# 8086 Hardware Specifications

***Course Teacher:***

**Md. Obaidur Rahman, Ph.D.**
Professor
Department of Computer Science and Engineering (CSE)
Dhaka University of Engineering & Technology (DUET), Gazipur.

**Course ID:** CSE - 4503
**Course Title:** Microprocessors and Assembly Language
Department of Computer Science and Engineering (CSE),
Islamic University of Technology (IUT), Gazipur.

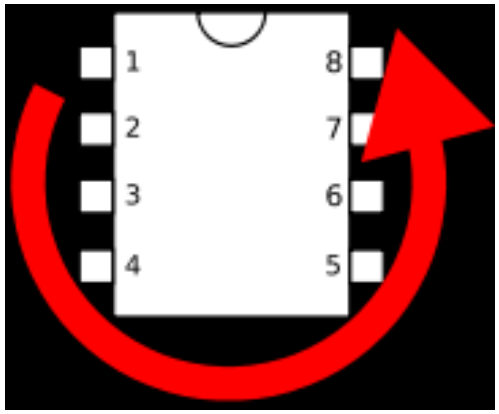
# Lecture References:

- **Book:**
  - *The 8086/8088 Family: Design, Programming, And Interfacing, Chapter # 6,* **Author:** John Uffenbeck.

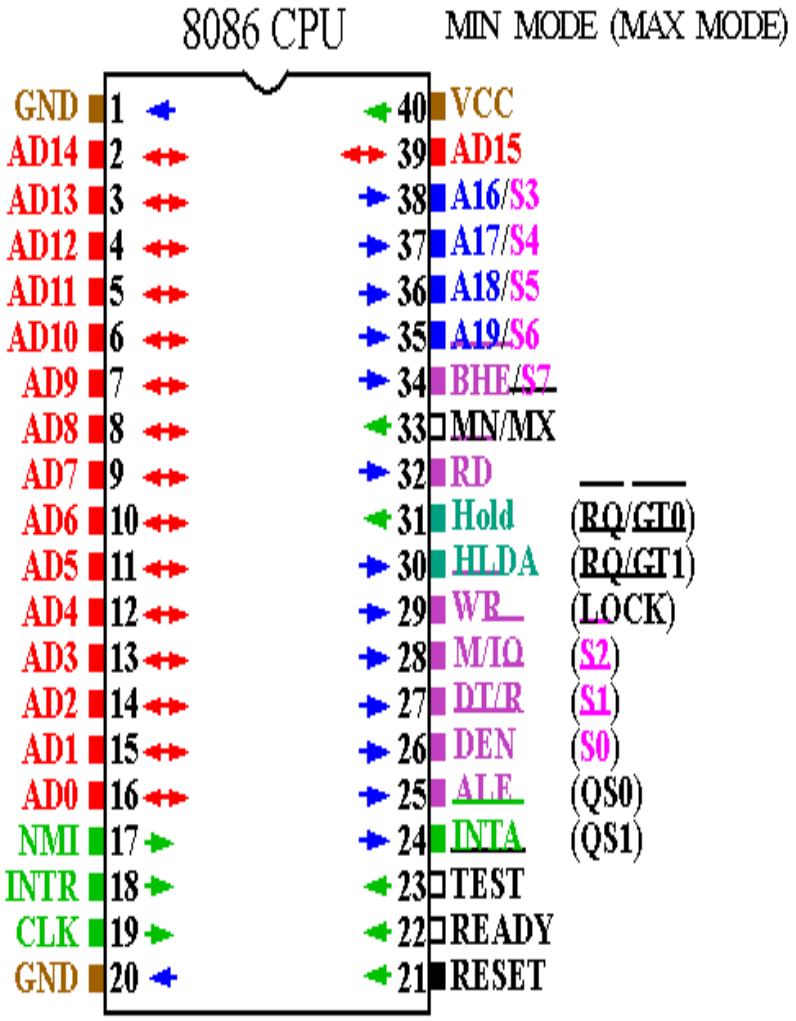
- **Lecture Materials:**
  - M. A. Sattar, Microprocessor and Microcontroller, BUET.
  - Md. Omar Faruque, BRAC University.

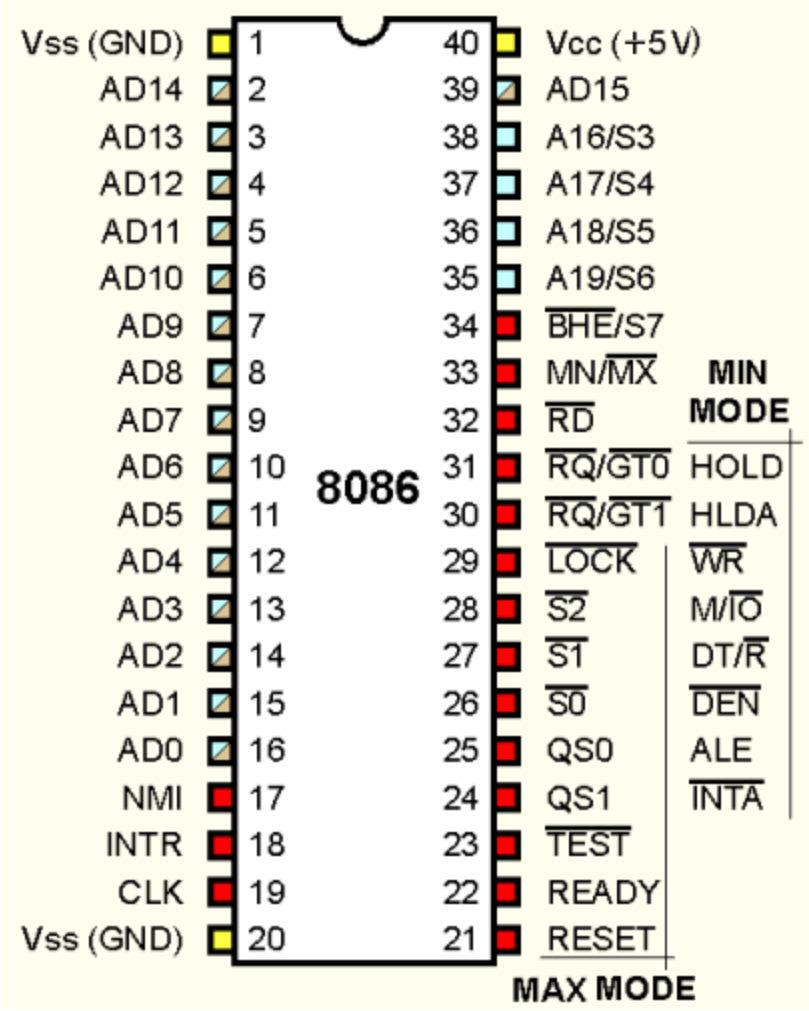
# 8086 Pin Specification

- 8086 is packaged as 40-pin DIPs.
- In micro-electronics **DIP** stands for **Dual in-line package.**
- DIP packaging refers to a rectangular housing with two parallel rows of electrical connection pins.
- DIP chips have a notch on one end to show its correct orientation.
- The pins are then numbered as shown in the figure below.



# 8086 Pin Specification



# Minimum and Maximum Mode

- The mode is selected by PIN 33
  - 1 = Minimum, 0 = Maximum
  - Use of Pin 24 to 31 changes with the mode.

- The **minimum mode** is intended for single-processor systems on one printed circuit board (PCB).

- The **maximum mode** is intended for more complex system with separate I/O and memory boards.

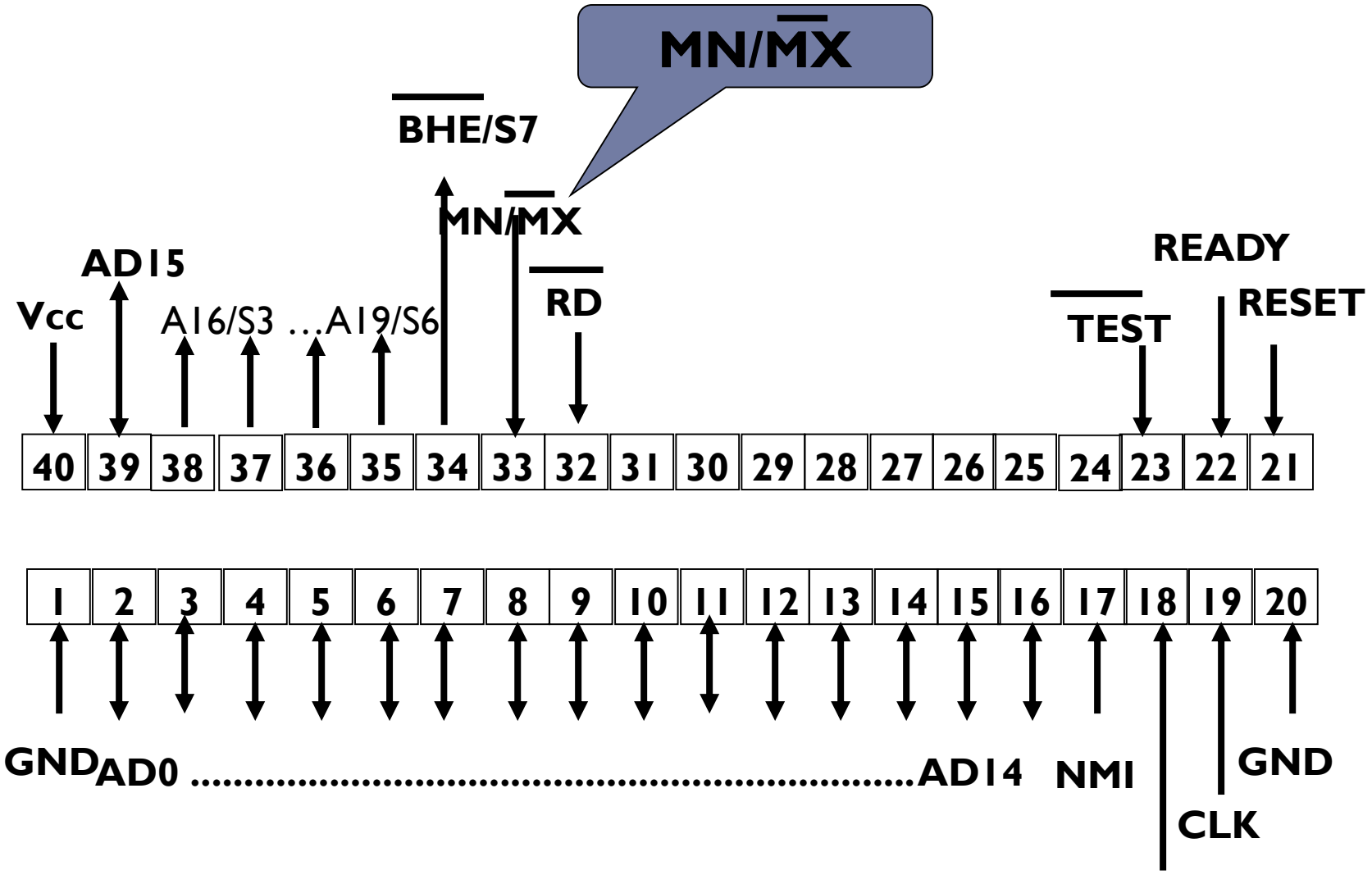
# Minimum and Maximum Mode

| Minimum mode | Maximum mode |
|---|---|
| In minimum mode there can be only one processor i.e. 8086. | In maximum mode there can be multiple processors with 8086, like 8087 and 8089. |
| MN/$\overline{MX}$ is 1 to indicate minimum mode. | MN/$\overline{MX}$ is 0 to indicate maximum mode. |
| ALE for the latch is given by 8086 as it is the only processor in the circuit. | ALE for the latch is given by 8288 bus controller as there can be multiple processors in the circuit. |
| $\overline{DEN}$ and DT/$\overline{R}$ for the trans-receivers are given by 8086 itself. | And DT/$\overline{R}$ for the trans-receivers are given by 8288 bus controller. |
| Direct control signals M/$\overline{IO}$, $\overline{RD}$ and $\overline{WR}$ are given by 8086. | Instead of control signals, each processor generates status signals called $\overline{S2}$, $\overline{S1}$ and $\overline{S0}$. |

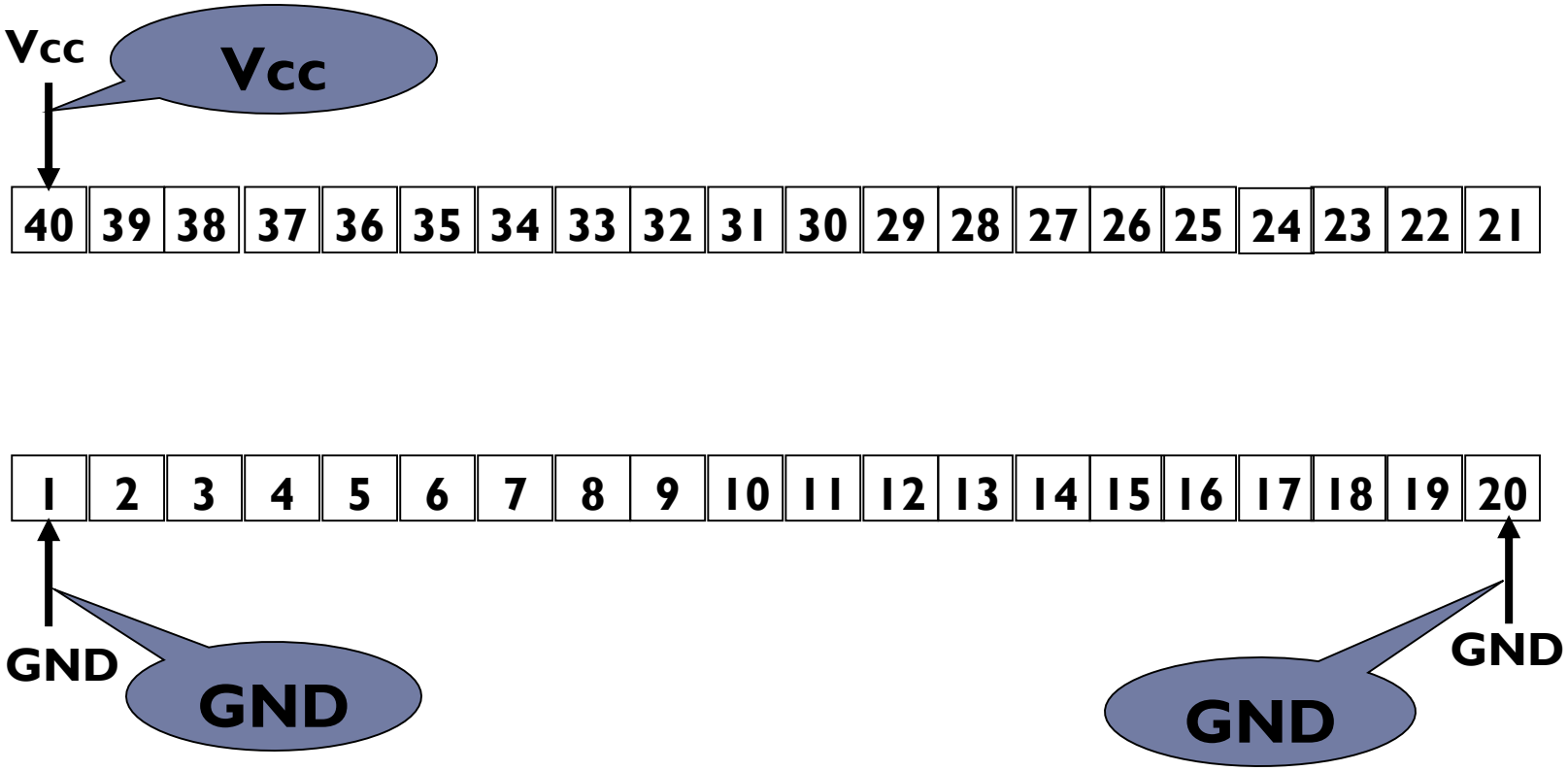
# Minimum and Maximum Mode

| Minimum mode | Maximum mode |
|---|---|
| Control signals $M/\overline{IO}$, $\overline{RD}$ and $\overline{WR}$ are decoded by a 3:8 decoder like 74138. | Status signals $\overline{S2}$, $\overline{S1}$ and $\overline{S0}$ are decoded by a bus controller like 8288 to produce control signals. |
| $\overline{INTA}$ is given by 8086 in response to an interrupt on INTR line. | $\overline{INTA}$ is given by 8288 bus controller in response to an interrupt on INTR line. |
| HOLD and HLDA signals are used for bus request with a DMA controller like 8237. | $\overline{RQ}/\overline{GT}$, lines are used for bus requests by other processors like 8087 or 8089. |
| The circuit is simpler. | The circuit is more complex. |
| Multiprocessing cannot be performed hence performance is lower. | As multiprocessing can be performed, it can give very high performance. |

# Control Signal – Minimum and Maximum Mode



# Power Supply



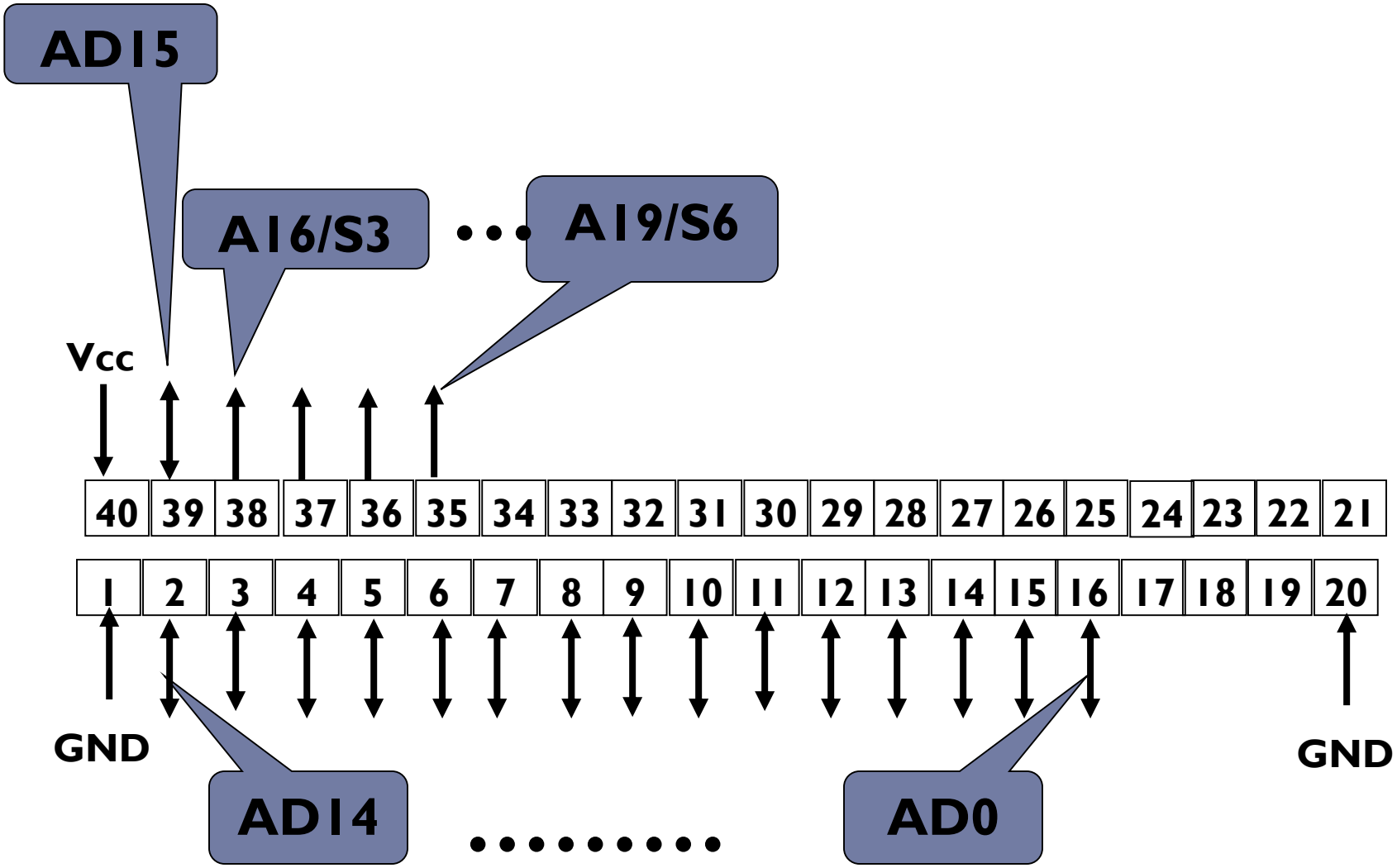
# Pins for Data and Address Bus

- **Data Bus (AD0 – AD15)**
  - These 16 pins form the CPU's bidirectional data bus

- **Address Bus (AD0 – AD15 and A16/S3 – A19/S6)**
  - These 20 pins correspond to the CPU's 20-bit address bus and allow the processor to access $2^{20}$ or 10, 48, 576 unique memory locations.

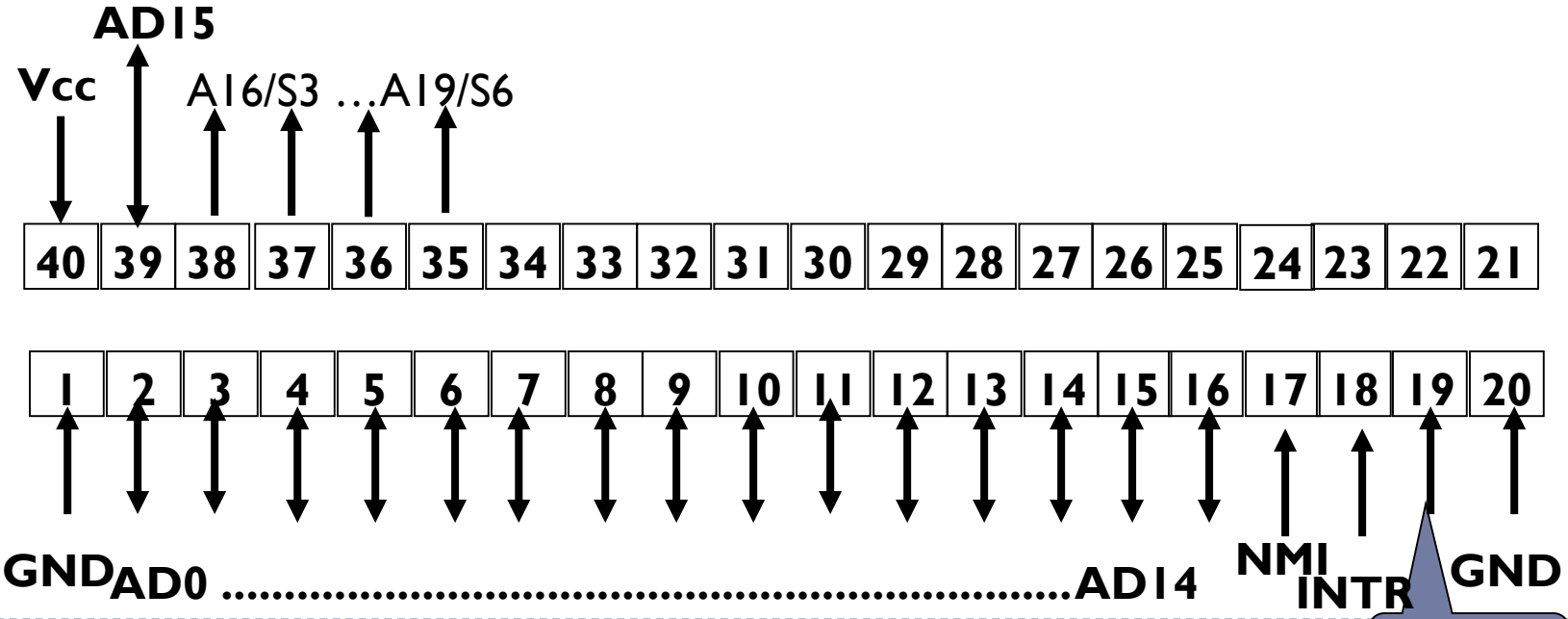
- **Address Latch Enable (ALE)** (Pin 25)
  - Line carries address when ALE =1
  - Line carries data when ALE =0

# Pins for Data and Address Bus



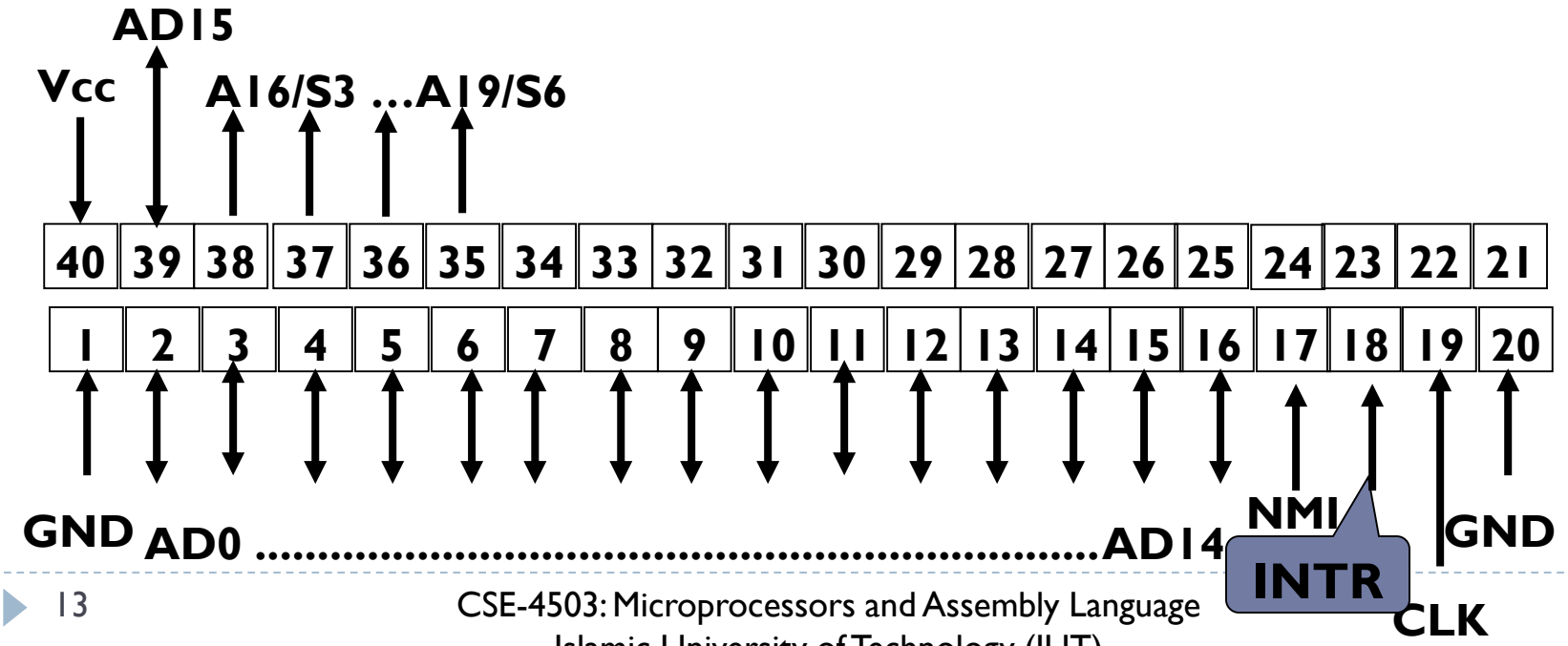
# Control Signal – Clock

- Clock provides the basic timing for the processor and bus controller.
- It is asymmetric with a 33% duty cycle to provide optimized internal timing.
- 8086 is found to operate in 5 and 10 Mhz.



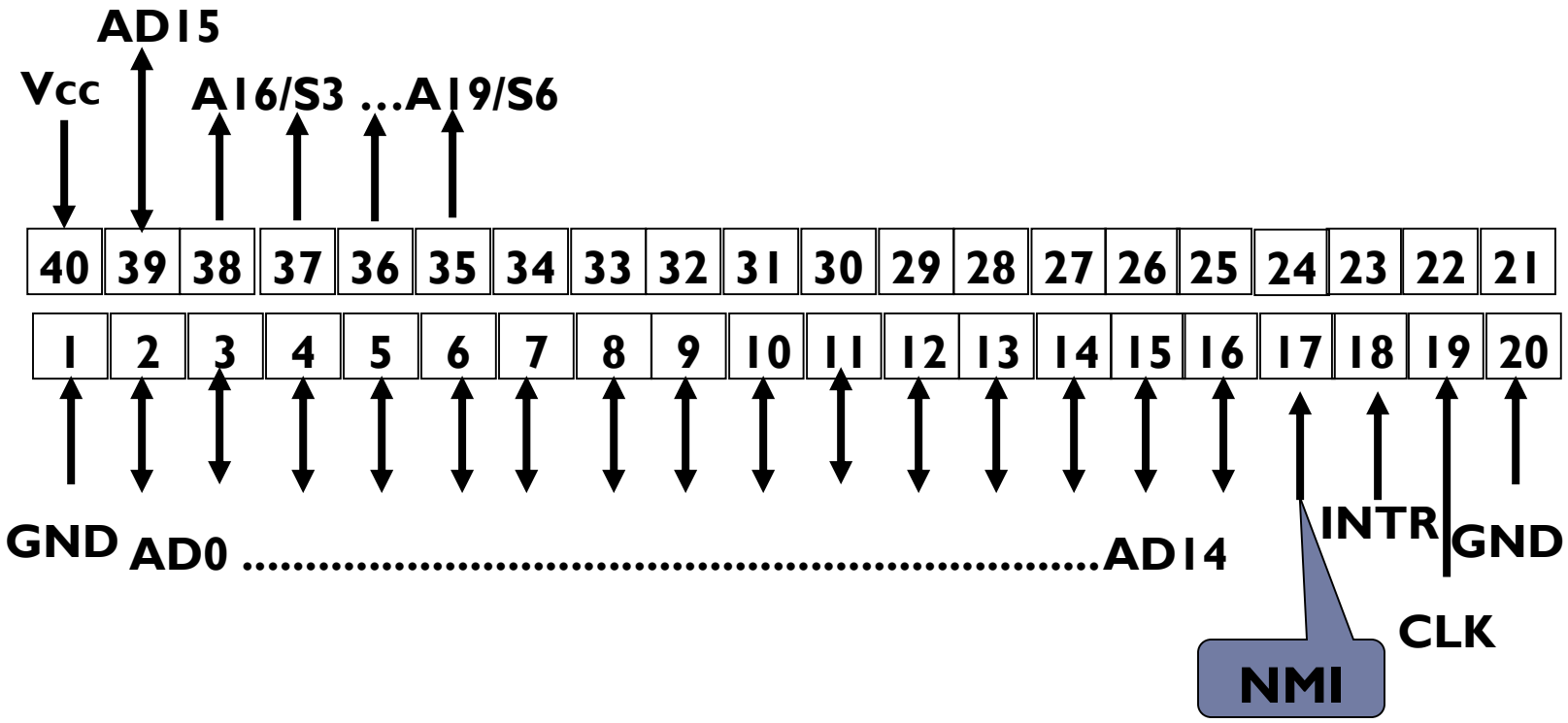
# Control Signal – Interrupts

- **INTR: Interrupt request** pin is used to request a hardware interrupt.
- If INTR is held at high when **Interrupt Flag (IF=1)**, the processor goes into the interrupt acknowledgement cycle. INTA becomes active when interrupt is being serviced.



