MP UNIT 1
80386DX- Basic Programming Model and Applications Instruction Set

Memory Organization and Segmentation

- Architecture: 8086, 80386
- Descriptor Table
- Data Types
- Registers
- Instruction Format and Operand Selection
- Interrupts and Exceptions

What is Microprocessor

- A microprocessor, sometimes called a ***logic chip***, is a computer processor on a microchip.
- It is also called as **"Heart of Computer."**
- The microprocessor contains all, or most of, the **central processing unit (CPU)** functions.
- A microprocessor is designed to perform arithmetic and logic operations that make use of small number-holding areas called ***registers***.

- Typical microprocessor operations include **adding, subtracting, comparing two numbers, and fetching numbers** from one area to another.

- These operations are the result of a set of instructions that are part of the microprocessor design.

Three basic characteristics differentiate microprocessors:

- **Instruction set**: The set of instructions that the microprocessor can execute.
- **Bandwidth :** The number of bits processed in a single instruction.
- **Clock speed :** Given in megahertz (MHz), the clock speed determines how many instructions per second the processor can execute.

In both cases, the higher the value, the more powerful the CPU.

For example, a 32-bit microprocessor that runs at 50MHz is more powerful than a 16-bit microprocessor that runs at 25MHz.

How It Looks

DYNA86-V4
1801
PRINTER
DYNA BUS
CPU
8086
8087
NDP
RS-232 SERIAL
TIMER
DYNA-86
EVEN
DYNA-86
ODD
MON EVEN
EVEN
RAM ODD
U34 KBRD/DISP
DYNALOG
REV. 4.0

Pinless Microprocessor

intel
DELL

History of Microprocessor

MP	Introduction	Data Bus	Address Bus
4004	1971	4	8
8008	1972	8	8
8080	1974	8	16
8085	1977	8	16
8086	1978	16	20
80186	1982	16	20
80286	1983	16	24
80386	1986	32	32
80486	1989	32	32
Pentium	1993 onwards	32	
Core solo	2006	32	
Dual Core	2006	32	
Core 2 Duo	2006	32	
Core to Quad	2008	32	
I3,i5,i7	2010	64	

Family tree of 80386

Chip	Introduction	Data bus	Address Bus	Memory
4004	1971	4	8	256 Byte
8008	1972	8	8	256 Byte
8080	1974	8	16	64 KB
8086/88	1978	16/8	20	1M
80186/188	1982	16/8	20	1M
80286	1983	16	24	16M:Clock speed is high
80386	DX (1986:not compatibility) SX (1988: mostly used, Not Co-Processor)	DX:32+132 pin SX:16+100 pin	32 24	DX:4G (275,000 transistor) SX:16MB
80486		32	32	Memory Size: 4G +16K cache

80286 vs 80386 vs 80486

Specifications	80286	80386	80486
CPU Speed	6 to 25 MHz	12 to 40 MHz	16 to 100 MHz
Cores	1	1	1
RAM	16MB	4GB	4GB
Functional Units	4	6	9
Pipeline stages	3	3	5
Cache off chip	0	YES (Support)	YES (Support)
Cache on chip	0	0	8 KB
Transistors	134,000	275,000	>1000000

Difference between 80386 SX/DX

80386DX	80386SX
32 bit address bus 32 bit data bus	24 bit address bus 16 bit data bus
Packaged in 132 pin PGA	100 pin flat package
Address 4GB of memory	16 MB of memory

About 8086

- **It is 16 bit processor.** So that it has 16 bit ALU, 16 bit registers and internal data bus and 16 bit external data bus.

- 8086 has 20 bit address lines to access memory. Hence it can access.

2^20 = 1 MB memory location

- **Pipelining:-**8086 uses two stage of pipelining. First is Fetch Stage and the second is Execute Stage.
  - **Fetch stage** that prefetch upto 6 bytes of instructions stores them in the queue.
  - **Execute stage** that executes these instructions.

- Pipelining improves the performance of the processor so that operation is faster.

- **Operates in two modes:-**8086 operates in two modes:
  - **Minimum Mode:** A system with only one microprocessor.
  - **Maximum Mode:** A system with multiprocessor.

- **8086 uses memory banks:-**The 8086 uses a memory banking system. It means entire data is not stored sequentially in a single memory of 1 MB but memory is divided into two banks of 512KB.

