (LSB) and DI_7 is the most significant bit (MSB).
I_{OUT1} :	DAC Current Output 1. I_{OUT1} is a maximum for a digital code of all 1's in the DAC register, and is zero for all 0's in DAC register.
I_{OUT2} :	DAC Current Output 2. $I_{OUT1} + I_{OUT2} = \text{constant}$ (I full scale for a fixed reference voltage).
R_{fb} :	Feedback Resistor. The feedback resistor is provided on the IC chip for use as the shunt feedback resistor for the external opamp which is used to provide an output voltage for the DAC. This on-chip resistor should always be used (not an external resistor) since it matches the resistors which are used in the on-chip R-2R ladder and tracks these resistors over temperature.
V_{REF} :	Reference Voltage Input. This input connects an external precision voltage source to the internal R-2R ladder. V_{REF} can be selected over the range of +10V to -10V. This is also the analog voltage input for a 4-quadrant multiplying DAC application.
V_{CC} :	Digital Supply Voltage. This is the power supply pin for the part. V_{CC} can be from +5V _{DC} to +15V _{DC} . Operation is optimum for +15V _{DC} .
GND:	The pin 10 voltage must be at the same ground potential as I_{OUT1} and I_{OUT2} for current switching applications. Any difference of potential (V_{OS} pin 10) will result in a linearity change of $\frac{V_{OS \text{ pin 10}}}{3 V_{REF}}$. For example, if $V_{REF} = 10V$ and pin 10 is 9mV offset from I_{OUT1} and I_{OUT2} , the linearity change will be 0.03%. Pin 3 can be offset $\pm 100mV$ with no linearity change, but the logic input threshold will shift.

DAC INTERFACING WITH 8086



A/D and D/A converter

SAWTOOTH WAVEFORM

```
LABEL  MNEMONICS
        MOV AL,80H
        OUT 76,AL
START:  MOV AL,0H
        OUT 70H,AL
        OUT 72,AL
        INC AL
        JMP START
```

DAC

```
LABEL  MNEMONICS
      MOV AL,80
      OUT 76,AL
START  MOV AL,00
REPEAT OUT 70,A
      OUT 72,AL
      INC AL
      CMP AL,FF
      JNZ REPEAT
      MOV AL,FF
AGAIN  OUT 70,AL
      OUT 72,AL
      DEC AL
      CMP AL,00
      JNZ AGAIN
      JMP START
```

Memory interfacing to 8086

Memory interface

Memory is divided into two banks ODD and EVEN.

The data bus is 16-bits wide.

The IO/ M pin is replaced with M/ IO (8086).

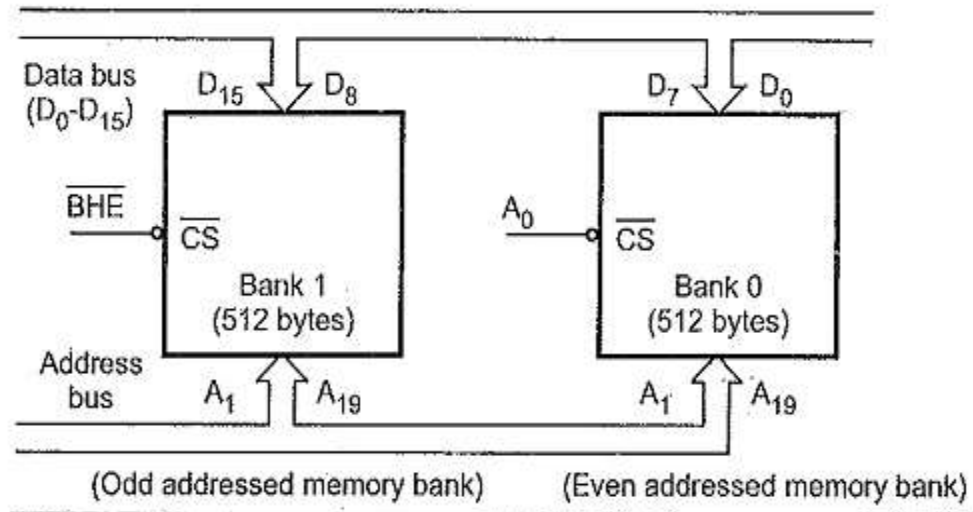
BHE , Bus High Enable, control signal is added.

Address pin A 0 (or BLE , Bus Low Enable) is used differently.

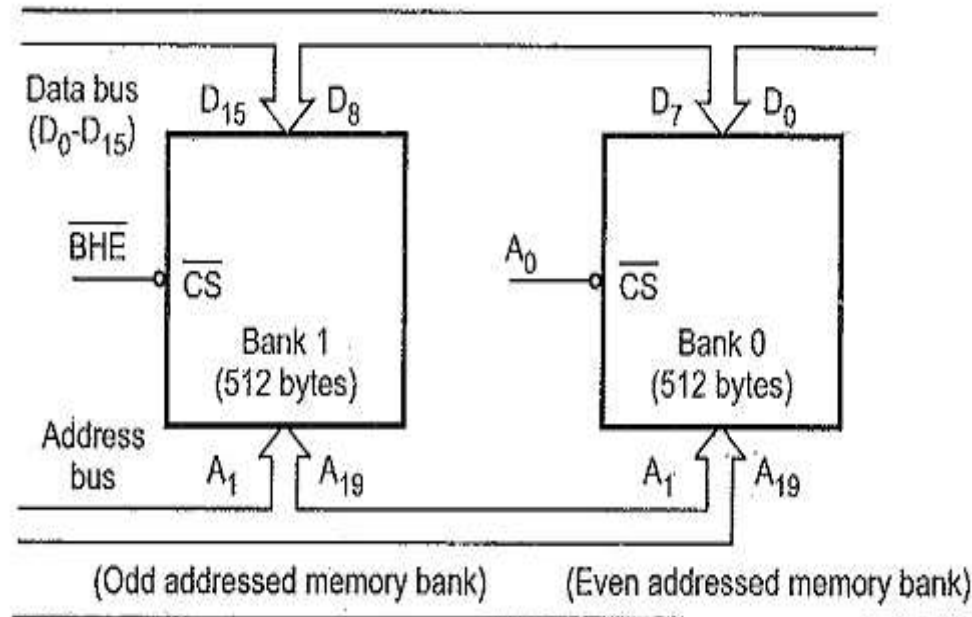
The 16-bit data bus presents a new problem:

The microprocessor must be able to read and write data to any 16-bit location in addition to any 8-bit location.

The data bus and memory are divided into banks:



Memory interfacing



Memory interface

BHE and BLE are used to select one or both:

BHE BLE(A0) Function

0 0 Both banks enabled for 16-bit transfer

0 1 High bank enabled for an 8-bit transfer

1 0 Low bank enabled for an 8-bit transfer

1 1 No banks selected Bank selection can be accomplished in two ways:

Separate write decoders for each bank (which drive CS).

A separate write signal (strobe) to each bank (which drive WE).

Note that 8-bit read requests in this scheme are handled by the microprocessor (it selects the bits it wants to read from the 16-bits on the bus).

Memory interfacing

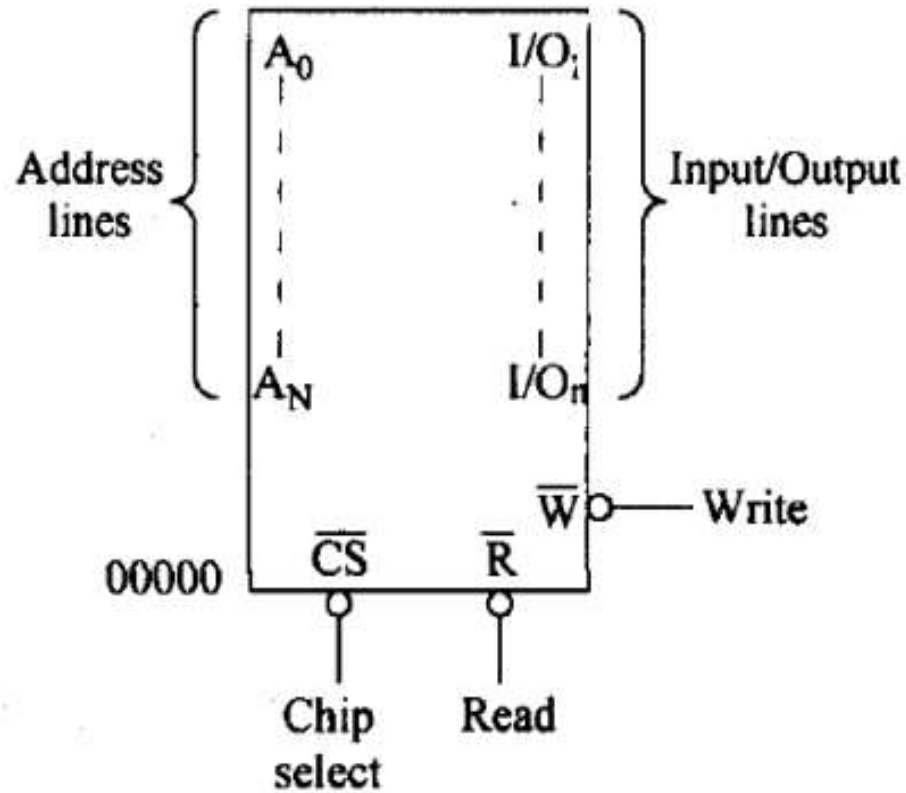
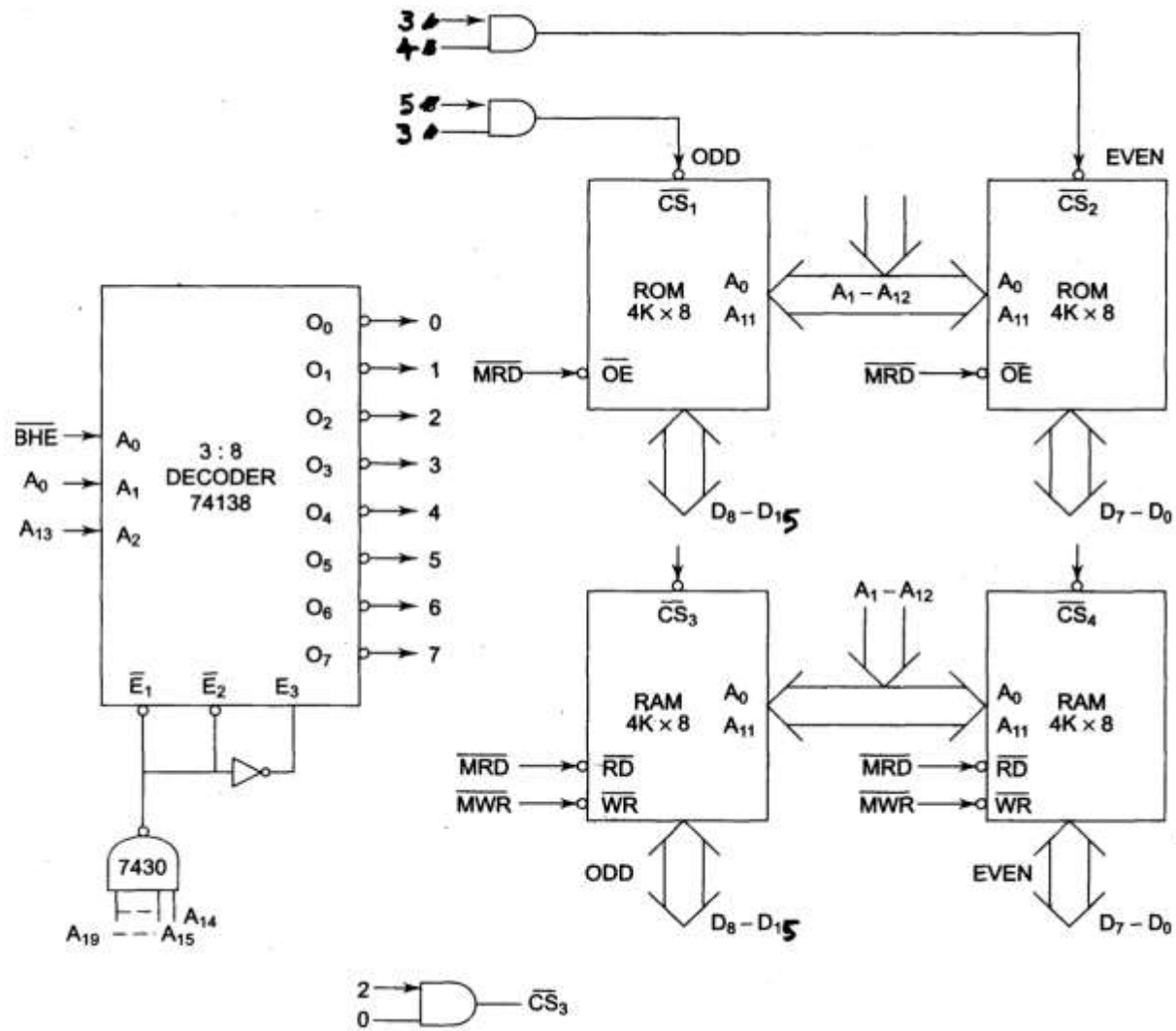


Fig . Schematic diagram of a memory



Memory interfacing

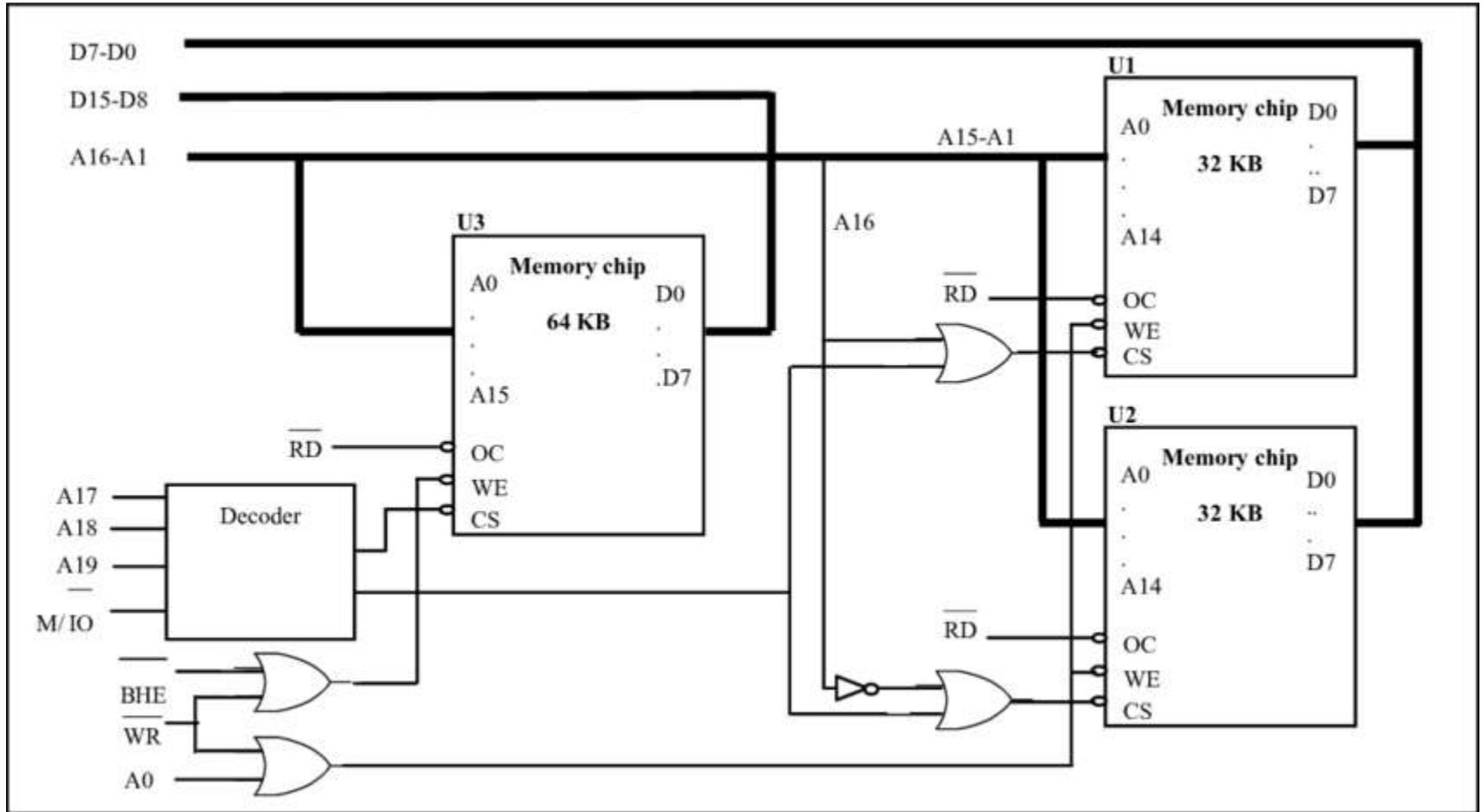
Address	A_{19}	A_{18}	A_{17}	A_{16}	A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_{09}	A_{08}	A_{07}	A_{06}	A_{05}	A_{04}	A_{03}	A_{02}	A_{01}	A_{00}
FFFFH	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
EPROM								8K × 8												
FE00H	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0
FDFFFH	1	1	1	1	1	1	0	1	1	1	1	1	1	1	1	1	1	1	1	1
RAM								8K × 8												
FC000H	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fig. Memory map

Decoder I/P →	A_2	A_1	A_0	Selection/Comment
Address/BHE →	A_{13}	A_0	\overline{BHE}	
Word transfer on $D_0 - D_{15}$	0	0	0	Even and odd addresses in RAM
Byte transfer on $D_7 - D_0$	0	0	1	Only even address in RAM
Byte transfer on $D_8 - D_{15}$	0	1	0	Only odd address in RAM
Word transfer on $D_0 - D_{15}$	1	0	0	Even and odd addresses in ROM
Byte transfer on $D_0 - D_7$	1	0	1	Only even address in ROM
Byte transfer on $D_8 - D_{15}$	1	1	0	Only odd address in ROM

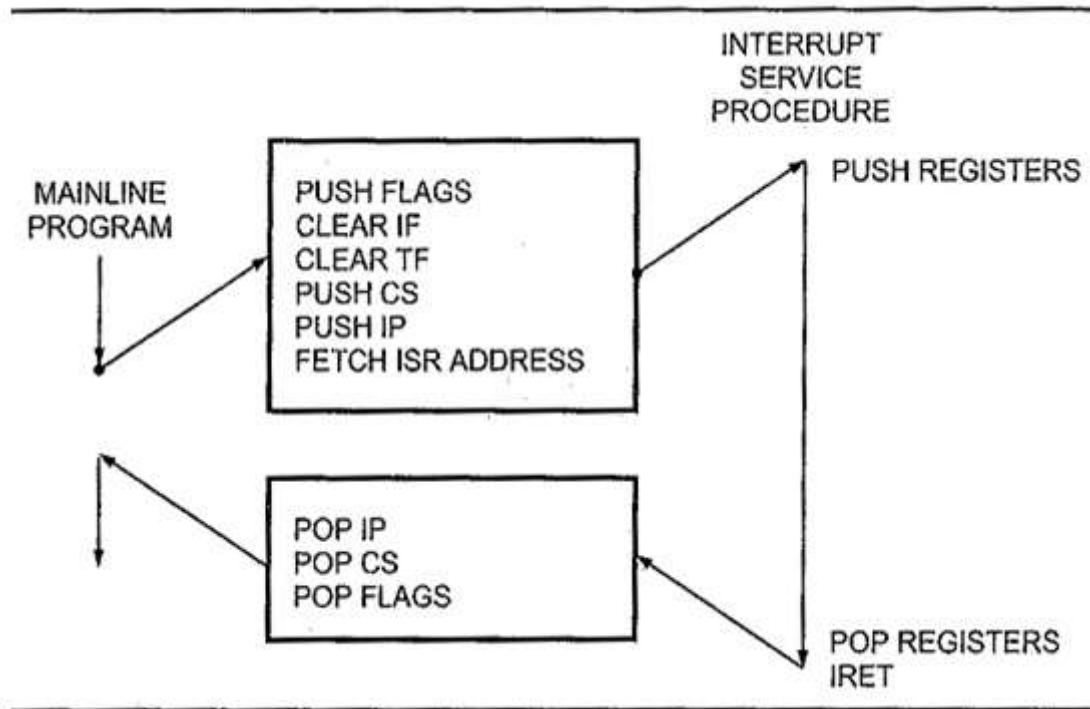
Fig. Memory chip selection

Memory interfacing with 8086



Interrupt structure of 8086

8086 Interrupt response



Interrupt structure of 8086

Interrupt is the method of creating a temporary halt during program execution and allows peripheral devices to access the microprocessor. The microprocessor responds to that interrupt with an ISR (Interrupt Service Routine), which is a short program to instruct the microprocessor on how to handle the interrupt.

There are two types of interrupts in a 8086 microprocessor.

They are hardware interrupts and software interrupts.

NMI

It is a single non-maskable interrupt pin (NMI) having higher priority than the maskable interrupt request pin (INTR) and it is of type 2 interrupt.

When this interrupt is activated, these following actions take place

- Completes the current instruction that is in progress.**
- Pushes the Flag register values on to the stack.**
- Pushes the CS (code segment) value and IP (instruction pointer) value of the return address on to the stack.**
- IP is loaded from the contents of the word location 00008H.**
- CS is loaded from the contents of the next word location 0000AH.**
- Interrupt flag and trap flag are reset to 0.**

Interrupt structure of 8086

INTR

The INTR is a maskable interrupt because the microprocessor will be interrupted only if interrupts are enabled using set interrupt flag instruction. It should not be enabled using clear interrupt Flag instruction.

The INTR interrupt is activated by an I/O port. If the interrupt is enabled and NMI is disabled, then the microprocessor first completes the current execution and sends '0' on INTA pin twice. The first '0' means INTA informs the external device to get ready and during the second '0' the microprocessor receives the 8 bit, say X, from the programmable interrupt controller.

These actions are taken by the microprocessor

- First completes the current instruction.
 - Activates INTA output and receives the interrupt type, say X.
 - Flag register value, CS value of the return address and IP value of the return address are pushed on to the stack.
 - IP value is loaded from the contents of word location $X \times 4$
 - CS is loaded from the contents of the next word location.
- Interrupt flag and trap flag is reset to 0

Software Interrupts

Some instructions are inserted at the desired position into the program to create interrupts. These interrupt instructions can be used to test the working of various interrupt handlers.

It includes

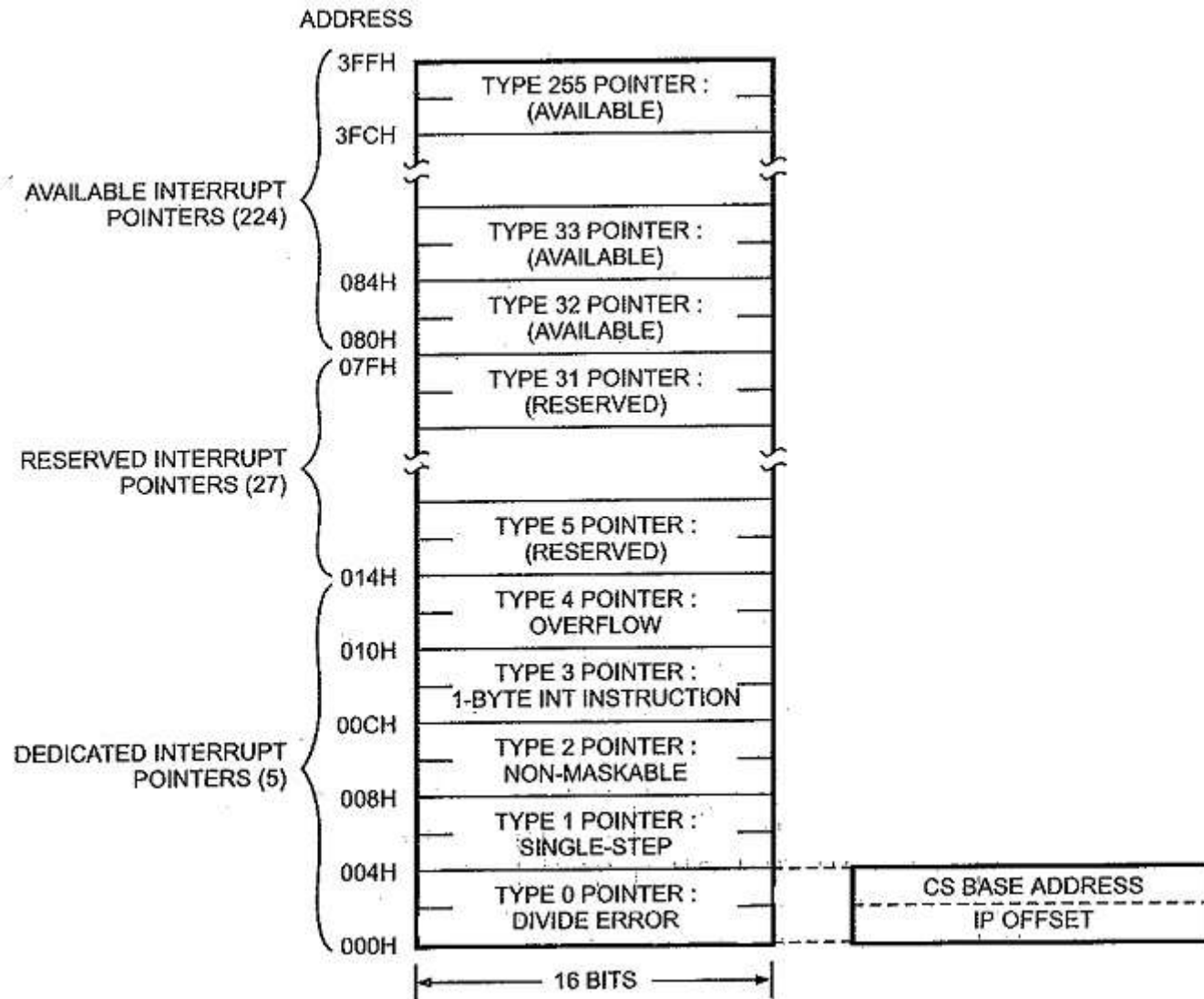
INT- Interrupt instruction with type number like int 06

It is 2-byte instruction. First byte provides the op-code and the second byte provides the interrupt type number. There are 256 interrupt types under this group.

Its execution includes the following steps

- Flag register value is pushed on to the stack.**
- CS value of the return address and IP value of the return address are pushed on to the stack.**
- IP is loaded from the contents of the word location 'type number' \times 4**
- CS is loaded from the contents of the next word location.**
- Interrupt Flag and Trap Flag are reset to 0**

Interrupt types of 8086



Interrupts of 8086

The starting address for type0 interrupt is 00000H, for type1 interrupt is 00004H similarly for type2 is 00008H andso on. The first five pointers are dedicated interrupt pointers. i.e.

- TYPE 0 interrupt represents division by zero situation.
- TYPE 1 interrupt represents single-step execution during the debugging of a program.
- TYPE 2 interrupt represents non-maskable NMI interrupt.
- TYPE 3 interrupt represents break-point interrupt.
- TYPE 4 interrupt represents overflow interrupt.

The interrupts from Type 5 to Type 31 are reserved for other advanced microprocessors, and interrupts from 32 to Type 255 are available for hardware and software interrupts.

Interrupts of 8086

INT 3-Break Point Interrupt Instruction

It is a 1-byte instruction having op-code is CCH. These instructions are inserted into the program so that when the processor reaches there, then it stops the normal execution of program and follows the break-point procedure.