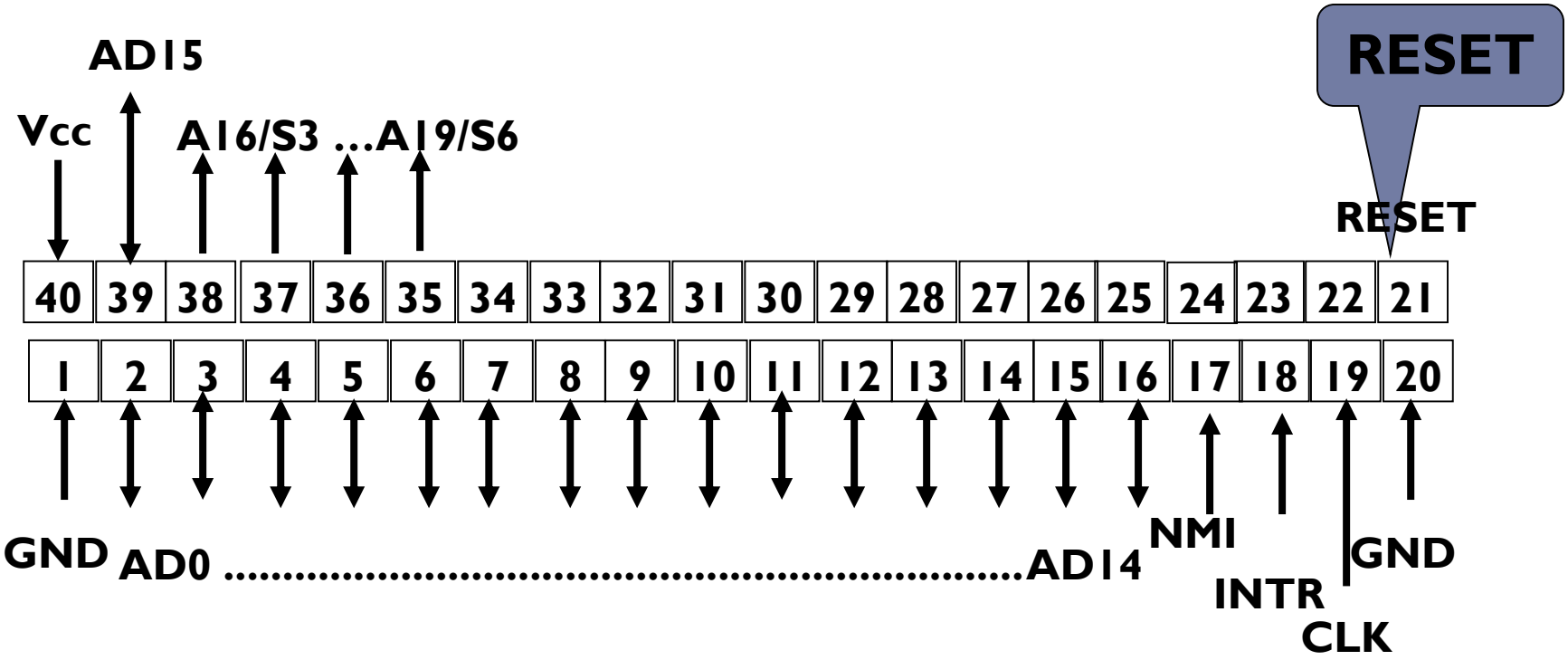
# Control Signal – Interrupts

- **NMI: Non-maskable interrupt** input is similar to INTR expect that the NMI interrupt does not check **Interrupt Flag (IF)** or priority.



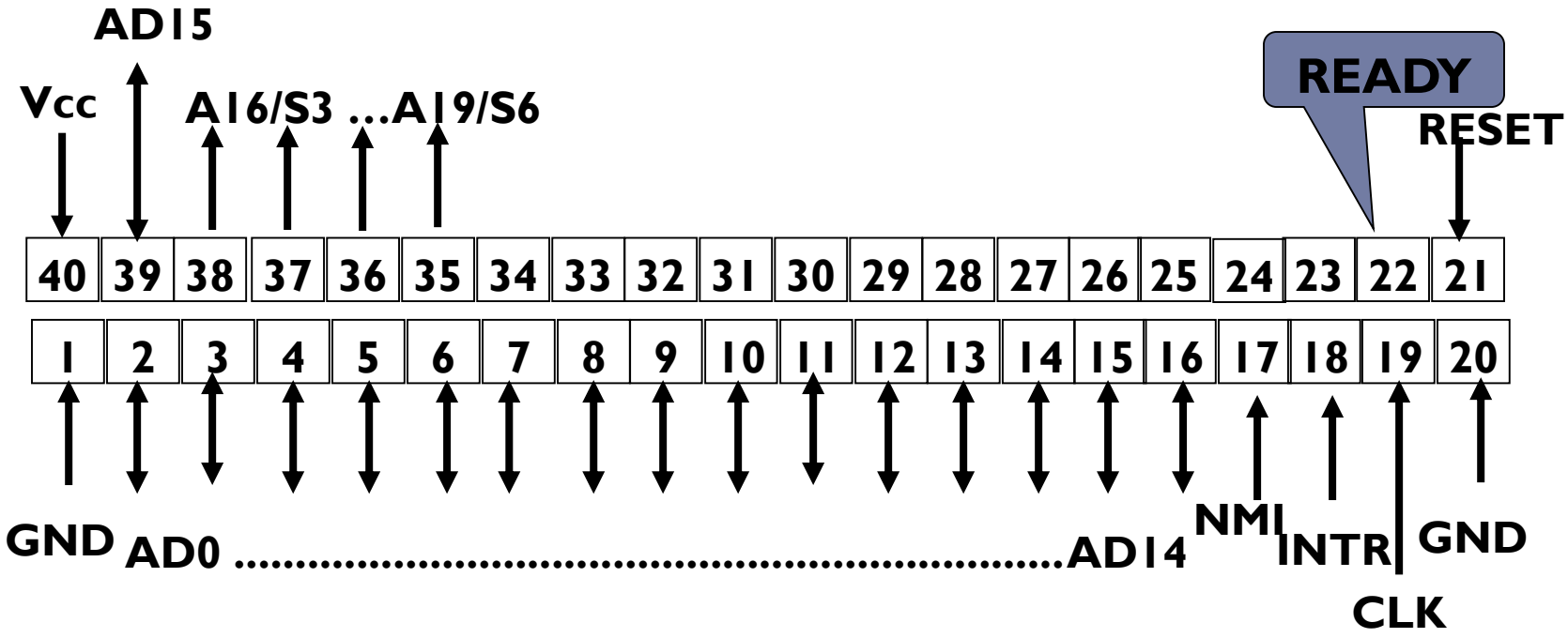
# Control Signal – RESET

- **RESET** pin is held HIGH for at least 4 clock cycles to reset the microprocessor. It causes the processor to immediately terminates its present activity.



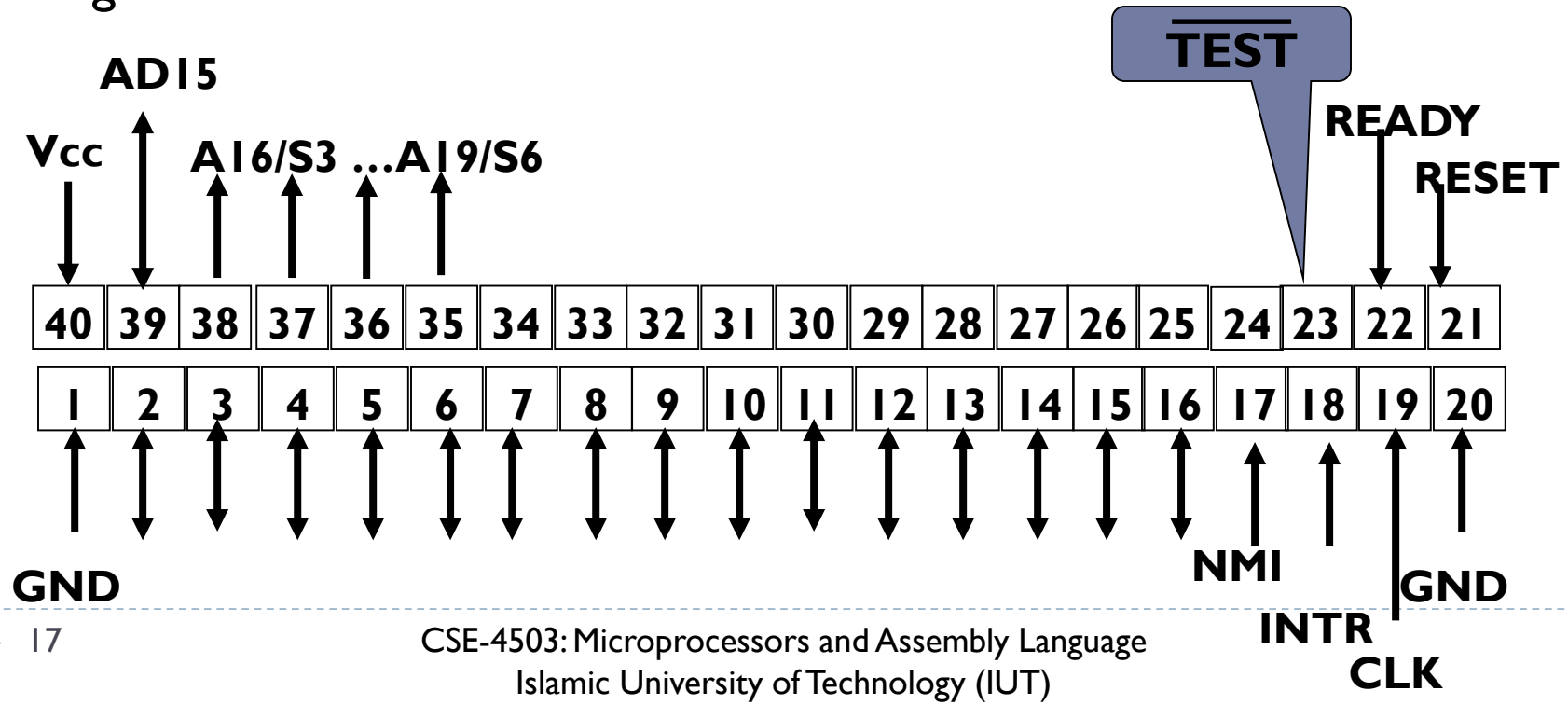
# Control Signal – READY

- The READY pin is used to enforce a waiting state.
  - READY pin at 0 – the microprocessor goes into idle state.
    READY pin at 1 – the microprocessor does normal operation.



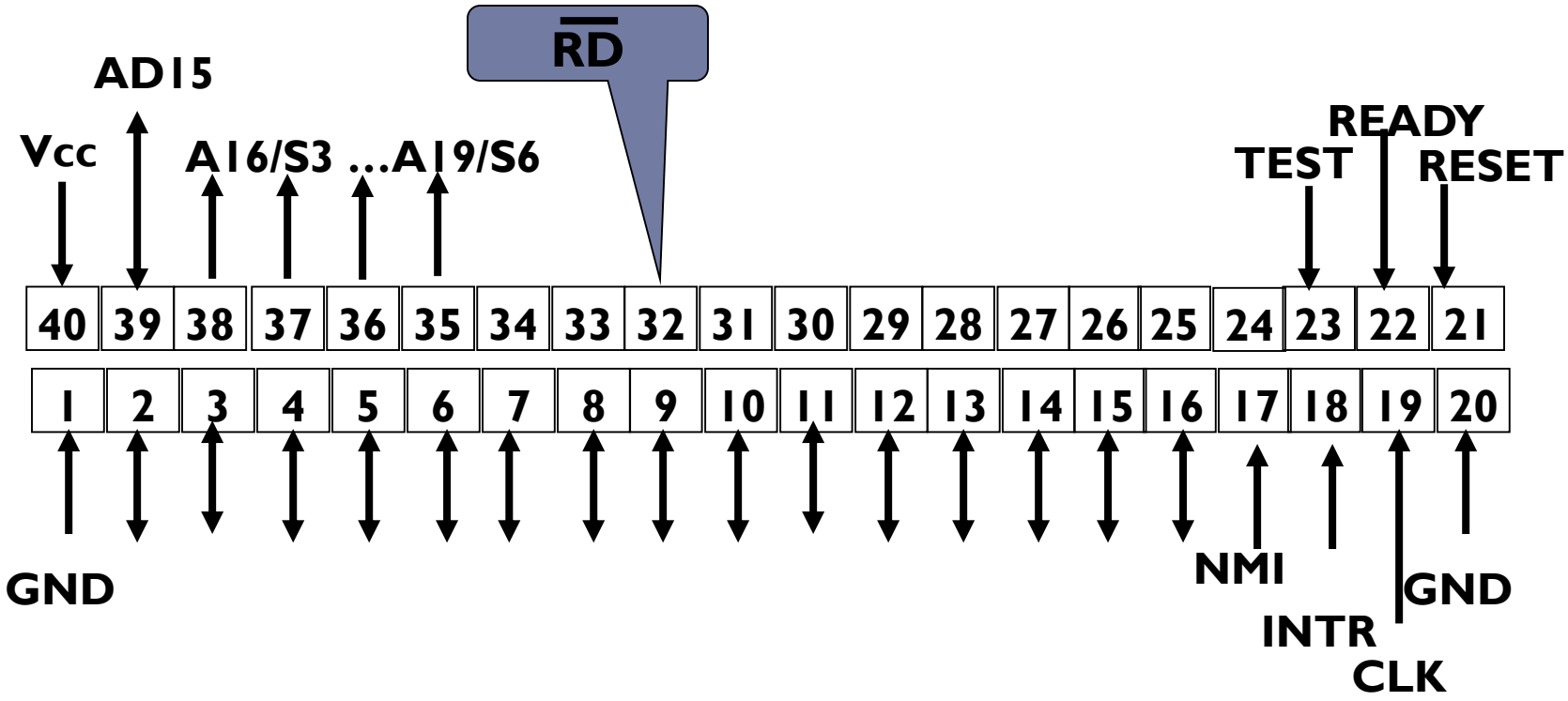
# Control Signal – TEST

- **$\overline{\text{TEST}}$:** Test pin is an input that is tested by the **WAIT** instruction.
- If the test pin is at logic 0 the WAIT instruction functions as NOP.
- If test is a logic 1, the WAIT instruction wait for TEST to become logic 0.



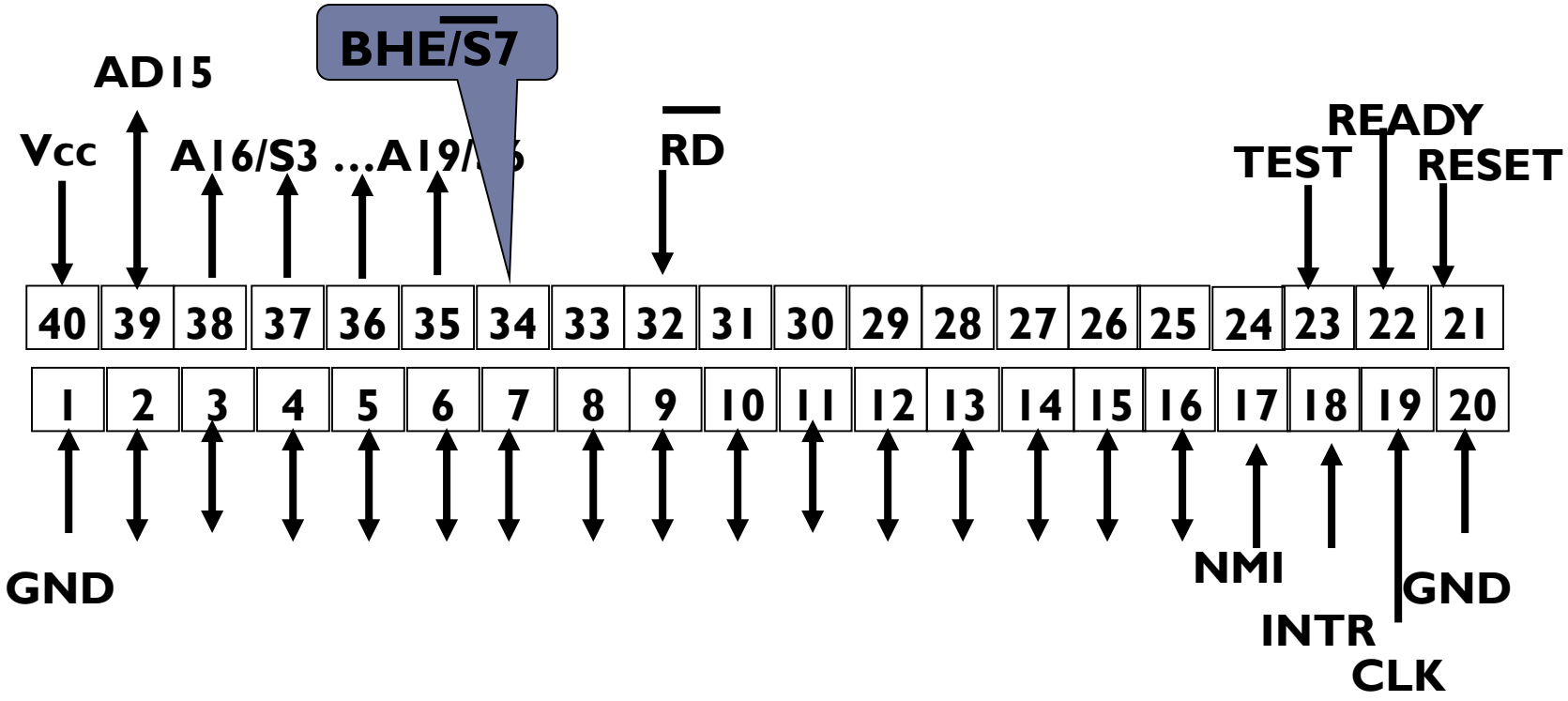
# Control Signal – READ

- **$\overline{RD}$:** When this **read signal** pin is at logic 0, the data bus is receptive to data from memory or I/O devices.



# Control Signal – BHE

- The **Bus High Enable (BHE)** pin is used in the 8086 to enable the Most significant data bus bits (**AD8 to AD15**) during a read or write operation.



# Control Signal –Segment Register Status

- This information indicates which segment register is presently being used for data accessing.

| S4 | S3 | Function |
|---|---|---|
| 0 | 0 | ES |
| 0 | 1 | SS |
| 1 | 0 | CS |
| 1 | 1 | DS |

**S5** indicates the status of I-flag

**S6** low indicates that μP is using the bus

**S7** spare status bit

# Control Signals – Minimum Mode

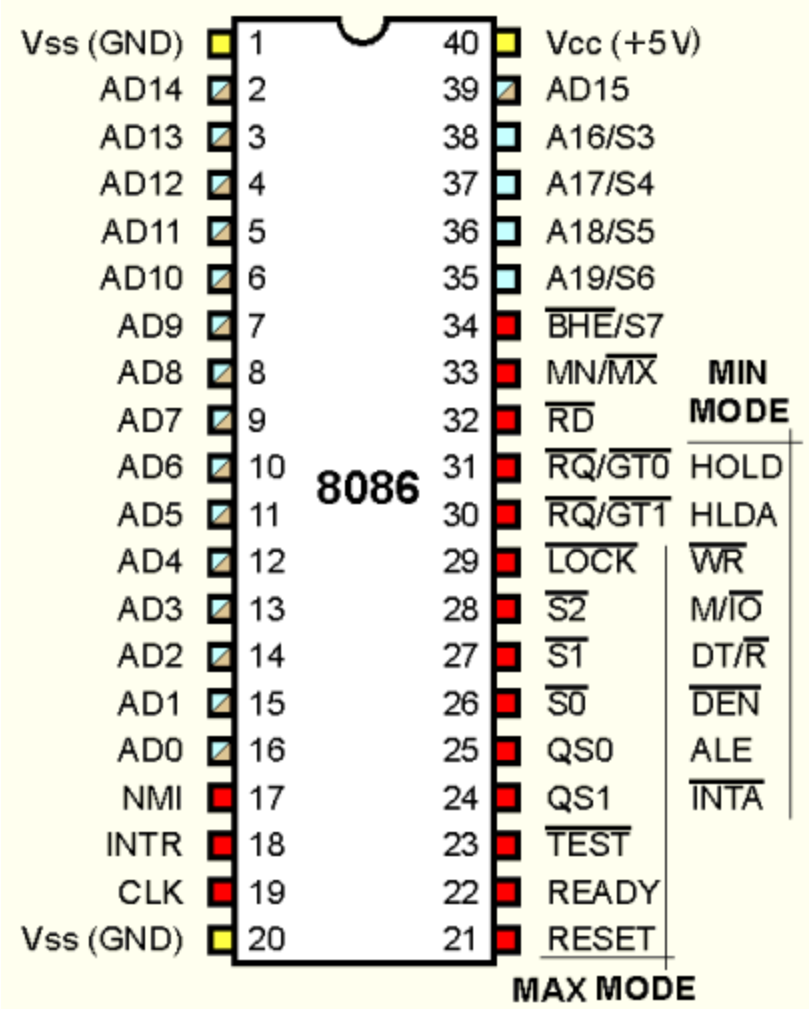
- **$\overline{\text{INTA}}$ (Pin 24): Interrupt acknowledgement** signals is a response to INTR input pin. This is used when the interrupt vector is placed on the address bus by the microprocessor.
- **ALE (Pin 25): Address Latch enable** shows whether the multiplexed AD lines carry address or data.
- **DEN (Pin 26): Data Enable bus** activates external data bus buffers.
- **DT/$\overline{\text{R}}$ (Pin 27): Data transmit/receive** shows that the microprocessor data bus is transmitting(1) or receiving(0) data. This is used to control buffers.

# Control Signals – Minimum Mode

- **M/$\overline{IO}$ (Pin 28):** This pin indicates whether the address bus contains a memory address or an I/O port address.
- **$\overline{WR}$ (Pin 29):** The **write line** is a used when the microprocessor is writing data to memory and the memory bus contains a valid address.
- **HOLD (Pin 30): HOLD** pin is used to input request DMA (Direct Memory Access). Hold set to 1 microprocessor gives up control of buses to DMA controller.
- **HLDA (Pin 31): HLDA** pin is used to acknowledge DMA request.

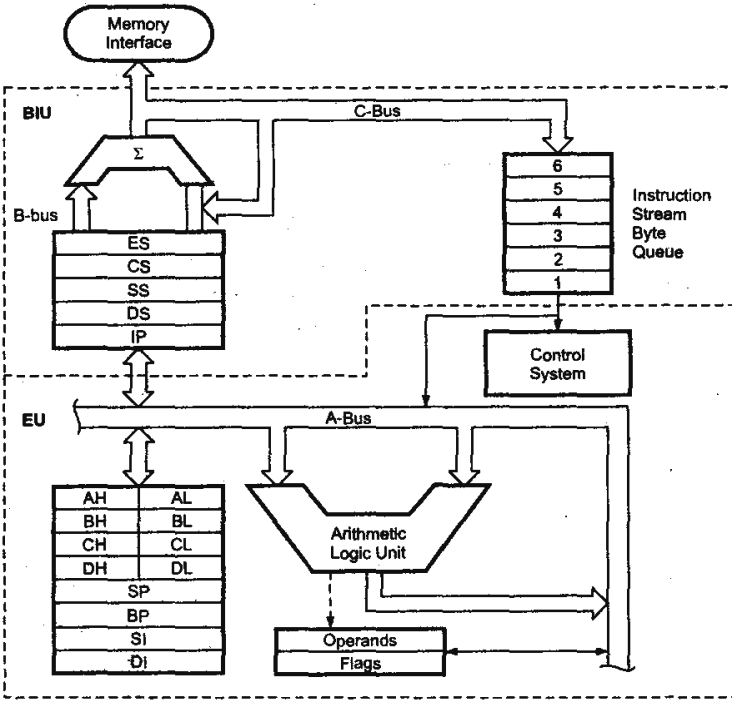
# Control Signals – Maximum Mode

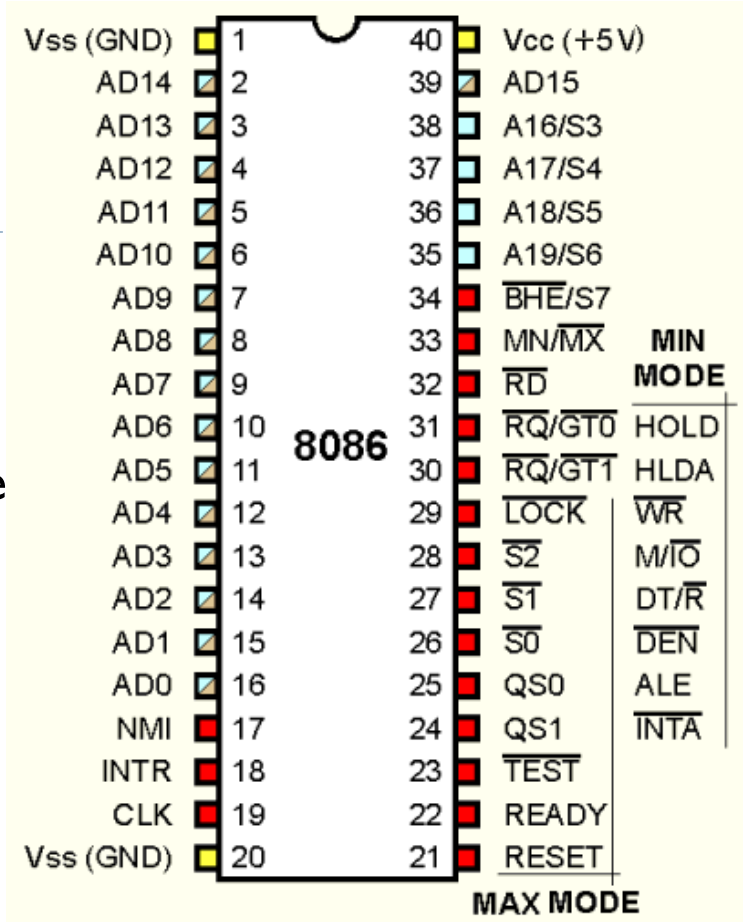
| S2 | S1 | S0 | |
|---|---|---|---|
| 0 | 0 | 0 | Interrupt Acknowledge |
| 0 | 0 | 1 | Read I/O port |
| 0 | 1 | 0 | Write I/O port |
| 0 | 1 | 1 | Halt |
| 1 | 0 | 0 | Code Access |
| 1 | 0 | 1 | Read Memory |
| 1 | 1 | 0 | Write Memory |
| 1 | 1 | 1 | Inactive |



# Instruction Queue Status

| QS1 | QS0 | |
|---|---|---|
| 0 | 0 | No operation |
| 0 | 1 | First byte of opcode fetched from queue |
| 1 | 0 | Empty queue |
| 1 | 1 | Subsequent byte from queue |



# Thank You !!

# 8086 Interrupts

***Course Teacher:***

**Md. Obaidur Rahman, Ph.D.**
Professor
Department of Computer Science and Engineering (CSE)
Dhaka University of Engineering & Technology (DUET), Gazipur.

**Course ID:** CSE - 4503
**Course Title:** Microprocessors and Assembly Language
Department of Computer Science and Engineering (CSE),
Islamic University of Technology (IUT), Gazipur.

# Lecture References:

- Book:
  - *Microprocessors and Interfacing: Programming and Hardware, Chapter # 08,* **Author:** Douglas V. Hall
  - *Microprocessor and Microcomputer – Based System Design,* **Author:** Mohamed Rafiquzzaman

- Lecture Materials:
  - *M. A. Sattar, BUET, Dhaka, Bangladesh.*

# Interrupt

- Interrupt is a process where a normal program execution to be interrupted by some external signal or by a special instruction in the program.

- Microprocessor pay attention to the interrupt stopping the the current execution.