- **Interrupts:-**8086 has 256 vectored interrupts.

- **Multiplication And Division:-**8086 has a powerful instruction set. So that it supports Multiply and Divide operation.

- First processor who has implemented **Segmentation** and **Pipelining**.

Architecture of 8086

Architecture of 8086

- The architecture of 8086 includes
  - Arithmetic Logic Unit (ALU)
  - Flags
  - General registers
  - Instruction byte queue
  - Segment registers

EU & BIU

- The 8086 CPU logic has been partitioned into two functional units namely **Bus Interface Unit (BIU) and Execution Unit (EU).**
- The major **reason for this separation is to increase the processing speed** of the processor.
- The **BIU** has to interact with memory and input and output devices in **fetching the instructions and data** required by the EU.
- **EU** is responsible for **executing the instructions of the programs** and to carry out the required processing.

BUS INTERFACE UNIT (BU)

The BIU performs all bus operations for EU.

- Fetching instructions
- Responsible for executing all external bus cycles.
- Read operands and write result.

EXECUTION UNIT (EU)

Execution unit contains the complete infrastructure required to execute an instruction.

Bus Interface Unit

- The BIU has
  - Instruction stream byte queue
  - A set of segment registers
  - Instruction pointer

BIU – Instruction Byte Queue

- 8086 instructions vary from 1 to 6 bytes.
- Therefore fetch and execution are taking place concurrently in order to improve the performance of the microprocessor.
- The BIU feeds the instruction stream to the execution unit through a 6 byte prefetch queue.

BIU – Instruction Byte Queue

- Execution and decoding of certain instructions do not require the use of buses.
- While such instructions are executed, the BIU fetches up to six instruction bytes for the following instructions (the subsequent instructions).
- The BIU store these prefetched bytes in a **first-in-first out** register by name instruction byte queue.
- When the EU is ready for its next instruction, it simply reads the instruction byte(s) for the instruction from the queue in BIU.

Execution Unit

- The Execution Unit (EU) has
  - Control unit
  - Instruction decoder
  - Arithmetic and Logical Unit (ALU)
  - General registers
  - Flag register
  - Pointers
  - Index registers

Execution Unit

- Control unit is **responsible for the co-ordination of all other units of the processor.**

- ALU performs various arithmetic and logical operations over the data.

- The instruction decoder translates the instructions fetched from the memory into a series of actions that are carried out by the EU.

Programmer's Model of 8086

- General Purpose Register

		7 … 0	7 … 0
Accumulator	AX	AH	AL
Base	BX	BH	BL
Count	CX	CH	CL
Data	DX	DH	DL

H: High Order Byte

L: Low Order Byte

- **Segment Registers**

Different Areas in Memory

- **Program memory** – Program can be located anywhere in memory.
- **Data memory** – The processor can access data in any one out of 4 available segments.
- **Stack memory** – A stack is a section of the memory set aside to store addresses and data while a subprogram executes.
- **Extra segment** – This segment is also similar to data memory where additional data may be stored and maintained.

Segment Registers

- **Code Segment (CS)** register is a 16-bit register containing address of 64 KB segment with processor instructions.
- The processor uses CS segment for all accesses to instructions referenced by instruction pointer (IP) register.

- **Stack Segment (SS)** register is a 16-bit register containing address of 64KB segment with program stack.
- By default, the processor assumes that all data referenced by the stack pointer (SP) and base pointer (BP) registers is located in the stack segment.

Segment Registers

- **Data Segment (DS)** register is a 16-bit register containing address of 64KB segment with program data.
- By default, the processor assumes that all data referenced by general registers (AX, BX, CX, DX) and index register (SI, DI) is located in the data segment.

- **Extra Segment (ES)** register is a 16-bit register containing address of 64KB segment, usually with program data.
- By default, the processor assumes that the DI register references the ES segment in string manipulation instructions.

- **Flag Register**

Pin Diagram 8086

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

Pin Diagram 8086

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

8086 can be work in two modes

- Minimum Mode: For single processor systems.
- Maximum Mode: For system with two or more processors.