Its execution includes the following steps

- Flag register value is pushed on to the stack.
- CS value of the return address and IP value of the return address are pushed on to the stack.
- IP is loaded from the contents of the word location $3 \times 4 = 0000CH$
- CS is loaded from the contents of the next word location.
- Interrupt Flag and Trap Flag are reset to 0

INTO - Interrupt on overflow instruction

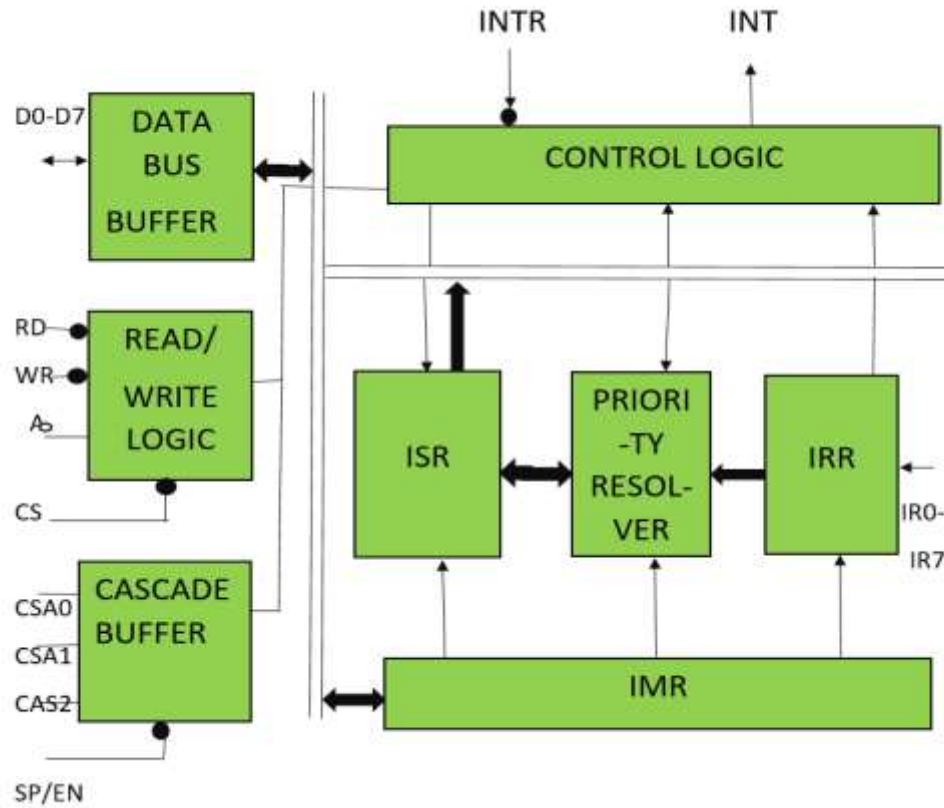
It is a 1-byte instruction and their mnemonic INTO. The op-code for this instruction is CEH. As the name suggests it is a conditional interrupt instruction, i.e. it is active only when the overflow flag is set to 1 and branches to the interrupt handler whose interrupt type number is 4. If the overflow flag is reset then, the execution continues to the next instruction.

Interrupts of 8086

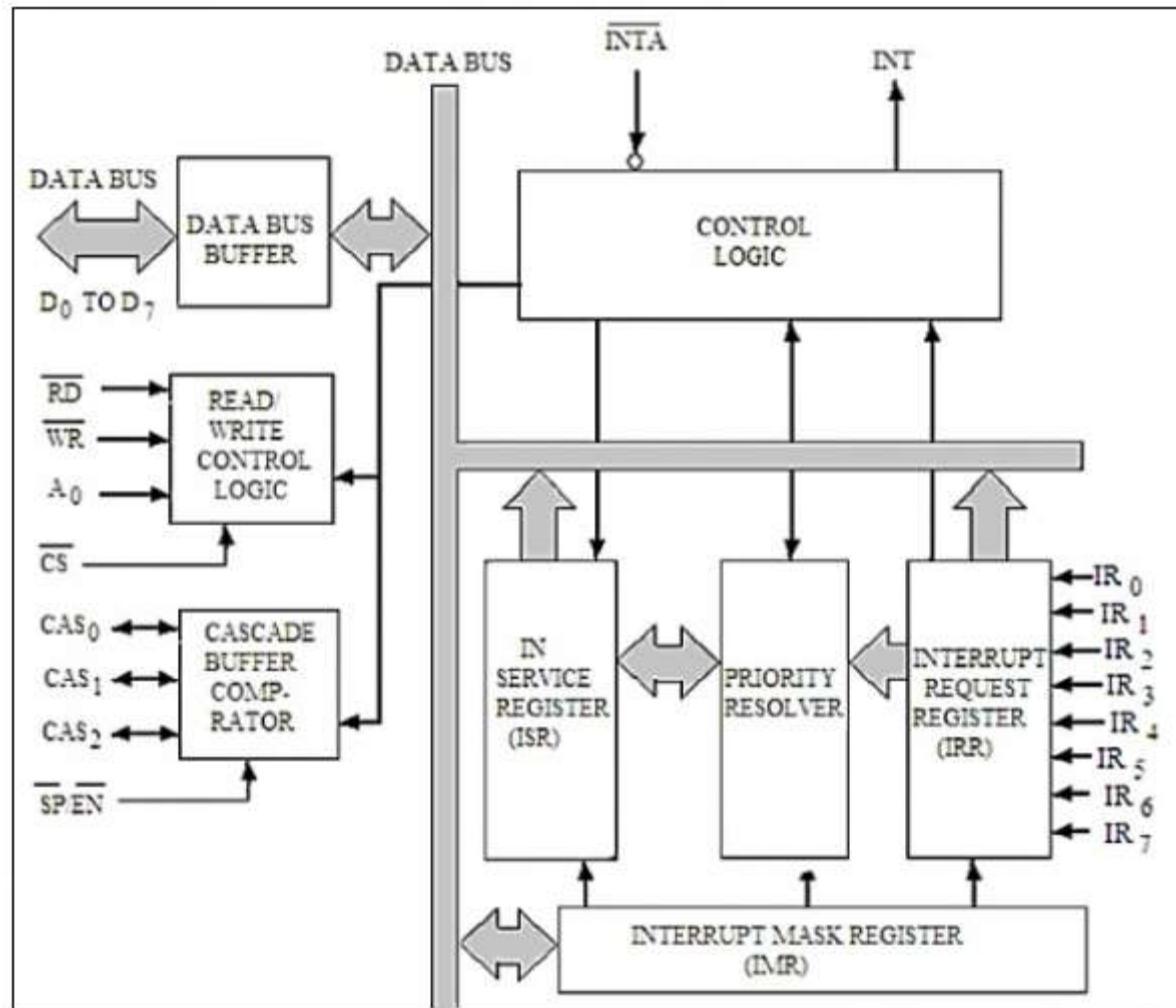
Its execution includes the following steps

- Flag register values are pushed on to the stack.
- CS value of the return address and IP value of the return address are pushed on to the stack.
- IP is loaded from the contents of word location $4 \times 4 = 00010H$
- CS is loaded from the contents of the next word location.
- Interrupt flag and Trap flag are reset to 0

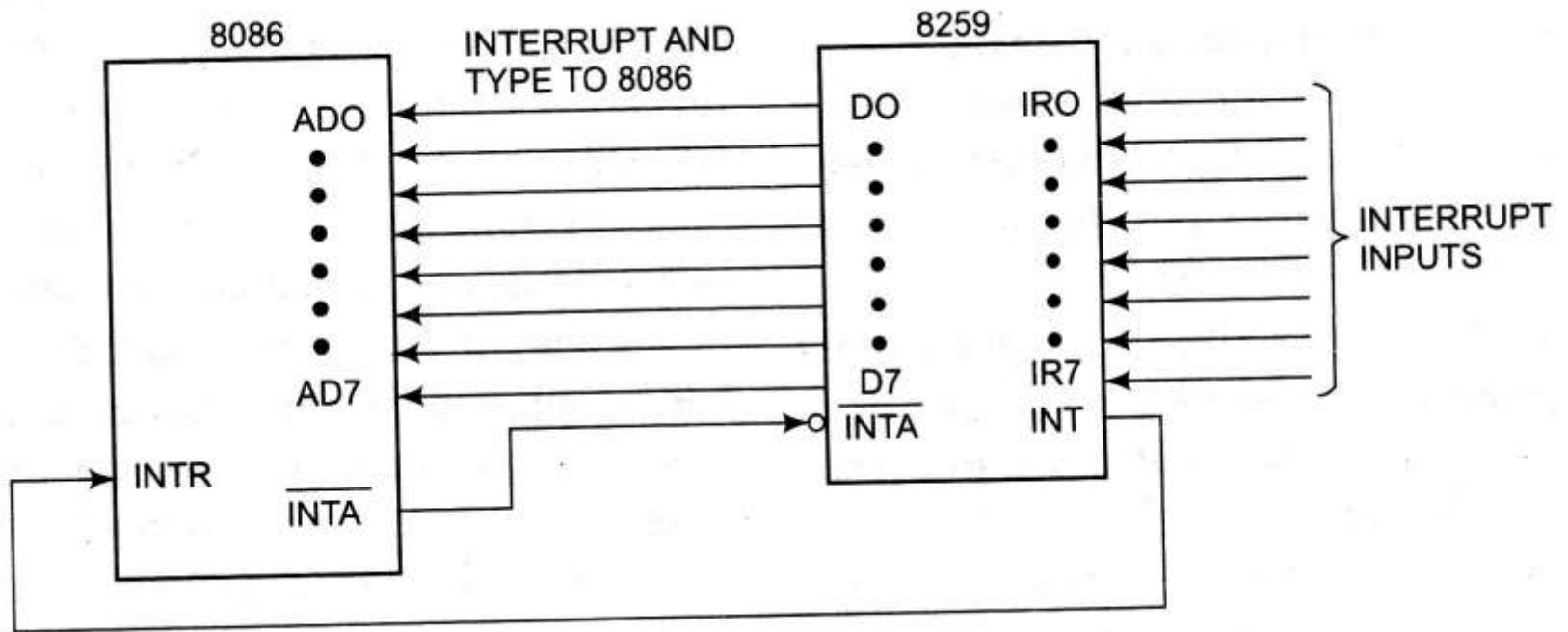
8259 interrupt control



8259 interrupt controller



8259 interrupt controller



Data Bus Buffer	This block is used to communicate between 8259 and 8085/8086 by acting as buffer. It takes the control word from 8085/8086 and send it to the 8259. It transfers the opcode of the selected interrupts and address of ISR to the other connected microprocessor. It can send maximum 8-bit at a time.
R/W Control Logic	This block works when the value of pin CS is 0. This block is used to flow the data depending upon the inputs of RD and WR. These are active low pins for read and write.
Control Logic	It controls the functionality of each block. It has pin called INTR. This is connected to other microprocessors for taking the interrupt request. The INT pin is used to give the output. If 8259 is enabled, and also the interrupt flags of other microprocessors are high then this causes the value of the output INT pin high, and in this way this chip can responds requests made by other microprocessors.

8259 interrupt controller

The Block Diagram consists of 8 blocks which are – Data Bus Buffer, Read/Write Logic, Cascade Buffer Comparator, Control Logic, Priority Resolver and 3 registers- ISR, IRR, IMR.

Data bus buffer

This Block is used as a mediator between 8259 and 8086 microprocessor by acting as a buffer. It takes the control word from the 8086 microprocessor and transfer it to the control logic of 8259 microprocessor. Also, after selection of Interrupt by 8259 microprocessor, it transfer the opcode of the selected Interrupt and address of the Interrupt service sub routine to the connected microprocessor. The data bus buffer consists of 8 bits represented as D0-D7 in the block diagram. Thus, shows that a maximum of 8 bits data can be transferred at a time.

Read/Write logic

This block works only when the value of pin CS is low (as this pin is active low). This block is responsible for the flow of data depending upon the inputs of RD and WR. These two pins are active low pins used for read and write operations.

8259 interrupt controller

Control logic : It is the centre of the microprocessor and controls the functioning of every block. It has pin INTR which is connected with other microprocessor for taking interrupt request and pin INT for giving the output. If 8259 is enabled, and the other microprocessor interrupt flag is high then this causes the value of the output INT pin high and in this way 8259 responds to the request made by other microprocessor.

Interrupt request register (IRR) : It stores all the interrupt levels which are requesting for interrupt services.

Interrupt service register (ISR) : It stores the interrupt levels which are currently being executed.

Interrupt mask register (IMR) : It stores the interrupt levels which have to be masked by storing the masking bits of the interrupt level.

8259 interrupt controller

Priority resolver :

It examines all the three registers and set the priority of interrupts and according to the priority of the interrupts, interrupt with highest priority is set in ISR register. Also, it reset the interrupt level which is already been serviced in IRR.

Cascade buffer :

To increase the Interrupt handling capability, cascading is done for more number of pins by using cascade buffer. So, during increment of interrupt capability, CSA lines are used to control multiple interrupt structure.

SP/EN (Slave program/Enable buffer) pin is when set to high, works in master mode else in slave mode. In Non Buffered mode, SP/EN pin is used to specify whether 8259 work as master or slave and in Buffered mode, SP/EN pin is used as an output to enable data bus.

8259 Interrupt Controller

Features of 8259 PIC microprocessor –

Intel 8259 is designed for Intel 8085 and Intel 8086 microprocessor.

It can be programmed either in level triggered or in edge triggered interrupt level.

We can mask individual bits of interrupt request register.

We can increase interrupt handling capability upto 64 interrupt level by cascading further 8259 PIC.

Clock cycle is not required.

8259 interrupt controller

\overline{CS}	1		28	<u>Vcc</u>
\overline{WR}	2		27	A0
\overline{RD}	3		26	\overline{INTA}
D7	4		25	IR7
D6	5		24	IR6
D5	6		23	IR5
D4	7	8259	22	IR4
D3	8	PIC	21	IR3
D2	9		20	IR2
D1	10		19	IR1
D0	11		18	IR0
CAS0	12		17	INT
CAS1	13		16	$\overline{SP/EN}$
<u>Gnd</u>	14		15	CAS2

Control words of 8259

Command word of 8259 is divided into two parts :

Initialization command words(ICW)

Operating command words(OCW)

Initialization command words(ICW) :

ICW is given during the initialization of 8259

ICW₁ and ICW₂ commands are compulsory for initialization.

ICW₃ command is given during a cascaded configuration.

If ICW₄ is needed, then it is specified in ICW₁.

The sequence order of giving ICW commands is fixed i.e. ICW₁ is given first and then ICW₂ and then ICW₃.

Any of the ICW commands can not be repeated, but the entire initialization process can be repeated if required.

Operating command words(OCW) :

OCW is given during the operation of 8259 i.e. microprocessor starts using 8259.

OCW commands are not compulsory for 8259.

The sequence order of giving OCW commands is not fixed.

The OCW commands can be repeated.

8259 interrupt controller

ICW₁

A ₀	D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀
0	A ₇	A ₆	A ₅	1	LTIM	ADI	SINGL	IC ₄

1 = ICW₄ needed
0 = No ICW₄ needed

1 = Single mode
0 = Cascaded mode

CALL address interval
1 = Interval of 4
0 = Interval of 8

1 = Level triggered mode
0 = Edge triggered mode

A₇ - A₅ of interrupt vector address
(MCS-80/85 mode only)

ICW₂

A ₀	D ₇	D ₆	D ₅	D ₄	D ₃	D ₂	D ₁	D ₀
1	A ₁₅ T ₇	A ₁₄ T ₆	A ₁₃ T ₅	A ₁₂ T ₄	A ₁₁ T ₃	A ₁₀	A ₉	A ₈

A₁₅ - A₈ of interrupt vector address
(MCS-80/85 mode only)

T₇ - T₃ of interrupt vector address
(8086/8088 mode only)

8259 interrupt controller

When the ICW_1 is loaded, then the initializations performed are:

The edge sense circuit is reset because, by default, 8259 interrupt is edge triggered.

The interrupt mask register is cleared.

IR7 is assigned to priority 7.

Slave mode address is assigned as 7.

When $D_0 = 0$, this means IC_4 command is not required. Therefore, functions used in IC_4 are reset.

Special mask mode is reset and status read is assigned to IRR.

ICW₂ command :

The control word is recognized as ICW_2 when $A_0 = 1$.

It stores the information regarding the interrupt vector address.

In the 8085 based system, the A_{15} to A_8 bits of control word is used for interrupt vector addresses.

In the 8086 based system, T_6 to T_3 bits are inserted instead of A_{15} to A_8 and A_{10} to A_8 are used for selecting interrupt level, i.e. 000 for IR_0 and 111 for IR_7 .

8259 Operational command word

Operational command word 1

It is used to set and reset the mask bits in IMR(interrupt mask register). $M_7 - M_0$ describes 8 mask bits

OCW₁



Interrupt mask
1 = Mask set
0 = Mask reset

Direct Memory Access (DMA)

Direct Memory Access (DMA)

DMA Controller is a hardware device that allows I/O devices to directly access memory with less participation of the processor.

DMA controller needs the same old circuits of an interface to communicate with the CPU and Input/ Output devices.

The unit communicates with the CPU through data bus and control lines. Through the use of the address bus and allowing the DMA and RS register to select inputs, the register within the DMA is chosen by the CPU.

RD and WR are two-way inputs.

When BG (bus grant) input is 0, the CPU can communicate with DMA registers.

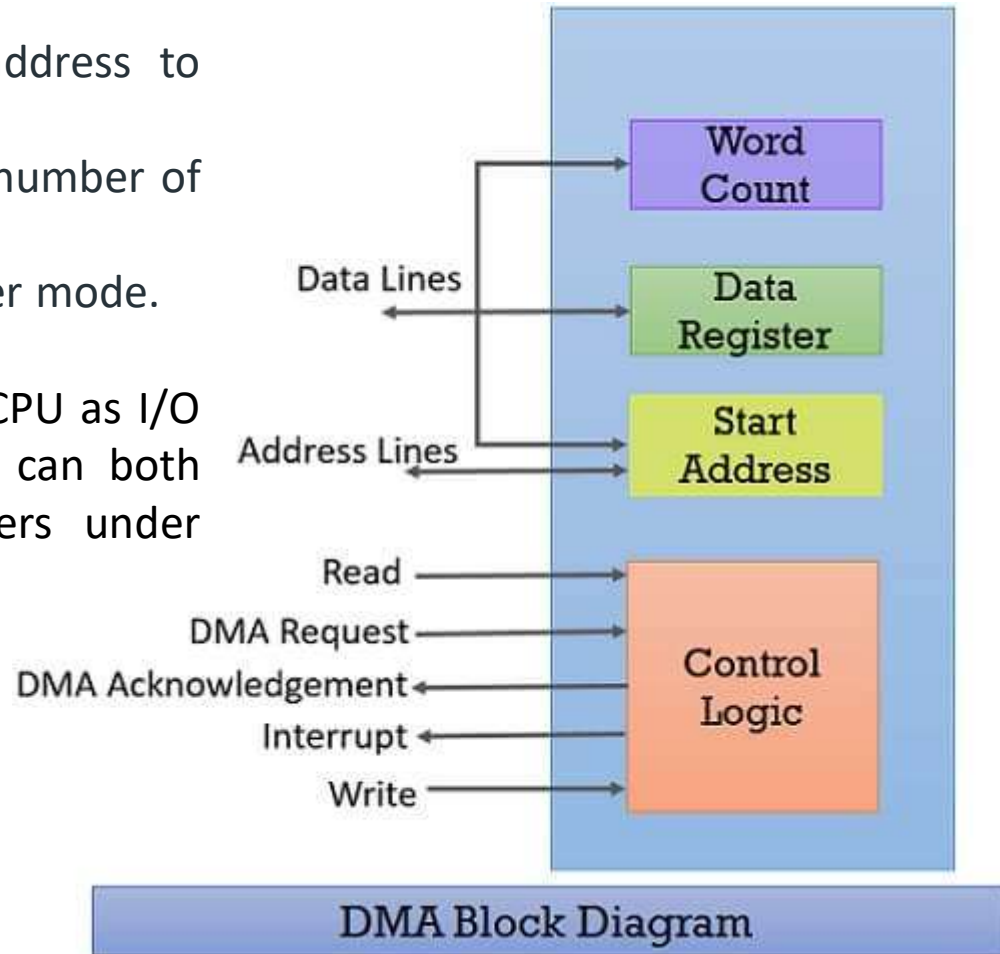
When BG (bus grant) input is 1, the CPU has relinquished the buses and DMA can communicate directly with the memory.

Direct Memory Access (DMA)

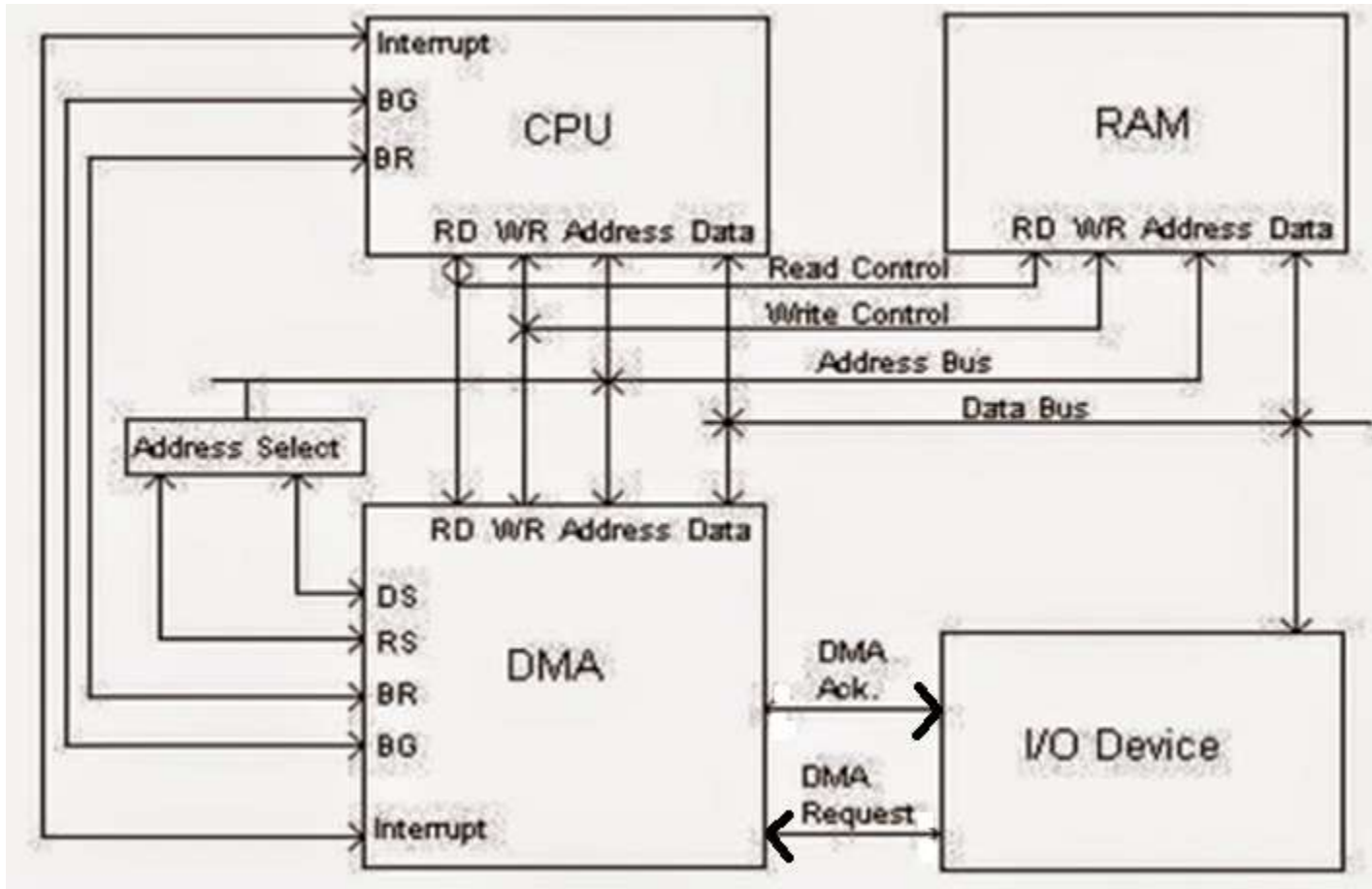
DMA registers

- **Address register** – It contains the address to specify the desired location in memory.
- **Word count register** – It contains the number of words to be transferred.
- **Control register** – It specifies the transfer mode.

All registers in the DMA appear to the CPU as I/O interface registers. Therefore, the CPU can both read and write into the DMA registers under program control via the data bus.



Direct Memory Access (DMA)



Serial communication

Serial communication is a communication method that uses one or two transmission lines to send and receive data, and that data is continuously sent and received one bit at a time. Since it allows for connections with few signal wires, one of its merits is its ability to hold down on wiring material and relaying equipment costs.

Serial communication standards

RS-232C/RS-422A/RS-485 are EIA (Electronic Industries Association) communication standards. Of these communication standards, RS-232C has been widely adopted in a variety of applications, and it is even standard equipment on computers and is often used to connect modems and mice. Sensors and actuators also contain these interfaces, many of which can be controlled via serial communication.

Serial Standard	Operation mode	Total nr. of devices	Cable length	Speed	Wires
RS-232	Single Ended	1 Sender / 1 Receiver	15 m	20 Kbits/s	min. 3
RS-422	Differential	1 Sender / 10 Receiver	1200 m	10 Mbit/s	4
RS-485	Differential	32 Sender / 32 Receiver	1200 m	10 Mbit/s	2

Serial communication

Single-ended signaling is the simplest and most commonly used method of transmitting electrical signals over wires.

One wire carries a varying voltage that represents the signal, while the other wire is connected to a reference voltage, usually ground.

Differential signaling is a method for electrically transmitting information using two complementary signals.

The technique sends the same electrical signal as a **differential pair** of signals, each in its own conductor.

The pair of conductors can be wires in a twisted-pair or ribbon cable or traces on a printed circuit board.

Serial communication

RS-232C

This serial communication standard is widely used and is often equipped on computers as standard. It is also called "EIA-232". The purpose and timing of the signal lines and the connectors have been defined (D-sub 25-pin or D-sub 9-pin). Currently the standard has been revised with the addition of signal lines and is formally called "ANSI/EIA-232-E". However, even now it is generally referred to as "RS-232C".

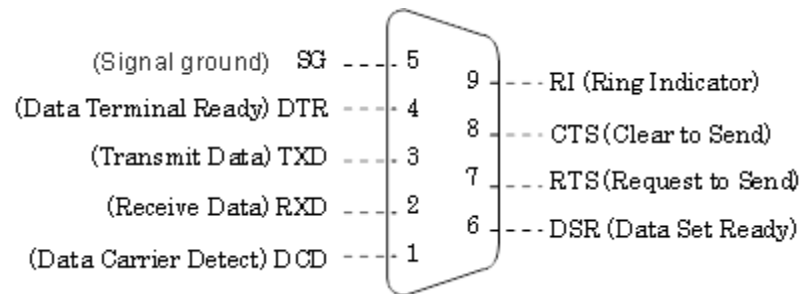
RS-422A

This standard fixes problems in RS-232C such as a short transmission distance and a slow transmission speed. It is also called "EIA-422A". The purpose and timing of the signal lines are defined, but the connectors are not. Many compatible products primarily adopt D-sub 25-pin and D-sub 9-pin connectors.

RS-485

This standard fixes the problem of few connected devices in RS-422A. It is also called "EIA-485". RS-485 is forward compatible standard with RS-422A. The purpose and timing of the signal lines are defined, but the connectors are not. Many compatible products primarily adopt D-sub 25-pin and D-sub 9-pin connectors.

In RS-232C, the connectors to use and the signal assignments have been defined and are standardized. The figure to the right describes the D-sub 9-pin signal assignments and signal lines.



Pin No.	Signal name	Description	
1	DCD	Data Carrier Detect	Carrier detect
2	RxD	Received Data	Received data
3	TxD	Transmitted Data	Transmitted data
4	DTR	Data Terminal Ready	Data terminal ready
5	SG	Signal Ground	Signal ground or common return
6	DSR	Data Set Ready	Data set ready
7	RTS	Request To Send	Request to send
8	CTS	Clear To Send	Clear to send
9	RI	Ring Indicator	Ring indicator
CASE	FG	Frame Ground	Maintenance ground or earth

RS 232 Connection method

In RS-232C, the connectors and signal assignments have been standardized, so many standard-compliant cables are available commercially. However, equipment comes in the following types, and depending on the equipment that will be connected, a straight cable or a crossover cable is required.

Equipment type

DCE

Data communication equipment. This term indicates equipment that passively operates such as modems, printers, and plotters.

DTE

Data terminal equipment. This term indicates equipment that actively operates such as computers.