# Classifications of 8086 Interrupts

- An 8086 interrupt can come from any of the ***three*** sources:
  - An ***external signal*** applied to NMI or INTR pin.
    - known as ***hardware interruption***
    - *It is a* ***user-defined interrupt***
    - *Example: Connecting I/O Device*
  - Execution of interrupt instruction ***INT.***
    - referred as ***software interruption***
    - *It is also a* ***user-defined interrupt***
    - *Example: INT 21h, INT0, INT 3*
  - Some error condition produced by execution of an instruction, e.g., trying to divide some number by zero.
    - It is known as ***pre-defined interrupt***

# Classifications of 8086 Interrupts

- Hardware Interrupts can be classified into two types:
  - Maskable Interrupts (Can be delayed or Rejected)
    - User-defined interrupts
  - Non-Maskable Interrupts (Can not be delayed or Rejected)
    - System Interrupts for Major system faults occur.
- Interrupt priority Hierarchy (Highest to Lowest)
  - Reset
  - Internal Interrupt and exceptions (e.g., divide by zero)
  - Software Interrupt
  - Non-maskable Interrupt
  - External Hardware Interrupt

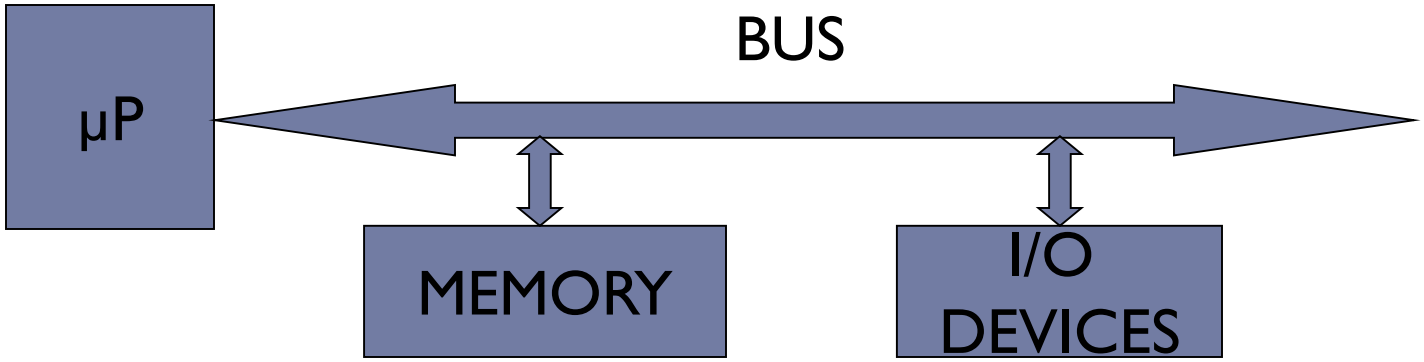
# Interrupt & Its Consequences over MP

- An interrupt is considered to be an emergency signal that may be serviced.
  - The Microprocessor may respond to it as soon as possible.

- **What happens when MP is interrupted ?**
  - When the Microprocessor receives an interrupt signal, it suspends the currently executing program and jumps to an **Interrupt Service Routine** (ISR) to respond to the incoming interrupt.
  - Each interrupt will have its own ISR.
  - After finishing the second program/interrupt, automatically return to the first program and start execution from where it was left

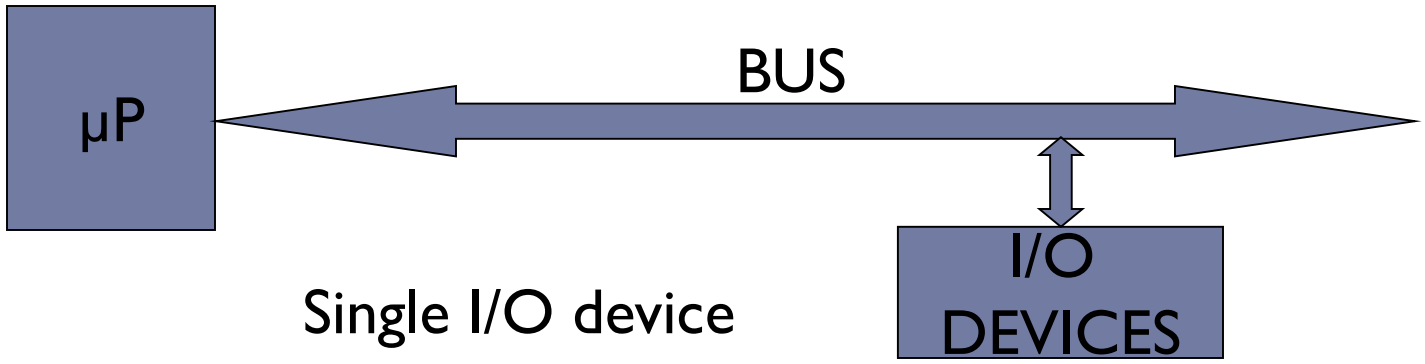
# Why Interrupt is Necessary?

- To understand this we will have to review how μP/μC communicate with the outside world.



# Why Interrupt is Necessary?

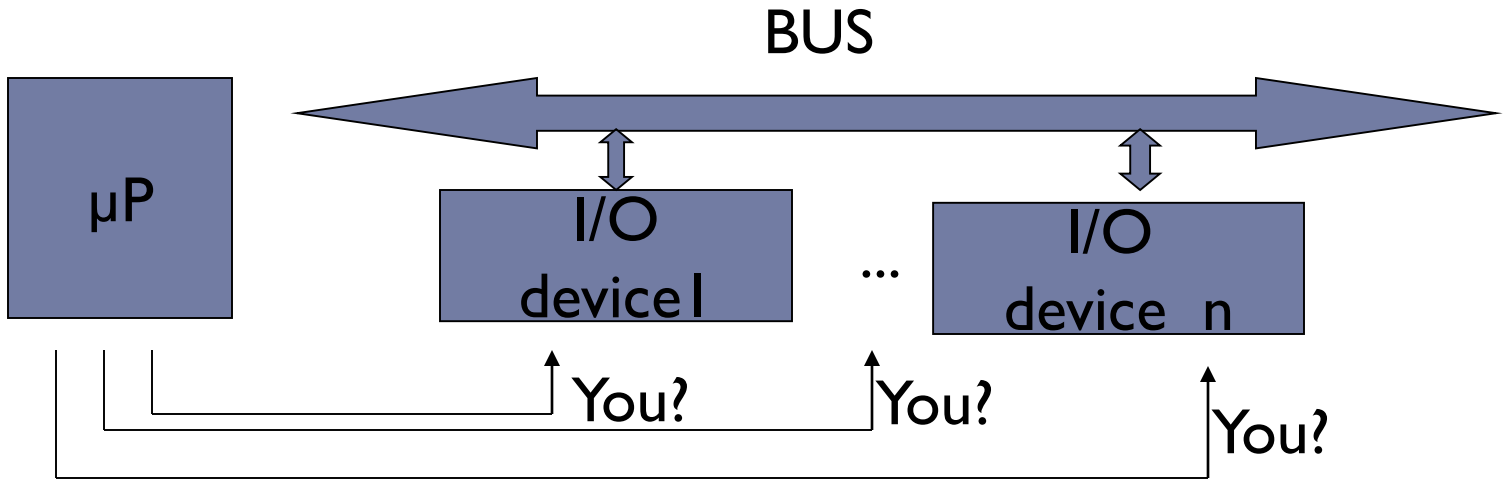
- **First Type:** DEDICATED communication between MP and I/O devices.




Single I/O device

# Why Interrupt is Necessary?

- **Second Type:** POLLED I/O or PROGRAMMED I/O communication between MP and I/O devices.

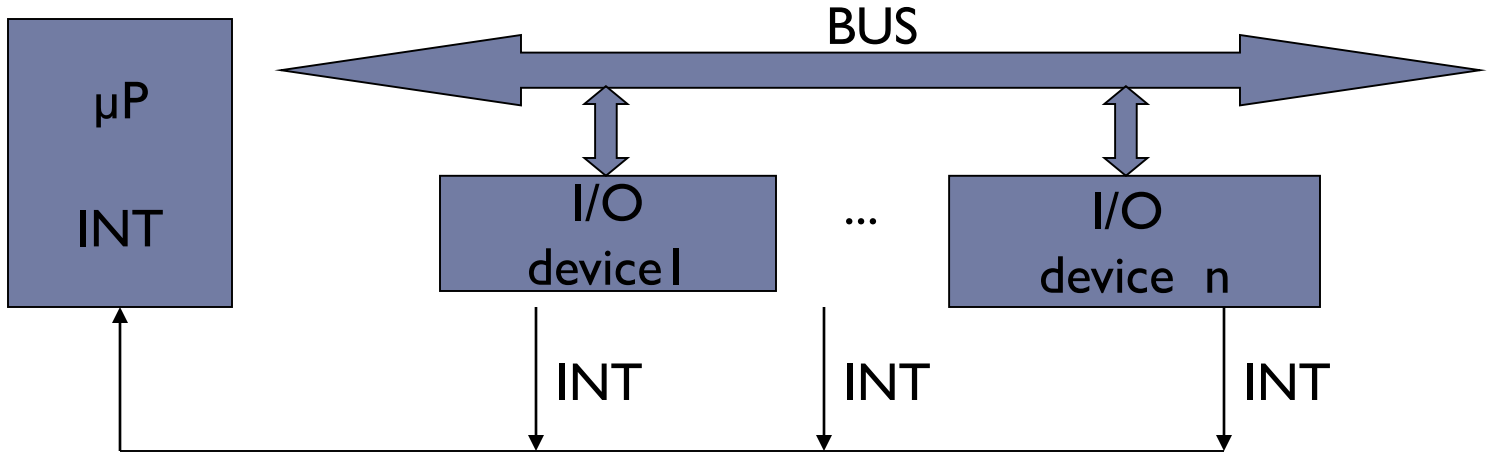



**Disadvantages of Second Type Communication:**

- not fast enough
- waste too much microprocessor time

# Why Interrupt is Necessary?

- **Third Type:** INTERRUPTED I/O communication between MP and I/O devices.




Interrupts are particularly useful when I/O devices are slow

# Polling and Interrupt

- Both are methods to notify processor that I/O device needs attention
- **Polling**
  - simple, but slow
  - processor check status of I/O device regularly to see if it needs attention
  - *similar to checking a telephone without bells*!
- **Interrupt**
  - fast, but more complicated
  - processor is notified by I/O device (interrupted) when device needs attention
  - *similar to a telephone with bells*

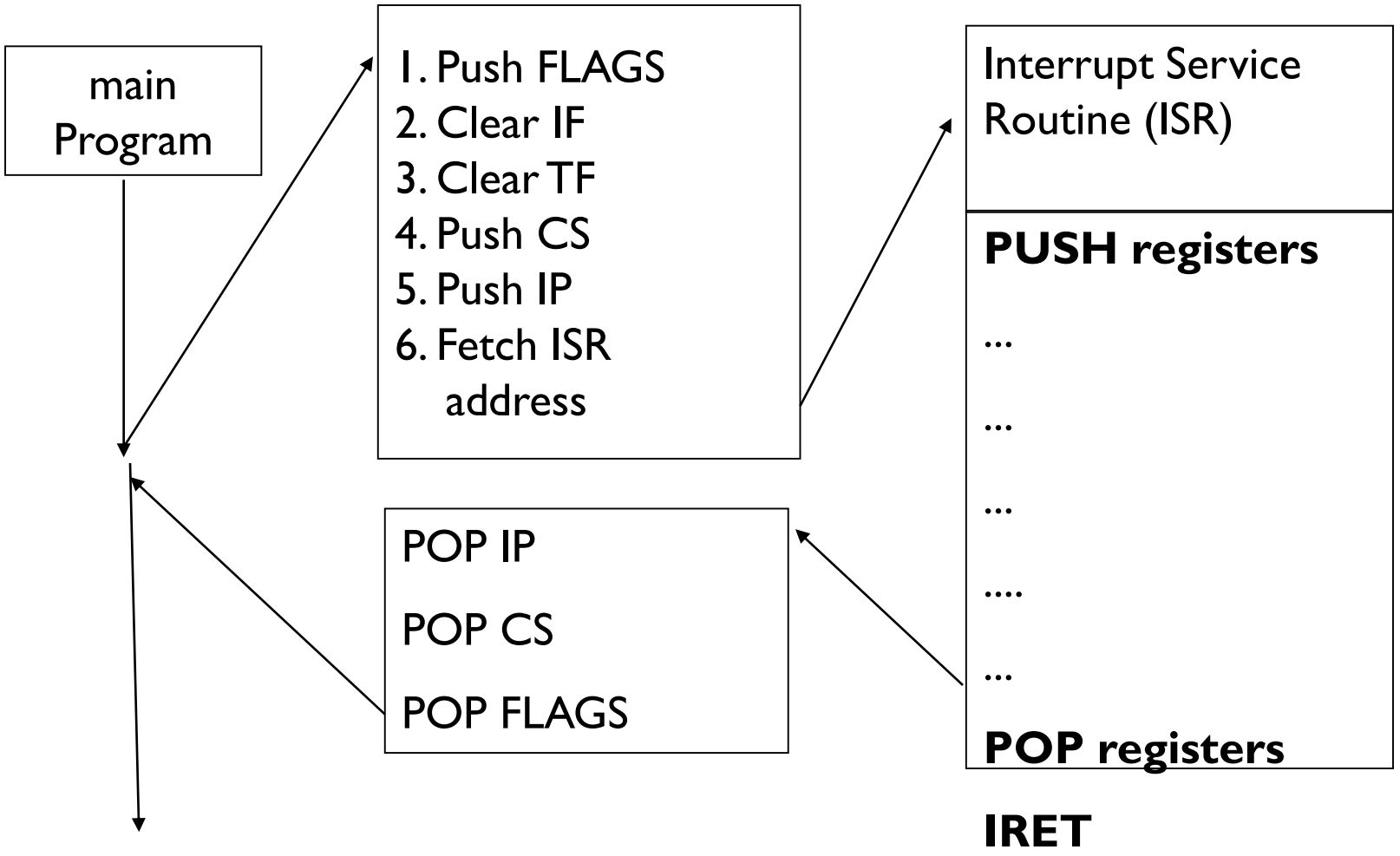
# Interrupt Concept

- Intel processors include two hardware pins (INTR and NMI) that request interrupts.
- And one hardware pin (INTA) to acknowledge the interrupt requested through INTR.
- The processor also has software interrupts INT, INTO and INT 3.
- Flag bits IF (interrupt flag) and TF (trap flag), are also used with the interrupt structure.

# Function of 8086 during Interrupts

- At the end of each instruction cycle, 8086 checks to see if any interrupts have been requested.
- If yes, then 8086 responds to the interrupt by stepping through the following series of major actions:
  - It decremented SP by 2 and pushes ***Flag register*** on the stack.
  - It disables 8086 **INTR** input by clearing **IF (Interrupt) flag** in Flag register, which is currently IF=1.
  - It resets the **TF (Trap) flag** in Flag register
  - It decremented SP again by 2 and pushes current **CS (Code Segment)** contents on the stack.
  - It decremented SP again by 2 and pushes current **IP (Instruction Pointer)** contents on the stack.
  - It does an indirect far **Jump** to the start of the procedure (**ISR**) written to respond to the interrupt.

# Function of 8086 during Interrupts



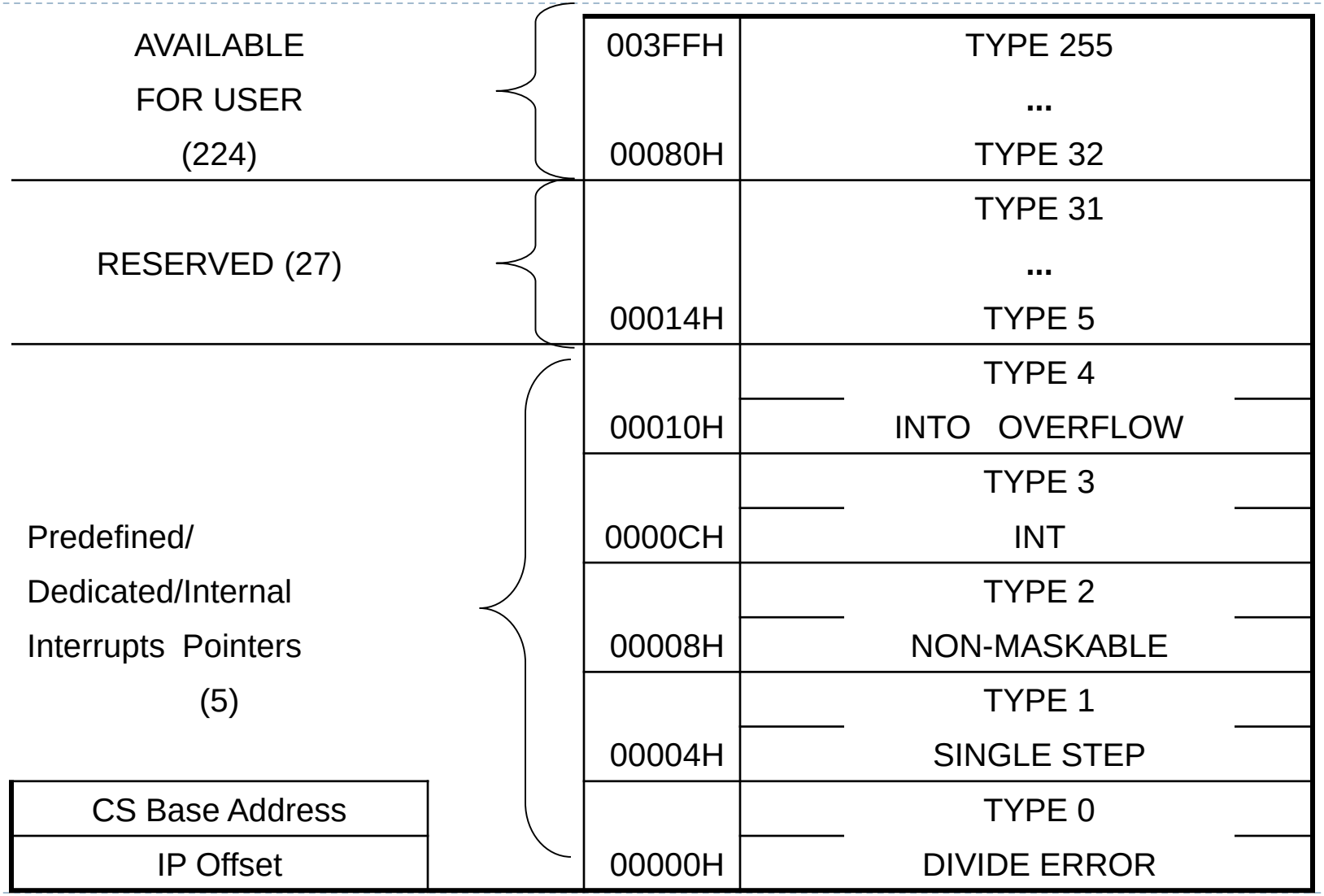
# Interrupt Vectors and Vector Table

- An **interrupt vector** is a **pointer** to where the ISR is stored in memory.
- All interrupts (vectored or otherwise) are mapped onto a memory area called the **Interrupt Vector Table (IVT)**.
  - The IVT is usually located in the first 1 Kbyte of memory segment (from 00000 H - 003FF H).
  - The purpose of the IVT is to hold the vectors that redirect the microprocessor to the right place when an interrupt arrives.
- The starting address of an ISR is often called
  - the ***interrupt vector*** or the ***interrupt pointer.***
- So the Table is referred to as
  - ***interrupt-vector table*** or ***interrupt-pointer table.***

# Interrupt Types based on ISR ID

- Note that
  - The **IP** value is put in as the **low word** of the vector
  - **CS** as **high word** of the vector
- 4 bytes are required to store the CS and IP values for each interrupt service procedure, the ***interrupt-vector table*** can hold starting addresses for up to 256 interrupt procedures.
- Each ***Double Word*** interrupt vector is identified by a number from 0 to 255
- *INTEL* calls this number the ***TYPE*** of the interrupt

# Interrupt Types based on ISR ID



# Interrupt Types based on ISR ID

- The first five interrupt vectors are identical in all Intel processors
- Intel reserves the first 32 interrupt vectors
- The last 224 interrupt vectors are user-available
- Each is four bytes long in real mode and contains the starting address of the interrupt service procedure.
  - The first two bytes contain the offset address
  - The last two contain the segment address

# Interrupt Types based on ISR ID

- **Type 0**
  - The **divide error** whenever the result from a division overflows or an attempt is made to divide by zero.

- **Type 1**
  - Single-step or trap occurs after execution of each instruction if the trap (TF) flag bit is set.
  - Upon accepting this interrupt, TF bit is cleared so the interrupt service procedure executes at full speed.

# Interrupt Types based on ISR ID

- **Type 2**
  - The **non-maskable interrupt** occurs when a logic 1 is placed on the NMI input pin to the microprocessor.
  - Non-maskable—it cannot be disabled
- **Type 3**
  - A special one-byte instruction (INT 3) that uses this vector to access its interrupt-service procedure.
  - Often used to store a breakpoint in a program for debugging

# Interrupt Types based on ISR ID

- **Type 4**
  - **Overflow** is a special vector used with the INTO instruction. The INTO instruction interrupts the program if an overflow condition exists.
  - As reflected by the overflow flag (OF)

# Summary of 8086 Interrupt Function …

- **How does 8086 get to Interrupt Service Routine (ISR)?**
  - Simple. It loads its CS and IP registers with the address of ISR.
  - So, the next instruction to be executed is the first instruction of ISR.

- **How does 8086 get the address of Interrupt Service Routine (ISR)?**
  - It goes to **specified memory location** to fetch **four consecutive bytes**
    - higher two bytes to be used as CS (Code Segment)
    - lower two bytes to be used as IP (Instruction Pointer)

# Summary of 8086 Interrupt Function …

- **How does 8086 get the address of that specified memory location?**
  - In an 8086 system, the first 1Kbytes of memory, from 00000 to 003FF, is set aside as a **Table** for storing the starting addresses of **Interrupt Service Routines** (ISR).
  - Since 4 bytes are required to store **CS and IP** values for each ISR, the **Table** can hold the starting addresses for up to 256 ISRs.

# Summary of 8086 Interrupt Function …

- **How does 8086 get the address of a particular ISR?**
  - In an 8086 system, each "interrupter" has an id #
  - 8086 treat this id # as interruption type #
  - After receiving INTR signal, 8086 sends an INTA signal
  - After receiving INTA signal, interrupter releases it's id #, i.e., type # of the interruption.
  - 8086 multiplies this id# or type# by 4 to produced the desired address in the ***vector table***
  - 8086 reads 4 bytes of memory starting from this address to get the starting address of ISR
    - lower 2 byte is loaded in to IP
    - higher 2 bytes to CS

# Summary of 8086 Interrupt Function …

- **What happens if two or more interrupts occur at the same time?**
  - Higher priority interrupts will be served first

| Interrupt Type | Priority |
|---|---|
| Reset | HIGHEST |
| DIVIDE ERROR | |
| INT n, INTO | |
| NMI | |
| INTR | |
| SINGLE STEP | LOWEST |

# Thank You !!

# 8086 System Connections and BUS Timing

***Course Teacher:***

**Md. Obaidur Rahman, Ph.D.**
Professor
Department of Computer Science and Engineering (CSE)
Dhaka University of Engineering & Technology (DUET), Gazipur.

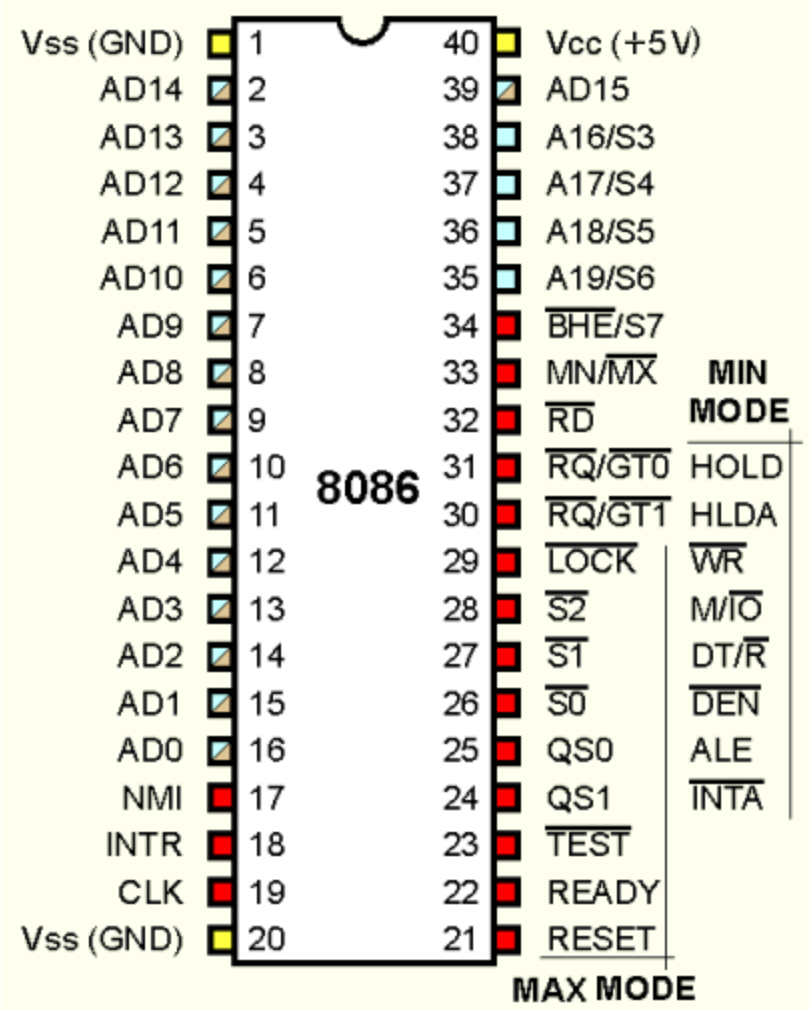
**Course ID:** CSE - 4503
**Course Title:** Microprocessors and Assembly Language
Department of Computer Science and Engineering (CSE),
Islamic University of Technology (IUT), Gazipur.

# Lecture References:

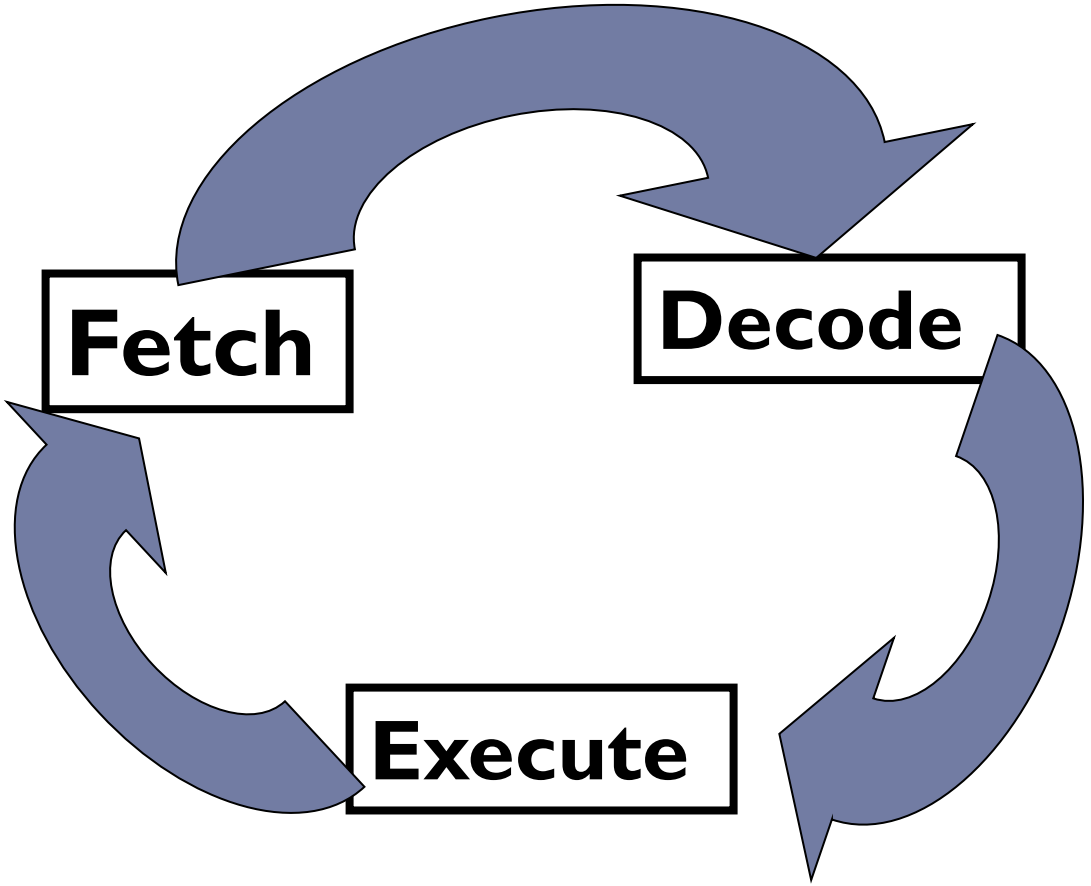
- **Book:**
  - *Microprocessors and Interfacing: Programming and Hardware, Chapter # 7,* **Author:** Douglas V. Hall.

- **Lecture Materials:**
  - M. A. Sattar, Microprocessor and Microcontroller, BUET.
  - Md. Omar Faruque, BRAC University.