Depending upon modes signals can be divided into

- Signals having common functions in both modes
- Signals for Minimum Mode
- Signals for Maximum Mode

Signals having common functions in both modes

- **AD15 – AD0: Address /Data Bus (BI)**

  T1 state: Address Bus

  T2,T3,Tw and T4: Data Bus

- **A19/S6 - A16/S3: Address/Status (Output)**

  T1 state : used to address upper 4 bits of address.

  T2, T3, Tw and T4 : Used to output status.

S3 and S4 indicates segment registers being used

S5: Status of Interrupt enable flag updated every clock cycle.

S6: When 8086 Shared system bus, then goes low or goes high.

S4	S3	Register
0	0	ES
0	1	SS
1	0	CS or none
1	1	DS

- **BHE/S7: BHE (Bus High Enable) (Output)**

Low: If transfer is using higher order bytes (AD15-AD8)

High: If transfer is using lower order bytes (AD7-AD0)

BHE	A0	Characteristics
0	0	Whole word
0	1	Upper byte from/to odd address
1	0	Lower byte from/to even address
1	1	None

- **NMI (NON-MASKABLE INTERRUPT) (Input):**

Interrupts can not be avoided.

- **RESET (Input):**

Causes the processor to immediately terminate its present activity.

- **CLK (Input):**

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

- **READY (Input):**

It is the acknowledgement from the addressed memory or I/O device that it will complete the data transfer. Otherwise 8086 will move to wait state.

- **TEST (Input):**

This signal is used by WAIT instruction. Execution will continue, until TEST is low else it will be in idle state. TEST is synchronized internally during each clock cycle.

- **RD (Output):**

This signal remains low when 8086 is reading data from memory or I/O devices.

- **MN/MX : MINIMUM/MAXIMUM (Input):**

Indicates what mode the processor is to operate in.

- **GND: Ground**

To prevent 8086 from thermal heating ,two ground signals are used.

- **VCC (Input):**

+5V power supply pin.

Signals functions in Maximum modes

- **QS1,QS0 (Output):**

Reflects status of instruction queue.

QS1	QS0	Status
0	0	No Operation
0	1	First Byte of Op Code from Queue
1	0	Queue is empty
1	1	Subsequent Byte from Queue

- **S2,S1,S0 (Output):**

Indicates type of transfer takes place during current bus cycle.

S2	S1	S0	Machine Cycle
0	0	0	Interrupt ACK
0	0	1	I/O Read
0	1	0	I/O Write
0	1	1	Halt

S2	S1	S0	Machine Cycle
1	0	0	Instruction Fetch
1	0	1	Memory Read
1	1	0	Memory Write
1	1	1	Inactive-Passive (In T3)

- **LOCK (Input):**

Bus is not used by another processor, current system have locked the rights of system bus.

- **RQ/GT1 & RQ/GT0: (Bus Request/Bus Grant) (BI)**

Using bus request signal another master can request a system bus and processor sends a grant signal as a acceptance.

RQ/GT0 is having greater priority than RQ/GT1.

Signals functions in Minimum modes

- **INTA (Interrupt Ack Output):**

It indicates recognition of an interrupt request. It then sends two negative going pulse in next to clk cycles, first informs interface that its interrupt request in accepted, in next pulse interface replies with interrupt type.

- **ALE (Address Latch Enable) output:**

It is provided to demultiplex AD0-AD15 to A0-A15 and D0-D15.

- **DEN (Data Enable) output:**

This signal informs transceivers that 8086 is ready to send or receive data.

- **DT/R (Data Transmit/Receive):**

It is used to control the direction of data flow through the transceiver.

High: 8086 is transmitting data

Low: 8086 is receiving data

- **M/IO :**

It is used to distinguish a memory access from an I/O access.

High: Memory Access

Low: I/O Access

- **WR (WRITE):**

Indicates that the processor is performing a writing data to memory or I/O.

- **HOLD/ HLDA:**

HOLD: indicates that another master is requesting a local bus, on request processor replies High HLDA signal as a Ack.

Minimum-Mode and Maximum-Mode System (cont.)

Common signals		
Name	Function	Type
AD7-AD0	Address/data bus	Bidirectional, 3-state
A15-A8	Address bus	Output, 3-state
A19/S6-A16/S3	Address/status	Output, 3-state
MN/$\overline{MX}$	Minimum/maximum Mode control	Input
$\overline{RD}$	Read control	Output, 3-state
$\overline{TEST}$	Wait on test control	Input
READY	Wait state control	Input
RESET	System reset	Input
NMI	Nonmaskable Interrupt request	Input
INTR	Interrupt request	Input
CLK	System clock	Input
V_{CC}	+5 V	Input
GND	Ground	

Signals common to both minimum and maximum mode

Minimum-Mode and Maximum-Mode System (cont.)