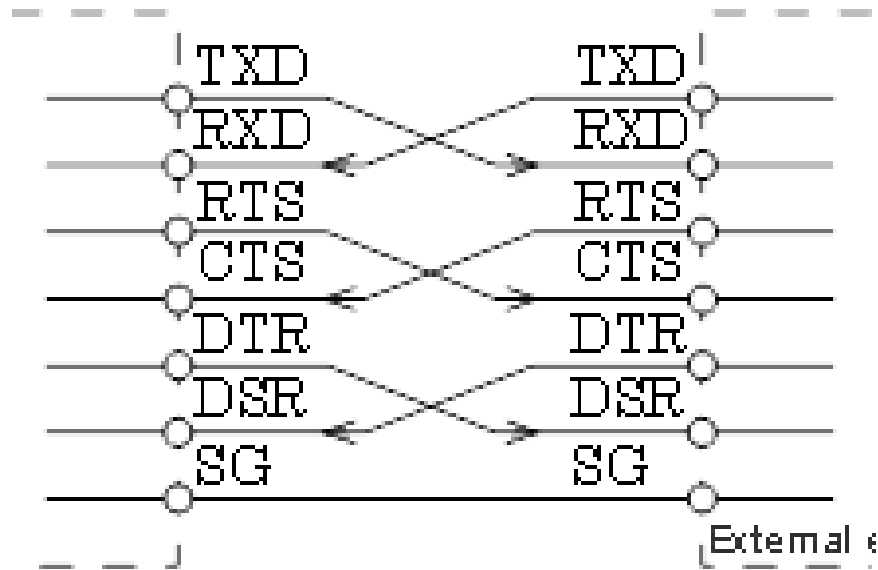
Full-duplex communication

A method where send and receive both have their own transmission line so data can be simultaneously sent and received.

Half-duplex communication

A method where communication is performed using one transmission line while switching between send and receive. For this reason, simultaneous communication cannot be performed.

Crossover cable connection



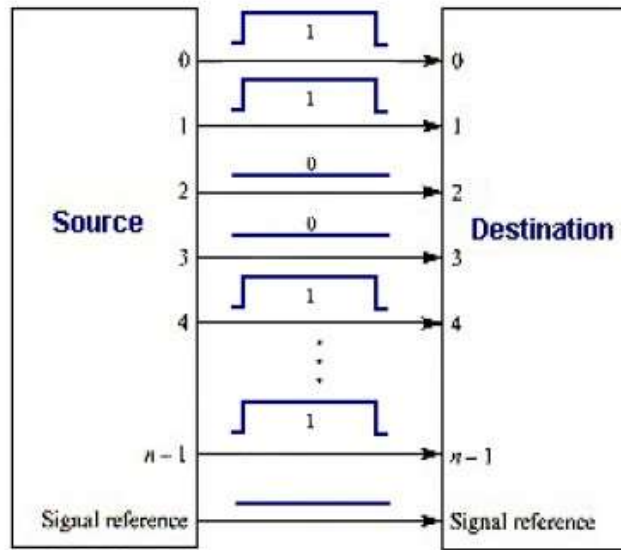
Full-duplex communication A method where send and receive both have their own transmission line so data can be simultaneously sent and received. Half-duplex communication A method where communication is performed using one transmission line while switching between send and receive. For this reason, simultaneous communication cannot be performed

Serial Data transfer schemes

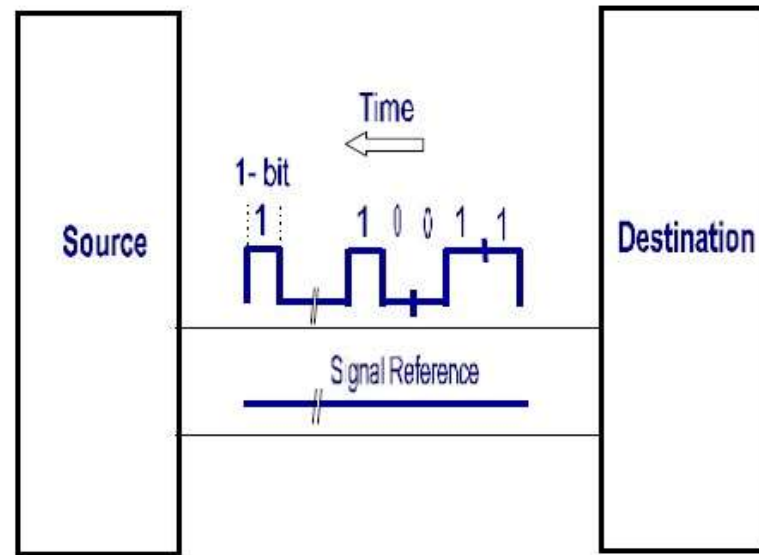
Serial communication transmits data one bit at a time, sequentially, over a single communication line to a receiver.

Serial is also a most popular communication protocol that is used by many devices for instrumentation.

This method is used when data transfer rates are very low or the data must be transferred over long distances and also where the cost of cable and synchronization difficulties makes parallel communication impractical. Serial communication is popular because most



Parallel Transmission

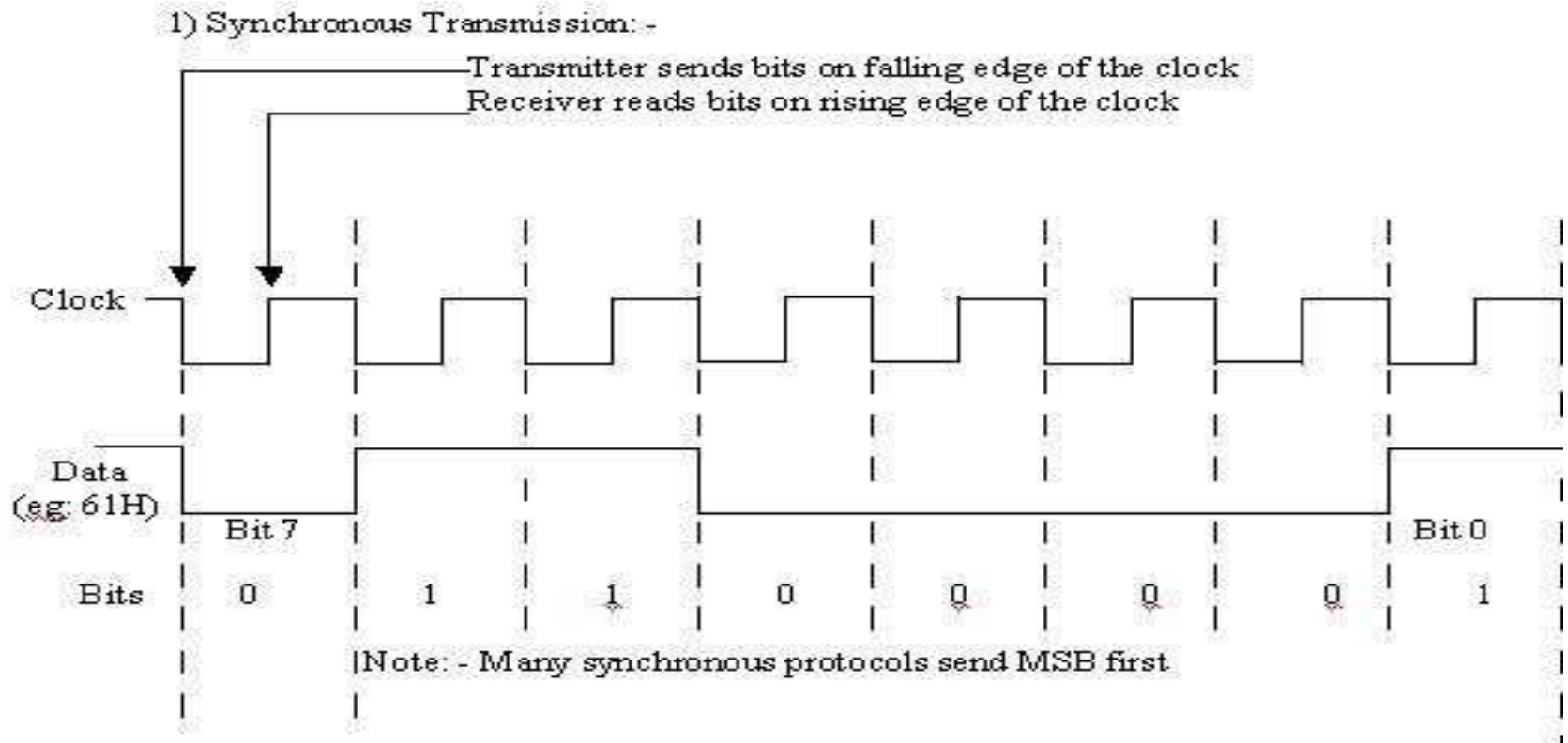


Serial Transmission

Synchronous data transmission

The synchronous signaling methods use two different signals. A pulse on one signal line indicates when another bit of information is ready on the other signal line.

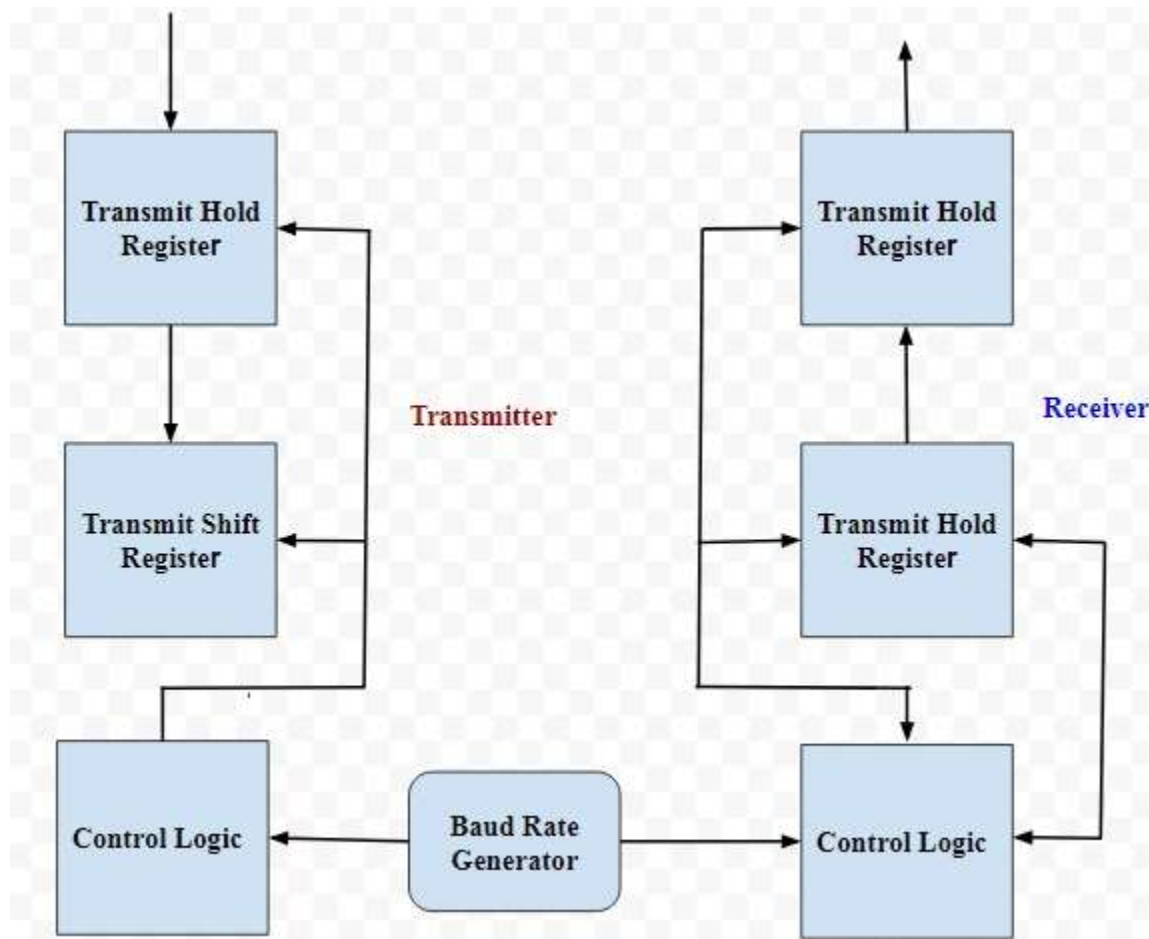
In synchronous transmission, the stream of data to be transferred is encoded and sent on one line, and a periodic pulse of voltage which is often called the "clock" is put on another line, that tells the receiver about the beginning and the ending of each bit.



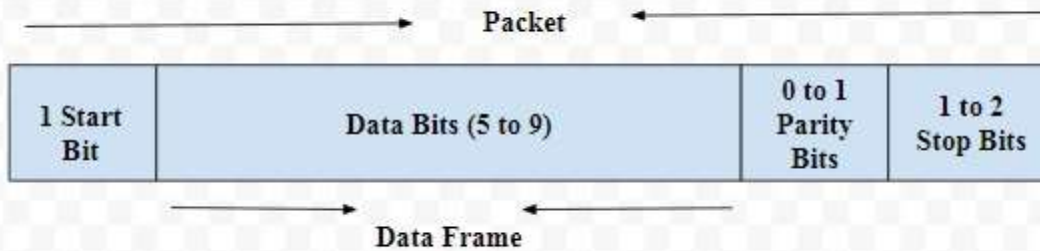
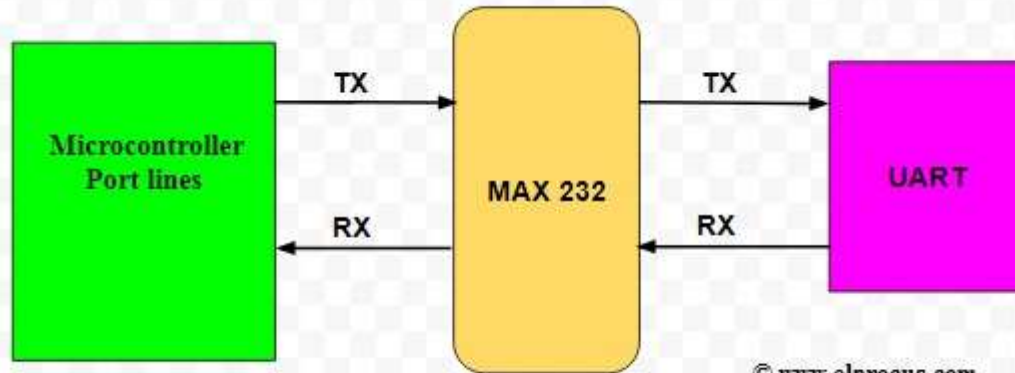
UART BLOCK DIAGRAM

The **UART full form** is “Universal Asynchronous Receiver/Transmitter”, and it is an inbuilt IC within a microcontroller but not like a communication protocol (I2C & SPI). The main function of UART is to serial data communication.

In UART, the communication between two devices can be done in two ways namely serial data communication and parallel data communication.



UART COMMUNICATION



Serial communication

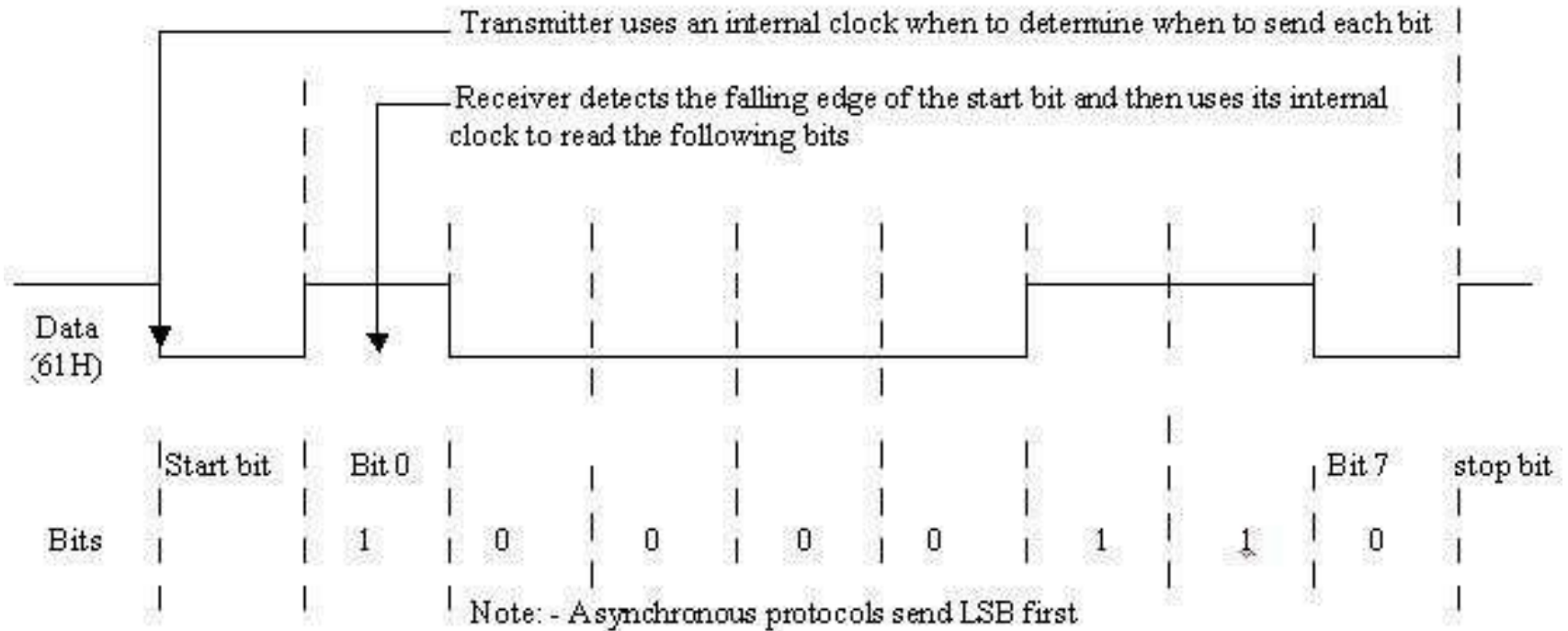
The asynchronous signaling methods use only one signal. The receiver uses transitions on that signal to figure out the transmitter bit rate (known as auto baud) and timing.

A pulse from the local clock indicates when another bit is ready. That means synchronous transmissions use an external clock, while asynchronous transmissions use special signals along the transmission medium.

Asynchronous communication is the commonly prevailing communication method in the personal computer industry, due to the reason that it is easier to implement and has the unique advantage that bytes can be sent whenever they are ready, a no need to wait for blocks of data to accumulate.

Asynchronous data transmission

2) Asynchronous Transmission: -



THANK YOU

Lecture 2

The 8051 Microcontroller architecture

Contents:

- **Introduction**
- **Block Diagram and Pin Description of the 8051**
- **Registers**
- **Some Simple Instructions**
- **Structure of Assembly language and Running an 8051 program**
- **Memory mapping in 8051**
- **8051 Flag bits and the PSW register**
- **Addressing Modes**
- **16-bit, BCD and Signed Arithmetic in 8051**
- **Stack in the 8051**
- **LOOP and JUMP Instructions**
- **CALL Instructions**
- **I/O Port Programming**



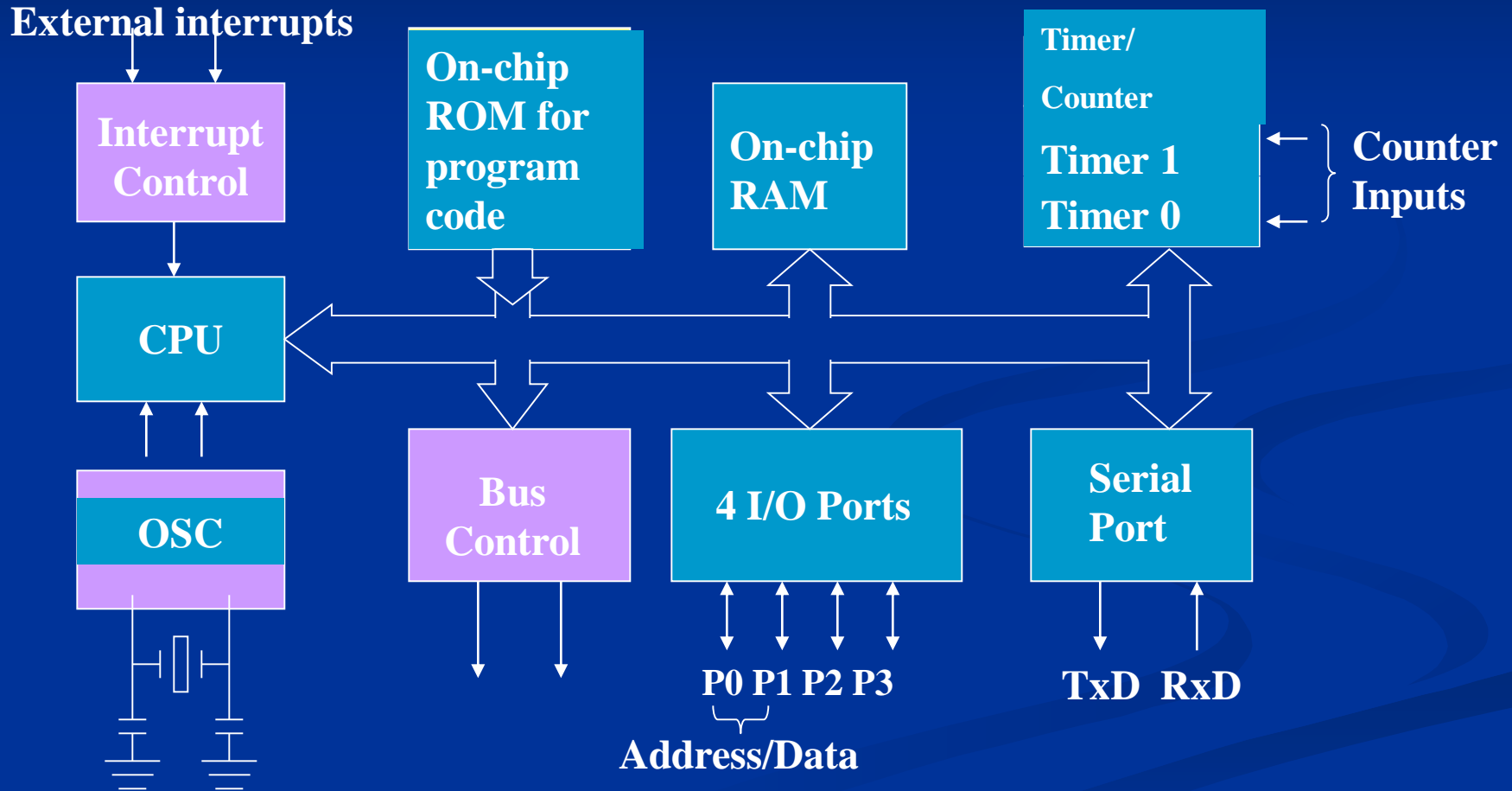
Three criteria in Choosing a Microcontroller

1. meeting the computing needs of the task efficiently and cost effectively
 - speed, the amount of ROM and RAM, the number of I/O ports and timers, size, packaging, power consumption
 - easy to upgrade
 - cost per unit
2. availability of software development tools
 - assemblers, debuggers, C compilers, emulator, simulator, technical support
3. wide availability and reliable sources of the microcontrollers.

The 8051 microcontroller

- a Harvard architecture (separate instruction/data memories)
- single chip microcontroller (μC)
- developed by Intel in 1980 for use in embedded systems.
- today largely superseded by a vast range of faster and/or functionally enhanced 8051-compatible devices manufactured by more than 20 independent manufacturers

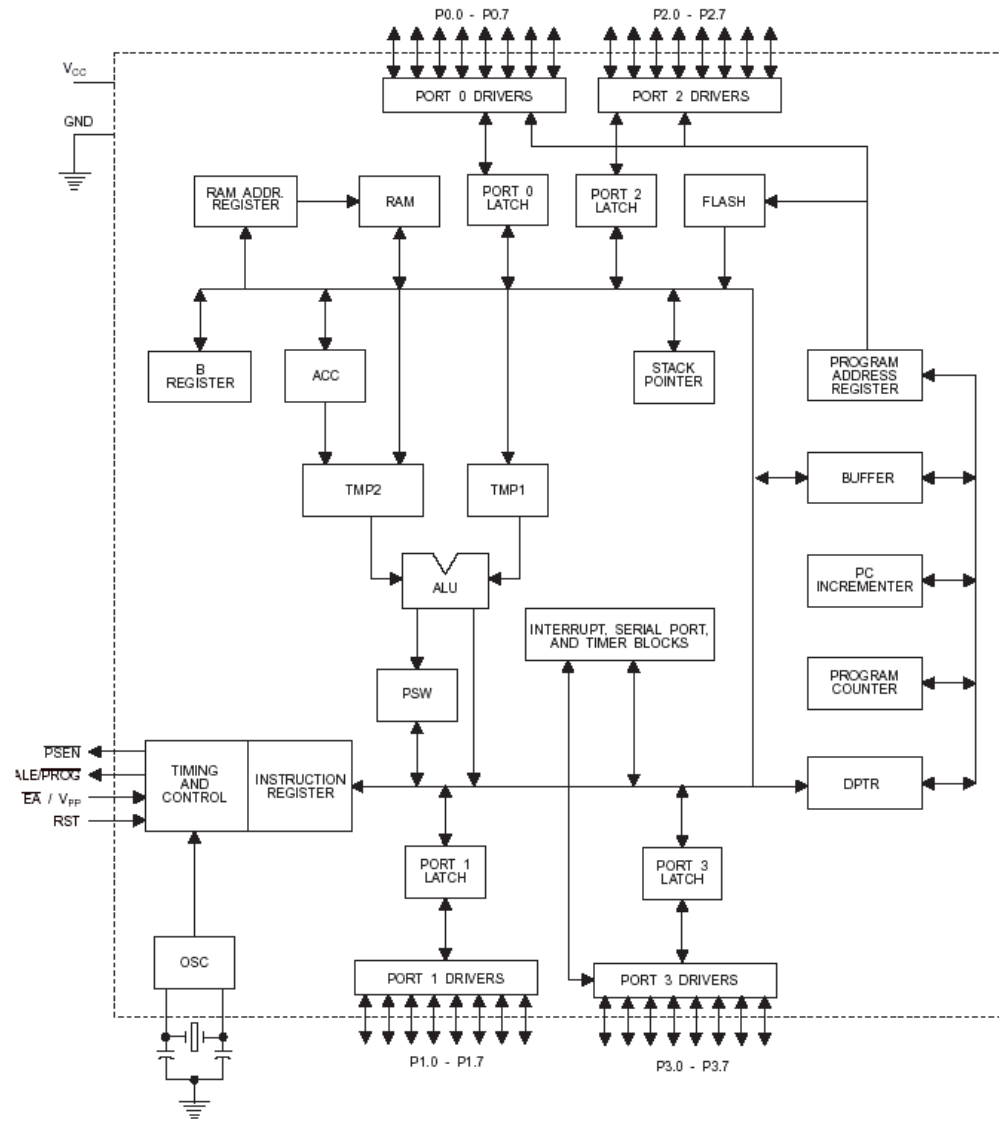
Block Diagram



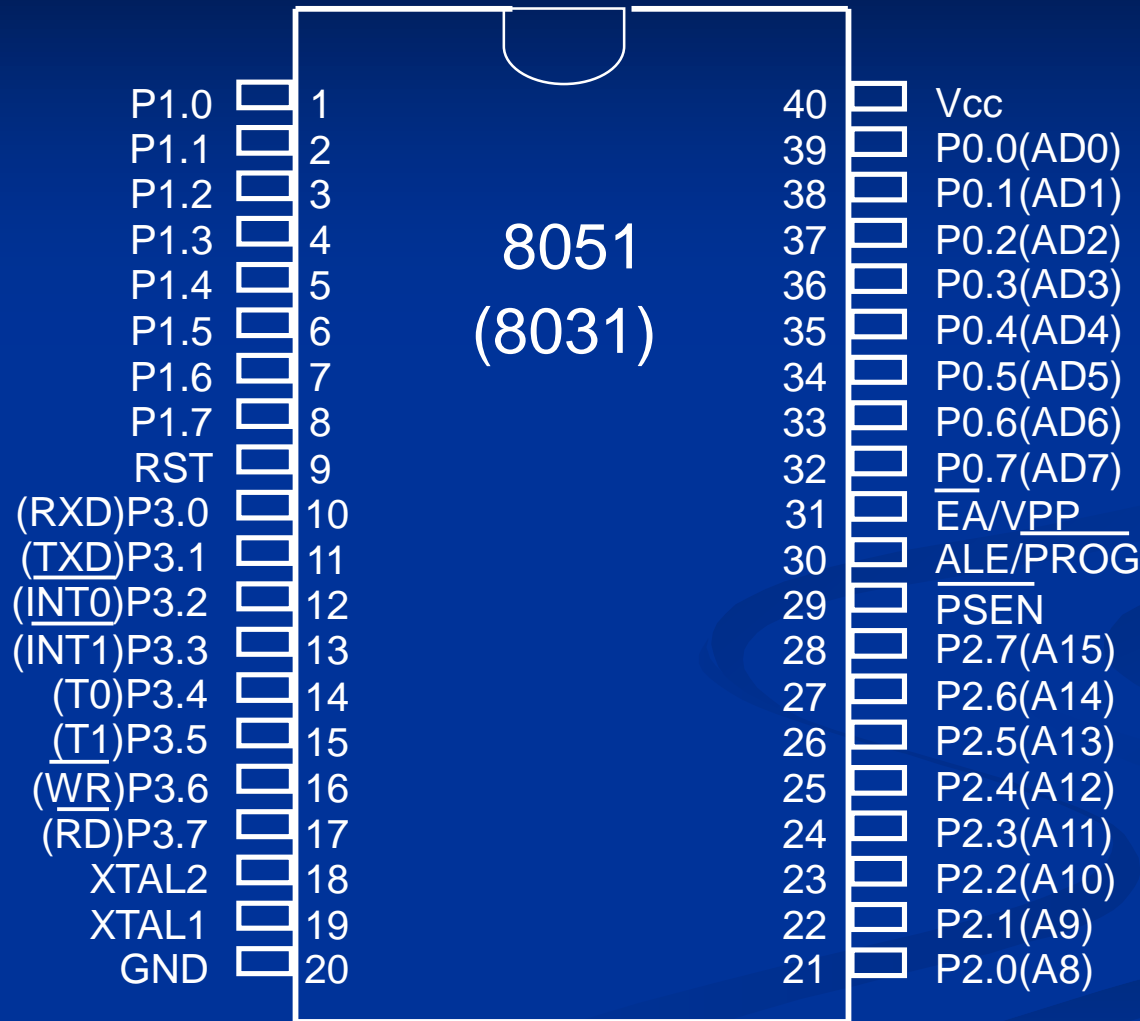
Comparison of the 8051 Family Members

Feature	8051	8052	8031
ROM (program space in bytes)	4K	8K	0K
RAM (bytes)	128	256	128
Timers	2	3	2
I/O pins	32	32	32
Serial port	1	1	1
Interrupt sources	6	8	6

Block Diagram



Pin Description of the 8051



Pins of 8051 (1/4)

- Vcc (pin 40) :
 - Vcc provides supply voltage to the chip.
 - The voltage source is +5V.
- GND (pin 20) : ground
- XTAL1 and XTAL2 (pins 19,18) :
 - These 2 pins provide external clock.
 - Way 1 : using a quartz crystal oscillator
 - Way 2 : using a TTL oscillator
 - Example 4-1 shows the relationship between XTAL and the machine cycle.

