# SYLLABUS

## PART - A

**UNIT – 1 (7 Hours)**

**8086 PROCESSORS:** Historical background, The microprocessor-based personal computer system, 8086 CPU Architecture, Machine language instructions, Instruction execution timing, The 8088

**UNIT – 2 (7 Hours)**

**INSTRUCTION SET OF 8086:** Assembler instruction format, data transfer and arithmetic, branch type, loop, NOP & HALT, flag manipulation, logical and shift and rotate instructions. Illustration of these instructions with example programs, Directives and operators

**UNIT - 3**

**BYTE AND STRING MANIPULATION:** String instructions, REP Prefix, Table translation, Number format conversions, Procedures, Macros, Programming using keyboard and video display

**6 Hours**

**UNIT - 4**

**8086 INTERRUPTS:** 8086 Interrupts and interrupt responses, Hardware interrupt applications, Software interrupt applications, Interrupt examples

**6 Hours**

## PART - B

**UNIT - 5**

**8086 INTERFACING**: Interfacing microprocessor to keyboard (keyboard types, keyboard circuit connections and interfacing, software keyboard interfacing, keyboard interfacing with hardware), Interfacing to alphanumeric displays (interfacing LED displays to microcomputer), Interfacing a microcomputer to a stepper motor

**6 Hours**

**UNIT - 6**

**8086 BASED MULTIPROCESSING SYSTEMS:** Coprocessor configurations, The 8087 numeric data processor: data types, processor architecture, instruction set and examples

**6 Hours**

**UNIT - 7**

**SYSTEM BUS STRUCTURE:** Basic 8086 configurations: minimum mode, maximum mode, Bus Interface: peripheral component interconnect (PCI) bus, the parallel printer interface (LPT), the universal serial bus (USB)

**7 Hours**

**UNIT - 8**

**80386, 80486 AND PENTIUM PROCESSORS:** Introduction to the 80386 microprocessor, Special 80386 registers, Introduction to the 80486 microprocessor, Introduction to the Pentium microprocessor.

**7 Hours**

**TEXT BOOKS:**

1. **Microcomputer systems-The 8086 / 8088** Family – Y.C. Liu and G. A. Gibson, 2E PHI -2003
2. **The Intel Microprocessor, Architecture, Programming and Interfacing**-Barry B. Brey, 6e, Pearson Education / PHI, 2003

**REFERENCE BOOKS:**

1. **Microprocessor and Interfacing- Programming & Hardware**, Douglas hall, 2e TMH, 1991
2. **Advanced Microprocessors and Peripherals** - A.K. Ray and K.M. Bhurchandi, TMH, 2001
3. **8088 and 8086 Microprocessors - Programming, Interfacing, Software, Hardware & Applications** - Triebel and Avtar Singh, 4e, Pearson Education, 2003

INDEX SHEET

# PART-A

## UNIT -1:

**8086 PROCESSORS:** Historical background, The microprocessor-based personal computer system, 8086 CPU Architecture, Machine language instructions, Instruction execution timing,

**TEXT BOOKS:**

3. **Microcomputer systems-The 8086 / 8088** Family – Y.C. Liu and G. A. Gibson, 2E PHI - 2003
4. **The Intel Microprocessor, Architecture, Programming and Interfacing**-Barry B. Brey, 6e, Pearson Education / PHI, 2003

## Historical Background:

The historical events leading to the development of microprocessors are outlined as follows:

### The Mechanical age:

The computing system existed long before modern electrical and electronic devices were invented. During 500 BC, the Babylonians invented the first mechanical calculator called Abacus. The abacus which uses strings of beads to perform calculations was used by Babylonian priests to keep track of their vast storehouses of grains. Abacus was in use until, Blaise Pascal, a mathematician, invented a mechanical calculator constructed of gears and wheels during 1642. Each gear contained 10 teeth that, when moved one complete resolution, advanced a second gear one place. This is the same principle employed in a car's odometer mechanism and is the basis for all mechanical calculators. The arrival of the first, practical geared, mechanical machines used to compute information automatically dates to early 1800's, which is much earlier to the invention of electricity.

Only early pioneer of mechanical computing machinery was Charles Babbage. Babbage was commissioned in 1823 by the astronomical society of Britain to produce a programmable computing machine. This machine was to generate navigational tables for the royal navy. He accepted the challenge and began to create what he called as Analytical Engine. Analytical Engine was a mechanical computer that could store 1000 20-digit decimal numbers and a variable program that could modify the function of machine so it could perform various calculating tasks. Input to the analytical engine was punched cards, which is an idea developed by Joseph Jaquard.

The development of analytical engine stopped because the machinists at that time were unable to create around 50, 000 mechanical parts with enough precision.

### The Electrical age:

The invention of electric motor by Michael Faraday during 1800's lead the way to the development of motor-driven adding machines all based on the mechanical calculator developed by Blaise Pascal. These electrically driven mechanical calculators were in use until the small hand-held electronic calculator developed by Bomar was introduced in 1970's. Monroe is another person who introduced electronic calculators, whose four-function models the size of cash register.

In 1889, Herman Hollerith developed the punched card for storing data, basically the idea was of Jaquard. He also developed a mechanical machine, driven by one of the new electric motors that counted, sorted and collated information stored on punched cards. The punched cards used in

computer systems are often called Hollerith cards, In honor of Herman Hollerith. The 12-bit code used on a punched card is called the Hollerith code.

Electric motor driven mechanical machines dominated the computing world until the German inventor konrad Zuse constructed the first electronic calculating machine, Z3 in the year 1941. Z3 was used in aircraft and missile design during world war II for the German war effort.

In the year 1943, Allan Turing invented the first electronic computing system made of vacuum tubes, which is called as Colossus. Colossus was not programmable; it was a fixed-program computer system, called as special purpose computer.

The first general-purpose, programmable electronic computer system was developed in 1946 at the University of Pennsylvania. This first modern computer was called the ENIAC (Electronics Numerical Integrator And Calculator). The ENIAC was a huge machine, containing over 17,000 vacuum tubes and over 500 miles of wires. This massive machine weighted over 30 tons, yet performed only about 100,000 operations per second. The ENIAC thrust the world into the age of electronic computers. The ENIAC was programmed by rewriting its circuits – a process that took many workers several days to accomplish. The workers changed the electrical connections on plug boards that looked like early telephone switch boards. Another problem with the ENIAC was the life of the vacuum tube components, which required frequent maintenance.

More advancement followed in the computer world with the development of the transistor in 1948 at Bell labs followed by the invention of integrated circuit in 1958 by jack Kilby of Texas instruments.

## The Microprocessor age:

With the invention of integrated circuit technology, Intel introduced the world's first microprocessor, a 4-bit Intel 4004 microprocessor. It addresses a mere 4096 4-bit wide memory locations. The 4004 instructions set contained only 45 instructions. It was fabricated with the P-channel MOSFET technology that only allowed it to execute instructions at the slow rate of 50 KIPS. At first, the applications abounded for this device, like video game systems and small microprocessor based control systems. The main problem with this early microprocessors were its speed, word width and memory size.

Later Intel introduced 4040, an updated version of 4004, which operated at higher speed, although it lacked improvements in word width and memory size.

Intel Corporation released the 8008 an extended 8-bit version of 4004. The 8008 addressed an expanded memory size (16 Kbytes) and contained additional instruction, totally 48 instructions, that provided an opportunity for its application in more advanced systems.

Microprocessors were then used very extensively for many application developments. Many companies like Intel, Motorola, Zilog and many more recognized the demanding requirement of powerful microprocessors to the design world. In fulfilling this requirement many powerful microprocessors were arrived to the market of which we study the Intel's contribution.

## The Intel 8080 & 8085:

- 8080 address more memory and execute additional instructions, but it executed them 10 times faster than 8008.
- The 8080 was compatible with TTL, whereas the 8008 was not directly compatible.
- The 8080 also addressed four times more memory (64 Kbytes) than the 8008 (16 Kbytes).

Intel corporation introduced 8085, an updated version of 8080. Although only slightly more advanced than an 8080 microprocessor, the 8085 executed software at an higher speed. The main advantages of the 8085 were its internal clock generator, internal system controller, and higher clock frequency. This higher level of component integration reduced the 8085's cost and increased its usefulness.

## The 16-bit Microprocessor:

The Intel released 16-bit microprocessors 8086 & 8088, which executed instructions in as little as 400ns (2.5 MIPS). The 8086 & 8088 addressed 1 Mbytes of memory, which was 16 timers more memory than the 8085. 8086/8088 have a small 6-byte instruction cache or queue that pre-fetched a few instructions before they were executed, which leads to the faster processing. 8086/8088 has multiply and divide instructions which were missing in 8085. These microprocessors are called as CISC (Complex Instruction Set Computers) because of the number and complexity of the instructions. The 16-bit microprocessor also provided more internal register storage space that the 8-bit microprocessor. Applications such as spread sheets, word processors, spelling checkers, and computer-based thesauruses on personal computers are few developed using 8086/8088 microprocessors.

## The 80286 microprocessor:

Even the 1 Mbyte memory on 8086/8088 found limited for the advanced applications. This led Intel to introduce the 80286 microprocessor. 80286 follow 8086's 16-bit architecture, except it can address 16 Mbyte memory system. The instruction set was similar to 8086 except few instructions for managing extra 15 Mbytes of memory. The clock speed of 80286 was increased, so it executed some instructions in as little as 250 ns (4 MIPS).

## The 32-bit Microprocessors:

### The 80386 Microprocessor:

The 80386 is the Intel's first 32-bit microprocessor. The 80386 has 32-bit data bus and a 32-bit memory addresses. The 80386 was available in a few modified versions such as 80386SX, 80386SL & 80386SLC, which vary in the amount of memory they address. Applications that require Graphical User Interface (GUI) were using the 80386 microprocessors. Even applications which involve floating-point numbers were using the 80386 microprocessors.

The 80386 included a memory management unit that allowed memory resources to be allocated and managed by the operating system.

### The 80486 Microprocessor:

The 80486 has 80386 like microprocessor, an 80387 like numeric co-processor, and an 8 Kbyte cache memory integrated in it. Most of the instructions in 80486 can be executed in a single clock instead of two clocks compared to 80386. the average speed improvement of instructions was about 50% over the 80386 that operated at the same clock speed.

### The Pentium Microprocessor:

The Pentium microprocessor was introduced late in 1993 with higher speeds compared to 80486. in Pentium cache size was increased to 16 Kbytes from the 8K cache found in the basic version of 80486. After Pentium, many versions were introduced, like Pentium Pro, Pentium II, Pentium III and Pentium IV with higher capacities.

## The Microprocessor-based personal computer system:

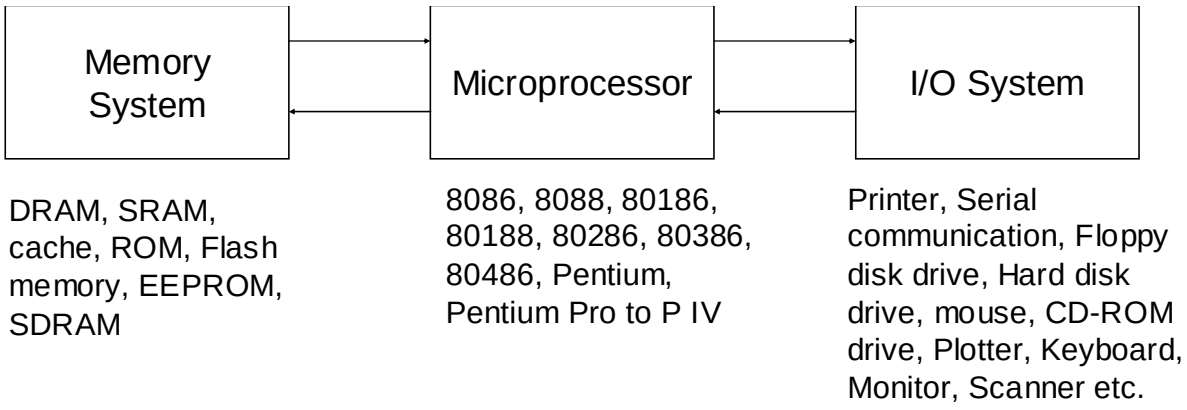



fig (a): The block diagram of a microprocessor-based computer system

The above figure (a) shows the block diagram of a microprocessor based personal computer system. The block diagram comprises of three blocks-memory system, microprocessor and I/O system, which are interconnected by the buses. A bus is a set of common connections that carry the same type of information. There are 3 types of buses – Address bus, Data bus and Control bus in a computer system.

## The Memory System:

The memory structure remains same for all the Intel 80x86 through Pentium IV personal computer systems. Fig (b) illustrates the memory map of a personal computer system.

The memory system is divided into three main parts:

- Transient Program Area (TPA) – 640 Kbytes.
- System Area – 384 Kbytes.
- Extended Memory system (XMS) – amount of memory depends on the microprocessor used in the personal computer system.

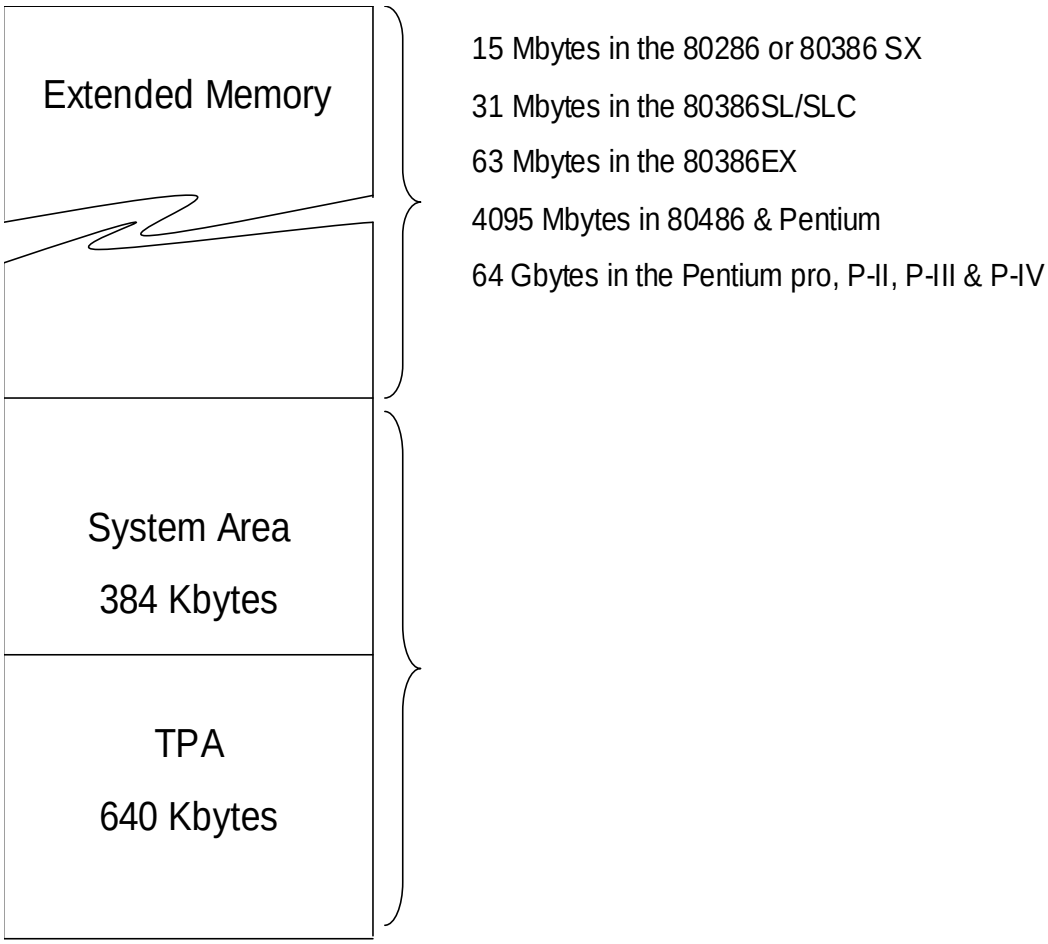



Fig (b): The memory map of the personal computer

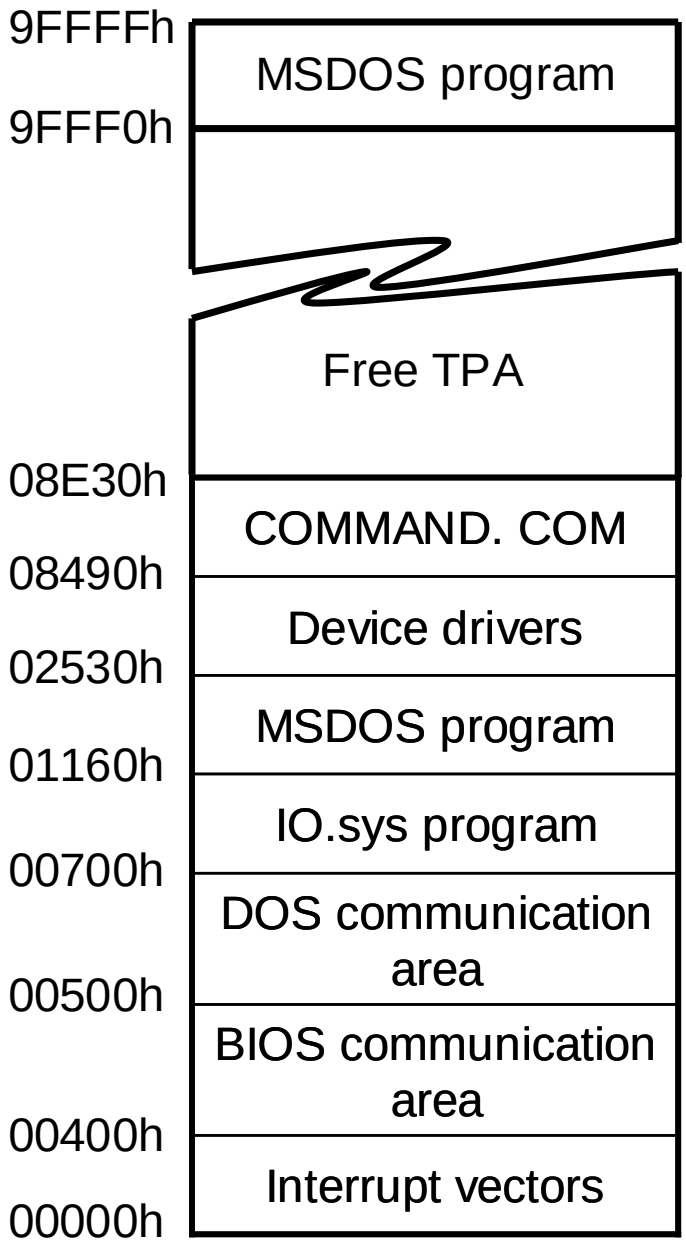
The type of microprocessor in the personal computer system determines whether XMS exists or not. 8086 or 8088 (PC or XT10) based computer system consists of 640 Kbytes of TPA and 384 Kbytes of system area which accounts to the 1 Mbyte of memory and there is no extended memory area. The first 1M bytes of memory are called the real or conventional memory because each Intel

microprocessor is designed to function in this area by using its real mode of operation. Computer systems based on the 80286 through P-IV not only contain the TPA (640K bytes) and system area (384K bytes), they also contain the extended memory.

## a. Transient Program Area (TPA):

The memory map shown in fig (c) illustrates how the many areas of the TPA are used for system programs, data and drivers. It also shows a large area of memory available for application programs.

The TPA holds the DOS operating system and other programs that control the computer system. If the MSDOS version 7.x is used as an operating system, of the 640k bytes of TPA, 628k bytes of the memory will be available for application programs.



The interrupt vectors accesses various features of DOS, BIOS & applications. The BIOS is a collection of programs stored in either a ROM or flash memory that operate many of the I/O devices connected to the computer system.

The BIOS & DOS communication areas contain transient data used by programs to access I/O devices and the internal features of the computer system. These are stored in the TPA so they can be changed as the system operates.

The IO.sys is a program that loads into the TPA from the disk whenever an MSDOS or PCDOS system is started. The IO.sys contains programs that allow DOS to use the keyboard, video display, printer, and other I/O devices.

The MSDOS program occupies two areas of memory. One area is 16 bytes in length and is located at the top of TPA. The other is much larger and is located near the bottom of TPA.

The size of the driver area and number of drivers change from one computer to another. Drivers are programs that control installable I/O devices such as CD-ROM, Mouse etc. drivers are normally files that have an extension of .sys. The COMMAND.com (command processor) controls the operation of the computer from the keyboard. The free TPA area holds application programs as they are executed. These application programs include word processors, spread sheet programs, CAD programs and many more.

## b. The System Area:

The system area contains programs on either a ROM or flash memory and areas of read/write (RAM) memory for the storage. The length of the system area is 384k bytes. Fig (d) shows the system area of a typical computer system.

The first area of the system space contains video display RAM and video control programs on ROM or flash memory. This area starts at location A0000h and extends to location C7FFFh. The size and amount of memory used depends on the type of the video display adapter attached to the system. Ex: CGA (Color Graphics Adapter), EGE (Extended Graphics Adapter) and VGA (Variable Graphics Adapter). Generally the video RAM located at A0000h – AFFFFh stores text data. The video BIOS, located on a ROM or flash memory, are at locations C0000h – C7FFFh and contain programs that control the video display.

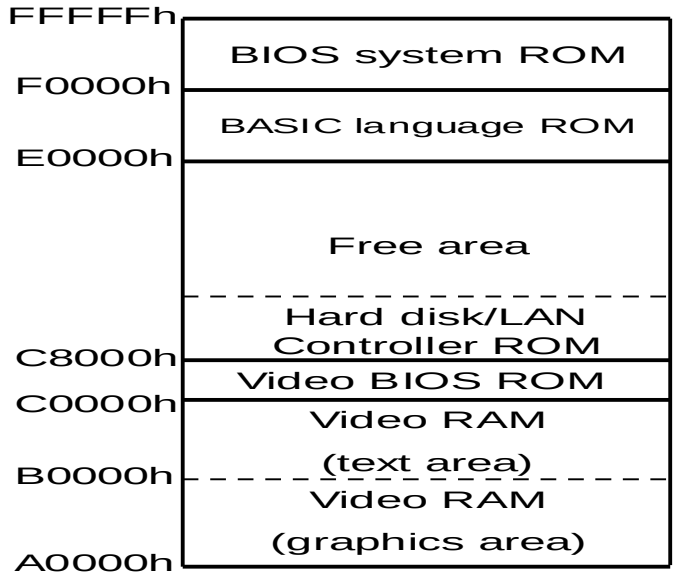
If a hard disk memory is attached to the computer, the low-level format software will be at location C8005h.

The area at locations C8000h – DFFFFh is often open or free. This area is used for the expanded memory system (EMS) in a PC or XT system, or for the upper memory system in an AT system.

The expanded memory system allows a 64k byte page frame of memory to be used by application programs.

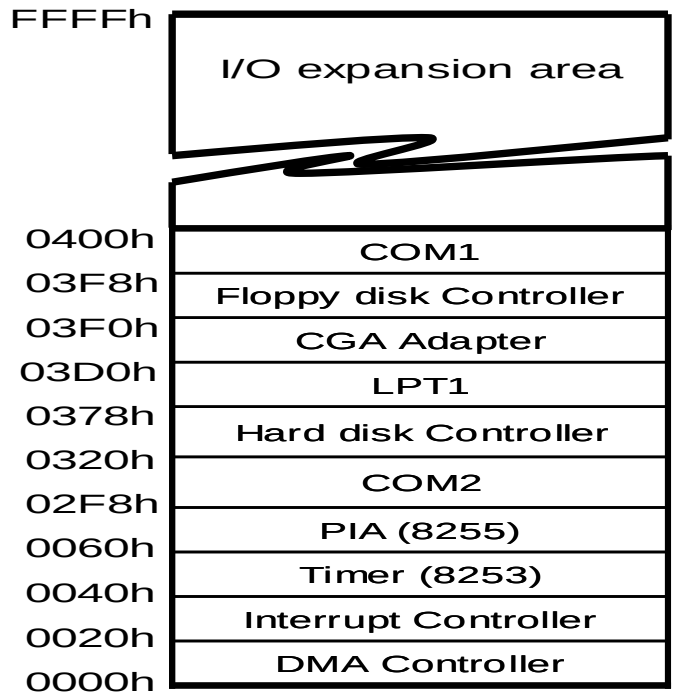
Memory locations E0000h – EFFFFh contain the cassette BASIC language on ROM found in early IBM personal computer systems. This area is often open or free in newer systems.

The system BIOS ROM is located in the top 64k bytes of the system area (F0000h – FFFFFh). This ROM controls the operation of the basic I/O devices connected to the computer system. It doesn't control the operation of the video system, which has its own BIOS ROM at location C0000h. The first part of the system BIOS (F0000h – F7FFFh) often contains the programs that setup the computer and the second part contains procedures that control the basic I/O system.




## The I/O space:

The I/O devices allow the microprocessor to communicate b/w itself and the outside world. The I/O space in a computer system extends from I/O port 0000h to port 0FFFFh. This address range can access up to 64k different 8-bit I/O devices.




The I/O area contains two major sections. The area below I/O location 0400h is considered reserved for system devices. The remaining area is available I/O space for expansion on newer systems that extends from I/O port 0400h through 0FFFFh. Generally, I/O addresses b/w 0000h and 00FFh address components on the main board of the computer, while addresses between 0100h and 03FFh address devices located on plug-in cards.

## The Microprocessor:

The microprocessor is the heart of the microprocessor-based computer system. Microprocessor is the controlling element and is sometimes referred to as the Central Processing Unit (CPU). The microprocessor controls memory and I/O through a series of connections called buses.

The microprocessor performs three main tasks for the computer system:

- Data transfer between itself and the memory or I/O systems.
- Simple arithmetic and logic operations, and
- Program flow via simple decisions.

Although, these are simple tasks, but through them the microprocessor performs virtually any series of operations.

**Simple Microcomputer Bus Operation**

1. A microcomputer fetches each program instruction in sequence, decodes the instruction, and executes it.

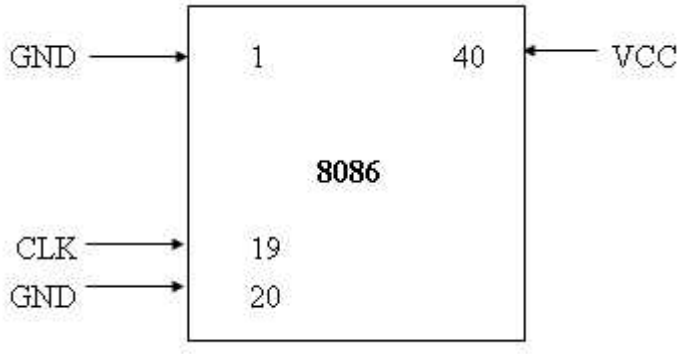
2. The CPU in a microcomputer fetches instructions or reads data from memory by sending out an address on the address bus and a Memory Read signal on the control bus. The memory outputs the addressed instruction or data word to the CPU on the data bus.
3. The CPU writes a data word to memory by sending out an address on the address bus, sending out the data word on the data bus, and sending a Memory write signal to memory on the control bus.
4. To read data from a port, the CPU sends out the port address on the address bus and sends an I/O Read signal to the port device on the control bus. Data from the port comes into the CPU on the data bus.
5. To write data to a port, the CPU sends out the port address on the address bus, sends out the data to be written to the port on the data bus, and sends an I/O Write signal to the port device on the control bus.

## 8086 Pin diagram

| | | | | Maximum Mode | Minimum Mode |
|---|---|---|---|---|---|
| GND | 1 | | 40 | VCC | |
| AD14 | 2 | | 39 | $AD_{15}$ | |
| AD13 | 3 | | 38 | $A_{17}/S_4$ | |
| AD12 | 4 | | 37 | $A_{16}/S_3$ | |
| AD11 | 5 | | 36 | $A_{18}/S_5$ | |
| AD10 | 6 | | 35 | $A_{19}/S_6$ | |
| AD9 | 7 | | 34 | $\overline{BHE}/S_7$ | |
| AD8 | 8 | | 33 | MN/$\overline{MX}$ | |
| AD7 | 9 | | 32 | $\overline{RD}$ | |
| AD6 | 10 | | 31 | $\overline{RQ}/GT_1$ | (HOLD) |
| AD5 | 11 | **8086** | 30 | $\overline{RQ}/GT_0$ | (HLDA) |
| AD4 | 12 | | 29 | $\overline{LOCK}$ | ($\overline{WR}$) |
| AD3 | 13 | | 28 | $\overline{S_2}$ | (M/$\overline{IO}$) |
| AD2 | 14 | | 27 | $\overline{S_1}$ | $DT/\overline{R}$ |
| AD1 | 15 | | 26 | $\overline{S_0}$ | $\overline{DEN}$ |
| AD0 | 16 | | 25 | $QS_0$ | (ALE) |
| NMI | 17 | | 24 | $QS_1$ | ($\overline{INTA}$) |
| INTR | 18 | | 23 | $\overline{TEST}$ | |
| CLK | 19 | | 22 | READY | |
| GND | 20 | | 21 | RESET | |

8086 is a 40 pin DIP using MOS technology. It has 2 GND's as circuit complexity demands a large amount of current flowing through the circuits, and multiple grounds help in dissipating the accumulated heat etc. 8086 works on two modes of operation namely, Maximum Mode and Minimum Mode.

## (i) Power Connections



**Pin Description:**

**GND – Pin no. 1, 20**

Ground

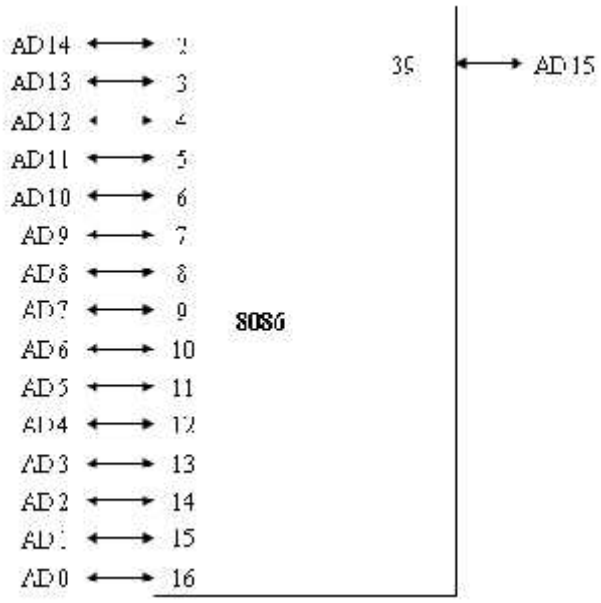
**CLK – Pin no. 19 – Type I**

Clock: provides the basic timing for the processor and bus controller. It is asymmetric with a 33% duty cycle to provide optimized internal timing.

**$V_{CC}$ – Pin no. 40**

$V_{CC}$: +5V power supply pin

## (ii) Address/ Data Lines

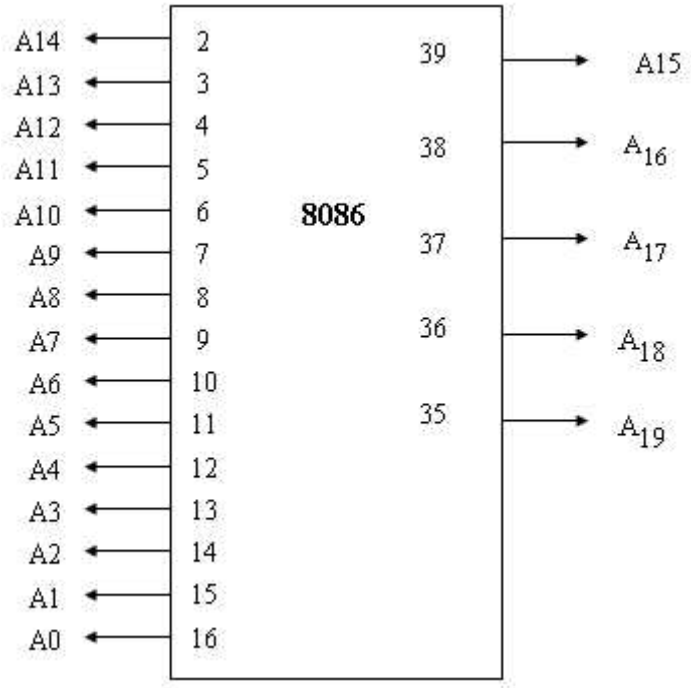



**Pin Description**

**$AD_{15}$-$AD_0$ – Pin no. 2-16, 39 – Type I/O**

**Address Data bus:** These lines constitute the time multiplexed memory/ IO address (T1) and data ($T_2$, $T_3$, $T_W$, $T_4$) bus. $A_0$ is analogous to $\overline{BHE}$ for the lower byte of of the data bus, pins $D_7$-$D_0$. It iss low when a byte is to be transferred on the lower portion of the bus in memory or I/O operations. Eight –bit oriented devices tied to the lower half would normally use $A_0$ to condition chip select functions. These lines are active HIGH and float to 3-state OFF during interrupt acknowledge and local bus “hold acknowledge”.

## (iii) Address Lines



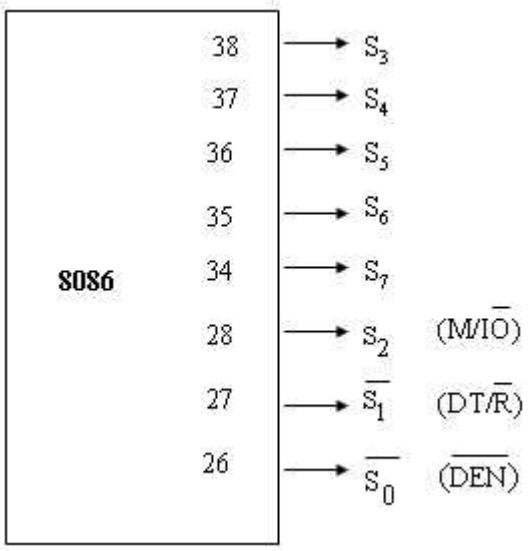

**$A_{19}/S_6$, $A_{18}/S_5$, $A_{17}/S_4$, $A_{16}/S_3$ – Pin no. 35-38 – Type O**

Address / Status: During $T_1$ these are the four most significant address lines for memory operations. During I/O operations these lines are low. During memory and I/O operations, status information is available on these lines during $T_2$, $T_3$, $T_W$ and $T_4$. The status of the interrupt enable FLAG bit (S5) is updated at the beginning of each CLK cycle. $A_{17}/S_4$ and $A_{16}/S_3$ are encoded as shown.

| $A_{17}/S_4$ | $A_{16}/S_3$ | Characteristics |
|---|---|---|
| 0 (LOW) | 0 | Alternate Data |
| 0 | 1 | Stack |
| 1(HIGH) | 0 | Code or None |
| 1 | 1 | Data |
| $S_6$ is 0 (LOW) | | |

This information indicates which relocation register is presently being used for data accessing. These lines float to 3-state OFF during local bus “hold acknowledge”.

## (iv) Status Pins $S_0$ - $S_7$




**Pin Description**

$\overline{S_2}, \overline{S_1}, \overline{S_0}$ **- Pin no. 26, 27, 28 – Type O**

Status: active during $T_4$, $T_1$ and $T_2$ and is returned to the passive state (1,1,1) during $T_3$ or during $T_W$ when READY is HIGH. This status is used by the 8288 Bus Controller to generate all memory and I/O access control signals. Any change by $\overline{S_2}$, $\overline{S_1}$ or $\overline{S_0}$ during $T_4$ is used to indicate the beginning of a bus cycle and the return to the passive state in $T_3$ or $T_W$ is used to indicate the end of a bus cycle.

These signals float to 3-state OFF in "hold acknowledge". These status lines are encoded as shown.

| $\overline{S_2}$ | $\overline{S_1}$ | $\overline{S_0}$ | **Characteristics** |
|---|---|---|---|
| 0(LOW) | 0 | 0 | Interrupt acknowledge |
| 0 | 0 | 1 | Read I/O Port |
| 0 | 1 | 0 | Write I/O Port |
| 0 | 1 | 1 | Halt |
| 1(HIGH) | 0 | 0 | Code Access |
| 1 | 0 | 1 | Read Memory |
| 1 | 1 | 0 | Write Memory |
| 1 | 1 | 1 | Passive |

**Status Details**

| $\overline{S_2}$ | $\overline{S_1}$ | $\overline{S_0}$ | Indication |
|---|---|---|---|
| 0 | 0 | 0 | Interrupt Acknowledge |

| | | | |
|---|---|---|---|
| 0 | 0 | 1 | Read I/O port |
| 0 | 1 | 0 | Write I/O port |
| 0 | 1 | 1 | Halt |
| 1 | 0 | 0 | Code access |
| 1 | 0 | 1 | Read memory |
| 1 | 1 | 0 | Write memory |
| 1 | 1 | 1 | Passive |

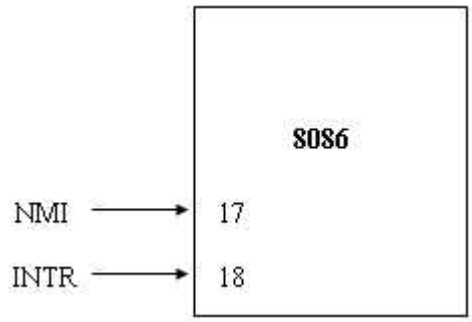
| **S4** | **S3** | **Indications** |
|---|---|---|
| 0 | 0 | Alternate data |
| 0 | 1 | Stack |
| 1 | 0 | Code or none |
| 1 | 1 | Data |

$\overline{S_5}$ ----- Value of Interrupt Enable flag

$\overline{S_6}$ ----- Always low (logical) indicating 8086 is on the bus. If it is tristated another bus master has taken control of the system bus.

$\overline{S_7}$ ----- Used by 8087 numeric coprocessor to determine whether the CPU is a 8086 or 8088

# (v) Interrupts




**Pin Description:**

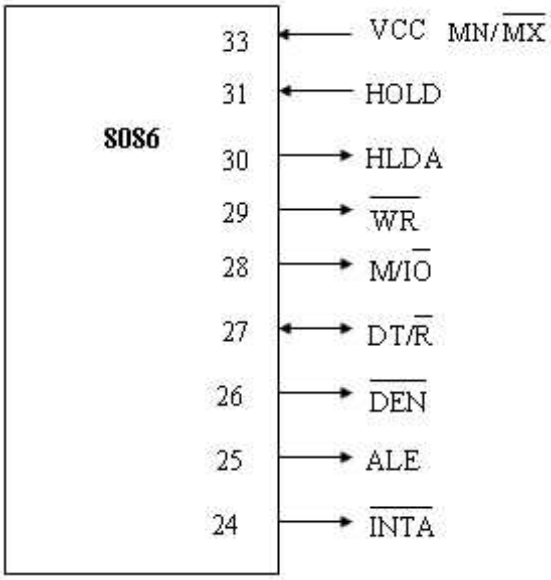
**NMI – Pin no. 17 – Type I**

Non – Maskable Interrupt: an edge triggered input which causes a type 2 interrupt. A subroutine is vectored to via an interrupt vector lookup table located in system memory. NMI is not maskable internally by software. A transition from a LOW to HIGH initiates the interrupt at the end of the current instruction. This input is internally synchronized.

**INTR – Pin No. 18 – Type I**

Interrupt Request: is a level triggered input which is sampled during the last clock cycle of each instruction to determine if the processor should enter into an interrupt acknowledge operation. A subroutine is vectored to via an interrupt vector lookup table located in system memory. It can be internally masked by software resetting the interrupt enable bit. INTR is internally synchronized. This signal is active HIGH.

## (vi) Min mode signals




**Pin Description:**

**HOLD, HLDA – Pin no. 31, 30 – Type I/O**

**HOLD:** indicates that another master is requesting a local bus "hold". To be acknowledged, HOLD must be active HIGH. The processor receiving the "hold" request will issue HLDA (HIGH) as an acknowledgement in the middle of a $T_1$ clock cycle. Simultaneous with the issuance of HLDA the processor will float the local bus and control lines. After HOLD is detected as being LOW, the processor will LOWer the HLDA, and when the processor needs to run another cycle, it will again drive the local bus and control lines.

The same rules as $\overline{RQ}/\overline{GT}$ apply regarding when the local bus will be released.

HOLD is not an asynchronous input. External synchronization should be provided if the system can not otherwise guarantee the setup time.

$\overline{WR}$ **- Pin no. 29 – Type O**

Write: indicates that the processor is performing a write memory or write I/O cycle, depending on the state of the $M/\overline{IO}$ signal. $\overline{WR}$ is active for $T_2$, $T_3$ and $T_W$ of any write cycle. It is active LOW, and floats to 3-state OFF in local bus "hold acknowledge".

$M/\overline{IO}$ **- Pin no. 28 – type O**

Status line: logically equivalent to $S_2$ in the maximum mode. It is used to distinguish a memory access from an I/O access. $M/\overline{IO}$ becomes valid in the $T_4$ preceding a bus cycle and remains valid until the final $T_4$ of the cycle (M=HIGH), IO=LOW). $M/\overline{IO}$ floats to 3-state OFF in local bus "hold acknowledge".

$DT/\overline{R}$ **-Pin no. 27 – Type O**

Data Transmit / Receive: needed in minimum system that desires to use an 8286/8287 data bus transceiver. It is used to control the direction of data flow through the transceiver. Logically $DT/\overline{R}$ is equivalent to $\overline{S_1}$ in the maximum mode, and its timing is the same as for $M/\overline{IO}$. (T=HIGH, R=LOW). This signal floats to 3-state OFF in local bus "hold acknowledge".

$\overline{DEN}$ **- Pin no. 26 – Type O**

Data Enable: provided as an output enable for the 8286/8287 in a minimum system which uses the transceiver. $\overline{DEN}$ is active LOW during each memory and I/O access and for INTA cycles. For a read or $\overline{INTA}$ cycle it is active from the middle of $T_2$ until the middle of $T_4$, while for a write cycle it is active from the beginning of $T_2$ until the middle of $T_4$. $\overline{DEN}$ floats to 3-state OFF in local bus "hold acknowledge".

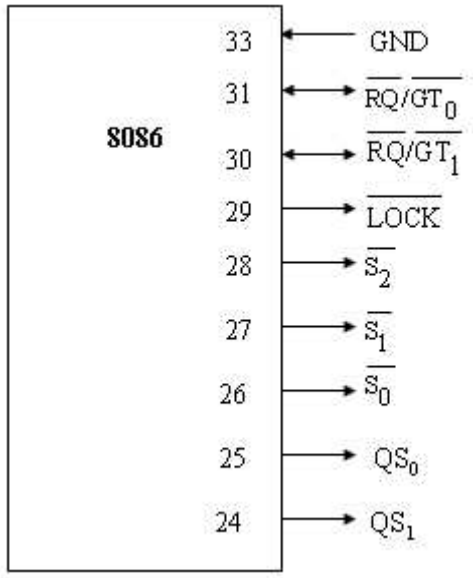
**ALE – Pin no. 25 – Type O**

Address Latch Enable: provided by the processor to latch the address into the 8282/8283 address latch. It is a HIGH pulse active during $T_1$ of any bus cycle. Note that ALE is never floated.

$\overline{INTA}$ **- Pin no. 24 – Type O**

$\overline{INTA}$ is used as a read strobe for interrupt acknowledge cycles. It is active LOW during $T_2$, $T_3$ and $T_W$ of each interrupt acknowledge cycle.