Minimum mode signals ($MN/\overline{MX} = V_{CC}$)		
Name	Function	Type
HOLD	Hold request	Input
HLDA	Hold acknowledge	Output
$\overline{WR}$	Write control	Output, 3-state
$IO/\overline{M}$	IO/memory control	Output, 3-state
$DT/\overline{R}$	Data transmit/receive	Output, 3-state
$\overline{DEN}$	Data enable	Output, 3-state
$\overline{SSO}$	Status line	Output, 3-state
ALE	Address latch enable	Output
$\overline{INTA}$	Interrupt acknowledge	Output

Unique minimum-mode signals

Segmentation in 8086

- The process of dividing memory is called Segmentation.
- Intel 8086 has 20 lines address bus.
- With 20 address lines, the memory that can be addressed is 2^20 bytes

  2^20 = 1,048,576 bytes (1 MB).
- 8086 can access memory with address ranging from 00000 H to FFFFF H.

FFFFFH
64 KB
CODE SEGMENT
XXXXOH
STACK SEGMENT
+ OFFSET
SEGMENT
REGISTER FILE
CS
SS
DS
ES
DATA SEGMENT
EXTRA DATA SEGMENT
00000H
231455-3

- In 8086, memory has four different types of segments.

These are:

  - Code Segment
  - Data Segment
  - Stack Segment
  - Extra Segment

- These registers are 16-bit in size.
- Each register stores the base address (starting address) of the corresponding segment.
- Because the segment registers cannot store 20 bits, they only store the upper 16 bits.

Logical to physical address Translation in 8086

- The 20-bit address of a byte is called its **Physical Address.**
- But, it is specified as a **Logical Address.**
- Logical address is in the form of:

  Base Address : Offset
- Offset is the displacement of the memory location from the starting location of the segment.

Example

- The value of Data Segment Register (DS) is 2222 H.
- To convert this 16-bit address into 20-bit, the BIU appends 0H to the LSBs of the address.
- After appending, the starting address of the Data Segment becomes 22220H.
- If the data at any location has a logical address specified as:

  2222 H : 0016 H
- Then, the number 0016 H is the offset. 2222 H is the value of DS.

- To calculate the effective address of the memory, BIU uses the following formula:

  Effective Address =

  Starting Address of Segment + Offset

- To find the starting address of the segment, BIU appends the contents of Segment Register with 0H.
- Then, it adds offset to it.

  Therefore:

```
EA =   22220 H
     + 0016 H
     ------------
       22236 H
```

Segment	Offset Registers	Function
CS	IP	Address of the next instruction
DS	BX, DI, SI	Address of data
SS	SP, BP	Address in the stack
ES	BX, DI, SI	Address of destination data (for string operations)

Question

The contents of the following registers are:

- CS = 1111 H
- DS = 3333 H
- SS = 2526 H
- IP = 1232 H
- SP = 1100 H
- DI = 0020 H

Calculate the corresponding physical addresses for the address bytes in CS, DS and SS.

1. CS = 1111 H

- The base address of the code segment is 11110 H.
- Effective address of memory is given by 11110H + 1232H = **12342H.**

2. DS = 3333 H

- The base address of the data segment is 33330 H.
- Effective address of memory is given by 33330H + 0020H = **33350H.**

3. SS = 2526 H

- The base address of the stack segment is 25260 H.
- Effective address of memory is given by 25260H + 1100H = **26350H.**

New in 80386

- **Data bus = 32bit**, all registers (except Segment) of 32 bit and Eflags is also of 32 bit.
- **Address Bus = 32 bit**. (4 GB Memory)
- Enhanced **Memory Management Unit**.
- Supports **Virtual addressing**.
- **Faster execution** of arithmetic operations.
- Works in :-
    1. Real Mode (8086)
    2. Protected Mode
    3. Virtual 8086 Mode
- Additional Interrupts in IVT.