# Remember !! 8086 Pin Specifications



# Microprocessor Operation

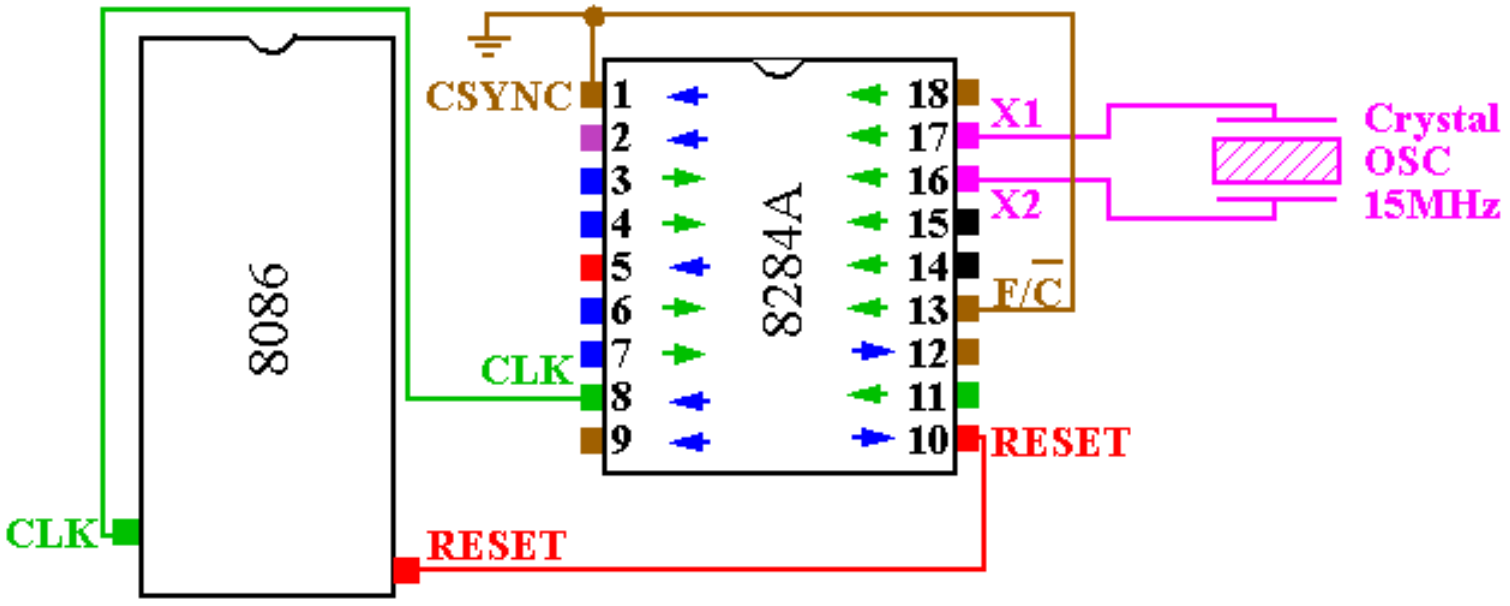


# Microprocessor Operation

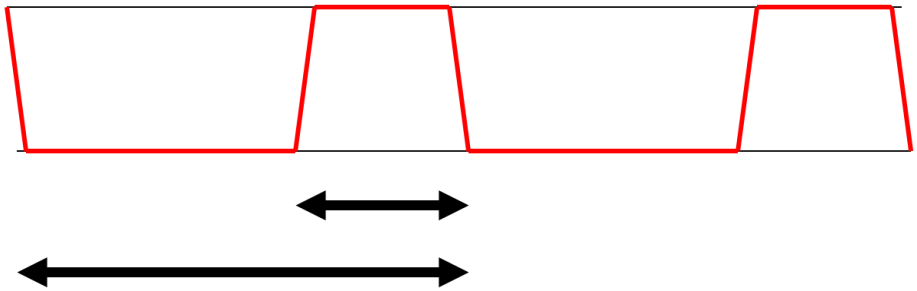
- The time a µP requires to complete **fetch-decode-execute** operation of a single instruction is known as ***Instruction Cycle***
- An ***Instruction Cycle*** consists of one or more ***Machine Cycles***
- A basic µP operation such as reading or writing a byte/word from or to memory or I/O port is called a ***Machine Cycle or Bus cycle***
- A ***Machine (bus) cycle*** consists of at least **four** clock cycles, called **T** states.
- One cycle of a clock is called a **State**

# Clock Generation

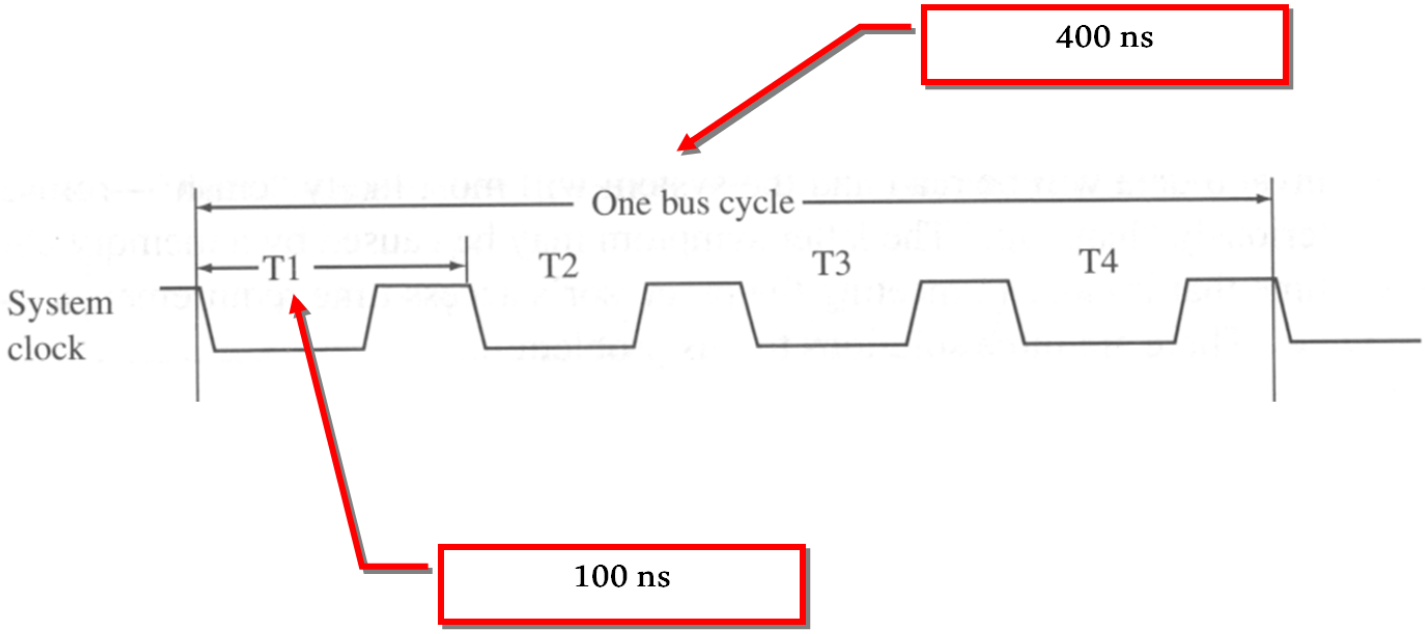
- Clock generator circuit is 8254A and connected to **pin 19** (**CLK**) of 8086.



# System Clock Concept



33% Duty Cycle



# System Clock Concept

- 8086 is found to operate in between 5 to 10 Mhz.
- Each **bus cycle** consists of at least 4 **clock cycles**.
- An 8086 running at 5MHz, it's clock pulses will be of 200ns and it would take 800ns for a complete bus cycle.
- Again, an 8086 running at 10MHz, it's clock pulses will be of 100ns and it would take 400ns for a complete bus cycle.
- Each **read** or **write** operation take 1 bus cycles.

# Clock States

- **Why are there T states?**
  - In the 8086, the address and data lines are multiplexed.
  - The microprocessor needs time to change the signals during each bus cycle.
  - Memory devices need time to interpret the address value and then **read/write** the data (*access time*)

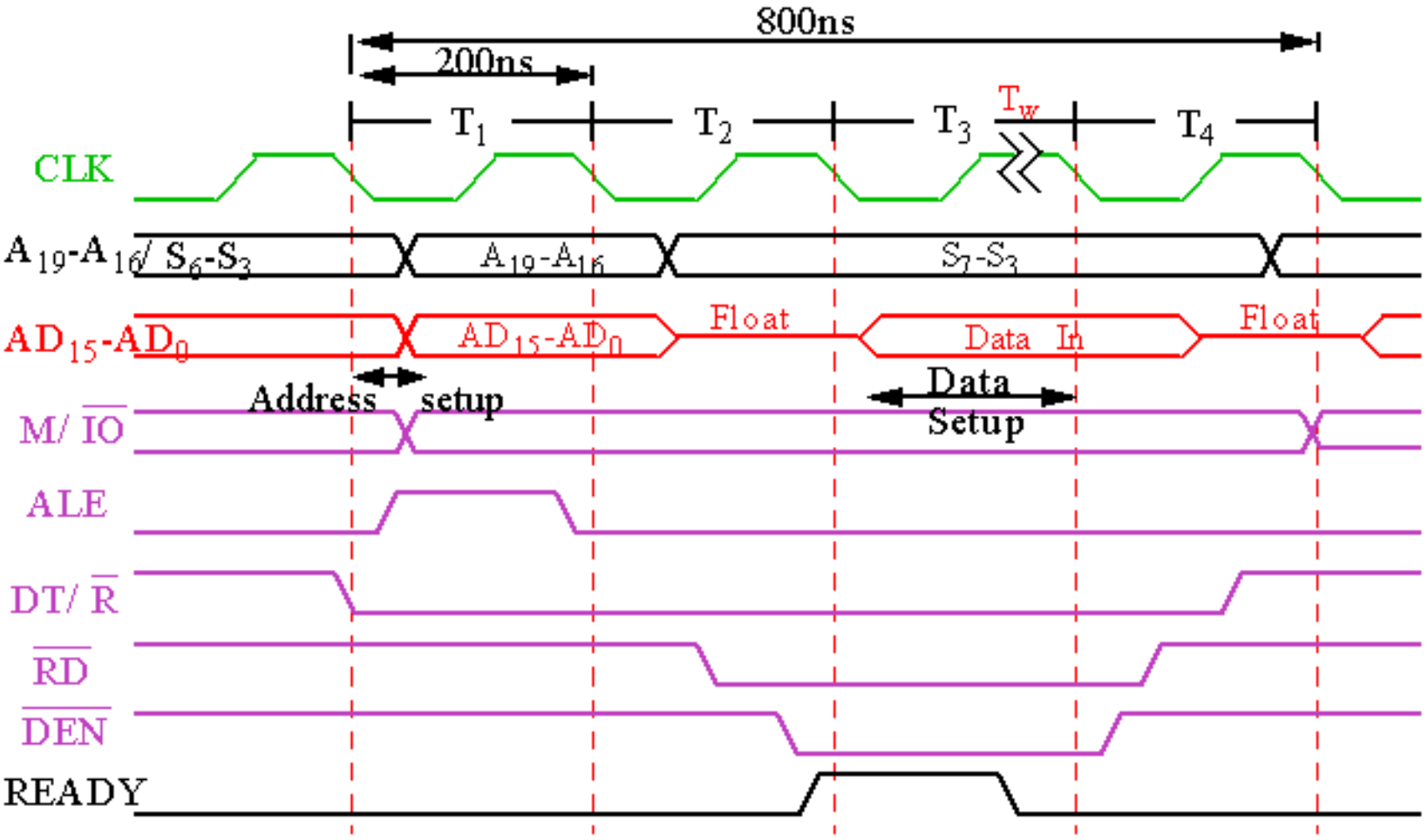
# Clock States

- A specific, defined action occurs during each T states (labeled $T_1 - T_4$)
- **$T_1$: *Address is output***
  - Address of memory or I/O is sent out by 8086 via address bus
  - Used Control signals: ALE, DT/R', M/IO' shows some output
- **$T_2$: *Bus cycle type (MEMORY/IO, READ/WRITE)***
  - 8086 issues either RD' or WR' and DEN'
  - In case of **WRITE (WR)** operation, data to be written appear on data bus

# Clock States

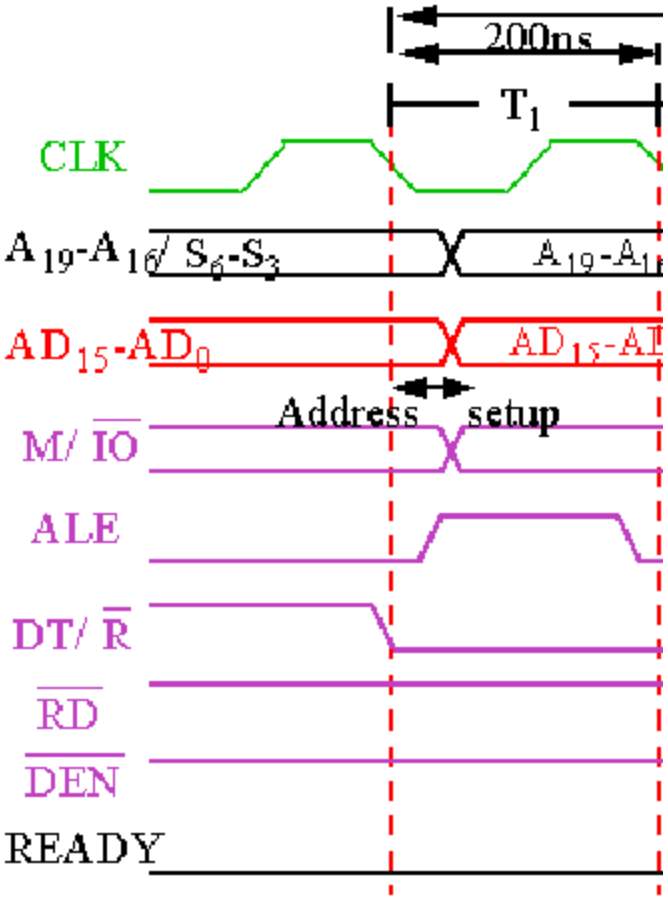
- **$T_3$: *Data is supplied***
  - READY is sampled at the end of T2
    - If READY is low, T3 becomes a wait state (TW), means no operation (NOP).
    - In **READ** bus cycle data bus is sampled at end of $T_3$
- **T4: *Data latched by μP, and control signals removed***
  - All bus signals deactivated in preparation for next bus cycle
  - μP sampled data bus for data that read from M or I/O
  - At trailing edge of WR', transfer data to M or I/O

# READ BUS Timing (Complete BUS Cycle)




**Bus Timing for a Read Operation**

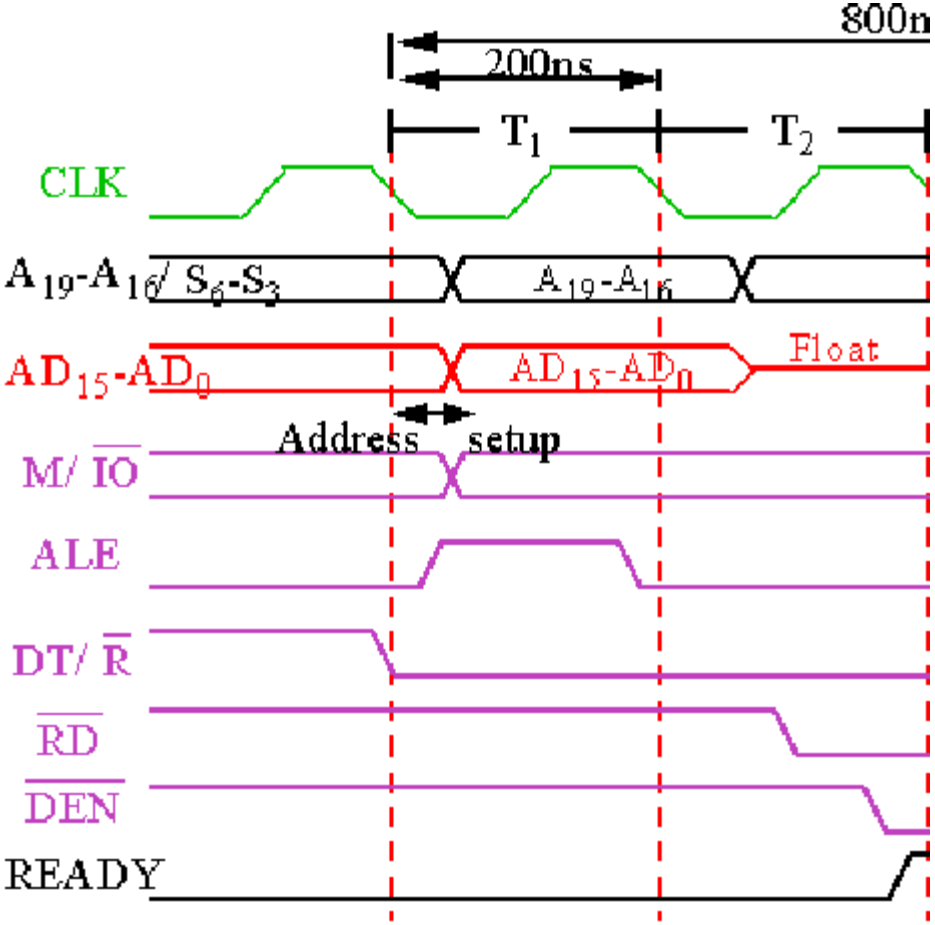
# READ BUS Timing (During $T_1$ State)




***During $T_1$ :***

- *The address is placed on the Address/Data bus.*
- *Control signals*
  - ***M/ IO'*** *specify memory or I/O,*
  - ***ALE*** *latch the address onto the address bus and*
  - ***DT/R'*** *set the direction of data transfer on data bus.*

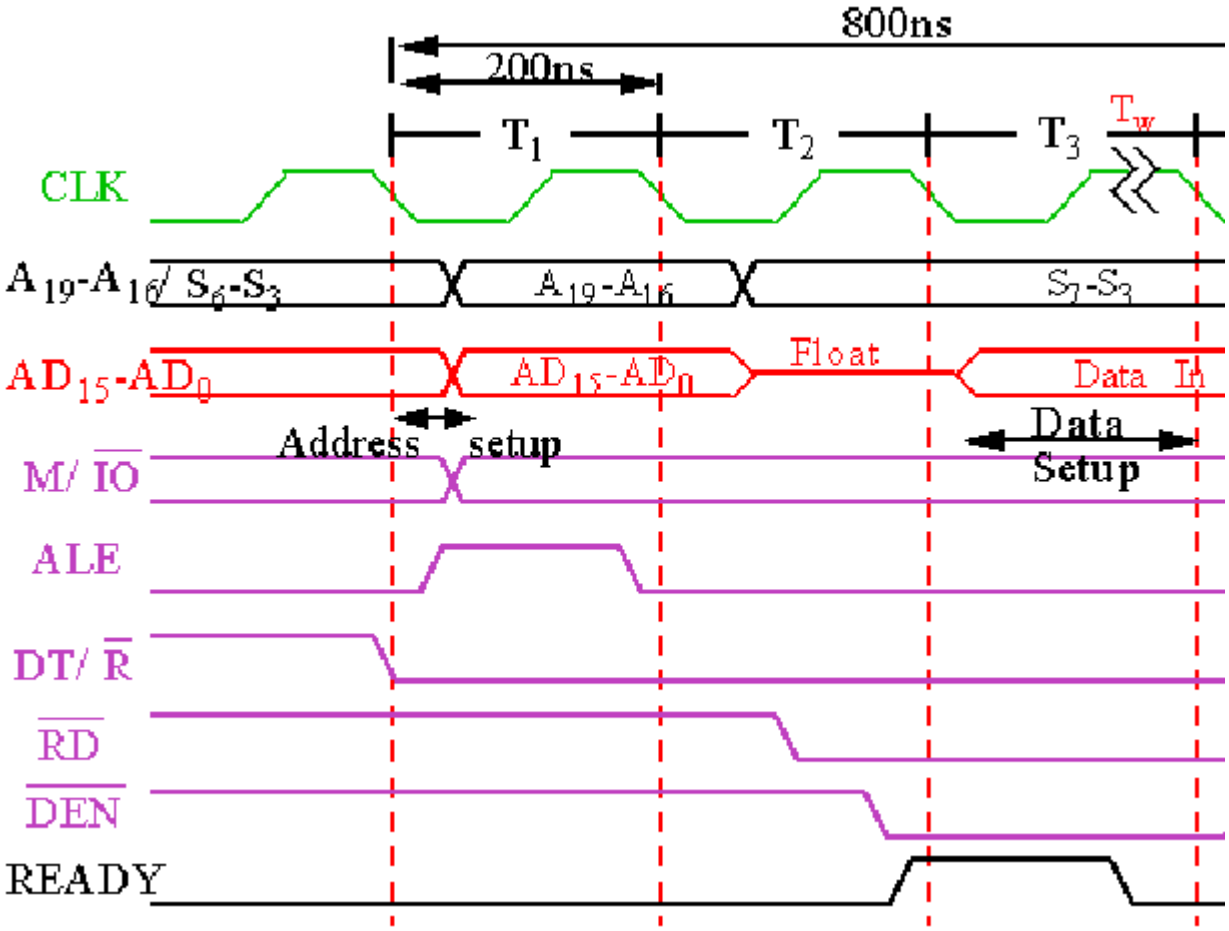
# READ BUS Timing (During $T_2$ State)




***During $T_2$ :***

- *8086 issues the* **RD'** *(or* **WR'** in case of write operation) *signal.*
- **DEN'** *enables the 8086 to receive the data for* ***READ*** *operation (or the memory or I/O device to receive the data for* ***WRITE*** *operation).*

# READ BUS Timing (During $T_3$ State)

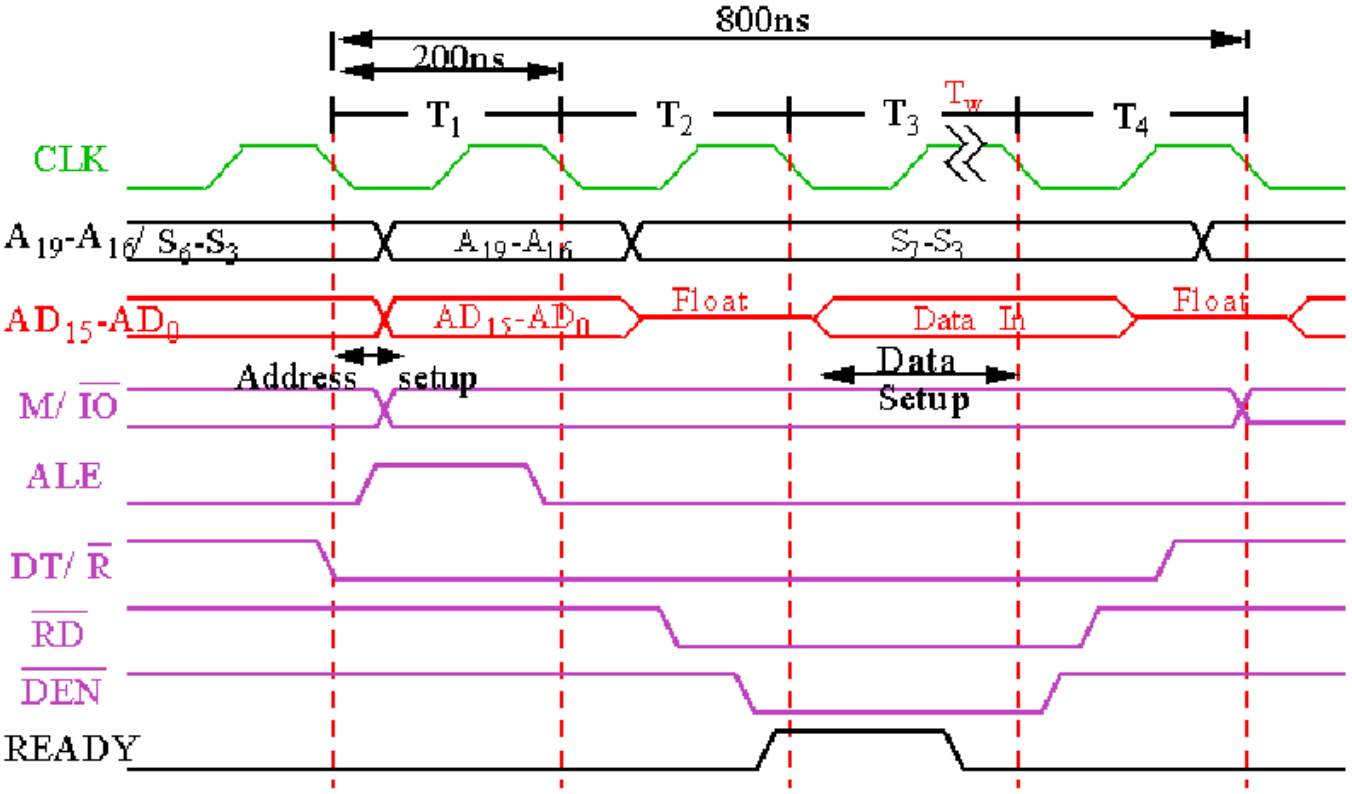



***During $T_3$ :***

- *This cycle is provided to allow memory to access data.*

- ***READY** is sampled at the end of $T_2$ .*
  - *If low, $T_3$ becomes a **wait** state.*
  - *Otherwise, the data bus is sampled at the end of $T_3$ .*

# READ BUS Timing (During $T_4$ State)

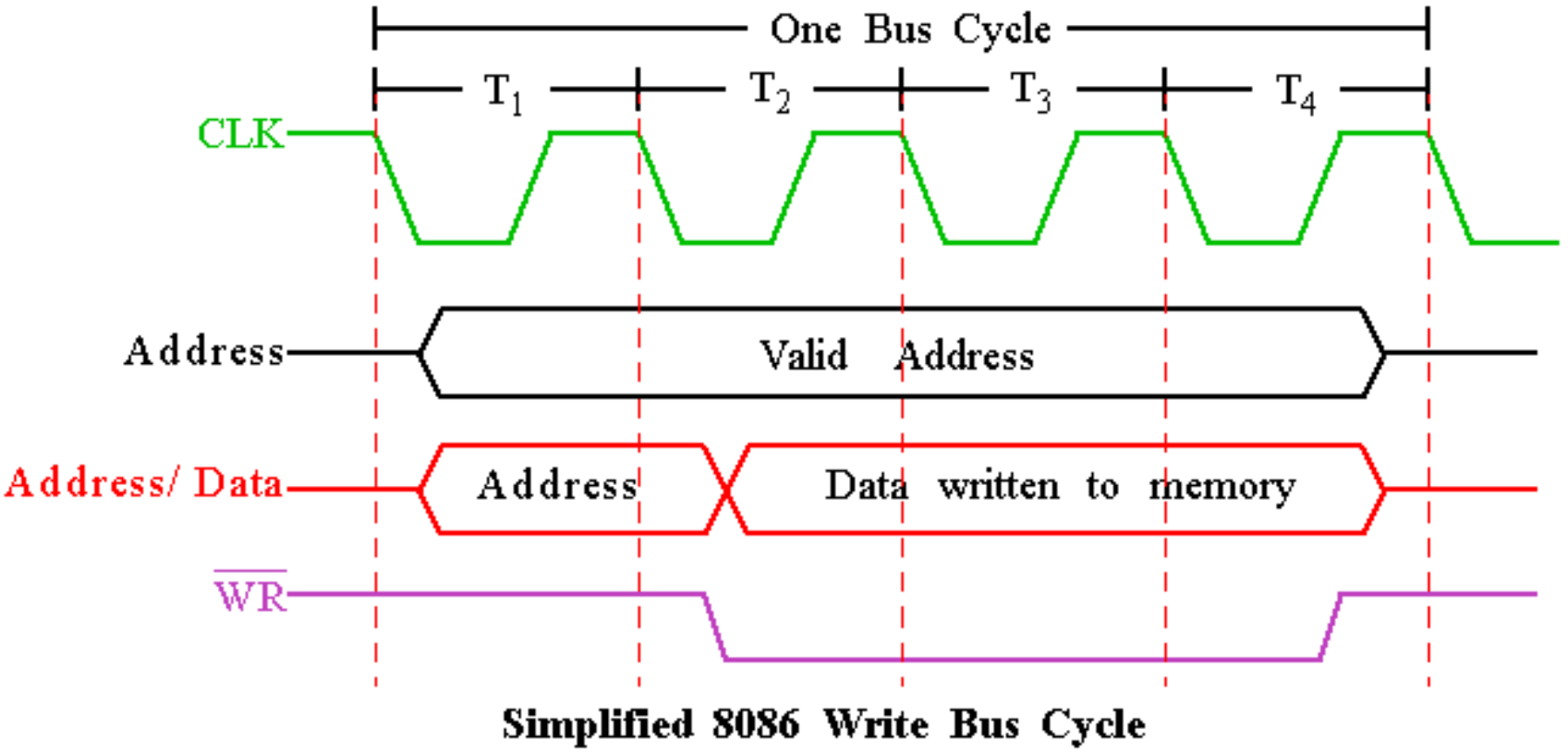



Bus Timing for a Read Operation

***During $T_4$ :***

- *All bus signals are deactivated, in preparation for next bus cycle.*
- *Data is sampled for **READ** (or **WRITE** occurs for write) data.*

# WRITE BUS Timing




Simplified 8086 Write Bus Cycle

- What are the functions of each pin in different **T states** during **WRITE** operation ??

# Thank You !!

# 80186 and 80286 Microprocessor

***Course Teacher:***

**Md. Obaidur Rahman, Ph.D.**
Professor
Department of Computer Science and Engineering (CSE),
Dhaka University of Engineering & Technology (DUET), Gazipur.

**Course ID:** CSE - 4503
**Course Title:** Microprocessors and Assembly Language
Department of Computer Science and Engineering (CSE),
Islamic University of Technology (IUT), Gazipur.

# Lecture References:

- Book:
  - *Microprocessors and Interfacing: Programming and Hardware*, Chapter # 15, **Author:** Douglas V. Hall
  - *The Intel Microprocessor 8086…Arch. Prog, Interfacing.* **Author:** Bary, Bray
  - *Microprocessor and Microcomputer – Based System Design*, **Author:** Mohamed Rafiquzzaman

- Lecture Materials:
  - *M. A. Sattar, BUET, Dhaka, Bangladesh.*

# 80186 Microprocessor

- 80186 contains 8086 processor and several additional functional chips:
  - Clock generator
  - 2 independent DMA channels
  - PIC (Programmable IC)
  - 3 programmable 16-bit timers
- It is more of a microcontroller than a microprocessor
- Used mostly in industrial control applications

# Intel 80286: Salient Features

- High performance microprocessor with memory management and protection
  - 80286 is the first member of the family of advanced microprocessors with built-in/on-chip memory management and protection abilities primarily designed for multi-user/multitasking systems
- Available in 12.5MHz, 10MHz & 8MHz clock frequencies

# Intel 80286: Salient Features

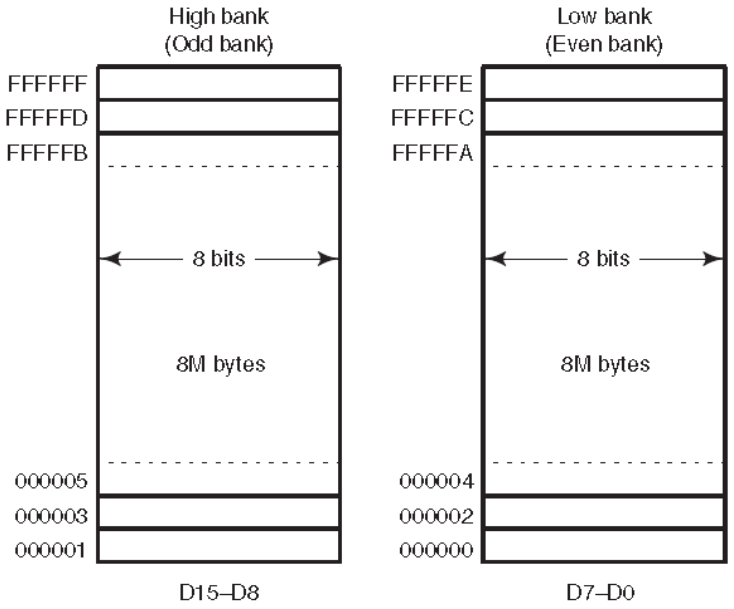
- The 80286 CPU, with its 24-bit address bus is able to address 16MB of physical memory.
- 1GB of virtual memory for each task

| Microprocessor | Data bus width | Address bus width | Memory size |
|---|---|---|---|
| 8086 | 16 | 20 | 1M |
| 80186 | 16 | 20 | 1M |
| **80286** | **16** | **24** | **16M** |

# Intel 80286: Salient Features

Intel 80286 has 2 operating modes:

- **Real Address Mode :**
  - 80286 is just a fast 8086 - up to 6 times faster
  - All memory management and protection mechanisms are disabled
  - It allows the microprocessor to address only the first 1M byte of memory space.
  - The first 1M byte of memory is called the real memory, conventional memory, or DOS memory system.
  - Windows does not use the real mode.
  - The concept of **Segment** and **Offset** is used.