Pins of 8051 (2/4)

- RST (pin 9) : reset
 - It is an input pin and is active high (normally low) .
 - The high pulse must be high at least 2 machine cycles.
 - It is a power-on reset.
 - Upon applying a high pulse to RST, the microcontroller will reset and all values in registers will be lost.
 - Reset values of some 8051 registers
 - Way 1 : Power-on reset circuit
 - Way 2 : Power-on reset with debounce

Pins of 8051 (3/4)

- /EA (pin 31) : external access
 - There is no on-chip ROM in 8031 and 8032 .
 - The /EA pin is connected to GND to indicate the code is stored externally.
 - /PSEN & ALE are used for external ROM.
 - For 8051, /EA pin is connected to Vcc.
 - “/” means active low.
- /PSEN (pin 29) : program store enable
 - This is an output pin and is connected to the OE pin of the ROM.
 - See Chapter 14.

Pins of 8051 (4/4)

- ALE (pin 30) : address latch enable
 - It is an output pin and is active high.
 - 8051 port 0 provides both address and data.
 - The ALE pin is used for de-multiplexing the address and data by connecting to the G pin of the 74LS373 latch.
- I/O port pins
 - The four ports P0, P1, P2, and P3.
 - Each port uses 8 pins.
 - All I/O pins are bi-directional.

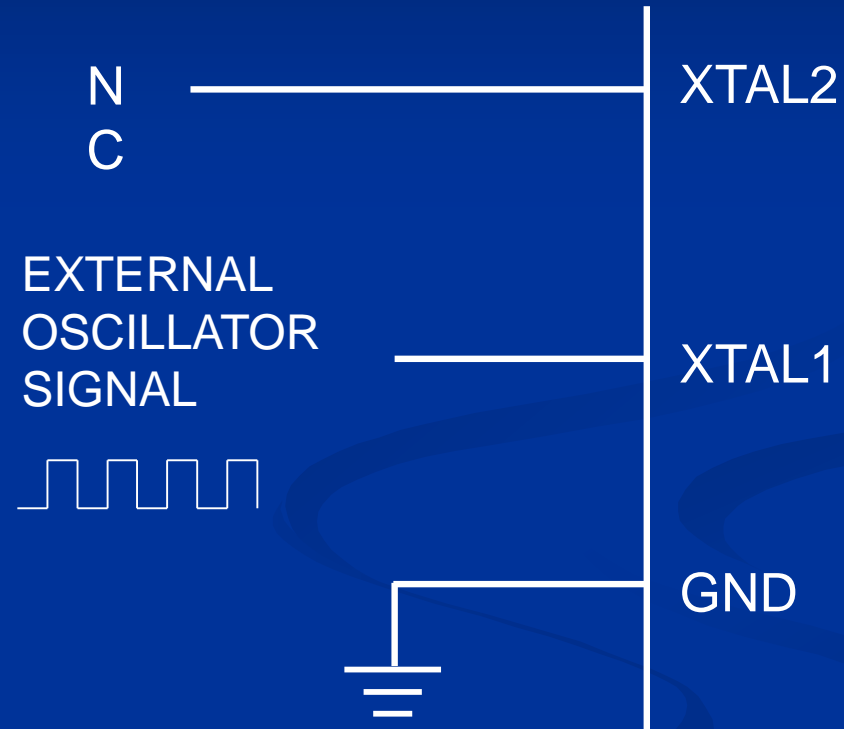
Figure 4-2 (a). XTAL Connection to 8051

- Using a quartz crystal oscillator^{XTAL1}
- We can observe the frequency on the XTAL2 pin.



Figure 4-2 (b). XTAL Connection to an External Clock Source

- Using a TTL oscillator
- XTAL2 is unconnected.



RESET Value of Some 8051 Registers:

Register	Reset Value
PC	0000
ACC	0000
B	0000
PSW	0000
SP	0007
DPTR	0000

RAM are all zero.



Figure 4-3 (a). Power-On RESET Circuit

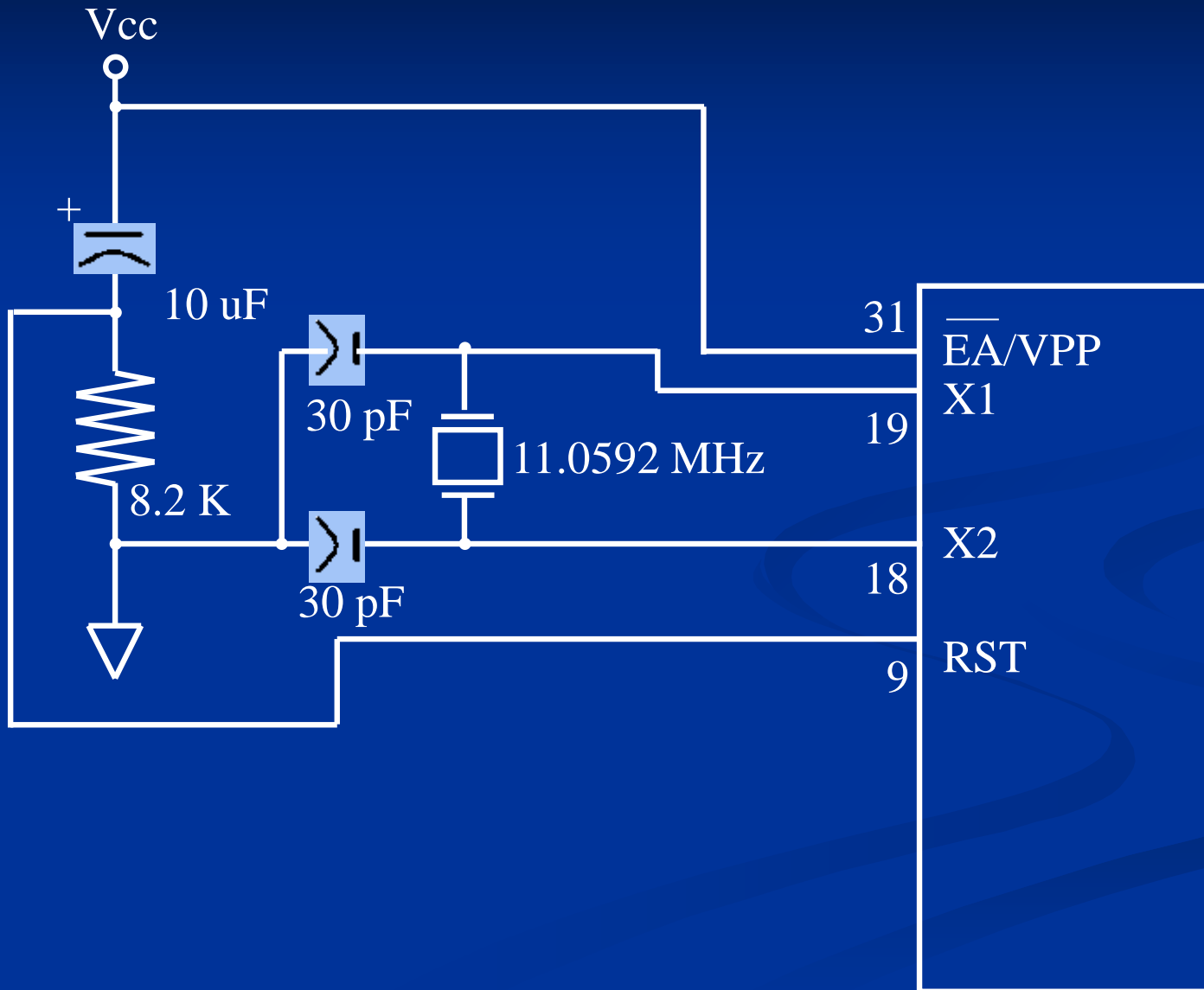
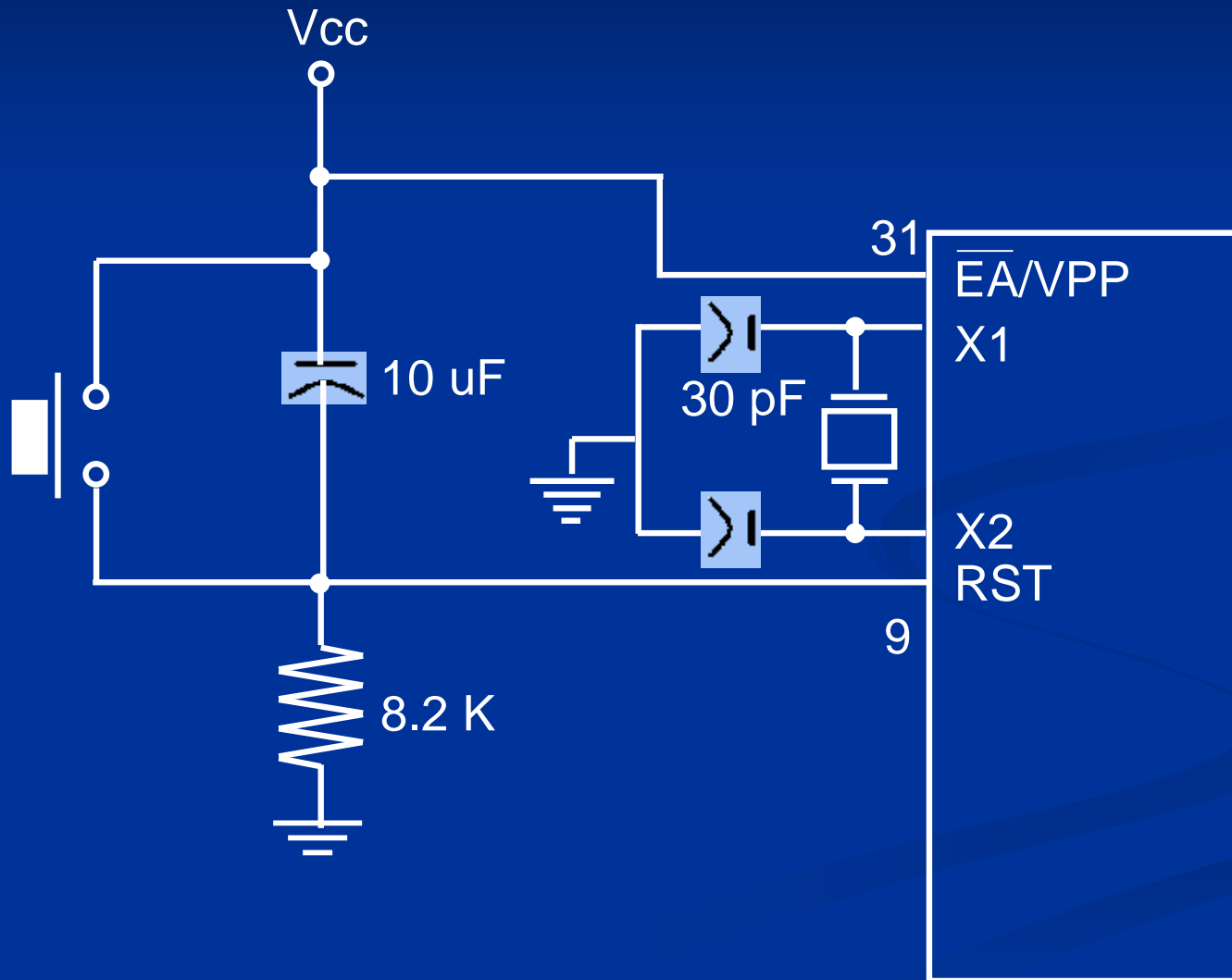


Figure 4-3 (b). Power-On RESET with Debounce

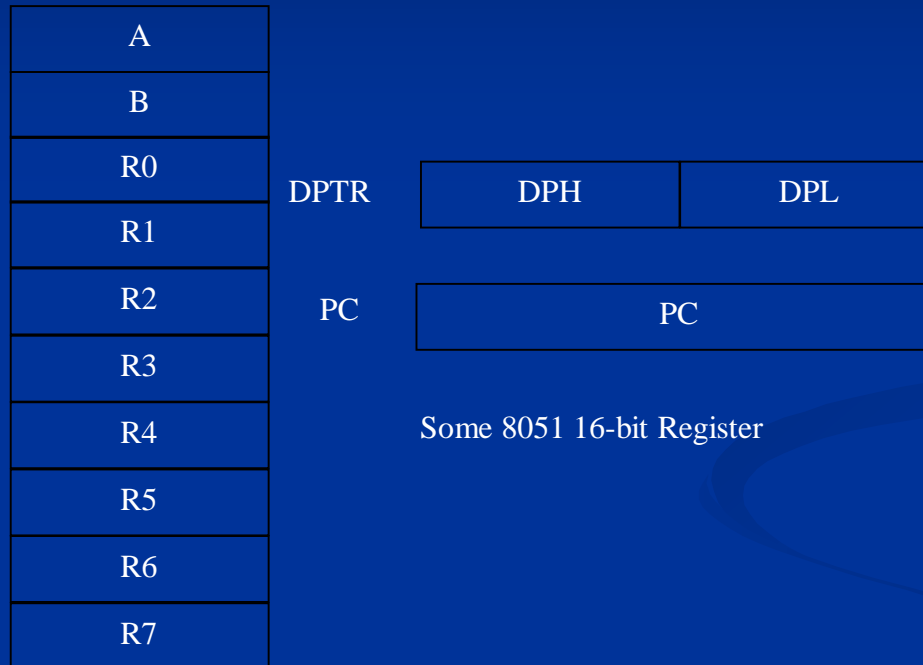


Pins of I/O Port

- The 8051 has four I/O ports
 - Port 0 (pins 32-39) : P0 (P0.0~P0.7)
 - Port 1 (pins 1-8) : P1 (P1.0~P1.7)
 - Port 2 (pins 21-28) : P2 (P2.0~P2.7)
 - Port 3 (pins 10-17) : P3 (P3.0~P3.7)
 - Each port has **8 pins**.
 - Named P0.X (X=0,1,...,7) , P1.X, P2.X, P3.X
 - Ex : P0.0 is the bit 0 (LSB) of P0
 - Ex : P0.7 is the bit 7 (MSB) of P0
 - These 8 bits form a byte.
- Each port can be used as input or output (bi-direction).



Registers



Some 8-bit Registers of
the 8051

Some 8051 16-bit Register

Memory Map (RAM)

IRAM
Addr

00

R0 R1 R2 R3 R4 R5 R6 R7

Reg. Bank 0

08

R0 R1 R2 R3 R4 R5 R6 R7

Reg. Bank 1

10

R0 R1 R2 R3 R4 R5 R6 R7

Reg. Bank 2

18

R0 R1 R2 R3 R4 R5 R6 R7

Reg. Bank 3

20

00 08 10 18 20 28 30 38

Bits 00-3F

28

40 48 50 58 60 68 70 78

Bits 40-7F

30

General User RAM
& Stack Space
(80 bytes, 30h-7Fh)

General
IRAM

7F

80

⋮
⋮
⋮

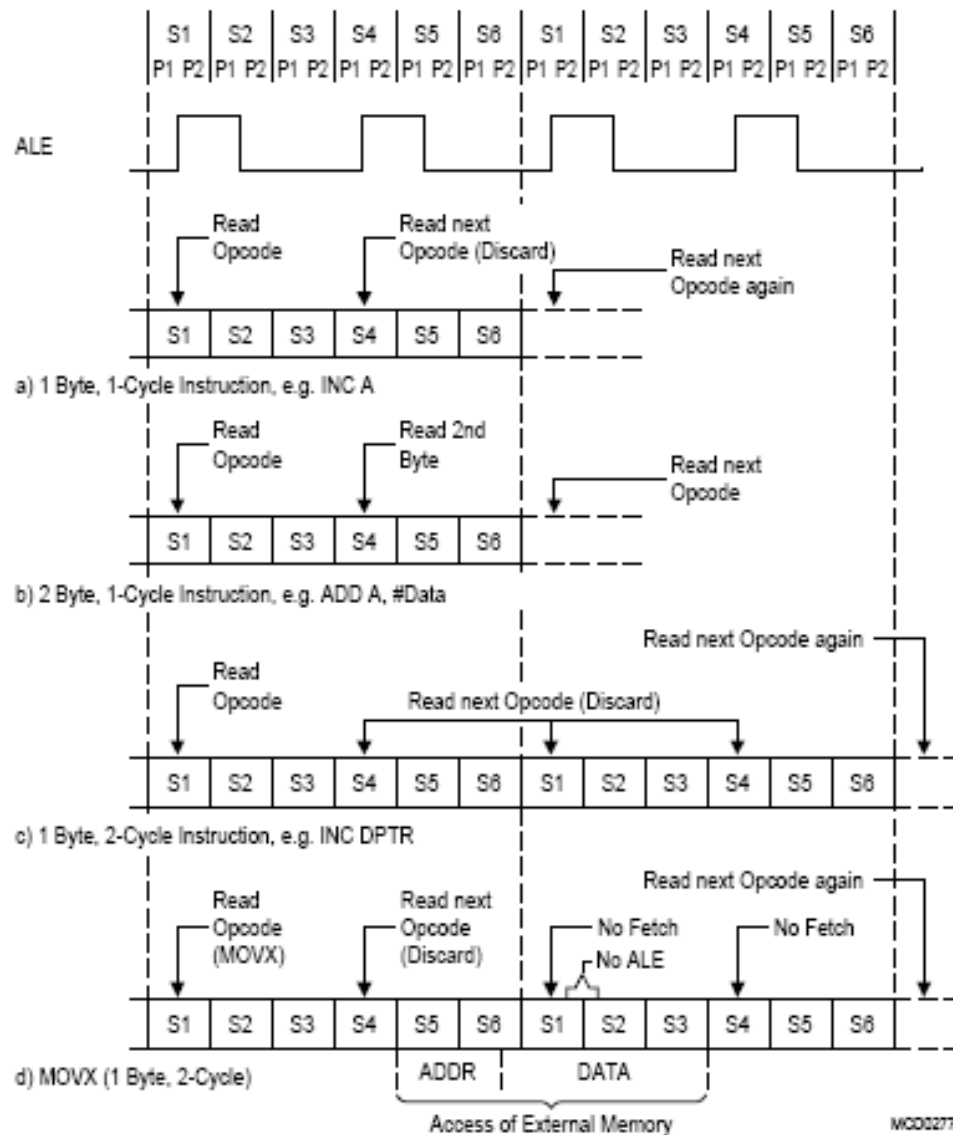
Special Function
Registers (SFRs)
(80h - FFh)

SFRs

CPU timing

- Most 8051 instructions are executed in one cycle.
- MUL (multiply) and DIV (divide) are the only
- instructions that take more than two cycles to complete (four cycles)
- Normally two code bytes are fetched from the program memory during every machine cycle.
- The only exception to this is when a MOVX instruction is executed. MOVX is a one-byte, 2-cycle instruction that accesses external data memory.
- During a MOVX, the two fetches in the second cycle are skipped while the external data memory is being addressed and strobed.

8051 machine cycle



Example :

Find the machine cycle for

(a) XTAL = 11.0592 MHz

(b) XTAL = 16 MHz.

Solution:

(a) $11.0592 \text{ MHz} / 12 = 921.6 \text{ kHz}$;

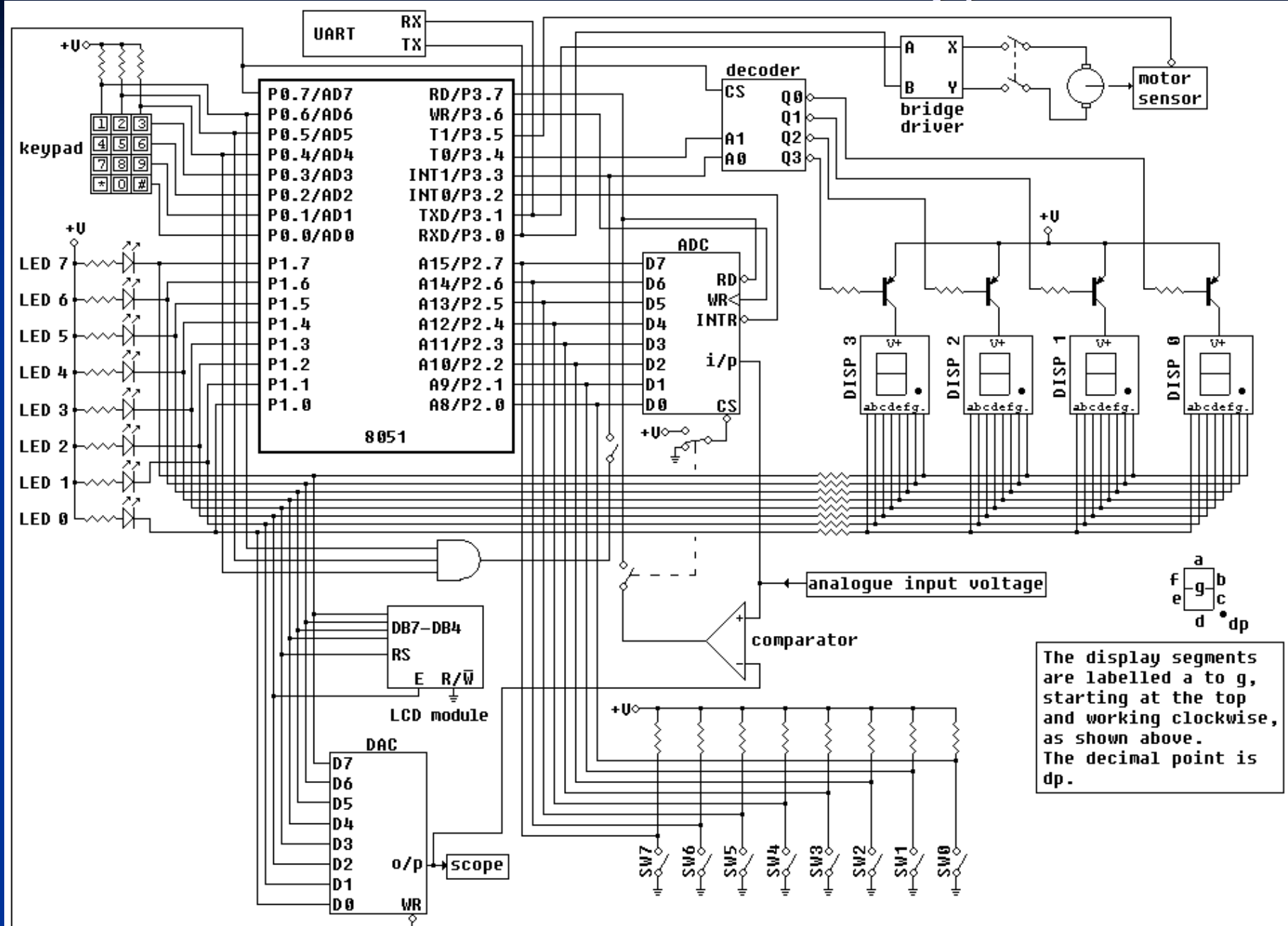
machine cycle = $1 / 921.6 \text{ kHz} = 1.085 \mu\text{s}$

(b) $16 \text{ MHz} / 12 = 1.333 \text{ MHz}$;

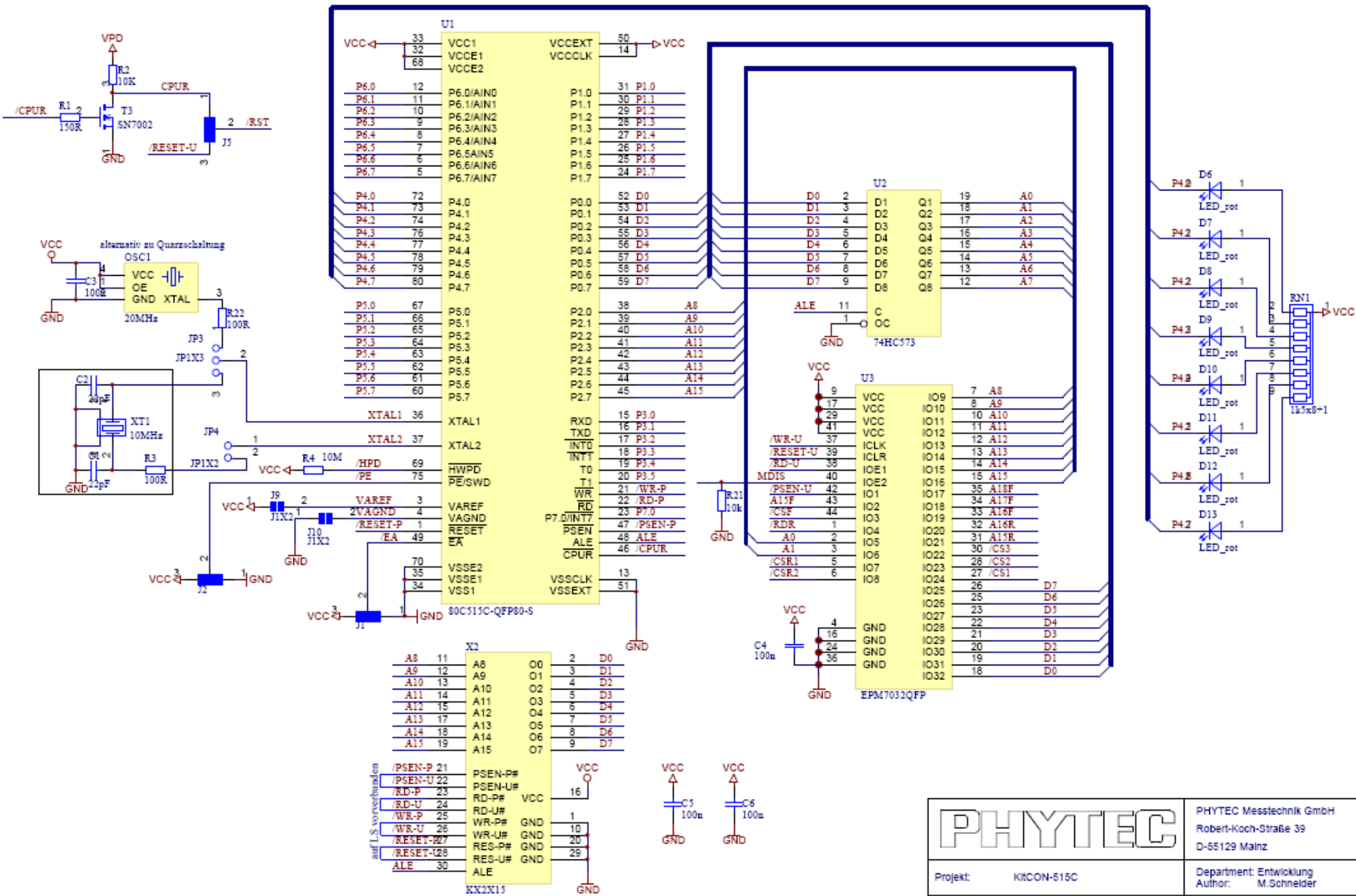
machine cycle = $1 / 1.333 \text{ MHz} = 0.75 \mu\text{s}$



Edsim51 emulator diagram



KitCON-515 schematic



Timers

- 8051 has two 16-bit on-chip timers that can be used for timing durations or for counting external events
- The high byte for timer 1 (TH1) is at address 8DH while the low byte (TL1) is at 8BH
- The high byte for timer 0 (TH0) is at 8CH while the low byte (TL0) is at 8AH.
- Timer Mode Register (TMOD) is at address 88H

Timer Mode Register

- Bit 7: Gate bit; when set, timer only runs while $\setminus INT$ high. (T0)
- Bit 6: Counter/timer select bit; when set timer is an event counter when cleared timer is an interval timer (T0)
- Bit 5: Mode bit 1 (T0)
- Bit 4: Mode bit 0 (T0)
- Bit 3: Gate bit; when set, timer only runs while $\setminus INT$ high. (T1)
- Bit 2: Counter/timer select bit; when set timer is an event counter when cleared timer is an interval timer (T1)
- Bit 1: Mode bit 1 (T1)
- Bit 0: Mode bit 0 (T1)

Timer Modes

- M1-M0: 00 (Mode 0) – 13-bit mode (not commonly used)
- M1-M0: 01 (Mode 1) - 16-bit timer mode
- M1-M0: 10 (Mode 2) - 8-bit auto-reload mode
- M1-M0: 11 (Mode 3) – Split timer mode

8051 Interrupt Vector Table

Interrupt	Flag	Vector
System reset	RST	0000H
External interrupt 0	IE0	0003H
Timer 0	TF0	000BH
External interrupt 1	IE1	0013H
Timer 1	TF1	001BH
Serial port	RI or TI	0023H

The Stack and Stack Pointer

- The Stack Pointer, like all registers except DPTR and PC, may hold an 8-bit (1-byte) value.
- The Stack Pointer is used to indicate where the next value to be removed from the stack should be taken from.
- When you push a value onto the stack, the 8051 first increments the value of SP and then stores the value at the resulting memory location.
- When you pop a value off the stack, the 8051 returns the value from the memory location indicated by SP, and then decrements the value of SP.
- This order of operation is important. When the 8051 is initialized SP will be initialized to 07h. If you immediately push a value onto the stack, the value will be stored in Internal RAM address 08h. This makes sense taking into account what was mentioned two paragraphs above: First the 8051 will increment the value of SP (from 07h to 08h) and then will store the pushed value at that memory address (08h).
- SP is modified directly by the 8051 by six instructions: PUSH, POP, ACALL, LCALL, RET, and RETI. It is also used intrinsically whenever an interrupt is triggered

DEPT & SEM : EEE & III/II SEM

SUBJECT NAME: DIGITAL COMPUTE PLATFORMS

COURSE CODE: 19A02601T

UNIT : IV

PREPARED BY :
C.MUNIKANTHA

OUTLINE – UNIT-1

Introduction to the TMS320LF2407 DSP Controller

Basic architectural features

Physical Memory

Software Tools.