FEATURES

- Manufactured using Intel's complementary High-performance Metal-oxide-semiconductor 3 process.
- 8 General Purpose Registers of 32-bit .
- 32-bit Address and Data Bus.
- Supports 8 bit,16 bit,32 bit data.
- Prefetch Queue of 16B.
- Very Large address space i.e VM of 64 TB and PM of 4GB.
- Supports Segmentation and Paging.

- 4 levels of Protection.
- Uses 3-stage Pipelines.
- Supports Multitasking with Protection.
- On chip cache memory for TLB.
- Pipelined Instruction Execution.
- Memory Management unit.
- High speed numeric support via 80287 and 80387 coprocessor.
- It can operate in Real , Protected and Virtual 8086 mode.

Architecture of 80386

PLA: Programmable Logic Array

- The Internal Architecture of 80386 is divided into 3 sections.

1) Central Processing Unit

2) Memory Management Unit

3) Bus Interface Unit

Central Processing Unit

- Central processing unit is further divided into **Execution Unit and Instruction Unit**.

- The **Execution Unit** has 8 General purpose and 8 Special purpose registers which are either used for handling data or calculating offset addresses.

- The **Instruction Unit** decodes the opcode bytes received from the 16-byte instruction code queue and arranges them in a 3- instruction decoded instruction queue.

- After decoding them pass it to the control section for deriving the necessary control signals. The **barrel shifter increases the speed of all shift and rotate operations.**

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

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

Memory Management Unit

- The Memory management unit consists of a **Segmentation Unit** and a **Paging Unit**.

- Segmentation unit allows the use of two address components, viz. segment and offset for relocability and sharing of code and data.

- Segmentation unit allows segments of size 4 Gbytes at max.

- The **Paging unit organizes the physical memory in terms of pages of 4 kbytes size each.**

- Paging unit works under the control of the segmentation unit, i.e. each segment is further divided into pages.

- The virtual memory is also organizes in terms of segments and pages by the memory management unit.

- The Segmentation unit provides a **4 level Protection Mechanism** for protecting and isolating the system code and data from those of the application program.

- Paging unit converts **Linear addresses into Physical addresses.**

- The **Control and Attribute PLA checks the privileges at the page level.** Each of the pages maintains the paging information of the task.

- The **Limit and Attribute PLA checks segment limits and attributes at segment level** to avoid invalid accesses to code and data in the memory segments.

Bus Interface Unit

- The Bus control unit has a **priority-izer to resolve the priority of the various bus requests.**

- This controls the access of the bus. The address driver drives the bus enable and address signal A0–A31.

- The pipeline and dynamic bus sizing unit handle the related control signals.

- The data buffers interface the internal data bus with the system bus.

Register Set

- The 80386 contain **total 16 registers,**

  These registers grouped as:

  1. General Purpose
  2. Segment
  3. Status and Instruction
  4. Control Registers
  5. System Address Registers
  6. Debug Registers
  7. Test Registers

General Purpose Registers

Segment Registers

15	7	0
CS (CODE SEGMENT)		
SS (STACK SEGMENT)		
DS (DATA SEGMENT)		
ES (DATA SEGMENT)		
FS (DATA SEGMENT)		
GS (DATA SEGMENT)		

Status and IP

Flag Registers

- **VM (Virtual 8086 Mode):** If set while the 80386 is in Protected Mode, the 80386 will switch to Virtual 8086 operation.
- The VM bit can be set only in Protected Mode, by the IRET instruction (if current privilege level is 0).

- **RF (Resume Flag):** The RF flag is used in conjunction with the debug register breakpoints.
- When RF is set, it causes any debug fault to be ignored on the next instruction.

- **NT (Nested Task):** This flag applies to Protected Mode.

- NT is set to indicate that the execution of this task is nested within another task

- The value of NT in EFLAGS is tested by the IRET instruction to determine whether to do an inter-task return or an intra-task return.

IOPL (Input / Output Privilege Level)

- This two-bit field applies to Protected Mode.

  IOPL indicates the numerically maximum **CPL (Current Privilege Level)** value permitted to execute I/O instructions without generating an Exception.

- It also indicates the maximum CPL value can be altered by IF (INTR Enable Flag) bit when new values are popped into the EFLAG register.

- **IF (INTR Enable Flag):** The IF flag, when set, allows recognition of external interrupts signaled on the INTR pin.