# Intel 80286: Salient Features

Intel 80286 has 2 operating modes:

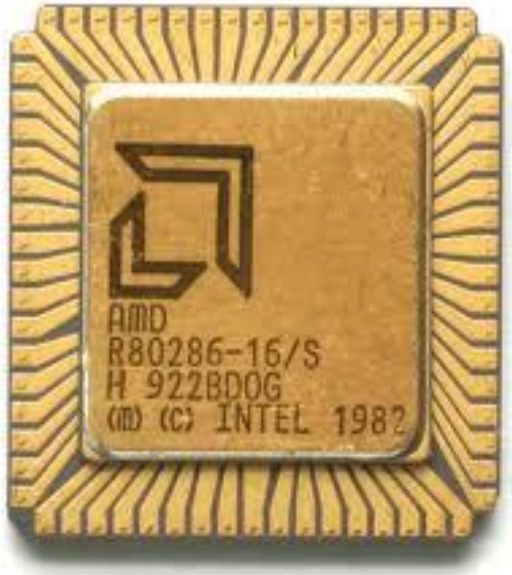
- **Protected Virtual Address Mode**
  - 80286 works with all of its memory management and protection capabilities with the advanced instruction set.
  - It uses all 24 address lines to access upto 16 Mbytes of physical memory and provides upto Gigabyte range virtual memory.
  - **Protected mode** is where Windows operates.

# Memory Management Unit

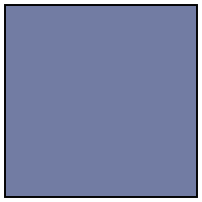
- An 80286 switches to protected mode and start using virtual memory.
- In protected mode, the concept of **Segment** is not used.
- A 80286 virtual address consists of a 16-bit **Selector** and 16-bit **Offset**.
- A memory management unit (MMU) uses 16-bits of selector to access a **Descriptor** for the desired segment in a table of descriptors.
- Each 80286 descriptor describes a 64K-byte memory segment and the 80286 allows 16K descriptors. This (64K × 16K) allows a maximum of 1G bytes of memory to be described for the system.
- The descriptor contains the 24-bit physical address.

# Intel 80286: Salient Features

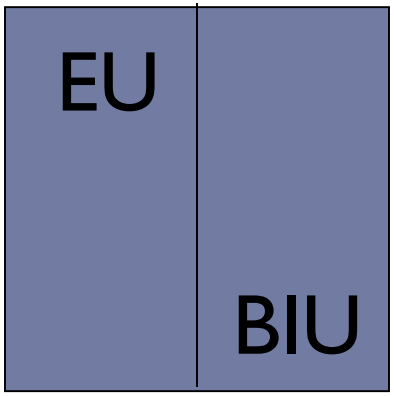
- 80286 includes special instructions to support operating system.
  - for example, one instruction can
    - i) End the current task
    - ii) Save its states
    - iii) Switch to a new task
    - iv) Load its states and
    - v) Begin executing the new task
- Housed in 68-pin package
- 1,34,000 Transistors



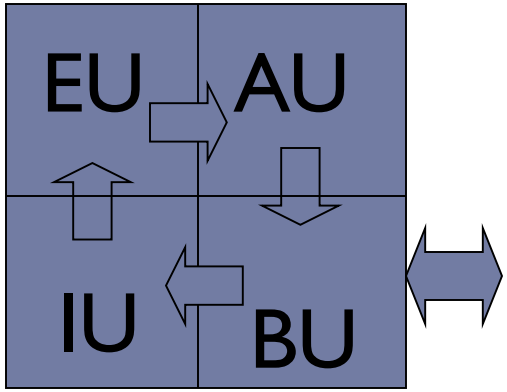
# Internal Block Diagram of 80286



8085

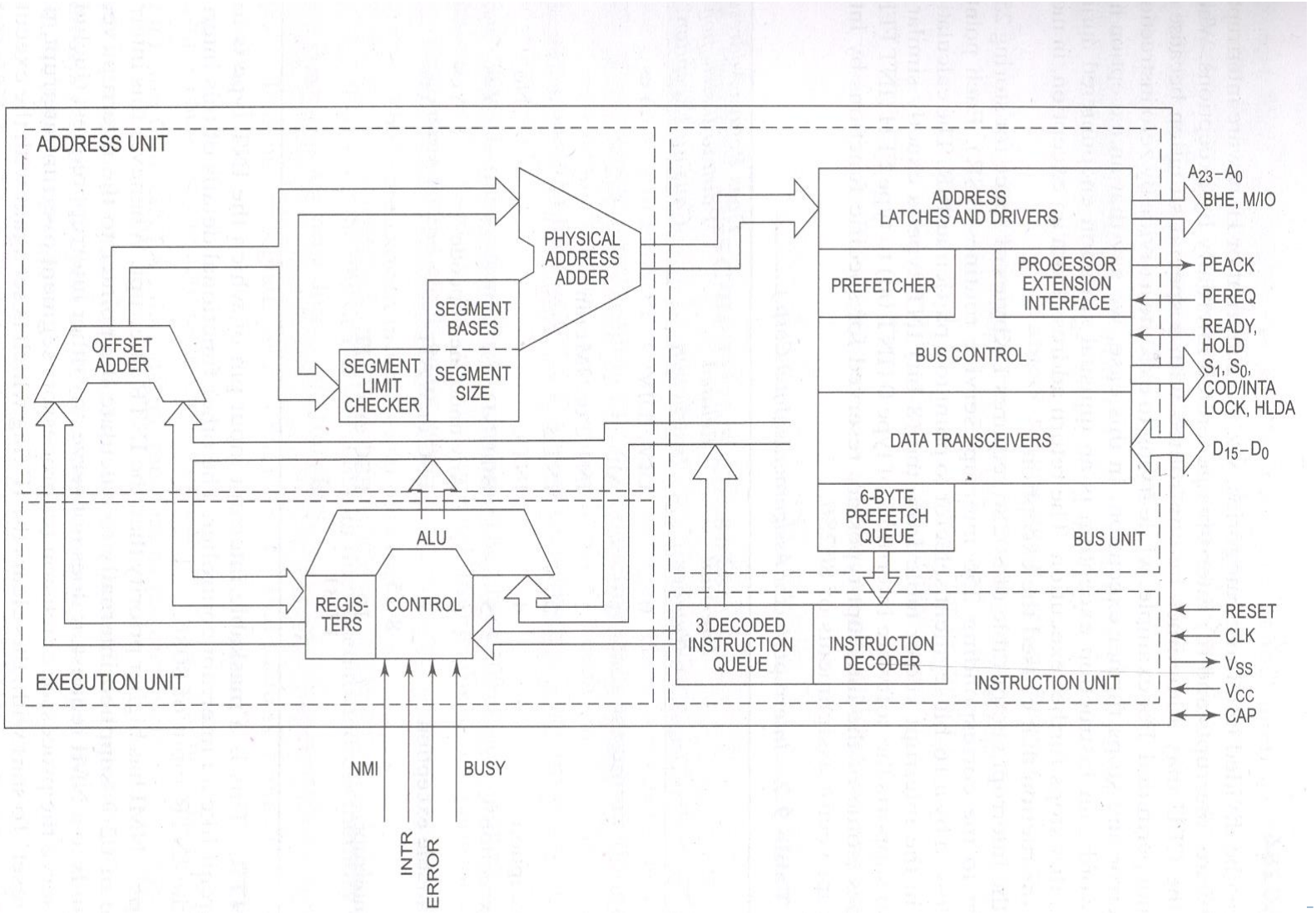



8086




80286

# Internal Block Diagram of 80286



# Functional Parts of 80286

- Address Unit (AU)
- Bus Unit (BU)
- Instruction Unit (IU)
- Execution Unit (EU)

# Address Unit (AU)

- Calculates the physical addresses of the instruction and data that the CPU want to access
- Address lines derived by this unit may be used to address different peripherals.
- Physical address computed by the Address Unit (AU) is handed over to the BUS Unit (BU).

# BUS Unit (BU)

- Performs all memory and I/O **read** and **write** operations.
- Take care of communication between CPU and a coprocessor.
- Transmit the physical address over address bus $A_0$ – $A_{23}$.
- Pre-fetcher module in the BU performs the task of pre-fetching.
- Bus controller controls the pre-fetcher module.
- Fetched instructions are arranged in a 6 – byte pre-fetch queue.

# Instruction Unit (IU)

- IU receives arranged instructions from 6 byte pre-fetch queue.
- Instruction decoder decodes up to 3 pre-fetched instruction and are latched them onto a decoded instruction queue.
- Output of the decoding circuit drives a control circuit in the Execution Unit (EU).

# Execution Unit (EU)

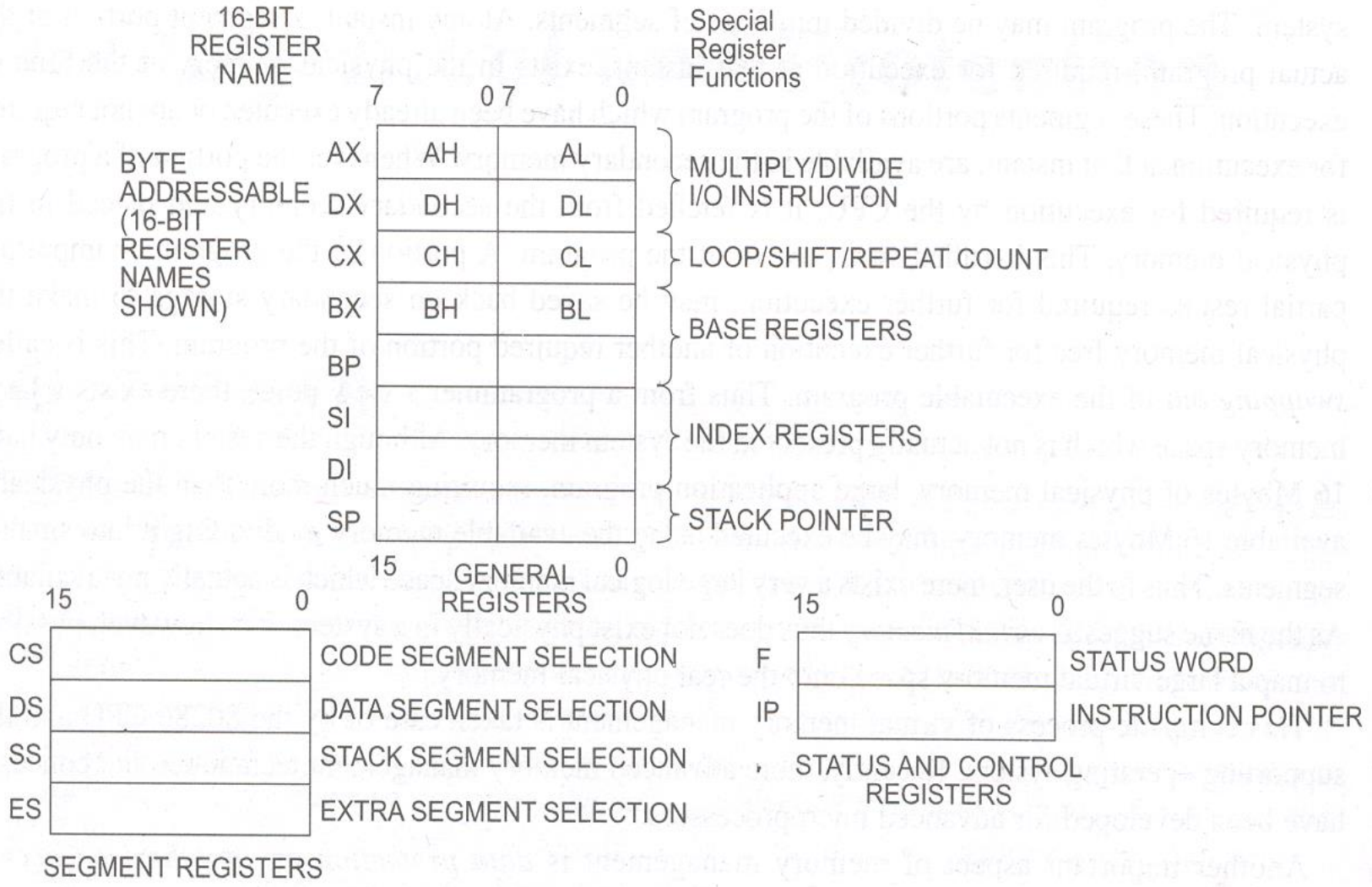
- EU executes the instructions received from the decoded instruction queue sequentially.
- Contains Register Bank.
- Contains one additional special 16-bit register called **Machine status word (MSW)** register **---** lower 4 bits are only used.
- ALU is the heart of Execution Unit (EU).
- After execution ALU sends the result either over data bus or back to the register bank.

# Register Organization of 80286

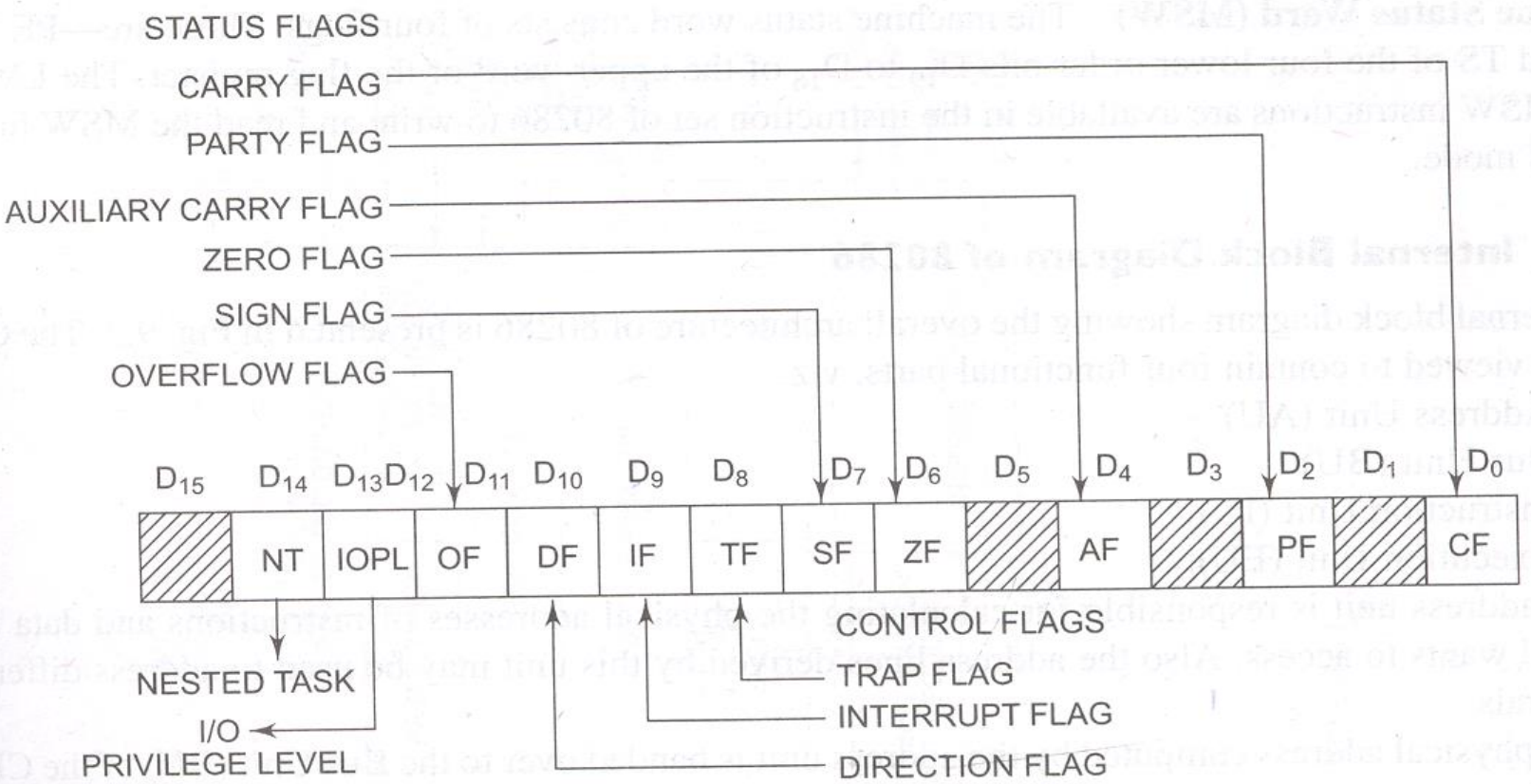
The 80286 CPU contains the same set of registers, as in 8086.

- Eight 16-bit general purpose registers (Data, Base-pointer and Index)
- Four 16 bit segment registers (Segment)
- Status and control register (Flag)
- Instruction pointer (IP)
- Machine Status Word (MSW)

# Register Organization of 80286



# Only Change in Flag Register



# IOPL – Input Output Privilege Level Flags (Bit $D_{12}$ and $D_{13}$)

- IOPL is used in protected mode operation to select the privilege level for I/O devices.
- If the current privilege level is higher or more trusted than the IOPL, I/O is executed without hindrance.
- Note that IPOL 00 is the highest or more trusted and IOPL 11 is the lowest or least trusted.
- If the IOPL value is lower than the current privilege level, an interrupt occurs, causing execution to suspend.

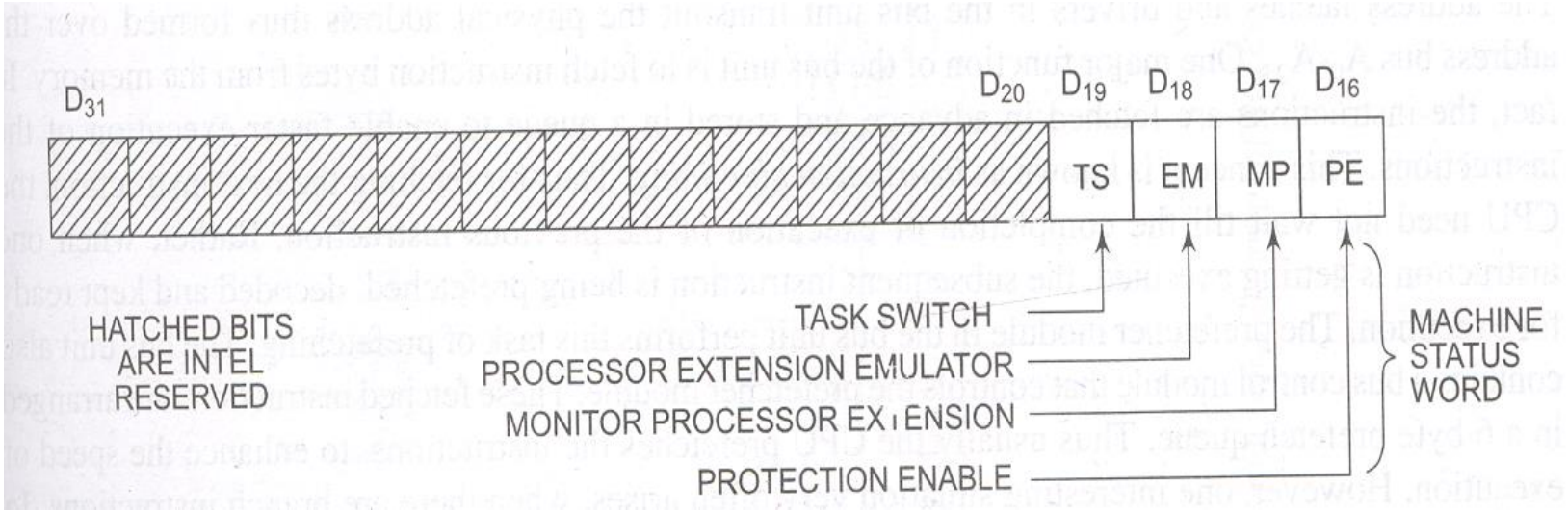
# NT – Nested Task Flag (Bit $D_{14}$)

- When NT is set, it indicates that one system task has invoked another through a CALL instruction as opposed to a JMP.
- For multitasking this can be manipulated to have advantage.

# Machine Status Word (MSW) Register

- Consist of four flags. These are – **PE**, **MP**, **EM** and **TS**

- **Instructions** are available in the instruction set of 80286 to **write** and **read** the MSW in real address mode.



# Machine Status Word (MSW) Register

- PE – Protection Enable
  - Protection enable flag places the 80286 in protected mode, If it is set. This can only be cleared by resetting the CPU.
- MP – Monitor Processor Extension
  - Flag allows WAIT instruction to generate a processor extension (when any Coprocessor is connected or attached).
- EM – Emulate Processor Extension Flag
  - If set , causes a processor extension (Coprocessor) is absent and permits the emulation of processor extension by CPU.
- TS – Task Switch
  - This flag permits the CPU to test whether the current processor extension is for current task or not. As the time progresses, it might be needed to change the current task; to set aside logically the segments that comprise current task and make sub-segment for another task.

# Thank You !!

# 80386 Microprocessor

***Course Teacher:***

**Md. Obaidur Rahman, Ph.D.**
Professor
Department of Computer Science and Engineering (CSE)
Dhaka University of Engineering & Technology (DUET), Gazipur.

**Course ID:** CSE - 4503
**Course Title:** Microprocessors and Assembly Language
Department of Computer Science and Engineering (CSE),
Islamic University of Technology (IUT), Gazipur.

# Lecture References:

- Book:
  - *Microprocessors and Interfacing: Programming and Hardware*, Chapter # 15, **Author:** Douglas V. Hall
  - *The Intel Microprocessor 8086…Arch. Prog, Interfacing.* **Author:** Bary, Bray
  - *Microprocessor and Microcomputer – Based System Design*, **Author:** Mohamed Rafiquzzaman

- Lecture Materials:
  - *M. A. Sattar, BUET, Dhaka, Bangladesh.*

# Limitations of 80286

- 16-bit ALU.
- 64K segment size for the programs.
- 1 GB of virtual memory
- Cannot be easily switched back and forth between **real** and **protected mode**
  - To come back to the **real mode** from **protected mode**, it is needed to switched off the 80286.

# 80386 Overcomes 80286 Limitations

- It has 32 bit ALU.
- Segment size can be as large as 4GB
  - A program can have as many as 16K segments
  - So, a program has access to 4GBx16K=64TB of virtual memory
- 80386 has a ***virtual 86 mode*** which allows easy switching between **real** and **protected modes**.

# 80386: Salient Features

- Alternatively referred to as a **386** or the **i386**
- Intel introduced the first 32-bit chip, 80386, in October 1985 as an upgrade to the 80286 processor
- Intel stopped producing 386 since September 2007.
- 386 incorporates 275,000 transistor
- 386 was capable of performing more than five million instructions every second (**MIPS**)
- 386 was available in clock speeds between 12 and 40MHz.

# Versions of 80386

- Two versions were commonly available:
  1) 80386 DX
  2) 80386 SX

- The original 80386 processor was renamed as 80386DX or 386DX after introducing 386SX.

- 80386SX was introduced in 1988 as a low cost solution alternative to the original 80386.

- 80386SX was developed after the DX, for the application that didn't require the full 32-bit capabilities.

# Versions of 80386

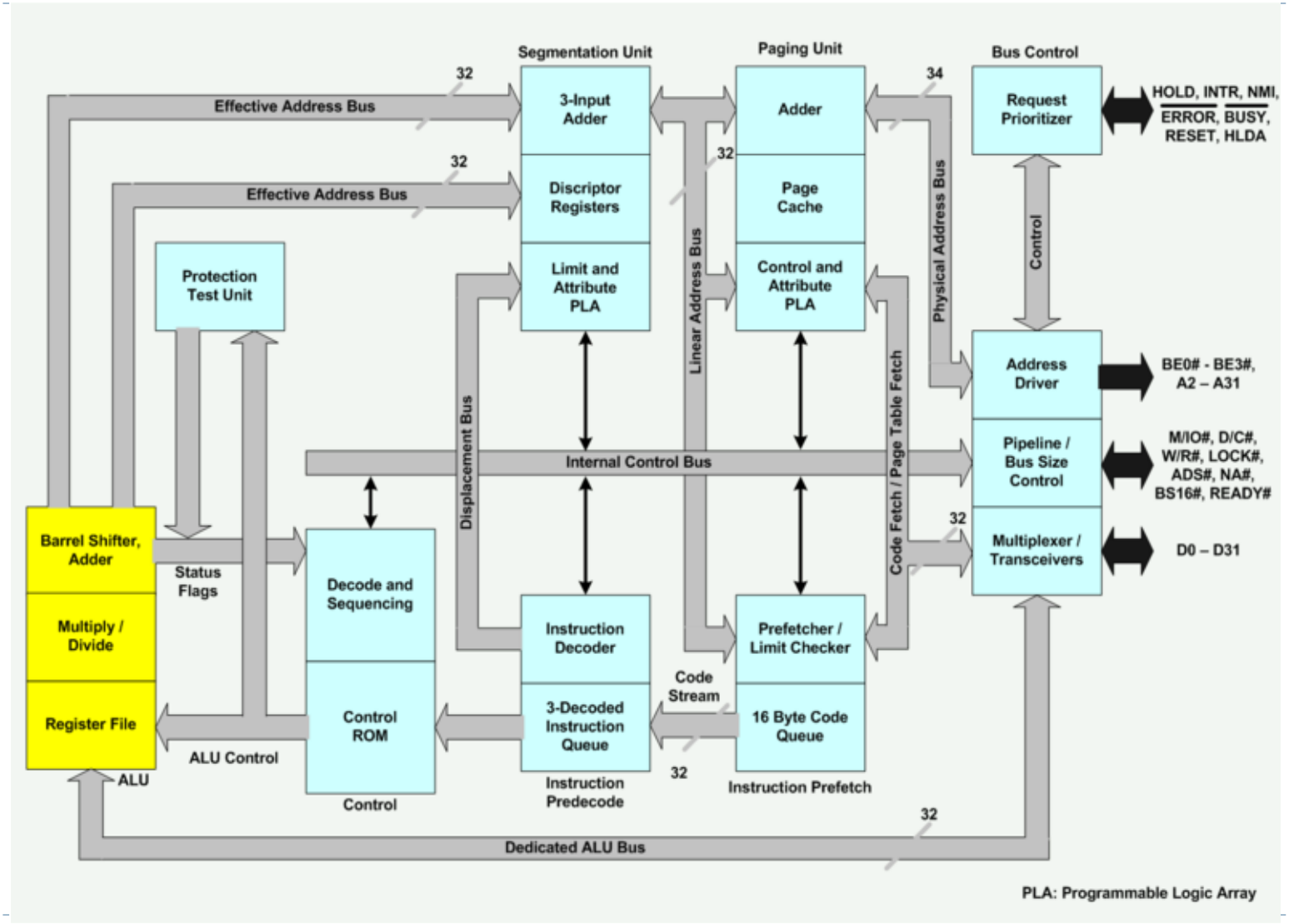
- It  is found in many PCs where it uses the same basic mother board design as the 80286.
- Most application need less than the 16MB of memory, so the SX is popular and less costly version of the 80386 microprocessor.
- The 80386SX lacked a math coprocessor but still featured the 32-bit architecture and built-in multitasking.
- The chip was available in clock speeds of 16MHz, 20MHz, 25MHz, and 33MHz.

# 80386DX Vs. 80386SX

| 80386DX | 80386SX |
|---|---|
| 32 bit address bus<br>32 bit data bus | 24 bit address bus<br>16 bit data bus |
| Packaged in 132 pin | 100 pin flat package |
| Address 4GB of memory | 16 MB of memory |

- Both have the same internal architecture.
- Lower cost package and the ease of interfacing to 8-bit and 16-bit memory and peripherals make SX suitable for use in low cost systems.

# Internal Block Diagram of 80386



# Architecture of 80386: **Instruction Unit**

- The Instruction unit decodes the op-code bytes received from the 16-byte **instruction code queue** and arranges them in a 3-instruction decoded instruction queue.
- After decoding them pass it to the control section for deriving the necessary control signals.
- The barrel shifter increases the speed of all shift and rotate operations.

# Architecture of 80386: **Instruction Unit**

- The multiply / divide logic implements the **bit-shift-rotate** algorithms to complete the operations in minimum time.

- Even 32- bit multiplications can be executed within one microsecond by the multiply / divide logic.

# Architecture of 80386: **Segmentation Unit**

- Segmentation unit allows the use of two address components, viz. **segment** and **offset** for relocate ability and sharing of code and data.
- Segmentation unit allows segments of size 4Gbytes at max.
- The Segmentation unit provides a 4 level protection mechanism for protecting and isolating the system code and data from those of the application program.

# Architecture of 80386: **Paging Unit**

- The Paging unit organizes the physical memory in terms of pages of 4kbytes size each.
- Paging unit works under the control of the segmentation unit, i.e. each segment is further divided into pages.
- The virtual memory is also organizes in terms of segments and pages by the memory management unit.
- Paging unit converts linear addresses into physical addresses.

# Architecture of 80386: **Bus Control Unit**

- The Bus control unit has a prioritizer to resolve the priority of the various bus requests.
- This controls the access of the bus.
- The address driver drives the bus enable and address signal A2 – A31.
- The pipeline and dynamic bus sizing unit handle the related control signals.
- The data buffers interface the internal data bus with the system bus.