# (vii) Max mode signals




**Pin Description:**

$\overline{RQ}/\overline{GT_0}$ , $\overline{RQ}/\overline{GT_1}$ **- Pin no. 30, 31 – Type I/O**

Request /Grant: pins are used by other local bus masters to force the processor to release the local bus at the end of the processor's current bus cycle. Each pin is bidirectional with $\overline{RQ}/\overline{GT_0}$ having higher priority than $\overline{RQ}/\overline{GT_1}$. $\overline{RQ}/\overline{GT}$ has an internal pull up resistor so may be left unconnected. The request/grant sequence is as follows:

1. A pulse of 1 CLK wide from another local bus master indicates a local bus request ("hold") to the 8086 (pulse 1)
2. During a $T_4$ or $T_1$ clock cycle, a pulse 1 CLK wide from the 8086 to the requesting master (pulse 2), indicates that the 8086 has allowed the local bus to float and that it will enter the "hold acknowledge" state at the next CLK. The CPU's bus interface unit is disconnected logically from the local bus during "hold acknowledge".
3. A pulse 1 CLK wide from the requesting master indicates to the 8086 (pulse 3) that the "hold" request is about to end and that the 8086 can reclaim the local bus at the next CLK.

Each master-master exchange of the local bus is a sequence of 3 pulses. There must be one dead CLK cycle after each bus exchange. Pulses are active LOW.

If the request is made while the CPU is performing a memory cycle, it will release the local bus during $T_4$ of the cycle when all the following conditions are met:

1. Request occurs on or before $T_2$.
2. Current cycle is not the low byte of a word (on an odd address)
3. Current cycle is not the first acknowledge of an interrupt acknowledge sequence.

4. A locked instruction is not currently executing.

$\overline{LOCK}$ **- Pin no. 29 – Type O**

$\overline{LOCK}$: output indicates that other system bus masters are not to gain control of the system bus while $\overline{LOCK}$ is active LOW. The $\overline{LOCK}$ signal is activated by the "LOCK" prefix instruction and remains active until the completion of the next instruction. This signal is active LOW, and floats to 3-state OFF in "hold acknowledge".

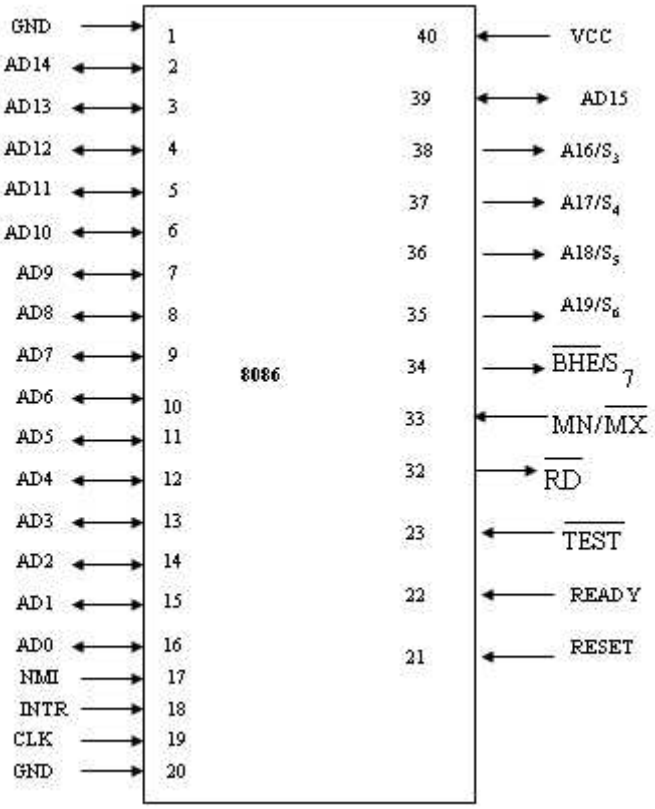
**$QS_1$, $QS_0$ – Pin no. 24, 25 – Type O**

Queue Status: the queue status is valid during the CLK cycle after which the queue operation is performed.

$QS_1$ and $QS_0$ provide status to allow external tracking of the internal 8086 instruction queue.

| **$QS_1$** | **$QS_0$** | **Characteristics** |
|---|---|---|
| 0(LOW) | 0 | No operation |
| 0 | 1 | First Byte of Op Code from Queue |
| 1 (HIGH) | 0 | Empty the Queue |
| 1 | 1 | Subsequent byte from Queue |

# (viii) Common Signals




**Pin Description:**

**$\overline{RD}$ - Pin no. 34, Type O**

Read: Read strobe indicates that the processor is performing a memory of I/O read cycle, depending on the state of the $S_2$ pin. This signal is used to read devices which reside on the 8086 local bus. $\overline{RD}$ is active LOW during $T_2$, $T_3$ and $T_W$ of any read cycle, and is guaranteed to remain HIGH in $T_2$ until the 8086 local bus has floated.

This signal floats to 3-state OFF in "hold acknowledge".

**READY – Pin no. 22, Type I**

READY: is the acknowledgement from the addressed memory or I/O device that it will complete the data transfer. The READY signal from memory / IO is synchronized by the 8284A Clock Generator to form READY. This signal is active HIGH. The 8086 READY input is not synchronized. Correct operation is not guaranteed if the setup and hold times are not met.

**$\overline{TEST}$ - Pin No 23 – Type I**

$\overline{TEST}$: input is examined by the "Wait" instruction. If the $\overline{TEST}$ input is LOW execution continues, otherwise the processor waits in an "idle" state. This input is synchronized internally during each clock cycle on the leading edge of CLK.

**RESET – Pin no. 21 – Type I**

Reset: causes the processor to immediately terminate its present activity. The signal must be active HIGH for at least four clock cycles. It restarts execution, as described in the instruction set description, when RESET returns LOW. RESET is internally synchronized.

**$\overline{BHE}/S_7$ - Pin No. 34 – Type O**

Bus High Enable / Status: During $T_1$ the Bus High Enable signal ($\overline{BHE}$)should be used to enable data onto the most significant half of the data bus, pins $D_{15}$-$D_8$. Eight bit oriented devices tied to the upper half of the bus would normally use $\overline{BHE}$ to condition chip select functions. $\overline{BHE}$ is LOW during $T_1$ for read, write, and interrupt acknowledge cycles when a byte is to be transferred on the high portion of the bus. The $S_{,7}$ status information is available during $T_2$, $T_3$ and $T_4$. The signal is active LOW and floats to 3-state OFF in "hold". It is LOW during $T_1$ for the first interrupt acknowledge cycle.

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

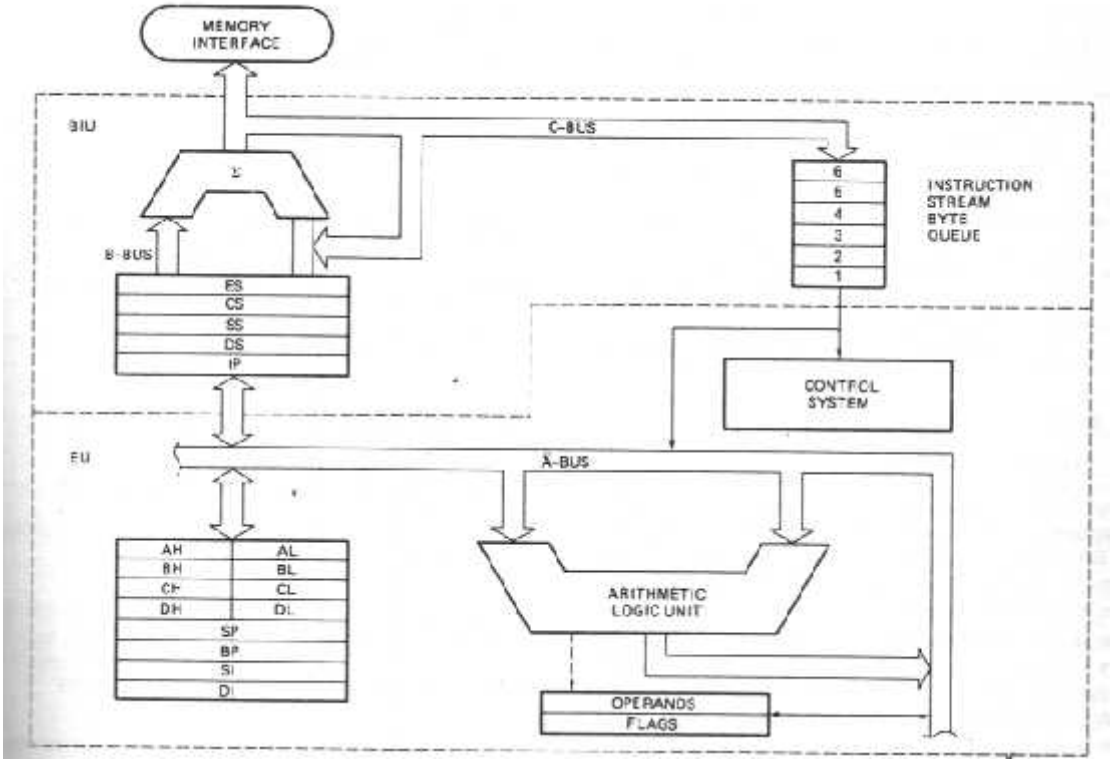
**MN/$\overline{MX}$ - Pin no. 33 – Type - I**

Minimum / Maximum: indicates what mode the processor is to operate in.

If the local bus is idle when the request is made the two possible events will follow:

1. Local bus will be released during the next clock.
2. A memory cycle will start within 3 clocks. Now the four rules for a currently active memory cycle apply with condition number 1 already satisfied.

## 8086 CPU ARCHITECTURE




The block diagram of 8086 is as shown. This can be subdivided into two parts, namely the Bus Interface Unit and Execution Unit. The Bus Interface Unit consists of segment registers, adder to generate 20 bit address and instruction prefetch queue.

Once this address is sent out of BIU, the instruction and data bytes are fetched from memory and they fill a First In First Out 6 byte queue.

**Execution Unit:**

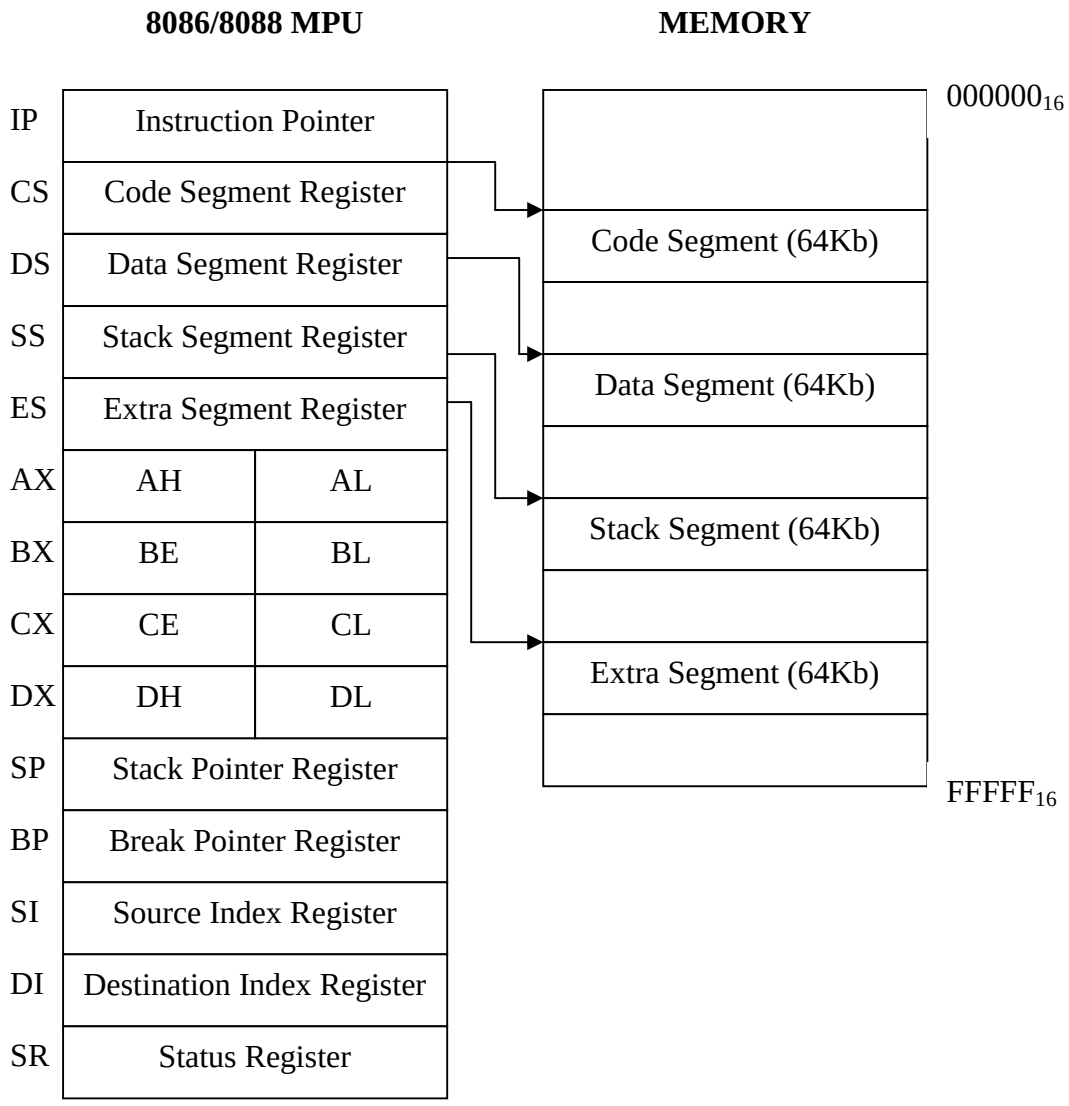
The execution unit consists of scratch pad registers such as 16-bit AX, BX, CX and DX and pointers like SP (Stack Pointer), BP (Base Pointer) and finally index registers such as source index and destination index registers. The 16-bit scratch pad registers can be split into two 8-bit registers. For example, AX can be split into AH and AL registers. The segment registers and their default offsets are given below.

| Segment Register | Default Offset |
|---|---|
| CS | IP (Instruction Pointer) |
| DS | SI, DI |
| SS | SP, BP |
| ES | DI |

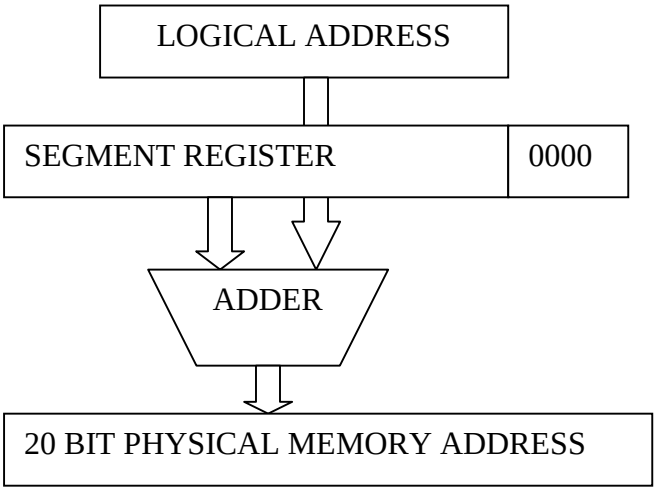
The Arithmetic and Logic Unit adjacent to these registers perform all the operations. The results of these operations can affect the condition flags.

Different registers and their operations are listed below:

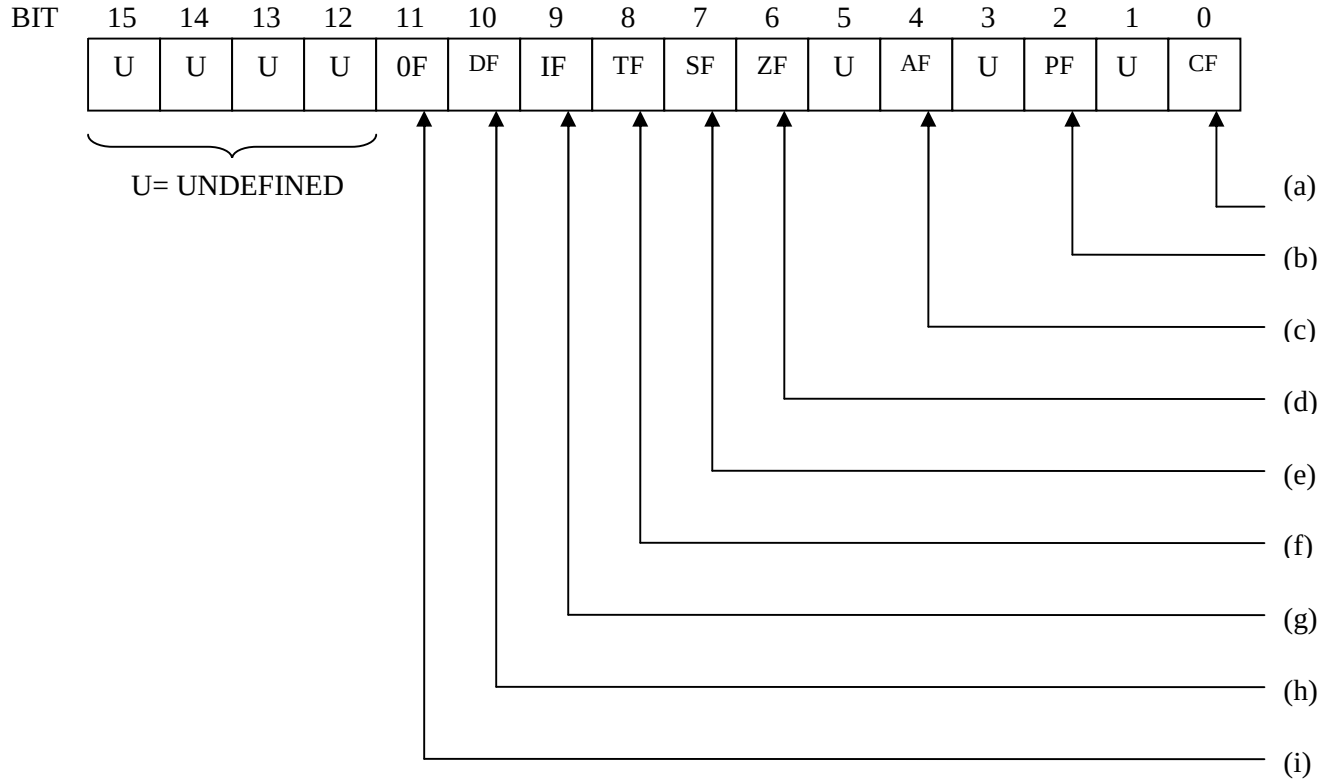
| **Register** | **Operations** |
|---|---|
| AX | Word multiply, Word divide, word I/O |
| AL | Byte Multiply, Byte Divide, Byte I/O, translate, Decimal Arithmetic |
| AH | Byte Multiply, Byte Divide |
| BX | Translate |
| CX | String Operations, Loops |
| CL | Variable Shift and Rotate |
| DX | Word Multiply, word Divide, Indirect I/O |




**Generation of 20-bit Physical Address:**


LOGICAL ADDRESS
SEGMENT REGISTER
0000
ADDER
20 BIT PHYSICAL MEMORY ADDRESS

**8086 flag register format**




(a) : CARRY FLAG – SET BY CARRY OUT OF MSB
(b) : PARITY FLAG – SET IF RESULT HAS EVEN PARITY
(c) : AUXILIARY CARRY FLAG FOR BCD
(d) : ZERO FLAG – SET IF RESULT = 0
(e) : SIGN FLAG = MSB OF RESULT
(f) : SINGLE STEP TRAP FLAG
(g) : INTERRUPT ENABLE FLAG
(h) : STRING DIRECTION FLAG
(i) : OVERFLOW FLAG

There are three internal buses, namely A bus, B bus and C bus, which interconnect the various blocks inside 8086.

The execution of instruction in 8086 is as follows:

The microprocessor unit (MPU) sends out a 20-bit physical address to the memory and fetches the first instruction of a program from the memory. Subsequent addresses are sent out and the queue is filled upto 6 bytes. The instructions are decoded and further data (if necessary) are fetched from memory. After the execution of the instruction, the results may go back to memory or to the output peripheral devices as the case may be.

Machine language:

# Addressing Modes

### Addressing modes of 8086

When 8086 executes an instruction, it performs the specified function on data. These data are called its operands and may be part of the instruction, reside in one of the internal registers of the microprocessor, stored at an address in memory or held at an I/O port, to access these different types of operands, the 8086 is provided with various addressing modes (Data Addressing Modes).

**Data Addressing Modes of 8086**

The 8086 has 12 addressing modes. The various 8086 addressing modes can be classified into five groups.

A. Addressing modes for accessing immediate and register data (register and immediate modes).
B. Addressing modes for accessing data in memory (memory modes)
C. Addressing modes for accessing I/O ports (I/O modes)
D. Relative addressing mode
E. Implied addressing mode

**8086 ADDRESSING MODES**

**A. Immediate addressing mode:**

In this mode, 8 or 16 bit data can be specified as part of the instruction.

| OP Code | Immediate Operand |
|---|---|

Example 1 : MOV CL, 03 H
Moves the 8 bit data 03 H into CL

Example 2 : MOV DX, 0525 H
Moves the 16 bit data 0525 H into DX

In the above two examples, the source operand is in immediate mode and the destination operand is in register mode.

A constant such as "VALUE" can be defined by the assembler EQUATE directive such as VALUE EQU 35H

Example : MOV BH, VALUE
Used to load 35 H into BH

**Register addressing mode :**

The operand to be accessed is specified as residing in an internal register of 8086. Fig. below shows internal registers, any one can be used as a source or destination operand, however only the data registers can be accessed as either a byte or word.

| **Register** | **Operand sizes** | |
|---|---|---|
| | **Byte (Reg 8)** | **Word (Reg 16)** |
| Accumulator | AL, AH | Ax |
| Base | BL, BH | Bx |
| Count | CL, CH | Cx |
| Data | DL, DH | Dx |
| Stack pointer | - | SP |
| Base pointer | - | BP |
| Source index | - | SI |
| Destination index | - | DI |
| Code Segment | - | CS |
| Data Segment | - | DS |
| Stack Segment | - | SS |
| Extra Segment | - | ES |

**Example 1** : MOV DX (Destination Register) , CX (Source Register)
Which moves 16 bit content of CS into DX.

**Example 2** : MOV CL, DL
Moves 8 bit contents of DL into CL

MOV BX, CH is an illegal instruction.

* The register sizes must be the same.

**B. Direct addressing mode :**

The instruction Opcode is followed by an affective address, this effective address is directly used as the 16 bit offset of the storage location of the operand from the location specified by the current value in the selected segment register.

**The default segment is always DS.**

The 20 bit physical address of the operand in memory is normally obtained as

PA = DS : EA

But by using a segment override prefix (SOP) in the instruction, any of the four segment registers can be referenced,

PA = { CS / DS / SS / ES } : { Direct Address }

The Execution Unit (EU) has direct access to all registers and data for register and immediate operands. However the EU cannot directly access the memory operands. It must use the BIU, in order to access memory operands.

In the direct addressing mode, the 16 bit effective address (EA) is taken directly from the displacement field of the instruction.

**Example 1 :** MOV CX, START

If the 16 bit value assigned to the offset START by the programmer using an assembler pseudo instruction such as DW is 0040 and [DS] = 3050.

Then BIU generates the 20 bit physical address 30540 H.

The content of 30540 is moved to CL

The content of 30541 is moved to CH

**Example 2 :** MOV CH, START

If [DS] = 3050 and START = 0040

8 bit content of memory location 30540 is moved to CH.

**Example 3 :** MOV START, BX

With [DS] = 3050, the value of START is 0040.

Physical address : 30540

MOV instruction moves (BL) and (BH) to locations 30540 and 30541 respectively.

**Register indirect addressing mode :**

The EA is specified in either pointer (BX) register or an index (SI or DI) register. The 20 bit physical address is computed using DS and EA.

Example : MOV [DI], BX

register indirect

If [DS] = 5004, [DI] = 0020, [Bx] = 2456 PA=50060.

The content of BX(2456) is moved to memory locations 50060 H and 50061 H.

PA = { CS, DS, SS, ES } = { BX, SI, DI }

**Based addressing mode**:

PA = { CS, DS, SS, ES } : { BX or BP } + displacement

when memory is accessed PA is computed from BX and DS when the stack is accessed PA is computed from BP and SS.

**Example** : MOV AL, START [BX]

or

MOV AL, [START + BX]

based mode

EA : [START] + [BX]

PA : [DS] + [EA]

The 8 bit content of this memory location is moved to AL.

**Indexed addressing mode:**

PA = { CS, DS, SS, ES } : { SI or DI } + 8 or 16bit displacement

**Example** : MOV BH, START [SI]

PA : [SART] + [SI] + [DS]

The content of this memory is moved into BH.

**Based Indexed addressing mode:**

PA = { CS / DS / SS } : { BX / or } + { SI / or } + 8 or 16bit displacement

ES BP DI

**Example** : MOV ALPHA [SI] [BX], CL

If [BX] = 0200, ALPHA – 08, [SI] = 1000 H and [DS] = 3000

Physical address (PA) = 31208

8 bit content of CL is moved to 31208 memory address.

**String addressing mode:**

The string instructions automatically assume SI to point to the first byte or word of the source operand and DI to point to the first byte or word of the destination operand. The contents of SI and DI are automatically incremented (by clearing DF to 0 by CLD instruction) to point to the next byte or word.

Example : MOV S BYTE

If [DF] = 0, [DS] = 2000 H, [SI] = 0500,

[ES] = 4000, [DI] = 0300

Source address : 20500, assume it contains 38

PA : [DS] + [SI]

Destination address : [ES] + [DI] = 40300, assume it contains 45

After executing MOV S BYTE,

[40300] = 38

[SI] = 0501 } incremented

[DI] = 0301

**C. I/O mode (direct) :**

Port number is an 8 bit immediate operand.

Example : OUT 05 H, AL

Outputs [AL] to 8 bit port 05 H

**I/O mode (indirect):**

The port number is taken from DX.

Example 1 : INAL, DX

If [DX] = 5040

8 bit content by port 5040 is moved into AL.

Example 2 : IN AX, DX

Inputs 8 bit content of ports 5040 and 5041 into AL and AH respectively.

**D. Relative addressing mode:**

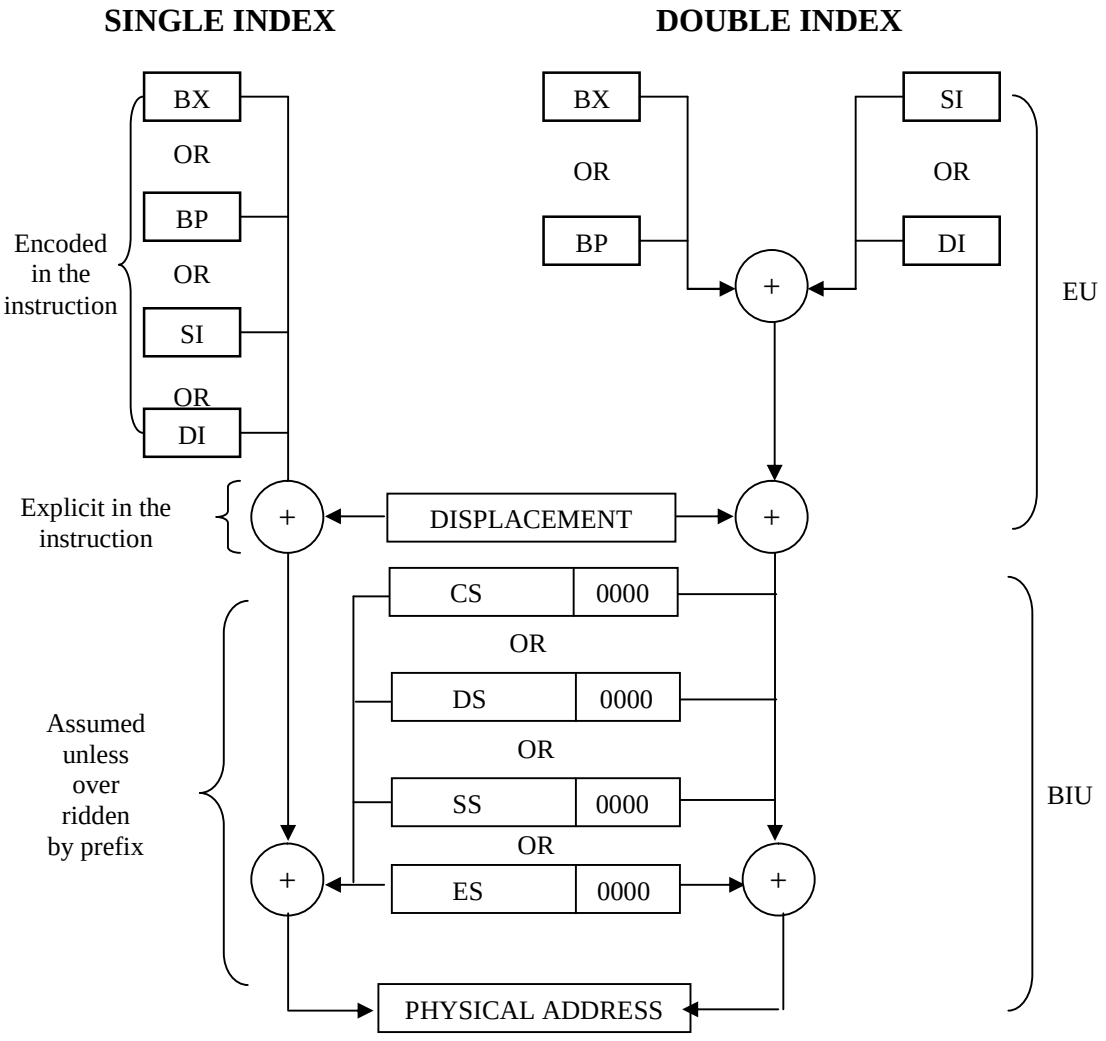
Example : JNC START

If CY=O, then PC is loaded with current PC contents plus 8 bit signed value of START, otherwise the next instruction is executed.

**E. Implied addressing mode:**

Instruction using this mode have no operands.

Example : CLC which clears carry flag to zero.



**Special functions of general-purpose registers:**

**AX & DX registers:**

In 8 bit multiplication, one of the operands must be in AL. The other operand can be a byte in memory location or in another 8 bit register. The resulting 16 bit product is stored in AX, with AH storing the MS byte.

In 16 bit multiplication, one of the operands must be in AX. The other operand can be a word in memory location or in another 16 bit register. The resulting 32 bit product is stored in DX and AX, with DX storing the MS word and AX storing the LS word.

**BX register** : In instructions where we need to specify in a general purpose register the 16 bit effective address of a memory location, the register BX is used (register indirect).

**CX register** : In Loop Instructions, CX register will be always used as the implied counter.

In I/O instructions, the 8086 receives into or sends out data from AX or AL depending as a word or byte operation. In these instructions the port address, if greater than FFH has to be given as the contents of DX register.

**Ex :** IN AL, DX

DX register will have 16 bit address of the I/P device

**Physical Address (PA) generation :**

Generally Physical Address (20 Bit) = Segment Base Address (SBA)

\+ Effective Address (EA)

**Code Segment :**

Physical Address (PA) = CS Base Address

\+ Instruction Pointer (IP)

Data Segment (DS)

PA = DS Base Address + EA can be in BX or SI or DI

Stack Segment (SS)

PA + SS Base Address + EA can be SP or BP

Extra Segment (ES)

PA = ES Base Address + EA in DI

**Instruction Format :**

The 8086 instruction sizes vary from one to six bytes. The OP code occupies six bytes and it defines the operation to be carried out by the instruction.

Register Direct bit (D) occupies one bit. It defines whether the register operand in byte 2 is the source or destination operand.

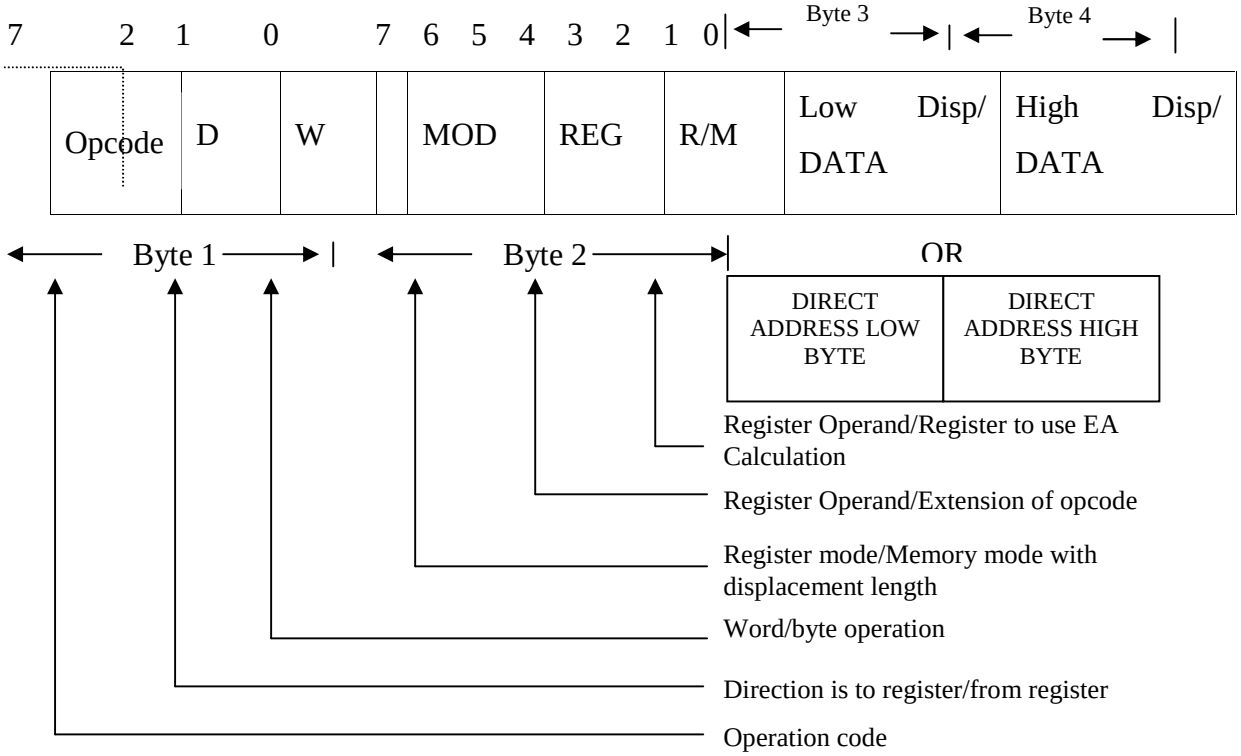
D=1 Specifies that the register operand is the destination operand.

D=0 indicates that the register is a source operand.

Data size bit (W) defines whether the operation to be performed is an 8 bit or 16 bit data

W=0 indicates 8 bit operation

W=1 indicates 16 bit operation




The second byte of the instruction usually identifies whether one of the operands is in memory or whether both are registers.

This byte contains 3 fields. These are the mode (MOD) field, the register (REG) field and the Register/Memory (R/M) field.

| **MOD (2 bits)** | **Interpretation** |
|---|---|
| 00 | Memory mode with no displacement follows except for 16 bit displacement when R/M=110 |
| 01 | Memory mode with 8 bit displacement |
| 10 | Memory mode with 16 bit displacement |

| | |
|---|---|
| 11 | Register mode (no displacement) |

Register field occupies 3 bits. It defines the register for the first operand which is specified as source or destination by the D bit.

| **REG** | **W=0** | **W=1** |
|---|---|---|
| 000 | AL | AX |
| 001 | CL | CX |
| 010 | DL | DX |
| 011 | BL | BX |
| 100 | AH | SP |
| 101 | CH | BP |
| 110 | DH | SI |
| 111 | BH | DI |

The R/M field occupies 3 bits. The R/M field along with the MOD field defines the second operand as shown below.

**MOD 11**

| **R/M** | **W=0** | **W=1** |
|---|---|---|
| 000 | AL | AX |
| 001 | CL | CX |
| 010 | DL | DX |
| 011 | BL | BX |
| 100 | AH | SP |
| 101 | CH | BP |
| 110 | DH | SI |
| 111 | BH | DI |

**Effective Address Calculation**

| **R/M** | **MOD=00** | **MOD 01** | **MOD 10** |
|---|---|---|---|
| 000 | (BX) + (SI) | (BX)+(SI)+D8 | (BX)+(SI)+D16 |
| 001 | (BX)+(DI) | (BX)+(DI)+D8 | (BX)+(DI)+D16 |
| 010 | (BP)+(SI) | (BP)+(SI)+D8 | (BP)+(SI)+D16 |
| 011 | (BP)+(DI) | (BP)+(DI)+D8 | (BP)+(DI)+D10 |
| 100 | (SI) | (SI) + D8 | (SI) + D16 |

| | | | |
|---|---|---|---|
| 101 | (DI) | (DI) + D8 | (DI) + D16 |
| 110 | Direct address | (BP) + D8 | (BP) + D16 |
| 111 | (BX) | (BX) + D8 | (BX) + D16 |

In the above, encoding of the R/M field depends on how the mode field is set. If MOD=11 (register to register mode), this R/M identifies the second register operand.

MOD selects memory mode, then R/M indicates how the effective address of the memory operand is to be calculated. Bytes 3 through 6 of an instruction are optional fields that normally contain the displacement value of a memory operand and / or the actual value of an immediate constant operand.

**Example 1** : MOV CH, BL

This instruction transfers 8 bit content of BL

**Into CH**

The 6 bit Opcode for this instruction is $100010_2$ D bit indicates whether the register specified by the REG field of byte 2 is a source or destination operand.

D=0 indicates BL is a source operand.

W=0 byte operation

In byte 2, since the second operand is a register MOD field is $11_2$.

The R/M field = 101 (CH)

Register (REG) field = 011 (BL)

Hence the machine code for MOV CH, BL is

10001000 11 011 101

Byte 1 Byte2

= 88DD16

**Example 2** : SUB Bx, (DI)

This instruction subtracts the 16 bit content of memory location addressed by DI and DS from Bx.

The 6 bit Opcode for SUB is $001010_2$.

D=1 so that REG field of byte 2 is the destination operand. W=1 indicates 16 bit operation.

MOD = 00

REG = 011

R/M = 101

The machine code is 0010 1011 0001 1101

2 B 1 D

**$2B1D_{16}$**

| MOD / R/M | Memory Mode (EA Calculation) | | | Register Mode | |
|---|---|---|---|---|---|
| | **00** | **01** | **10** | **W=0** | **W=1** |
| 000 | (BX)+(SI) | (BX)+(SI)+d8 | (BX)+(SI)+d16 | AL | AX |
| 001 | (BX) + (DI) | (BX)+(DI)+d8 | (BX)+(DI)+d16 | CL | CX |
| 010 | (BP)+(SI) | (BP)+(SI)+d8 | (BP)+(SI)+d16 | DL | DX |
| 011 | (BP)+(DI) | (BP)+(DI)+d8 | (BP)+(DI)+d16 | BL | BX |
| 100 | (SI) | (SI) + d8 | (SI) + d16 | AH | SP |
| 101 | (DI) | (DI) + d8 | (DI) + d16 | CH | BP |
| 110 | d16 | (BP) + d8 | (BP) + d16 | DH | SI |
| 111 | (BX) | (BX) + d8 | (BX) + d16 | BH | DI |

**Summary of all Addressing Modes**

**Example 3 : Code for MOV 1234 (BP), DX**

Here we have specify DX using REG field, the D bit must be 0, indicating the DX is the source register. The REG field must be 010 to indicate DX register. The W bit must be 1 to indicate it is a word operation. 1234 [BP] is specified using MOD value of 10 and R/M value of 110 and a displacement of 1234H. The 4 byte code for this instruction would be 89 96 34 12H.

| Opcode | D | W | MOD | REG | R/M | LB displacement | HB displacement |
|---|---|---|---|---|---|---|---|
| 100010 | 0 | 1 | 10 | 010 | 110 | 34H | 12H |

**Example 4 :** Code for MOV DS : 2345 [BP], DX

Here we have to specify DX using REG field. The D bit must be o, indicating that Dx is the source register. The REG field must be 010 to indicate DX register. The w bit must be 1 to indicate it is a word operation. 2345 [BP] is specified with MOD=10 and R/M = 110 and displacement = 2345 H.

Whenever BP is used to generate the Effective Address (EA), the default segment would be SS. In this example, we want the segment register to be DS, we have to provide the segment override prefix byte (SOP byte) to start with. The SOP byte is 001 SR 110, where SR value is provided as per table shown below.

| SR | Segment register |
|---|---|
| 00 | ES |
| 01 | CS |
| 10 | SS |
| 11 | DS |

To specify DS register, the SOP byte would be 001 11 110 = 3E H. Thus the 5 byte code for this instruction would be 3E 89 96 45 23 H.

| SOP | Opcode | D | W | MOD | REG | R/M | LB disp. | HD disp. |
|---|---|---|---|---|---|---|---|---|
| 3EH | 1000 10 | 0 | 1 | 10 | 010 | 110 | 45 | 23 |

Suppose we want to code MOV SS : 2345 (BP), DX. This generates only a 4 byte code, without SOP byte, as SS is already the default segment register in this case.

## UNIVERSITY QUESTIONS & SOLUTIONS

1. Explain the five types of data transfer instructions with example. [ jan 2008( 10 marks)]

Soln:

### MOV SB, MOV SW:

An element of the string specified by the source index (SI) register with respect to the current data segment (DS) register is moved to the location specified by the destination index (DI) register with respect to the current extra segment (ES) register.

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

Example : Block move program using the move string instruction

(LOD SB/LOD SW and STO SB/STO SW)

LOD SB: Loads a byte from a string in memory into AL. The address in SI is used relative to DS to determine the address of the memory location of the string element.

$(AL) \leftarrow [(DS) + (SI)]$

$(SI) \leftarrow (SI) \pm 1$

LOD SW : The word string element at the physical address derived from DS and SI is to be loaded into AX. SI is automatically incremented by 2.