Introduction to Interrupts

interrupt Hierarchy

Interrupt Control Registers.

C2xx DSP CPU

Instruction Set:

Introduction & code Generation

Components of the C2xx DSP core

Mapping External Devices to the C2xx core

peripheral interface - system configuration registers

Memory

Memory Addressing Modes

Assembly Programming Using the C2xx DSP Instruction set.

DSP CONTROLLER ARCHITECTURE

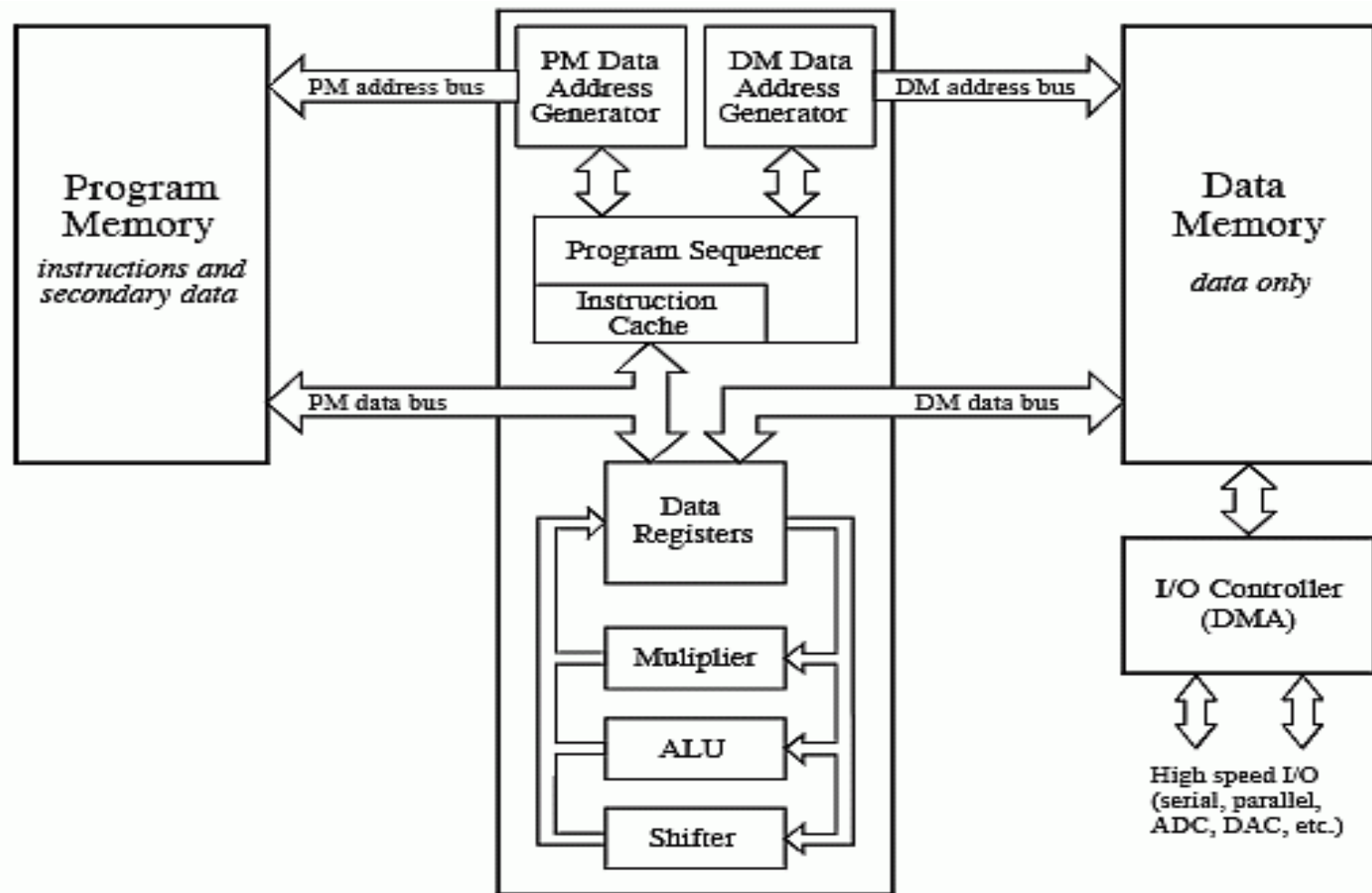
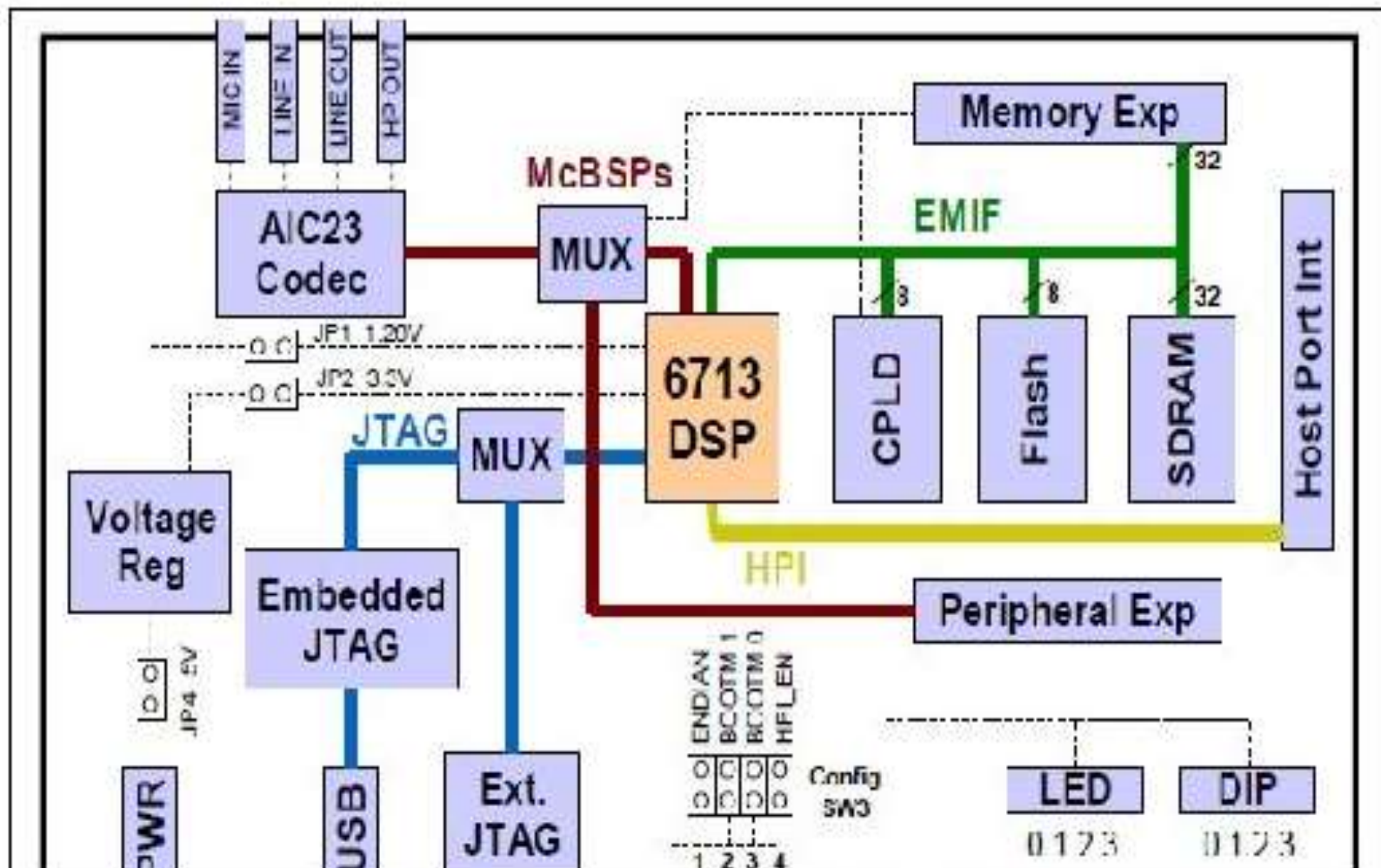


FIGURE 1
Typical DSP architecture. Digital Signal Processors are designed to implement tasks in parallel. This simplified diagram is of the Analog Devices SHARC DSP.

Architecture of TMS320C67xx

TMS320C6713 DSP Starter Kit (DSK) Block Diagram



-
- A TMS320C6713 DSP operating at 225 MHz.
 - 16 Mbytes of synchronous DRAM
 - 512 Kbytes of non-volatile Flash memory
 - (256 Kbytes usable in default configuration)
 - 4 user accessible LEDs and DIP switches
 - Software board configuration through
 - registers implemented in CPLD

-
- **JTAG emulation through on-board JTAG**
 - **emulator with USB host interface or external emulator**

A TMS 320 C 6713 DSP operating at 225 MHz.

- 16 Mbytes of synchronous DRAM
- 512 Kbytes of non-volatile Flash memory
- (256 Kbytes usable in default configuration)
- 4 user accessible LEDs and DIP switches
- Software board configuration through
- registers implemented in CPLD ACOE 343 - Embedded Real-Time Processor Systems

DSP vs. Microcontroller

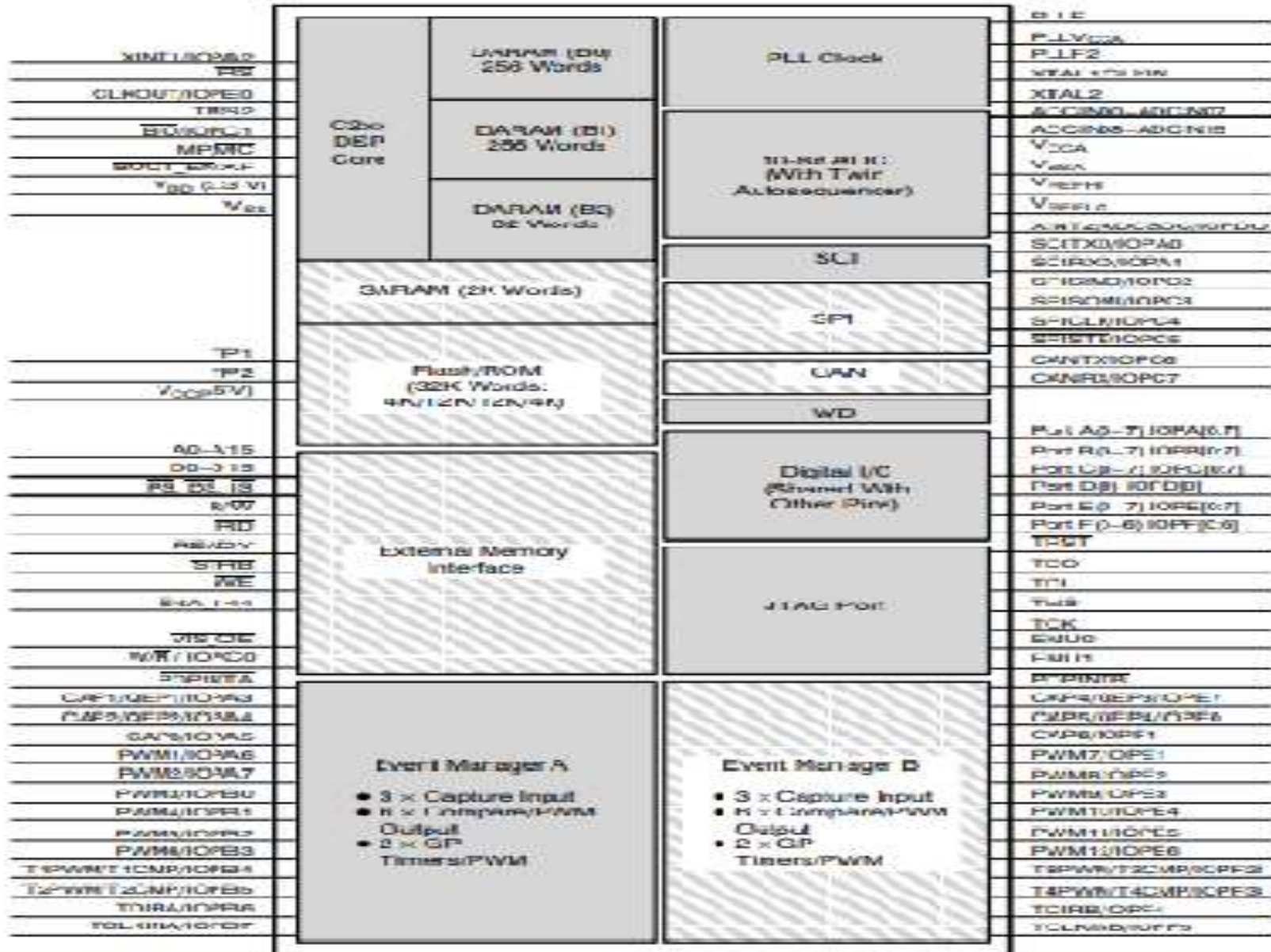
DSP

- Harvard Architecture
- VLIW/SIMD (parallel execution units)
- No bit level operations
- Hardware MACs
- DSP applications

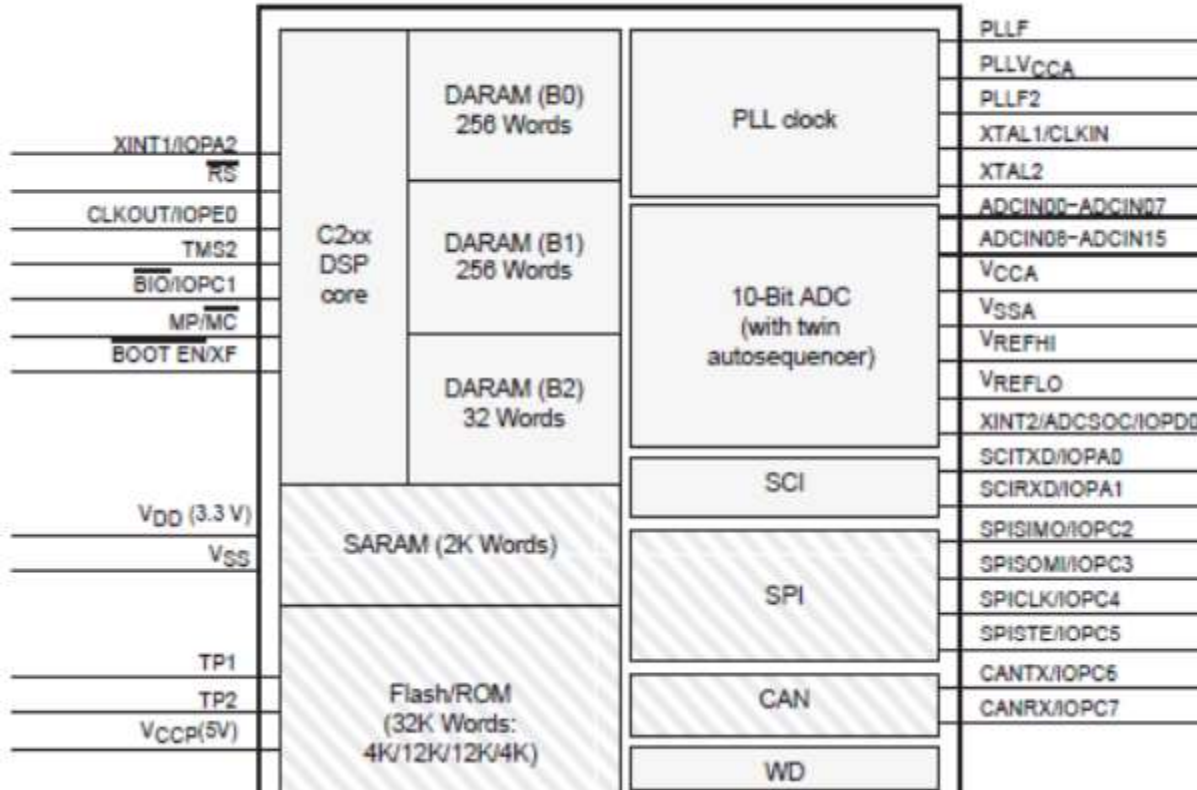
• Microcontroller

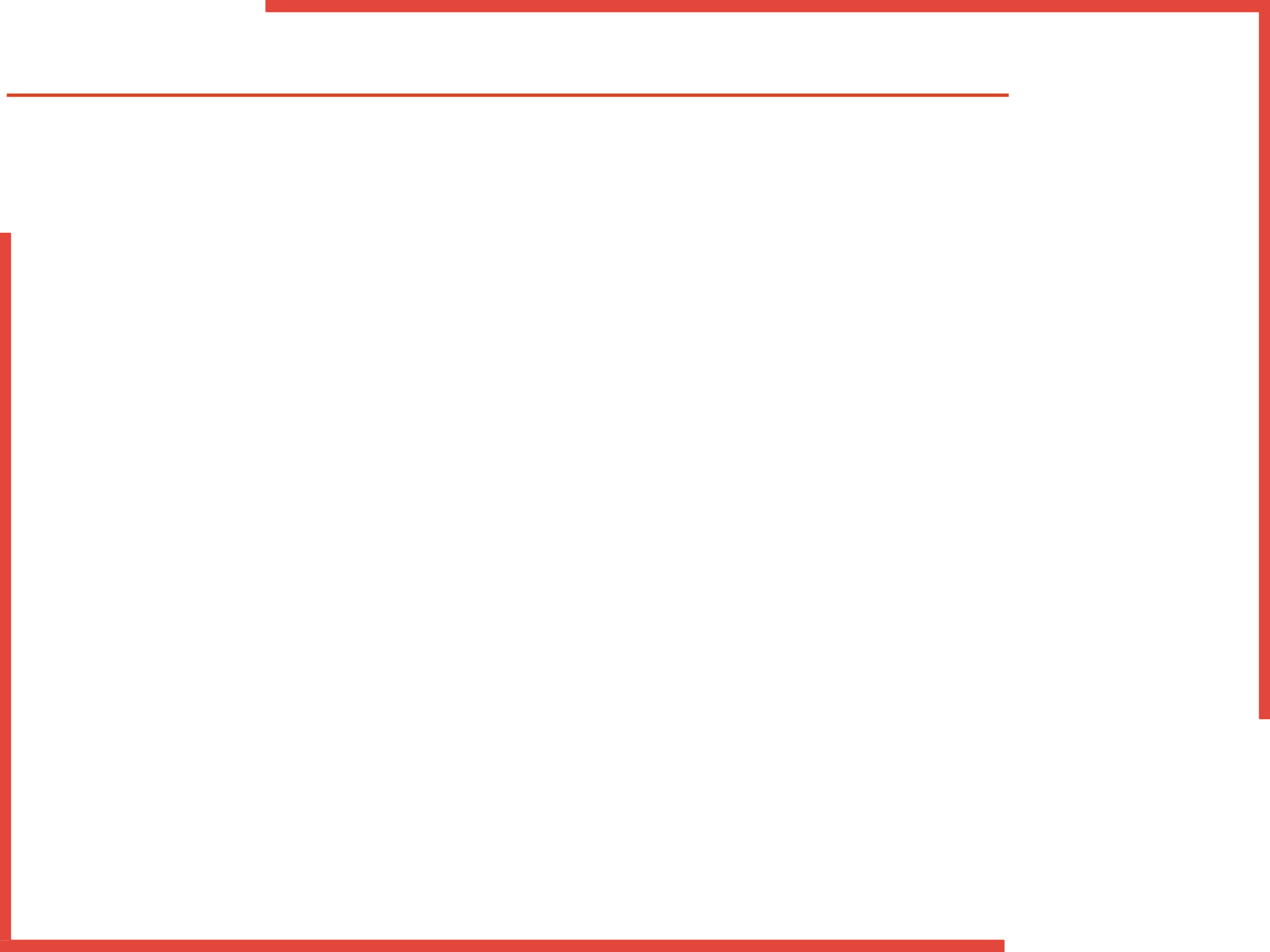
- Mostly von Neumann Architecture
- Single execution unit
- Flexible bit-level operations
- No hardware MACs
- Control applications

TMS320LF2407 DSP Controller functional diagram

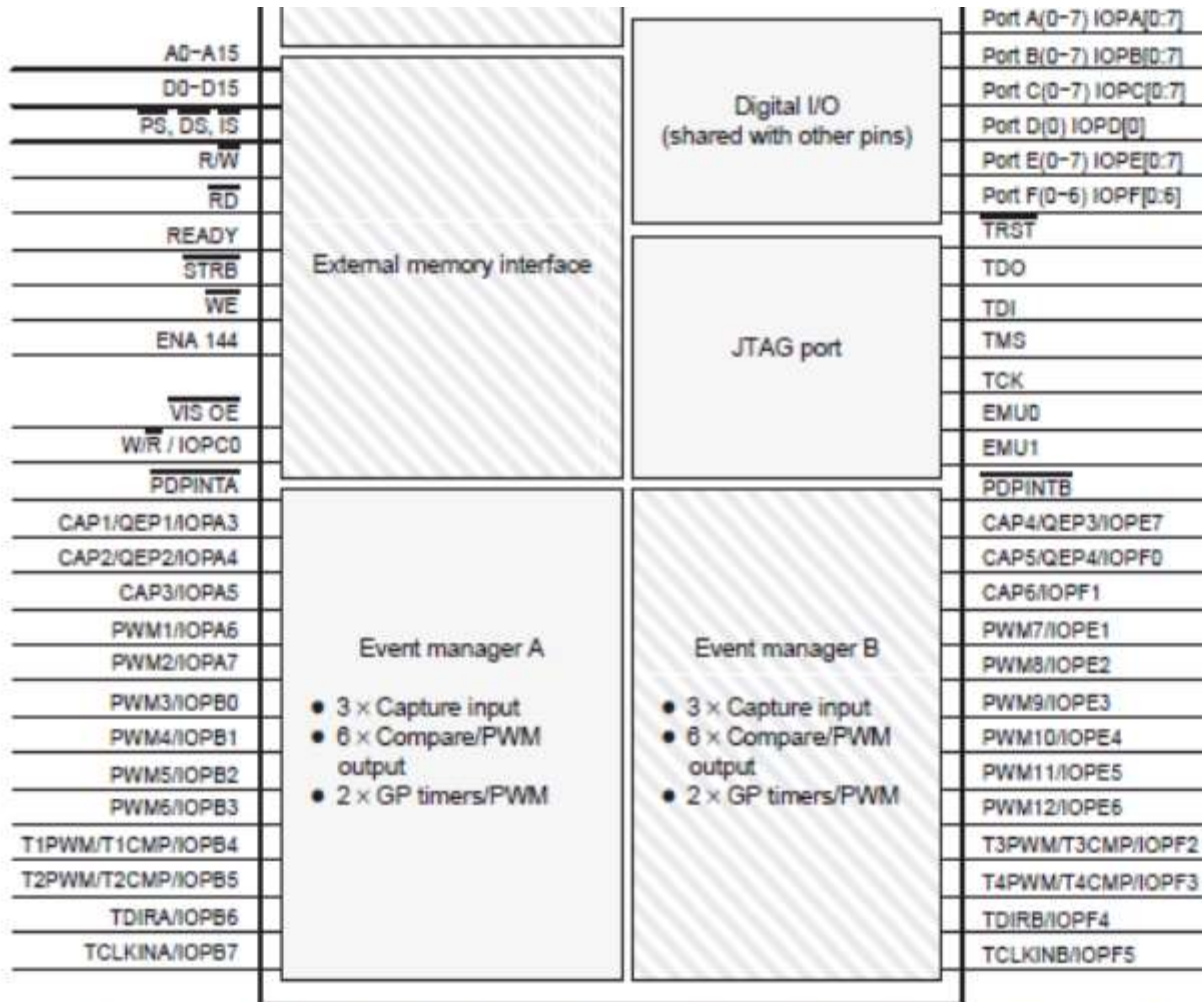


TMS320LF2407 DSP Controller functional diagram 1/2





TMS320LF2407 DSP Controller functional diagram 2/2



Indicates optional modules. The memory size and peripheral selection of these modules change for different

TMS320LF2407 DSP Controller

It is a C2xx core CPU for low-cost, low-power, and high-performance processing capabilities.

Several advanced peripherals, optimized for digital motor and motion control applications, have been integrated to provide a true single-chip DSP controller.