# 80386 Data Bus

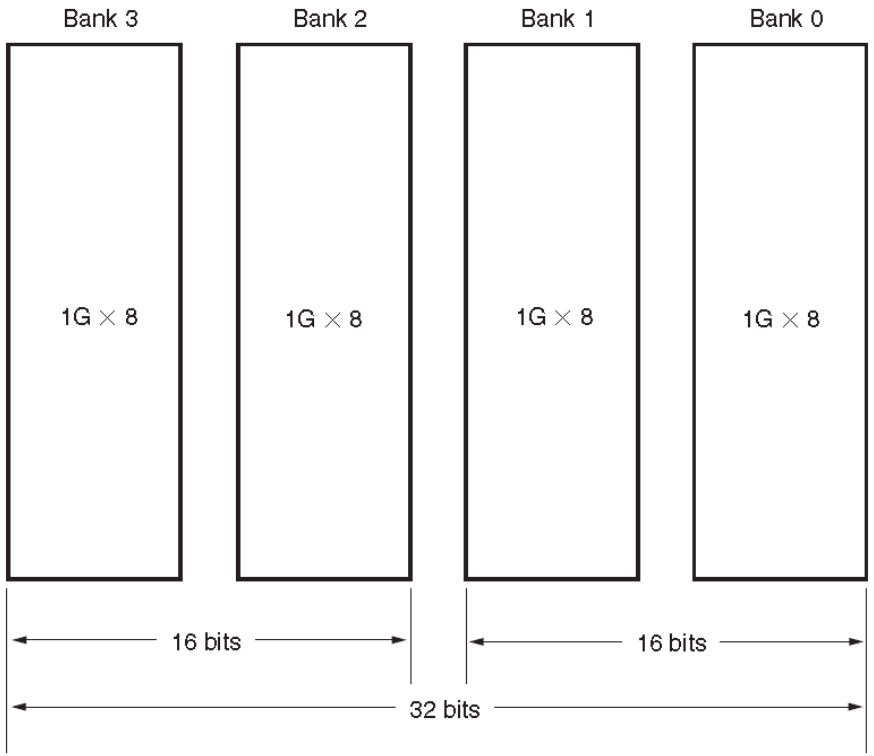
- 32-bit data bus
- D0 through D31 (Data Bus)
- Bi-Directional

# 80386 Address Bus

- Address bus consists of A2 to A31 address lines and BE0 to BE3 byte/bank enable lines
- No A0 & A1 address lines are available in 386
  - they are internally decoded to produce BE0 to BE3 signals

# 80386 Address Bus

- **BE0 through BE3**
  - Byte (Bank???) Enable lines
  - Memory are arranged in 4 Banks
- BE0-BE3 also allow 80386 to transfer ***byte***, ***word*** and ***double word***



# Register Organization of 80386

- **Segment Descriptor Registers:**
  - This registers are not available for programmers, rather they are internally used to store the descriptor information, like attributes, limit and base addresses of segments.
  - *Six Segment Registers* have corresponding six 73 bit descriptor registers.
  - Each of them contains 32 bit base address and 32 bit base limit and 9 bit attributes.
  - These are automatically loaded when the corresponding segments are loaded with selectors.

# Register Organization of 80386

- **System Address Registers:**
  - Four special registers are defined to refer to the descriptor tables supported by 80386.
  - The 80386 supports four types of descriptor table, viz.
    - Global descriptor table (GDT)
    - Interrupt descriptor table (IDT)
    - Local descriptor table (LDT) and
    - Task state segment descriptor (TSS)

# Register Organization of 80386

- **Control Registers:**
  - The 80386 has three (3) 32 bit control registers CR0, CR2 and CR3.
  - These hold global machine status independent of the executed task.
  - Load and store instructions are available to access different registers of 80386 microprocessor.

# Register Organization of 80386

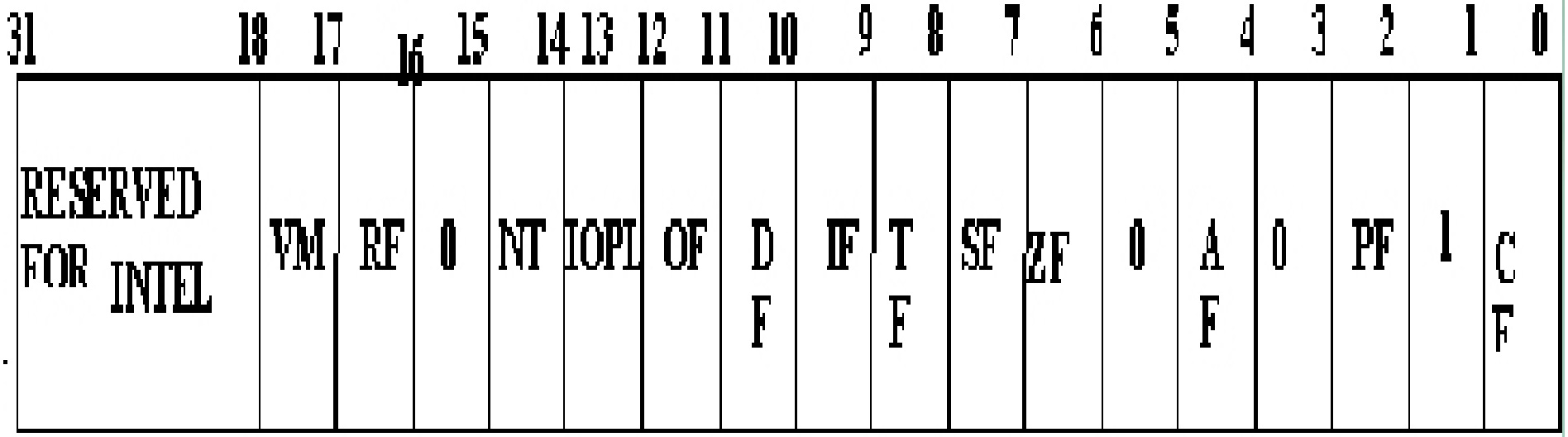
- **Debug and Test Registers:**
  - Intel has provide a set of 8 debug registers for hardware debugging.
  - Out of these eight registers DR0 to DR7, two registers DR4 and DR5 are Intel reserved.
  - **The initial four registers DR0 to DR3** store four program controllable breakpoint addresses, while DR6 and DR7 respectively hold breakpoint status and breakpoint control information.
  - **Two more test register** are provided by 80386 for page caching namely test control and test status register.

# Register Organization of 80386

- **Flag Register:**
  - The Flag register of 80386 is a 32 bit register.
  - Out of the 32 bits, Intel has reserved bits D18 to D31, D5 and D3 and set to 0.
  - While D1 is always set at 1.
  - Two extra new flags are added to the 80286 flag to derive the flag register of 80386.
  - They are VM and RF flags.

# Register Organization of 80386

Old flags of 286




New flags for 386

# Register Organization of 80386

- **VM - Virtual Mode Flag in Flag Register**
- If this flag is set to 1, the 80386 enters the virtual 8086 mode within the protection mode.
- When VM bit is 0, 386 operates in protected mode
- This is to be set only when the 80386 is in protected mode.
- This bit can be set using IRET instruction or any task switch operation only in the protected mode.

# Register Organization of 80386

- **Resume Flag (RF) in Flag Register**
- If RF=1, then 80386 ignores debug faults
  - Does not take another exception so that an instruction can be restarted after a normal debug exception.
- If RF=0, then 80386 takes another debug exception to service debug faults
- This flag is used with the debug register breakpoints.

# Register Organization of 80386

- **Resume Flag (RF) in Flag Register**
- It is checked at the starting of every instruction cycle and if it is set, any debug fault is ignored during the instruction cycle.
- The RF is automatically reset after successful execution of every instruction, except for IRET and POPF instructions.
- Also, it is not automatically cleared after the successful execution of JMP, CALL and INT instruction causing a task switch.
- These instruction are used to set the RF to the value specified by the memory data available at the stack.

# 80386 Modes of Operation

- There are 3 modes of operations:
  - **Real Mode**
    - Already discussed it in Lecture-10 (80286 MP)
  - **Protected Mode**
    - Already discussed it in Lecture-10 (80286 MP) --- same as 80286.
    - Only difference is in descriptor description (to be discussed in coming lectures)
  - **Virtual 8086 Mode**
    - In the 80386, virtual 8086 mode (also called virtual real mode, V86-mode or VM86) **allows the execution of real mode applications that are incapable of running directly** in protected mode while the processor is running a protected mode operating system.
      - Memory Addressing in real mode
      - Interrupt in real mode

# Protected Mode

- Same as 80286

- Only difference is in
  - Descriptor's description
  - Optional use of ***page***

- If the paging unit is disabled, then linear address produced by segment unit is used as physical address
- Otherwise the paging unit converts the linear address into page address.
- The paging mechanism allows handling of large segments of memory in terms of pages of 4Kbyte size.

# Virtual 8086 Mode

- In its protected mode of operation, 386 provides a virtual 8086 operating environment to execute the 8086 programs.
- The real mode can also used to execute the 8086 programs along with the capabilities of 386, like protection and a few additional instructions.
- Once the 386 enters the protected mode from the real mode, it cannot return back to the real mode without a reset operation.

# Virtual 8086 Mode

- Thus, the virtual 8086 mode of operation of 386, offers an advantage of executing 8086 programs while in protected mode.
- The address forming mechanism in virtual 8086 mode is exactly identical with that of 8086 real mode.
- In virtual mode, 8086 can address 1Mbytes of physical memory that may be anywhere in the 4Gbytes address space of the protected mode of 386.

# Virtual 8086 Mode

- Like 386 real mode, the addresses in virtual 8086 mode lie within 1Mbytes of memory.
- In virtual mode, the paging mechanism and protection capabilities are available at the service of the programmers.
- The 386 supports multiprogramming, hence more than one programmer may be use the CPU at a time.

# Thank You !!

# Pentium Microprocessor

***Course Teacher:***

**Md. Obaidur Rahman, Ph.D.**
Professor
Department of Computer Science and Engineering (CSE)
Dhaka University of Engineering & Technology (DUET), Gazipur.

**Course ID:** CSE - 4503
**Course Title:** Microprocessors and Assembly Language
Department of Computer Science and Engineering (CSE),
Islamic University of Technology (IUT), Gazipur.

# Lecture References:

- Book:
  - *Microprocessors and Interfacing: Programming and Hardware*, Chapter # 15, **Author:** Douglas V. Hall
  - *The Intel Microprocessor 8086…Arch. Prog, Interfacing.* **Author:** Bary, Bray
  - *Microprocessor and Microcomputer – Based System Design*, **Author:** Mohamed Rafiquzzaman

- Lecture Materials:
  - *M. A. Sattar, BUET, Dhaka, Bangladesh.*

# Introduction of Pentium

- The concepts of 80386 microprocessor and 80387 coprocessor together evolved the 80486 microprocessor.
  - **Only new idea here is the introduction of 8K cache**
  - The cache is used for storing both *data* and *instructions*

- Pentium was introduced in 1993
- Pentium processor is an improvement to the architecture found in 80486.

# Pentium Improvements

- Improved cache structure
  - Reorganized to form **two level (L2) caches** that are each 8K bytes in size
    - One for caching *data*
    - Another for caching *instructions*
- Wider data bus width
  - Increased from 32 bit to 64 bits
- Faster numeric processor
  - Operates about 5 times faster than the 80486 numeric processor

# Pipelining in Pentium

- Pentium has two pipelines
- **U pipeline**
  - U-pipeline can execute ***any Pentium instruction***
- **V pipeline**
  - V-pipeline only executes only ***simple instructions***
- Each pipeline has 5 stages
  - i. Pre-fetch
  - ii. Instruction Decode
  - iii. Address Generation
  - iv. Execute, Cache, and ALU Access
  - v. Write back

# Pipelining Stages in Pentium

- **Pre-fetch:**
  - Instructions are fetched from the Instruction cache and aligned in prefetch buffers for decoding
- **Instruction Decode:**
  - Instructions are decoded into the Pentium's internal instruction format.
- **Address Generation:**
  - Address computations take place at this stage
- **Execute, Cache, and ALU Access:**
  - The integer hardware executes the instruction
- **Write-back:**
  - The results of the computation are written back to the register file

# Super Scalar Machine

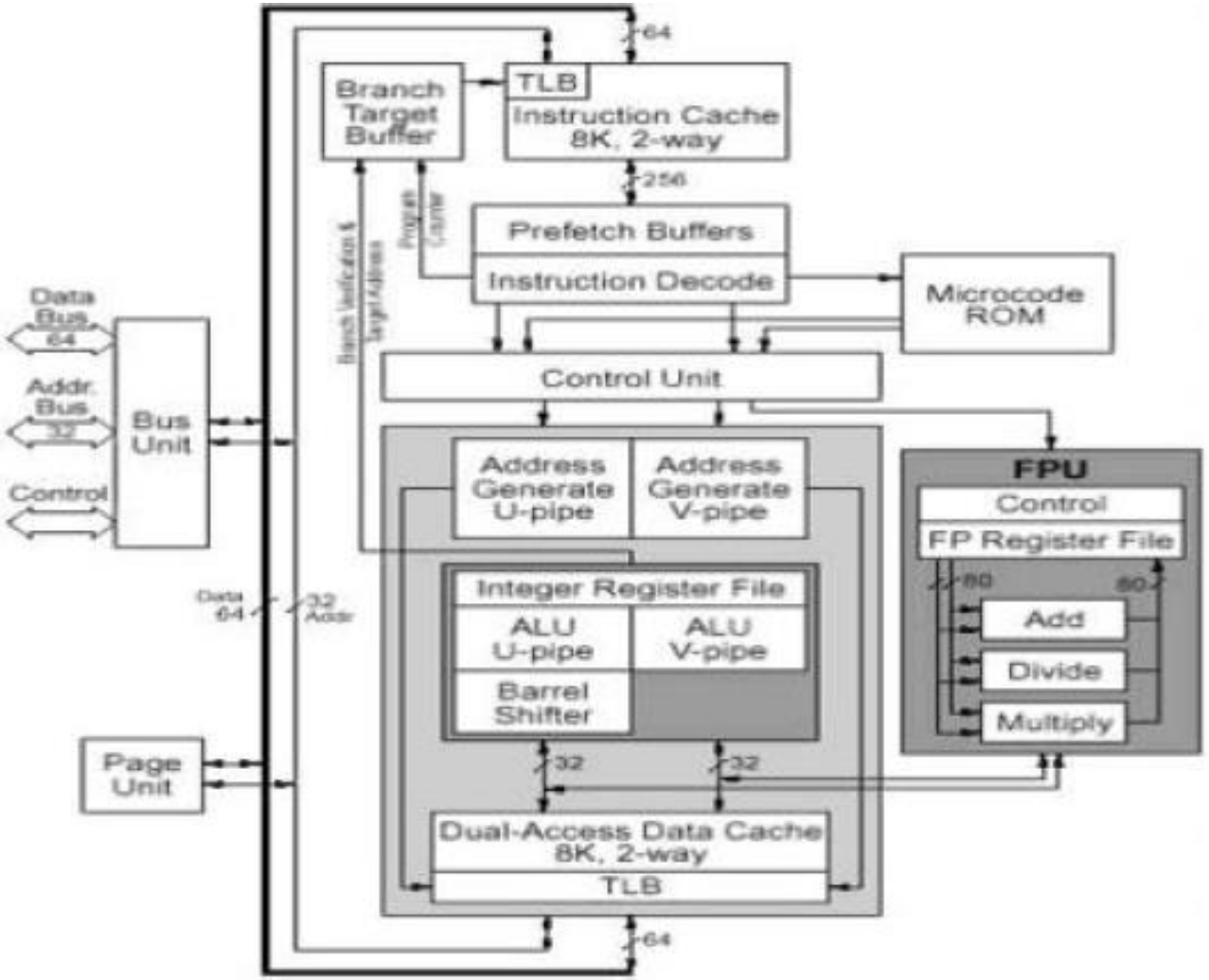
- Any Processor capable of parallel instruction execution of multiple instructions is known as **superscalar machine.**

- As, in Pentium, there are two execution lines --- U-line and V-line, so Pentium is a ***Super Scalar Processor***

- What are the differences? to understand the concepts in terms of processor execution stages:
  - Parallelelism
  - Simultaneity – Simultaneously Parallel
  - Pipelining – Parallel and Simultaneous

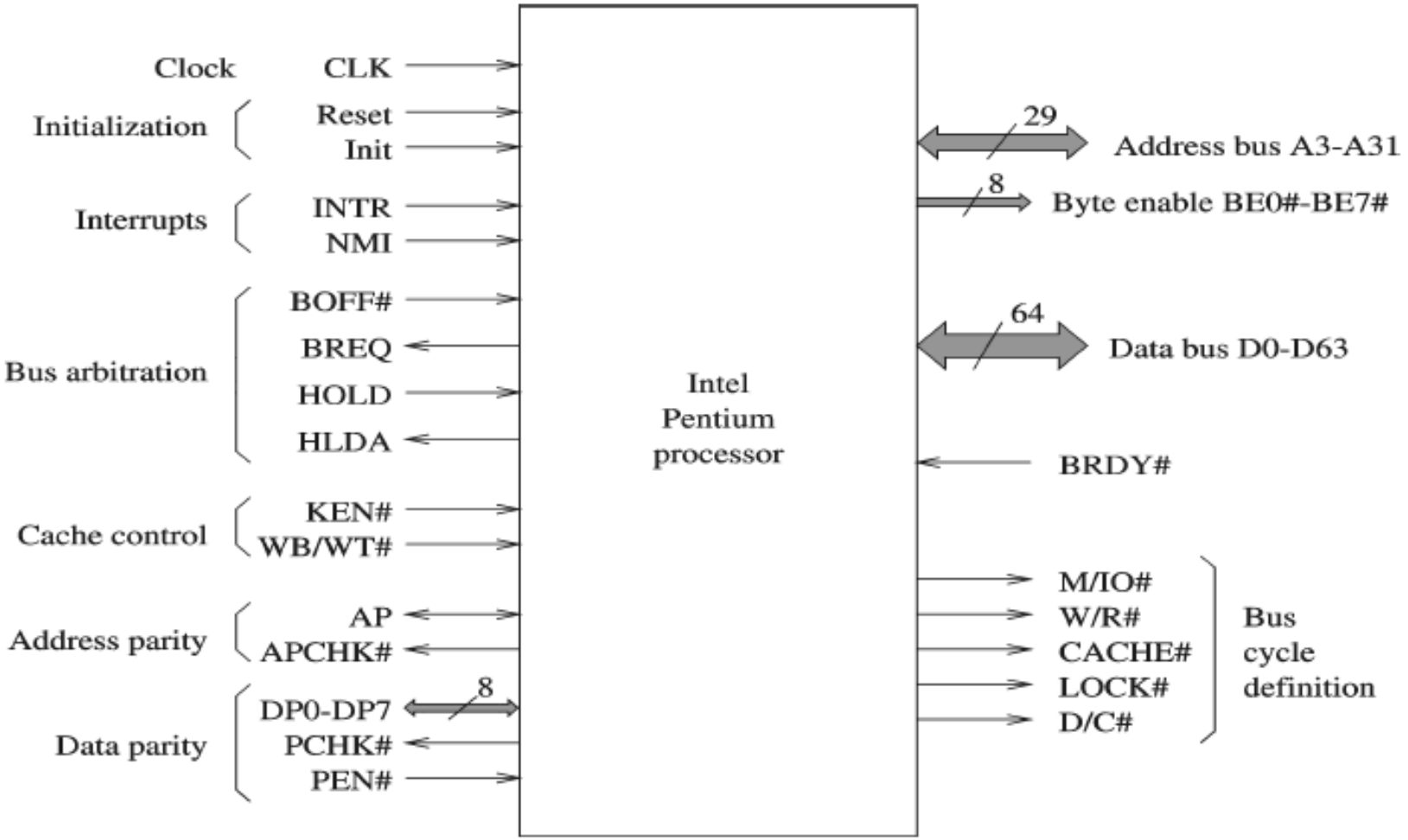
# Pentium Architecture

- It has all the units similar to 80386.
  - *Instruction unit*
  - *Segmentation unit*
  - *Paging unit*
  - *Bus unit*
  - *Execution unit*
- The newer one introduced as-
  - *Floating Point unit (FPU)*



# Pentium Architecture

- Pin Diagram



# Pentium Architecture: **Floating Point Unit (FPU)**

- FPU has 8 80-bit general purpose floating point registers, ST(0) through ST(7)
- **It has 8-stage pipeline**
  - First 5 stages are identical to U and V pipelines
  - 2 additional execution stages
    - First execution stage (X1 stage)
    - Second execution stage (X2 stage)
    - In these two stages FPU reads the data from data cache and executes the floating point computation
  - One additional error reporting stage

# Pentium Architecture: **Pins**

- Packaged in 273 pins
  - Data Bus 64 pins
  - Address Bus 29 pins plus 3 pins
  - Control Bus 75 pins
  - Vcc + Ground 99 pins
  - No Connection (NC) 6 pins

# Pentium Data Bus

- 64-bit data bus
- D0 through D63 (Data Bus)
- Bi-Directional
- These signals make up the Pentium's 64-bit bidirectional data bus.

# Pentium Address Bus

- 32-bit address bus.
- A3 through A31 (Address Lines)
  - Output/Input --- bi-directional
- These 29 address lines, together with the byte enable outputs *BE0-BE7, form the* Pentium's 32-bit address bus.
- A memory space of 4 GB is possible, along with 65536 I/O ports.
- The address lines are used as input during an inquire cycle to read an address *into the Pentium, for* examination by the internal cache.

# Pentium Address Bus

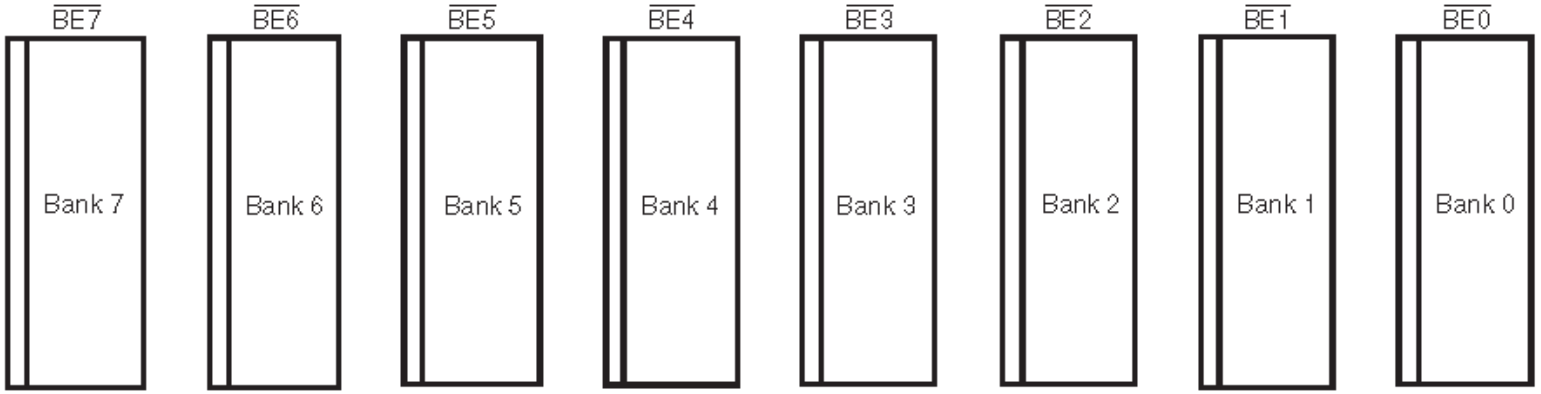
- No A0, A1 and A2 address lines are available in Pentium
  - They are internally decoded to produce BE0 to BE7 signals

- **BE0 through BE7**
  - Byte (Bank???) Enable lines
  - Memory are arranged in total 8 banks

- **Output**
  - These, together with A3 through A31, make up the 32-bit address output by the Pentium.
  - Each byte enable is used to control a different 8-bit portion of the processor's 64-bit data bus.
  - BE0 enables Bank 0

# Pentium Memory System

- Pentium's memory arranged in 8 banks
- Each bank stores 1 byte of data ***with parity bit***
  - **It helps for error detection and correction in data**
- BE0 to BE7 selects the banks
- New feature added to Pentium is its capability to generate and check parity for address bus



# Features of All Pentium Processors

# **Pentium Pro:** Salient Features

- The notable difference in the Pentium Pro than the earlier Pentium is that
  - **There are provisions for a 36-bit address bus, which allows access to 64G bytes of memory.**
- ***This is meant for future use*** because no system today contains anywhere near that amount of memory.
- Pentium Pro is available in two versions.
  - One version contains a 256K level 2 cache;
  - The other contains a 512K level 2 cache
- Pentium Pro microprocessor is packaged in an immense 387-pin PGA (pin grid array).

# **Pentium II:** Salient Features

- Extension to Pro architecture with some differences
  - Internal cache in PII has been moved out of the chip
  - PII is not available as a single chip
  - Rather is available on a small plug-in circuit board, known as ***Cartridge, along with level 2*** (L2) cache chip

- Various versions are available
  - **Celeron** is a version without L2 cache
  - **Xeon** is enhanced by having up to 2M L2 cache

# **Pentium III:** Salient Features

- Based on Pro architecture, not on Pentium II
- Like PII, PIII is packaged in a cartridge instead of IC chip
- Additionally a **Coppermine** is packaged in an IC with 370 pins
- **Coppermine** is an internal cache with 256K advanced transfer mechanism within the IC running at processor speed
- ***Why not used 512K Cache?***
  - ***It has been observed that, increasing cache size from 256K to 512K improves the performance by only a few percent***

# **Pentium III:** Salient Features

- Various versions of Pentium III are also available like Pentium II
  - **Standard** Pentium III
  - **Celeron** Pentium III uses 66MHz bus speed
  - **Xeon** Pentium III allows larger cache for server applications
    - Still Xeon is popular for server processors

# **Pentium IV:** Salient Features

- Based on Pro architecture, not on P II or P III
- P IV is packaged in 421 pins IC
- It uses physically smaller transistors
  - Makes it much smaller and faster than P III
- Released initially in November 2000 with 1.3GHz speed
  - Now available with speed more than 3GHz

# Thank You !!

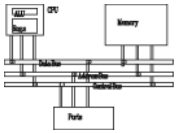
# x86 Processor Memory Management

***Course Teacher:***

**Md. Obaidur Rahman, Ph.D.**
**Professor**
**Department of Computer Science and Engineering (CSE)**
**Dhaka University of Engineering & Technology (DUET), Gazipur.**

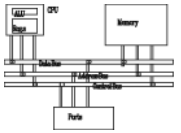
**Course ID:** CSE - 4503
**Course Title:** Microprocessors and Assembly Language
Department of Computer Science and Engineering (CSE),
Islamic University of Technology (IUT), Gazipur.

# Lecturer Reference



- Lecture Material:
  - Intel x86 Architecture: *Computer Organization and Assembly Languages, Yung-Yu Chuang*
  - *with slides by Kip Irvine*

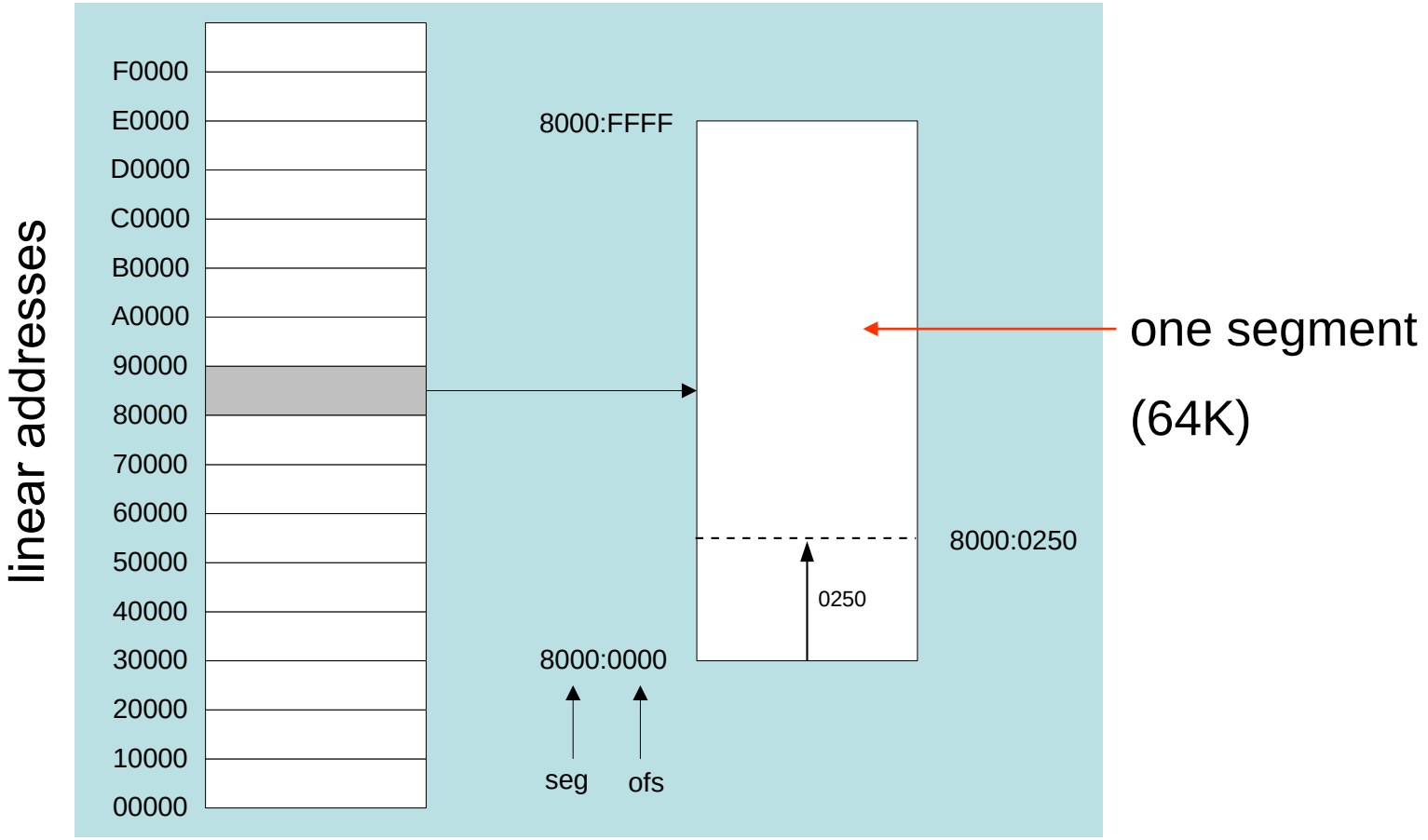
# Real-address mode



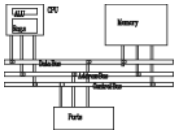
- 1 MB RAM maximum addressable (20-bit address)
- Application programs can access any area of memory
- Single tasking
- Supported by MS-DOS operating system

# Real-address Mode: Segmented memory

Segmented memory addressing: absolute (linear) address is a combination of a 16-bit segment value added to a 16-bit offset



## Real-address Mode: Calculating linear addresses



- Given a segment address, multiply it by 16 (add a hexadecimal zero), and add it to the offset
- Example: convert 08F1:0100 to a linear address

| | | | | | |
|---|---|---|---|---|---|
| **Adjusted Segment value:** | **0** | **8** | **F** | **1** | **0** |
| **Add the offset:** | | **0** | **1** | **0** | **0** |
| **Linear address:** | **0** | **9** | **0** | **1** | **0** |

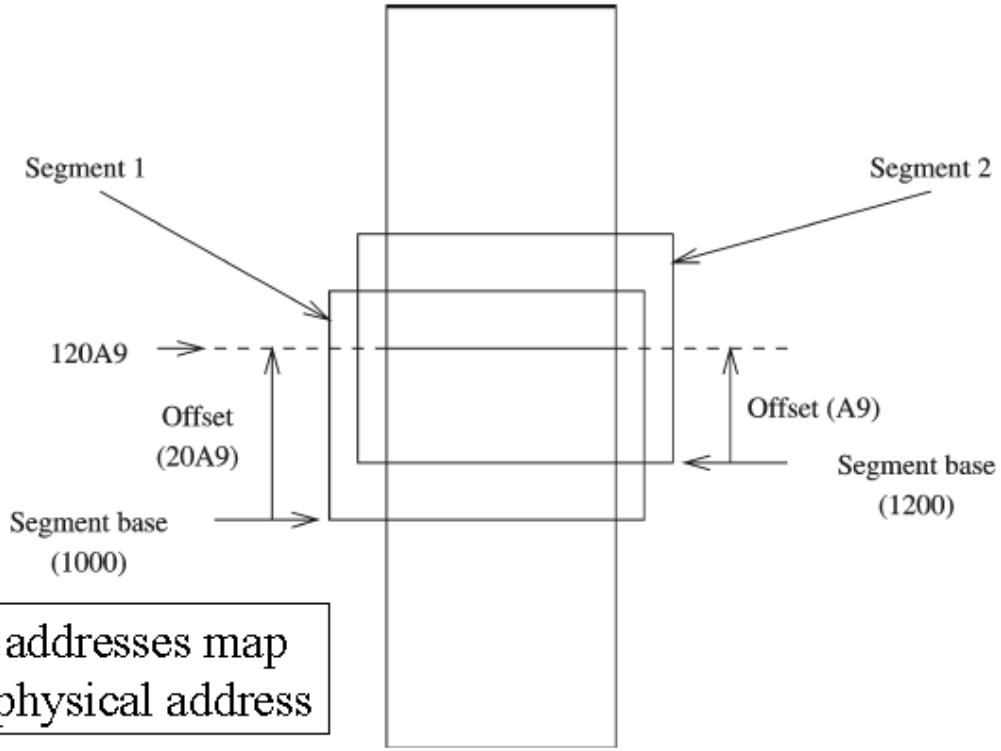
- A typical program has three segments: code, data and stack. Segment registers CS, DS and SS are used to store them separately.