$(AX) \leftarrow [(DS) + (SI)]$

(SI) ← (SI) ± 2

STO SB : Stores a byte from AL into a string location in memory. This time the contents of ES and DI are used to form the address of the storage location in memory

[(ES) + (DI)] ← (AL)

(DI) ← (DI) ± 1

**STO SW :** [(ES) + (DI)] ← (AX)

(DI) ← (DI) ± 2

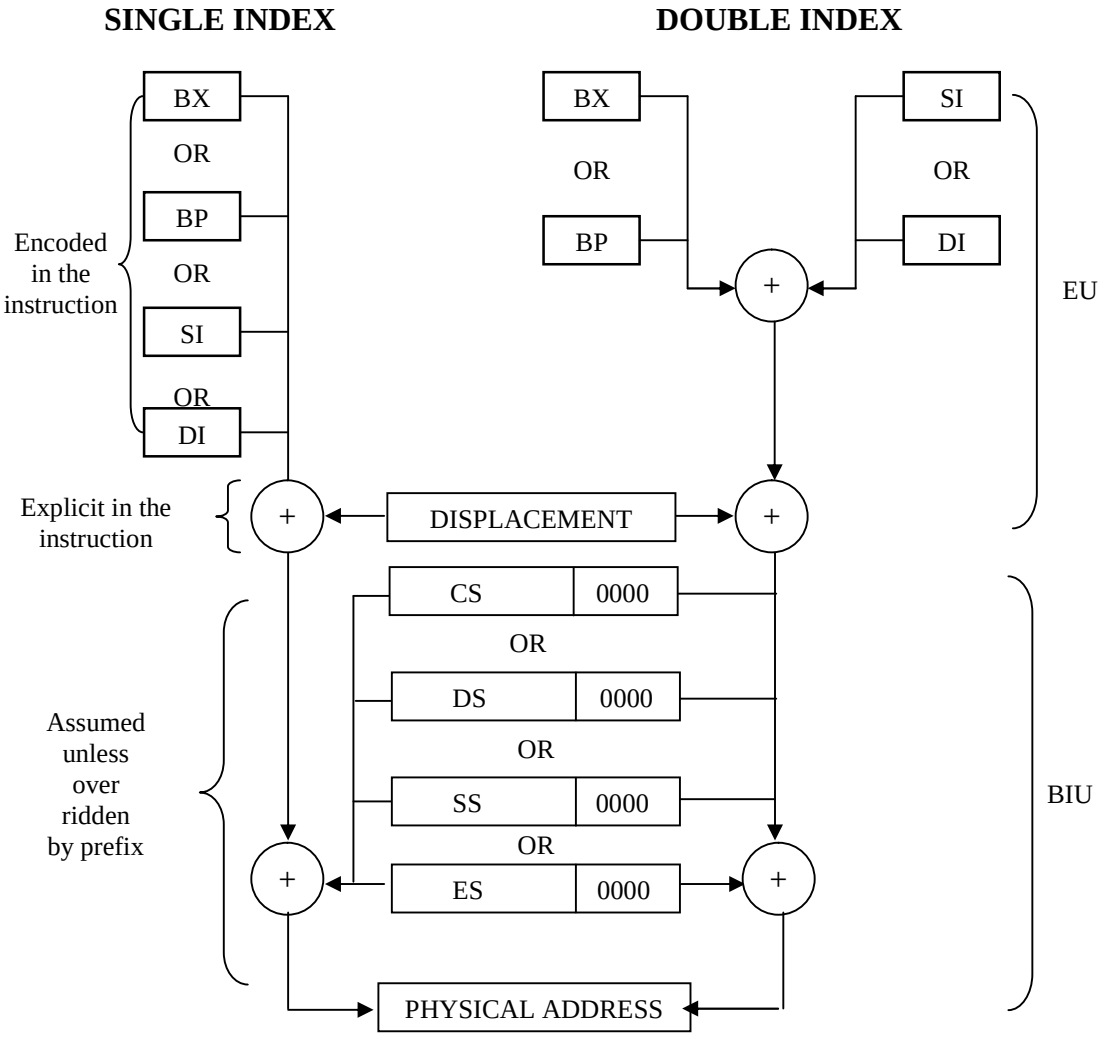
2. Explain the based and implied addressing modes of 8086 july 2009( 10 marks)

Soln:

**Implied addressing mode:**

Instruction using this mode have no operands.

Example : CLC which clears carry flag to zero.




**Based addressing mode**:

$$PA = \begin{Bmatrix} CS \\ DS \\ SS \\ ES \end{Bmatrix} : \begin{Bmatrix} BX \\ or \\ BP \end{Bmatrix} + \text{displacement}$$

when memory is accessed PA is computed from BX and DS when the stack is accessed PA is computed from BP and SS.

**Example** : MOV AL, START [BX]

or

MOV AL, [START + BX]

based mode

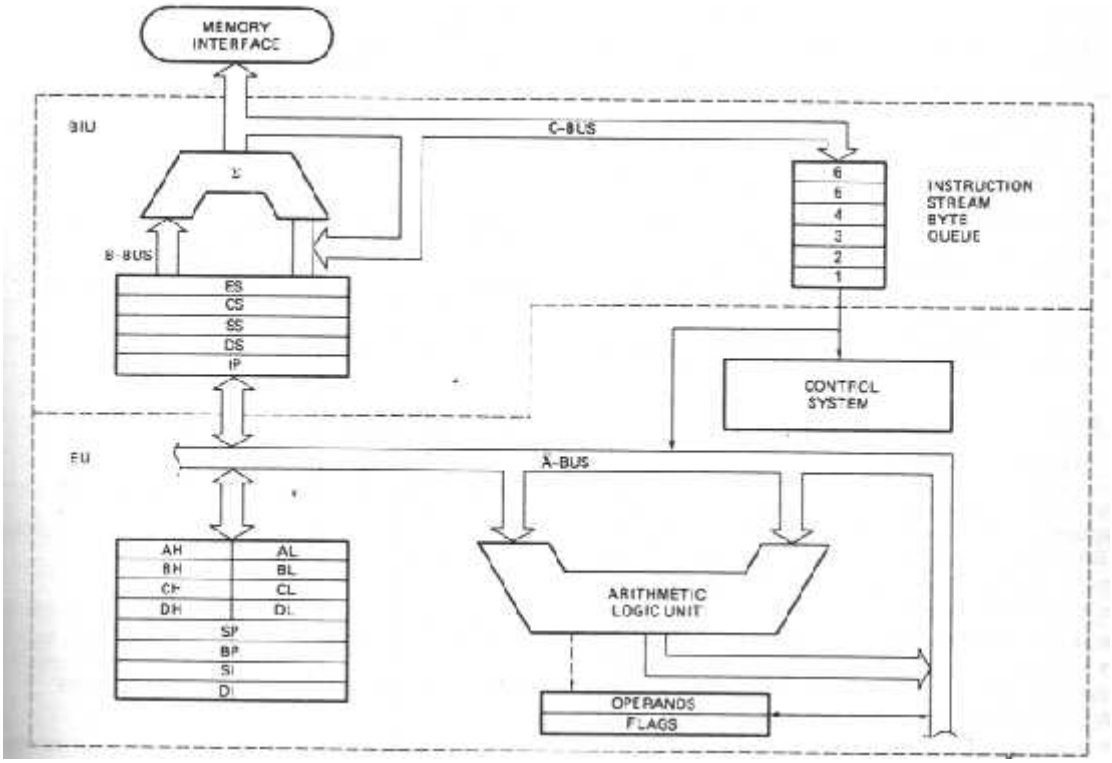
EA : [START] + [BX]

PA : [DS] + [EA]

The 8 bit content of this memory location is moved to AL.

3. Draw the internal architecture of the 8086 and explain. Briefly explain the flag register. July 2009( 10 marks)

Soln:



The block diagram of 8086 is as shown. This can be subdivided into two parts, namely the Bus Interface Unit and Execution Unit. The Bus Interface Unit consists of segment registers, adder to generate 20 bit address and instruction prefetch queue.

Once this address is sent out of BIU, the instruction and data bytes are fetched from memory and they fill a First In First Out 6 byte queue.

**Execution Unit:**

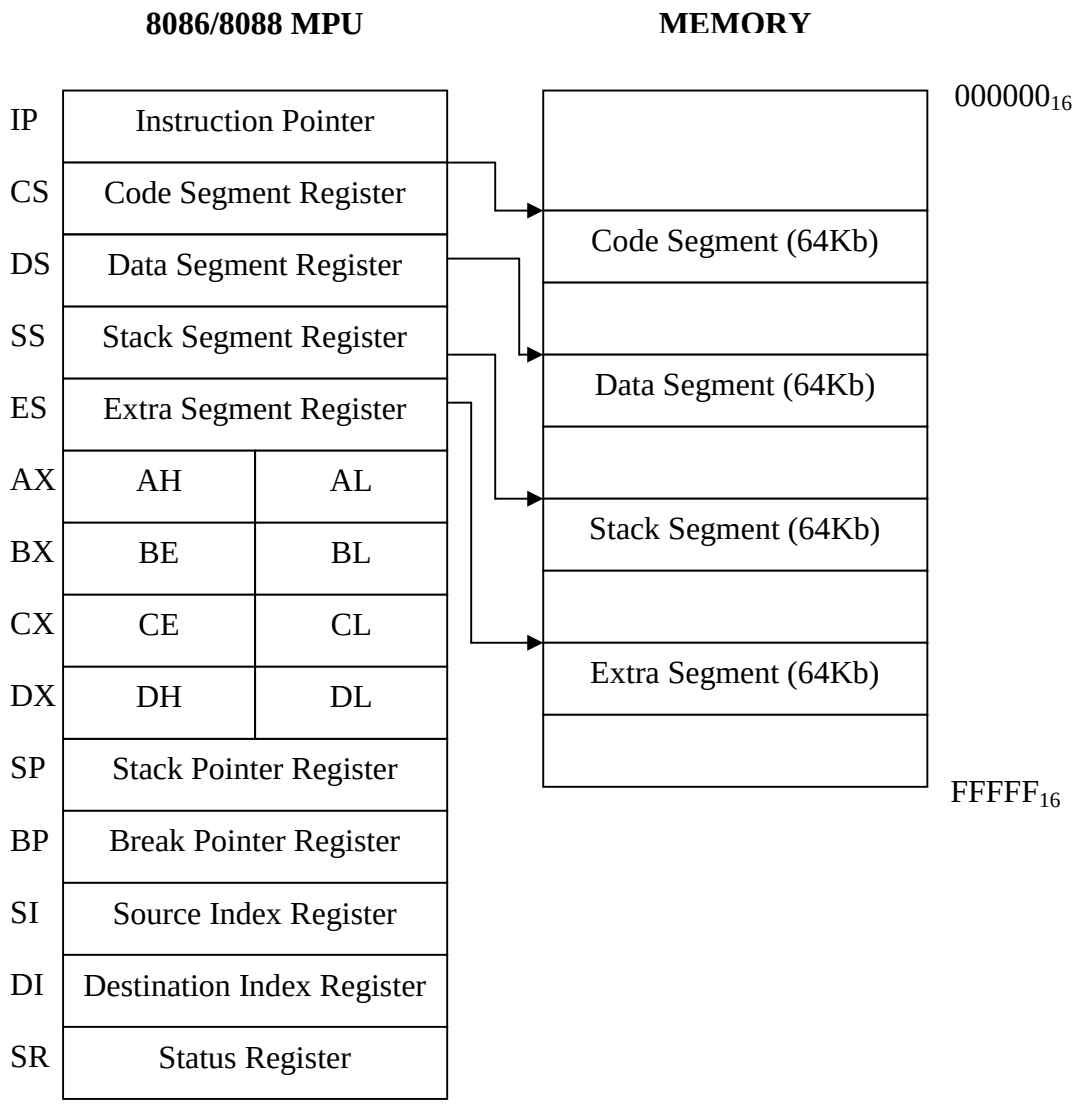
The execution unit consists of scratch pad registers such as 16-bit AX, BX, CX and DX and pointers like SP (Stack Pointer), BP (Base Pointer) and finally index registers such as source index and destination index registers. The 16-bit scratch pad registers can be split into two 8-bit registers. For example, AX can be split into AH and AL registers. The segment registers and their default offsets are given below.

| **Segment Register** | **Default Offset** |
|---|---|
| CS | IP (Instruction Pointer) |
| DS | SI, DI |
| SS | SP, BP |
| ES | DI |

The Arithmetic and Logic Unit adjacent to these registers perform all the operations. The results of these operations can affect the condition flags.

Different registers and their operations are listed below:

| **Register** | **Operations** |
|---|---|
| AX | Word multiply, Word divide, word I/O |
| AL | Byte Multiply, Byte Divide, Byte I/O, translate, Decimal Arithmetic |
| AH | Byte Multiply, Byte Divide |
| BX | Translate |
| CX | String Operations, Loops |
| CL | Variable Shift and Rotate |
| DX | Word Multiply, word Divide, Indirect I/O |

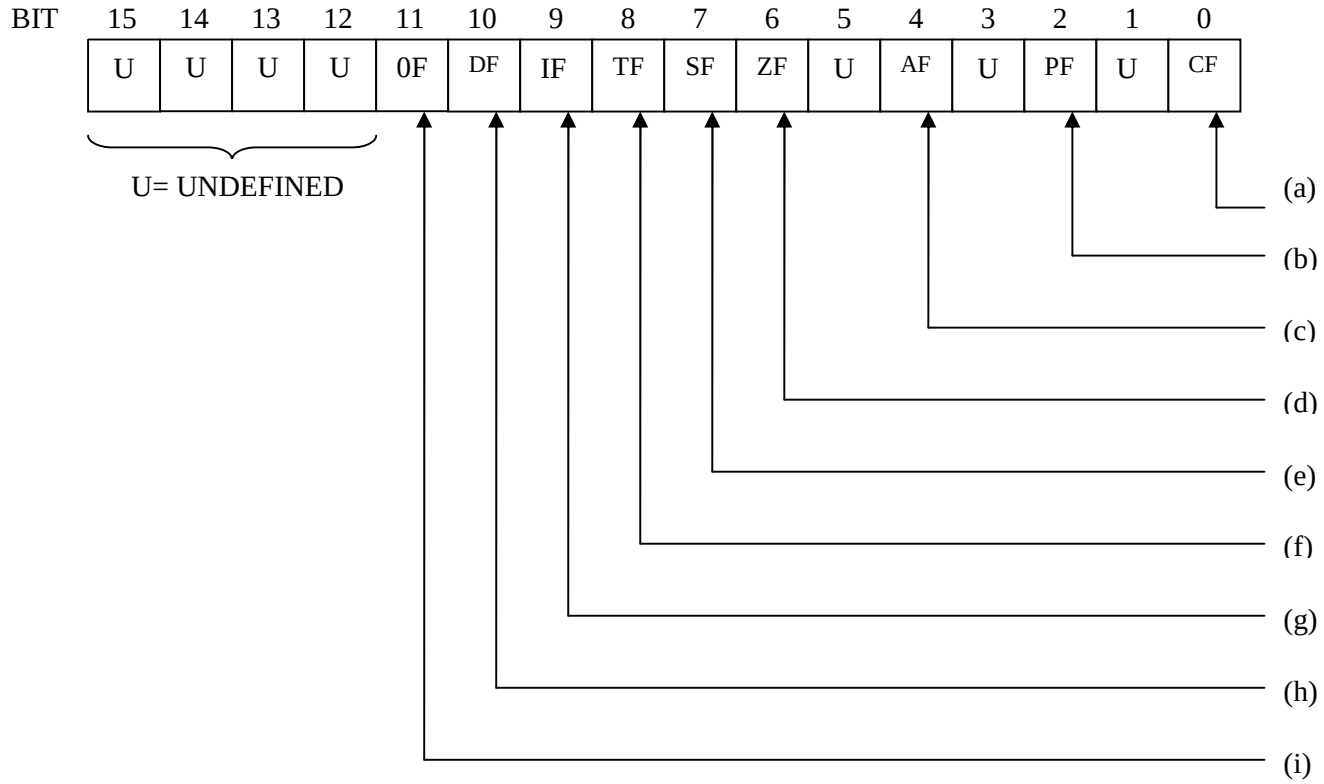



**Generation of 20-bit Physical Address:**

4. Explain flag register of 8086. July 2009 (2009)

Soln:

**8086 flag register format**




(j) : CARRY FLAG – SET BY CARRY OUT OF MSB
(k) : PARITY FLAG – SET IF RESULT HAS EVEN PARITY
(l) : AUXILIARY CARRY FLAG FOR BCD
(m): ZERO FLAG – SET IF RESULT = 0
(n) : SIGN FLAG = MSB OF RESULT
(o) : SINGLE STEP TRAP FLAG
(p) : INTERRUPT ENABLE FLAG
(q) : STRING DIRECTION FLAG
(r) : OVERFLOW FLAG

There are three internal buses, namely A bus, B bus and C bus, which interconnect the various blocks inside 8086.

The execution of instruction in 8086 is as follows:

The microprocessor unit (MPU) sends out a 20-bit physical address to the memory and fetches the first instruction of a program from the memory. Subsequent addresses are sent out and the queue is filled upto 6 bytes. The instructions are decoded and further data (if necessary) are fetched from memory. After the execution of the instruction, the results may go back to memory or to the output peripheral devices as the case may be.

Machine language:

## Recommended questions

1. Describe the physical addresses generated in 8086.
2. Explain the five types of string instructions with example.
3. Explain the direct and register addressing modes of 8086.
4. Draw the internal architecture of the 8086 and explain. Briefly explain the flag register.
5. Explain flag register of 8086.
6. Explain the instruction format of 8086.
7. What are addressing modes? Mention and Explain the various types of addressing modes.

## UNIT: 2

**INSTRUCTION SET OF 8086:** Assembler instruction format, data transfer and arithmetic, branch type, loop, NOP & HALT, flag manipulation, logical and shift and rotate instructions. Illustration of these instructions with example programs, Directives and operators

**TEXT BOOKS:**

1. **Microcomputer systems-The 8086 / 8088** Family – Y.C. Liu and G. A. Gibson, 2E PHI - 2003
2. **The Intel Microprocessor, Architecture, Programming and Interfacing**-Barry B. Brey, 6e, Pearson Education / PHI, 2003

# Instruction Set

We only cover the *small* subset of the 8088 instruction set that is essential. In particular, we will not mention various registers, addressing modes and instructions that could often provide faster ways of doing things.A summary of the 80x86 protected-mode instruction set is available on the course Web page and should be printed out if you don't have another reference.

# Data Transfer

The MOV instruction is used to transfer 8 and 16-bit data to and from registers. Either the source or destination has to be a register. The other operand can come from another register, from memory, from immediate data (a value included in the instruction) or from a memory location "pointed at" by register BX.

For example, if COUNT is the label of a memory location the following are possible **assembly-language instructions : ;**

register: move contents of BX to AX

MOV AX,BX ; direct: move contents of AX to memory

MOV COUNT,AX ; immediate: load CX with the value 240

MOV CX,0F0H; memory: load CX with the value at

; address 240

MOV CX,[0F0H]; register indirect: move contents of AL

; to memory location in BX

MOV [BX],AL

Most 80x86 assemblers keep track of the type of each symbol and require a type "override" when thesymbol is used in a different way. The OFFSET operator to convert a memory reference to a 16-bit value.

**For example:**

MOV BX,COUNT ; load the value at location COUNT

MOV BX,OFFSET COUNT ; load the offset of COUNT

16-bit registers can be pushed (the SP is first decremented by two and then the value stored at SP) or popped (the value is restored from the memory at SP and then SP is incremented by 2). For example:

PUSH AX ; push contents of AX

POP BX ; restore into BX

**Arithmetic instruction:**

**Arithmetic/Logic**

Arithmetic and logic instructions can be performed on byte and 16-bit values. The first operand has to be a register and the result is stored in that register.

; increment BX by 4

ADD BX,4

; subtract 1 from AL

SUB AL,1

; increment BX

INC BX

; compare (subtract and set flags

; but without storing result)

CMP AX,[MAX]

; mask in LS 4 bits of AL

AND AL,0FH

; divide AX by two

SHR AX

; set MS bit of CX

OR CX,8000H

; clear AX

XOR AX,AX

**The LOOP Instruction**

This instruction decrements the cx register and then branches to the target location if the cx register does not contain zero. Since this instruction decrements cx then checks for zero, if cx

originally contained zero, any loop you create using the loop instruction will repeat 65,536 times. If you do not want to execute the loop when cx contains zero, use jcxz to skip over the loop. There is no "opposite" form of the loop instruction, and like the jcxz/jecxz instructions the range is limited to ±128 bytes on all processors. If you want to extend the range of this instruction, you will need to break it down into discrete components:

; "loop lbl" becomes:

```
dec cx
jne lbl
```

There is no eloop instruction that decrements ecx and branches if not zero (there is a loope instruction, but it does something else entirely). The reason is quite simple. As of the 80386, Intel's designers stopped wholeheartedly supporting the loop instruction. Oh, it's there to ensure compatibility with older code, but it turns out that the dec/jne instructions are actually *faster* on the 32 bit processors. Problems in the decoding of the instruction and the operation of the pipeline are responsible for this strange turn of events. Although the loop instruction's name suggests that you would normally create loops with it, keep in mind that all it is really doing is decrementing cx and branching to the target address if cx does not contain zero after the decrement. You can use this instruction anywhere you want to decrement cx and then check for a zero result, not just when creating loops. Nonetheless, it is a very convenient instruction to use if you simply want to repeat a sequence of instructions some number of times. For example, the following loop initializes a 256 element array of bytes to the values 1, 2, 3, ...

```
mov ecx, 255
ArrayLp: mov Array[ecx], cl
loop ArrayLp
mov Array[0], 0
```

The last instruction is necessary because the loop does not repeat when cx is zero. Therefore, the last element of the array that this loop processes is Array[1], hence the last instruction.

The loop instruction does not affect any flags.

**The LOOPE/LOOPZ Instruction**

Loope/loopz (loop while equal/zero, they are synonyms for one another) will branch to the target address if cx is not zero and the zero flag is set. This instruction is quite useful

The 80x86 Instruction Set after cmp or cmps instruction, and is marginally faster than the comparable 80386/486 instructions *if you use all the features of this instruction*. However, this instruction plays havoc with the pipeline and superscalar operation of the Pentium so you're

probably better off sticking with discrete instructions rather than using this instruction. This instruction does the following:

cx := cx - 1

if ZeroFlag = 1 and cx ¹ 0, goto target The loope instruction falls through on one of two conditions. Either the zero flag is clear or the instruction decremented cx to zero. By testing the zero flag after the loop instruction (with a je or jne instruction, for example), you can determine the cause of termination. This instruction is useful if you need to repeat a loop while some value is equal to another, but there is a maximum number of iterations you want to allow. For example, the following loop scans through an array looking for the first non-zero byte, but it does not scan beyond the end of the array:

```
mov cx, 16 ;Max 16 array elements.
mov bx, -1 ;Index into the array (note next inc).
SearchLp: inc bx ;Move on to next array element.
cmp Array[bx], 0 ;See if this element is zero.
loope SearchLp ;Repeat if it is.
je AllZero ;Jump if all elements were zero.
```

Note that this instruction is not the opposite of loopnz/loopne. If you need to extend this jump beyond ±128 bytes, you will need to synthesize this instruction using discrete instructions. For example, if loope target is out of range, you would need to use an instruction sequence like the following:

```
jne quit
dec cx
je Quit2
jmp Target
quit: dec cx ;loope decrements cx, even if ZF=0.
quit2:
```

The loope/loopz instruction does not affect any flags.

**The LOOPNE/LOOPNZ Instruction**

This instruction is just like the loope/loopz instruction in the previous section except loopne/loopnz (loop while not equal/not zero) repeats while cx is not zero and the zero flag is clear. The algorithm is

cx := cx - 1

if ZeroFlag = 0 and cx ¹ 0, goto target

You can determine if the loopne instruction terminated because cx was zero or if the zero flag was set by testing the zero flag immediately after the loopne instruction. If the zero flag is clear at that point, the loopne instruction fell through because it decremented cx to zero. Otherwise it fell through because the zero flag was set.

This instruction is *not* the opposite of loope/loopz. If the target address is out of range,

you will need to use an instruction sequence like the following:

```
je quit
dec cx
je Quit2
jmp Target
quit: dec cx ;loopne decrements cx, even if ZF=1.
quit2:
```

You can use the loopne instruction to repeat some maximum number of times while waiting for some other condition to be true. For example, you could scan through an array until you exhaust the number of array elements or until you find a certain byte using a loop like the following:

```
mov cx, 16 ;Maximum # of array elements.
mov bx, -1 ;Index into array.
LoopWhlNot0: inc bx ;Move on to next array element.
cmp Array[bx],0 ;Does this element contain zero?
loopne LoopWhlNot0 ;Quit if it does, or more than 16 bytes.
```

Although the loope/loopz and loopne/loopnz instructions are slower than the individual instruction from which they could be synthesized, there is one main use for these instruction forms where speed is rarely important; indeed, being faster would make them less useful – timeout loops during I/O operations. Suppose bit #7 of input port 379h contains a one if the device is busy and contains a zero if the device is not busy. If you want to output

data to the port, you *could* use code like the following:

```
mov dx, 379h
WaitNotBusy: in al, dx ;Get port
 test al, 80h ;See if bit #7 is one
 jne WaitNotBusy ;Wait for “not busy”
```

The only problem with this loop is that it is conceivable that it would loop forever. In a real system, a cable could come unplugged, someone could shut off the peripheral device, and any

number of other things could go wrong that would hang up the system.Robust programs usually apply a *timeout* to a loop like this. If the device fails to become busy within some specified amount of time, then the loop exits and raises an error condition. The following code will accomplish this:

```
mov dx, 379h ;Input port address
mov cx, 0 ;Loop 65,536 times and then quit.
WaitNotBusy: in al, dx ;Get data at port.
test al, 80h ;See if busy
loopne WaitNotBusy ;Repeat if busy and no time out.
jne TimedOut ;Branch if CX=0 because we timed out.
```

You could use the loope/loopz instruction if the bit were zero rather than one.

The loopne/loopnz instruction does not affect any flags.

**Logical, Shift, Rotate and Bit Instructions**

The 80x86 family provides five logical instructions, four rotate instructions, and three shift instructions. The logical instructions are and, or, xor, test, and not; the rotates are ror, rol, rcr, and rcl; the shift instructions are shl/sal, shr, and sar. The 80386 and later processors provide an even richer set of operations. These are bt, bts, btr, btc, bsf, bsr, shld, shrd, and theconditional set instructions (setcc). These instructions can manipulate bits, convert values, do logical operations, pack and unpack data, and do arithmetic operations. The following sections describe each of these instructions in detail.

**The Logical Instructions: AND, OR, XOR, and NOT**

The 80x86 logical instructions operate on a bit-by-bit basis. Both eight, sixteen, and thirty-two bit versions of each instruction exist. The and, not, or, and xor instructions do the following:

```
and dest, source ;dest := dest and source
or dest, source ;dest := dest or source
xor dest, source ;dest := dest xor source
not dest ;dest := not dest
```

The specific variations are

```
and reg, reg
and mem, reg
and reg, mem
and reg, immediate data
and mem, immediate data
```

and eax/ax/al, immediate data

or uses the same formats as AND

xor uses the same formats as AND

not register

not mem

Except not, these instructions affect the flags as follows:

• They clear the carry flag.

• They clear the overflow flag.

• They set the zero flag if the result is zero, they clear it otherwise.

• They copy the H.O. bit of the result into the sign flag.

• They set the parity flag according to the parity (number of one bits) in the result.

• They scramble the auxiliary carry flag. The not instruction does not affect any flags. Testing the zero flag after these instructions is particularly useful. The and instruction sets the zero flag if the two operands do not have any ones in corresponding bit positions (since this would produce a zero result); for example, if the source operand contained a single one bit, then the zero flag will be set if the corresponding destination bit is zero, itwill be one otherwise. The or instruction will only set the zero flag if both operands contain zero. The xor instruction will set the zero flag only if both operands are equal. Noticethat the xor operation will produce a zero result if and only if the two operands are equal. Many programmers commonly use this fact to clear a sixteen bit register to zero since an instruction of the form

Xor reg16, reg16 is shorter than the comparable mov reg,0 instruction. Like the addition and subtraction instructions, the and, or, and xor instructions provide special forms involving the accumulator register and immediate data. These forms are shorter and sometimes faster than the general “register, immediate” forms. Although one does not normally think of operating on signed data with these instructions, the 80x86 does provide a special form of the “reg/mem, immediate” instructions that sign extend a value in the range -128..+127 to sixteen or thirty-two bits, as necessary.The instruction’s operands must all be the same size. On pre-80386 processors theycan be eight or sixteen bits. On 80386 and later processors, they may be 32 bits long as well. These instructions compute the obvious bitwise logical operation on their operands, You can use the and instruction to set selected bits to zero in the destination operand. This is known as *masking out* data; see for more details. Likewise, you can use the or instruction to force certain bits to one in the destination operand; see “Masking Operations

with the OR Instruction” on page 491 for the details. You can use these instructions,

along with the shift and rotate instructions described next, to pack and unpack data.

**The Shift Instructions: SHL/SAL, SHR, SAR, SHLD, and SHRD**

The 80x86 supports three different shift instructions (shl and sal are the same instruction): shl (shift left), sal (shift arithmetic left), shr (shift right), and sar (shift arithmetic right). The 80386 and later processors provide two additional shifts: shld and shrd. The shift instructions move bits around in a register or memory location. The general format for a shift instruction is

shl dest, count

sal dest, count

shr dest, count

sar dest, count

Dest is the value to shift and count specifies the number of bit positions to shift. For example, the shl instruction shifts the bits in the destination operand to the left the number of bit positions specified by the count operand. The shld and shrd instructions use the format:

shld dest, source, count

shrd dest, source, count

The specific forms for these instructions are

shl reg, 1

shl mem, 1

shl reg, imm (2)

shl mem, imm (2)

shl reg, cl

shl mem, cl

sal is a synonym for shl and uses the same formats.

shr uses the same formats as shl.

sar uses the same formats as shl.

shld reg, reg, imm (3)

shld mem, reg, imm (3)

shld reg, reg, cl (3)

shld mem, reg, cl (3)

shrd uses the same formats as shld.

2- This form is available on 80286 and later processors only.

3- This form is available on 80386 and later processors only.

For 8088 and 8086 CPUs, the number of bits to shift is either “1” or the value in cl. On 80286 and later processors you can use an eight bit immediate constant. Of course, the value in cl or the immediate constant should be less than or equal to the number of bits in the destination operand. It would be a waste of time to shift left al by nine bits (eight would produce the same result, as you will soon see). Algorithmically, you can think of the shift operations with a count other than one as follows:

```
for temp := 1 to count do
shift dest, 1
```

There are minor differences in the way the shift instructions treat the overflow flag when the count is not one, but you can ignore this most of the time. The shl, sal, shr, and sar instructions work on eight, sixteen, and thirty-two bit operands. The shld and shrd instructions work on 16 and 32 bit destination operands only.

**SHL/SAL**

The shl and sal mnemonics are synonyms. They represent the same instruction and use identical binary encodings. These instructions move each bit in the destination operand one bit position to the left the number of times specified by the count operand. Zeros fill vacated positions at the L.O. bit; the H.O. bit shifts into the carry flag (see Figure 6.2).

The shl/sal instruction sets the condition code bits as follows:

• If the shift count is zero, the shl instruction doesn’t affect any flags.

• The carry flag contains the last bit shifted out of the H.O. bit of the operand.

• The overflow flag will contain one if the two H.O. bits were different prior to a single bit shift. The overflow flag is undefined if the shift count is not one.

• The zero flag will be one if the shift produces a zero result.

• The sign flag will contain the H.O. bit of the result.

• The parity flag will contain one if there are an even number of one bits in the L.O. byte of the result.

• The A flag is always undefined after the shl/sal instruction.

The shift left instruction is especially useful for packing data. For example, suppose you have two nibbles in al and ah that you want to combine. You could use the following code to do this:

```
shl ah, 4 ;This form requires an 80286 or later
or al, ah ;Merge in H.O. four bits.
```

Of course, al must contain a value in the range 0..F for this code to work properly (the shift left operation automatically clears the L.O. four bits of ah before the or instruction). If the H.O. four bits of al are not zero before this operation, you can easily clear them with an and instruction:

```
shl ah, 4 ;Move L.O. bits to H.O. position.
and al, 0Fh ;Clear H.O. four bits.
or al, ah ;Merge the bits.
```

Since shifting an integer value to the left one position is equivalent to multiplying that value by two, you can also use the shift left instruction for multiplication by powers of two:

```
shl ax, 1 ;Equivalent to AX*2
shl ax, 2 ;Equivalent to AX*4
shl ax, 3 ;Equivalent to AX*8
shl ax, 4 ;Equivalent to AX*16
shl ax, 5 ;Equivlaent to AX*32
shl ax, 6 ;Equivalent to AX*64
shl ax, 7 ;Equivalent to AX*128
shl ax, 8 ;Equivalent to AX*256
etc.
```

Note that shl ax, 8 is equivalent to the following two instructions:

```
mov ah, al
mov al, 0
```

The shl/sal instruction multiplies both signed and unsigned values by two for each shift. This instruction sets the carry flag if the result does not fit in the destination operand (i.e., unsigned overflow occurs). Likewise, this instruction sets the overflow flag if the signed result does not fit in the destination operation. This occurs when you shift a zero into the H.O. bit of a negative number or you shift a one into the H.O. bit of a non-negative number.

**SAR**

The sar instruction shifts all the bits in the destination operand to the right one bit, replicating the H.O. bit (see Figure 6.3). The sar instruction sets the flag bits as follows:

• If the shift count is zero, the sar instruction doesn’t affect any flags.

• The carry flag contains the last bit shifted out of the L.O. bit of the operand.

• The overflow flag will contain zero if the shift count is one. Overflow can never occur with this instruction. However, if the count is not one, the value of the overflow flag is undefined.

• The zero flag will be one if the shift produces a zero result.

• The sign flag will contain the H.O. bit of the result.

• The parity flag will contain one if there are an even number of one bits in the L.O. byte of the result.

• The auxiliary carry flag is always undefined after the sar instruction.

The sar instruction's main purpose is to perform a signed division by some power of two. Each shift to the right divides the value by two. Multiple right shifts divide the previous shifted result by two, so multiple shifts produce the following results:

```
sar ax, 1 ;Signed division by 2
sar ax, 2 ;Signed division by 4
sar ax, 3 ;Signed division by 8
sar ax, 4 ;Signed division by 16
sar ax, 5 ;Signed division by 32
sar ax, 6 ;Signed division by 64
sar ax, 7 ;Signed division by 128
sar ax, 8 ;Signed division by 256
```

There is a very important difference between the sar and idiv instructions. The idiv instruction always truncates towards zero while sar truncates results toward the smaller result. For positive results, an arithmetic shift right by one position produces the same result as an integer division by two. However, if the quotient is negative, idiv truncates towards zero while sar truncates towards negative infinity. The following examples demonstrate the difference:

```
mov ax, -15
cwd
mov bx, 2
idiv ;Produces -7
mov ax, -15
sar ax, 1 ;Produces -8
```

Keep this in mind if you use sar for integer division operations.

The sar ax, 8 instruction effectively copies ah into al and then sign extends al into ax. This is because sar ax, 8 will shift ah down into al but leave a copy of ah's H.O. bit in all the bit positions of ah. Indeed, you can use the sar instruction on 80286 and later processors to sign extend one register into another. The following code sequences provide examples of this usage:

```
; Equivalent to CBW:
mov ah, al
sar ah, 7
```

```
; Equivalent to CWD:
mov dx, ax
sar dx, 15
; Equivalent to CDQ:
mov edx, eax
sar edx, 31
```

it may seem silly to use two instructions where a single instruction might suffice; however, the cbw, cwd, and cdq instructions only sign extend al into ax, ax into dx:ax, and eax into edx:eax. Likewise, the movsx instruction copies its sign extended operand into a destination operand twice the size of the source operand. The sar instruction lets you sign extend one register into another register of the same size, with the second register containing the sign extension bits:

```
; Sign extend bx into cx:bx
mov cx, bx
sar cx, 15
```

**SHR**

The shr instruction shifts all the bits in the destination operand to the right one bit shifting a zero into the H.O. bit (see Figure 6.4).

The shr instruction sets the flag bits as follows:

• If the shift count is zero, the shr instruction doesn't affect any flags.

• The carry flag contains the last bit shifted out of the L.O. bit of the operand.

• If the shift count is one, the overflow flag will contain the value of the H.O. bit of the operand prior to the shift (i.e., this instruction sets the overflow flag if the sign changes). However, if the count is not one, the value of the overflow flag is undefined.

• The zero flag will be one if the shift produces a zero result.

• The sign flag will contain the H.O. bit of the result, which is always zero.

• The parity flag will contain one if there are an even number of one bits in the L.O. byte of the result.

• The auxiliary carry flag is always undefined after the shr instruction.

The shift right instruction is especially useful for unpacking data. For example, suppose you want to extract the two nibbles in the al register, leaving the H.O. nibble in ah and the L.O. nibble in al. You could use the following code to do this:

```
mov ah, al ;Get a copy of the H.O. nibble
shr ah, 4 ;Move H.O. to L.O. and clear H.O. nibble
```

and al, 0Fh ;Remove H.O. nibble from al

Since shifting an unsigned integer value to the right one position is equivalent to dividing that value by two, you can also use the shift right instruction for division by powers of two:

shr ax, 1 ;Equivalent to AX/2

shr ax, 2 ;Equivalent to AX/4

shr ax, 3 ;Equivalent to AX/8

shr ax, 4 ;Equivalent to AX/16

shr ax, 5 ;Equivlaent to AX/32

shr ax, 6 ;Equivalent to AX/64

shr ax, 7 ;Equivalent to AX/128

shr ax, 8 ;Equivalent to AX/256

etc.

Note that shr ax, 8 is equivalent to the following two instructions:

mov al, ah

mov ah, 0

Remember that division by two using shr only works for *unsigned* operands. If ax contains -1 and you execute shr ax, 1 the result in ax will be 32767 (7FFFh), not -1 or zero as you would expect. Use the sar instruction if you need to divide a signed integer by some power of two.

**The SHLD and SHRD Instructions**

The shld and shrd instructions provide double precision shift left and right operations, respectively. These instructions are available only on 80386 and later processors. Their generic forms are

shld operand1, operand2, immediate

shld operand1, operand2, cl

shrd operand1, operand2, immediate

shrd operand1, operand2, cl

Operand2 must be a sixteen or thirty-two bit register. Operand1 can be a register or a memory location. Both operands must be the same size. The immediate operand can be a value in the range zero through n-1, where n is the number of bits in the two operands; it specifies the number of bits to shift. The shld instruction shifts bits in operand1 to the left. The H.O. bit shifts into the carry flag and the H.O. bit of operand2 shifts into the L.O. bit of perand1. Note that this instruction does not modify the value of operand2, it uses a temporary copy of operand2 during the shift. The immediate operand specifies the number of bits to shift. If the count is *n*, then shld shifts bit *n*-1

into the carry flag. It also shifts the H.O. *n* bits of operand2 into the L.O. *n* bits of operand1. Pictorially, the shld instruction appears in Figure 6.5.The shld instruction sets the flag bits as follows:

• If the shift count is zero, the shld instruction doesn't affect any flags.

• The carry flag contains the last bit shifted out of the H.O. bit of the operand1.

• If the shift count is one, the overflow flag will contain one if the sign bit of operand1 changes during the shift. If the count is not one, the overflow flag is undefined.

• The zero flag will be one if the shift produces a zero result.

• The sign flag will contain the H.O. bit of the result.

The shld instruction is useful for packing data from many different sources. For example, suppose you want to create a word by merging the H.O. nibbles of four other words. You could do this with the following code:

```
mov ax, Value4 ;Get H.O. nibble
shld bx, ax, 4 ;Copy H.O. bits of AX to BX.
mov ax, Value3 ;Get nibble #2.
shld bx, ax, 4 ;Merge into bx.
mov ax, Value2 ;Get nibble #1.
shld bx, ax, 4 ;Merge into bx.
mov ax, Value1 ;Get L.O. nibble
shld bx, ax, 4 ;BX now contains all four nibbles.
```

The shrd instruction is similar to shld except, of course, it shifts its bits right rather than left.

Double Precision Shift Right Operation

The shrd instruction sets the flag bits as follows:

• If the shift count is zero, the shrd instruction doesn't affect any flags.

• The carry flag contains the last bit shifted out of the L.O. bit of the operand1.

• If the shift count is one, the overflow flag will contain one if the H.O. bit of operand1 changes. If the count is not one, the overflow flag is undefined.

• The zero flag will be one if the shift produces a zero result.

• The sign flag will contain the H.O. bit of the result.

Quite frankly, these two instructions would probably be slightly more useful if

Operand2 could be a memory location. Intel designed these instructions to allow fast multiprecision
(64 bits, or more) shifts. For more information on such usage, see “Extended
Precision Shift Operations” on page 482.

The shrd instruction is marginally more useful than shld for packing data. For example, suppose that ax contains a value in the range 0..99 representing a year (1900..1999), bx contains a value in the range 1..31 representing a day, and cx contains a value in the range 1..12 representing a month (see “Bit Fields and Packed Data” on page 28). You can easily use the shrd instruction to pack this data into dx as follows:

shrd dx, ax, 7
shrd dx, bx, 5
shrd dx, cx, 4

**The Rotate Instructions: RCL, RCR, ROL, and ROR**

The rotate instructions shift the bits around, just like the shift instructions, except the bits shifted out of the operand by the rotate instructions recirculate through the operand. They include rcl (rotate through carry left), rcr (rotate through carry right), rol (rotate left), and ror (rotate right). These instructions all take the forms:

Figure 6.7 Packing Data with an SHRD Instruction

15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
**Y Y Y Y Y Y Y**
After SHRD DX, AX, 7 Instruction
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
**Y Y Y Y Y Y Y**
After SHRD DX, BX, 5 Instruction
15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0
**M M M M Y Y Y Y Y Y Y**
After SHRD DX, CX, 4 Instruction

rcl dest, count
rol dest, count
rcr dest, count
ror dest, count

The specific forms are

rcl reg, 1
rcl mem, 1

rcl reg, imm (2)

rcl mem, imm (2)

rcl reg, cl

rcl mem, cl

rol uses the same formats as rcl.

rcr uses the same formats as rcl.

ror uses the same formats as rcl.

2- This form is avialable on 80286 and later processors only.

**RCL**

The rcl (rotate through carry left), as its name implies, rotates bits to the left, through the carry flag, and back into bit zero on the right (see Figure 6.8).

Note that if you rotate through carry an object *n+1* times, where *n* is the number of bits in the object, you wind up with your original value. Keep in mind, however, that some flags may contain different values after *n+1* rcl operations.

The rcl instruction sets the flag bits as follows:

• The carry flag contains the last bit shifted out of the H.O. bit of the operand.

• If the shift count is one, rcl sets the overflow flag if the sign changes as a result of the rotate. If the count is not one, the overflow flag is undefined.

• The rcl instruction does not modify the zero, sign, parity, or auxiliary carry flags.

**Important warning:** unlike the shift instructions, the rotate instructions do not affect the sign, zero, parity, or auxiliary carry flags. This lack of orthogonality can cause you lots of grief if you forget it and attempt to test these flags after an rcl operation. If you need to test one of these flags after an rcl operation, test the carry and overflow flags first (if necessary) then compare the result to zero to set the other flags.

**RCR**

The rcr (rotate through carry right) instruction is the complement to the rcl instruction. It shifts its bits right through the carry flag and back into the H.O. bit (see Figure 6.9).This instruction sets the flags in a manner analogous to rcl:

• The carry flag contains the last bit shifted out of the L.O. bit of the operand.

• If the shift count is one, then rcr sets the overflow flag if the sign changes (meaning the values of the H.O. bit and carry flag were not the same before the execution of the instruction). However, if the count is not one, the value of the overflow flag is undefined.

• The rcr instruction does not affect the zero, sign, parity, or auxiliary carry flags.

**ROL**

The rol instruction is similar to the rcl instruction in that it rotates its operand to the left the specified number of bits. The major difference is that rol shifts its operand's H.O. bit,rather than the carry, into bit zero. Rol also copies the output of the H.O. bit into the carry flag (see Figure 6.10).The rol instruction sets the flags identically to rcl. Other than the source of the value shifted into bit zero, this instruction behaves exactly like the rcl instruction Like shl, the rol instruction is often useful for packing and unpacking data. For example,

suppose you want to extract bits 10..14 in ax and leave these bits in bits 0..4. The following code sequences will both accomplish this:

```
shr ax, 10
and ax, 1Fh
rol ax, 6
and ax, 1Fh
```

**ROR**

The ror instruction relates to the rcr instruction in much the same way that the rol instruction relates to rcl. That is, it is almost the same operation other than the source of the input bit to the operand. Rather than shifting the previous carry flag into the H.O. bit of the destination operation, ror shifts bit zero into the H.O. bit (see Figure 6.11).