The 240xA offers increased processing performance (40 MIPS) and a higher level of peripheral integration.

Flash devices of up to 32K words offer a cost-effective reprogrammable solution for volume production.

The 240xA devices offer a password-based “code security” feature which is useful in preventing unauthorized duplication of proprietary code stored in on-chip Flash/ROM.

The 240xA family also includes ROM devices.

All 240xA devices offer at least one event manager module which has been optimized for digital motor control and power conversion applications.

Capabilities of this module include center- and/or edge-aligned PWM generation.

TMS320LF2407 DSP Controller

It prevent shoot-through faults, and synchronized analog-to-digital conversion. Devices with dual event managers enable multiple motor and/or converter control with a single 240xA DSP controller.

The high-performance, 10-bit analog-to-digital converter (ADC) has a minimum conversion time of 375 ns and offers up to 16 channels of analog input.

A serial communications interface (SCI) is integrated on all devices to provide asynchronous communication to other devices in the system.

It offer a controller area network (CAN) communications module.

JTAG-compliant scan-based emulation has been integrated into all devices.

A complete suite of code-generation tools from C compilers to the industry-standard Code Composer Studio debugger supports this family.

It is a member of the C2000 platform of fixed point DSPs.

The C3x and C4x floating-point DSPs

TMS320LF2407 DSP Controller

The LF2407A combines the high-performance CPU core with a set of peripherals acting as the “heavy artillery” to meet interfacing requirements for the most demanding of problems in terms of digital signal processing, communications and general purpose I/O operations.

The Event Managers, incorporating Timers and PWM generators.

The Controller Area Network (CAN) Module.

The Analog to Digital Converter.

The Serial Peripheral Interface (SPI) for synchronous serial communications.

The Serial Communications Interface (SCI) - asynchronous serial port (universal asynchronous receiver and transmitter – UART).

The Watchdog timer.

General bi-directional digital I/O (GPIO) pins.

the peripherals commonly employed are the event managers and the ADC.

TMS320LF2407 DSP Controller

The event managers.

These peripherals include a set of modules to facilitate creation of pulse width modulated signals and capture rising/falling edges in pulses.

In the “heart” of the event managers, lie the general purpose timers, providing clocking to the modules of the device; they may also be used to synchronize the occurrence of events in our programs.

The event managers are highly configurable to the last detail, with an extensive list of configuration/control registers.

The CAN module.

The controller area network module implements the multi-master CAN bus communications protocol interface with a set of six mailboxes.

TMS320LF2407 DSP Controller

The Analog to Digital Converter.

The ADC is used to sample analog signals and produce the corresponding digital value stored in a 10-bit integer result.

Arguably, it is one of the most significant peripherals on the LF2407A. It utilizes two sequencers as finite state machines that synchronize the sampling process.

The Serial Peripheral Interface.

The SPI is used for synchronous master-slave high speed serial communications. Typically, the SPI is used for communications with external devices such as LCDs and Digital to Analog Converters.

The Serial Communications Interface. The SCI implements typical asynchronous serial communications (UART). Typical applications of the SCI include communications with other controllers, or a PC. The designated lines for reception (RX) and transmission (TX) are not level-shifted on the DSP board (i.e., they operate at 0-3.3V).

TMS320LF2407 DSP Controller

The Watchdog Timer.

The watchdog is essentially a timer, acting as a safety precaution against possible program locks in endless loops.

When enabled, the watchdog increases an internal 8-bit counter using a clocking signal running at a sub-multiple frequency of the CPU clock signal.

The program should be able to reset the counter before an overflow occurs; if, for any reason (which may possibly be an execution “hung”), the program fails to reset the watchdog in time, the counter will overflow and a system reset will be asserted.

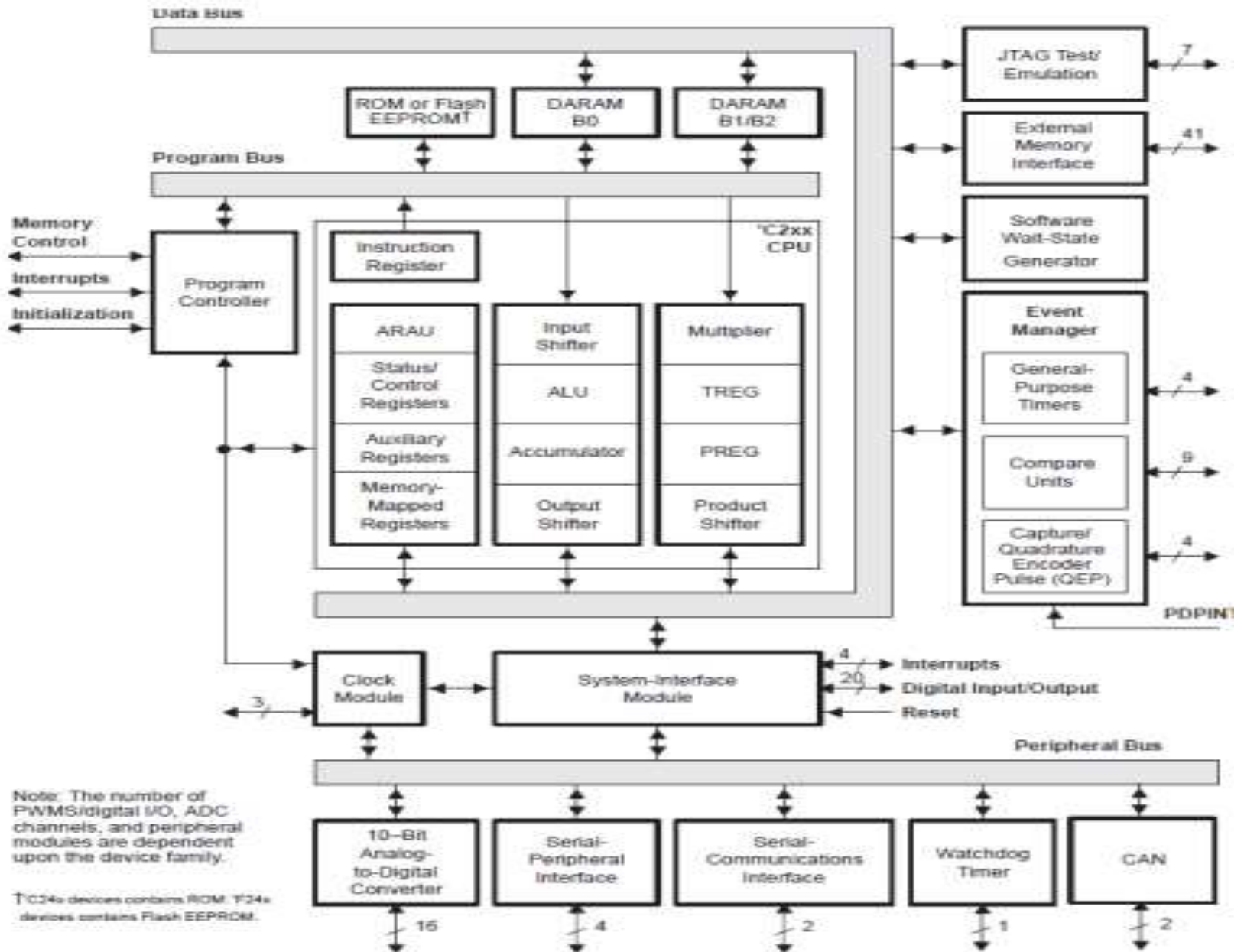
The General bi-directional I/O pins.

The LF2407A has a set of general I/O pins organized in ports A, B, C, D, E and F.

Most of the I/O pins on the LF2407A are multiplexed with other devices (e.g., general I/O pin A6 is multiplexed with the PWM1 pin) and must be configured prior to use, either for their primary (non - general I/O) or secondary (general I/O) function.

Moreover, general I/O pins can be configured either as input or output.

TMS320LF2407 DSP Controller



The 'C24x DSP controllers are designed to meet the needs of control-based applications.

- By integrating the high performance of a DSP core and the on-chip peripherals of a microcontroller into a single-chip solution, the 'C24x series yields a device that is an affordable alternative to traditional microcontroller units (MCUs) and expensive multichip designs.
- At 20 million instructions per second (MIPS), the 'C24x DSP controllers offer significant performance over traditional 16-bit microcontrollers and microprocessors.
 - The 16-bit, fixed-point DSP core of the 'C24x device provides analog designers a digital solution that does not sacrifice the precision and performance of their systems. The 'C24x DSP controllers offer reliability and programmability. Analog control systems, on the other hand, are hardwired solutions and can experience performance degradation due to aging, component tolerance, and drift.
- The high-speed central processing unit (CPU) allows the digital designer to process algorithms in real time rather than approximate results with look-up tables
- The 'C24x architecture is also well-suited for processing control signals.
- It uses a 16-bit word length along with 32-bit registers for storing intermediate results, and has two hardware shifters available to scale numbers independently of the CPU. This combination minimizes quantization and truncation errors, and increases processing power for additional functions. Two examples of these additional functions are: a notch filter that cancels mechanical resonances in a system, and an estimation technique that eliminates state sensors in a system

TMS320C24x Nomenclature

TMS – stands for qualified device

320 - TMS320 Family

C – CMOS technology

24x - device

C24x CPU Internal Bus Structure

C24x CPU Internal Bus Structure

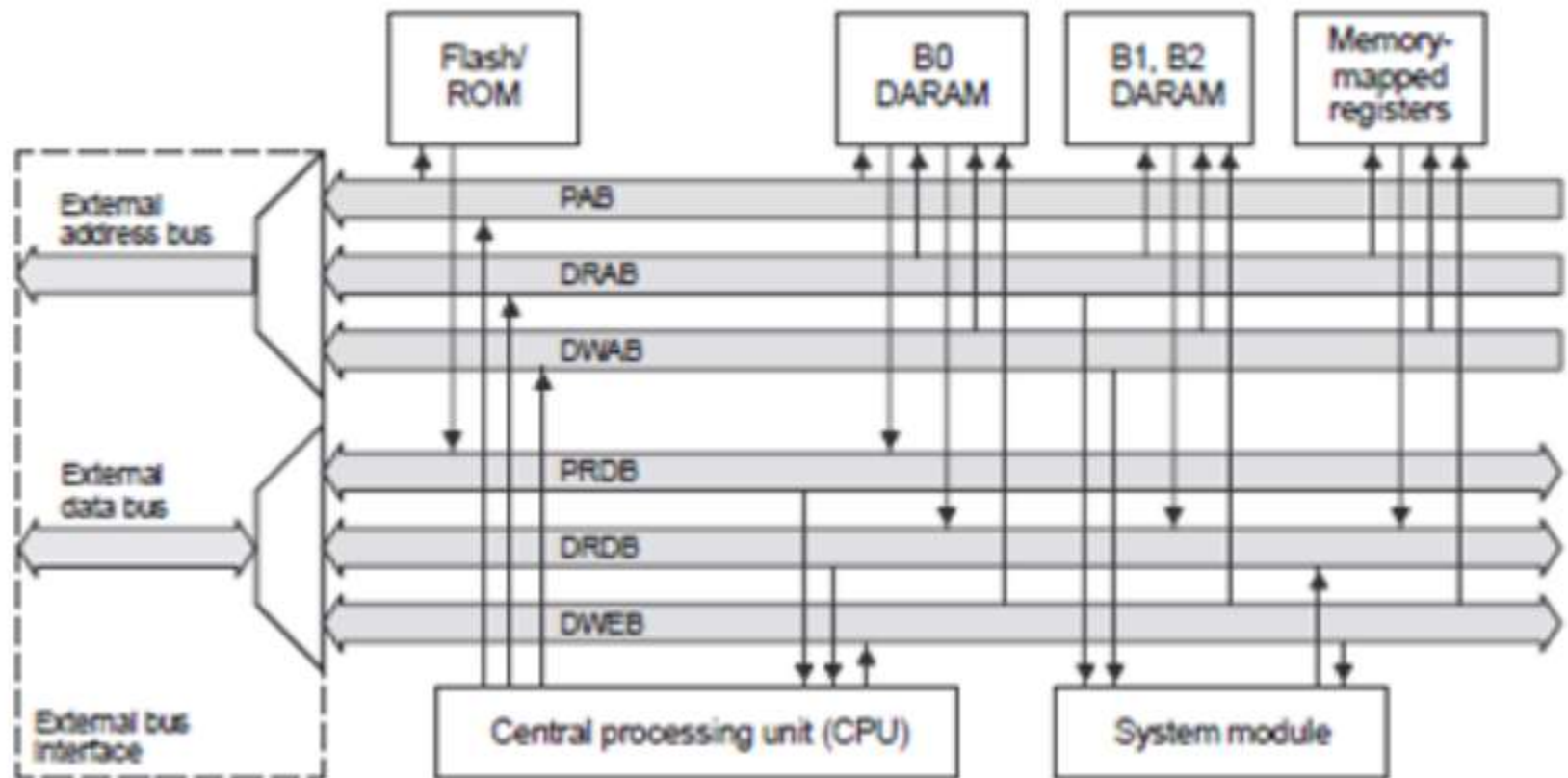
The 'C24x DSP, a member of the TMS320 family of DSPs, includes a 'C2xx DSP core designed using the '2xLP ASIC core.

The 'C2xx DSP core has an internal data and program bus structure that is divided into six 16-bit buses.

The six buses are:

- PAB. The program address bus provides addresses for both reads from and writes to program memory.
 - DRAB. The data-read address bus provides addresses for reads from data memory.
 - DWAB. The data-write address bus provides addresses for writes to data memory.
 - PRDB. The program read bus carries instruction code and immediate operands, as well as table information, from program memory to the CPU.
 - DRDB. The data-read bus carries data from data memory to the central arithmetic logic unit (CALU) and the auxiliary register arithmetic unit (ARAU).
 - DWEB. The data-write bus carries data to both program memory and data memory.
- Having separate address buses for data reads (DRAB) and data writes (DWAB) allows the CPU to read and write in the same machine cycle.

'C24x Address and Data Bus Structure



The 'C24x contains the following types of on-chip memory:

- Dual-access RAM (DARAM)
 - Flash EEPROM or ROM (masked)
- The 'C24x memory is organized into four individually-selectable spaces:
- Program (64K words)
 - Local data (64K words)
 - Global data (32K words)
 - Input/Output (64K words)
- These spaces form an address range of 224K words.

On-Chip Dual-Access RAM (DARAM)

- The 'C24x has 544 words of on-chip DARAM, which can be accessed twice per machine cycle. This memory is primarily intended to hold data, but when needed, can also be used to hold programs.
- The memory can be configured in one of two ways, depending on the state of the CNF bit in status register ST1. → When CNF = 0, all 544 words are configured as data memory. → When CNF = 1, 288 words are configured as data memory and 256 words are configured as program memory.
- Because DARAM can be accessed twice per cycle, it improves the speed of the CPU.
- The CPU operates within a 4-cycle pipeline. In this pipeline, the CPU reads data on the third cycle and writes data on the fourth cycle.

However, DARAM allows the CPU to write and read in one cycle; the CPU writes to DARAM on the master phase of the cycle and reads from DARAM on the slave phase.

For example, suppose two instructions, A and B, store the accumulator value to DARAM and load the accumulator with a new value from DARAM. Instruction A stores the accumulator value during the master phase of the CPU cycle, and instruction B loads the new value in the accumulator during the slave phase. Because part of the dual-access operation is a write, it only applies to RAM.

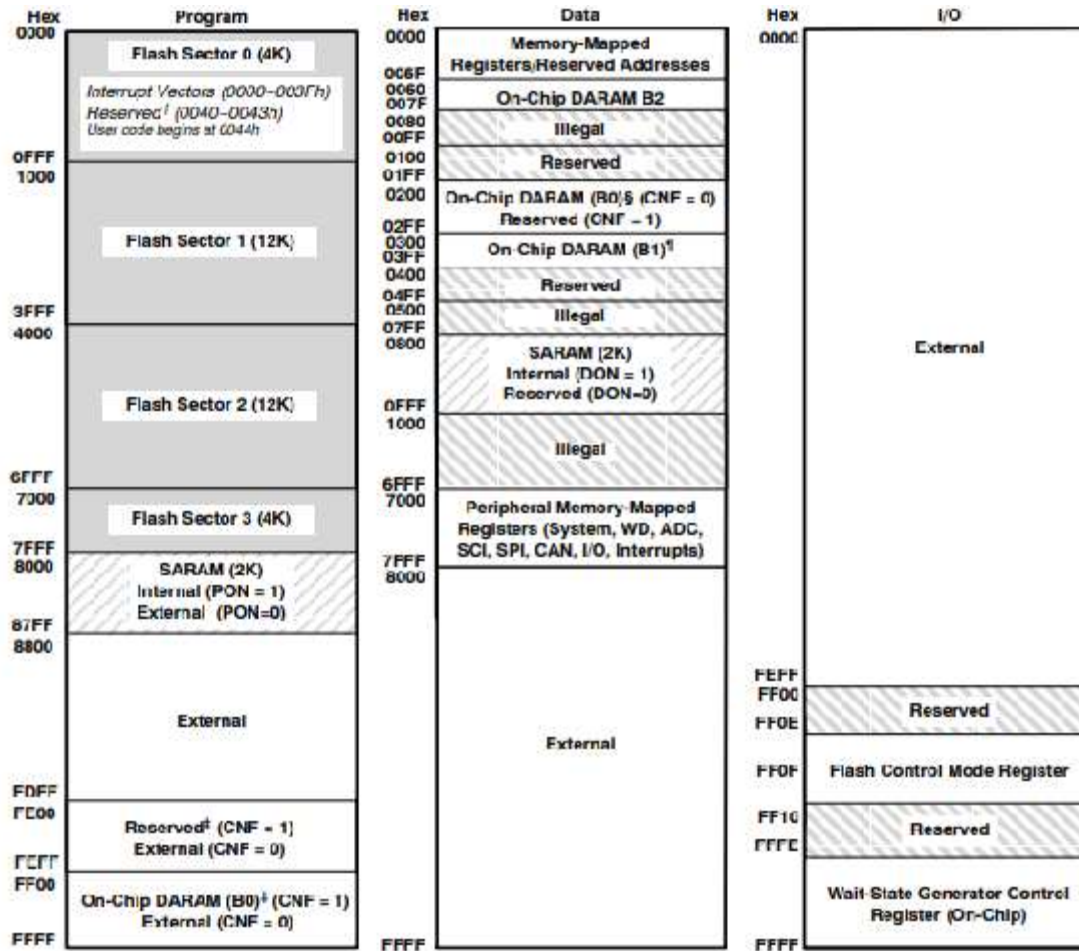
- Flash EEPROM provides an attractive alternative to masked program ROM
 - Like ROM, flash is a nonvolatile memory type; however, it has the advantage of in-target reprogrammability.
- The 'F24x incorporates one 16K/8K × 16-bit flash EEPROM module in program space.
 - This type of memory expands the capabilities of the 'F24x in the areas of prototyping, early field testing, and single-chip applications.
- Unlike most discrete flash memory, the 'F24x flash does not require a dedicated state machine because the algorithms for programming and erasing the flash are executed by the DSP core. This enables several advantages, including reduced chip size and sophisticated adaptive algorithms..
- Other key features of the flash include zero-wait-state access rate and single 5-V power supply. The following four algorithms are required for flash operations:
 - clear, erase, flash-write, and program.

ROM Memory MAP

'C24x ROM Memory Map



Memory maps



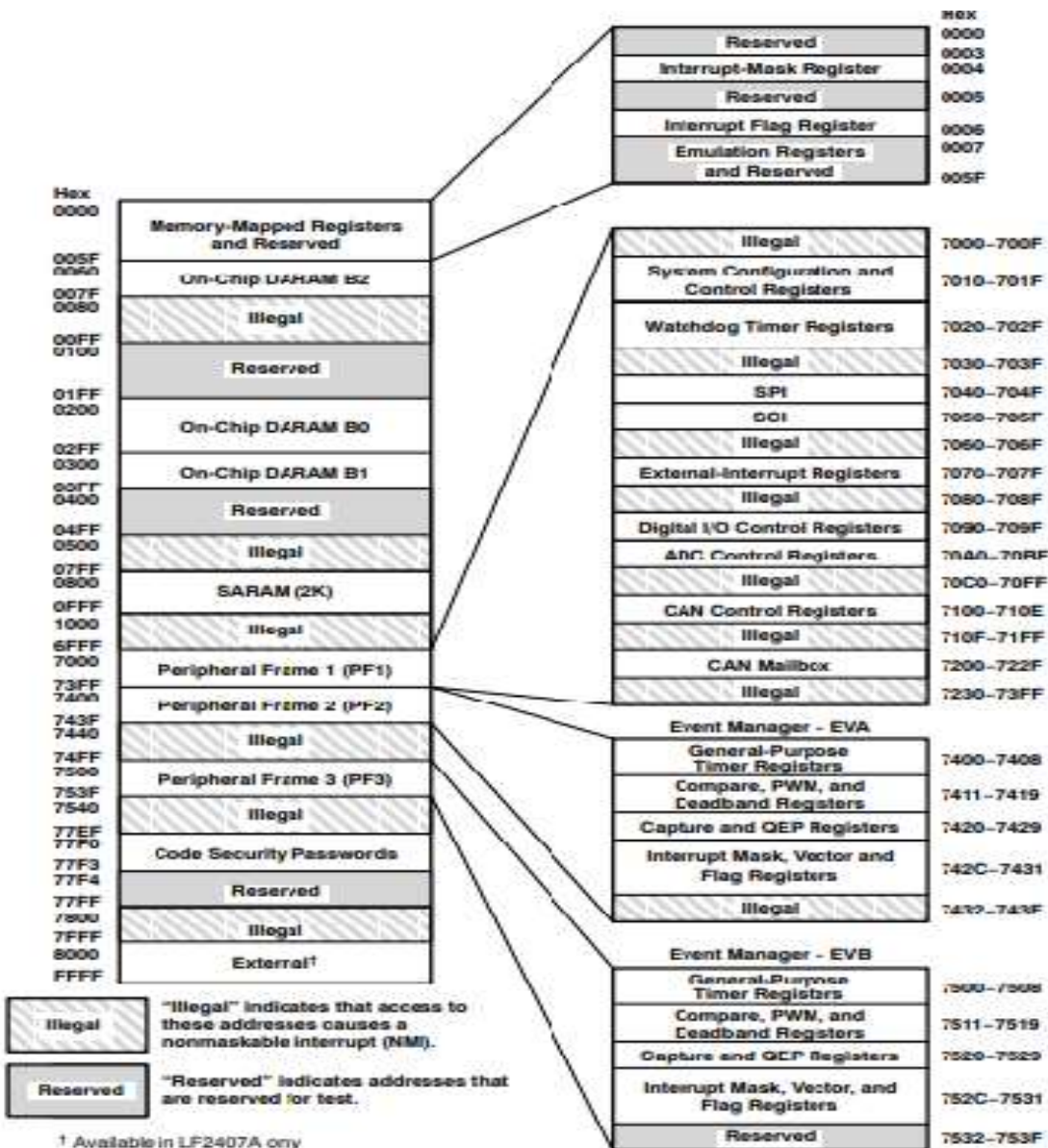
On-Chip Flash Memory (Sectored) - if MP/MC = 0
 External Program Memory - if MP/MC = 1
 SARAM (See Table 1 for details.)
 Reserved or Illegal

Memory map

Hex	Program
0000	On-Chip ROM (6K) Interrupt Vectors (0000-0007h) Reserved† (0040-0043h) User code begins at 0044h
17BF	Reserved
17C0	
17FF	
1800	
7FFF	Reserved
8000	
87FF	
8800	
FDFF	Reserved†
FE00	
FEFF	On-Chip DARAM (D0)‡ (CNF = 1) Reserved (CNF = 0)
FF00	
FFFF	

Hex	Data
0000	Memory-Mapped Registers, Reserved Addresses
005F	On-Chip DARAM D2
0060	
007F	Illegal
0080	
00FF	Reserved
0100	
01FF	On-Chip DARAM (B0)§ (CNF = 0) Reserved (CNF = 1)
0200	
02FF	On-Chip DARAM (B1)¶
0300	
03FF	Reserved
0400	
04FF	Illegal
0500	
07FF	Reserved
0800	
0FFF	Illegal
1000	
6FFF	Peripheral Memory-Mapped Registers (System, WD, ADC, SCI, I/O, Interrupts)
7000	
7FFF	Illegal
8000	
FFFF	

PERIPHERAL MEMORY MAP

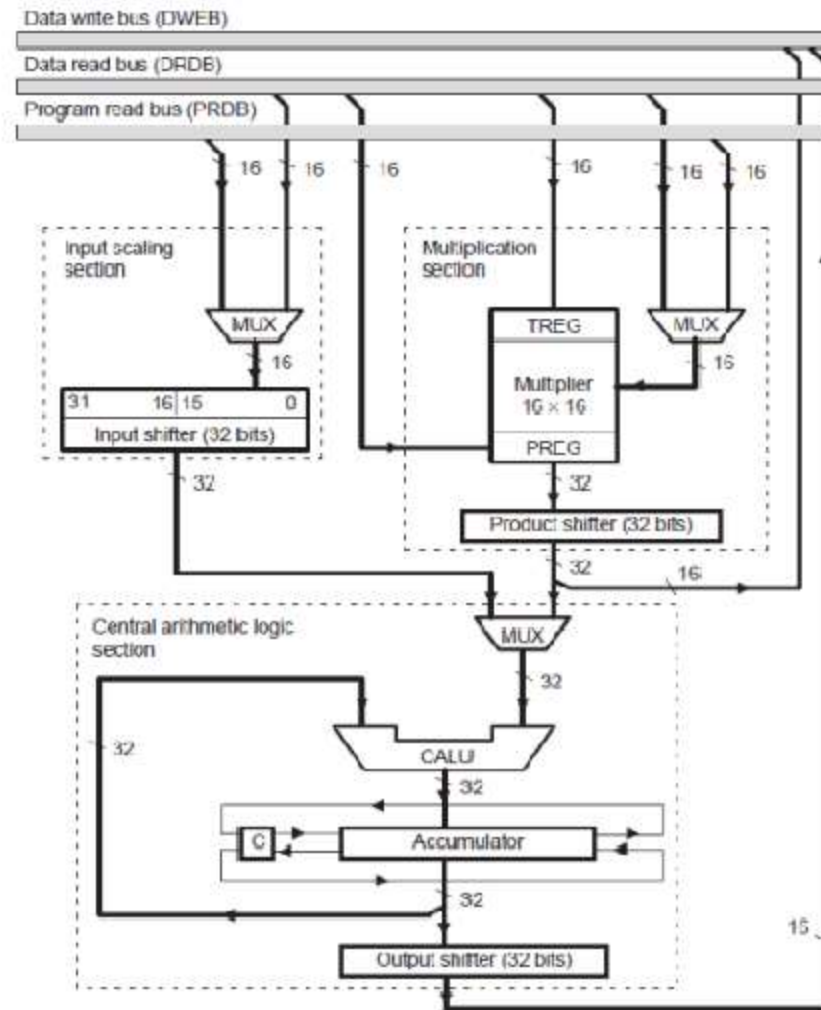


- In addition to full, on-chip memory support, some of the 'C24x devices provide access to external memory by way of the External Memory Interface Module.
- This interface provides 16 external address lines, 16 external data lines, and relevant control signals to select data, program, and I/O spaces. An on-chip wait-state generator allows interfacing with slower off-chip memory and peripherals.

A 32-bit central arithmetic logic unit (CALU)

- A 32-bit accumulator
- Input and output data-scaling shifters for the CALU
- A 16-bit × 16-bit multiplier
 - A product-scaling shifter
- Data-address generation logic, which includes eight auxiliary registers and an auxiliary register arithmetic unit (ARAU)
- Program-address generation logic

Central Processing Unit



Input Scaling Section

A 32-bit input data-scaling shifter (input shifter) aligns the 16-bit value from memory to the 32-bit central arithmetic logic unit (CALU).

This data alignment is necessary for data-scaling arithmetic, as well as aligning masks for logical operations.

The input shifter operates as part of the data path between program or data space and the CALU; and therefore, requires no cycle overhead.

Input. Bits 15 through 0 of the input shifter accept a 16-bit input from either of two

The data read bus (DRDB). This input is a value from a data memory location referenced in an instruction operand.