# Real-address Mode: Example

What linear address corresponds to the segment/offset address 028F:0030?

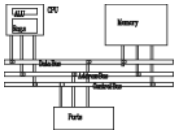
028F0 + 0030 = 02920

Always use hexadecimal notation for addresses.

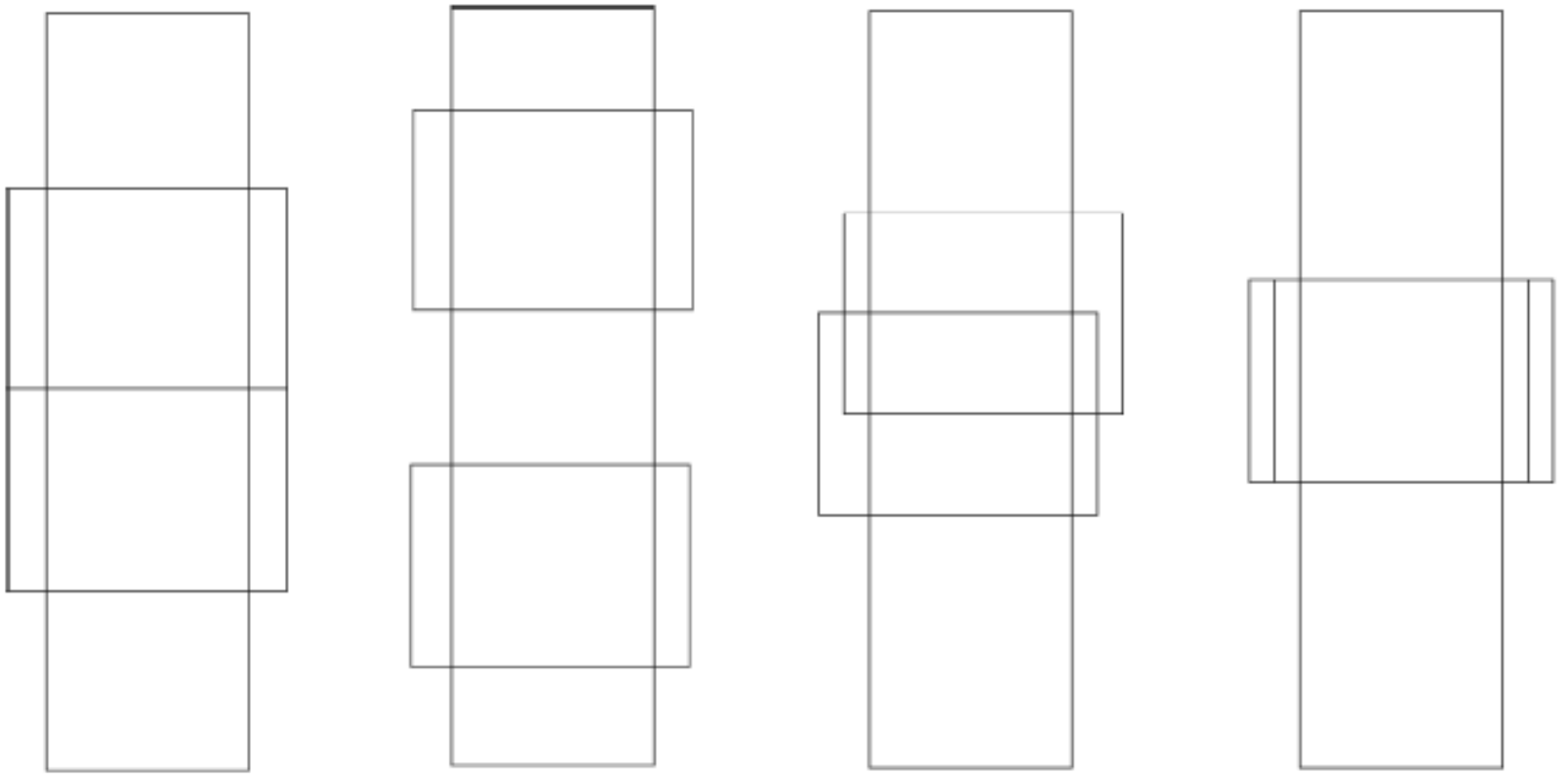



Two logical addresses map to the same physical address

# Real-address Mode: Example

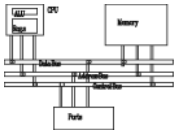


- Segment Overlapping



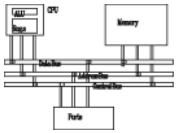
(a) Adjacent (b) Disjoint (c) Partially overlapped (d) Fully overlapped

# Protected mode



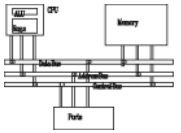
- 4 GB addressable RAM (32-bit address)
  - (00000000 to FFFFFFFFh)
- Each program assigned a memory partition which is protected from other programs
- Designed for multitasking
- Supported by Linux & MS-Windows

# Protected mode

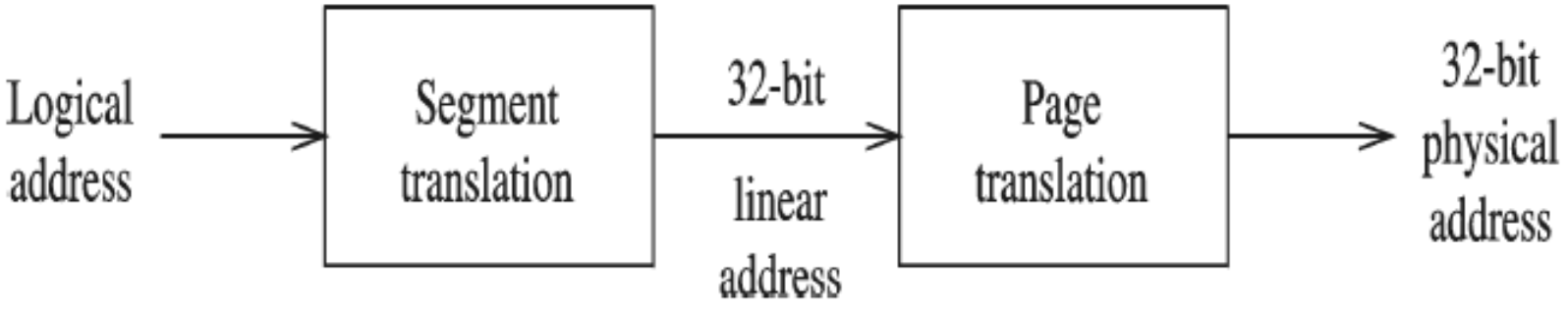


- Started with 80286 up to Pentium, all processors use 2 modes for memory address management
  - Real mode
    - Uses 16-bit addresses
    - Runs 8086 programs
    - Pentium acts as a faster 8086
  - Protected mode
    - 32-bit mode
    - Native mode of Pentium
    - Supports segmentation and paging

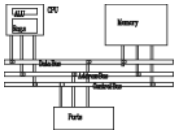
# Protected mode



- Supports sophisticated segmentation
- Segment unit translates 32-bit logical address to 32-bit linear address
- Paging unit translates 32-bit linear address to 32-bit physical address
  - If no paging is used
    - **Linear address = physical address**

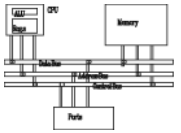


# Protected mode

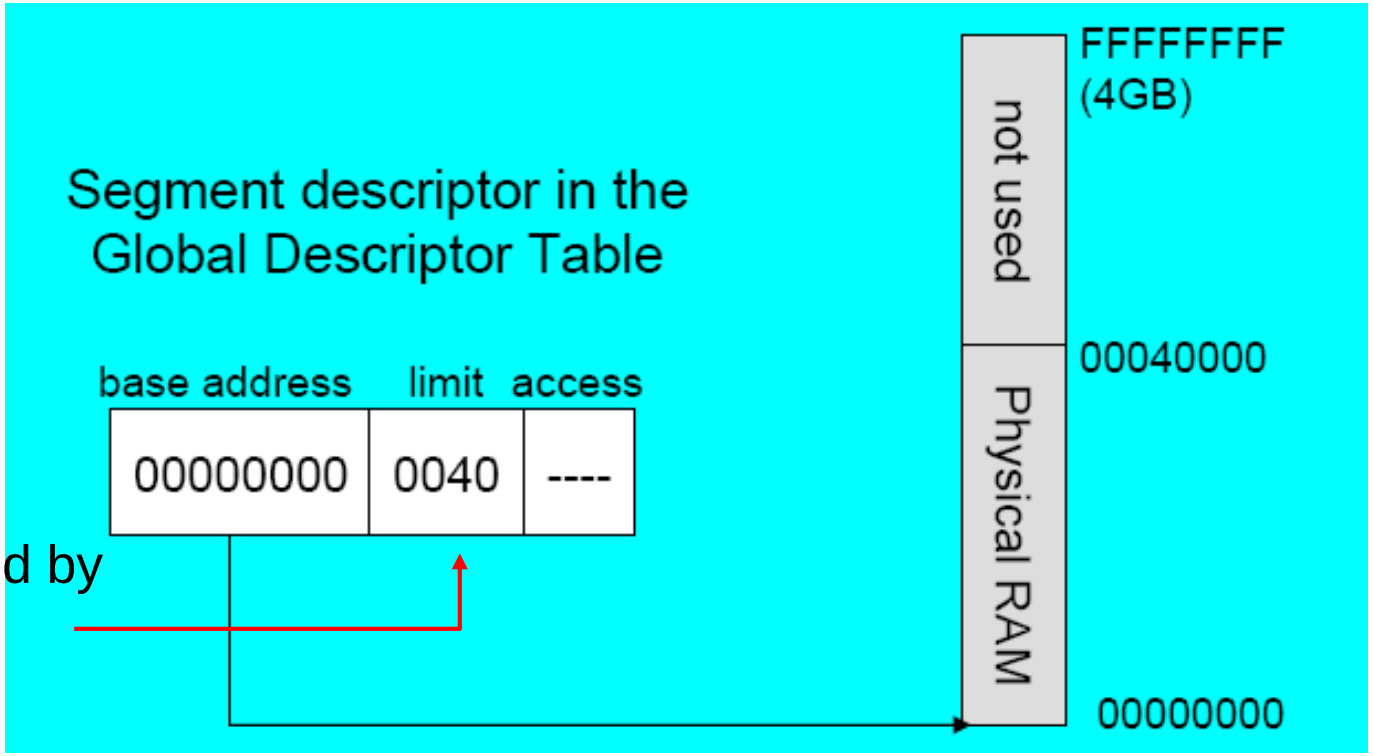


- In this mode there is a Segment Descriptor Table
- Typical Program structure follows:
  - Code, Data, and Stack areas
  - CS, DS, SS segment descriptors
  - Global Descriptor Table (GDT)
  - Local Descriptor Table (LDT)
- MASM Programs use the Microsoft flat memory model

# Flat segmentation model

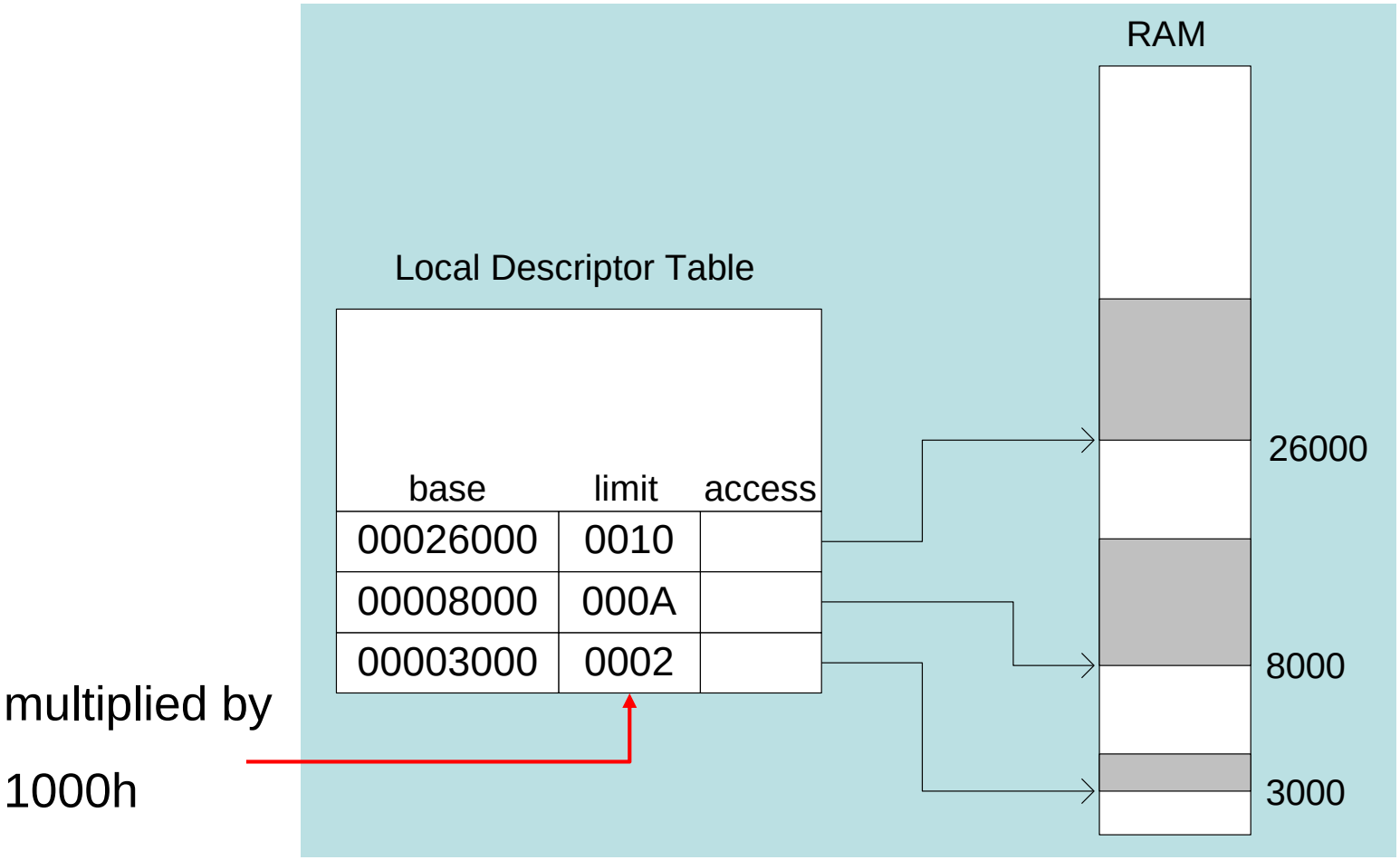


- All segments are mapped to the entire 32-bit physical address space, at least two, one for data and one for code
- Global Descriptor Table (GDT)

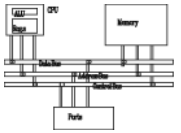


# Multi-segment model

- Each program has a local descriptor table (LDT)
  - holds descriptor for each segment used by the program

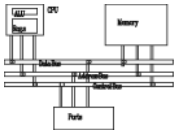


# Translating Addresses

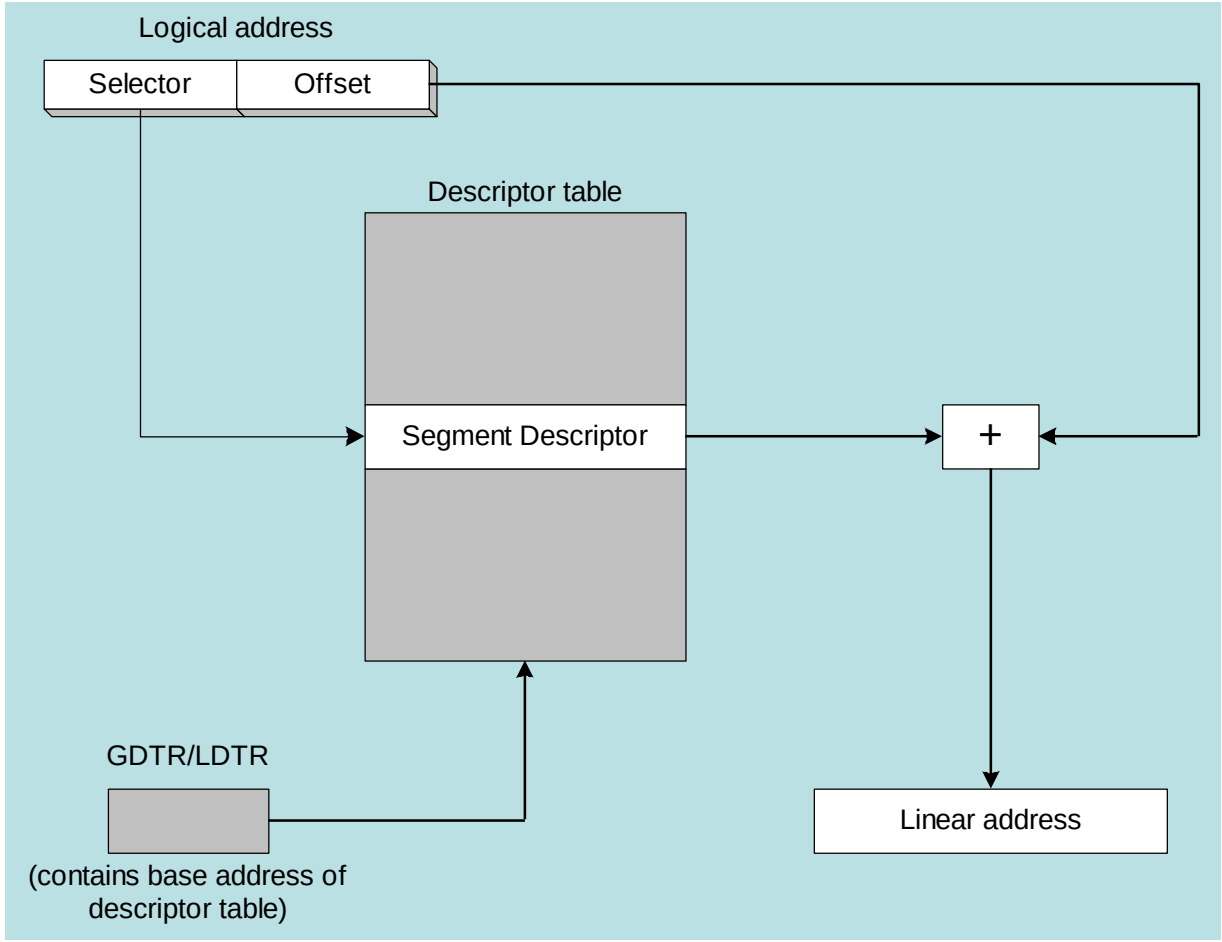


- The processor uses a one- or two-step process to convert a variable's logical address into a unique memory location.
- The first step combines a segment value with a variable’s offset to create a linear address.
- The second optional step, called page translation, converts a linear address to a physical address.

# Converting Logical to Linear Address

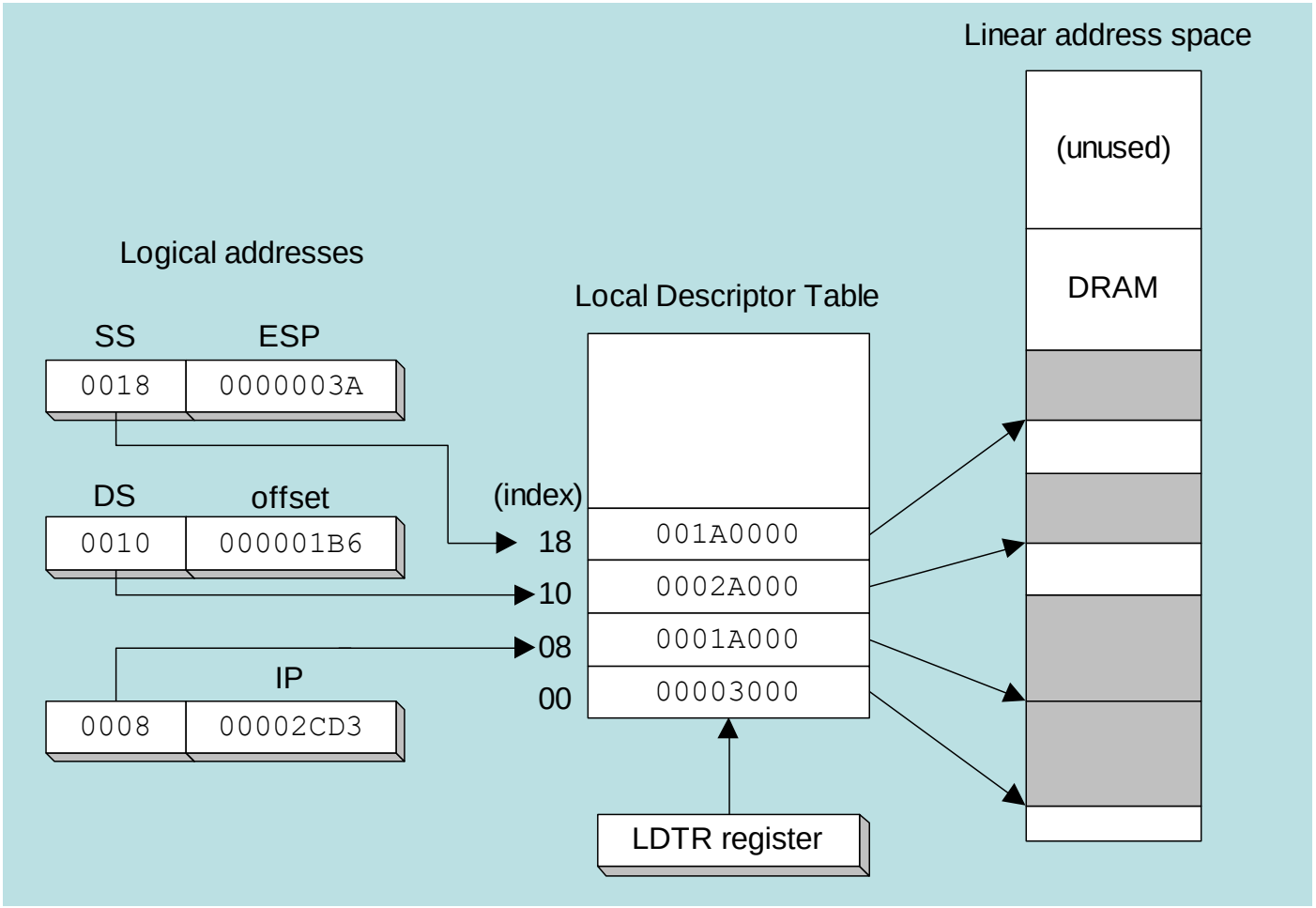


The segment selector points to a segment descriptor, which contains the base address of a memory segment. The 32-bit offset from the logical address is added to the segment's base address, generating a 32-bit linear address.

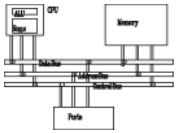


# Indexing into a Descriptor Table

Each segment descriptor indexes into the program's local descriptor table (LDT). Each table entry is mapped to a linear address:

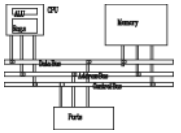


# Paging



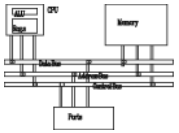
- Virtual memory uses disk as part of the memory, thus allowing sum of all programs can be larger than physical memory
- Only part of a program must be kept in memory, while the remaining parts are kept on disk.
- The memory used by the program is divided into small units called pages (4096-byte).
- As the program runs, the processor selectively unloads inactive pages from memory and loads other pages that are immediately required.

# Paging

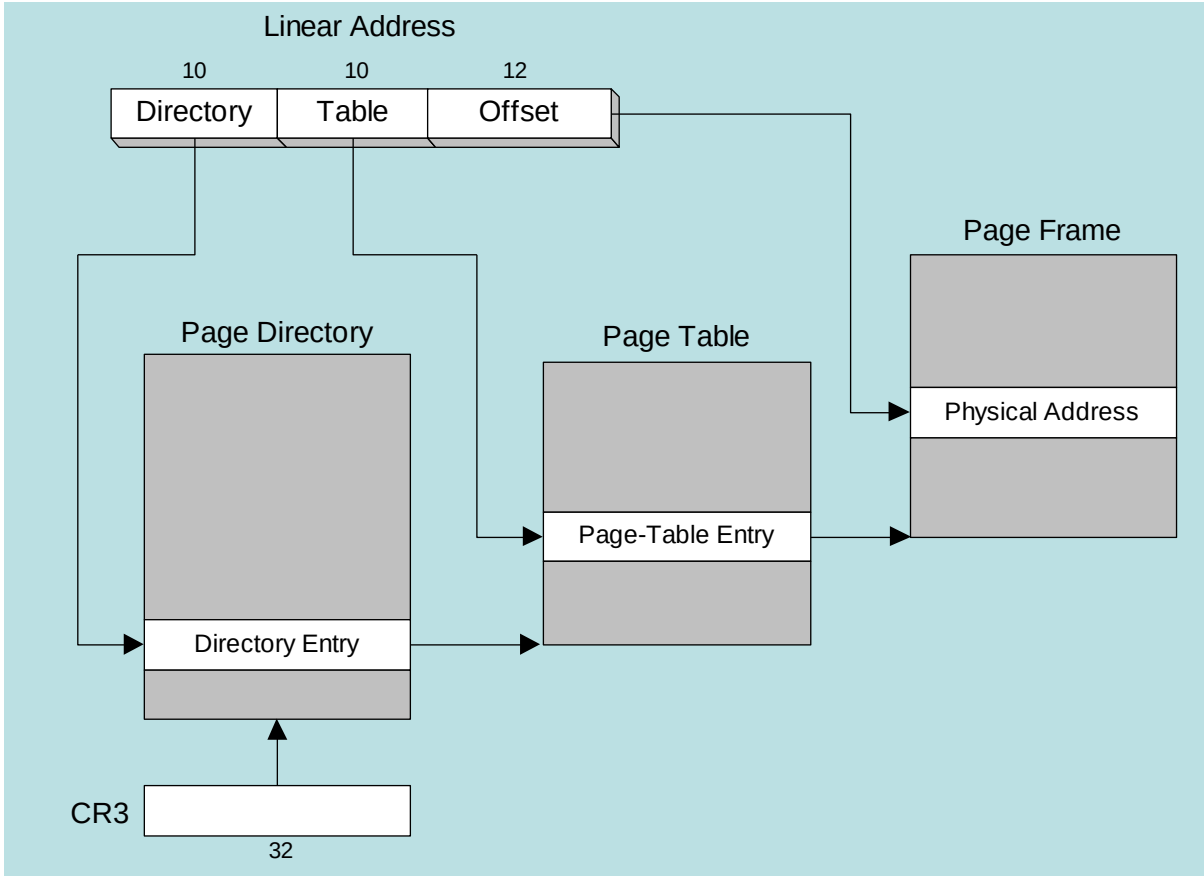


- OS maintains **page directory** and **page tables**
- **Page Translation:** CPU converts the linear address into a physical address
- **Page Fault:** Occurs when a needed page is not in memory, and the CPU interrupts the program
- **Virtual Memory Manager (VMM):** OS utility that manages the loading and unloading of pages
- OS copies the page into memory, program resumes execution

# Page Translation



A linear address is divided into a page directory field, page table field, and page frame offset. The CPU uses all three to calculate the physical address.