***Segment Over Ride Prefix***

SOP is used when a particular offset register is not used with its default base segment register, but with a different base register. This is a byte put before the OPCODE byte.

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | S | R | 1 | 1 | 0 |

| **SR** | **Segment Register** |
|---|---|
| 00 | ES |
| 01 | CS |
| 10 | SS |
| 11 | DS |

Here SR is the new base register. To use DS as the new register 3EH should be prefix.

| Operand Register | Default | With over ride prefix |
|---|---|---|
| IP (Code address) | CS | Never |
| SP(Stack address) | SS | Never |
| BP(Stack Address) | SS | BP+DS or ES or CS |
| SI or DI(not including Strings) | DS | ES, SS or CS |
| SI (Implicit source Address for strings) | DS | ” |
| DI (Implicit Destination Address for strings) | ES | Never |

Examples: MOV AX, DS: [BP], LODS ES: DATA1

| $S_4$ | $S_3$ | Indications |
|---|---|---|
| 0 | 0 | Alternate data |
| 0 | 1 | Stack |
| 1 | 0 | Code or none |
| 1 | 1 | Data |

**Bus High Enable / Status**

| $\overline{BHE}$ | $A_0$ | Indications |
|---|---|---|
| 0 | 0 | Whole word |
| 0 | 1 | Upper byte from or to odd address |
| 1 | 0 | Lower byte from or to even address |
| 1 | 1 | none |

**Segmentation:**

The 8086 microprocessor has 20 bit address pins. These are capable of addressing $2^{20}$ = 1Mega Byte memory.

To generate this 20 bit physical address from 2 sixteen bit registers, the following procedure is adopted. The 20 bit address is generated from two 16-bit registers. The first 16-bit register is called the segment base register. These are code segment registers to hold programs, data segment register to keep data, stack segment register for stack operations and extra segment register to keep strings of data. The contents of the segment registers are shifted left four times with zeroes (0’s) filling on the right hand side. This is similar to multiplying four hex numbers by the base 16. This multiplication process takes place in the adder and thus a 20 bit number is generated. This is called the base address. To this a 16-bit offset is added to generate the 20-bit physical address.

Segmentation helps in the following way. The program is stored in code segment area. The data is stored in data segment area. In many cases the program is optimized and kept unaltered for the specific application. Normally the data is variable. So in order to test the program with a different set of data, one need not change the program but only have to alter the data. Same is the case with stack and extra segments also, which are only different type of data storage facilities.

Generally, the program does not know the exact physical address of an instruction. The assembler, a software which converts the Assembly Language Program (MOV, ADD etc.) into machine code (3EH, 4CH etc) takes care of address generation and location.

**DIRECTIVES AND OPERATOR**

- Assembler: is a program that accepts an assembly language program as input and converts it into an object module and prepares for loading the program into memory for execution.
- Loader (linker) further converts the object module prepared by the assembler into executable form, by linking it with other object modules and library modules.
- The final executable map of the assembly language program is prepared by the loader at the time of loading into the primary memory for actual execution.
- The assembler prepares the relocation and linkages information (subroutine, ISR) for loader.
- The operating system that actually has the control of the memory, which is to be allotted to the program for execution, passes the memory address at which the program is to be loaded for execution and the map of the available memory to the loader.
- Based on this information and the information generated by the assembler, the loader generates an executable map of the program and further physically loads it into the memory and transfers control to for execution.
- Thus the basic task of an assembler is to generate the object module and prepare the loading and linking information.

## Procedure for assembling a program

- Assembling a program proceeds statement by statement sequentially.

- The first phase of assembling is to analyze the program to be converted. This phase is called Pass1 defines and records the symbols, pseudo operands and directives. It also analyses the segments used by the program types and labels and their memory requirements.
- The second phase looks for the addresses and data assigned to the labels. It also finds out codes of the instructions from the instruction machine, code database and the program data.
- It processes the pseudo operands and directives.
- It is the task of the assembler designer to select the suitable strings for using them as directives, pseudo operands or reserved words and decides syntax.

**Directives**

- Also called as pseudo operations that control the assembly process.
- They indicate how an operand or section of a program to be processed by the assembler.
- They generate and store information in the memory.

**Assembler Memory models**

- Each model defines the way that a program is stored in the memory system.
- Tiny: data fits into one segment written in .COM format
- Small: has two segments data and memory.
- There are several other models too.

**Directive for string data in a memory segment**

- DB define byte
- DW define word
- DD define double word
- DQ define 10 bytes

Example

```
Data1  DB   10H,11H,12H
Data2  DW   1234H
```

- SEGMENT: statement to indicate the start of the program and its symbolic name.
- Example

```
Name  SEGMENT
      Variable_name     DB    …….
      Variable_name     DW    …….
Name  ENDS

Data  SEGMENT
      Data1       DB    …….
      Data2       DW    …….
Data  ENDS

Code  SEGMENT
      START: MOV AX,BX
      …
      …
      …
Code  ENDS
```

Similarly the stack segment is also declared.

- For small models

```
.DATA
…
…
ENDS
```

The ENDS directive indicates the end of the segment.

- Memory is reserved for use in the future by using a **?** as an operand for DB DW or DD directive. The assembler sets aside a location and does not initialize it to any specific value (usually stores a zero). The DUP (duplicate) directive creates an array and stores a zero.
- Example

```
Data1  DB    5 DUP(?)
```

This reserves 5 bytes of memory for a array data1 and initializes each location with 05H

- ALIGN: memory array is stored in word boundaries.
- Example

ALIGN 2 means storing from an even address

| | |
|---|---|
| Address 0 | XX |
| Address 1 | YY |
| Address 2 | XX |

The data XX is aligned to the even address.

- ASSUME, EQU, ORG
- ASSUME tells the assembler what names have been chosen for Code, Data Extra and Stack segments. Informs the assembler that the register CS is to be initialized with the address allotted by the loader to the label CODE and DS is similarly initialized with the address of label DATA.
- Example

```
ASSUME CS: Name of code segment
ASSUME DS: Name of the data segment

ASSUME CS: Code1, DS: Data1
```

- EQU: Equates a numeric, ASCII(American Standard Code for Information Interchange) or label to another label.
- Example

```
Data  SEGMENT
        Num1  EQU   50H
        Num2  EQU   66H
Data ENDS
```

Numeric value 50H and 66H are assigned to Num1 and Num2

- ORG: Changes the starting offset address of the data in the data segment
- Example

```
ORG 100H
100     data1   DB      10H
```

it can be used for code too.

- PROC & ENDP: indicate the start and end of the procedure. They require a label to indicate the name of the procedure.
- NEAR: the procedure resides in the same code segment. (Local)
- FAR: resides at any location in the memory.
- Example

```
Add   PROC  NEAR
      ADD AX,BX
      MOV CX,AX
      RET
Add   ENDP
```

PROC directive stores the contents of the register in the stack.

- EXTRN, PUBLIC informs the assembler that the names of procedures and labels declared after this directive have been already defined in some other assembly language modules.
- Example

If you want to call a Factorial procedure of Module1 from Module2 it must be declared as PUBLIC in Module1.

- Example

A sample for full segment definition

```
Data   SEGMENT
       Num1  DB     10H
       Num2  DB     20H
       Num3  EQU    30H
Data   ENDS

       ASSUME CS:Code,DS:Data
Code   SEGMENT
       START: MOV          AX,Data
              MOV   DS,AX
              MOV   AX,Num1
              MOV   CX,Num2
```

```
                ADD   AX,CX
Code    ENDS
```

- Example

A sample for small model

```
. MODEL SMALL

.Data
        Num1  DB    10H
        Num2  DB    20H
        Num3  EQU   30H

.Code
        HERE: MOV   AX,@Data
              MOV   DS,AX
              MOV   AX,Num1
              MOV   CX,Num2
              ADD   AX,CX
```

## UNIVERSITY QUESTIONS & SOLUTIONS

1. Explain the assembler directives used in 8086:

SOLN:

**Directives**

- Also called as pseudo operations that control the assembly process.
- They indicate how an operand or section of a program to be processed by the assembler.
- They generate and store information in the memory.

**Assembler Memory models**

- Each model defines the way that a program is stored in the memory system.
- Tiny: data fits into one segment written in .COM format
- Small: has two segments data and memory.
- There are several other models too.

**Directive for string data in a memory segment**

- DB define byte
- DW define word
- DD define double word
- DQ define 10 bytes

Example

```
Data1  DB    10H,11H,12H
Data2  DW    1234H
```

- SEGMENT: statement to indicate the start of the program and its symbolic name.

2. Write a program to add two double words and store the result in data segment

SOLN:

```
.model small
.stack
.data
        num1 dd 01234567h          ;// two 32-bit numbers
        num2 dd 89abcdefh          ;   to be added\\
        sum dd (0)
        carry db (0)

.code
        mov ax, @data                      ;//Intialize the
        mov ds, ax                 ;  data segment\\
        mov ax, word ptr num1              ;ax = LSBs of 1st num
        mov bx, word ptr num2              ;bx = LSBs of 2nd num
        clc
        add ax, bx                 ;ax = ax + bx
        mov bx, word ptr num1+2            ;bx = MSBs of 1st num
        mov cx, word ptr num2+2            ;cx = MSBs of 2nd num
        adc bx, cx                 ;bx = bx + cx + CF
        mov word ptr sum+2, bx             ;// store the result
        mov word ptr sum, ax       ;   in the memory\\
        adc carry, 00h             ;save carry if any
        mov ah, 4ch                ;// Terminate
        int 21h                    ;   the program\\
end
```

## RECOMMENDED QUESTIONS

**1.** With an example explain the difference between MUL and IMUL instructions.

2. Explain the instruction templates for the following instructions:

a) MOV 46H [BP], DX b) TEST AX, 83H

3. What is meant by segment override prefix? Explain

4. Explain the assembler directives used in 8086:

i) MODEL ii) PUBLIC iii)EQU iv)ALIGN v)PTR

5. Write a program to add two ASCII number and store the result in data segment

## UNIT-3: BYTE AND STRING MANIPUATON

**BYTE AND STRING MANIPULATION:** String instructions, REP Prefix, Table translation, Number format conversions, Procedures, Macros, Programming using keyboard and video display

**TEXT BOOKS:**

1. **Microcomputer systems-The 8086 / 8088** Family – Y.C. Liu and G. A. Gibson, 2E PHI - 2003
2. **The Intel Microprocessor, Architecture, Programming and Interfacing**-Barry B. Brey, 6e, Pearson Education / PHI, 2003

# Strings and String Handling Instructions :

The 8086 microprocessor is equipped with special instructions to handle string operations. By string we mean a series of data words or bytes that reside in consecutive memory locations. The string instructions of the 8086 permit a programmer to implement operations such as to move data from one block of memory to a block elsewhere in memory. A second type of operation that is easily performed is to scan a string and data elements stored in memory looking for a specific value. Other examples are to compare the elements and two strings together in order to determine whether they are the same or different.

## Move String : MOV SB, MOV SW:

An element of the string specified by the source index (SI) register with respect to the current data segment (DS) register is moved to the location specified by the destination index (DI) register with respect to the current extra segment (ES) register.

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

Example : Block move program using the move string instruction

```
        MOV AX, DATA SEG ADDR
        MOV DS, AX
        MOV ES, AX
        MOV SI, BLK 1 ADDR
        MOV DI, BLK 2 ADDR
        MOV CK, N
        CDF    ; DF=0

NEXT :  MOV SB
        LOOP NEXT
        HLT
```

**Load and store strings :** (LOD SB/LOD SW and STO SB/STO SW)

LOD SB: Loads a byte from a string in memory into AL. The address in SI is used relative to DS to determine the address of the memory location of the string element.

(AL) ← [(DS) + (SI)]

(SI) ← (SI) $\pm$ 1

LOD SW : The word string element at the physical address derived from DS and SI is to be loaded into AX. SI is automatically incremented by 2.

(AX) ← [(DS) + (SI)]

(SI) ← (SI) $\pm$ 2

STO SB : Stores a byte from AL into a string location in memory. This time the contents of ES and DI are used to form the address of the storage location in memory

[(ES) + (DI)] ← (AL)

(DI) ← (DI) $\pm$ 1

**STO SW :** [(ES) + (DI)] ← (AX)

(DI) ← (DI) $\pm$ 2

| Mnemonic | Meaning | Format | Operation | Flags affected |
|---|---|---|---|---|
| MOV SB | Move String Byte | MOV SB | ((ES)+(DI))←((DS)+(SI))<br>(SI)←(SI) $\mp$ 1<br>(DI) ← $\mp$ 1 | None |
| MOV SW | Move | MOV | ((ES)+(DI))←((DS)+(SI)) | None |

| | | | | |
|---|---|---|---|---|
| | String Word | SW | ((ES)+(DI)+1)←(DS)+(SI)+1)<br>(SI) ← (SI) ∓ 2<br>(DI) ← (DI) ∓ 2 | |
| LOD SB / LOD SW | Load String | LOD SB/ LOD SW | (AL) or (AX) ←((DS)+(SI))<br>(SI)←(SI) ∓ 1 or 2 | None |
| STOSB/ STOSW | Store String | STOSB/ STOSW | ((ES)+(DI))←(AL) or (AX)<br>(DI) ← (DI) 71 or 2 | None |

Example : Clearing a block of memory with a STOSB operation.

```
        MOV AX, 0
        MOV DS, AX
        MOV ES, AX
        MOV DI, A000
        MOV CX, OF
        CDF
AGAIN : STO SB
        LOOP NE AGAIN
```

NEXT : Clear A000 to A00F to $00_{16}$

# Repeat String : REP

The basic string operations must be repeated to process arrays of data. This is done by inserting a repeat prefix before the instruction that is to be repeated.

Prefix REP causes the basic string operation to be repeated until the contents of register CX become equal to zero. Each time the instruction is executed, it causes CX to be tested for zero, if CX is found to be nonzero it is decremented by 1 and the basic string operation is repeated.

Example : Clearing a block of memory by repeating STOSB

```
        MOV AX, 0
        MOV ES, AX
        MOV DI, A000
        MOV CX, OF
        CDF
```

```
        REP STOSB
        NEXT:
```

The prefixes REPE and REPZ stand for same function. They are meant for use with the CMPS and SCAS instructions. With REPE/REPZ the basic compare or scan operation can be repeated as long as both the contents of CX are not equal to zero and zero flag is 1.

REPNE and REPNZ works similarly to REPE/REPZ except that now the operation is repeated as long as CX≠0 and ZF=0. Comparison or scanning is to be performed as long as the string elements are unequal (ZF=0) and the end of the string is not yet found (CX≠0).

| **Prefix** | **Used with** | **Meaning** |
|---|---|---|
| REP | MOVS<br>STOS | Repeat while not end of string CX≠0 |
| REPE/ REPZ | CMPS<br>SCAS | CX≠0 & ZF=1 |
| REPNE/REPNZ | CMPS<br>SCAS | CX≠0 & ZF=0 |

Example :

```
CLD                     ; DF =0
MOV AX, DATA SEGMENT ADDR
MOV DS, AX
MOV AX, EXTRA SEGMENT ADDR
MOV ES, AX
MOV CX, 20
MOV SI, OFFSET MASTER
MOV DI, OFFSET COPY
REP MOVSB
```

Moves a block of 32 consecutive bytes from the block of memory locations starting at offset address MASTER with respect to the current data segment (DS) to a block of locations starting at offset address copy with respect to the current extra segment (ES).

# Auto Indexing for String Instructions :

SI & DI addresses are either automatically incremented or decremented based on the setting of the direction flag DF. When CLD (Clear Direction Flag) is executed DF=0 permits auto increment by 1.When STD (Set Direction Flag) is executed DF=1 permits auto decrement by 1.

| Mnemonic | Meaning | Format | Operation | Flags affected |
|---|---|---|---|---|
| CLD | Clear DF | CLD | (DF) ← 0 | DF |
| STD | Set DF | STD | (DF) ← 1 | DF |

## 1. LDS Instruction:

LDS register, memory (Loads register and DS with words from memory)

This instruction copies a word from two memory locations into the register specified in the instruction. It then copies a word from the next two memory locations into the DS register. LDS is useful for pointing SI and DS at the start of the string before using one of the string instructions. LDS affects no flags.

Example 1 :LDS BX [1234]

Copy contents of memory at displacement 1234 in DS to BL. Contents of 1235H to BH. Copy contents at displacement of 1236H and 1237H is DS to DS register.

Example 2 : LDS, SI String – Pointer

(SI) ← [String Pointer]

(DS) ← [String Pointer +2]

DS, SI now points at start and desired string

## 2. LEA Instruction :

Load Effective Address (LEA register, source)

This instruction determines the offset of the variable or memory location named as the source and puts this offset in the indicated 16 bit register.

LEA will not affect the flags.

Examples :

LEA BX, PRICES

Load BX with offset and PRICES in DS

LEA BP, SS : STACK TOP

Load BP with offset of stack-top in SS

LEA CX, [BX] [DI]

Loads CX with EA : (BX) + (DI)

## 3. LES instruction :

LES register, memory

Example 1: LES BX, [789A H]

(BX) ← [789A] in DS

(ES) ← [789C] in DS

Example 2 : LES DI, [BX]

(DI) ← [BX] in DS

(ES) ← [BX+2] in DS

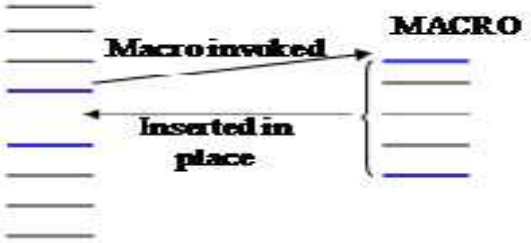
## Macros

- Macros provide several powerful mechanisms useful for the development of generic programs.
- A Macro is a group of instructions with a name.
- When a macro is invoked, the associated set of instructions is **inserted in place** in to the source, replacing the macro name. This "macro expansion" is done by a Macro Preprocessor and it happens before assembly. Thus the actual Assembler sees the "expanded" source!
- We could consider the macro as shorthand for a piece of text; somewhat like a new pseudo-code instruction.

### Macros and Procedures:

- Macros are similar to procedures in some respects, yet are quite different in many other respects.
- **Procedure:**
  - Only one copy exists in memory. Thus memory consumed is less.
  - "Called" when required;
  - Execution time overhead is present because of the call and return instructions.
- **Macro:**
  - When a macro is "invoked", the corresponding text is "inserted" in to the source. Thus multiple copies exist in the memory leading to greater space requirements.
  - However, there is no execution overhead because there are no additional call and return instructions. The code is in-place.

These concepts are illustrated in the following figure:


**MACRO Definition:**

A macro has a name. The body of the macro is defined between a pair of directives, MACRO and ENDM. Two macros are defined in the example given below.

**Examples of Macro Definitions:**

```
; Definition of a Macro named PA2C
PA2C    MACRO
        PUSH  AX
        PUSH  BX
        PUSH  CX
        ENDM


; Another Macro named POPA2C is defined here
POPA2C  MACRO
        POP  CX
        POP  BX
        POP  AX
        ENDM
```

**Examples of Macro usage:**

The following examples illustrate the use of macros. We first show the source with macro invocation and then show how the expanded source looks.

Program with macro invocations:

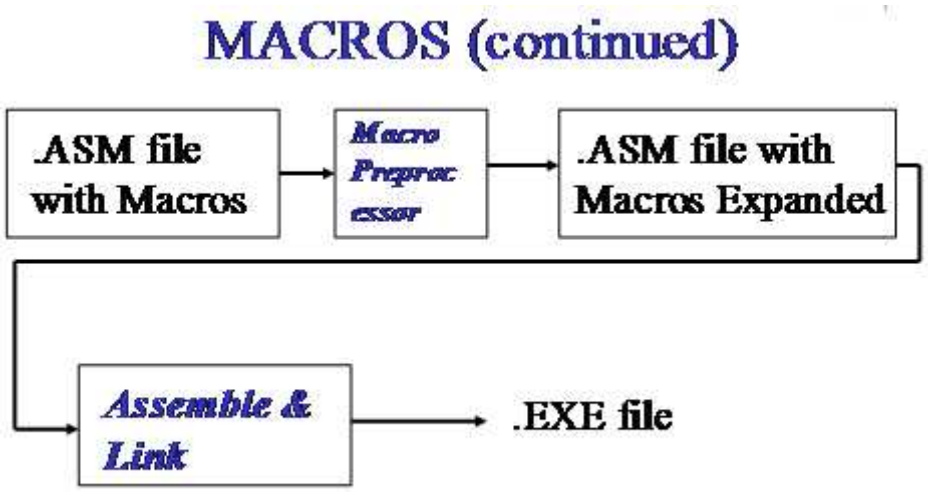
```
        PA2C
        MOV  CX, DA1
        MOV  BX, DA2
        ADD   AX, BX
        ADD   AX, CX
```

```
MOV  DA2, AX
POPA2C
```

When the Macro Preprocessor expands the macros in the above source, the expanded source looks as shown below:

```
PUSH AX
PUSH  BX
PUSH  CX
MOV  CX, DA1
MOV  BX, DA2
ADD  AX, BX
ADD   AX, CX
MOV  DA2, AX
POP   CX
POP   BX
POP   AX
```

Note how the macro name is replaced by the associated set of instructions. Thus, macro name does not appear in the expanded source code. In other words, the actual Assembler does not "see" the macros. What gets assembled is the expanded source. This process is illustrated in the following figure:




**MACROS with Parameters:**

Macros have several other interesting and powerful capabilities. One of these is the definition and use of macros with parameters.

A macro can be defined with parameters. These are dummy parameters. When the macro is invoked, we provide the actual parameters. During the macro expansion, the dummy

parameters are replaced by the corresponding actual parameters. The association between the dummy and actual parameters is positional. Thus the first actual parameter is associated with the first dummy parameter, the second actual parameter with the second dummy one and so on. This is illustrated in the following example where a Macro named COPY is defined with two parameters called A and B.

Example:

```
COPY    MACRO    A , B
        PUSH AX
        MOV  AX, B
        MOV  A, AX
        POP  AX
        ENDM
```

The macro is invoked in the following code with actual parameters as VAR1 and VAR2. Thus during the macro expansion, the parameter A is replaced by VAR1 and the parameter B is replaced by VAR2.

```
COPY    VAR1, VAR2
```

The expanded code is:

```
PUSH AX
MOV  AX, VAR2
MOV  VAR1, AX
POP  AX
```

**Local Variables in a Macro:**

- Assume that a macro definition includes a label RD1 as in the following example:

```
READ MACRO    A
     PUSH     DX
RD1: MOV      AH, 06
     MOV      DL, 0FFH
     INT      21H
     JE       RD1    ;; No key, try again
     MOV      A, AL
     POP      DX
     ENDM
```

- If READ macro is invoked more than once, as in

```
READ VAR1
READ  VAR2
```

assembly error results!

- The problem is that the label RD1 appears in the expansion of READ VAR1 as well as in the expansion of READ VAR2. Hence, the label RD1 appears in both the expansions. In other words, the Assembler sees the label RD1 at two different places and this results in the "Multiple Definition" error!
- **SOLUTION:** Define RD1 as a **local variable** in the macro.

```
READ MACRO    A
     LOCAL    RD1
     PUSH     DX
RD1: MOV      AH, 06
     MOV      DL, 0FFH
     INT      21H
     JE       RD1    ;; No key, try again
     MOV      A, AL
     POP      DX
     ENDM
```

- Now, in each invocation of READ, the label RD1 will be replaced, **automatically,** with a unique label of the form ??xxxx ; where xxxx is a unique number generated by Assembler. This eliminates the problem of multiple definitions in the expanded source.

- With the use of local variable as illustrated above,

```
READ    VAR1
```

gets expanded as:

```
        PUSH    DX
??0000: MOV     AH, 06
        MOV     DL, 0FFH
        INT     21H
        JE      ??0000  ;; No key, try again
        MOV     VAR1, AL
        POP     DX
```

Subsequently, if we write

```
READ    VAR2
```

it gets expanded as:

```
        PUSH    DX
??0001: MOV     AH, 06
        MOV     DL, 0FFH
        INT     21H
        JE      ??0001  ;; No key, try again
        MOV     VAR2, AL
        POP     DX
```

Note how each invocation of the READ macro gets expanded with a new and unique label, generated automatically by the Assembler, in place of the local variable RD1. Further, note that LOCAL directive must immediately follow the MACRO directive. Another feature to note is that Comments in Macros are preceded by ;; (two semicolons) , and not as usual by ; (a single semicolon).

**File of Macros:**

- We can place all the required Macros in a file of its own and then include the file into the source.
- Example: Suppose the Macros are placed in D:\MYAPP\MYMAC.MAC
  In the source file, we write

**Advanced Features:**

- Conditional Assembly
- REPEAT , WHILE, and FOR statements in MACROS

**Conditional Assembly:**

- A set of statements enclosed by **IF** and **ENDIF** are ***assembled*** if the condition stated with IF is true; otherwise, the statements are ***not assembled; no code is generated.***
- This is an Assembly time feature; ***not run-time*** behavior!
- Allows development of generic programs. From such a generic program, we can produce specific **source programs** for specific application contexts.
- Example: Assume that our generic program has the following statements:

```
IF  WIDT
  WIDE      DB    72
ELSE
  WIDE      DB    80
```

```
ENDIF
```

Now the assembly language program that is generated depends on the value of WIDT. Assume the block is preceded by

```
WIDT EQU 1
```

Then the assembled code is:

```
WIDE   DB    72
```

It is important to note that the Assembler sees a source file that has only the above statement.

- Another case:

```
WIDT EQU   0
IF  WIDT
   WIDE  DB  72
ELSE
   WIDE  DB  80
ENDIF
```

What gets assembled is: **WIDE DB 80**

- There are several other directives that can be used for Conditional Assembly as listed below:

| | |
|---|---|
| **IF** | **If the expression is true** |
| **IFB** | **If the argument is blank** |
| **IFNB** | **If the argument is not blank** |
| **IFDEF** | **If the label has been defined** |
| **IFNDEF** | **If the label has not been defined** |
| **IFIDN** | **If argument 1 equals argument 2** |
| **IFDIF** | **If argument 1 does not equal argument 2** |

With each of the above constructs, the code that follows gets assembled only if the stated condition is true.

**REPEAT Statement:**

This statement allows a block of code to be repeated the specified number of times. This avoids repetitive typing and is much more elegant than Editor-level Copy-and-Paste operation.

Example:

```
REPEAT 3
        INT 21H
        INC   DL
```

```
ENDM
```

The generated code would be 3 repetitions of the block of 2 statements enclosed within REPEAT and ENDM as shown below:

```
INT   21H
INC   DL
INT   21H
INC   DL
INT   21H
INC   DL
```

**WHILE Statement:**

This statement allows a block of code to be repeated while the condition specified with the WHILE is true.

Example: Consider the following code

```
SQ  LABEL  BYTE
SEED = 1
RES = SEED * SEED
WHILE  RES LE 9
        DB      RES
SEED = SEED + 1
RES = SEED * SEED
ENDM
```

Note that SEED and the arithmetic statements involving SEED and RES are all Assembly time actions. Apart from the initial label **SQ**, the only statement to actually get repeated is **DB RES.** The logic is follows: Initially the label SQ is generated. SEED is initialized to 1 and RES is computed as 1 * 1 = 1. Now RES LE 9 is true as the value of RES is 1 which is less than 9. So the code DB 1 is generated. The next statement within the scope of WHILE, “SEED = SEED + 1” is executed making SEED assume the value of 2. The next statement within the scope of WHILE is RES = SEED * SEED. This is also executed and RES assumes the value of 4. This completes one pass of execution of the WHILE block. So, the condition associated with WHILE is again evaluated. This is again TRUE as 4 is less than 9. So again DB 9 is generated. Reasoning as before, we see that DB 9 is also generated. However, in the next pass SEED is 4 and RES is 16. So the condition RES LE 9 evaluates to FALSE and WHILE loop is exited!

**Thus the generated code is:**

```
SQ    DB    01
      DB    04
      DB    09
```

**FOR Statement:**

This is very similar to the FOR of languages like PERL. With the FOR statement, a control variable and a list of values are specified. The control variable is successively assigned values from the specified list and for each such value, the following block of statements is repeated.

**Example:**

```
DISP   MACRO CHR:VARARG
       MOV AH, 2
       FOR  ARG, <CHR>
       MOV  DL, ARG
       INT    21H
       ENDM
       ENDM
```

The outer Macro has one parameter which is specified as sequence of characters of variable length. The inner FOR statement has two enclosed statements which will be repeated for each value in the list <CHR>. Thus in the following illustration, DISP is invoked with 3 characters as parameters. The two statements within FOR scope are thus repeated 3 times with ARG successively assuming the 3 characters.

**Thus, the statement**

```
DISP  'V','T','U'
```

**gets expanded as**

```
MOV       AH, 2
MOV       DL,'V'
INT       21H
MOV       DL, 'T'
INT       21H
MOV       DL, 'U'
INT       21H
```

## NUMBER FORMAT CONVERSION:

- Often Data available in one format needs to be converted in to some other format.

  Examples:
  - ASCII to Binary
  - Binary to ASCII
  - BCD to 7-Segment Code … … …
- Data Conversion may be based on
  - Algorithm
  - Look –Up Table

**Converting from Binary to ASCII:**

In many contexts, for example, when displaying a number on the screen, we must produce a sequence of ASCII characters representing the number to be displayed. Thus the given number must be converted to a string of equivalent ASCII characters.

- Example: Binary number: 0100 0011 = 43H = 67 D

  To display this on the screen, we need to convert this binary number in to **Two ASCII** characters, ‘6’ and ‘7’.

  ASCII code for character ‘6’ is 36H and

  ASCII code for character ‘7’ is 37H.

  So, we need to produce 36H and 37H as output given 43H as input.

- Another Example: Binary number: 0000 0010 0100 0011 = 0243H = 579 D

  To display this on the screen, we need Three **ASCII** characters, ‘5’, ‘7’ and ‘9’.

  ASCII code for character ‘5’ is 35H,

  ASCII code for character ‘7’ is 37H, and

  ASCII code for character ‘9’ is 39H

  So, we need to produce 35H, 37H and 39H as output given 0243H as input

**Binary to ASCII Algorithm:**

Example: Binary number: 0000 0010 0100 0011 = 579 D

- Divide 579 by 10 ; Quotient = 57 ; Remainder = 9 , Save **9**
- Divide 57 by 10; Quotient = 5 ; Remainder = 7 , Save **7**
- Divide 5 by 10; Quotient = 0 ; Remainder = 5 , Save **5**

- Quotient = 0 → Conversion Complete.
- Remainders saved in the order of 9, 7, and 5.
- Retrieve remainders in the order of 5, 7, and 9.

  (As the order of retrieval is the reverse of the order of producing these digits, the most convenient technique is to Save & Retrieve the digits using Stack)
- While retrieving, add 30H to convert the digit to ASCII code and then display it (or print it, or save it…)
- Thus the algorithm is:

  While the number is not equal to 0

  Divide the number by 10;

  Push the remainder digit on the stack;

  Set number <- quotient

  While stack not empty

  Pop a digit from the stack

  Add 30H to covert it to ASCII and display it

  Return.
- This algorithm is implemented in the following program:

**Binary to ASCII Program:**

; Input : 16-Bit Binary Number in AX

; Output: Equivalent ASCII displayed on screen

```
        .MODEL TINY
        .CODE
        .STARTUP
        MOV  AX, 2A5H     ; Test value
        CALL B2A          ; Binary to ASCII and Display
        .EXIT
B2A     PROC   NEAR
        PUSH   DX
        PUSH   CX
        PUSH   BX
        MOV    CX, 0      ; Count of ASCII digits, Initialized to 0
        MOV    BX, 10     ; Divisor is 10
B2A1:   MOV    DX, 0      ; Dividend in DX, AX. So set DX = 0
```

```
        DIV     BX          ; Divide by 10
        PUSH    DX          ; Save remainder digit on the stack
        INC     CX          ; Increment digit count
        OR      AX, AX      ; Conversion completed ? (Quotient, i.e AX = 0 ?)
        JNZ     B2A1        ; No, continue division
; Conversion is complete as quotient in AX = 0
; Count of remainder digits is in CX
B2A2:   POP     DX          ; Retrieve remainder in DL
        ADD     DL, 30H     ; Convert to ASCII
        MOV     AH, 06H     ; Console Display Function
        INT     21H         ; DOS Service, display digit
        LOOP    B2A2        ; Repeat for all digits

; Clean up & Return. AX is destroyed
        POP     BX
        POP     CX
        POP     DX
        RET
B2A     ENDP
        END
```

**Another Method for Binary to ASCII Conversion:**

- When the input number is less than 100, an alternative, simpler method exists.
- AAM (ASCII Adjust AX After Multiplication) instruction converts value in AX in to 2-Digit Unpacked BCD and leaves it in AX.
- Example: AX = 0027H (39 Decimal)
  Execute AAM ; Now, AX = 0309H ; This is Unpacked BCD.
- Now, add 3030H to AX to get 3339H ; This is Packed ASCII representation.
- Separate the two bytes (unpack) to get the two ASCII characters representing the given number (33H and 39H).
- Works only when the number is less than 100 as the maximum unpacked BCD that we can have in the AX register is 0909H only.
- The following program is developed based on this idea.

```
; Input :  Binary Number in AL, Assumed <100
; Output: Equivalent ASCII displayed on screen
```

```
.MODEL TINY
        .CODE
        .STARTUP
         MOV   AL, 2AH     ; Test value
         CALL  B2A         ; Binary to ASCII and Display
        .EXIT
B2A   PROC  NEAR
        PUSH  DX
        MOV   AH, 0        ; AX = Number
         AAM               ; AX = Unpacked BCD
        ADD   AX, 3030H    ; Convert to ASCII
        PUSH  AX
; Now, unpack and display
MOV   DL, AH       ; First Digit
        MOV   AH, 06H      ; Display Function
        INT   21H          ; Display first digit
        POP   AX           ; Retrieve value
        MOV   DL, AL       ; Second Digit
        MOV   AH, 06H      ; Display Function
         INT   21H         ; Display second digit
; Clean up & Return. AX is destroyed
        POP    DX
        RET
     B2A   ENDP
        END
```

**Refinements:**

- Suppose the input is: AL = 7H. What is displayed is 07
- Can we replace leading 0 with a blank so that the display looks better? Thus, instead of 07, the display should be 7.
- Yes. We need to check if the first digit is 0. If so, display 20H (blank); else, display the digit.
- We need to modify the previous program to incorporate this check for a leading 0.
- Old Code for displaying first digit:

```
        MOV    DL, AH   ; First Digit
        MOV    AH, 06H ; Display  Function
        INT     21H        ; Display first digit
```

- Revised Code for displaying first digit:

```
        ADD  AH, 20H
        CMP  AH, 20H    ; First Digit = 0?
        JZ      B2A1        ; Display blank (ASCII Code is 20H)
        ADD  AH, 10H    ; Add 10H more to get the correct ASCII Code for the digit
B2A1:   MOV  DL , AH    ; First Digit
        MOV   AH , 06H  ; Display Function
        INT     21H        ; Display first digit
```

- Incorporating this change, the program will be as shown below:

```
; Input :   Binary Number in AL, Assumed <100
; Output: Equivalent ASCII displayed on screen
            .MODEL TINY
            .CODE
            .STARTUP
             MOV    AL, 2AH     ; Test value
             CALL    B2A         ; Binary to ASCII and Display
            .EXIT
B2A         PROC    NEAR
            PUSH   DX
            MOV    AH, 0        ; AX = Number
             AAM                 ; AX = Unpacked BCD
            ADD    AX, 3030H    ; Convert to ASCII
            PUSH   AX
; Now, unpack and display
            ADD  AH, 20H
            CMP  AH, 20H        ; First Digit = 0?
            JZ      B2A1        ; YES. So, display a blank (ASCII Code is 20H)
            ADD  AH, 10H        ; No, we have already added 20H. Add 10H more
  B2A1:     MOV  DL, AH       ; First Digit itself  if not 0 , Or Blank (if 0)
            MOV    AH, 06H      ; Display Function
            INT     21H         ; Display first digit
```

```
        POP   AX          ; Retrieve value
        MOV   DL, AL      ; Second Digit
        MOV   AH, 06H     ; Display Function
        INT   21H         ; Display second digit
; Clean up & Return. AX is destroyed
        POP   DX
        RET
B2A     ENDP
        END
```

**ASCII to Binary Algorithm:**

In many contexts, for example, when reading a number from the key board, we get a sequence of ASCII characters representing the number. This string of ASCII characters must be converted to the equivalent number for further processing.

Example: Assume that ASCII character sequence '156' is the input.

- 3 characters, '1', '5', and '6'; with codes as 31H, 35H, and 36H.
- Converted Binary Value must be:
  0000 0000 1001 1100 = 009CH = 156 (decimal)

**Conversion Procedure:**

- Start with (Binary) Result = 0
- First ASCII digit 31H; Subtract 30H to get corresponding BCD digit 01H.
- Result = Result * 10 + Next BCD Digit
  Result = 0 * 10 + 01 = 0000 0000 0000 0001
- Next ASCII digit 35H; Subtract 30H to get corresponding BCD digit 05H.
- Result = Result * 10 + Next BCD Digit
  Result = 01 * 10 + 05 = 0000 0000 0000 1111
- Next ASCII digit 36H; Subtract 30H to get corresponding BCD digit 06H.
- Result = Result * 10 + Next BCD Digit
  Result = 15 * 10 + 06 = 0000 0000 1001 1100
- ASCII digits exhausted. So, conversion is completed.
- Final Result = 0000 0000 1001 1100 = 009CH = 156 (decimal)
- Based on the above ideas, the following program implements the ASCII to Binary Conversion.

```
; ASCII to Binary Program
```

```
; ASCII characters representing a number are read from key board.
; The first non-digit character (any character other than 0 through 9) typed
; signals the end of the number entry
; Result returned in AX, which is then stored in memory location TEMP.
; Result assumed not to exceed 16 bits!
; Program can be modified to accept larger numbers by implementing
; 32- bit addition.
                .MODEL      SMALL
                .DATA
        TEMP    DW    ?
                .CODE
                .STARTUP
                CALL RDNUM
                MOV  TEMP, AX
                .EXIT
        RDNUM   PROC NEAR
                PUSH BX
                PUSH CX
                MOV  CX, 10          ; Multiplier is 10
                MOV  BX, 0           ; Result initialized to 0
         RDN1:  MOV  AH, 1           ; Read Key with Echo
                INT  21H
; Check the character. If less than '0' or greater than '9' Number entry is over
                CMP  AL, '0'
                JB   RDN2
                CMP  AL,'9'
                JA   RDN2
; Is digit. Update Result
                SUB   AL, 30H       ; BCD Digit
                PUSH  AX
                MOV   AX, BX
                MUL    CX
                MOV    BX, AX       ; Result = Result * 10
                POP    AX
```

```
                MOV   AH, 0       ; AX = Current Digit
                ADD   BX, AX      ; Update Result
                JMP   RDN1        ; Repeat
; Non- digit. Clean Up and Return
     RDN2:      MOV   AX, BX      ; Result in AX
                POP   CX
                POP   BX
                RET
   RDNUM        ENDP
                END
```

**Notes:**

- The constant multiplier 10 is held in the register CX.
- In the procedure, RDNUM, the result is accumulated in the register BX and at the end, it is moved in to register AX. The result in AX is moved, in the calling program, in to the memory location TEMP.
- The BCD digit is in AL. AH is cleared to 0 so that the 16-bit value in AX represents the correct value and thus can be added directly to the accumulating result in BX. This part of the code must be changed to implement 32-bit addition if larger results are to be supported.

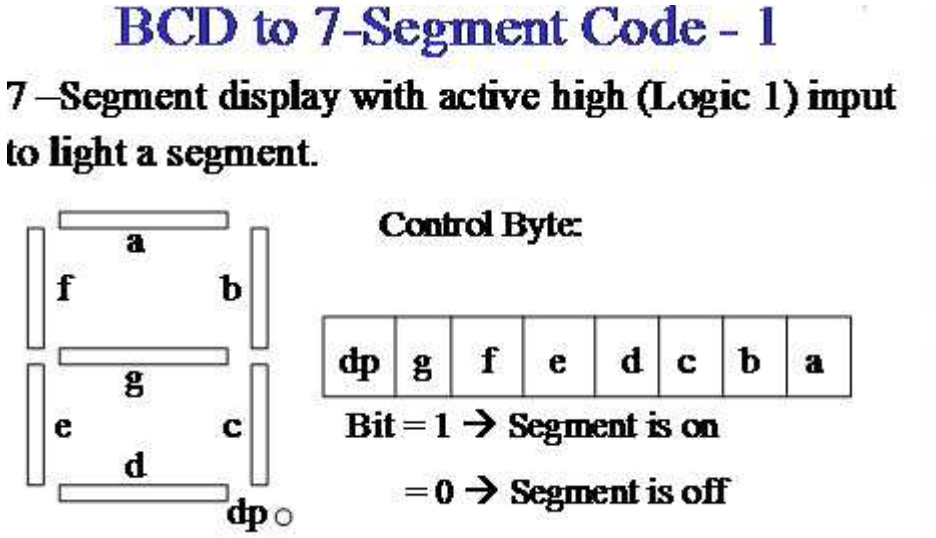
**Using Look – Up Tables for Data Conversion:**

- Often, a look-up table simplifies data conversion.
- XLAT can be used if table has up to 256 byte-entries
- Value to be converted is used to index in to the table containing conversion values.
- As an example, we will demonstrate BCD to 7-Segment code conversion.

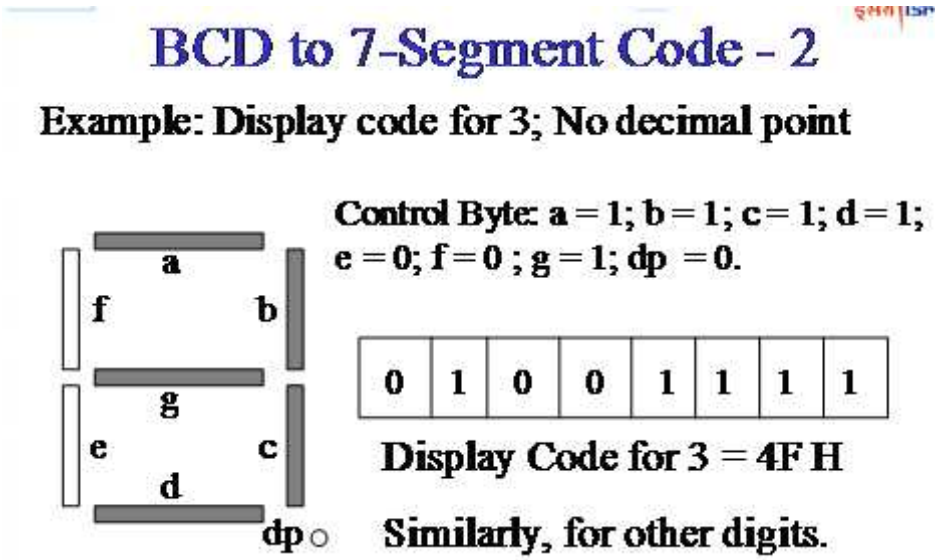
**BCD to 7-Segment Code Conversion:**

In many applications, we need to display BCD values on a 7-Segment display. The 7-Segment display device, as the name suggests, has 7 segments which can be independently controlled to be ON or OFF. Further, the device has a decimal point also that can be switched ON or OFF. The 7 segments and the decimal point are controlled by 8 bits, with one bit controlling one segment or the decimal point. The bit value required to switch on a segment depends on whether the device is of a Common – Anode type or Common – Cathode type. Here, we are assuming a Common – Anode type. Thus the segment will be ON if the corresponding controlling bit is 1 and will be off if the bit is 0.