The program read bus (PRDB). This input is a constant value given as an instruction operand. Output. After a value has been accepted into bits 15 through 0, the input shifter aligns the 16-bit value to the 32-bit bus of the CALU

The shifter shifts the value left 0 to 16 bits and then sends the 32-bit result to the CALU.

Multiplication Section

The 'C24x uses a 16-bit \times 16-bit hardware multiplier that can produce a signed or unsigned 32-bit product in a single machine cycle.

The multiplication section consists of:

The 16-bit temporary register (TREG), which holds one of the multiplicands

The multiplier, which multiplies the TREG value by a second value from data memory or program memory .

The 32-bit product register (PREG), which receives the result of the multiplication

The product shifter, which scales the PREG value before passing it to the CALU

Multiplier

The 16-bit × 16-bit hardware multiplier can produce a signed or unsigned 32-bit product in a single machine cycle.

The two numbers being multiplied are treated as 2s-complement numbers, except during unsigned multiplication (MPYU instruction).

Descriptions of the inputs to, and output of, the multiplier

Inputs. The multiplier accepts two 16-bit inputs:

One input is always from the 16-bit temporary register (TREG). The TREG is loaded before the multiplication with a data-value from the data read bus (DRDB).

The other input is one of the following: _ A data-memory value from the data read bus (DRDB) _ A program memory value from the program read bus (PRDB) Output.

After the two 16-bit inputs are multiplied, the 32-bit result is stored in the product register (PREG).

The output of the PREG is connected to the 32-bit product-scaling shifter.

Through this shifter, the product is transferred from the PREG to the CALU or to data memory (by the SPH and SPL instructions).

Interrupts

The 'C24x DSP supports both hardware and software interrupts.

The hardware interrupts INT1 – INT6, along with NMI, TRAP, and RS, provide a flexible interrupt scheme.

The software interrupts offer flexibility to access interrupt vectors using software instructions.

Since most of the 'C24x DSPs come with multiple peripherals, the core interrupts (INT1–INT6) are expanded using additional system or peripheral interrupt logic.

Although the core interrupts are the same, the peripheral interrupt structure varies slightly among 'C240 and 'C24x class of DSP controllers.

CPU Interrupt Registers

There are two CPU registers for controlling interrupts:

The interrupt flag register (IFR) contains flag bits that indicate when maskable interrupt requests have reached the CPU on levels INT1 through INT6.

The interrupt mask register (IMR) contains mask bits that enable or disable each of the interrupt levels (INT1 through INT6).

The Reset interrupt Vector

If no boot ROM is present, then, following a reset, the DSP loads address 0x0000 (program memory) to the instruction pointer.

Address 0x0000 is the location for the reset vector and it should contain a branching instruction (jump) to whatever you want the DSP to do immediately after reset.

The LF2407A Core Interrupts .

The 2407A is capable of six (6) maskable interrupts and several software (TRAPS) and non-maskable interrupts (NMI). Practically, code will be dealing with the six maskable core interrupts (INT1-6), with the extreme exception of the occasional use of software interrupts.

The first 6 interrupts (except INT0 which is the reset interrupt vector) correspond to the peripherals of the 2407A through a peripheral interrupt expansion controller (will be discussed later), and it is important that the branching instructions in “cvector.asm” are pointing to the appropriate interrupt service routines (ISR) implemented in the C code.

INTERRUPTS

Interrupts are special events, normally triggered by external sources involving Peripherals.

you may choose to “interrupt” the sequential flow of execution and branch code execution to a special function that handles the event that triggered the interrupt. The special routines that handle interrupt signals are usually called interrupt handlers, but you will find the term interrupt service routines (ISR) rather more commonly used.

The 2407A is able to “sense” numerous interrupt sources, mainly related to its peripherals.

The Peripheral Interrupt Expansion Controller (PIE)

The 2407A acknowledges interrupts in two levels. The core itself provides six maskable interrupts (INT1-6). Technically, each of those interrupts may correspond to one specific source. When programming interrupts for a PC, we know that there is a one-to-one mapping from a peripheral interrupt source to a core interrupt in the CPU.

To overcome the problem of having a great number of hardware interrupts (as opposed to the six available maskable core interrupts) to be served by the CPU, these interrupts are organized in groups or levels, each one corresponding to one of the six core maskable interrupts (INT1-6). This is actually where the peripheral interrupt expansion controller (PIE) kicks-in. The PIE “intercepts” interrupt signals from the various peripherals and consequently triggers the appropriate core interrupt.

Addressing Modes

The three modes are:

Immediate addressing mode

Direct addressing mode

Indirect addressing mode

Immediate addressing mode

In the immediate addressing mode, the instruction word contains a constant to be manipulated by the instruction. The two types of immediate addressing modes are:

Short-immediate addressing Instructions that use short-immediate addressing have an 8-bit, 9-bit, or 13-bit constant as an operand.

Example.

`RPT #99` ;Execute the instruction that follows `RPT` ;100 times.

Long-immediate addressing mode

Instructions that use long-immediate addressing have a 16-bit constant as an operand and require two instruction words

`ADD #16384,2` ;Shift the value 16384 left by two bits ;and add the result to the accumulator.

Direct Addressing Mode

In the direct addressing mode, data memory is addressed in blocks of 128 words called data pages. The entire 64K of data memory consists of 512 data pages labeled 0 through 511.

The current data page is determined by the value in the 9-bit data page pointer (DP) in status register ST0.

For example, if the DP value is 0 0000 00002, the current data page is 0. If the DP value is 0 0000 00102, the current data page is 2.

DP Value	Offset	Data Memory
0000 0000 0	000 0000	Page 0: 0000h–007Fh
...	...	
0000 0000 0	111 1111	Page 1: 0080h–00FFh
0000 0000 1	000 0000	
...	...	
0000 0000 1	111 1111	Page 2: 0100h–017Fh
0000 0001 0	000 0000	
...	...	
0000 0001 0	111 1111	...
...	...	
...	...	
...	...	
...	...	
1111 1111 1	000 0000	Page 511: FF80h–FFFFh
...	...	
1111 1111 1	111 1111	

Indirect Addressing Mode

Eight auxiliary registers (AR0–AR7) provide flexible and powerful indirect addressing. Any location in the 64K data memory space can be accessed using a 16-bit address contained in an auxiliary register.

Indirect Addressing Options The 'C24x provides four types of indirect addressing options:

No increment or decrement. The instruction uses the content of the current auxiliary register as the data memory address but neither increments nor decrements the content of the current auxiliary register.

Increment or decrement by 1. The instruction uses the content of the current auxiliary register as the data memory address and then increments or decrements the content of the current auxiliary register by one.

Increment or decrement by an index amount.

The value in AR0 is the index amount. The instruction uses the content of the current auxiliary register as the data memory address and then increments or decrements the content of the current auxiliary register by the index amount.

The addition and subtraction process is accomplished with the carry propagation reversed for fast Fourier transforms (FFTs).

Assembly Language Instructions

Instruction Set Summary

This section provides six tables (Table 7–1 to Table 7–6) that summarize the instruction set according to the following functional headings:

Accumulator, arithmetic, and logic instructions
Auxiliary register and data page pointer instructions (see Table 7–2 on page 7-7)

TREG, PREG, and multiply instructions (see Table 7–3 on page 7-8)

Branch instructions (see Table 7–4 on page 7-9)

Control instructions (see Table 7–5 on page 7-10)

I/O and memory operations (see Table 7–6 on page 7-11)

definitions of the symbols used in the six summary tables:

ACC The accumulator
AR The auxiliary register
ARX A 3-bit value used in the LAR and SAR instructions to designate which auxiliary register will be loaded (LAR) or have its contents stored (SAR)

BITX A 4-bit value (called the bit code) that determines which bit of a designated data memory value will be tested by the BIT instruction.

CM A 2-bit value. The CMPR instruction performs a comparison specified by the value of CM:

If CM = 00, test whether current AR = AR0

If CM = 01, test whether current AR < AR0

If CM = 10, test whether current AR > AR0

If CM = 11, test whether current AR p AR0

Definitions of the symbols used in the six summary tables:

I AAA AAAA (One I followed by seven As)

The I at the left represents a bit that reflects whether direct addressing (I = 0) or indirect addressing (I = 1) is being used.

When direct addressing is used, the seven As are the seven least significant bits (LSBs) of a data memory address.

For indirect addressing, the seven As are bits that control auxiliary register manipulation,

IIII IIII (Eight Is) An 8-bit constant used in short immediate addressing

I IIII IIII (Nine Is) A 9-bit constant used in short immediate addressing for the LDP instruction

I IIII IIII IIII (Thirteen Is) A 13-bit constant used in short immediate addressing for the MPY instruction.

I NTR# A 5-bit value representing a number from 0 to 31.

The INTR instruction uses this number to change program control to one of the 32 interrupt vector addresses.

PM A 2-bit value copied into the PM bits of status register ST1 by the SPM instruction

SHF A 3-bit left-shift value

SHFT A 4-bit left-shift value

TP A 2-bit value used by the conditional execution instructions to represent four conditions

BIO pin low TP = 00

TC bit = 1 TP = 01, TC bit = 0 TP = 10 and for No condition TP = 11

DIGITAL RESOURCES

❖ [Lecture Notes –](#)

<https://classroom.google.com/c/NDc1MzQ3NDk1MDM5/m/NDkzMzE5NzI0OTEz/details>

❖ Vidéo Lectures –
watch?v=GapjjO_8Kuk

THANK YOU

DEPT & SEM : EEE & III/II SEM

SUBJECT NAME: DIGITAL COMPUTER PLATFORM

COURSE CODE: 19A02601T

UNIT : V

PREPARED BY : C MUNIKANTHA

OUTLINE

Introduction to Field Programmable Gate Arrays

CPLD Vs FPGA

Types of FPGA

Xilinx, XC3000 series

Configurable logic Blocks (CLB)

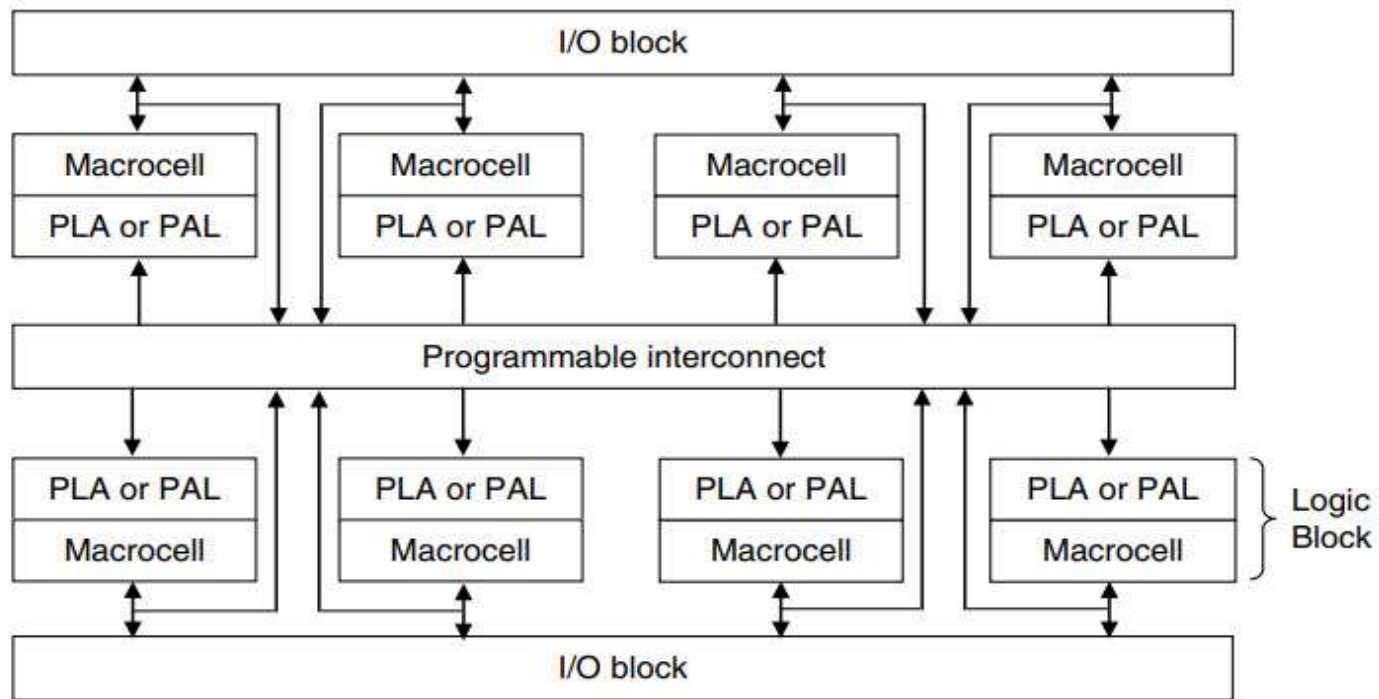
Input / Output Block (IOB)

Programmable Interconnect Point (PIP)

Xilinx 4000 series – HDL programming

overview of Spartan 3E and Virtex II pro FPGA boards- case study

Field Programmable Gate Arrays



FPGA stands for **Field Programmable Gate Array**. It is a semiconductor device that consists of a matrix of configurable logic blocks connected using programmable interconnects. It is possible to reprogram an FPGA according to the requirements after manufacturing. There are around 330000 logic blocks with 1100 inputs and outputs in modern FPGAs.

Programmable Logic Devices (PLD)

Programmable Logic Devices PLDs.

PLDs are the integrated circuits. They contain an array of AND gates & another array of OR gates. There are three kinds of PLDs based on the type of arrays, which has programmable feature.

Programmable Read Only Memory

Programmable Array Logic

Programmable Logic Array

The process of entering the information into these devices is known as **programming**. Basically, users can program these devices or ICs electrically in order to implement the Boolean functions based on the requirement. Here, the term programming refers to hardware programming but not software programming.

Field Programmable Gate Arrays

The architecture of an FPGA is completely different as it consists of

programmable Logic Cells,
programmable interconnects
and programmable IO blocks.

Field Programmable Gate Arrays or FPGAs in short are pre-fabricated Silicon devices that consists of a matrix of reconfigurable logic circuitry and programmable interconnects arranged in a two-dimensional array.

The programmable Logic Cells can be configured to perform any digital function and the programmable interconnects (or switches) provide the connections among different logic cells.

Using an FPGA, you can implement any custom design by specifying the logic or function of each logic block and setting the connection of each programmable switch. Since this process of designing a custom circuit is done in the field rather than in a fab, the device is known as “Field Programmable”.

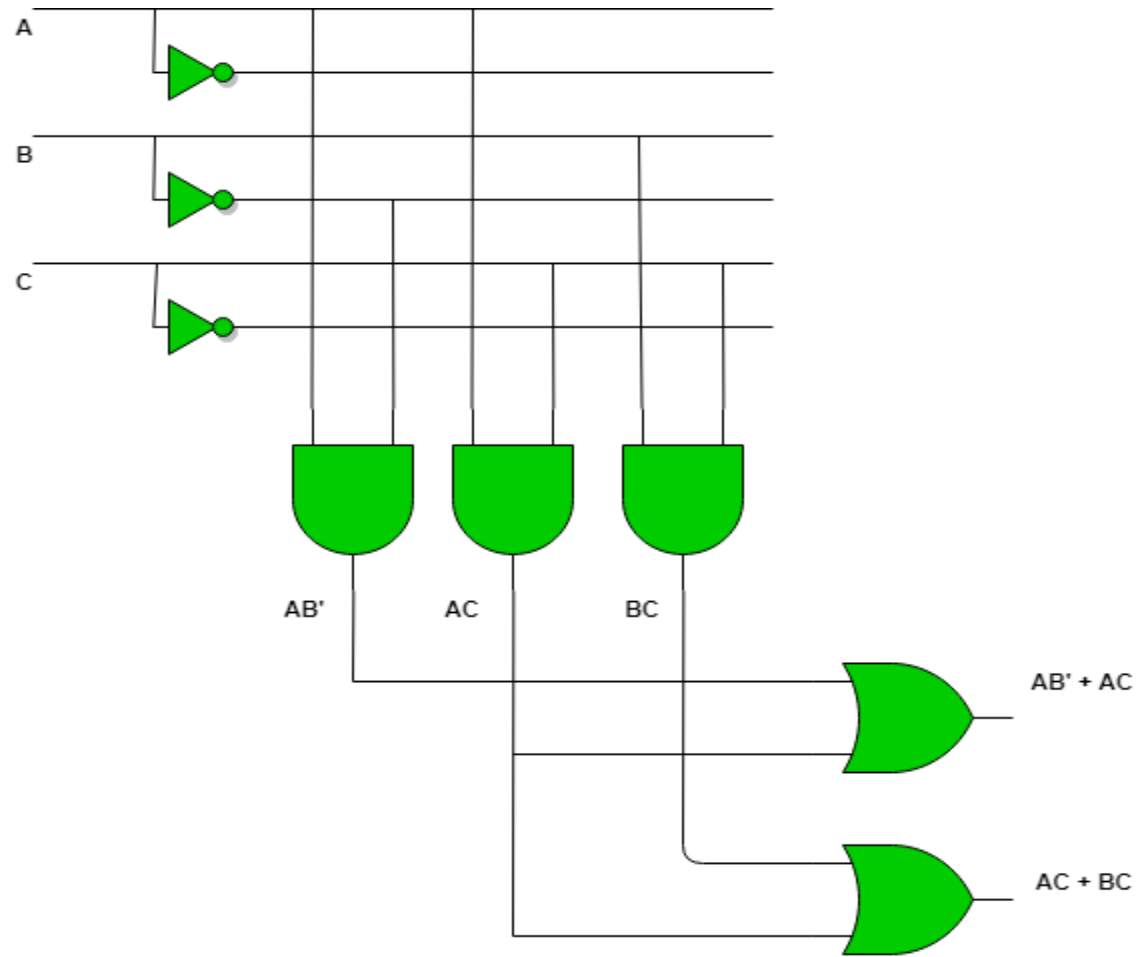
An FPGA consists of three basic components. They are:

Programmable Logic Cells (or Logic Blocks) – responsible for implementing the core logic functions.

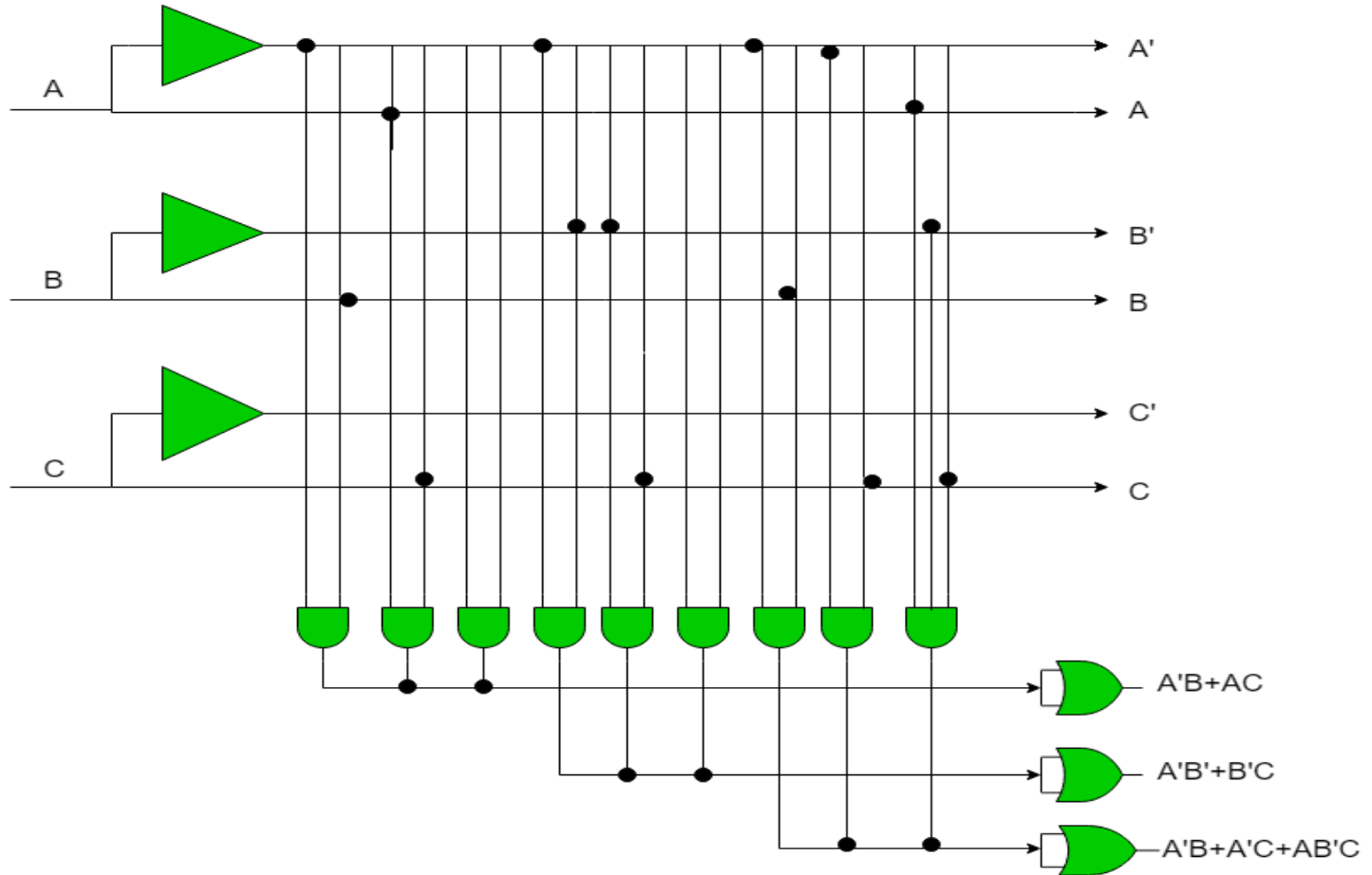
Programmable Routing – responsible for connecting the Logic Blocks.

IO Blocks – which are connected to the Logic Blocks through the routing and help to make external connections

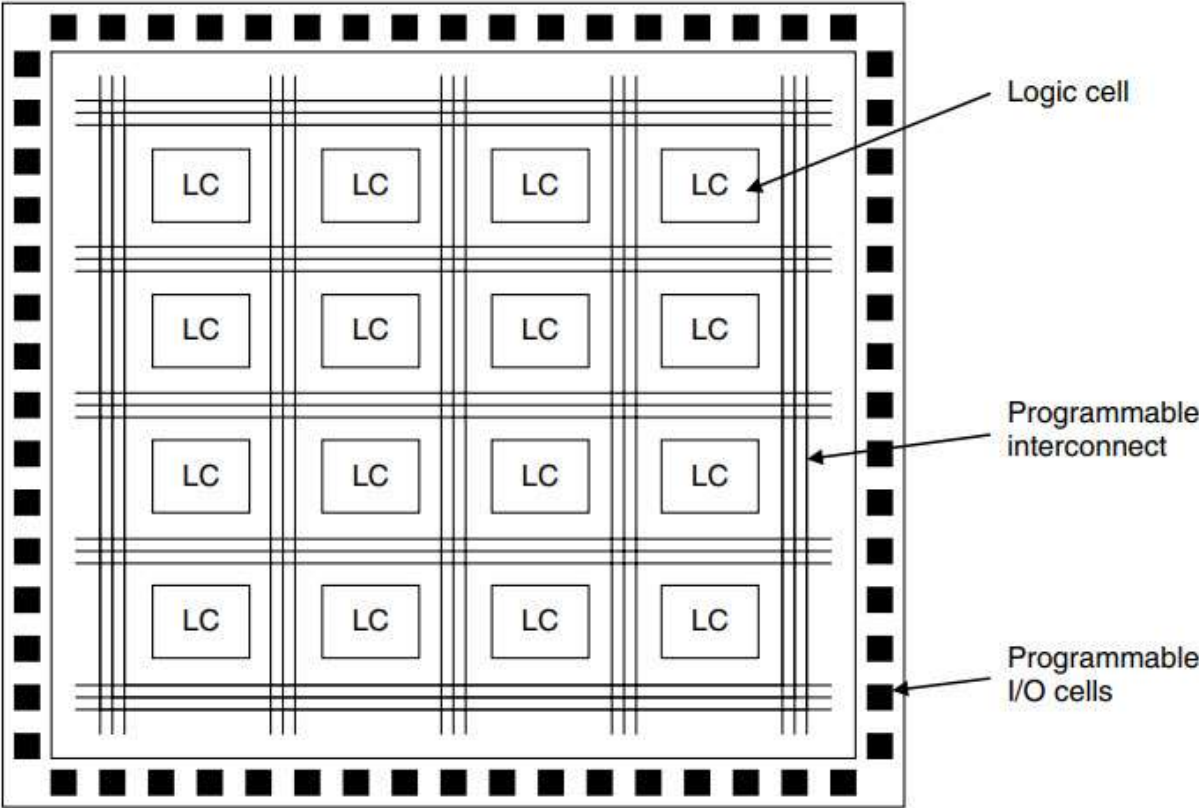
PLA (Programmable logic array)



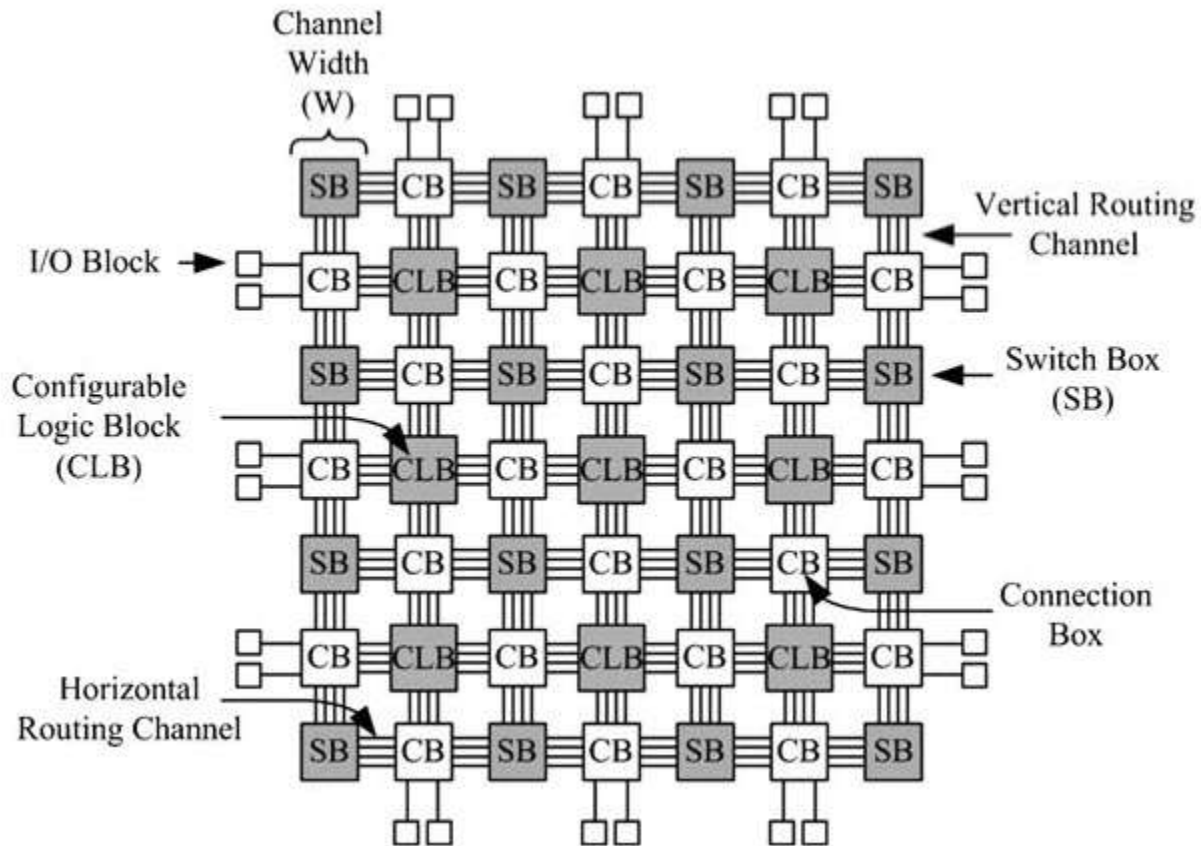
PAL (Programmable array logic)



FPGA



FPGA



CPLD versus FPGA

FPGAs and CPLDs are two of the well-known types of digital logic chips. When it comes to the internal architecture, the two chips are obviously different.