# Math Co-processor
# and
# Multi-core Processor

***Course Teacher:***

**Md. Obaidur Rahman, Ph.D.**
Professor
Department of Computer Science and Engineering (CSE)
Dhaka University of Engineering & Technology (DUET), Gazipur.

**Course ID:** CSE - 4503
**Course Title:** Microprocessors and Assembly Language
Department of Computer Science and Engineering (CSE),
Islamic University of Technology (IUT), Gazipur.

# Lecture References:

- Book:
  - *Microprocessors and Interfacing: Programming and Hardware*, **Author:** Douglas V. Hall
  - *An Introduction to Parallel Programming*, **Author:** Peter Pacheco

- Lecture Materials:
  - *M. A. Sattar, BUET, Dhaka, Bangladesh.*

# Math Co-processor

- Intel family of math coprocessor are generally labelled as 80x87, which support 80x86 processors in computing.
- Instruction sets and programming of all devices are almost identical
- Intel family of math-coprocessors is able to
  - ***add***
  - ***subtract***
  - ***multiply***
  - ***divide***
  - ***find square root***
  - ***calculate partial tangent***
  - ***calculate partial arctangent***
  - ***logarithms***

# Intel Processors and Co-processors

| Processor | Corresponding Co-processor |
|---|---|
| 8086/8088 | 8087 |
| 80186/80188 | 80187 |
| 80286 | 80287 |
| 80386 | 80387SX, 80387DX |
| 80486SX | 80487SX |

# CISC and RISC Processors

- **CISC (Complex Instruction Set Computers):**
  - CISC was developed to make compiler development simpler.
  - It shifts most of the burden of generating machine instructions to the processor.
    - ***For example**, instead of having to make a compiler write long machine instructions to calculate a square-root, a CISC processor would have a built-in ability to do this.*

# CISC and RISC Processors

- **RISC (Reduced Instruction Set Computers):**
  - RISC is a type of microprocessor architecture that utilizes a small, highly-optimized set of instructions, rather than a more specialized set of instructions often found in other types of architectures.
  - **History**
    The first RISC projects came from IBM, Stanford, and UC-Berkeley in the late 70s and early 80s. The IBM 801, Stanford MIPS, and Berkeley RISC 1 and 2 were all designed with a similar philosophy which has become known as RISC.

# CISC and RISC Processors

The main characteristics of **CISC** microprocessors are:

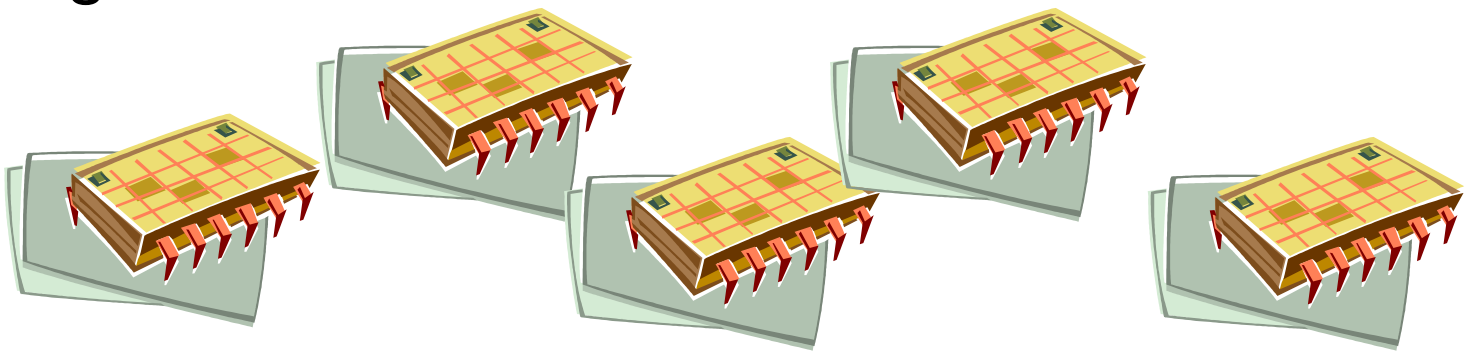
- Extensive instructions.
- Complex and efficient machine instructions.
- Extensive addressing capabilities for memory operations.
- Relatively few registers in use.

In comparison, **RISC** processors are more or less the opposite of the above:

- Reduced instruction set.
- Less complex, simple instructions.
- *Pipelining* is used to allow for simultaneous execution of parts, or stages, of instructions
- Many symmetric registers are used.

# Age of Multi-core Processors

- From 1986 – 2002, microprocessors were speeding like a rocket, increasing in performance an average of 50% per year.
- Since then, it's dropped to about 20% increase per year.
- **An Intelligent Solution:**
  - Instead of designing and building faster microprocessors, put multiple processors on a single integrated circuit.



# Is Multi-core Processors is the Solution?

- Up to now, performance increases have been attributable to increasing density of transistors in multiple microprocessors.

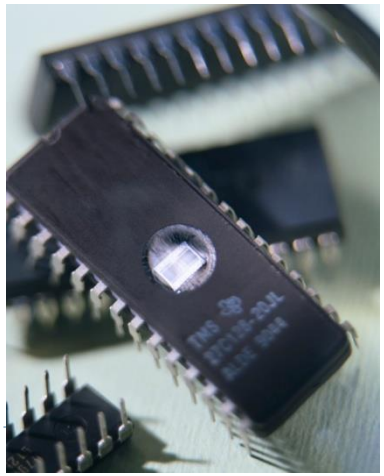
- But there are inherent problems:
  - *Smaller transistors = faster processors*
  - *Faster processors = increased power consumption*
  - *Increased power consumption = increased heat*
  - *Increased heat = unreliable processors*

# Multi-core Processor and Parallel Programming

- Adding more processors doesn't help much if programmers aren't aware of them…
- … or don't know how to use them.

- Serial programs don't benefit from this approach (in most cases).
- Move away from single-core systems **to multi-core processors** and **Introduce parallelism**!!!

- "Core" = central processing unit (CPU)



# Parallel Programming Concept for Multi-core Processors

- Compute n values and add them together.
- **Serial solution:**

```
sum = 0;
for (i = 0; i < n; i++) {
    x = Compute_next_value(. . .);
    sum += x;
}
```

- n iteration requires to add.

# Parallel Programming Concept for Multi-core Processors

- 8 cores, n = 24, then the calls to
- 1,4,3, 9,2,8, 5,1,1, 5,3,7, 2,5,0, 4,1,8, 6,5,1, 2,3,9
- The Master Core runs the serial program

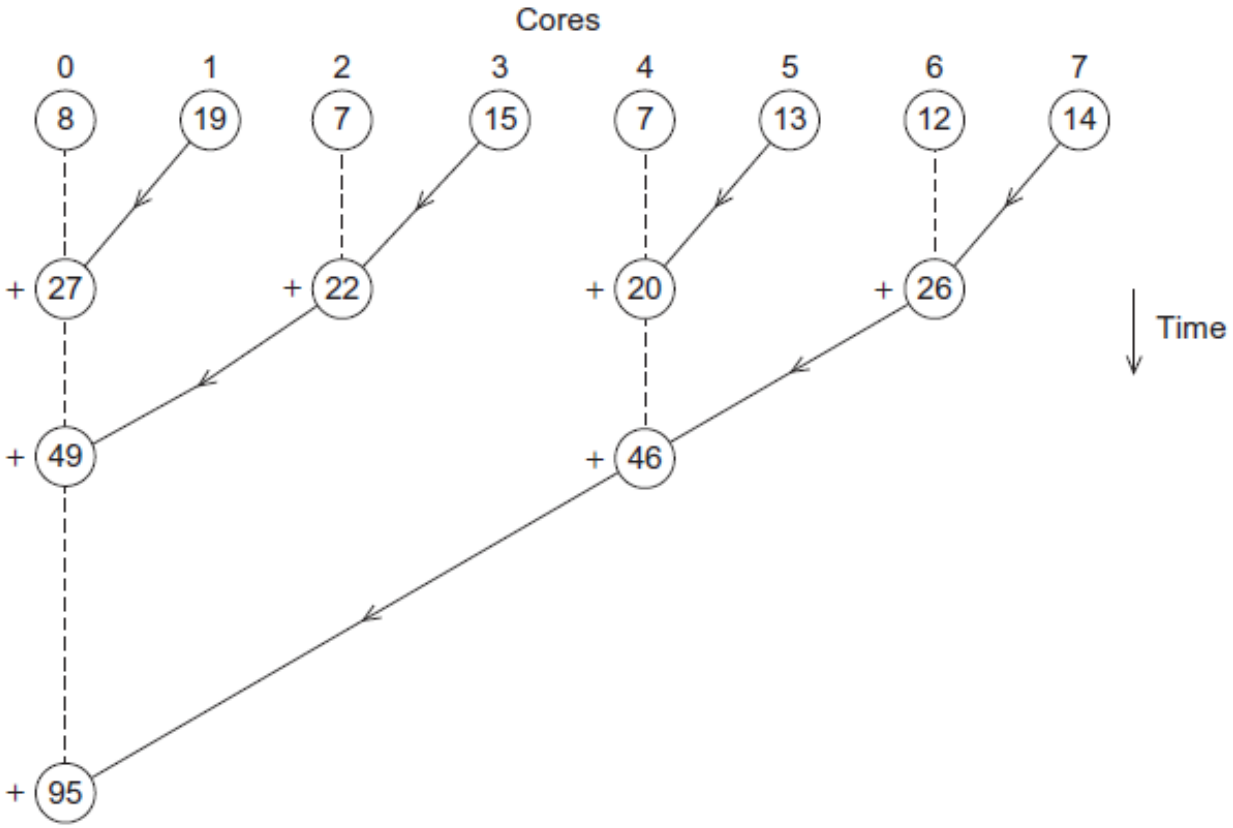
```
if (I'm the master core) {
   sum = my_x;
   for each core other than myself {
      receive value from core;
      sum += value;
   }
} else {
   send my_x to the master;
}
```

| Core | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| my_sum | 8 | 19 | 7 | 15 | 7 | 13 | 12 | 14 |

| Core | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| my_sum | **95** | 19 | 7 | 15 | 7 | 13 | 12 | 14 |

# Parallel Programming Concept for Multi-core Processors

But wait! There's a much better way to compute the global sum.



# Parallel Programming Concept for Multi-core Processors

- **Analysis**
  - In the first example, the master core performs 7 receives and 7 additions.
  - In the second example, the master core performs 3 receives and 3 additions.
  - The improvement is more than a factor of 2!
  - **Imagine, If we have 1000 cores:**
    - The first example would require the master to perform 999 receives and 999 additions.
    - The second example would only require 10 receives and 10 additions.
  - That's an improvement of almost a factor of 100!

# Salient Features: Dual Core and Core 2 Duo

- **Architecture**

- *Dual Core:* The dual core is the older of the two having an architecture that sports two cores on one processor, but its older technology puts it in a position of disadvantage.

- *Core 2 Duo:* The core 2 duo has two cores on the same processor. However while the architecture sounds vaguely the same, it is more advanced than the dual core processors.

# Salient Features: Dual Core and Core 2 Duo

- **Performance**
- *Dual Core:* This is without a doubt one of the best Intel processors, however it lacks the bite in terms of performance. Acceptable enough clock speeds of about 2.33 GHz for the best ones.

- *Core 2 Duo:* This newer processor beats the dual core in all bench-marking tests and therefore performance-wise, is definitely the better pick. Slams the opposition clocking speeds of 3.33 GHz clock as seen of the higher end ones.

# Salient Features: Core i3

- Entry level processor
- 2-4 Cores
- 4 Threads

(a **thread** of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system. Multiple threads can exist within one process, executing concurrently and sharing resources such as memory, while different processes do not share these resources.)

- Hyper-Threading (efficient use of processor resources)
- 3-4 MB Cache
- 32 nm Silicon (less heat and energy)
- Core i3 processors do support 64-bit versions of Windows
- **Suitability:**
  - If you use your computer for basic tasks such as word processing, email, surfing the web, etc., a Core i3 processor is more than enough to handle all of that with ease

# Salient Features: Core i5

- Mid range processor
- 2-4 Cores
- 4 Threads
- Turbo Mode (turn off core if not used)
- Hyper-Threading (efficient use of processor resources)
- 3-8 MB Cache
- 32-45 nm Silicon (less heat and energy)
- Suitability:
  - Core i5's offer enough performance to do stuff like video editing and gaming, and more than enough performance to do basic stuff like-word processing, internet surfing, and email.

# Salient Features: Core i7

- High end processor
- 4-8 Cores
- 8 Threads
- Turbo Mode (turn off core if not used)
- Hyper-Threading (efficient use of processor resources)
- 4-8 MB Cache
- 32-45 nm Silicon (less heat and energy)

- Suitability:
  - Core i7's offer enough performance to do stuff like video editing and gaming.

# Salient Features: Core i9
## (10th and upper Generation Processor)

- 8-16 cores
- Supports Hyper-Threading
- Upto 16 MB of Smart Cache
- 1400 Billions Transistor
- Turbo-mode compatible

- Suitability:
  - Core i9's offer enough performance to do stuff like video editing and gaming.

# Thank You !!

**Course Title:** Microprocessors and Assembly Language Lab (CSE-4504)

Department of Computer Science and Engineering (CSE)
**Islamic University of Technology (IUT), Gazipur**

**Lab # 01**
*Program Structure and Arithmetic Operations using Assembly Language Program in EMU8086.*

**Objective:**
Getting familiar with Program Structure and Arithmetic operations using Assembly Language Program in EMU8086.

**Installation of EMU8086 and Run for the First Time:**

**Step 1:** Run the setup.exe file to install the program.
**Step 2:** Launch the EMU8086 emulator. Choose "New" and specify "empty workspace" template.
**Step 3:** Using the assembler editor, get familiar with the example codes.
**Step 4:** Start emulation by clicking the "emulate" button on the toolbar. A new emulator window will appear.
**Step 5:** Debug the program codes by pressing the "single step" button on the toolbar of the emulator window.
**Step 6:** Each time after pressing the "single step" button, check and record down the contents of registers like AX, BX, CX, DX etc.

**Theory:**

- **Data Transfer Instructions:**

Format: *MOV* ***Destination***, ***Source***

**Registers (Direct):** Move contents of one register to another register

```
MOV AL, BL
MOV AX, BX
```

**Immediate:** Load a register with an immediate value or equivalent binary/hexa-decimal

```
MOV CL, 240
MOV CL, 11110000B
MOV CL, 00F0H

MOV CX, 256
MOV CX, 0000000100000000B
MOV CX, 0100H
```

**Direct:** Move contents of the variable named COUNT to a register

```
MOV DL, COUNT ; here COUNT is a 8-bit variable
MOV DX, COUNT ; here COUNT is a 16-bit variable
```

- **Arithmetic / Logic Instructions:**

Arithmetic and logic instructions can be performed on 8-bit (byte) and 16-bit values.

**Increment** the contents of a register by a value (decimal/binary/hexa-decimal)

```
ADD AX, 4
```

**Add** the contents of a register with the contents of another register

```
ADD AX, BX
```

**Subtract** a value (decimal/binary/hexa-decimal) from the contents of a register

```
SUB DL, 4
```

**Subtract** the contents of a register from the contents of another register

```
SUB DX, CX
```

**Multiply** AX by BL, the result will be in AX

```
MUL BL
```

**Divide** the contents of AX register with the value of CL and store the result in AX

```
DIV CL
```

**Increase** or Decrease the contents of BX register by 1

```
INC BX      ; Increase          DEC BX    ; Decrease
```

**Clear** the contents of AX register

```
XOR AX, AX
```

**Negation and NOT** of a register value

```
NEG AL;     2's Complement
NOT AL;     1's Complement
```

**Example for Assembly Language Program Structure:**

```
ORG 0100h     ; Offset of the program in memory

.DATA         ; Data Segment Starts
A DB 11       ; Variable A got a BYTE value 11
B DW 500      ; Variable B got a WORD value 500
SUM DW ?          ; Variable SUM is defined as a WORD variable without any value
DIFFERENCE DB ?   ; Variable DIFFERENCE is defined as a WORD variable without any value
MULTIPLICATION DW ?
DIVISION DB ?

.CODE         ; Code Segment Starts

MAIN PROC     ; Initialize Data Segment Register

   MOV AL, 30       ; Move decimal 30 to AL register
   ADD AL, 15       ; Add decimal 15 to the content of AL and store the result in AL

   MAIN ENDP        ; End Procedure
END MAIN            ; End MAIN
RET                 ; Return to DOS
```

**Tasks to do:**

1. Write appropriate assembly language codes to accomplish the following tasks (use as many as possible arithmetic instructions with less number of registers):
   - (30 + 15) * (575 – 225) + 210
   - 0Bh * (200 - 225) + 127
   - 0FFFh * 10h + 1111b
   - Convert 260$^{o}$ C (Celsius) to F (Fahrenheit) using the following expression and store in a variable F:
     °F = °C x 10/5 + 32 - 1

**Course Title:** Microprocessors and Assembly Language Lab (CSE-4504)

Department of Computer Science and Engineering (CSE)
**Islamic University of Technology (IUT), Gazipur**

**Lab # 02**
*8086* ***I/O Instructions*** *and* ***Condition Control Instructions*** *using Assembly Language.*

**Objective:**
To understand the 8086 I/O instructions using Assembly Language Program in EMU8086.

**Theory:**

- **The INT Instruction**

To invoke a DOS, the **INT** (interrupt) instruction is used. It has the format

INT interrupt_number

Where, **interrupt_number** is a number that specifies a routine. In the following, we use a particular DOS routine, INT 21h.

**INT 21h**
INT 21h may be used to invoke a large number of DOS functions; a particular function is requested by placing a function number in the AH register and invoking INT 21h. Here we are interested in the following functions:

| Function Number | Routine |
|---|---|
| 1 | single-key input |
| 2 | single-key output |
| 9 | character string output |

INT 21h functions expect input values to be in certain registers and return output values in other registers. These are listed as we describe each function.

**Function 1:**
**Single-key Input**
Input: AH=1
Output: AL = ASCII code if character key is pressed
= 0 if non-character key is pressed.
To invoke the routine, execute these instructions:

```
MOV AH, 1   ; input key function
INT 21h     ; ASCII code in AL
```

**Function 2:**
**Single-key Output**
Input: AH=2
DL = ASCII code of the display character or control character
Output: AL = ASCII code of the display character or control character

To display a character with this function, we put its ASCII code in DL. For example, the following instructions cause a question mark to appear on the screen:

```
MOV AH, 2   ; display character function
MOV DL, '?' ; character is '?'
INT 21h     ; display character
```

After the character is displayed, the cursor advances to the next position on the line. Function 2 may also be used to perform control functions. If DL contains the ASCII codes of a control character, INT 21h causes the control function to be performed. The principle control characters are as follows:

| **ASCII code** | **Symbol** | **Fucntion** |
|---|---|---|
| A | LF | line feed (new line) |
| D | CR | carriage return (start of a line) |

- **Conditional Control Transfer Instruction**

Conditional jumps transfer control to another address depending on the values of the flags in the flag register. The jump condition often provided by the CMP instruction:

**CMP *destination, source***

| Condition | Instruction | Condition | Instruction |
|---|---|---|---|
| Jump if zero flag ZF=1 | **JZ *zero*** | Jump if zero flag ZF=0 | **JNZ *notzero*** |
| Jump if greater | **JG *greater*** | Jump if greater than or equal | **JGE *notless*** |
| Jump if less | **JL *less*** | Jump if less than or equal | **JLE *notgreater*** |
| Jump if Below | **JB *smaller*** | Jump if carry flag CF=1 | **JC *carry*** |

**Assembly Language Program Example:**

```
ORG 0100h
MAIN PROC
; display prompt
MOV AH, 2
MOV DL, '?'
INT 21h
; input a character
MOV AH, 1
INT 21h
MOV BL, AL
; go to a new line with carriage return
MOV AH, 2
MOV DL, 0DH
INT 21h
MOV DL, 0AH
INT 21h
; display character
MOV DL, BL
INT 21h
; return to DOS
MOV AH, 4CH
INT 21H

MAIN ENDP
END MAIN
RET
```

```
org 100h

START:      mov cl, 03h

LABEL_JNZ:  dec cl
            jnz LABEL_JNZ
            mov bl, 04h
            mov al, 04h

LABEL_JZ:   dec al
            dec bl
            xor bl, al
            jz LABEL_JZ
            mov bl, 02h
            mov al, 06h

LABEL_JG:       dec al
            cmp al, bl
            jg LABEL_JG
            mov bl, 06h
            mov al, 00h

LABEL_JL:   inc al
            cmp al, bl
            jl LABEL_JL

ret
```

**Tasks to do:**

1. Write an assembly language program that inputs a single letter and shows the same letter in it's opposite case in a new line. (Lower-case to Upper-case or vice-versa).
   **Sample Input / Output:**

| | | | |
|---|---|---|---|
| Input: | a | Input: | Z |
| Output: | A | Output: | z |

2. Write an assembly language program that inputs a single letter and shows the next 5 (five) letters in opposite case of input (Lower-case to Upper-case or vice-versa) in a row of a new line and also shows the previous 5 (five) letters in the next line in opposite case of input (Lower-case to Upper-case or vice-versa).
   **Sample Input / Output:**

| | | | |
|---|---|---|---|
| Input: | a | Input: | Z |
| Output: | BCDEF | Output: | abcde |
| | ZYXWV | | yxwvu |

**Course Title:** Microprocessors and Assembly Language Lab (CSE-4504)
Department of Computer Science and Engineering (CSE)
**Islamic University of Technology (IUT), Gazipur**

**Lab # 03**
*8086 String Display and Loop Instructions using Assembly Language in EMU8086.*

**Objective:**
Understand 8086 string display and loop instructions using Assembly Language Program.

**Theory:**

- **String Display Instruction**
  At first define the string to be displayed under DATA SEGMENT:

  ***.DATA***
  ***test_string DB 'My first string', 0Dh, 0Ah, '$'***

  Then, display the string in the command prompt as:
  *MOV AH, 9*
  *LEA DX, test_string*
  *INT 21h*
- **Loop:** LOOP instruction is a combination of a decrement of CX (i.e., count register). In 8086, LOOP decrements CX and if CX is not equal to zero, it jumps to the address indicated by the label. If CX becomes a 0, the next sequential instruction executes.

**Assembly Language Program Example for Loop:**
*Count-controlled LOOP to display a row of 50 stars (*).*

```
org 100h

.DATA       ; Data segment starts
.CODE       ; Code segment starts

MAIN PROC
        mov ax, @DATA
        mov ds, ax
        xor cx, cx  ; reset the CX register

        mov cx, 50
        mov ah, 2
        mov dl, '*'

        top:    int 21h
                loop top

        mov ah, 4ch     ; equivalent function number of RETURN
        int 21h         ; Input-Output Interrupt
MAIN ENDP
END MAIN
RET
```

**Tasks to do:**

1. Write an assembly language program that stores a string in a variable. Now, display the string and using loop calculate number of characters in that string and store that number in DL part of Data Register DX.

   **Sample Input / Output:**
   **Input in a String:** *input_string DB* 'We are IUT Students', 0Dh, 0Ah, '$'
   **Output at DL:** ***We are IUT Students***
   **15h** (Equivalent of Decimal 21 or Binary 00010101 B)

**Course Title:** Microprocessors and Assembly Language Lab (CSE-4504)
Department of Computer Science and Engineering (CSE)
**Islamic University of Technology (IUT), Gazipur**

## Lab # 04

*Understanding Advanced 8086 I/O Instructions using Array in Assembly Language Program.*

## Objective:

To understand some advanced 8086 instructions and getting familiar with the use of Array in Assembly Language Program.

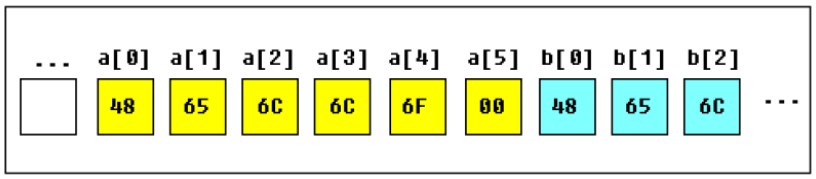
## Theory:

- **Array**

Arrays can be seen as chains of variables. A text string is an example of a byte array; each character is presented as an ASCII code value (0..255). Here are some array definition examples:

```
a DB 48h, 65h, 6Ch, 6Ch, 6Fh, 00h
b DB 'Hello', 0
```

b is an exact copy of the an array, when compiler sees a string inside quotes it automatically converts it to set of bytes. This chart shows a part of the memory where these arrays are declared:




You can access the value of any element in array using square brackets, for example:

```
MOV AL, a[3]
```

You can also use any of the memory index registers **BX, SI, DI, BP**, for example:

```
MOV SI, 3
MOV AL, a[SI]
```

If you need to declare a large array with same value you can use **DUP** operator. The syntax for **DUP**: For example:

```
c DB 5 DUP(0)
c DB 0, 0, 0, 0, 0        ; is an alternative way of declaring:
```

one more example:

```
d DB 5 DUP(1, 2)
d DB 1, 2, 1, 2, 1, 2, 1, 2, 1, 2          ; is an alternative way of declaring:
```

Of course, you can use **DW** instead of **DB** if it's required to keep values larger then 255, or smaller then -128. **DW** cannot be used to declare strings!

**Assembly Language Program Example for Array:**

*To derive summation of a series 1 + 2 + 3 using array. Here, value of N is given by user where N=3 and output is shown in the output window:*

```
org 100h
.DATA                                   ; Data segment starts
A db 3, 1, 2                            ;1-D array for number
B db 00h
message db 'Enter the value of N:$'     ;1-D array for string
.CODE                                   ; Code segment starts
MAIN PROC
mov ax, @DATA
mov ds, ax
xor ax,ax
mov si, OFFSET A
mov di, OFFSET B
mov dx, OFFSET message ; Load Effective Address of the message in DX register
; lea dx, message ; (similar meaning like Load Effective Address)
mov ah, 09h                             ;display string function
int 21h                                 ;display message
mov ah, 01h
int 21h
mov cl, al
sub cl, 48          ; to convert the ascii value of 3 to decimal 3
xor al, al
Loop_1:
        add al, [Si]
        inc Si
        loop Loop_1
mov bl, al
add bl, 48          ; to convert the ascii value of the output to decimal
mov ah, 02h
mov dl, bl
int 21h
MAIN ENDP
END MAIN
RET
```

**Tasks to do:**

1. Write an assembly language code to search the frequency of appearance of a given character from a declared array string (to do this **use ARRAY and Loop**). In your code, declare the string array named 'Input' and also display it as output.

   **Sample Input / Output:**

   islamic university of technology
   Given Character: y
   Number of Frequency: 2

**Course Title:** Microprocessors and Assembly Language Lab (CSE-4504)
Department of Computer Science and Engineering (CSE)
**Islamic University of Technology (IUT), Gazipur**

## Lab # 05

*Understanding **Procedure** using Assembly Language Program.*

## Objective:

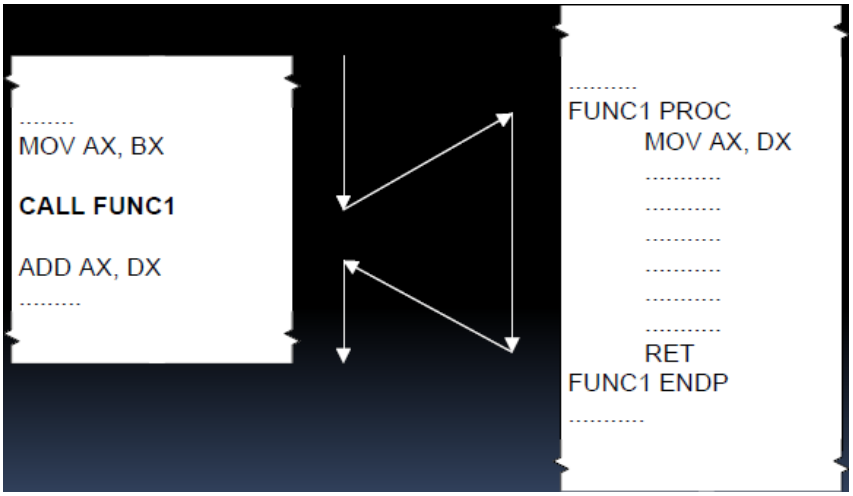
To understand 8086 instructions related to Procedure using Assembly Language Program.

## Theory:

- **Procedures**
  With procedures we are able to write a separate piece of code, **call** it within our program, and return to the point that we left, having completed the code in the procedure. Procedures are also known as subroutines, functions or methods.

  **Call and Return Instructions**

  - We use the **CALL** instruction to transfer execution to the procedure
  - We use the **RET** instruction to return to where the procedure was called from




**Execution of Call instruction results-**
- IP is incremented to point to the next instruction and stored (on the stack)
- The address of the first instruction in the procedure is put into IP
- Execution is restarted in the procedure

**Execution of Return instruction results-**
- The old IP is restored (from the stack)
- Execution is restarted at the point where the procedure was called from

**Assembly Language Program Example for Procedure:**

```
ORG 0100H
.DATA
StrArray DB 'Hello World!!$'          ; define string to display
.CODE
MAIN PROC
        MOV AX, @DATA
        MOV DS,AX
        LEA DX, StrArray      ; set DX to point to 1st element of string array StrArray
        CALL USER             ; call procedure
        MOV AH, 4Ch
        MOV AL, 00h           ; a code after procedure call and return
        INT 21h               ; exit to DOS
MAIN ENDP
USER PROC                     ; declare a procedure named USER
        MOV AH, 09h
        INT 21h
        RET                   ; return to MAIN procedure
USER ENDP                     ; end of procedure USER
END MAIN                      ; end of program
```

**Tasks to do:**

1. Write an Assembly Language code that takes any 5 of decimal digits (0 ~ 9) as input and calculates the average, largest and smallest of them in ***three different procedures*** and store the results in variables like AVERAGE, LARGEST, SMALLEST.

   **Sample Input / Output:**

   Input: 24135

   Output: AVERAGE = 3 LARGEST = 5 SMALLEST = 1