Based on the digit to be displayed, we must determine the segments that must be ON and the ones that must be OFF. The bits controlling the segments that must be ON are set to 1 and the bits controlling the segments that must be OFF are cleared to 0. The resulting bit pattern determines the value of the 7-Segemnt code that must be output. This display structure is shown in the following figure on the next page:




As an example of determining the display code corresponding to a given BCD digit, the following figure shows the display of digit 3 and the determination of the corresponding 7-Segment code:




Based on the above logic, the following **FAR Procedure** returns the 7-Segment code in the AL register, corresponding to the BCD digit provided as input parameter in the AL register before calling the procedure.

```
; BCD to 7-Segment Code Program
; Input:    AL = BCD Digit
; Output: AL = 7-Segment code.
```

```
BT7SEG    PROC FAR
          PUSH BX
          MOV  BX, OFFSET TABLE
          XLAT CS: TABLE
          POP  BX
          RET
TABLE     DB       3FH  ; 0
          DB       06H  ; 1
          DB       5BH  ; 2
          DB       4FH  ; 3
          DB       66H  ; 4
          DB       6DH  ; 5
          DB       7DH  ; 6
          DB       07H  ; 7
          DB       7FH  ; 8
          DB       6FH  ; 9
BT7SEG    ENDP
```

**Notes:**

- XLAT instruction does not normally contain an operand. Here we are using the operand (TABLE). It is a dummy operand! It is being used here only to specify segment override. XLAT uses DS by default. Here the table is in CS. So segment override is being specified.
- More examples are discussed in the Text Book.

| | | Extended ASCII code with.... | | | |
|---|---|---|---|---|---|
| *Key* | *Scan Code* | *Nothing* | *Shift* | *Control* | *Alternate* |
| Esc | 01 | | | | 01 |
| 1 | 02 | | | | 78 |
| 2 | 03 | | | 03 | 79 |
| 3 | 04 | | | | 7A |
| 4 | 05 | | | | 7B |
| 5 | 06 | | | | 7C |
| 6 | 07 | | | | 7D |
| 7 | 08 | | | | 7E |
| 8 | 09 | | | | 7F |
| 9 | 0A | | | | 80 |
| 0 | 0B | | | | 81 |
| - | 0C | | | | 82 |
| + | 0D | | | | 83 |
| Bksp | 0E | | | | 0E |
| Tab | 0F | | 0F | 94 | A5 |
| Q | 10 | | | | 10 |
| W | 11 | | | | 11 |
| E | 12 | | | | 12 |
| R | 13 | | | | 13 |
| T | 14 | | | | 14 |
| Y | 15 | | | | 15 |
| U | 16 | | | | 16 |
| I | 17 | | | | 17 |
| O | 18 | | | | 18 |
| P | 19 | | | | 19 |
| [ | 1A | | | | 1A |
| ] | 1B | | | | 1B |
| Enter | 1C | | | | 1C |
| Enter | 1C | | | | A6 |
| Lctrl | 1D | | | | |
| Rctrl | 1D | | | | |
| A | 1E | | | | 1E |
| S | 1F | | | | 1F |
| D | 20 | | | | 20 |
| F | 21 | | | | 21 |
| G | 22 | | | | 22 |
| H | 23 | | | | 23 |
| J | 24 | | | | 24 |
| K | 25 | | | | 25 |
| L | 26 | | | | 26 |
| ; | 27 | | | | 27 |
| ' | 28 | | | | 28 |
| ` | 29 | | | | 29 |
| Lshft | 2A | | | | |
| \ | 2B | | | | |

| | | Extended ASCII code with.... | | | |
|---|---|---|---|---|---|
| *Key* | *Scan Code* | *Nothing* | *Shift* | *Control* | *Alternate* |
| Z | 2C | | | | 2C |
| X | 2D | | | | 2D |
| C | 2E | | | | 2E |
| V | 2F | | | | 2F |
| B | 30 | | | | 30 |
| N | 31 | | | | 31 |
| M | 32 | | | | 32 |
| , | 33 | | | | 33 |
| . | 34 | | | | 34 |
| / | 35 | | | | 35 |
| Gray / | 35 | | | 95 | A4 |
| Rshft | 36 | | | | |
| PrtSc | E0 2A E0 37 | | | | |
| L alt | 38 | | | | |
| R alt | 38 | | | | |
| Space | 39 | | | | |
| Caps | 3A | | | | |
| F1 | 3B | 3B | 54 | 5E | 68 |
| F2 | 3C | 3C | 55 | 5F | 69 |
| F3 | 3D | 3D | 56 | 60 | 6A |
| F4 | 3E | 3E | 57 | 61 | 6B |
| F5 | 3F | 3F | 58 | 62 | 6C |
| F6 | 40 | 40 | 59 | 63 | 6D |
| F7 | 41 | 41 | 5A | 64 | 6E |
| F8 | 42 | 42 | 5B | 65 | 6F |
| F9 | 43 | 43 | 5C | 66 | 70 |
| F10 | 44 | 44 | 5D | 67 | 71 |
| F11 | 57 | 85 | 87 | 89 | 8B |
| F12 | 58 | 86 | 88 | 8A | 8C |
| Num | 45 | | | | |
| Scroll | 46 | | | | |
| Home | E0 47 | 47 | 47 | 77 | 97 |
| Up | 48 | 48 | 48 | 8D | 98 |
| Pgup | E0 49 | 49 | 49 | 84 | 99 |
| Gray - | 4A | | | | |
| Left | 4B | 4B | 4B | 73 | 9B |
| Center | 4C | | | | |
| Right | 4D | 4D | 4D | 74 | 9D |
| Gray + | 4E | | | | |
| End | E0 4F | 4F | 4F | 75 | 9F |
| Down | E0 50 | 50 | 50 | 91 | A0 |
| Pgdn | E0 51 | 51 | 51 | 76 | A1 |
| Ins | E0 52 | 52 | 52 | 92 | A2 |
| Del | E0 53 | 53 | 53 | 93 | A3 |
| Pause | E0 10 45 | | | | |

character to the screen even if it is an unwanted character. The DOS function number 01H also responds to the control–C key combination and exits to DOS if it is typed.

```
                                .MODEL SMALL            ;select SMALL model
0000                            .DATA                   ;start DATA segment

0000  0D 0A 0A 54  MES          DB   13,10,10,'This is a test line.$'
      68 69 73 20
      69 73 20 61
      20 74 65 73
      74 20 6C 69
      6E 65 2E 24
0000                            .CODE                   ;start CODE segment
                                .STARTUP                ;start program

0017  B4 09                     MOV  AH,9               ;select function 09H
0019  BA 0000 R                 MOV  DX,OFFSET MES      ;address character string
001C  CD 21                     INT  21H                ;access DOS

                                .EXIT                   ;exit to DOS
                                END                     ;end of file
```

This example program can be entered into the assembler, linked, and executed to produce *"This is a test line"* on the video display.

The .EXIT directive embodies the DOS function 4CH. As shown in Appendix A, DOS function 4CH terminates a program. The .EXIT directive inserts a series of two instructions in the program, MOV AH,4CH, followed by an INT 21H instruction.

# UNIVERSITY QUESTIONS & SOLUTIONS

1. Write an ALP to search a character in a string of N characters using linear search. (July 2007)

Soln:

```
.model small
.stack
.data
        string db 'ELECTRONICS $'
        str_cnt equ 0bh
        msg1 db 'character found $'
        msg2 db 'character not found $'
        loc db ?

.code
        mov ax, @data               ;//Intialize
        mov ds, ax          ;   data and
        mov es, ax          ;    extra segment\\
        mov cx, str_cnt             ;intialize string counter
        lea di, string      ; di = offset of string
        mov al, 'T'         ;al = character to be searched
back:   cmp al, byte ptr[di]  ;compare char with first element in the string
        je found                    ;if found jump to found
        inc di              ;//else proceed with
        loop back           ;    comparing all chars in string\\
        lea dx, msg2        ;//  if not found display
        mov ah, 09h         ;      msg2 using DOS services\\
        int 21h
        jmp last
found:  mov ah, 09h         ;//if found display
        lea dx, msg1        ;    msg1 using DOS services\\
        int 21h
        mov dx, str_cnt             ;// store the offset
        sub dx, cx          ;    location of the desired
        mov loc, dx         ;      char in a string\\
last:   mov ah, 4ch         ;//Terminate
        int 21h             ;    the program\\
end
```

2. explain the procedure for converting from binary to ASCII. Jan 2009 (2008)

## Soln

**Converting from Binary to ASCII:**

In many contexts, for example, when displaying a number on the screen, we must produce a sequence of ASCII characters representing the number to be displayed. Thus the given number must be converted to a string of equivalent ASCII characters.

- Example: Binary number: 0100 0011 = 43H = 67 D

To display this on the screen, we need to convert this binary number in to **Two ASCII** characters, ‘6’ and ‘7’.

ASCII code for character ‘6’ is 36H and

ASCII code for character ‘7’ is 37H.

So, we need to produce 36H and 37H as output given 43H as input.

- Another Example: Binary number: 0000 0010 0100 0011 = 0243H = 579 D

  To display this on the screen, we need Three **ASCII** characters, ‘5’, ‘7’ and ‘9’.

  ASCII code for character ‘5’ is 35H,

  ASCII code for character ‘7’ is 37H, and

  ASCII code for character ‘9’ is 39H

  So, we need to produce 35H, 37H and 39H as output given 0243H as input

## RECOMMENDED QUESTIONS

**1.** Explain the five types of string instruction with example

2. Explain the following instructions:

i) MOVSB ii) repeat prefix iii)STOSW iv)SCASB v)CMPS

3. Write an ALP to convert lowercase to upper case using the modular programming approach.

4. Use two far procedures one for reading from keyboard and one for displaying.

# UNIT: 4:

**8086 INTERRUPTS:** 8086 Interrupts and interrupt responses, Hardware interrupt applications, Software interrupt applications, Interrupt examples

What is an interrupt ?

An interrupt is the method of accessing the MPU by a peripheral device. An interrupt is used to cause a temporary halt in the execution of a program. The MPU responds to the interrupt with an interrupt service routine, which is a short program or subroutine that instructs the MPU on how to handle the interrupt. When the 8086 is executing a program, it can get interrupted because of one of the following.

1.Due to an interrupt getting activated. This is called as hardware interrupt .

2.Due to an exceptional happening during an instruction execution, such as division of a number by zero. This is generally termed as exceptions or Traps.

3.Due to the execution of an Interrupt instruction like "INT 21H". This is called a Software interrupt. The action taken by the 8086 is similar for all the three cases, except for minor differences. There are two basic types of interrupts, Maskable and non-maskable.

Nonmaskable interrupt requires an immediate response by the MPU. It is usually used for serious circumstances like power failure. A maskable interrupt is an interrupt that theMPU can ignore depending upon some predetermined condition defined by the status register. Interrupts are also prioritized to allow for the case when more than one interrupt needs to be serviced at the same time.

**Hardware interrupts of 8086**

In a microcomputer system whenever an I/O port wants to communicate with themicroprocessor urgently, it interrupts the microprocessor. In such a case, themicroprocessor completes the instruction it is presently executing. Then, it saves theaddress of the next instruction on the stack top. Then it branches to an Interrupt Service Subroutine (ISS), to service the interrupting I/O port. An ISS is also commonly called as an Interrupt Handler . After completing the ISS, the processor returns to the original program, making use of the return address that was saved on the stack top.In 8086 there are two interrupt pins. They are NMI and INTR. NMI stands for non maskable interrupt. Whenever an external device activates this pin, themicroprocessor will be interrupted. This signal cannot be masked. NMI is a vectored

Definition: The meaning of ‘interrupts’ is to break the sequence of operation.While the cpu is executing a program,on ‘interrupt’ breaks the normal sequence of execution of

instructions, diverts its execution to some other program called Interrupt Service Routine (ISR).After executing ISR , the control is transferred back again to the main program.Interrupt processing is an alternative to polling.

Need for Interrupt: Interrupts are particularly useful when interfacing I/O devices, that provide or require data at relatively low data transfer rate.

Types of Interrupts: There are two types of Interrupts in 8086. They are:

(i)Hardware Interrupts and

(ii)Software Interrupts

(i) Hardware Interrupts (External Interrupts). The Intel microprocessors support hardware interrupts through:

Two pins that allow interrupt requests, INTR and NMI

One pin that acknowledges, INTA, the interrupt requested on INTR.

INTR and NMI

INTR is a maskable hardware interrupt. The interrupt can be enabled/disabled using STI/CLI instructions or using more complicated method of updating the FLAGS register with the help of the POPF instruction.

When an interrupt occurs, the processor stores FLAGS register into stack, disables further interrupts, fetches from the bus one byte representing interrupt type, and jumps to interrupt processing routine address of which is stored in location 4 * <interrupt type>. Interrupt processing routine should return with the IRET instruction.

NMI is a non-maskable interrupt. Interrupt is processed in the same way as the INTR interrupt. Interrupt type of the NMI is 2, i.e. the address of the NMI processing routine is stored in location 0008h. This interrupt has higher priority than the maskable interrupt.

– Ex: NMI, INTR.

(ii) Software Interrupts (Internal Interrupts and Instructions) .Software interrupts can be caused by:

INT instruction - breakpoint interrupt. This is a type 3 interrupt.

INT <interrupt number> instruction - any one interrupt from available 256 interrupts.

INTO instruction - interrupt on overflow

Single-step interrupt - generated if the TF flag is set. This is a type 1 interrupt.

When the CPU processes this interrupt it clears TF flag before calling the interrupt processing routine.

Processor exceptions: Divide Error (Type 0), Unused Opcode (type 6) and Escape opcode (type 7).

Software interrupt processing is the same as for the hardware interrupts.

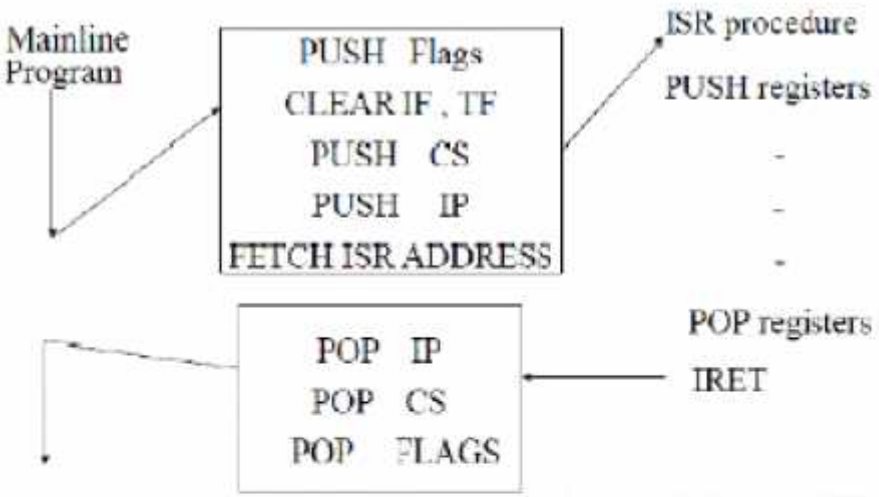
- Ex: INT n (Software Instructions)

Control is provided through:

o IF and TF flag bits

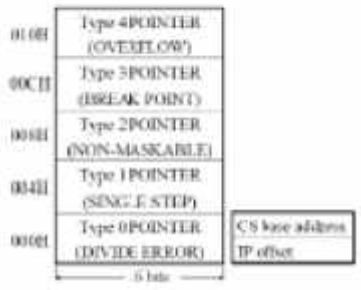
o IRET and IRETD

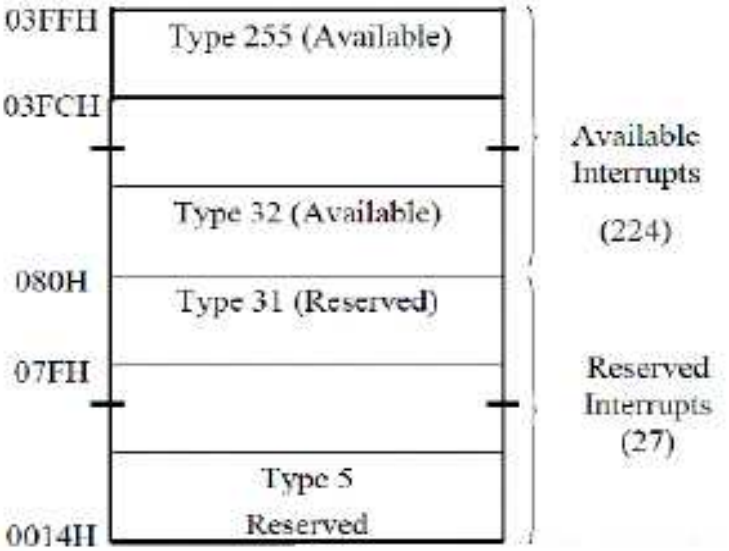
## Performance of Software Interrupts




1. It decrements SP by 2 and pushes the flag register on the stack.
2. Disables INTR by clearing the IF.
3. It resets the TF in the flag Register.
5. It decrements SP by 2 and pushes CS on the stack.
6. It decrements SP by 2 and pushes IP on the stack.
6. Fetch the ISR address from the interrupt vector table.

**Interrupt Vector Table (Click the picture for full view)**






| INT Number | Physical Address |
|---|---|
| INT 00 | 00000 |
| INT 01 | 00004 |
| INT 02 | 00008 |
| . | . |
| . | . |
| INT FF | 003FC |

**Functions associated with INT00 to INT04**

**INT 00 (divide error)**

- INT00 is invoked by the microprocessor whenever there is an attempt to divide a number by zero.
- ISR is responsible for displaying the message "Divide Error" on the screen

**INT 01**

- For single stepping the trap flag must be 1
- After execution of each instruction, 8086 automatically jumps to 00004H to fetch 4 bytes for CS: IP of the ISR.
- The job of ISR is to dump the registers on to the screen

**INT 02 (Non maskable Interrupt)**

- When ever NMI pin of the 8086 is activated by a high signal (5v), the CPU Jumps to physical memory location 00008 to fetch CS:IP of the ISR assocaiated with NMI.

**INT 03 (break point)**

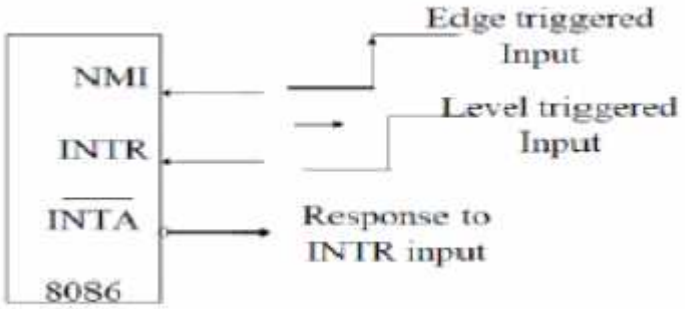
- A break point is used to examine the cpu and memory after the execution of a group of Instructions.
- It is one byte instruction whereas other instructions of the form "INT nn" are 2 byte instructions.

**INT 04 ( Signed number overflow)**

- There is an instruction associated with this INT 0 (interrupt on overflow).
- If INT 0 is placed after a signed number arithmetic as IMUL or ADD the CPU will activate INT 04 if 0F = 1.
- In case where 0F = 0 , the INT 0 is not executed but is bypassed and acts as a NOP.

## Performance of Hardware Interrupts

- NMI : Non maskable interrupts - TYPE 2 Interrupt
- INTR : Interrupt request - Between 20H and FFH




| Interrupt | Priority |
|---|---|
| Divide Error, INT(n),INTO | Highest |
| NMI | |
| INTR | |
| Single Step | Lowest |

TEXT BOOKS:

1. **Microcomputer systems-The 8086 / 8088** Family – Y.C. Liu and G. A. Gibson, 2E PHI - 2003
2. **The Intel Microprocessor, Architecture, Programming and Interfacing**-Barry B. Brey, 6e, Pearson Education / PHI, 2003

REVIEW QUESTION

1. Explain the operation of the following DOS function using INT 21H interrupt. Jan 09 (10 marks)

2. Write a program that clears the screen (monitoring )using MACRO HOME. JAN 09.(10 marks)

3. Describe the working of 8086 in minimum mode configuration with 8284 clock generator jan 09(10 marks)
4. Explain the function of type 0 to type 4 interrupts of 8086 with interrupt vector table. Jan 09 ( 10 marks)

# UNIT: 5-8086 INTERFACING

## Interfacing

## Minimum Mode Interface

- When the Minimum mode operation is selected, the 8086 provides all control signals needed to implement the memory and I/O interface.
- The minimum mode signal can be divided into the following basic groups :

1. **Address/data bus**
2. **Status**
3. **Control**
4. **Interrupt and**
5. **DMA.**

Each and every group is explained clearly.

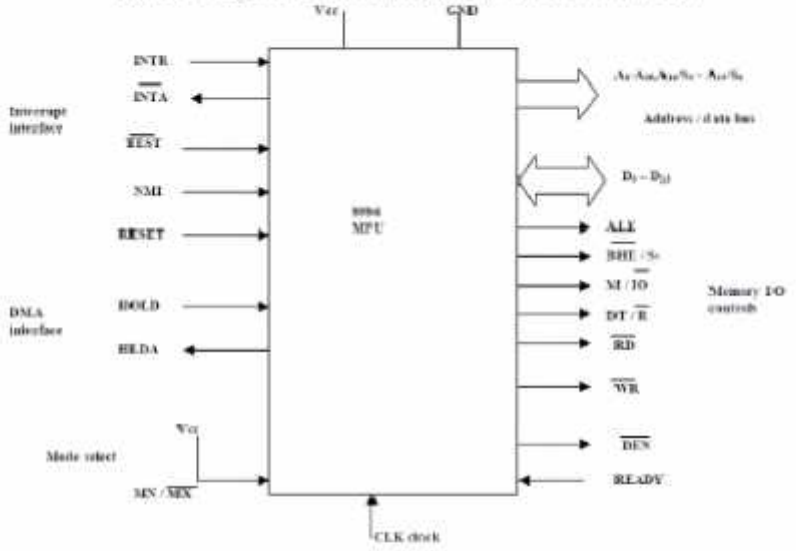
**Address/Data Bus :**

- These lines serve two functions. As an address bus is 20 bits long and consists of signal lines A0 through A19. A19 represents the MSB and A0 LSB. A 20bit address gives the 8086 a 1Mbyte memory address space. More over it has an independent I/O address space which is 64K bytes in length.
- The 16 data bus lines D0 through D15 are actually multiplexed with address lines A0 through A15 respectively. By multiplexed

we mean that the bus work as an address bus during first machine cycle and as a data bus during next machine cycles.

- D15 is the MSB and D0 LSB. When acting as a data bus, they carry read/write data for memory, input/output data for I/O devices, and interrupt type codes from an interrupt controller.

Block Diagram of the Minimum Mode 8086 MPU




**Status signal:**

- The four most significant address lines A19 through A16 are also multiplexed but in this case with status signals S6 through S3. These status bits are output on the bus at the same time that data are transferred over the other bus lines.
- Bit S4 and S3 together from a 2 bit binary code that identifies which of the 8086 internal segment registers are used to generate the physical address that was output on the address bus during the current bus cycle.Code S4S3 = 00 identifies a register known as extra segment register as the source of the segment address.
- Status line S5 reflects the status of another internal characteristic of the 8086. It is the logic level of the internal enable flag. The last status bit S6 is always at the logic 0 level.

| S4 | S3 | Segment Register |
|---|---|---|

| | | |
|---|---|---|
| 0 | 0 | Extra |
| 0 | 1 | Stack |
| 1 | 0 | Code / none |
| 1 | 1 | Data |

Memory segment status codes

**Control Signals :**

- The control signals are provided to support the 8086 memory I/O interfaces. They control functions such as when the bus is to carry a valid address in which direction data are to be transferred over the bus, when valid write data are on the bus and when to put read data on the system bus.
- ALE is a pulse to logic 1 that signals external circuitry when a valid address word is on the bus. This address must be latched in external circuitry on the 1-to-0 edge of the pulse at ALE.
- Another control signal that is produced during the bus cycle is BHE bank high enable. Logic 0 on this used as a memory enable signal for the most significant byte half of the data bus D8 through D1. These lines also serves a second function, which is as the S7 status line.
- Using the M/IO and DT/R lines, the 8086 signals which type of bus cycle is in progress and in which direction data are to be transferred over the bus. The logic level of M/IO tells external circuitry whether a memory or I/O transfer is taking place over the bus. Logic 1 at this output signals a memory operation and logic 0 an I/O operation.
- The direction of data transfer over the bus is signaled by the logic level output at DT/R. When this line is logic 1 during the data transfer part of a bus cycle, the bus is in the transmit mode. Therefore, data are either written into memory or output to an I/O device. On the other hand, logic 0 at DT/R signals that the bus is in the receive mode. This corresponds to reading data from memory or input of data from an input port.

- The signal read RD and write WR indicates that a read bus cycle or a write bus cycle is in progress. The 8086 switches WR to logic 0 to signal external device that valid write or output data are on the bus.

- On the other hand, RD indicates that the 8086 is performing a read of data of the bus. During read operations, one other control signal is also supplied. This is DEN ( data enable) and it signals external devices when they should put data on the bus. There is one other control signal that is involved with the memory and I/O interface. This is the READY signal.

- READY signal is used to insert wait states into the bus cycle such that it is extended by a number of clock periods. This signal is provided by an external clock generator device and can be supplied by the memory or I/O sub-system to signal the 8086 when they are ready to permit the data transfer to be completed.

**Interrupt signals :**

- The key interrupt interface signals are interrupt request (INTR) and interrupt acknowledge ( INTA).
- INTR is an input to the 8086 that can be used by an external device to signal that it need to be serviced.
- Logic 1 at INTR represents an active interrupt request. When an interrupt request has been recognized by the 8086, it indicates this fact to external circuit with pulse to logic 0 at the INTA output.
- The TEST input is also related to the external interrupt interface. Execution of a WAIT instruction causes the 8086 to check the logic level at the TEST input.
- If the logic 1 is found, the MPU suspend operation and goes into the idle state. The 8086 no longer executes instructions, instead it repeatedly checks the logic level of the TEST input waiting for its transition back to logic 0.
- As TEST switches to 0, execution resume with the next instruction in the program. This feature can be used to synchronize the operation of the 8086 to an event in external hardware.
- There are two more inputs in the interrupt interface: the nonmaskable interrupt NMI and the reset interrupt RESET.
- On the 0-to-1 transition of NMI control is passed to a nonmaskable interrupt service routine. The RESET input is used

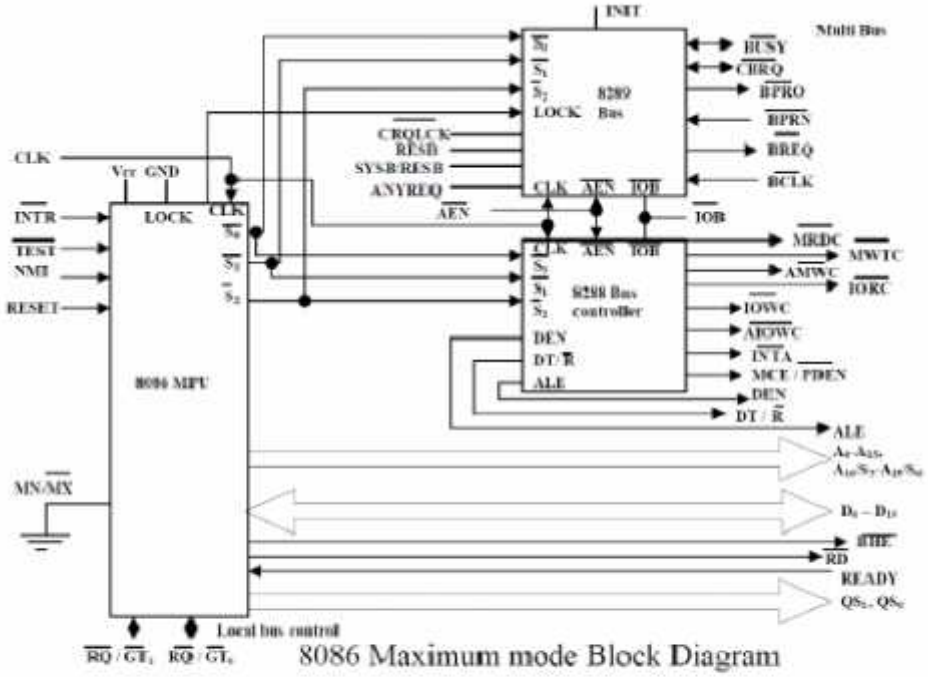
to provide a hardware reset for the 8086. Switching RESET to logic 0 initializes the internal register of the 8086 and initiates a reset service routine.

**DMA Interface signals :**

- The direct memory access DMA interface of the 8086 minimum mode consist of the HOLD and HLDA signals.
- When an external device wants to take control of the system bus, it signals to the 8086 by switching HOLD to the logic 1 level. At the completion of the current bus cycle, the 8086 enters the hold state. In the hold state, signal lines AD0 through AD15, A16/S3 through A19/S6, BHE, M/IO, DT/R, RD, WR, DEN and INTR are all in the high Z state.
- The 8086 signals external device that it is in this state by switching its HLDA output to logic 1 level.

## Maximum Mode Interface

- When the 8086 is set for the maximum-mode configuration, it provides signals for implementing a multiprocessor / coprocessor system environment.
- By multiprocessor environment we mean that one microprocessor exists in the system and that each processor is executing its own program.
- Usually in this type of system environment, there are some system resources that are common to all processors.They are called as global resources. There are also other resources that are assigned to specific processors. These are known as local or private resources.
- Coprocessor also means that there is a second processor in the system. In this two processor does not access the bus at the same time. One passes the control of the system bus to the other and then may suspend its operation.
- In the maximum-mode 8086 system, facilities are provided for implementing allocation of global resources and passing bus control to other microprocessor or coprocessor.




8086 Maximum mode Block Diagram

**8288 Bus Controller – Bus Command and Control Signals:**

- 8086 does not directly provide all the signals that are required to control the memory, I/O and interrupt interfaces.
- Specially the WR, M/IO, DT/R, DEN, ALE and INTA, signals are no longer produced by the 8086. Instead it outputs three status signals S0, S1, S2 prior to the initiation of each bus cycle. This 3- bit bus status code identifies which type of bus cycle is to follow.
- S2S1S0 are input to the external bus controller device, the bus controller generates the appropriately timed command and control signals.

| S2 | S1 | S0 | Indication | 8288 Command |
|---|---|---|---|---|

| | | | | |
|---|---|---|---|---|
| 0 | 0 | 0 | Interrupt Acknowledge | INTA |
| 0 | 0 | 1 | Read I/O port | IORC |
| 0 | 1 | 0 | Write I/O port | IOWC , AIOWC |
| 0 | 1 | 1 | Halt | None |
| 1 | 0 | 0 | Instruction Fetch | MRDC |
| 1 | 0 | 1 | Read Memory | MRDC |
| 1 | 1 | 0 | Write Memory | MWTC, AMWC |
| 1 | 1 | 1 | Passive | None |

- The 8288 produces one or two of these eight command signals for each bus cycles. For instance, when the 8086 outputs the code S2S1S0 equals 001, it indicates that an I/O read cycle is to be performed.
- In the code 111 is output by the 8086, it is signaling that no bus activity is to take place.
- The control outputs produced by the 8288 are DEN, DT/R and ALE. These 3 signals provide the same functions as those described for the minimum system mode. This set of bus commands and control signals is compatible with the Multibus and industry standard for interfacing microprocessor systems.
- The output of 8289 are bus arbitration signals:

Bus busy (BUSY), common bus request (CBRQ), bus priority out (BPRO), bus priority in (BPRN), bus request (BREQ) and bus clock (BCLK).

- They correspond to the bus exchange signals of the Multibus and are used to lock other processor off the system bus during the execution of an instruction by the 8086.
- In this way the processor can be assured of uninterrupted access to common system resources such as global memory.
- Queue Status Signals : Two new signals that are produced by the 8086 in the maximum-mode system are queue status outputs QS0 and QS1. Together they form a 2-bit ueue status code, QS1QS0.

- Following table shows the four different queue status.

| QS1 | QS0 | Queue Status |
|---|---|---|
| 0 (low) | 0 | Queue Empty. The queue has been reinitialized as a result of the execution of a transfer instruction. |
| 0 | 1 | First Byte. The byte taken from the queue was the first byte of the instruction. |
| 1 | 0 | Queue Empty. The queue has been reinitialized as a result of the execution of a transfer instruction. |
| 1 | 1 | Subsequent Byte. The byte taken from the queue was a subsequent byte of the instruction. |

**Table - Queue status codes**

- Local Bus Control Signal – Request / Grant Signals: In a maximum mode configuration, the minimum mode HOLD, HLDA interface is also changed. These two are replaced by request/grant lines RQ/ GT0 and RQ/ GT1, respectively. They provide a prioritized bus access mechanism for accessing the local bus.

# Example Programs of 8086

- Write an 8086 program that adds two packed BCD numbers input from the keyboard and computes and displays the result on the system video monitor
- Data should be in the form 64+89= The answer 153 should appear in the next line.

**Program:**

```
Mov dx, buffer address

Mov ah,0a

Mov si,dx

Mov byte ptr [si], $

Int 21
```

```
Mov ah,0eh
Mov al,0ah
Int 10 ;                BIOS service 0e line feed position cursor
sub byte ptr[si+2], 30h
sub byte ptr[si+3], 30h
sub byte ptr[si+5], 30h
sub byte ptr[si+6], 30h
Mov cl,4
Rol byte ptr [si+3],cl
Rol byte ptr [si+6],cl
Ror word ptr [si+2], cl
Ror word ptr [si+2], cl
Mov al, [si+3]
Add al, [si+6]
Daa
Mov bh,al
Jnc display
Mov al,1
Call display
Mov al,bh
Call display
Int 20
Display Subroutine:
```

```
mov bl,al ;      Save original number and al,f0 ;Force bits 0-3 low
mov cl,4 ;       Four rotates
ror al,cl ;      Rotate MSD into LSD
add al,30 ;      Convert to ASCII
mov ah,0e ;      BIOS video service 0E
int 10 ;         Display character
mov al,bl ;      Recover original number
and al,0f ;      Force bits 4-7 low
add al,30 ;      Convert to ASCII
int 10 ;         Display character
ret ;            Return to calling program
                 ;
                 ;Input buffer begins here
```

# UNIT: 6: 8086 BASED MULTIPROCESSSING SYSTEMS

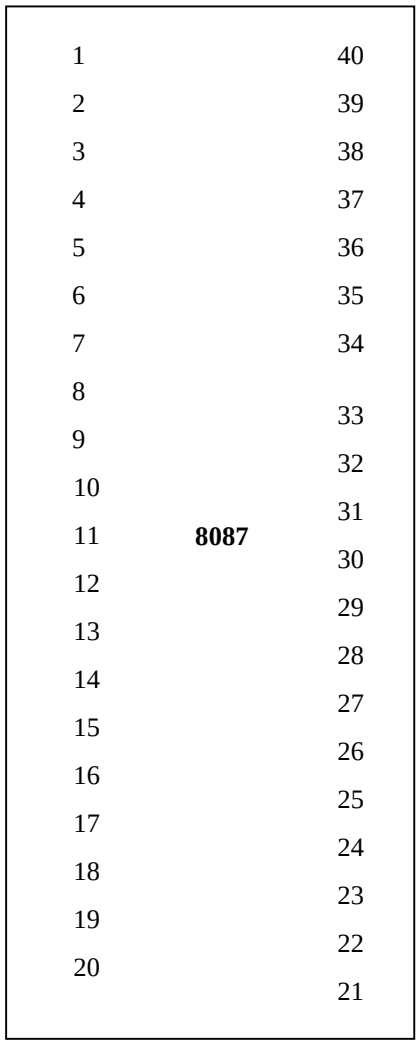
## 8087 Numeric Co-processor

## Need for a numeric co-processor

The 8086 microprocessor is basically an integer processing unit and works directly on a variety of integer data types. Many programs used in engineering, science, business, need to perform mathematical operations like logarithms of a number, square root of a number, sine of an angle etc. It may also be needed to perform computations with very large numbers like $10^{+56}$, or very small numbers like $10^{-67}$. There are no instructions in 8086 to directly find sine of an angle etc. Also 8086 can only perform computations on 16 bit fixed point numbers, with a range of –32768 to +32767. In other words, 8086 does not provide any intrinsic support for operations on floating point numbers.

It is possible to perform any calculations using only 8086. But if speed becomes important, it is necessary to use the dedicated Numeric co-processor Intel 8087, to speed up the matters. It typically provides a 100 fold speed increase for floating point operations. A numeric co-processor is also variously termed as arithmetic co-processor, math co-processor, numeric processor extension, numeric data processor, floating point processor etc.

## 8087 Pin diagram




## Description of 8087 pins

**INT:** This is an active high output pin. The 8087 activates this pin whenever an exception occurs during 8087 instruction execution, provided the 8087 interrupt system is enabled and the relevant exceptions is not masked using the 8087 control register.

The INT output of 8087 is connected directly to NMI or INTR input of 8086. Alternatively, INT output of 8087 is connected to an interrupt request input of 8259 Interrupt controller, which in turn interrupts the 8086 on its INTR input.

**BUSY:** Let us say, the 8086 is used in maximum mode and is required to wait for some result from the co-processor 8087 before proceeding with the next instruction. Then we can make the 8086 execute the WAIT instruction. Then the 8086 enters an idle state, where it is not performing any processing. The 8086 will stay

in this idle state till TEST* input of 8086 is made 0 by the co-processor, indicating that the co-processor has finished its computation.

When the 8087 is busy executing an arithmetic instruction, its BUSY output line will be in the 1 state. This pin is connected to TEST*pin of 8086. Thus when the BUSY pin is made 0 by the 8087 after the completion of execution of an arithmetic instruction, the 8086 will carry on with the next instruction after the WAIT instruction.

## Internal Structure of the 80X87

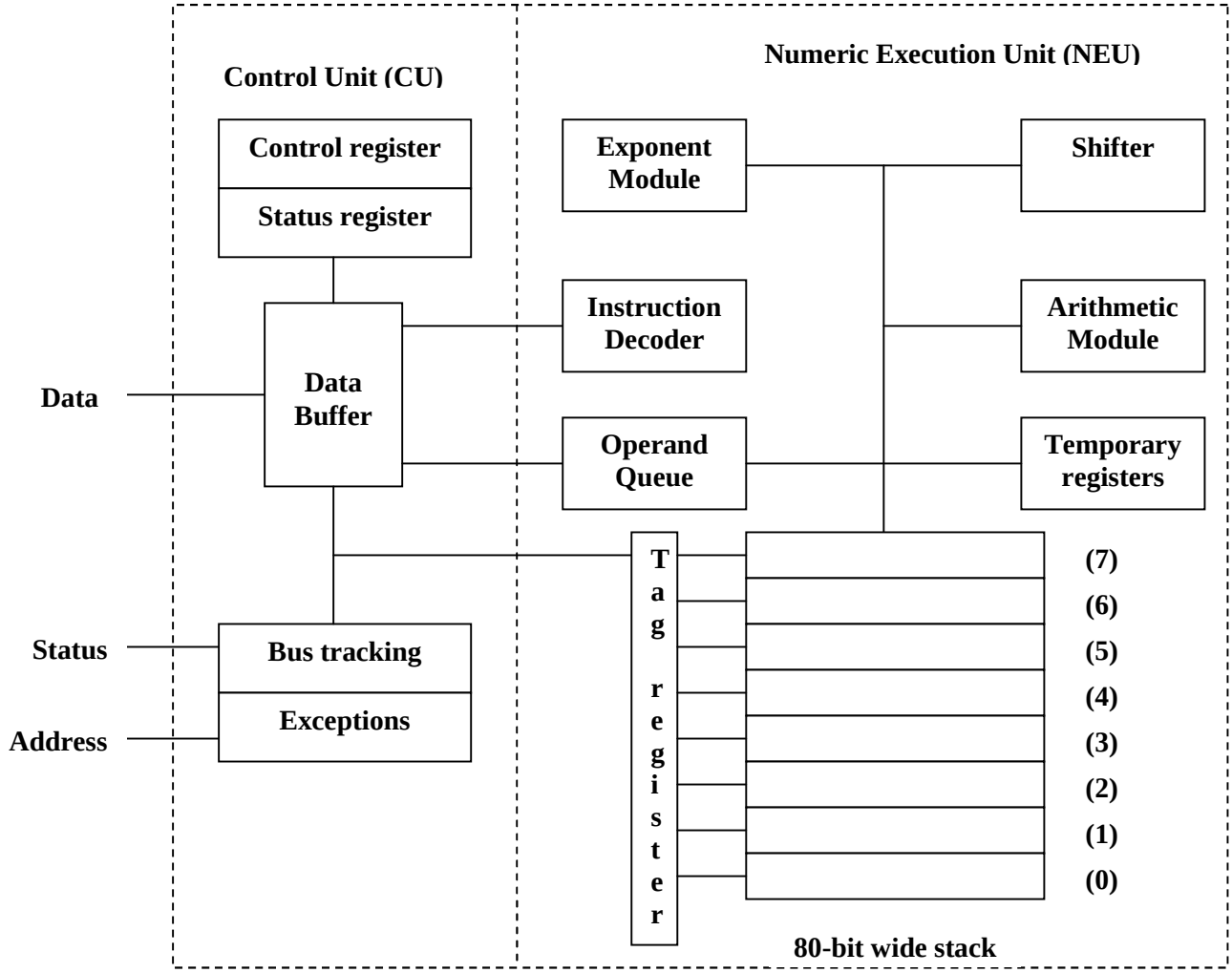



**Fig: The internal structure of the 80X87 arithmetic coprocessor**

## 8087 Data Types

The 8087 always works on 80 bit data internally. This 80 bit floating point format is termed as Temporary Real format. However, it can read from memory a number, which is represented using any of the following data types.

a. Signed integers of size 16, 32 or 64 bits

b. 18 digit signed integer packed BCD number using 80 bits
c. Floating point numbers using 32, 64, or 80 bits

This number read from memory is internally converted to the 80 bit temporary real format before performing any computations. Similarly, the result is converted automatically by the 8087 to one of the formats mentioned above before storing it in memory.

**8087 Data Types:**

**1. Integer Data Types**

### (a) Word integer (16 Bit Signed Integer)

| S | Magnitude |
|---|---|

15 0

Sign bit is 0 for positive and 1 for negative.

Range: –32768<=X<=+32767. Negative number representation in 2's complement form.