FPGA is short for Field-Programmable Gate Array, is a type of a programmable logic chip. It is great chip as it can be programmed to do almost any kind of digital function. FPGA's architecture allows the chip to have a very high logic capacity. It is used in designs that require a high gate count and their delays are quite unpredictable because of its architecture.

The FPGA is considered as 'fine-grain' because it contains a lot of tiny logic blocks that could reach up to 100,000. It is with flip-flops, combination logic, and On the other hand, CPLD (Complex Programmable Logic Device) is designed by using EEPROM (electrically erasable programmable read-only memory) . It is more suitable in small gate count designs. Since it is a less complex architecture, the delays are much predictable and it is non-volatile.

CPLD versus FPGA

CPLD is often used for simple logic applications. It contains only a few blocks of logic and reaches up to 100. Having said that, CPLDs are considered as 'coarse-grain' type of devices. CPLDs are cheap and it also offers a much faster input to output duration because of its simpler, 'coarse grain' architecture.

FPGAs are cheaper per gate but expensive when it comes to package.

Working with FPGAs requires special procedures as it is RAM based. To program the device, you have to first describe the 'logic function' with the use of computer, either by drawing a schematic or simply describing the function on a text file.

Compilation of the 'logic function' usually requires a software. It creates a binary file to be downloaded into the FPGA and then the chip will behave just what you have instructed in the 'logic function'.

CPLD versus FPGA

1. FPGA contains up to 100,000 of tiny logic blocks while CPLD contains only a few blocks of logic that reaches up to a few thousands.
2. In terms of architecture, FPGAs are considered as 'fine-grain' devices while CPLDs are 'coarse-grain'.
3. FPGAs are great for more complex applications while CPLDs are better for simpler ones.
4. FPGAs are made up of tiny logic blocks while CPLDs are made of larger blocks.
5. FPGA is a RAM-based digital logic chip while CPLD is EEPROM-based.
6. Normally, FPGAs are more expensive while CPLDs are much cheaper.
7. Delays are much more predictable in CPLDs than in FPGAs.

Types of FPGA

There are two basic types of FPGAs:

SRAM-based reprogrammable (Multi-time Programmed MTP)

and (One Time Programmed) OTP.

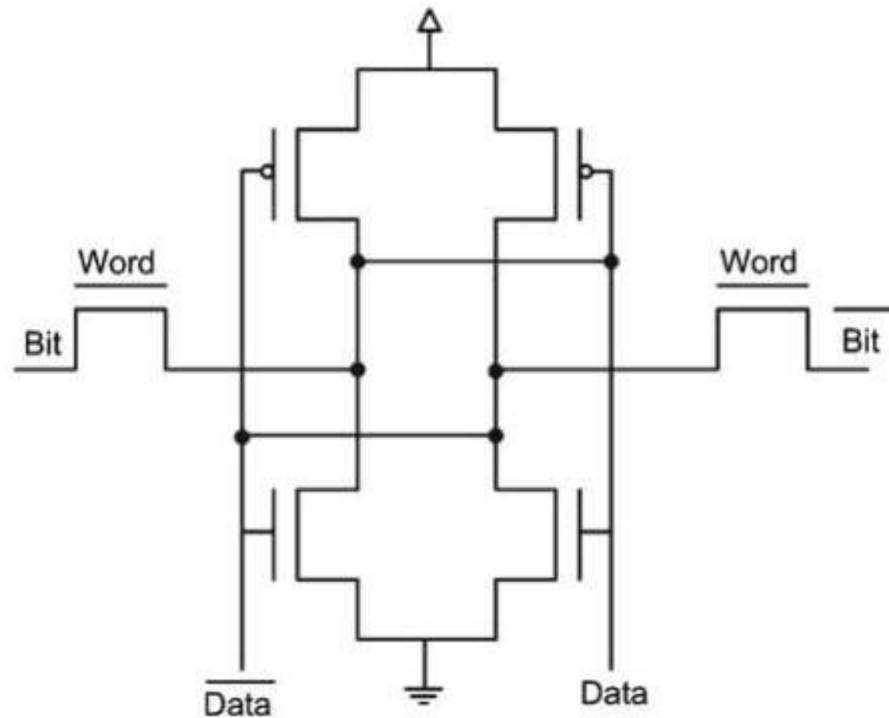
These two types of FPGAs differ in the implementation of the logic cell and the mechanism used to make connections in the device.

The dominant type of FPGA is SRAM-based and can be reprogrammed as often. In fact, an SRAM FPGA is reprogrammed every time it's powered up, because the FPGA is really a fancy memory chip. That's why you need a serial PROM or system memory with every SRAM FPGA

SRAM

A typical 6 transistor SRAM Cell to store 1 bit is shown in the following image.

SRAM is designed using transistors and the term static means that the value loaded on a basic SRAM Memory Cell will remain the same until deliberately changed or when the power is removed.



Types of FPGA

Property	OTP FPGA	MTP FPGA
Speed	smaller	larger
Power Consumption	lower	higher
Working Environment (Radiation)	Radiation hardened	NO radiation hardened
Design Cycle	Programmed once only	Many times
Price	Almost the same	Almost the same
Reliability	More (single Chip)	Less (2 Chips, FPGA & PROM)
Security	More secure	Less secure

Xilinx,

***Xilinx* is the inventor of the FPGA, programmable SoCs, and now, the ACAP.**

***Xilinx* delivers the most dynamic processing technology in the industry.**

Xilinx, Inc. (/ˈzɪlɪŋks/ ZEE-links) was an American technology and semiconductor company that primarily supplied programmable logic devices. The company was known for inventing the first commercially viable field-programmable gate array (FPGA) and creating the first fabless manufacturing model, Xilinx was co-founded by Ross Freeman Bernard Vonderschmitt and James V Barnett II in 1984 and the company went public on the NASDAQ in 1989.

AMD announced its acquisition of Xilinx in October 2020 and the deal was completed on February 14, 2022 through an all-stock transaction worth an estimated \$50 billion. Before 2010, Xilinx offered two main FPGA families: the high-performance Virtex series and the high-volume Spartan series, with a cheaper EasyPath option for ramping to volume production.

The company also provides two CPLD lines: the CoolRunner and the 9500 series. Each model series has been released in multiple generations since its launch. With the introduction of its 28 nm FPGAs in June 2010, Xilinx replaced the high-volume Spartan family with the Kintex family and the low-cost Artix family.

XC3000 series

Complete line of four related Field Programmable Gate Array product families - XC3000A, XC3000L, XC3100A, XC3100L

- Ideal for a wide range of custom VLSI design tasks - Replaces TTL, MSI, and other PLD logic - Integrates complete sub-systems into a single package - Avoids the NRE, time delay, and risk of conventional masked gate arrays
 - High-performance CMOS static memory technology - Guaranteed toggle rates of 70 to 370 MHz, logic delays from 7 to 1.5 ns - System clock speeds over 85 MHz - Low quiescent and active power consumption
- Flexible FPGA architecture - Compatible arrays ranging from 1,000 to 7,500 gate complexity - Extensive register, combinatorial, and I/O capabilities - High fan-out signal distribution, low-skew clock nets - Internal 3-state bus capabilities - TTL or CMOS input thresholds - On-chip crystal oscillator amplifier

XC3000 series

- Unlimited reprogrammability - Easy design iteration - In-system logic changes
 - Extensive packaging options - Over 20 different packages - Plastic and ceramic surface-mount and pin-gridarray packages - Thin and Very Thin Quad Flat Pack (TQFP and VQFP) options
 - Ready for volume production - Standard, off-the-shelf product availability - 100% factory pre-tested devices - Excellent reliability record
- Complete Development System - Schematic capture, automatic place and route - Logic and timing simulation - Interactive design editor for design optimization - Timing calculator - Interfaces to popular design environments like Viewlogic, Cadence, Mentor Graphics, and others

Configurable logic Blocks (CLB)

A configurable logic block (CLB) is the basic repeating logic resource on an FPGA. When linked together by routing resources, the components in CLBs execute complex logic functions, implement memory functions, and synchronize code on the FPGA.

CLBs contain smaller components, including flip-flops, look-up tables (LUTs), and multiplexers

Flip-Flop—A circuit capable of two stable states that represents a single bit. A flip-flop is the smallest storage resource on the FPGA. Each flip-flop in a CLB is a binary register used to save logic states between clock cycles on an FPGA circuit.

Look-up Table (LUT)—A collection of gates hardwired on the FPGA. An LUT stores a predefined list of outputs for every combination of inputs. LUTs provide a fast way to retrieve the output of a logic operation because possible results are stored and then referenced rather than calculated. The LUTs in a CLB can also implement FIFOs and memory items in LabVIEW.

Multiplexer—A circuit that selects between two or more inputs and then returns the selected input.

Configurable logic Blocks (CLB)

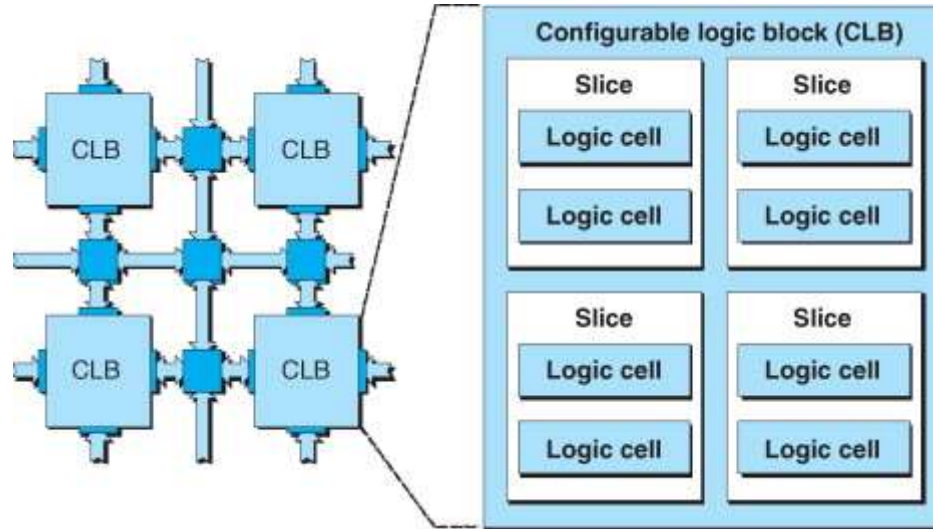


Figure Configurable logic Blocks

To run on an FPGA target, LabVIEW implements much of the code using flip-flops, LUTs, and multiplexers.

Input / Output Block (IOB)

The input/output block (IOB) is used for communication between the problem program and the system.

It provides the addresses of other control blocks, and maintains information about the channel program, such as the type of chaining and the progress of I/O operations.

First define the IOB and specify its address as the only parameter of the EXCP or EXCPVR macro instruction.

The input/output block (IOB) is not automatically constructed by a macro instruction; it must be defined as a series of constants and be on a word boundary.

For unit-record and tape devices, the IOB is 32 bytes long.

For direct access, teleprocessing, and graphic devices, 8 additional bytes must be provided. Use the system mapping macro IEZIOB, which expands into a DSECT, to help in constructing an IOB.

Input / Output Block (IOB)

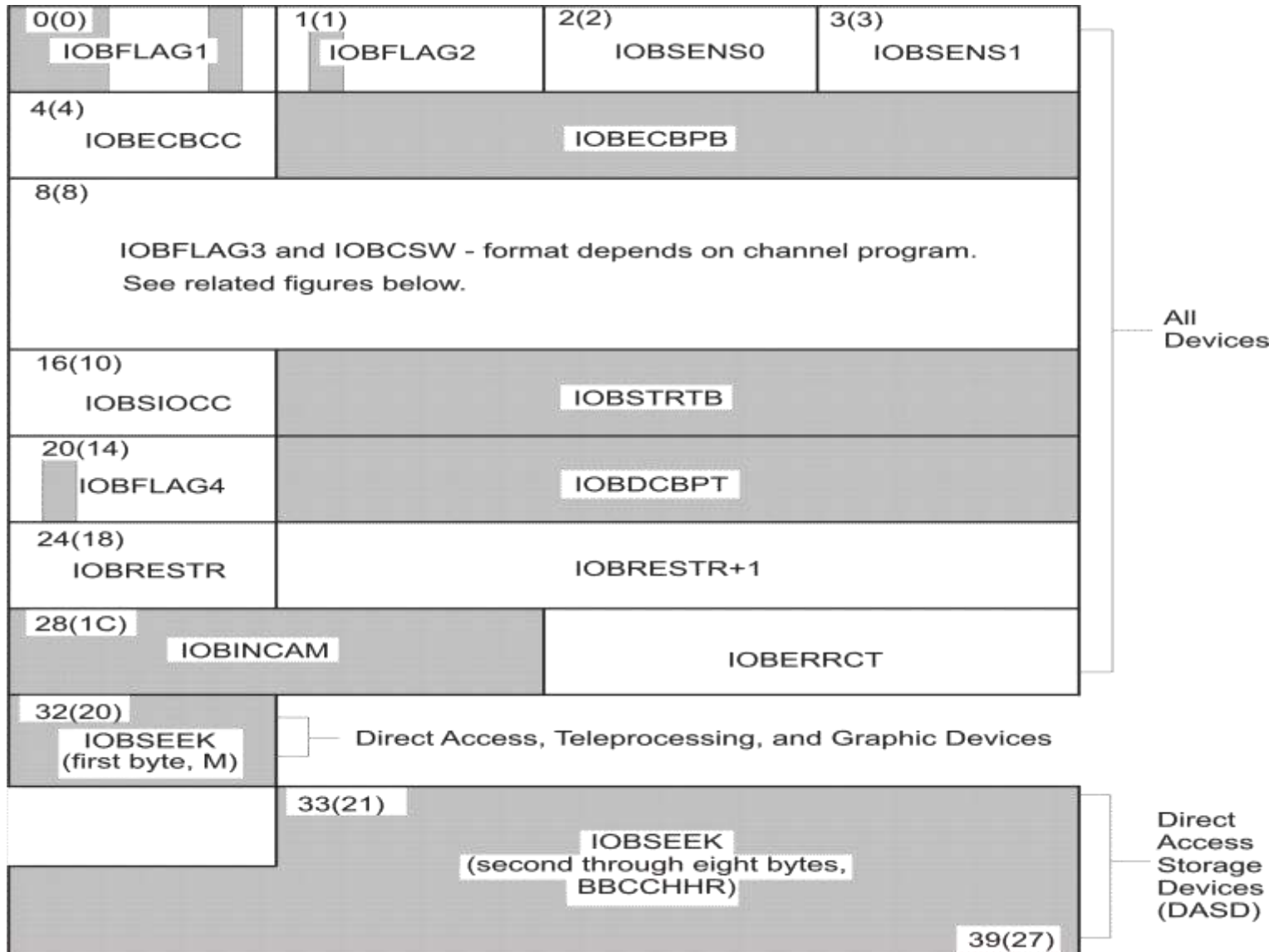


Figure Input/Output Block (IOB) Format

Input / Output Block (IOB)

IOBFLAG1 (1 byte)

Set bit positions 0, 1, 6, and 7. One-bits in positions 0 and 1 (IOBDATCH and IOBCMDCH) indicate data chaining and command chaining, respectively. (If you specify both data chaining and command chaining, the system does not use error recovery routines except for the direct access and tape devices.) If an I/O error occurs while your channel program executes, a failure to set the chaining bits in the IOB that correspond to those in the CCW might make successful error recovery impossible. The integrity of your data could be compromised.

A one-bit in position 6 (IOBUNREL) indicates that the channel program is not a related request; that is, the channel program is not related to any other channel program. See bits 2 and 3 of IOBFLAG2 below.

If you intend to issue an EXCP or XDAP macro with a BSAM, QSAM, or BPAM DCB, you should turn on bit 7 (IOBSPSVC) to prevent access-method appendages from processing the I/O request.

Input / Output Block (IOB)

IOBFLAG2 (1 byte)

If you set bit 6 in the IOBFLAG1 field to zero, bits 2 and 3 (IOBRRT3 and IOBRRT2) in this field must then be set to one of the following:

00, if any channel program or appendage associated with a related request might modify this IOB or channel program.

01, if the conditions requiring a 00 setting do not apply, but the CHE or ABE appendage might retry this channel program if it completes normally or with the unit-exception or wrong-length-record bits on in the CSW.

10 in all other cases.

The combinations of bits 2 and 3 represent related requests, known as type 1 (00), type 2 (01), and type 3 (10). The type you use determines how much the system can overlap the processing of related requests. Type 3 allows the greatest overlap, normally making it possible to quickly reuse a device after a channel-end interruption. (Related requests that were executed on a pre-MVS system are executed as type-1 requests if not modified.)

Programmable Interconnect Point (PIP)

Field Programmable Gate Arrays (FPGA) are very interesting integrated circuits. The possibility of completing different tasks by just reprogramming the FPGA made us think at first view it was a kind of microcontroller. We were far from the reality. FPGAs are reprogrammable logic/memory circuits and can be faster than any microcontroller. The main difference is that a microcontroller has a program written in memory, and a FPGA only has programmed connections/cells, so the data follows a continuous way through programmed logic and memory cells, instead of being processed by only one ALU.

Programmable Interconnect Points are vital to any FPGA, they allow us to link the output of a cell, or an input pad of the FPGA to any other cell/Pad in the circuit by making a “path” for the data through the FPGA. Every way is fixed during the programming by connecting metal lines with PIPs. These PIPs, well named, are programmable to permit us to build any path we want.

Programmable Interconnect Point (PIP)

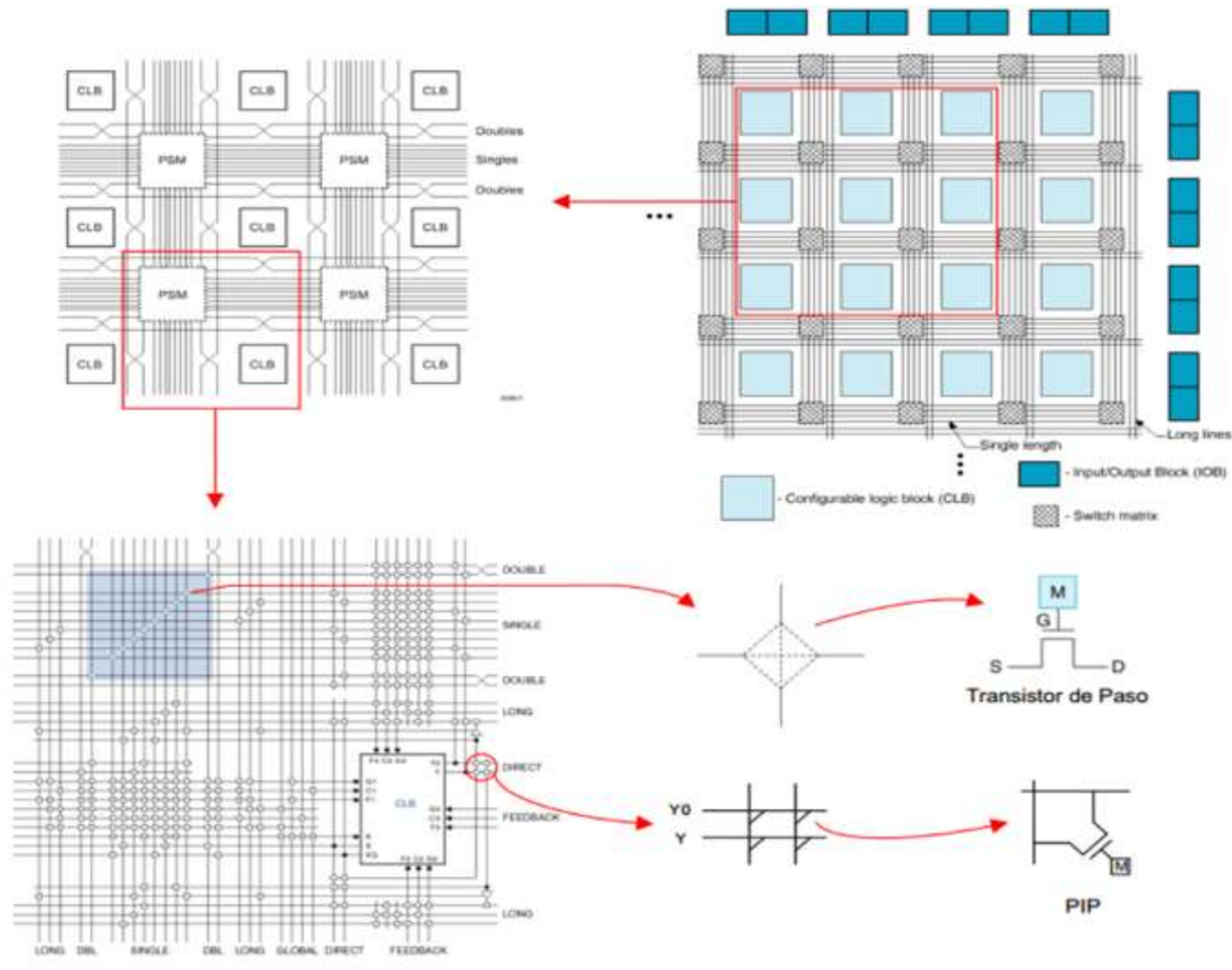


Fig. Programmable Interconnect Point

HDL Programming

Digital circuits consist primarily of interconnected transistors. We design and analyze these circuits with the aid of a hierarchical structure: we could, in theory, interpret a central processing unit (CPU) as a vast sea of transistors, but it is much easier to organize transistors into logic gates, logic gates into adders or registers or timing modules, registers into memory banks, and so forth.

To describe digital circuits, textual language is used that is specifically intended to clearly and concisely capture the defining features of digital design.

Such languages are called hardware description languages (HDLs).

The most popular hardware description languages are Verilog and VHDL. They are widely used in conjunction with FPGAs, which are digital devices that are specifically designed to facilitate the creation of customized digital circuits.

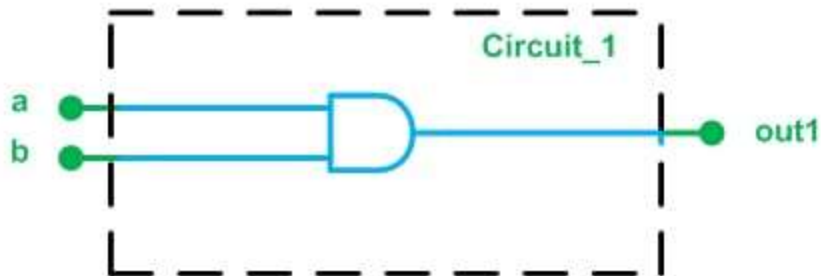
Hardware description languages allow you to describe a circuit using words and symbols, and then development software can convert that textual description into configuration data that is loaded into the FPGA in order to implement the desired functionality.

A hardware description language (HDL) is a programming language used to describe the behavior or structure of digital circuits (ICs). HDLs are also used to stimulate the circuit and check its response. Many HDLs are available, but VHDL and Verilog are by far the most popular. Most CAD tools available in the market support these languages. VHDL stands for “very high-speed integrated-circuit hardware description language.”

HDL Program for AND gate

```
entity Circuit_1 is  
  Port ( a : in STD_LOGIC;  
        b : in STD_LOGIC;  
        out1 : out STD_LOGIC);  
end Circuit_1;
```

```
-----  
architecture Behavioral of Circuit_1 is  
begin  
out1 <= ( a and b );  
end Behavioral;
```



NOT gate and half adder HDL Programs

```
LIBRARY IEEE;  
USE IEEE.STD_LOGIC_1164.ALL;  
ENTITY not1 IS  
PORT( a : IN STD_LOGIC; b : OUT STD_LOGIC; );  
END not1;  
ARCHITECTURE behavioral OF not1 IS  
BEGIN b <= NOT a;  
END behavioral;
```

```
entity HALF_ADDER is  
port (A, B: in BIT;  
SUM, CARRY: out BIT);  
end HALF_ADDER;
```


overview of Spartan 3E and Virtex II pro FPGA boards

The Spartan-3E family of Field-Programmable Gate Arrays (FPGAs) is specifically designed to meet the needs of high volume, cost-sensitive consumer electronic applications. The five-member family offers densities ranging from 100,000 to 1.6 million system gates. The Spartan-3E family builds on the success of the earlier Spartan-3 family by increasing the amount of logic per I/O, significantly reducing the cost per logic cell. New features improve system performance and reduce the cost of configuration.

These Spartan-3E FPGA enhancements, combined with advanced 90 nm process technology, deliver more functionality and bandwidth per dollar than was previously possible, setting new standards in the programmable logic industry.

Because of their exceptionally low cost, Spartan-3E FPGAs are ideally suited to a wide range of consumer electronics applications, including broadband access, home networking, display/projection, and digital television equipment.

The Spartan-3E family is a superior alternative to mask programmed ASICs. FPGAs avoid the high initial cost, the lengthy development cycles, and the inherent inflexibility of conventional ASICs. Also, FPGA programmability permits design upgrades in the field with no hardware replacement necessary, an impossibility with ASICs.

overview of Spartan 3E and Virtex II pro FPGA boards

Features of Spartan 3E

Very low cost

high-performance logic solution for high-volume consumer-oriented applications

- Proven advanced 90-nanometer process technology
- Multi-voltage, multi-standard SelectIO™ interface pins
 - Up to 376 I/O pins or 156 differential signal pairs
 - LVCMOS, LVTTTL, HSTL, and SSTL single-ended signal standards
 - 3.3V, 2.5V, 1.8V, 1.5V, and 1.2V signaling

622+ Mb/s data transfer rate per I/O

- True LVDS, RSDS, mini-LVDS, differential HSTL/SSTL differential I/O
- Enhanced Double Data Rate (DDR) support
- DDR SDRAM support up to 333 Mb/s

Abundant, flexible logic resources

- Densities up to 33,192 logic cells, including optional shift register or distributed RAM support

- Efficient wide multiplexers, wide logic
- Fast look-ahead carry logic
- - Enhanced 18 x 18 multipliers with optional pipeline
- - IEEE 1149.1/1532 JTAG programming/debug port

overview of Spartan 3E and Virtex II pro FPGA boards

Hierarchical SelectRAM memory architecture

- Up to 648 Kbits of fast block RAM

- - Up to 231 Kbits of efficient distributed RAM

- Up to eight Digital Clock Managers (DCMs)

- - Clock skew elimination (delay locked loop)

- Frequency synthesis, multiplication, division

- High-resolution phase shifting

- - Wide frequency range (5 MHz to over 300 MHz)

- Eight global clocks plus eight additional clocks per each half of device, plus abundant low-skew routing

- • Configuration interface to industry-standard PROMs

- - Low-cost, space

- saving SPI serial Flash PROM

- x8 or x8/x16 parallel NOR Flash PROM

- Low-cost Xilinx Platform Flash with JTAG

- Fully compliant 32-/64-bit 33 MHz PCI support (66 MHz in some devices)

- Low-cost QFP and BGA packaging options

- - Common footprints support easy density migration

- Pb-free packaging options

Virtex II pro FPGA boards

The Virtex-II Pro (V2-Pro) development system can be used at virtually any level of the engineering curricula, from introductory courses through advanced research projects. Based on the Virtex-II Pro FPGA, the board can function as a digital design trainer, a microprocessor development system, or a host for embedded processor cores and complex digital systems.

It is powerful enough to support advanced research projects, but affordable enough to be placed at every workstation. The expansion connectors can accommodate special-purpose circuits and systems for years to come, so the board can remain at the core of an engineering educational program indefinitely (see below for a current list of available expansion boards)

Virtex-II Pro

Key Specifications

Logic Cells 30,816

BRAM 2,448Kb

DDR Up to 2GB of DDR SDRAM

Ethernet On-board 10/100 Ethernet PHY

Clocks 100MHz system clock, 75MHz SATA

Connectivity and On-board I/O

RS-232

RS-232 DB9 serial port

PS/2 Two PS-2 serial ports

Audio AC-97 audio CODEC with audio amplifier and speaker/headphone output and line level output

Microphone

Microphone and line level audio input

Video

On-board XSGA output, up to 1200 x 1600 at 70Hz refresh

Virtex-II Pro

Switches 4
Push-buttons 5
LEDs 4 LEDs
User LED 4
User RGB LED 4
Electrical Power 4.5-5.5V
Logic Level 3.3V

DIGITAL RESOURCES

- ❖ [Lecture Notes – www.svec.edu.in](http://www.svec.edu.in)
- ❖ Vidéo Lectures –

THANK YOU