### (b) Short integer (32 Bit Signed Integer)

| S | Magnitude |
|---|---|

31 0

Range: $-2 \times 10^9 <= X <= 2 \times 10^9$

### (c) Long Integer (64 Bit Signed Integer)

| S | Magnitude |
|---|---|

63 0

This is called binary integer. Range: $-9 \times 10^{18} <= X <= 9 \times 10^{18}$

**2. Packed BCD type**

**Packed Decimal (18 BCD digits)**

| S | Don't care | Magnitude (BCD) |
|---|---|---|

79 72 71 0

-99… … 99<=X<=+99… …99(18 digits)

**3. 32 Bit Short real**

**Short real (Single precision)**

| S | Biased exponent | Significant |
|---|---|---|
| 31 | 23 | 0 |

0, 1, $2x10^{-38}$ <=! X! <=$3.4x10^{38}$

**Example 1:**

Let us say, we want to represent 23.25 in this the short real notation. First of all we represent 23.25 in binary as 10111.01. Then we represent this as $+1.011101x2^{+4}$. This is called the Normalized form of representation. In the normalized form, the mantissa will always have an integer part with value 1. The floating point notations supported by 8087 always represent a number in the normalized form. In the 32 bit and 64 bit floating point notations the integer part of mantissa, of value 1, is just implied to be present, but not explicitly indicated in the bit pattern for the number. Thus the LS 23 bits are used to indicate only the fractional part of the mantissa and so will be 011 1010 0000 0000 0000 0000. The MS bit will be 0 to indicate that the number is positive. The next 8 bits provide the exponent in excess 7FH format. Thus the next 8 bits will be 4 + 7F=83H = 1000 0011. Thus the 32 bit floating point representation for 23.25 will be

| **sign** | **Exp. In Ex 7FH** | **23 bit fractional part of mantissa** |
|---|---|---|
| 0 | 1000 0011 | 011 1010 0000 0000 0000 0000 |

**Example 2:**

Now let us see what is the value of the 32 bit floating point number 10111 1100 100 0000 0000 0000 0000 0000. It has its MS bit as a 1. Thus the number is negative. The next 8 bits are 0111 1100 = 7CH. Thus 7CH is the exponent in excess 7FH format. In other words, the actual exponent is 7CH-7FH=-03. the actual mantissa is obtained by appending 1. to the LS 23 bits. Thus the actual mantissa is 1.100 0000 0000 0000 0000 0000. Thus the value of the given 32 bit floating point number would be

$-1.100\ 0000\ 0000\ 0000\ 0000\ 0000 \times 2^{-03}$

$= -1.1 \times 2^{-03}$

$= -0.0011 \times 2^{0}$

$= -0.0011$

$= -0.1875$

Thus the given 32 bit number represents the value –0.1875

**4. 64 bit Long Real**

**Long Real (Double precision)**

| S | Biased exponent | Significand |
|---|---|---|
| 63 | 52 | 0 |

$0,2,3 \times 10^{-308}$ !X!<= $1.7 \times 10^{308}$

In both single and double precision cases the 1 after . is assumed to be present.

| | | |
|---|---|---|
| Sign<br>0 = +<br>1 = - | 11 bits exponent in Ex3FFH | 52 bits for fractional part with implied '1. ' before the fractional part. |

**Example 1:**

Let us say, we want to represent 23.255 in this notation. First of all we represent 23.25 in binary as 10111.01. Then we represent this as $+1.011101 \times 2^{+4}$. This is called the Normalized form of representation. In the normalized form, the mantissa will always have an integer part with value 1. The floating point notations supported by 8087 always represent a number in the normalized form. In the 32 bit and 64 bit floating point notations the integer part of the mantissa, of value 1, is just implied to be present, but not explicitly indicated in the bit pattern for the number. Thus the LS 52 bits are used to indicate only he fractional part of the mantissa and so will be 0111 0100 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000. The MS bit will be 0 to indicate that the number is positive. The next 11 bits provide the exponent in excess 3FFH format. Thus the next 11 bits will be 4+3FF=403H=100 0000 0011. Thus the 64 bit floating point representation for 23.25 will be

| **sign** | **Exp. In Ex 7FH** | **52 bit fractional part of mantissa** |
|---|---|---|
| 0 | 100 0000 0011 | 0111 0100 00…….00 |

**Example 2:**

Now let us see what is the value of the 64 bit floating point number 1 100 0000 0011 0100 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000. It has its MS bit as a 1. Thus the number is negative. The next 11 bits are 100 0000 0011 = 403H. Thus 403H is the exponent in excess 3FFH format. In other words, the actual exponent is 403H – 3FFH=+04. The actual mantissa is obtained by appending 1. to the LS 52 bits. Thus the actual mantissa is 1.0100 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000. Thus the value of the given 64 bit floating point number would be

$$\begin{aligned} & -1.0100\ 0000\ \ldots\ 0000 \times 2^{+04} \\ = \quad & -1.01 \times 2^{+04} \\ = \quad & -10100 \times 2^{0} \\ = \quad & -10100 \\ = \quad & -20 \end{aligned}$$

Thus the given 64 bit number represents the value –20.

### 5. Temporary Real

| S | Biased exponent | 1 | Significand |
|---|---|---|---|
| 79 | 64 | 63 | 0 |

$0,3.4x10^{-4932}$ <= !X! <= $1.1x10^{4932}$

**Example 1:**

Let us say, we want to represent 23.25 in this notation. First of all we represent 23.25 in binary as 10111.01. Then we represent this as $+1.011101 \times 2^{+4}$. This is called the normalized form of representation. In the normalized form, the mantissa will always have an integer part with value 1. The floating point notations supported by 8087 always represent a number in the normalized form.

Thus the LS 64 bits are used to indicate the mantissa and so will be 1011 1010 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000 0000. The MS bit will be 0 to indicate that the number is positive. The next 15 bits provide the exponent in excess 3FFFH format. Thus the next 15 bits will be 4+3FFF = 4003H = 100 0000 0000 0011. Thus the 80 bit floating point representation for 23.25 will be

| sign | Exp. In Ex. 3FFFH | 64 bit mantissa |
|---|---|---|
| 0 | 100 0000 0000 0011 | 1011 1010 00 …….00 |

**Example 2:**

Now let us see what is the value of the 64 bit floating point number 1 100 0000 0000 0011 1010 0000 …. 0000. It has its MS bit as a 1. Thus the number is negative. The next 15 bits are 100 0000 0000 0011 = 4003H. Thus 4003H is the exponent in excess 3FFFH format. In other words, the actual exponent is 4003H-3FFFH=+04. The actual mantissa is 1.010 0000 …. 0000, where the binary point is implied to be present after the MS bit of the mantissa. Thus the value of the given 80 bit floating point number would be

$= -1.010\ 0000 \ldots 0000 \times 2^{+04}$

$= -1.01 \times 2^{+04}$

$= -10100 \times 2^{0}$

$= -10100$

$= -20$

Thus the given 80bit number represents the value –20.

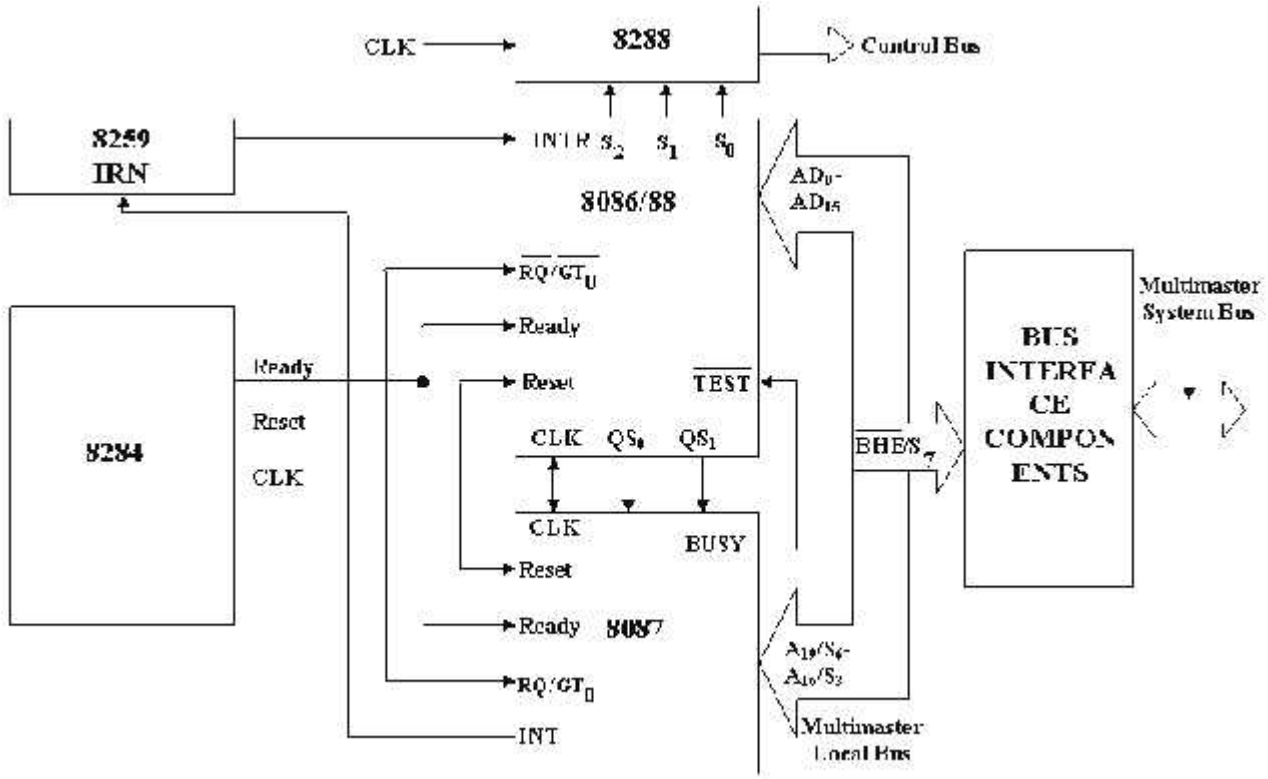
## 8087 Data types in a nut shell

| **Data format** | **Range** | **Precision** | **7 0!7 0!7 0!7 0!7 0!7 0!7 0!7 0!7 0!7 0!** |
|---|---|---|---|
| Word integer | $10^4$ | 16 bits | $1_{15}$ $1_0$ two's complement |

| | | | |
|---|---|---|---|
| Short integer | $10^4$ | 32 bits | $1_{31}$ $1_0$ two's complement |
| Long integer | $10^{18}$ | 64 bits | $1_{63}$ $1_0$ two's complement |
| Packed BCD | $10^{18}$ | 18 digits | S $D_{17}$ $D_{16}$ $D_0$ |
| Short real | 10+38 | 24 bits | $SE_7$ $E_0$ $F_1$ $F_{23}$ $F_0$ implicit |
| Long real | 10+308 | 53 bits | $SE_{10}$ $E_0$ $F_1$ $F_{52}$ $F_0$ implicit |
| Temporary real | 10+4932 | 64 bits | $SE_{14}$ $E_0$ $F_0$ $F_{63}$ |

- Integer : 1
- Packed BCD : (-1)S (D17 … D0)
- Real : (-1)S (2E-Bias) (F0.F1..)
- Bias = 127 for short Real
  = 1023 for long Real
  = 16383 for Temp. Real

## Interconnection of 8087 with 8086/88




8087 can be connected with any of the 8086/8088/80186/80188 CPU's only in their maximum mode of operation. I.e. only when the MN/MX* pin of the CPU is grounded. In maximum mode, all the control signals are derived using a separate chip known as bus controller. The 8288 is 8086/88 compatible bus controller while 82188 is 80186/80188 compatible bus controller.

The BUSY pin of 8087 is connected with the TEST* pin of the used CPU. The $QS_0$ and $QS_1$ lines may be directly connected to the corresponding pins in case of 8086/8088 based systems. However, in case of 80186/80188 systems these $QS_0$ and $QS_1$ lines are passed to the CPU through the bus controller. In case of 8086/8088 based systems the RQ*/$GT_0$* of 8087 may be connected to RQ*/$GT_1$* of the 8086/8088. The clock pin of 8087 may be connected with the CPU 8086/8088 clock input. The interrupt output of 8087 is routed to 8086/8088 via a programmable interrupt controller. The pins $AD_0$ - $AD_{15}$, BHE*/$S_7$, RESET, $A_{19}$ / $S_6$ - $A_{16}$ / $S_3$ are connected to the corresponding pins of 8086/8088. In case of 80186/80188 systems the RQ/GT lines of 8087 are connected with the corresponding RQ*/GT* lines of 82188. The interconnections of 8087 with 8086/8088 and 80186/80188 are shown in fig.

## Control Register of 8087

In addition to the 8 registers, which are 80 bits wide, the 8087 has a control register, a status register, and a Tag register each 16 bits wide.

The contents of the control register, generally referred to as the Control word, direct the working of the 8087. A common way of loading the control register from a memory location is by executing the instruction 'FLDCW src', where 'src' is the address of a memory location. FLDCW stands for 'Load Control Word'. For example, FLDCW [BX] instruction loads the control register of 8087 with the contents of the memory location whose 16 bit effective address is provided in BX register.

The bit description of the control register is shown below.

| **Bit No** | **15** | **14** | **13** | **12** | **11** | **10** | **9** | **8** | **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | **reserved** | | | **I C** | **Round ctrl** | | **Prec. ctrl** | | **Intr mask** | **x** | **P M** | **U M** | **O M** | **Z M** | **D M** | **I M** |

The LS 6 bits are used for individually masking 6 possible numerical error exceptions. If an exception is masked, by setting the corresponding bit to 1, the 8087 will handle the exception internally. It does not set the corresponding exception bit in the status register and it does not generate an interrupt request. This is termed the Masked response.

They LS 6 bits, which correspond to the exception mask bits, are briefly described below.

IM bit (Invalid operation Mask) at bit position 0 is used for masking invalid operation. An invalid operation exception generally indicates a stack overflow or underflow error, or an arithmetic error like, divisor is 0 or dividend is infinity.

DM bit (Denormalized operand mask) at bit position 1 is used for masking denormalized operand exception. A denormalized result occurs when there is a floating point underflow. Thus, this exception occurs, for example, when an attempt is made to load a denormalized operand from memory.

ZM bit (Zero divide mask) at bit position 2 is used for masking zero divide exception. This exception occurs when an attempt is made to divide a valid non zero operand by zero. This can happen in the case of explicit division instructions as well as for operations that perform division internally like in FXTRACT.

OM bit (Overflow exception Mask) at bit position 3 is used for masking overflow exception. A overflow exception occurs when the exponent of the actual result is too large for the destination.

UM bit (Underflow exception Mask) at bit position 4 is used for masking underflow exception. An underflow exception occurs when the exponent of the actual result is too small for the destination.

PM bit (Precision exception Mask) at bit position 5 is used for masking precision exception. A precision exception occurs when the result of an operation loses significant digits when stored in the destination.

**Precision control bits (bits 9 and 8)**

These bits control the internal operating precision of the 8087. Normally, the 8087 uses 64 bit mantissa for all internal calculations. However, this can be reduced to 53 or 24 bits, for compatibility with earlier generation math processors, as shown below.

| Bit 9 | Bit 8 | Length of mantissa |
|---|---|---|
| 0 | 0 | 24 bits |
| 0 | 1 | Reserved |
| 1 | 0 | 53 bits |
| 1 | 1 | 64 bits |

**Rounding control bits (bits 11 and 10)**

These bits control the type of rounding that is used in calculations, as shown below.

| Bit 11 | Bit 10 | Rounding scheme |
|---|---|---|
| 0 | 0 | Round to nearest |
| 0 | 1 | Round down, towards -∞ |
| 1 | 0 | Roundup, towards +∞ |
| 1 | 1 | Chop or truncate towards 0 |

**Infinity control bit (bit 12)**

This bit controls the way infinity is treated. In the affine model of infinity, + ∞ and - ∞ are treated as a single unsigned quantity.

| Bit 12 | Infinity model |
|---|---|
| 0 | Projective |
| 1 | Affine |

**Contents of Control register after reset of 8087**

When the 8087 is reset, the control register is loaded with 037FH = 000 0 00 11 0 1 11 1111, which means the following. The same condition results when FINIT (stands for Initialize) instruction is executed.

This condition is generally acceptable to a programmer. So, there is normally no need to explicitly load the control register using FLDCW instruction.

## Status register of 8087

The status register is 16 bits wide. The contents of the status register, generally referred to as the Status word, indicates the status of the 8087. A common way of storing the contents of the status register into a memory location is by executing the instruction ‘FSTSW dst’, where ‘dst’ is the address of a memory

location. FSTSW stands for Store Status Word'. For example, FSTSW [BX] instruction stores the status register of 8087 into the memory location whose 16 bit effective address is provided in BX register. This status can then be read by the 8086, by executing say MOV AX, [BX], to take action depending on the status of 8087.

The bit description of the status register is shown below.

| **Bit no.** | **15** | **14** | **13** | **12** | **11** | **10** | **9** | **8** | **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Busy | C3 | Stack pointer | | | C2 | C1 | C0 | Intr Req | x | PE | UE | OE | ZE | DE | IE |

If the 8087 encounters an error exception during execution of an instruction, the corresponding exception bit is set to the 1 state, if the exception is not masked using the control word. The possible exceptions, as already discussed, are as follows.

Invalid operation Exception (IE, bit 0 of the status register)

Denormalized operand Exception (DE, bit 1 of status register)

Zero divide Exception (ZE, bit 2 of status register)

Overflow Exception (OE, bit 3 of status register)

Underflow Exception (UE, bit 4 of status register)

Precision Exception (PE, bit 5 of status register)

The only way these exception bits are cleared is by the execution of FINIT, FCLEX (stands for clear exceptions), FLDENV (stands for load environment), FSAVE (stands for save environment and stack of registers), and FRSTOR (stands for restore environment and stack of registers). The term Environment stands for the following group of information of size 14 bytes.

1. control word (2 bytes)
2. Status word (2 bytes)
3. Tag word (2 bytes)
4. Exception pointer (8 bytes)

The interrupt request bit (bit 7) in the status word is set to 1 by the 8086, if one or more exception bits are set to 1. Then the INT output pin of 8087 is activated if interrupt is not masked using the control word.

C2, C2, C1, and C0 (bits 14, 10, 9, and 8) are the condition code flags of the 8087. The 8087 updates these flags depending on the status of arithmetic operations. The FTST (stands for Test) and FCOM (stands for Compare) instructions also use these flags to report the result of their operations. Some of these bits are discusses later when the Compare instruction is described.

Bits 13, 12, and 11 provide the address of the register which is currently the stack top. For example, if these bits are 110, it means that R6 is the current stack top. In other words, ST is R6, ST(1) is R7, ST(2) is R0, and so on.

The Busy bit (bit 15) is set to 1 when the 8087 is busy executing an instruction, or busy executing an exception routine. When this bit is a 1, the BUSY output pin of 8087 is activated.

A programmer needs to read the status register contents after the execution of FTST or FCOM instruction, to know the result of these instructions. In most of other cases, the programmer is not required to read the status register contents.

## Exception Pointer of 8087

When the 8086 comes across an 8087 instruction, it saves the following information in a 4 word area termed as the exception pointer.

1. 20 bit physical address of the instruction
2. 11 bit opcode of the instruction
3. 20 bit physical address of the data, if 8087 needs it.
4. Remaining 13 bits are zeros.

However, some instructions like FLDCW which need a memory operand, do not affect the 20 bit area of the exception pointer meant for address of data.

The exception pointer is located in the 8086, and not in 8087, but appears to be part of 8087.

## Tag register of 8087

The Tag register is 16 bits wide. The contents of the Tag register indicates the status of each of the 80 bit registers of the 8087. A common way of storing the contents of the Tag register is by executing the instruction 'FSTENV dst', where 'dst' is the address of a memory location. It stores the environment of 8087, of which Tag word is a part. FSTENV stands for 'Store environment'. For example, FSTENV [BX] instruction stores the environment of 8087 into 14 byte memory locations whose 16 bit effective address is provided in BX register.

The Tag register is loaded with a new value, when one of FINIT, FLDENV, or FRSTOR instructions are executed.

The bit description of the Tag register is as shown below.

| **Bit no.** | **15** | **14** | **13** | **12** | **11** | **10** | **9** | **8** | **7** | **6** | **5** | **4** | **3** | **2** | **1** | **0** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | TAG 7 | | TAG 6 | | TAG 5 | | TAG 4 | | TAG 3 | | TAG 2 | | TAG 1 | | TAG 0 | |

The status of each 80 bit stack register is provided using a 2 bit field in the Tag register. The field labeled TAG 3 indicates the status of R3. It should be noted that TAG 3 is not indicating the status of ST(3). The Tag bits indicate the status of a stack register as shown below.

| **Tag bits** | **Status** |
|---|---|
| 00 | Valid data in the register |

| | |
|---|---|
| 01 | Zero value in the register |
| 10 | Special number, like _ or decimal, in the register |
| 11 | The register is empty |

The Tag word is not normally used in programs. However it can be used to quickly interpret the contents of a floating point register, without the need for extensive decoding.

## FINIT instruction

- Infinity condition is projective (treats + ∞ and - ∞ as same)
- Rounds to nearest
- Length of mantissa is 64 bits
- Interrupt is enabled
- All exceptions are masked
- No need for FLDCW

## 8087 Instruction Set

The instruction set of 8087 starts with F, stands for floating point. The instruction of 8087 numeric data processor can be classified into following six groups:

1. Data transfer instructions
2. Arithmetic instructions
3. Compare Instructions
4. Transcendental instructions
5. Load constant instructions
6. Processor control instructions

## 1. Data Transfer Instructions

**(a) Real Transfers**

| S. No. | Instruction | Description with example |
|---|---|---|
| 1 | FLD source | Decrements stack pointer by one and copies a real number from a stack element or memory to the new ST. A short – real or long-real number from memory is automatically converted to temporary real format by the 8087 before it is put in ST.<br>Examples:<br>FLD ST(2) ; Copies ST(2) to ST |

| | | |
|---|---|---|
| | | FLD [BX] ; Number from memory pointed by BX copied to ST |
| 2 | FST Destination | Copies ST to a specified stack position or to a specified memory location.<br>Examples:<br>FST ST(3) ; Copy ST to ST(3)<br>FST [BX] ; Copy ST to memory pointed by [BX] |
| 3 | FSTP destination | Copies ST to a specified stack element or memory location and increments stack pointer by one to point to the next element on the stack. This is a stack POP operation. |
| 4 | FXCH destination | Exchanges contents of ST with the contents of a specified stack element. If no destination is specified, then ST(1) is used.<br>Example:<br>FXCH ST(4) ; Swap ST and ST(4) |

**(b) Integer transfers**

| S. No. | Instruction | Description with example |
|---|---|---|
| 5 | FILD source | Integer load. Converts integer number from memory to temporary real format and pushes converted number on 8087 stack.<br>Example:<br>FILD DWORD PTR [BX] ; Short integer from memory location pointed by [BX] |
| 6 | FIST destination | Integer store. Converts number from ST to integer form, and copies to memory.<br>Example:<br>FIST INT_NUM ; ST to memory locations named INT_NUM |
| 7 | FISTP destination | Integer store and pop. Similar to FIST except that stack pointer is incremented after copy. |

**(c) Packed Decimal Transfers**

| S. No. | Instruction | Description with example |
|---|---|---|
| 8 | FBLD source | Packed decimal (BCD) load. Convert number from memory to temporary-real format and push on top of 8087 stack.<br>Example:<br>FBLD AMOUNT ; Ten byte BCD number from memory |

| | | |
|---|---|---|
| | | location AMOUNT to ST |
| 9 | FBSTP destination | BCD store in memory and pop 8087 stack. Pops temporary – real from stack, converts to 10-byte BCD, and stores result to memory.<br>Example:<br>FBSTP MONEY ; Contents from top of stack are converted to BCD, and stored in memory. |

## 2. Arithmetic Instructions

| S. No. | Instruction | Description with example |
|---|---|---|
| 1 | FADD destination, source | Will add real number from specified source to real number at specified destination. Source can be stack element or memory location. Destination must be a stack element. If no source or destination is specified, then ST is added to ST(1) and the stack pointer is incremented so that the result of the addition is at ST.<br>Examples:<br>FADD ST(2), ST ; Add ST to ST(2), result in ST(2)<br>FADD ST, ST(5) ; Add ST(5) to ST, result in ST<br>FADD SUM ; Real number from memory + ST<br>FADD ; ST + ST(1), pop stack-result at ST |
| 2 | FADDP destination, source | Adds ST to specified stack element and increments stack pointer by one.<br>Example:<br>FADDP ST(2) ; Add ST(2) to ST<br>; Increment stack pointer so ST(2)<br>; becomes ST |
| 3 | FIADD source | Adds integer from memory to ST, Stores the result in ST.<br>Example:<br>FIADD CARS_SOLD ;Integer number from memory + ST |
| 4 | FSUB destination, source | Subtracts the real number at the specified source from the real number at the specified destination and puts the result in the specified destination.<br>Examples:<br>FSUB ST(3), ST ; ST(3)← ST(2) – ST<br>FSUB DIFFERENCE ; ST←ST-real from memory<br>FSUB ; ST←(ST(1)-ST) |

| | | |
|---|---|---|
| 5 | FSUBP destination, source | Subtracts ST from specified stack element and puts result in specified stack element. Then increments stack pointer by one.<br>Examples:<br>FSUBP ST(2) ; ST(2) – ST . ST(1) becomes new ST. |
| 6 | FISUB source | Subtracts integer number stored in memory from ST and stores result in ST.<br>Example:<br>FISUB DIFFERENCE ; ST←ST-integer from memory |
| 7 | FSUBR destination, source | These instructions operate same as FSUB instructions discussed earlier except that these instructions subtract the contents of the specified destination from the contents of the specified source and put the difference in the specified destination.<br>[Normal FSUB instruction subtracts source from destination.] |
| 8 | FSUBRP destination, source | |
| 9 | FISUBR source | |
| 10 | FMUL destination, source | Multiply real number from source by real number from specified destination, and put result in specified stack element.<br>Examples:<br>FMUL ST(2), ST ; Multiply ST(2) and ST, result in ST(2)<br>FMUL ST, ST(5) ; Multiply ST(5) to ST, result in ST |
| | FMULP destination, source | Multiplies the real number from specified source by real number from specified destination, puts result in specified stack element, and increment stack pointer by one. With no specified operands FMULP multiplies ST(1) by ST and Pops stack to leave result at ST.<br>Example:<br>FMULP ST(2) ; Multiply ST(2) to ST. increment stack pointer so STI(1) becomes ST |
| 11 | FIMUL source | Multiply integer from memory at ST and put result in ST.<br>Example:<br>FIMUL DWORD PTR [BX]<br>;Integer number from memory pointed by BX x ST and result in ST |
| 12 | FDIV destination, source | Divides destination real by source real, stores result in destination.<br>Example:<br>FDIV ST(2), ST ; Divides ST by ST(2)<br>; stores result in ST |
| 13 | FDIVP destination, | Same as FDIV, but also increments stack pointer by one after |

| | | |
|---|---|---|
| | source | DIV<br>Example:<br>FDIV ST(2), ST ; Divides ST by ST(2), stores result in ST and increments stack pointer |
| 14 | FIDIV source | Divides ST by integer from memory, stores result in ST.<br>Example:<br>FIDIV PERCENTAGE; ST←ST/integer number |
| 15 | FDIVR destination, source | |
| 16 | FDIVP destination, source | |
| 17 | FIDIVR source | These three instructions are identical in format to the FDIV, FDIVP and FIDIV instructions above except that they divide the source operand by the destination operand and put the result in the destination. |
| 18 | FSQRT | Contents of ST are replaced with its square root.<br>Example:<br>FSQRT |
| 19 | FSCALE | Scales the number in ST by adding an integer value in ST(1) to the exponent of the number in ST. Fast way of multiplying by integral powers of two. |
| 20 | FPREM | Partial reminder. The contents of ST(1) are subtracted from the contents of ST over and over again until the contents of ST are smaller than the contents of ST(1)<br>Example:<br>FPREM |
| 21 | FRNDINT | Round number in ST to an integer. The round – control (RC) bits in the control word determine how the number will be rounded. |
| 22 | FXTRACT | Separates the exponent and the significant parts of a temporary real number in ST. After the instruction executes, ST contains a temporary – real representation of the significant of the number and ST(1) contains a temporary real representation of the exponent of the number. |
| 23 | FABS | Replaces ST by its absolute value. Instruction simply makes sign positive. |
| 24 | FCHS | Complements the sign of the number in ST. |

## 3. Compare Instructions

These instructions compare the contents of ST with contents of specified or default source. The source may be another stack element or real number in memory. Such compare instructions set the condition code bits C3, C2 and C0 of the status words use as shown in the table below.

| C3 | C2 | C0 | Description |
|---|---|---|---|
| 0 | 0 | 0 | ST contents is greater than the other operand |
| 0 | 0 | 1 | ST contents is smaller than the other operand |
| 1 | 0 | 0 | ST contents is equal to the other operand |
| 1 | 1 | 1 | The operands are not comparable |

**Different compare instructions:**

| S. No. | Instruction | Description with example |
|---|---|---|
| 1 | FCOM source | Compares ST with real number in another stack element or memory.<br>Examples:<br>FCOM ; Compares ST with ST(1)<br>FCOM ST(4) ; Compares ST with ST(4)<br>FCOM VALUE ; Compares ST with real number from memory |
| 2 | FCOMP source | Identical to FCOM except that the stack pointer is incremented by one after the compare operation. |
| 3 | FCOMPP | Compares ST with ST(1) and increments stack pointer by 2 after compare. |
| 4 | FICOM source | Compares ST to a short or long integer from memory. |
| 5 | FICOMP source | Identical to FICOM except stack pointer is incremented by one after compare. |
| 6 | FTST | Compares ST with zero. |
| 7 | FXAM | Tests ST to see if it is zero, infinity, unnormalized, or empty. Sets bits $C_3$, $C_2$, $C_1$ and $C_0$ to indicate result. |

## 4. Transcendental Instructions (Trigonometric and Exponential Instructions)

| S. No. | Instruction | Description with example |
|---|---|---|
| 1 | FPTAN | Computes the values for a ratio of Y/X for an angle in ST. the angle must be expressed in radians, and the angle must be in the |

| | | |
|---|---|---|
| | | range of 0 < angle < π/4.<br>(FPTAN does not work correctly for angles of exactly 0 and π/4.) |
| 2 | FPATAN | Computes the angle whose tangent is Y/X. The X value must be in ST, and the Y value must be in ST(1). Also X and Y must satisfy the inequality 0 < Y < X < ∞. The resulting angle expressed in radians replaces Y in the stack. After the operation the stack pointer is incremented so the result is then ST. |
| 3 | F2XM1 | Computes the function $Y = 2^X - 1$ for an X value in ST. the result, Y replaces X in ST. X must be in the range $0 \le X \le 0.5$ |
| 4 | FYL2X | Calculates Y times the log to the base 2 of X or Y ($\log_2 X$). X must be in the range of 0 < X < ∞ and Y must be in the range -∞ < Y < +∞. X must initially be in ST and Y must be in ST(1). The result replaces Y and then the stack is popped so that the result is then at ST. |
| 5 | FYL2XP1 | Computes the function Y times the log to the base 2 of (X+1) or Y ($\log_2$ (X+1)). This instruction is almost identical to FYL2X except that it gives more accurate results when computing the logoff a number very close to one. |

## 5. Load constant Instructions

| S. No. | Instruction | Description |
|---|---|---|
| 1 | FLDZ | - Push 0.0 onto stack |
| 2 | FLDI | - Push + 1.0 onto stack |
| 3 | FLDPI | - Push the value π onto stack |
| 4 | FLD2T | - Push log of 10 to the base 2 onto stack ($\log_2 10$) |
| 5 | FLDL2E | - Push log of e to the base 2 onto stack ($\log_2 e$) |
| 6 | FLDLG2 | - Push log of 2 to the base 10 onto stack ($\log_{10} 2$) |

Note: The load constant instruction will just push indicated constant into the stack.

## 6. Processor Control Instructions

| S. No. | Instruction | Description |
|---|---|---|
| 1 | FINIT/FNINT | Initializes 8087. Disables interrupt output, sets stack pointer to register 7, sets default status. |
| 2 | FDISI/FNDISI | Disables the 8087 interrupt output pin so that it can not |

| | | |
|---|---|---|
| | | cause an interrupt when an exception (error) occurs. |
| 3 | FENI/FNENI | Enables 8087 interrupt output so it can cause an interrupt when an exception occurs. |
| 4 | FLDCW source | Loads a status word from a memory location into the 8087 status register. This instruction should be preceded by the FCLEX instruction to prevent a possible exception response if an exception bit in the status word is set. |
| 5 | FSTCW/FNSTCW destination | Copies the 8087 control word to a memory location. You can determine its current value with 8086 instructions. |
| 6 | FSTSW/FNSTW destination | Copies the 8087 status word to a memory location. You can check various status bits with 8086 instructions and take further action on the state of these bits. |
| 7 | FCLEX/FNCLEX | Clears all of the 8087 exception flag bits in the status register. Unasserts BUSY and INT outputs. |
| 8 | FSAVE/FNSAVE destination | Copies the 8087 control word, status word, pointers and entire register stack to 94-byte area of memory. After copying all of this the FSAVE/FNSAVE instruction initializes the 8087. |
| 9 | FRSTOR source | Copies a 94 byte area of memory into the 8087 control register, status register, pointer registers, and stack registers. |
| 10 | FSTENV / FNSTENV destination | Copies the 8087 control register, status register, tag words, and exception pointers to a series of memory locations. This instruction does not copy the 8087 register stack to memory as the FSAVE / FNSAVE instruction does. |
| 11 | FLDENV source | Loads the 8087 control register, status register, tag word and exception pointers from a named area in memory. |
| 12 | FINCSTP | Increment the 8087 stack pointer by one. |
| 13 | FDECSTP | Decrement the stack pointer by one. |
| 14 | FFREE destination | Changes the tag for the specified destination register to empty. |
| 15 | FNOP | Performs no operation. Actually copies ST to ST. |
| 16 | FWAIT | This instruction is actually an 8086 instruction which makes the 8086 wait until it receives a not busy signal from the 8087 to its TEST* pin. |

Note: the processor control instructions actually do not perform computations but they are made used to perform tasks like initializing 8087, enabling intempty, etc.

# 8087 Programs

**1. Calculate area of a circle ($A = ®R^2$) given R, radius of the circle.**

```
; Procedure that calculates the area of a circle.
; The radius must be stored at memory location RADIUS before calling this procedure.
; The result is found in memory location AREA after the procedure.

AREAS       PROC FAR
            FINIT           ; Initialize 8087
            FLD   RADIUS    ; radius to ST
            FMUL  ST, ST(0) ; square radius
            FLDPI           ; π to ST
            FMUL            ; multiply ST=ST X ST(1)
            FSTP  AREA      ; save area
            FWAIT           ; wait for coprocessor
            RET
AREAS       ENDP
```

## OR

Program to calculate the area of circle. This program takes test data from array RAD that contains five sample radii. The five areas are stored in a second array called AREA. No attempt is made in this program to use the data from the AREA array.

```
; A short program that finds the area of five circles whose radii are stored in array RAD
            . MODEL SMALL
            .386            ; Select 80386
            .387            ; Select 80387
            .DATA

            .CODE
            .STARTUP
```

```
                MOV   SI, 0          ; source element 0
                MOV   DI, 0          ; destination element 0
                MOV   CX, 5          ; count of 5
MAIN1:
                FLD   RAD [SI]       ; radius to ST
                FMUL  ST, ST (0)     ; square radius
                FLDPI                ; π to ST
                FMUL                 ; multiply ST=ST X ST(1)
                FSTP  AREA [DI]      ; save area
                INC   SI
                INC   DI
                LOOP MAIN1
                .EXIT
                END
```

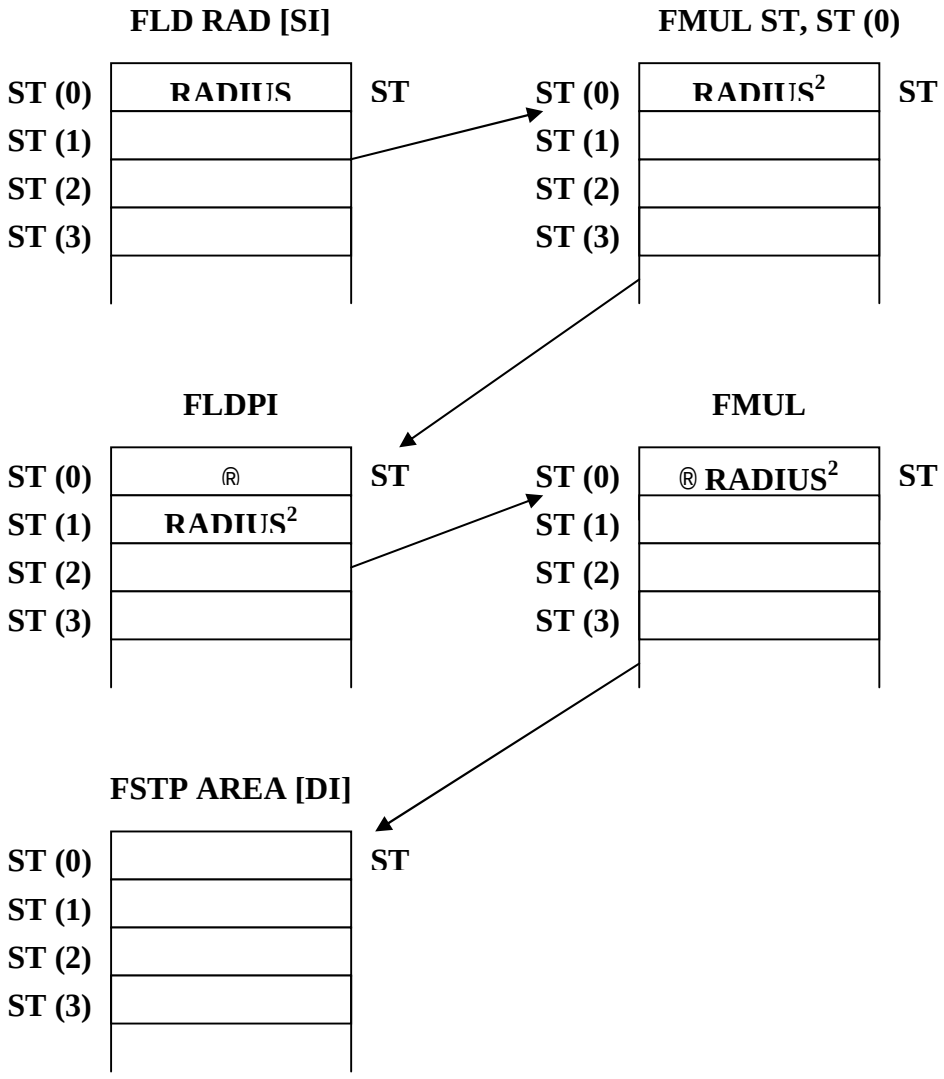


Fig: Operation of the stack for the above program. Note that the stack is shown after the execution of the indicated instruction.

2. Program for determining the resonant frequency of an LC circuit. The equation solved by the program is Fr = 1 / 2π √LC. This example uses L1 for inductance L, C1 for capacitor C, and RESO for the resultant resonant frequency.

; A sample program that finds the resonant frequency of an LC tank circuit.

```
                . MODEL SMALL
                .386            ; Select 80386
                .387            ; Select 80387
                .DATA
RESO            DD    ?         ; resonant frequency
L1              DD    0.000001  ; inductance
C1              DD    0.000001  ; capacitance
TWO             DD    2.0       ; constant
                .CODE
                .STARTUP
                FLD   L1        ; get L
                FMUL  C1        ; find LC
                FSQRT           ; find √LC
                FMUL  TWO       ; find 2√LC
                FLDPI           ; get π
                FMUL            ; get 2π√LC
                FLD1            ; get 1
                FDIVR           ; form 1/2π√LC
                FSTP  RESO      ; save frequency
                .EXIT
                END
```

**3. Program to find the roots of a polynomial expression (ax2+bx+cc=0) by using the quadratic equation. The quadratic equation is bï(Ěb$^2$ - 4ac)/2a**

Note: In this program R1 and R2 are the roots for the quadratic equation. The constants are stored in memory locations A1, B1, and C1. Note that no attempt is made to determine the roots if they are imaginary. This example tests for imaginary roots and exits to DOS with a zero in the roots (R1 and R2), if it finds them. In practice, imaginary roots could be solved for and stored in a separate set of memory locations.

; A program that finds the roots of a polynomial equation using the quadratic equation. Note ; imaginary roots are indicated if both root1 (R1) and root 2 (R2) are zero.

```
                . MODEL SMALL
                .386            ; Select 80386
                .387            ; Select 80387
                .DATA
TWO             DD    2.0
FOUR            DD    4.0
A1              DD    1.0
B1              DD    0.0
C1              DD    -9.0
R1              DD    ?
R2              DD    ?
                .CODE
                .STARTUP
                FLDZ
                FST   R1        ;clear roots
                FSTP  R2
                FLD   TWO
                FMUL  A1        ; form 2a
                FLD FOUR
                FMUL A1
                FMUL  C1        ; form 4ac
                FLD   B1
                FMUL  B1        ; form b2
                FSUBR           ; form b2 - 4ac
                FTST            ; test b2 – 4ac for zero
                FSTSW  AX       ; copy status register to AX
                SAHF            ; move to flags
                JZ    ROOTS1    ; if b2 - 4ac is zero
                FSQRT           ; find square root of b2 - 4ac
                FSTSW  AX
                TEST  AX, 1     ; test for invalid error (negative)
                JZ    ROOTS1
                FCOMPP          ; clear stack
                JMP   ROOTS2    ; end
ROOTS1:
```

```
                FLD     B1
                FSUB    ST, ST (1)
                FDIV    ST, ST (2)
                FSTP    R1              ; save root1
                FLD     B1
                FADD
                FDIVR
                FSTP    R2              ; save root2
ROOTS2:
                .EXIT
                                                        END
```

REVIEW QUESTION

1.Draw the internal struccture of 8087 arithmatic coprocessor july 2009 (10 marks)

2.explain the following coprocessor instruction: FSQRT,FSTP,F SCALE, F RNDINT, F COM July 2009 (10 marks)

3. Data transfer instruction of 8087. Jan 09 (10 marks)
4. Expain the interconnection of 8087 with 8086 july 2008 (10 marks)
5. Explain the control register of 8086 july 2008 (5 marks)

# UNIT:7: SYSTEM BUS STRUCTURE

## BASIC 8086 CONFIGURATION

### MINIMUM MODE VERSUS MAXIMUM MODE

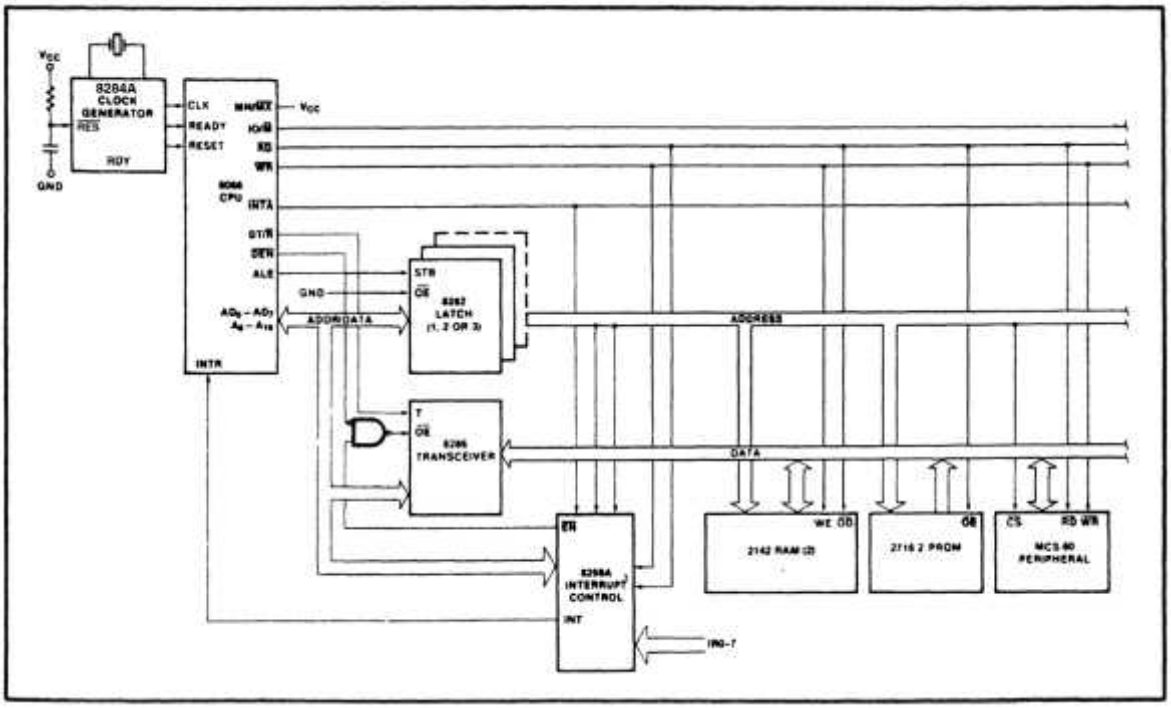
There are two available modes of operation for the 8086/8088 microprocessors: minimum mode and maximum mode. Minimum mode operation is obtained by connecting the mode selection pin MN/$\overline{MX}$ to +5.0 V, and maximum mode is selected by grounding this pin. Both modes enable different control structures for the 8086/8088 microprocessors. The mode of operation provided by minimum mode is similar to that of the 8085A, the most recent Intel 8-bit microprocessor, whereas maximum mode is new and unique and deigned to be used whenever a coprocessor exists in a system. Note that the maximum mode was dropped from the Intel family beginning with the 80286 microprocessor.

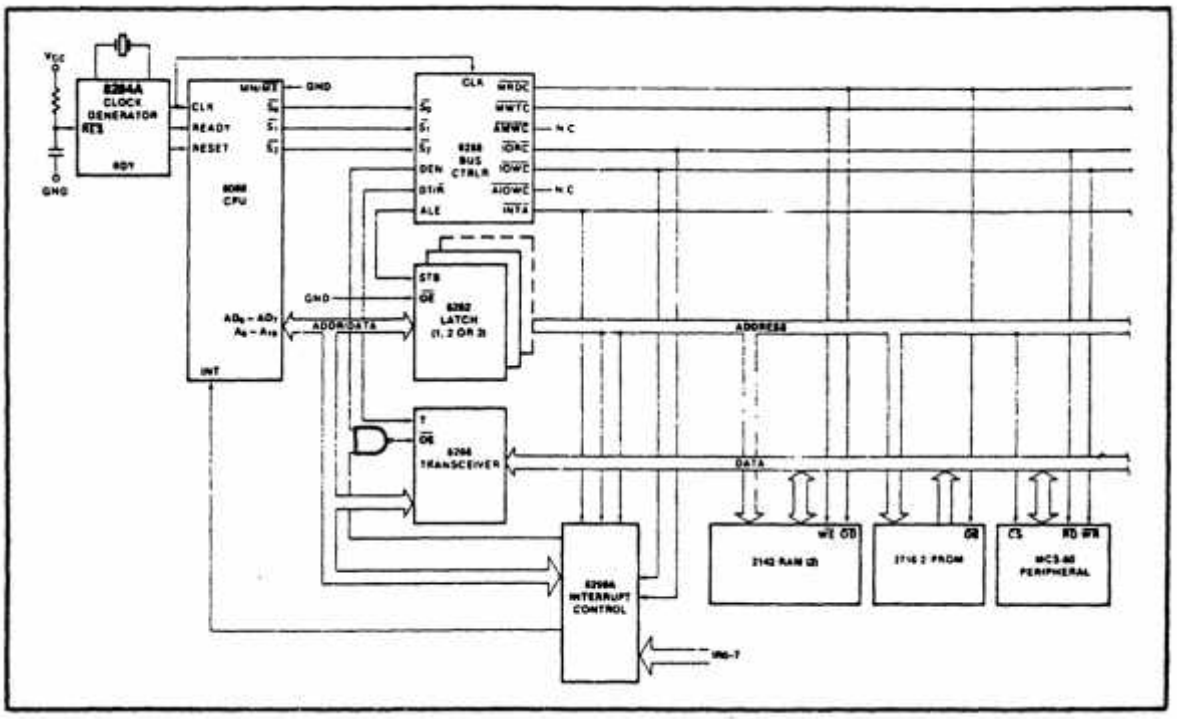
### Minimum Mode Operation

Minimum mode operation is the least-expensive way to operate the 8086/8088 microprocessors (see Figure 8–19 for the minimum mode 8088 system). It costs less because all the control signals for the memory and I/O are generated by the microprocessor. These control signals are identical to those of the Intel 8085A, an earlier 8-bit microprocessor. The minimum mode allows the 8085A, 8-bit peripherals to be used with the 8086/8088 without any special considerations.

### Maximum Mode Operation

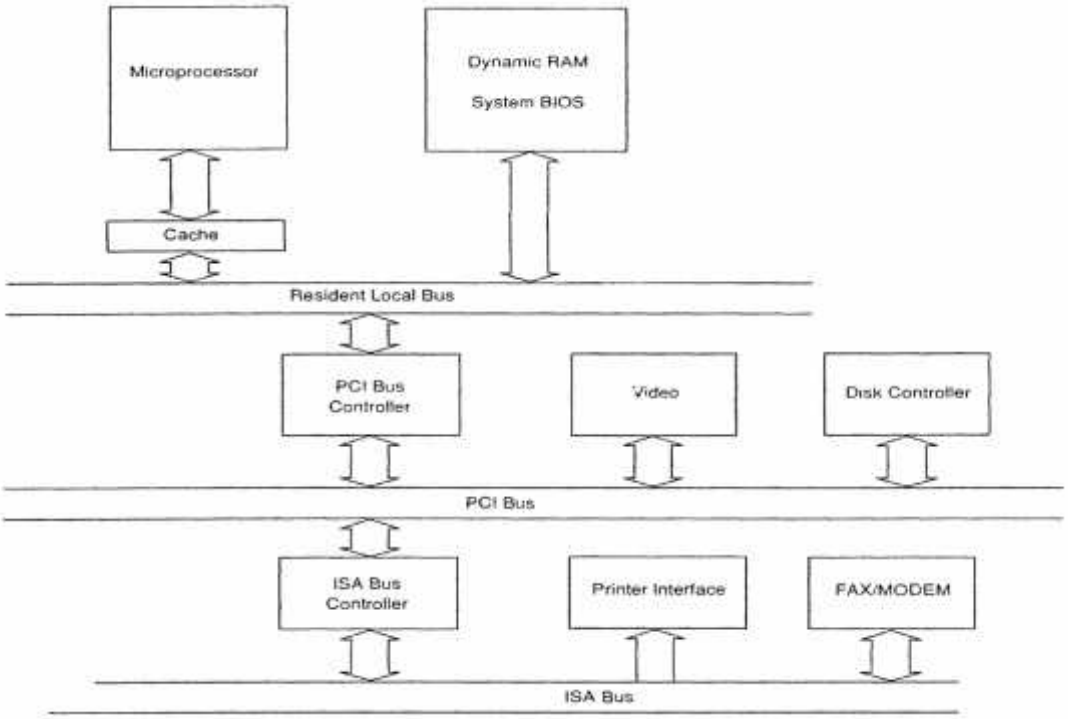
Maximum mode operation differs from minimum mode in that some of the control signals must be externally generated. This requires the addition of an external bus controller—the 8288 bus controller (see Figure 8–20 for the maximum mode 8088 system). There are not enough pins on



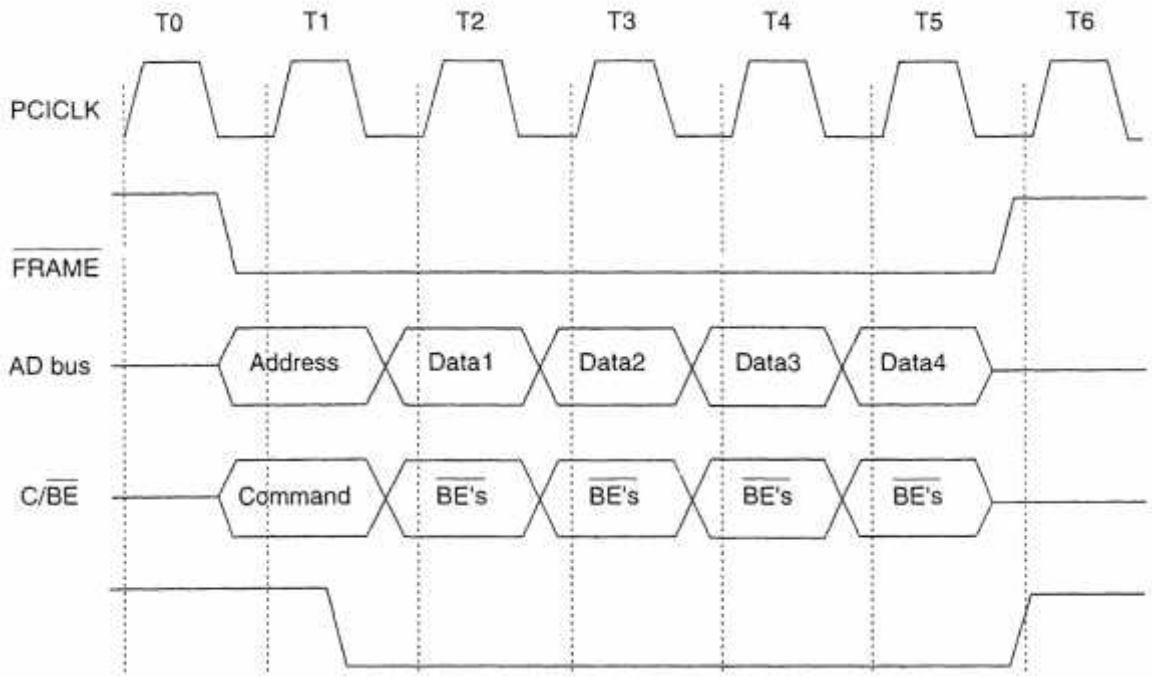

the 8086/8088 for bus control during maximum mode because new pins and new features have replaced some of them. Maximum mode is used only when the system contains external co-processors such as the 8087 arithmetic coprocessor.

connects to the PCI bus through an integrated circuit called a *PCI bridge*. This means that virtually any microprocessor can be interfaced to the PCI bus as long as a PCI controller or bridge is designed for the system. In the future, all computer systems may use the same bus. Even the Apple Macintosh system is switching to the PCI bus. Certainly, IBM will produce a Power-PC system that contains the PCI bus.



Back of computer

| Pin # | Solder Side | Component Side | Solder Side |
|---|---|---|---|
| 1 | -12V | TRST | |
| 2 | TCK | +12V | |
| 3 | GND | TMS | |
| 4 | TDO | TDI | |
| 5 | +5V | +5V | |
| 6 | +5V | INTA | |
| 7 | INTB | INTC | |
| 8 | INTD | +5V | |
| 9 | PRSNT 1 | | |
| 10 | | +VI/O | |
| 11 | PRSNT 2 | | |
| 12 | KEY | KEY | |
| 13 | KEY | KEY | |
| 14 | | | |
| 15 | GND | RST | |
| 16 | CLK | VI/O | |
| 17 | GND | VNT | |
| 18 | REQ | GND | |
| 19 | +V I/O | | |
| 20 | AD31 | AD30 | |
| 21 | AD29 | +3.3V | |
| 22 | GND | AD28 | |
| 23 | AD27 | AD26 | |
| 24 | AD25 | GND | |
| 25 | +3.3V | AD24 | |
| 26 | C/BE3 | IDSEL | |
| 27 | AD23 | +3.3V | |
| 28 | GND | AD22 | |
| 29 | AD21 | AD20 | |
| 30 | AD19 | GND | |
| 31 | +3.3V | AD18 | |
| 32 | AD17 | AD16 | |
| 33 | C/BE2 | +3.3V | |
| 34 | GND | FRAME | |
| 35 | IRDY | GND | |
| 36 | +3.3V | TRDY | |
| 37 | DEVSEL | GND | |
| 38 | GND | STOP | |
| 39 | LOCK | +3.3V | |
| 40 | PERR | SDONE | |




0000H indicating a processor shutdown, a 0001H for a processor halt, or a 0002H for 80X86 specific code or data.

**I/O Read Cycle** Data are read from an I/O device using the I/O address that appears on AD0–AD15. Burst reads are not supported for I/O devices.

**I/O Write Cycle** As with I/O read, this cycle accesses an I/O device, but writes data.

**Memory Read Cycle** Data are read from a memory device located on the PCI bus.

**Memory Write Cycle** As with memory read, data are accessed in a device located on the PCI bus. The location is written.

**Configuration Read** Configuration information is read from the PCI device using the configuration read cycle.

| C/BE3–C/BE0 | Command |
|---|---|
| 0000 | INTA sequence |
| 0001 | Special cycle |
| 0010 | I/O read cycle |
| 0011 | I/O write cycle |
| 0100–0101 | Reserved |
| 0110 | Memory read cycle |
| 0111 | Memory write cycle |
| 1000–1001 | Reserved |
| 1010 | Configuration read |
| 1011 | Configuration write |
| 1100 | Memory multiple access |
| 1101 | Dual addressing cycle |
| 1110 | Line memory access |
| 1111 | Memory write with invalidation |

| | |
|---|---|
| **Configuration Write** | The configuration write allows data to be written to the configuration area in a PCI device. Note that the address is specified by the configuration read. |
| **Memory Multiple Access** | This is similar to the memory read access, except that it is usually used to access many data instead of one. |
| **Dual Addressing Cycle** | Used for transferring address information to a 64-bit PCI device, which only contains a 32-bit data path. |
| **Line Memory Access** | Used to read more than two 32-bit numbers from the PCI bus. |
| **Memory Write with Invalidation** | This is the same as line memory access, but it is used with a write. This write bypasses the write-back function of the cache. |

**Review questions:**

1. Explain 8288 bus controller jan 09
2. Explain programmable interrupt controller jan 08
3. Explain LPT jan 09
4. Write a short notes on PCI, LPT, july09
5. Explain USB, USB data,USB commands. July 09

# UNIT:8:

**80386, 80486 AND PENTIUM PROCESSORS:** Introduction to the 80386 microprocessor, Special 80386 registers, Introduction to the 80486 microprocessor, Introduction to the Pentium microprocessor.

**7 Hours**

**TEXT BOOKS:**

1. **Microcomputer systems-The 8086 / 8088** Family – Y.C. Liu and G. A. Gibson, 2E PHI - 2003
2. **The Intel Microprocessor, Architecture, Programming and Interfacing**-Barry B. Brey, 6e, Pearson Education / PHI, 2003

## INTRODUCTION TO 80386 MICROPROCESSOR:

Introduced in 1986, the Intel 80386 provided a major upgrade to the earlier 8086 and 80286 processors in system architecture and features. The 80386 provided a base reference for the design of all Intel processors in the X86 family since that time, including the 80486, Pentium, Pentium Pro, and the Pentium II and III. All of these processors are extensions of the original design of the 80386. All are upwardly compatible with it. Programs written to run on the 80386 can be run with little or no modification on the later devices. The addressing scheme and internal architecture of the 80386 have been maintained and improved in the later microprocessors – thus a family of devices has evolved over the years that is the standard of a wide industry and upon which is based a vast array of software and operating system environments.

Major features of the 80386 include the following:

- A 32-bit wide address bus providing a real memory space of 4 gigabytes.
- A 32-bit wide data bus.
- Preemptive multitasking.
- Memory management, with four levels of protection.
- Virtual memory support, allowing 64 terabytes of virtual storage.
- Support for 8, 16, and 32-bit data types.
- Three primary modes of operation (Real, Protected, Virtual 8086).
- CMOS IV technology, 132-pin grid array.
- Object code compatibility with earlier X86 designs.

**PIN DESCRIPTIONS**

| Symbol | Type | Function |
|---|---|---|
| CLK2 | In | Provides the fundamental timing for the device. |
| D0 – D31 | I/O | **Data Bus** inputs data during memory, I/O, or interrupt read cycles, and |

| | | |
|---|---|---|
| | | outputs data during memory and I/O cycles. |
| A2 – A31 | Out | **Address Bus** provides physical memory or I/O port addresses. |
| BE0# - BE3# | Out | **Byte Enable** signals decode A0 and A1 to indicate specific banks for memory data transfers. |
| W/R# | Out | **Write/Read** defines nature of data transaction in progress. |
| D/C# | Out | **Data/Control** distinguishes data transfer cycles (memory or I/O) from control cycles (interrupt, halt, instruction fetch). |
| M/IO# | Out | **Memory/IO** identifies source/destination of current cycles. |
| LOCK# | Out | **Bus Lock** responds to a prefix byte on an instruction that indicates that other bus masters may not intercede the current cycle until it is complete. |
| ADS# | Out | **Address Status** indicates that a valid set of addressing signals are being driven onto the device pins.These include W/R#, D/C#, M/IO#, BE0#-BE3#, and A2-A31. |
| NA# | In | **Next Address** is used to request address pipelining. |
| READY# | In | **Bus Ready** requests a wait state from attached devices. |
| BS16# | In | **Bus Size 16** requests a 16-bit rather than a 32-bit data transfer. |
| HOLD | In | **Bus Hold Request** initiates a DMA cycle. |
| HLDA | Out | **Bus Hold Acknowledge** indicates that the processor is honoring a DMA request./TD> |
| BUSY# | In | **Busy** is a synchronization signal from an attached coprocessor, e.g., 80387. |
| ERROR# | In | **Error** signals an error condition in an attached coprocessor. |
| PEREQ | In | **Processor Extension Request** synchronizes a coprocessor data transfer via the 80386. |
| INTR | In | **Interrupt** accepts a request from a interrupting device (maskable). |
| NMI | In | **Non-Maskable Interrupt** forces an interrupt that cannot be ignored. |
| RESET | In | **Reset** causes the processor to enter a known state and destroys any execution in progress. |
| N/C | | **No Connect** indicates pins that are not to have any electrical connections. |
| VCC | In | **Power Supply** typically +5 volts. |
| VSS | In | **Ground.** |

**DATA FLOW**

Refer to the following diagram for illustration.
The Intel 80386 data flow consists of three primary areas. These are the bus interface unit (BIU), the central processing unit (CPU), and a memory management unit (MMU). These are interconnected within the device by several 32-bit-wide data busses and an internal control bus.
The **Bus Interface Unit (BIU)** provides the attachments of the device to the external bus system. The circuits include a set of address bus drivers which generate or receive the A2 – A31 address lines; the BE0 – BE3 byte selection lines; the control lines M/IO, D/C, W/R, Lock, ADS, NA, BS16, and Ready; and interface with the D0 – D31 data bus lines. The unit includes a pipeline control element which provides the memory access pipelining that permits fast data transfer from contiguous memory locations. The unit also includes a set of multiplex transceivers to handle the direction of incoming or outgoing data and address information. Also included is a control element that handles requests for interrupts, DMA cycles, and coprocessor synchronization.
The **Central Processing Unit (CPU)** is connected to the BIU via two paths. One is the direct ALU bus (across the bottom of the drawing) that allows exchange of addressing information and data between the CPU and the BIU if needed. The second is the normal path for instruction parts which go by way of an instruction prefetching element that is responsible for requesting instruction bytes from the memory as needed; an instruction predecoder that accepts bytes from the queue and ensures at least 3 instructions are available for execution; the instruction decoder and execution unit that causes the instruction to be performed. This is accomplished by the use of microprograms stored in the system control ROM which is stepped through to control the data flow within and around the Arithmetic Logic Unit (ALU).
The ALU consists of a register stack which contains both programmer-accessible and non-accessible 32-bit registers; a hardware multiply/divide element; and a 64-bit barrel shifter for shifts, rotates, multiplies, and divides. The ALU provides not only the data processing for the device but also is used to compute effective addresses (EAs) for protected mode addressing.
The **Memory Management Unit (MMU)** provides the support for both the segmentation of main memory for both protected mode and real mode, and the paging elements for virtual memory. In real mode, the segmentation of the main memory is limited to a maximum segment size of 64K bytes, and a maximum memory space of 1.024 megabytes. This is in concert with the Intel 8086 upon which this processor is based. In protected mode, several additional registers are added to support variable length segments to a maximum theoretical size of 4 gigabytes, which in turn supports multitasking and execution priority levels. Virtual mode using the device's paging unit allows a program or task to consume more memory than is physically attached to the device through the translation of supposed memory locations into either real memory or disk-based data.

**MODES OF OPERATION**

The Intel 80386 has three modes of operation available. These are Real Mode, Protected Mode, and Virtual 8086 mode.
**Real Mode** operation causes the device to function as would an Intel 8086 processor. It is faster by far that the 8086. While the 8086 was a 16-bit device, the 80386 can provide 32-bit extensions to the 8086's instructions. There are additional instructions to support the shift to protected mode as well as to service 32-bit data. In Real Mode, the address space is limited to 1.024 megabytes. The bottom 1,024 bytes contain the 256 4-byte interrupt vectors of the 8086. The Reset vector is FFFF0h. While the system can function as a simple DOS computer in this mode forever, the main purpose of the mode is to allow the initialization of several memory tables and flags so that a jump to Protected Mode may be made.
**Protected Mode** provides the 80386 with extensive capabilities. These include the memory management, virtual memory paging, multitasking, and the use of four privilege levels which allows the creation of sophisticated operating systems such as Windows NT and OS/2. (These will be further explained.)
**Virtual 8086 Mode** allows the system, once properly initialized in Protected Mode, to create one or more virtual 8086 tasks. These are implemented essentially as would be a Real Mode task,

except that they can be located anywhere in memory, there can be many of them, and they are limited by Real Mode constructs. This feature allows a 386-based computer, for example, to provide multiple DOS sessions or to run multiple operating systems, each one located in its own 8086 environment. OS/2 made use of this feature in providing multiple DOS sessions and to support its Windows 3.1 emulator. Windows NT uses the feature for its DOS windows.

**REGISTER ORGANIZATION**

***Programmer-visible Registers***

The ‘386 provides a variety of General Purpose Registers (GPRs) that are visible to the programmer. These support the original 16-bit registers of the 8086, and extend them to 32-bit versions for protected mode programming.

Chart goes here.

The AX, BX, CX, and DX registers exist in the same form as in the 8086. The may be used as 16-bit registers when called with the "X" in their name. They may also be used as 8-bit registers when defined with the "H" and "L" in their names. Hence, the AX register is used as a 16-bit device while the AH and AL are used as 8-bit devices. Similarly, Source Index (SI), Destination Index (DI), Base Pointer (BP) and Stack Pointer (SP) registers exist in their traditional 16-bit form.

To use any of these registers as 32-bit entities, the letter "E", for extended, is added to their names. Hence, the 16-bit AX register can become the 32-bit EAX register, the 16-bit DI register becomes the 32-bit EDI register, etc.

The registers of the ‘386 includes the 8086’s Code Segment (CS) register, Stack Segment (SS) register, Data Segment (DS) register, and Extra Segment (ES) register which are used as containers for values pointing to the base of these segments. Additionally, two more data-oriented segment registers, the FS and GS registers, are provided. In real mode, these registers contain values that point to the base of a segment in the real mode’s 1.048 megabyte address space. An offset is added to this displaced to the right which generates a real address. In protected mode, the segment registers contain a "selector" value which points to a location in a table where more information about the location of the segment is stored.

The ‘386 also provides an Instruction Pointer (IP) register and a Flags (FLAGS) register which operate as they did in the 8086 in real mode. In protected mode, these become 32-bit devices which provide extended features and addressing.

The 32-bit FLAGS register contains the original 16 bits of the 8086-80286 flags in bit positions 0 through 15 as follows. These are available to real mode.

| Bit | Flag | Description |
|---|---|---|
| 0 | CF | Carry Flag |
| 1 | 1 | Always a 1 |
| 2 | PF | Parity Flag |
| 3 | 0 | Always a 0 |
| 4 | AF | Auxiliary Carry Flag |
| 5 | 0 | Always a 0 |
| 6 | ZF | Zero Flag |
| 7 | SF | Sign Flag |
| 8 | TF | Trap Flag |

| | | |
|---|---|---|
| 9 | IF | Interrupt Enable |
| 10 | DF | Direction Flag |
| 11 | OF | Overflow Flag |
| 12-13 | PL1,2 | I/O Privilege Level Flags |
| 14 | NT | Nested Task Flag |
| 15 | 0 | Always a 0 |

Two more flags are provided to support protected mode.

| **Bit** | **Flag** | **Description** |
|---|---|---|
| 16 | RF | Resume Flag |
| 17 | VM | Virtual Mode |

Here are some brief descriptions of the functions of these flags.
CARRY FLAG – This flag is set when a mathematical function generated a carry out of the highest bit position of the result, such as when 9 + 1 = 10.
PARITY FLAG – This flag is set when the low order 8 bits of an operation results in an even number of one's set on, that is, even parity.
AUXILIARY CARRY FLAG – This flag is set when there is a carry out of the lower four bits of a 8-bit byte due to a mathematical operation. It supports the use of packed BCD encoding for accounting.
ZERO FLAG – This flag is set if all bits of a result are 0.
SIGN FLAG – This bit is set if the high-order bit of a result is a 1. In signed mathematics, this indicates a negative number.
TRAP ENABLE FLAG – This flag supports the use of Exception 1 when single stepping through code with a debugger package. When the flag is set, the '386 will execute an Exception 1 interrupt after the execution of the next instruction. If reset, the '386 will execute an Exception 1 interrupt only at breakpoints.
INTERRUPT ENABLE FLAG – This flag, when set, allows interrupts via the INTR device pin to be honored.
DIRECTION FLAG – This flag supports string OP codes that make use of the SI or DI registers. It indicates which direction the succeeding count should take, decrement if the flag is set, and increment if the flag is clear.
OVERFLOW FLAG – This flag is set if an operation results in a carry into the uppermost bit of the result value, that is, if a carry in the lower bits causes the sign bit to change.
I/O PRIVILEGE LEVEL - These two flags together indicate one of four privilege levels under which the processor operates in protected mode. These are sometimes called "rings", with ring 0 being the most privileged and ring 3 the least.
RESUME FLAG – This flag supports a debug register used to manage breakpoints in protected mode.
VIRTUAL MODE – This flag supports the third mode of operation of the processor, Virtual 8086 mode. Once in protected mode, if set, this flag causes the processor to switch to virtual 8086 mode.

***Programmer-invisible Registers***

To support protected mode, a variety of other registers are provided that are not accessible by the programmer. In real mode, the programmer can see and reference the segment registers CS, SS,

DS, ES, FS, and GS as 16-bit entities. The contents of these registers are shifted four bit positions to the left, then added to a 16-bit offset provided by the program. The resulting 20-bit value is the real address of the data to be accessed at that moment. This allows a real address space of $2^{20}$ or 1.048 megabytes. In this space, all segments are limited to 64K maximum size.
In protected mode, segments may from 1 byte to 4.3 gigabytes in size. Further, there is more information that is needed than in real mode. Therefore, the segment registers of real mode become holders for "selectors", values which point to a reference in a table in memory that contains more detail about the area in the desired segment. Also, a set of "Descriptor Registers" is provided, one for each segment register. These contain the physical base address of the segment, the segment limit (or the size of the segment relative to the base), and a group of other data items that are loaded from the descriptor table. In protected mode, when a segment register is loaded with a new selector, that selector references the table that has previously been set up, and the descriptor register for that segment register is given the new information from the table about that segment. During the course of program execution, addressing references to that segment are made using the descriptor register for that segment.
Four Control Registers CR0 – CR3 are provided to support specific hardware needs. CR0 is called the Machine Control Register and contains several bits that were derived in the 80286. These are:
PAGING ENABLED, bit 31 – This bits when set enables the on-chip paging unit for virtual memory.
TASK SWITCHED, bit 3 – This bit is set when a task switch is performed.
EMULATE COPROCESSOR, bit 2 – This bit causes all coprocessor OP codes to cause a Coprocessor-Not-Found exception. This is turn will cause 80387 math coprocessor instructions to have to be interpreted by software.
MONITOR COPROCESSOR, bit 1 – Works with the TS bit above to synchronize the coprocessor.
PROTECTION ENABLED, bit 0 – This bit enables the shift to protected mode from real mode.

1. A system reset.

**PROTECTED MODE ARCHTECTURE**
The 80386 is most impressive when running in protected mode. The linear address space can be as great as $2^{32}$ (4294967295) bytes. With the paging unit enabled, the limit is $2^{46}$ or about 64 terabytes. The device can run all 8086 and 80286 code. It provides a memory management and a hardware-assisted protection mechanism that keeps one program's execution from interfering with another. Additional instructions are provided to support multitasking. The programmer sees an expanded address space available to her/him, and different addressing scheme.
***Memory Segmentation***
Memory segmentation in protected mode uses a segment base value and an offset in the manner of real mode. However, because of the increased size of the address space now available, a more complex arrangement is used. The segment register now contains a value called a selector. This is a 16-bit value which contains an offset into a table. This table, called a descriptor table, contains descriptors which are 8-byte values that describe more about the segment in question. Two tables provided are the Global Descriptor Table (GDT) and the Local Descriptor Table (LDT). The GDT contains information about segments that are global in nature, that is, available to all programs and normally used most heavily by the operating system. The LDT contains descriptors that are application specific. Both of these tables have a limit of 64K, that is, 8,192 8-byte entries. There is also an Interrupt Descriptor Table (IDT) that contains information about segments containing code used in servicing interrupts. This table has a maximum of 256 entries.
The upper 13 bits of the selector are used as an offset into the descriptor table to be used. The lower 3 bits are:

- TI, a table selection bit – 0 = use the GDT, 1 = use the LDT.

- RPL, Requested Privilege Level bits = 00 is the highest privilege level, 11 is the lowest.

The selector identifies the table to be used and the offset into that table where a set of descriptor bytes identifies the segment specifically. Each table can be 64K bytes in size, so if there are 8 bytes per table entry, a total of 8,192 entries can be held in one table at a given time. The contents of a descriptor are:
Bytes 0 and 1 – A 16-bit value that is connected to bits 0 – 3 of byte 6 to form the uppermost offset, or limit, allowed for the segment. This 20 bit limit means that a segment can be between 1 byte and 1 megabyte in size. See the discussion of the granularity bit below.
Bytes 2 and 3 – A 16-bit value connected to byte 4 and byte 7 to form a 32-bit base value for the segment. This is the value added to the offset provided by the program execution to form the linear address.
AV bit – Segment available bit, where AV=0 indicates not available and AV=1 indicates available.
D bit – If D=0, this indicates that instructions use 16-bit offsets and 16-bit registers by default. If D=1, the instructions are 32-bit by default.
Granularity (G) bit – If G=0, the segments are in the range of 1 byte to 1 megabyte. If G=1, the segment limit value is multiplied by 4K, meaning that the segments can have a minimum of 4K bytes and a maximum limit of 4 gigabytes in steps of 4K.
Byte 5, Access Rights byte – This byte contains several flags to further define the segment:

- Bit 0, Access bit – A=0 indicates that the segment has not been accessed; A=1 indicates that the segment has been accessed (and is now "dirty").
- Bits 1, R/W bit; bit 2, ED/C bit; and bit 3, E bit. If bit 3 = 0, then the descriptor references a data segment and the other bits are interpreted as follows: bit 2, interpreted as the ED bit, if 0, indicates that the segment expands upward, as in a data segment; if 1, indicates that the segment expands in the downward direction, as in a stack segment; bit 1, the R/W bit, if 0, indicates that the segment may not be written, while if 1 indicates that the segment is writeable.

  If bit 3 = 1, then the descriptor references a code segment and the other bits are interpreted as follows: bit 2, interpreted as the C bit, if 0, indicates that we should ignore the descriptor privilege for the segment, while if 1 indicates that privilege must be observed; bit 1, the R/W bit, if 0, indicates that the code segment may not be read, while if 1 indicates that the segment is readable.
- Bit 4, System bit – If 0, this is a system descriptor; if 1, this is a regular code or data segment.
- Bits 5 and 6, Descriptor Privilege Level (DPL) bits – These two bits identify the privilege level of the descriptor.
- Bit 7, Segment Valid (P) bit – If 0, the descriptor is undefined. If 1, the segment contains a valid base and limit.

Use the illustration below to follow the flow of address translation. Numbers in circles on the drawing match those below.
File goes here

1. The execution of an instruction causes a request to access memory. The segment portion of the address to be used is represented by a selector value. This is loaded into the segment register. Generally, this value is not changed too often, and is controlled by the operating system.

2. The selector value in the segment register specifies a descriptor table and points to one of 8,192 descriptor areas. These contain 8 bytes that identify the base of the real segment, its limit, and various access and privilege information.
3. The base value in the descriptor identifies the base address of the segment to be used in linear address space.
4. The limit value in the descriptor identifies the offset of the top of the segment area from the base.
5. The offset provided by the instruction is used to identify the specific location of the desired byte(s) in linear address space, relative to the base value.

The byte(s) thus specified are read or written as dictated by the instruction.

***Program Invisible Registers***

Several additional registers are provided that are normally invisible to the programmer but are required by the hardware of the processor to expedite its functions.

Each of the segment registers (CS, DS, SS, ES, FS, and GS) have an invisible portion that is called a cache. The name is used because they store information for short intervals – they are not to be confused with the L1 or L2 cache of the external memory system. The program invisible portions of the segment registers are loaded with the base value, the limit value, and the access information of the segment each time the segment register is loaded with a new selector. This allows just one reference to the descriptor table to be used for multiple accesses to the same segment. It is not necessary to reference the descriptor table again until the contents of the segment register is changed indicating a new segment of that type is being accessed. This system allows for faster access to the main memory as the processor can look in the cache for the information rather than having to access the descriptor table for every memory reference to a segment.

The Global Descriptor Table Register (GDTR) and the Interrupt Descriptor Table Register (IDTR) contain the base address of the descriptor tables themselves and their limits, respectively. The limit is a 16-bit value because the maximum size of the tables is 64K.

***System Descriptors***

The Local Descriptor Table Register contains a 16-bit wide selector only. This value references a system descriptor, which is similar to that as described above, but which contains a type field that identifies one of 16 types of descriptor (specifically type 0010) that can exist in the system. This system descriptor in turn contains base and limit values that point to the LDT in use at the moment. In this way, there is one global descriptor table for the operating system, but there can be many local tables for individual applications or tasks if needed.

System descriptors contain information about operating system tables, tasks, and gates. The system descriptor can identify one of 16 types as follows. You will notice that some of these are to support backward compatibility with the 80286 processor.

| Type | Purpose |
|---|---|
| 0000 | Invalid |
| 0001 | Available 80286 Task State Segment |
| 0010 | Local Descriptor Table |
| 0011 | Busy 80286 Task State Segment |
| 0100 | 80286 Call Gate |
| 0101 | Task Gate |

| | |
|---|---|
| 0110 | 80286 Interrupt Gate |
| 0111 | 80286 Trap Gate |
| 1000 | Invalid |
| 1001 | Available 80386 Task State Segment |
| 1010 | Reserved |
| 1011 | Busy 80386 Task State Segment |
| 1100 | 80386 Call Gate |
| 1101 | Reserved |
| 1110 | 80386 Interrupt Gate |
| 1111 | 80386 Trap Gate |

***Protection and Privilege Levels***

The 80386 has four levels of protection which support a multitasking operating system. These serve to isolate and protect user programs from each other and from the operating system. The privilege levels manage the use of I/O instructions, privileged instructions, and segment and segment descriptors. Level 0 is the most trusted level, while level 3 is the least trusted level.
Intel lists the following rules for the access of data and instruction levels of a task:

- Data stored in a segment with privilege level P can be accessed only by code executing at a privilege level that is at least as privileged as P.
- A code segment or procedure with privilege level P can only by called by a task executing at the same or a less privileged level than P.

At any point in time, a task can be operating at any of the four privilege levels. This is called the task's Current Privilege Level (CPL). A task's privilege level may only be changed by a control transfer through a gate descriptor to a code segment with a different privilege level.
The lower two bits of selectors contain the Requested Privilege Level (RPL). When a change of selector is made, the CPL of the task and the RPL of the new selector are compared. If the RPL is more privileged than the CPL, the CPL determines the level at which the task will continue. If the CPL is more privileged than the RPL, the RPL value will determine the level for the task. Therefore, the lowest privilege level is selected at the time of the change. The purpose of this function is to ensure that pointers passed to an operating system procedure are not of a higher privilege than the procedure that originated the pointer.

***Gates***

Gates are used to control access to entry points within the target code segment. There are four types:

- Call Gates – those associated with Call, Jump, Return and similar operations codes. They provide a secure method of privilege transfer within a task.
- Task Gates – those involved with task switching.
- Interrupt Gates – those involved with normal interrupt service needs.
- Trap Gates – those involved with error conditions that cause major faults in the execution.

A gate is simply a small block of code in a segment that allows the system to check for privilege level violations and to control entry to the operating system services. The gate code lives in a segment pointed to by special descriptors. These descriptors contain base and offset values to locate the code for the gate, a type field, a two-bit Default Privilege Level (DPL) and a five-bit word count field. This last is used to indicate the number of words to be copied from the stack of the calling routine to that of the called routine. This is used only in Call Gates when there is a change in privilege level required. Interrupt and Trap gates work similarly except that there is no pushing of parameters onto the stack. For interrupt gates, further interrupts are disabled. Gates are part of the operating system and are mainly of interest to system programmers.

***Task Switching***

An important part of any multitasking system is the ability to switch between tasks quickly. Tasks may be anything from I/O routines in the operating system to parts of programs written by you. With only a single processor available in the typical PC, it is essential that when the needs of the system or operator are such that a switch in tasks is needed, this be done quickly.

The 80386 has a hardware task switch instruction. This causes the machine to save the entire current state of the processor, including all the register contents, address space information, and links to previous tasks. It then loads a new execution state, performs protection checks, and begins the new task, all in about 17 microseconds. The task switch is invoked by executing an intersegment jump or call which refers to a Task Switch Segment (TSS) or a task gate descriptor in the LDT or GDT. An INT n instruction, exception, trap, or external interrupt may also invoke a task switch via a task gate descriptor in the associated IDT.

Each task must have an associated Task Switch Segment. This segment contains an image of the system's conditions as they exist for that task. The TSS for the current task, the one being executed by the system at the moment, is identified by a special register called the Task Switch Segment Register (TR). This register contains a selector referring to the task state segment descriptor that defines the current TSS. A hidden base and limit register connected to the TR are loaded whenever TR is updated. Returning from a task is accomplished with the IRET instruction which returns control to the task that was interrupted with the switch. The current task's segment is stored and the previous task's segment is used to bring it into the current task.

***Control Registers***

The 80386 has four "Control Registers" called CR0 through CR3. CR0 contains several bit flags as follows:

PG – When set to 1, causes the translation of linear addresses to physical addresses. Indicates that paging is enabled and virtual memory is being used.

ET – When set to 1, indicates that the 80387 math coprocessor is in use.

TS – When set to 1, indicates that the processor has switched tasks.

EM – When set to 1, causes a type 7 interrupt for the ESC (escape) instruction for the math coprocessor.

MP – When set to 1, indicates that the math coprocessor is present in the system.

PE – Selects protected mode of operation.

CR 1 is not used by the '386. CR2 contains page fault linear addresses for the virtual memory manager. CR3 contains a pointer to the base of the page directory for virtual memory management.

***Switching to Protected Mode***

At reset, the 80386 begins operation in Real Mode. This is to allow setup of various conditions before the switch to Protected Mode is made. The actual switch is accomplished by setting the PE bit in CR0. The following steps are needed.

1. Initialize the interrupt descriptor table to contain valid interrupt gates for at least the first 32 interrupt types. The IDT can contain 256 8-byte gates.
2. Set up the GDT so that it contains a null descriptor at position 0, and valid descriptors for at least one code, one data, and one stack segment.

3. Switch to protected mode by setting PE to 1.
4. Execute a near JMP to flush the internal instruction queue and to load the TR with the base TSS descriptor.
5. Load all the data selectors with initial values.
6. The processor is now running in Protected Mode using the given GDT and IDT.

In the case of a multitasking system, an alternate approach is to load the GDT with at least two TSS descriptors in addition to the code and data descriptors needed for the first task. The first JMP following the setting of the PE bit will cause a task switch that loads all the data needed from the TSS of the first task to be entered. Multitasking is then initialized.

**VIRTUAL 8086 MODE**

The third mode of operation provided by the 80386 is that of Virtual 8086 Mode. Once in protected mode, one or more virtual 8086 tasks can be initiated. Virtual 8086 tasks appear to be like real mode. The task is limited to 1 megabyte of memory whose address space is located at 0 through FFFFFh; the segment registers are used as they are in real mode (no selectors or lookup tables are involved). Each of the virtual 8086 tasks are given a certain amount of time using a time-slice algorithm typical of mainframes (timesharing). The software for such tasks is written as if they were to run in a real mode address space. However, using paging, multiple such sessions can be located anywhere in the virtual memory space of the 80386.

Windows NT and OS/2 use this technique to support one or more DOS sessions, or low-priority utilities such as a print spooler.

**VIRTUAL MEMORY AND PAGING**

Using selectors and tables, the 80386 generates what Intel defines as a linear address as a means of locating data or instructions for real mode or for the current task in protected mode. If the system is not using virtual memory or paging, then the linear address is the physical address of the desired data or bytes, and is forwarded to the pins of the device to become the physical address.

Paging allows a level of interpretation to be inserted between the linear address and the physical address. The linear address is passed to the paging unit, and it in turn converts it to a physical address that will be different than the linear one. This allows several options, including 1) mapping a linear address to some other physical address according to the needs of a multitasking operating system to place tasks at convenient locations, or 2) mapping linear addresses to memory that does not exist in the system, but might be replaced by disk space.

Paging logically divides the available virtual space into "pages" that are 4Kbytes in size. Three elements are needed to implement paging. These are the page directory, the page table, and the actual physical memory page. Values in these tables are obtained by combining parts of the linear address with values from the tables which point to other values.

The page directory is a table of as many as 1,024 4-byte entries. (This is a maximum number; most systems use far fewer entries.) The base of the page directory is determined by the value contained in CR3. An offset into the directory is created from the uppermost 10 bits (positions 22-31) of the linear address. At this offset in the directory, we find a pointer to the base of a page table. This means that there can be as many as 1,024 page tables in a system.

There are 1,024 entries possible in each page table. The middle 10 bits of the linear address (bit positions 12 through 21) are used as a offset into the selected page table. The value thus determined is a pointer to the base of a 4K memory page. The offset into the page to located the specific data needed is contained in the lower 12 bits of the linear address.

The entries in the page directory and page tables are identical. They contain 10 bits of addressing, and the following flags:

D or DIRTY bit: This bit is not used in the page directory. In the page table entries, it indicates that the 4K area defined by this entry has been written to, and so must be saved (as to disk) if the area is to be reused for something else.

A or ACCESSED bit: This bit is set to a 1 when the processor accesses the 4K page.

R/W or Read/Write and U/S or User/Supervisor bits: These are used in conjunction with privilege management.
P or PRESENT bit: This bit when set to 1 indicates that the referenced page is present in memory. If 0, it can be used to indicate that the page is not in RAM, e.g., is on disk.
Performance of the paging system would be affected if the system needed to reference memory tables each time a reference to RWM was made. To offset this, a Translation Lookaside Buffer (TLB) is provided. This is a 4-way set-associative cache that contains entries for the last 32 pages needed by the processor. This provides immediate information about 98% of the time, causing only 2% of memory accesses to make the page directory-page table translation.

**HARDWARE HIGHLIGHTS**

The instructor will provide you with illustrations of the timing sequences for the various read and write cycles available on the 80386. There are two items of interest that we note here.

***Address Pipelining***

Under non-pipelined conditions, the bus signals of the '386 function very much like any other processor. A machine cycle consists of two T-states, T1 and T2. These are defined by the following edge of the system clock signal. At the beginning of T1, an address appears on the BE0# through BE3# and A2 through A31 lines, along with various control lines. The address is held valid until very near the end of T2. The ADS# line is pulled low (active) during T1 to indicate that the address bus contains a valid address; the ADS# line is pulled high (negated) during T2. The data is passed in or out at the transition between the end of T2 of the current cycle and the start of T1 of the following machine cycle. During this time, the NA# line is maintained high (negated).
In pipelining, the address bits are available ½ machine cycle earlier than with no pipelining. The ADS# line is pulled low during T2 of a cycle rather than T1, indicating that during T2, the address of the data to be exchanged during the ***next*** machine cycle is available. Pipelining is initiated by the incoming line NA#, that is controlled by the memory subsystem. If pulled low during a T1, the memory expects that the address of the next bytes needed will be available ½ cycle early.
The purpose of pipelining is to minimize the need for wait states. The time needed to read or write data remains the same. However, the time an address is available before the data is expected is lengthened so that a wait state may not be needed. The memory subsystem has to be designed to work within these parameters.

***Dynamic Bus Sizing***

Normally, the 80386 expects data to be transferred on a 32-bit wide data bus. However, it is possible to force the system to transfer 32-bit data as two 16-bit quantities in two successive bus cycles. This is initiated by the BS16# signal coming from the memory or I/O device subsystem. This line is pulled low during the middle of T2. It indicates to the processor that 32-bit data will be sent as two 16-bit words, with D0-D15 on the first transfer and D16-D31 on the second. The data is transferred on the D0-D15 bus lines; the D16-D31 lines are ignored.

**INSTRUCTION SET**

The instruction set of the 80386 is compatible with that of the 8086 and the programming for that processor can run on the '386 without modification. However, the '386 includes extension of the base instruction set to support 32-bit data processing and operation in protected mode. The reader is referred to the Intel documentation for full particulars on each instruction and its possible versions. Here we discuss the essential aspects of instruction organization.
Instructions vary in length, depending upon how much information must be given for the instruction, the addressing modes used, and the location of data to be processed. The generic instruction contains the following:
BYTE 1: This is the operation (OP) code for the instruction. Bit position 0 may be interpreted as the "w" bit, where w=0 indicates byte mode and w=1 indicates word mode. Also, bit position 1 may be interpreted as the operation direction bit in double operand instructions as follows:

| d | Direction of Operation |
|---|---|

| | |
|---|---|
| 0 | Register/Memory <- Register quot;reg" field indicates source operand "mod r/m" or "mod ss index base" indicates destination operand |
| 1 | Register <- Register/Memory "reg" field indicates destination operand "mod r/m" or "mod ss index base" indicates source operand |

BYTE 2 (optional): This second byte of OP code may or may not be used depending on the operation.
BYTE 3: This is the "mod r/m" byte. Bits 3, 4, and 5 contain more OP code information. Bits 0, 1, and 2 contain the "r/m", or "register/memory" of the instruction. These identify which registers are in use or how the memory is addressed (the addressing mode). The r/m bits are interpreted depending upon the two "mod" or mode bits according to this chart:

| **Mod r/m** | **16-bit Effective Address** | **32-bit Effective Address** |
|---|---|---|
| 00 000 | DS: [BX+SI] | DS: [EAX] |
| 00 001 | DS: [BX+DI] | DS: [ECX] |
| 00 010 | DS: [BP+SI] | DS: [EDX] |
| 00 011 | DS: [BP+DI] | DS: [EBX] |
| 00 100 | DS: [SI] | sib byte is present |
| 00 101 | DS: [DI] | DS: d32 |
| 00 110 | DS: d16 | DS: [ESI] |
| 00 111 | DS: [BX] | DS: [EDI] |
| 01 000 | DS: [BX+SI+d8] | DS: [EAX+d8] |
| 01 001 | DS: [BX+DI+d8] | DS: [ECX+d8] |
| 01 010 | SS: [BP+SI+d8] | DS: [EDX+d8] |
| 01 011 | SS: [BP+DI+d8] | DS: [EBX+d8] |
| 01 100 | DS: [SI+d8] | sib is present |
| 01 101 | DS: [DI+d8] | SS: [EBP+d8] |
| 01 110 | SS: [BP+d8] | DS: [ESI+d8] |
| 01 111 | DS: [BX+d8] | DS: [EDI+d8] |
| 10 000 | DS: [BX+SI+d16] | DS: [EAX+d32] |

| | | | | |
|---|---|---|---|---|
| 10 001 | DS: [BX+DI+d16] | | DS: [ECX+d32] | |
| 10 010 | SS: [BP+SI+d16] | | DS: [EDX+d32] | |
| 10 011 | SS: [BP+DI+d16] | | DS: [EBX+d32] | |
| 10 100 | DS: [SI+d16] | | sib is present | |
| 10 101 | DS: [DI+d16] | | SS: [EBP+d32] | |
| 10 110 | SS: [BP+d16] | | DS: [ESI+d32] | |
| 10 111 | DS: [BX+d16] | | DS: [EDI+d32] | |
| | **16-Bit Reg, w=0** | **16-Bit Reg, w=1** | **32-Bit Reg, w=0** | **32-Bit Reg, w=1** |
| 11 000 | AL | AX | AL | EAX |
| 11 001 | CL | CX | CL | ECX |
| 11 010 | DL | DX | DL | EDX |
| 11 011 | BL | BX | BL | EBX |
| 11 100 | AH | SP | AH | ESP |
| 11 101 | CH | BP | CH | EBP |
| 11 110 | DH | SI | DH | ESI |
| 11 111 | BH | DI | BH | EDI |

BYTE 4 (optional): This is the "sib" byte and is not found in the 8086. It appears only in some 80386 instructions as needed. This byte supports the "scaled index" addressing mode. Bit positions 0-2 identify a general register to be used as a base value. Bit positions 3-5 identify a general register which contains an index register. Bit positions 6 and 7 identify a scaling factor to be used to multiply the value in the index register as follows:

| **ss** | **Scale Factor** |
|---|---|
| 00 | 1 |
| 01 | 2 |
| 10 | 4 |
| 11 | 8 |

The index field of the sib byte is interpreted as follows:

| **Index** | **Index Register** |
|---|---|
| 000 | EAX |

| | |
|---|---|
| 001 | ECX |
| 010 | EDX |
| 011 | EBX |
| 100 | No index register used |
| 101 | EBP |
| 110 | ESI |
| 111 | EDI |

The mod field of the mod r/m byte taken with the base value of the sib byte generates the following scaled indexing modes:

| **Mod base** | **Effective Address** |
|---|---|
| 00 000 | DS: [EAX + (scaled index)] |
| 00 001 | DS: [ECX + (scaled index)] |
| 00 010 | DS: [EDX + (scaled index)] |
| 00 011 | DS: [EBX + (scaled index)] |
| 00 100 | SS: [ESP + (scaled index)] |
| 00 101 | DS: [d32 + (scaled index)] |
| 00 110 | DS: [ESI + (scaled index)] |
| 00 111 | DS: [EDI + (scaled index)] |
| 01 000 | DS: [EAX + (scaled index) + d8] |
| 01 001 | DS: [ECX + (scaled index) + d8] |
| 01 010 | DS: [EDX + (scaled index) + d8] |
| 01 011 | DS: [EBX + (scaled index) + d8] |
| 01 100 | SS: [ESP + (scaled index) + d8] |
| 01 101 | SS: [EBP + (scaled index) + d8] |
| 01 110 | DS: [ESI + (scaled index) + d8] |
| 01 111 | DS: [EDI + (scaled index) + d8] |
| 10 000 | DS: [EAX + (scaled index) + d32] |

| | |
|---|---|
| 10 001 | DS: [ECX + (scaled index) + d32] |
| 10 010 | DS: [EDX + (scaled index) + d32] |
| 10 011 | DS: [EBX + (scaled index) + d32] |
| 10 100 | SS: [ESP + (scaled index) + d32] |
| 10 101 | SS: [EBP + (scaled index) + d32] |
| 10 110 | DS: [ESI + (scaled index) + d32] |
| 10 111 | DS: [EDI + (scaled index) + d32] |

Following a possible byte 4, there may be 1, 2, or 4 bytes of address displacement which provide an absolute offset into the current segment for data location. Also following may be 1, 2, or 4 bytes to implement immediate data.

The byte and bit pattern of instructions vary. For instance, in conditional instructions a four-bit field called "tttn" implements the conditions to be tested:

| **Mnemonic** | **Condition** | **tttn** |
|---|---|---|
| O | Overflow | 0000 |
| NO | No Overflow | 0001 |
| B/NAE | Below/Not Above or Equal | 0010 |
| NB/AE | Not Below/Above or Equal | 0011 |
| E/Z | Equal/Zero | 0100 |
| NE/NZ | Not Equal/Not Zero | 0101 |
| BE/NA | Below or Equal/Not Above | 0110 |
| NBE/A | Not Below or Equal/Above | 0111 |
| S | Sign | 1000 |
| NS | Not Sign | 1001 |
| P/PE | Parity/Parity Even | 1010 |
| NP/PO | No Parity/Parity Odd | 1011 |
| L/NGE | Less Than/Not Greater or Equal | 1100 |
| NL/GE | Not Less Than/Greater or Equal | 1101 |
| LE/NG | Less Than or Equal/Not Greater Than | 1110 |
| NLE/G | Not Less Than or Equal/Greater Than | 1111 |

**Pentium**

About the Pentium Architecture
---

-- It is not a load/store architecture.

-- The instruction set is huge! We go over only a fraction of the instruction set. The text only presents a fraction.

-- There are lots of restrictions on how instructions/operands are put together, but there is also an amazing amount of flexibility.

Registers
---

The Intel architectures as a set just do not have enough registers to satisfy most assembly language programmers. Still, the processors have been around for a LONG time, and they have a sufficient number of registers to do whatever is necessary.

For our (mostly) general purpose use, we get

| 32-bit | 16-bit | 8-bit (high part of 16) | 8-bit (low part of 16) |
|---|---|---|---|
| EAX | AX | AH | AL |
| EBX | BX | BH | BL |
| ECX | CX | CH | CL |
| EDX | DX | DH | DL |

and

| | |
|---|---|
| EBP | BP |
| ESI | SI |
| EDI | DI |
| ESP | SP |

There are a few more, but we won't use or discuss them. They are only used for memory accessability in the segmented memory model.

Using the registers:
As an operand, just use the name (upper case and lower case both work interchangeably).

EBP is a frame pointer (see Chapter 11).
ESP is a stack pointer (see Chapter 11).

Oddities:
This is the only architecture that I know of where the programmer can designate part of a register as an operand. On ALL other machines, the whole register is designated and used.

ONE MORE REGISTER:
Many bits used for controlling the action of the processor and setting state are in the register called EFLAGS. This register contains the condition codes:

OF Overflow flag
SF Sign flag
ZF Zero flag
PF Parity flag
CF Carry flag

The settings of these flags are checked in conditional control instructions. Many instructions set one or more of the flags.

There are many other bits in the EFLAGS register: TO BE DISCUSSED LATER.

The use of the EFLAGS register is implied (rather than explicit) in instructions.

## Accessing Memory

There are 2 memory models supported in the Pentium architecture. (Actually it is the 486 and more recent models that support 2 models.)

In both models, memory is accessed using an address. It is the way that addresses are formed (within the processor) that differs in the 2 models.

### FLAT MEMORY MODEL

-- The memory model that we use. AND, the memory model that every

other manufactures' processors also use.

--

SEGMENTED MEMORY MODEL

-- Different parts of a program are assumed to be in their own, set-aside portions of memory. These portions are called segments.

-- An address is formed from 2 pieces: a segment location and an offset within a segment.

Note that each of these pieces can be shorter (contain fewer bits) than a whole address. This is much of the reason that Intel chose this form of memory model for its earliest single-chip processors.

-- There are segments for:

code
data
stack
other

-- Which segment something is in can be implied by the memory access involved. An instruction fetch will always be looking in the code segment. A push instruction (we'll talk about this with chapter 11) always accesses the stack segment. Etc.

Addressing Modes
---

Some would say that the Intel architectures only support 1 addressing mode. It looks (something like) this:

effective address = base reg + (index reg x scaling factor) + displacement

where
base reg is EAX, EBX, ECX, EDX or ESP or EBP
index reg is EDI or ESI
scaling factor is 1, 2, 4, or 8

The syntax of using this (very general) addressing mode will vary from system to system. It depends on the preprocessor and the syntax accepted by the assembler.

For our implementation, an operand within an instruction that uses this addressing mode could look like

[EAX][EDI*2 + 80]

The effective address calculated with be the contents of register EDI multiplied times 2 added to the constant 80, added to the contents of register EAX.

There are extremely few times where a high-level language compiler can utilize such a complex addressing mode. It is much more likely that simplified versions of this mode will be used.

SOME ADDRESSING MODES

-- register mode --

The operand is in a register. The effective address is the register

Example instruction:

```
mov  eax, ecx
```

Both operands use register mode. The contents of register ecx is copied to register eax.

-- immediate mode --

The operand is in the instruction. The effective address is within the instruction.

Example instruction:

```
mov  eax, 26
```

The second operand uses immediate mode. Within the instruction is the operand. It is copied to register eax.

-- register direct mode --

The effective address is in a register.

Example instruction:

```
mov  eax, [esp]
```

The second operand uses register direct mode. The contents of register esp is the effective address. The contents of memory at the effective address are copied into register eax.

-- direct mode --
The effective address is in the instruction.

Example instruction:

mov eax, var_name

The second operand uses direct mode. The instruction contains the effective address. The contents of memory at the effective address are copied into register eax.

-- base displacement mode --
The effective address is the sum of a constant and the contents of a register.

Example instruction:

mov eax, [esp + 4]

The second operand uses base displacement mode. The instruction contains a constant. That constant is added to the contents of register esp to form an effective address. The contents of memory at the effective address are copied into register eax.

-- base-indexed mode -- (Intel's name)
The effective address is the sum of the contents of two registers.

Example instruction:

mov eax, [esp][esi]

The contents of registers esp and esi are added to form an effective address. The contents of memory at the effective address are copied into register eax.

Note that there are restrictions on the combinations of registers that can be used in this addressing mode.

-- PC relative mode --
The effective address is the sum of the contents of the PC and a constant contained within the instruction.

Example instruction:

jmp a_label

The contents of the program counter is added to an offset that

is within the machine code for the instruction. The resulting sum is placed back into the program counter. Note that from the assembly language it is not clear that a PC relative addressing mode is used. It is the assembler that generates the offset to place in the instruction.

Instruction Set
---

Generalities:
-- Many (most?) of the instructions have exactly 2 operands. If there are 2 operands, then one of them will be required to use register mode, and the other will have no restrictions on its addressing mode.

-- There are most often ways of specifying the same instruction for 8-, 16-, or 32-bit oeprands. I left out the 16-bit ones to reduce presentation of the instruction set. Note that on a 32-bit machine, with newly written code, the 16-bit form will never be used.

Meanings of the operand specifications:
reg - register mode operand, 32-bit register
reg8 - register mode operand, 8-bit register
r/m - general addressing mode, 32-bit
r/m8 - general addressing mode, 8-bit
immed - 32-bit immediate is in the instruction
immed8 - 8-bit immediate is in the instruction
m - symbol (label) in the instruction is the effective address

Data Movement
---

```
mov   reg, r/m          ; copy data
        r/m, reg
        reg, immed
        r/m, immed

movsx reg, r/m8         ; sign extend and copy data

movzx reg, r/m8         ; zero extend and copy data

lea   reg, m            ; get effective address
      (A newer instruction, so its format is much restricted
       over the other ones.)
```

EXAMPLES:

```
mov EAX, 23  ; places 32-bit 2's complement immediate 23
             ; into register EAX
movsx ECX, AL  ; sign extends the 8-bit quantity in register
               ; AL to 32 bits, and places it in ECX
mov [esp], -1  ; places value -1 into memory, address given
               ; by contents of esp
lea EBX, loop_top ; put the address assigned (by the assembler)
                  ; to label loop_top into register EBX
```

Integer Arithmetic
---

```
add   reg, r/m          ; two's complement addition
      r/m, reg
      reg, immed
      r/m, immed

inc   reg               ; add 1 to operand
      r/m

sub   reg, r/m          ; two's complement subtraction
      r/m, reg
      reg, immed
      r/m, immed

dec   reg               ; subtract 1 from operand
      r/m

neg   r/m               ; get additive inverse of operand

mul   eax, r/m          ; unsigned multiplication
                        ; edx||eax <- eax * r/m

imul  r/m               ; 2's comp. multiplication
                        ; edx||eax <- eax * r/m
      reg, r/m          ; reg <- reg * r/m
      reg, immed        ; reg <- reg * immed

div   r/m               ; unsigned division
                                  ; does edx||eax / r/m
                                  ; eax <- quotient
                                  ; edx <- remainder

idiv  r/m               ; 2's complement division
                                  ; does edx||eax / r/m
                                  ; eax <- quotient
                                  ; edx <- remainder
```

```
cmp   reg, r/m          ; sets EFLAGS based on
      r/m, immed        ; second operand - first operand
      r/m8, immed8
      r/m, immed8       ; sign extends immed8 before subtract
```

EXAMPLES:

```
neg [eax + 4]    ; takes doubleword at address eax+4
                 ;   and finds its additive inverse, then places
                 ;   the additive inverse back at that address
                 ;   the instruction should probably be
                 ;     neg  dword ptr [eax + 4]

inc ecx          ; adds one to contents of register ecx, and
                 ;   result goes back to ecx
```

Logical
---

```
not   r/m               ; logical not

and   reg, r/m          ; logical and
      reg8, r/m8
            r/m, reg
      r/m8, reg8
            r/m, immed
            r/m8, immed8

or    reg, r/m          ; logical or
      reg8, r/m8
            r/m, reg
      r/m8, reg8
            r/m, immed
            r/m8, immed8

xor   reg, r/m          ; logical exclusive or
      reg8, r/m8
            r/m, reg
      r/m8, reg8
            r/m, immed
            r/m8, immed8

test  r/m, reg          ; logical and to set EFLAGS
      r/m8, reg8
            r/m, immed
            r/m8, immed8
```

EXAMPLES:

```
and edx, 00330000h   ; logical and of contents of register
                     ;   edx (bitwise) with 0x00330000,
                     ;   result goes back to edx
```

Floating Point Arithmetic
---

Since the newer architectures have room for floating point hardware on chip, Intel defined a simple-to-implement extension to the architecture to do floating point arithmetic. In their usual zeal, they have included MANY instructions to do floating point operations.

The mechanism is simple. A set of 8 registers are organized and maintained (by hardware) as a stack of floating point values. ST refers to the stack top. ST(1) refers to the register within the stack that is next to ST. ST and ST(0) are synonyms.

There are separate instructions to test and compare the values of floating point variables.

```
finit                ; initialize the FPU

fld   m32            ; load floating point value
      m64
      ST(i)

fldz                 ; load floating point value 0.0

fst   m32            ; store floating point value
      m64
      ST(i)

fstp  m32            ; store floating point value
      m64            ;   and pop ST
      ST(i)

fadd  m32            ; floating point addition
      m64
      ST, ST(i)
      ST(i), ST
```

```
faddp ST(i), ST          ; floating point addition
                         ;   and pop ST
```

I/O
---
The only instructions which actually allow the reading and writing of I/O devices are priviledged. The OS must handle these things. But, in writing programs that do something useful, we need input and output. Therefore, there are some simple macros defined to help us do I/O.

These are used just like instructions.

```
put_ch  r/m        ; print character in the least significant
                   ;   byte of 32-bit operand

get_ch  r/m        ; character will be in AL

put_str m          ; print null terminated string given
                   ; by label m
```

Control Instructions
---
These are the same control instructions that all started with the character 'b' in SASM.

```
jmp   m            ; unconditional jump
jg    m            ; jump if greater than 0
jge   m            ; jump if greater than or equal to 0
jl    m            ; jump if less than 0
jle   m            ; jump if less than or equal to 0
```

---

## Pin-out of the 80486DX and 80486SX Microprocessors

Figure 16–28 illustrates the pin-out of the 80486DX microprocessor, a 168-pin PGA. The 80486SX, also packaged in a 168-pin PGA, is not illustrated because only a few differences exist. Note that pin B15 is NMI on the 80486DX and pin A15 is NMI on the 80486SX. The only other differences are that pin A15 is $\overline{\text{IGNNE}}$ on the 80486DX (not present on the 80486SX), pin C14 is $\overline{\text{FERR}}$ on the 80486DX, and pins B15 and C14 on the 80486SX are not connected.

When connecting the 80486 microprocessor, all $V_{CC}$ and $V_{SS}$ pins must be connected to the power supply for proper operation. The power supply must be capable of supplying 5.0V ±10 %, with up to 1.2 A of surge current for the 33 MHz version. The average supply current is 650 mA for the 33 MHz version. Intel has also produced a 3.3V version that requires an average of 500 mA at a triple-clock speed of 100 MHz. Logic 0 outputs allow up to 4.0 mA of current, and logic 1 outputs allow up to 1.0 mA. If larger currents are required, as they often are, then the 80486 must be buffered. Figure 16–29 shows a buffered 80486DX system. In the circuit shown, only the address, data, and parity signals are buffered.

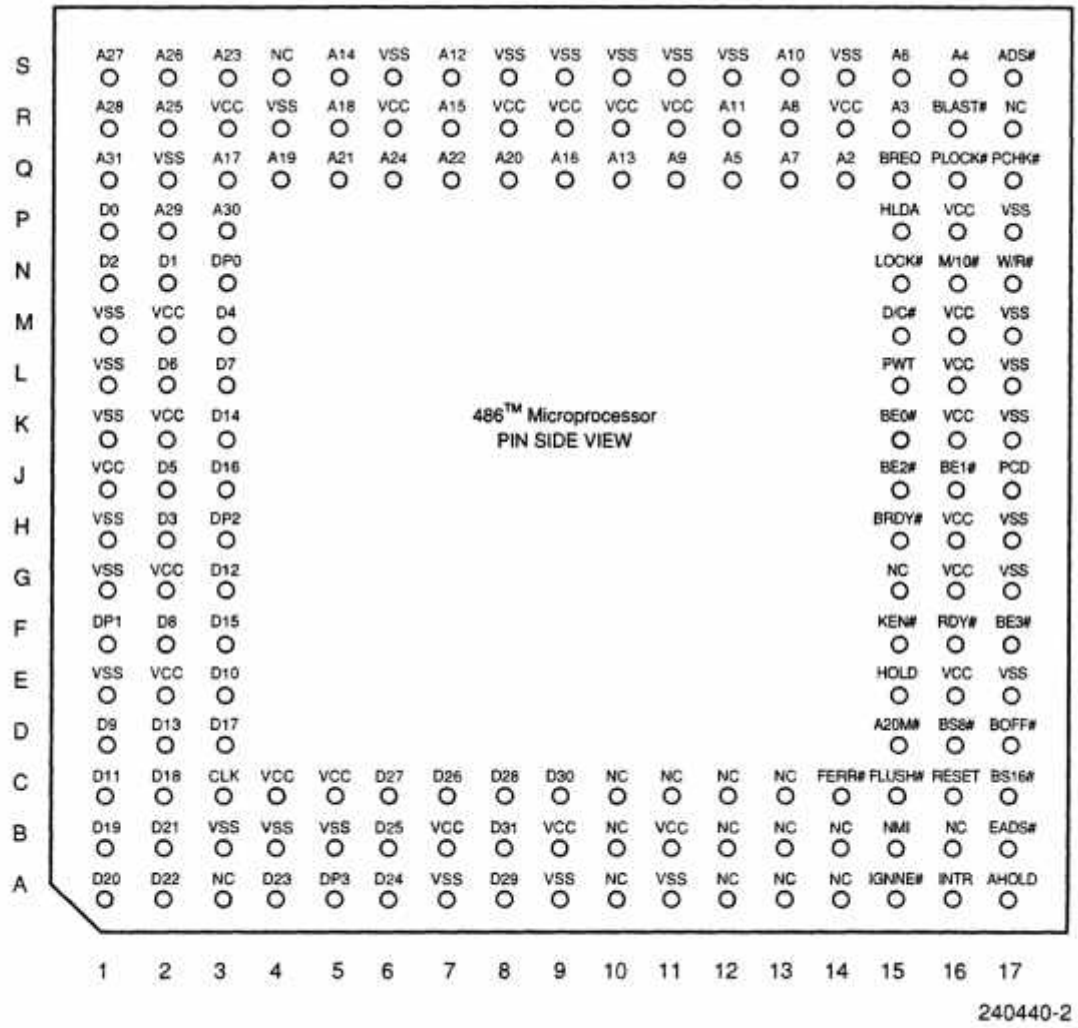



**FIGURE 16–28** The pin-out of the 80486 (Courtesy of Intel Corporation)

| | |
|---|---|
| $\overline{\text{A20M}}$ | The **address bit 20 mask** causes the 80486 to wrap its address around from location 000FFFFFH to 00000000H, as does the 8086 microprocessor. This provides a memory system that functions like the 1M-byte real memory system in the 8086 microprocessor. |
| $\overline{\text{ADS}}$ | The **address data strobe** becomes a logic zero to indicate that the address bus contains a valid memory address. |
| AHOLD | The **address hold** input causes the microprocessor to place its address bus connections at their high-impedance state, with the remainder of the buses staying active. It is often used by another bus master to gain access for a cache invalidation cycle. |
| $\overline{\text{BE3}}$–$\overline{\text{BE0}}$ | **Byte enable** outputs select a bank of the memory system when information is transferred between the microprocessor and its memory and I/O space. The $\overline{\text{BE3}}$ signal enables D31–D24, $\overline{\text{BE2}}$ enables D23–D16, $\overline{\text{BE1}}$ enables D15–D8, and $\overline{\text{BE0}}$ enables D7–D0. |
| $\overline{\text{BLAST}}$ | The **burst last** output shows that the burst bus cycle is complete on the next activation of the $\overline{\text{BRDY}}$ signal. |
| $\overline{\text{BOFF}}$ | The **back-off** input causes the microprocessor to place its buses at their high-impedance state during the next clock cycle. The microprocessor remains in the bus hold state until the $\overline{\text{BOFF}}$ pin is placed at a logic 1 level. |
| $\overline{\text{BRDY}}$ | The **burst ready** input is used to signal the microprocessor that a burst cycle is complete. |
| BREQ | The **bus request** output indicates that the 80486 has generated an internal bus request. |
| $\overline{\text{BS8}}$ | The **bus size 8** input causes the 80486 to structure itself with an 8-bit data bus to access byte-wide memory and I/O components. |
| $\overline{\text{BS16}}$ | The **bus size 16** input causes the 80486 to structure itself with a 16–bit data bus to access word-wide memory and I/O components. |
| CLK | The **clock** input provides the 80486 with its basic timing signal. The clock input is a TTL-compatible input that is 25 MHz to operate the 80486 at 25 MHz. |
| D31–D0 | The **data bus** transfers data between the microprocessor and its memory and I/O system. Data bus connections D7–D0 are also used to accept the interrupt vector type number during an interrupt acknowledge cycle. |

| | |
|---|---|
| **$\overline{\text{FLUSH}}$** | The **cache flush** input forces the microprocessor to erase the contents of its 8K-byte internal cache. |
| **HLDA** | The **hold acknowledge** output indicates that the HOLD input is active and that the microprocessor has placed its buses at their high-impedance state. |
| **HOLD** | The **hold** input requests a DMA action. It causes the address, data, and control buses to be placed at their high-impedance state and also, once recognized, causes HLDA to become a logic 0. |
| **$\overline{\text{IGNNE}}$** | The **ignore numeric error** input causes the coprocessor to ignore floating-point errors and to continue processing data. This signal does not affect the state of the $\overline{\text{FERR}}$ pin. |
| **D/$\overline{\text{C}}$** | The **data/control** output indicates whether the current operation is a data transfer or control cycle. Refer to Table 16–3 for the function of D/$\overline{\text{C}}$, M/$\overline{\text{IO}}$, and W/$\overline{\text{R}}$. |
| **DP3–DP0** | **Data parity I/O** provides even parity for a write operation and check parity for a read operation. If a parity error is detected during a read, the $\overline{\text{PCHK}}$ output becomes a logic 0 to indicate a parity error. If parity is not used in a system, these lines must be pulled-high to +5.0V or to 3.3V in a system that uses a 3.3V supply. |
| **$\overline{\text{EADS}}$** | The **external address strobe** input is used with AHOLD to signal that an external address is used to perform a cache invalidation cycle. |
| **$\overline{\text{FERR}}$** | The **floating-point error** output indicates that the floating-point coprocessor has detected an error condition. It is used to maintain compatibility with DOS software. |
| **INTR** | The **interrupt request** input requests a maskable interrupt as it does in all other family members. |
| **$\overline{\text{KEN}}$** | The **cache enable** input causes the current bus to be stored in the internal cache. |
| **$\overline{\text{LOCK}}$** | The **lock** output becomes a logic 0 for any instruction that is prefixed with the lock prefix. |
| **M/$\overline{\text{IO}}$** | **Memory/$\overline{\text{IO}}$** defines whether the address bus contains a memory address or an I/O port number. It is also combined with the W/R signal to generate memory and I/O read and write control signals. |
| **NMI** | The **non-maskable interrupt** input requests a type 2 interrupt. |
| **PCD** | The **page cache disable** output reflects the state of the PCD attribute bit in the page table entry or the page directory entry. |

read operation on the D7 - D0 pins.

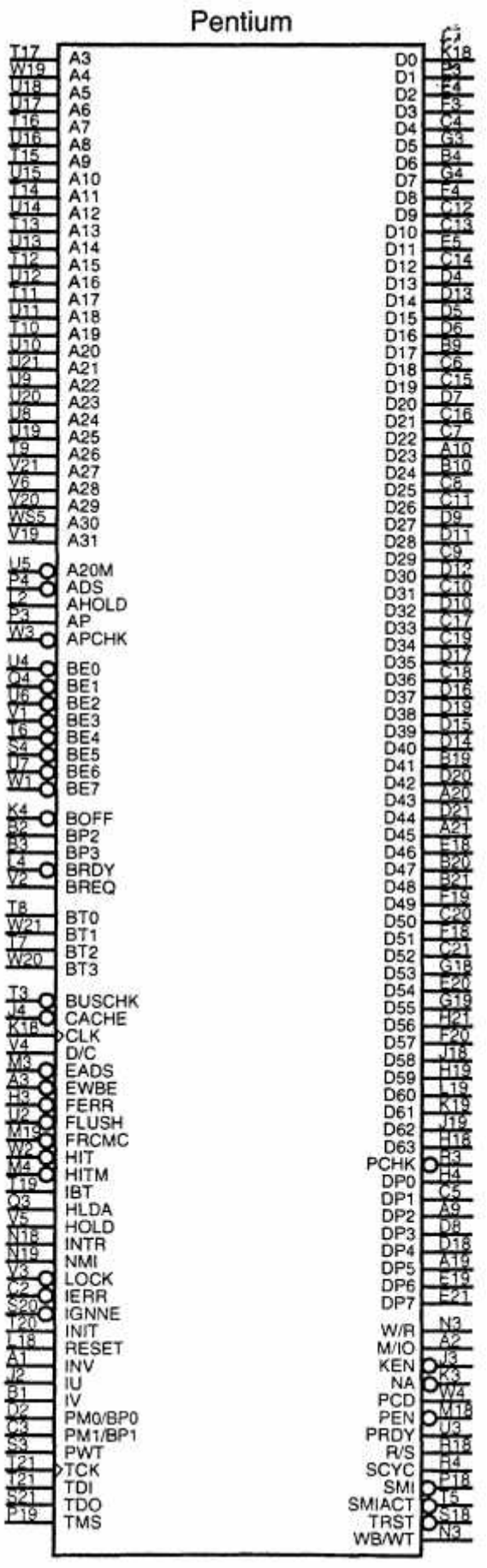
| | |
|---|---|
| $\overline{\text{PLOCK}}$ | The **pseudo-lock** output indicates that the current operation requires more than one bus cycle to perform. This signal becomes a logic 0 for arithmetic coprocessor operations that access 64- or 80-bit memory data. |
| **PWT** | The **page write through** output indicates the state of the PWT attribute bit in the page table entry or the page directory entry. |
| $\overline{\text{RDY}}$ | The **ready** input indicates that a non-burst bus cycle is complete. The RDY signal must be returned or the microprocessor places wait states into its timing until RDY is asserted. |
| **RESET** | The **reset** input initializes the 80486 as it does in other family members. Table 16–4 shows the effect of the RESET input on the 80486 microprocessor. |
| **W/$\overline{\text{R}}$** | **Write/read** signals that the current bus cycle is either a read or a write. |

Basic 80486 Architecture

## INTRODUCTION TO THE PENTIUM MICROPROCESSOR

Before the Pentium or any other microprocessor can be used in a system, the function of each pin must be understood. This section of the chapter details the operation of each pin, along with the external memory system and I/O structures of the Pentium microprocessor.

Figure 17–1 illustrates the pin-out of the Pentium microprocessor, which is packaged in a huge 237-pin PGA (pin grid array). Currently, the Pentium is available in two versions: the full-blown Pentium and the P24T version called the Pentium OverDrive. The P24T version contains a 32-bit data bus compatible for insertion into 80486 machines that contain the P24T socket. The P24T version also comes with a fan built into the unit. The most notable difference in the pin-out


Pentium
A3
A4
A5
A6
A7
A8
A9
A10
A11
A12
A13
A14
A15
A16
A17
A18
A19
A20
A21
A22
A23
A24
A25
A26
A27
A28
A29
A30
A31
A20M
ADS
AHOLD
AP
APCHK
BE0
BE1
BE2
BE3
BE4
BE5
BE6
BE7
BOFF
BP2
BP3
BRDY
BREQ
BT0
BT1
BT2
BT3
BUSCHK
CACHE
CLK
D/C
EADS
EWBE
FERR
FLUSH
FRCMC
HIT
HITM
IBT
HLDA
HOLD
INTR
NMI
LOCK
IERR
IGNNE
INIT
RESET
INV
IU
IV
PM0/BP0
PM1/BP1
PWT
TCK
TDI
TDO
TMS
D0
D1
D2
D3
D4
D5
D6
D7
D8
D9
D10
D11
D12
D13
D14
D15
D16
D17
D18
D19
D20
D21
D22
D23
D24
D25
D26
D27
D28
D29
D30
D31
D32
D33
D34
D35
D36
D37
D38
D39
D40
D41
D42
D43
D44
D45
D46
D47
D48
D49
D50
D51
D52
D53
D54
D55
D56
D57
D58
D59
D60
D61
D62
D63
PCHK
DP0
DP1
DP2
DP3
DP4
DP5
DP6
DP7
W/R
M/IO
KEN
NA
PCD
PEN
PRDY
R/S
SCYC
SMI
SMIACT
TRST
WB/WT

The function of each Pentium group of pins follows:

| | |
|---|---|
| $\overline{A20}$ | The **address A20 mask** is an input that is asserted in the real mode to signal the Pentium to perform address wraparound, as in the 8086 microprocessor, for use of the HIMEN.SYS driver. |
| **A31–A3** | **Address bus** connections address any of the 512K × 64 memory locations found in the Pentium memory system. Note that A0, A1, and A2 are encoded in the bus enable ($\overline{BE7}$–$\overline{BE0}$) to select any or all of the eight bytes in a 64-bit wide memory location. |
| $\overline{ADS}$ | The **address data strobe** becomes active whenever the Pentium has issued a valid memory or I/O address. This signal is combined with the W/$\overline{R}$ and M/$\overline{IO}$ signal to generate the separate read and write signals present in the earlier 8086–80286 microprocessor-based systems. |
| **AHOLD** | **Address hold** is an input that causes the Pentium to hold the address and AP signals for the next clock. |
| **AP** | **Address parity** provides even parity for the memory address on all Pentium-initiated memory and I/O transfers. The AP pin must also be driven with even parity information on all inquire cycles in the same clocking period as the $\overline{EADS}$ signal. |
| $\overline{APCHK}$ | **Address parity check** becomes a logic 0 whenever the Pentium detects an address parity error. |
| $\overline{BE7}$–$\overline{BE0}$ | **Bank enable signals** select the access of a byte, word, doubleword, or quad-word of data. These signals are generated internally by the microprocessor from address bits A0, A1, and A2. |
| $\overline{BOFF}$ | The **back-off** input aborts all outstanding bus cycles and floats the Pentium buses until $\overline{BOFF}$ is negated. After $\overline{BOFF}$ is negated, the Pentium restarts all aborted bus cycles in their entirety. |
| **BP[3:2] and PM/BP[1:0]** | The **breakpoint pins** BP3–BP0 indicate a breakpoint match when the **debug** registers are programmed to monitor for matches. The **performance monitoring** pins PM1 and PM0 indicate the settings of the performance monitoring bits in the debug mode control register. |
| $\overline{BRDY}$ | The **burst ready** input signals the Pentium that the external system has applied or extracted data from the data bus connections. This signal is used to insert wait states into the Pentium timing. |
| **BREQ** | The **bus request** output indicates that the Pentium has generated a bus request. |
| **BT3–BT0** | The **branch trace** outputs provide bits 2 -0 of the branch target linear address and the default operand size on BT3. These outputs become valid during a branch trace special message cycle. |
| $\overline{BUSCHK}$ | The **bus check** input allows the system to signal the Pentium that the bus transfer has been unsuccessful. |
| $\overline{CACHE}$ | The **cache** output indicates that the current Pentium cycle can cache data. |
| **CLK** | The **clock** is driven by a clock signal that is at the operating frequency of the Pentium. For example, to operate the Pentium at 66 MHz, we apply a 66 MHz clock to this pin. |

| | |
|---|---|
| **D63–D0** | **Data bus** connections transfer byte, word, doubleword, and quadword data between the microprocessor and its memory and I/O system. |
| **D/$\overline{C}$** | **Data/control** indicates that the data bus contains data for or from memory or I/O when a logic 1. If D/$\overline{C}$ is a logic 0, the microprocessor is either halted or is executing an interrupt acknowledge. |
| **DP7–DP0** | **Data parity** is generated by the Pentium and detects its eight memory banks through these connections. |
| **$\overline{EADS}$** | The **external address strobe** input signals that the address bus contains an address for an inquire cycle. |
| **$\overline{EWBE}$** | The **external write buffer empty** input indicates that a write cycle is pending in the external system. |
| **$\overline{FERR}$** | A **floating-point error** is comparable to the $\overline{ERROR}$ line in the 80386 and shows that the internal coprocessor has erred. |
| **$\overline{FLUSCH}$** | The **flush cache** input causes the cache to flush all write-back lines and invalidate its internal caches. If the $\overline{FLUSH}$ input is a logic 0 during a reset operation, the Pentium enters its test mode. |
| **$\overline{FRCMC}$** | The **functional redundancy check** is sampled during a reset to configure the Pentium in the master (1) or checker mode (0). |
| **$\overline{HIT}$** | **Hit** shows that the internal cache contains valid data in the inquire mode. |
| **$\overline{HITM}$** | **Hit modified** shows that the inquire cycle has found a modified cache line. This output is used to inhibit other master units from accessing data until the cache line is written to memory. |
| **HOLD** | **Hold** requests a DMA action. |
| **HLDA** | **Hold acknowledge** indicates that the Pentium is currently in a hold condition. |
| **IBT** | **Instruction branch taken** indicates that the Pentium has taken an instruction branch. |
| **$\overline{IERR}$** | The **internal error** output shows that the Pentium has detected an internal parity error or functional redundancy error. |


| | |
|---|---|
| **$\overline{FLUSCH}$** | The **flush cache** input causes the cache to flush all write-back lines and invalidate its internal caches. If the $\overline{FLUSH}$ input is a logic 0 during a reset operation, the Pentium enters its test mode. |
| **$\overline{FRCMC}$** | The **functional redundancy check** is sampled during a reset to configure the Pentium in the master (1) or checker mode (0). |
| **$\overline{HIT}$** | **Hit** shows that the internal cache contains valid data in the inquire mode. |
| **$\overline{HITM}$** | **Hit modified** shows that the inquire cycle has found a modified cache line. This output is used to inhibit other master units from accessing data until the cache line is written to memory. |
| **HOLD** | **Hold** requests a DMA action. |
| **HLDA** | **Hold acknowledge** indicates that the Pentium is currently in a hold condition. |
| **IBT** | **Instruction branch taken** indicates that the Pentium has taken an instruction branch. |
| **$\overline{IERR}$** | The **internal error** output shows that the Pentium has detected an internal parity error or functional redundancy error. |

| | |
|---|---|
| **INV** | The **invalidation** input determines the cache line state after an inquiry. |
| **IU** | The **U-pipe instruction complete** output shows that the instruction in the U-pipe is complete. |
| **IV** | The **V-pipe instruction complete** output shows that the instruction in the V-pipe is complete. |
| $\overline{\text{KEN}}$ | The **cache enable** input enables internal caching. |
| $\overline{\text{LOCK}}$ | **LOCK** becomes a logic 0 whenever an instruction is prefixed with the LOCK: prefix. This is most often used during DMA accesses. |
| M/$\overline{\text{IO}}$ | **Memory/IO** selects a memory device when a logic 1 or an I/O device when a logic 0. During the I/O operation, the address bus contains a 16-bit I/O address on address connections A15-A3. |
| $\overline{\text{NA}}$ | **Next address** indicates that the external memory system is ready to accept a new bus cycle. |
| **NMI** | **Non-maskable interrupt** requests a non-maskable interrupt, just as on the earlier versions of the microprocessor. |
| **PCD** | The **page cache disable** output shows that the internal page caching is disabled by reflecting the state of the CR3 PCD bit. |
| $\overline{\text{PCHK}}$ | The **parity check** output signals a parity check error for data read from memory or I/O. |
| $\overline{\text{PEN}}$ | The **parity enable** input enables the machine check interrupt or exception. |
| **PRDY** | The **probe ready** output indicates that the probe mode has been entered for debugging. |
| **PWT** | The **page write-through** output shows the state of the PWT bit in CR3. |
| R/$\overline{\text{S}}$ | This pin is provided for use with the Intel Debugging Port and causes an interrupt. |
| **RESET** | **Reset** initializes the Pentium, causing it to begin executing software at memory location FFFFFFF0H. The Pentium is reset to the real mode and the leftmost 12 address connections remain logic 1's (FFFH) until a far jump or far call is executed. This allows compatibility with earlier microprocessors. |