- **TF (Trap Enable Flag):** When TF is set, the 80386 generates an exception 1 trap after the next instruction is executed.

- When TF is reset, exception 1 traps occur only as a function of the breakpoint addresses loaded into debug registers DR0-DR3.

- **OF (Overflow Flag):** It is set if the operation resulted in a signed overflow. Signed overflow occurs when the operation resulted in carry/borrow into the sign bit (high-order bit) of the result.

- **DF (Direction Flag):** DF defines whether ESI and/or EDI registers post-decrement or post-increment during the string instructions.
- Post-decrement occurs if DF is set.

Flags

- The **Arithmetic Instructions** use CF, SF, ZF, AF, PF, CF.

- The control flag DF controls **String** instruction.

- Clearing DF flag causes string instructions to auto increment or to process string from low to high address.

Hidden Registers/ Program Invisible Registers/ Special Registers

Control Register

- The CR0 is identical to the **MSW (Machine Status Word)** found in 80286 except that, this is **32 bit wide.**

- CR1 is not used in 80386.

- CR2 hold the **Linear Page Address of the last page accessed** before a Page Fault Interrupt.

- CR3 holds the **Page Directory Base Address/Register (PDBA/R)**.

PG (Paging Enable):

Selects **Page Table Translation of Linear Addresses into Physical Addresses when PG=1**. Page table translation allows any linear address to be assigned any physical memory location.

ET (Processor Extension Type):

Selects the 80287 coprocessor when ET=0 or the **80387 coprocessor when ET=1**.

This bit was installed because there was no 80387 available when the 80386 first appeared.

TS (Task Switch):

This bit is **set** to Indicate that the **80386 has switched tasks** (In protected mode, changing the content of Task Register places a 1 in TS).

EM (Emulate Processor Extension):

The emulate bit **set** to cause a **Type 7 Interrupt for each ESC instruction.** (ESC instructions are used to encode instruction for the 80387 coprocessor. So when it is set co-processor generates coprocessor not available fault).

[Type 7 Interrupt: Co-Processor Not Available.]

MP (Monitor Processor Extension):

MP bit is **reset** to indicate that the **Arithmetic Coprocessor is present** in the system.

PE (Protection Enable):

PE bit is **set** to select the **Protected Mode of operation for the 80386.** It may also cleared to reenter the real mode.

Debug Register

31 … 16 15 … 0	
BREAKPOINT 0 LINEAR ADDRESS	DR0
BREAKPOINT 1 LINEAR ADDRESS	DR1
BREAKPOINT 2 LINEAR ADDRESS	DR2
BREAKPOINT 3 LINEAR ADDRESS	DR3
Intel reserved. Do not define.	DR4
Intel reserved. Do not define.	DR5

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
0																BT	BS	BD	0	0	0	0	0	0	0	0	0	B3	B2	B1	B0	DR6
LEN 3	R 3	W 3	LEN 2	R 2	W 2	LEN 1	R 1	W 1	LEN 0	R 0	W 0	0	0	GD	0	0	0	GE	LE	G3	L3	G2	L2	G1	L1	G0	L0					DR7

- These are **used to control Debug Functions.**

- The first four debug register contain **32 bit Linear Break Point Addresses.**

- The breakpoint addresses, which may locate an instruction, **are constantly compared with the addresses generated by the program.**

- If a match occurs, **the 80386 will cause a type 1 interrupt (trap or debug) to occur**, if directed by debug registers DR6 or DR7.

BT:

If set the **debug interrupt was caused by a Task Switch.**

BS:

If set the **debug interrupt was caused by the TF bit** in the flag register.

BD:

If set the **debug interrupt was caused by an attempt to read the debug register with the GD bit set.**

The GD bit protects access to the debug registers.

B3-B0:

Indicate which of the 4 debug breakpoints addresses caused the debug interrupt.

LEN:

Defines the **size of access at the breakpoint** address as 00 (byte), 01 (word), 10 (currently not used) or 11 (double word).

RW:

Selects the cause of action that that enabled breakpoint address as 00 (instruction access), 01 (data write), 10 (Currently Not Used), 11 (data read n write).

Test Register

31 … 12	11											0	
LINEAR ADDRESS	V	D	D #	U	U #	W	W #	0	0	0	0	C	TR6
PHYSICAL ADDRESS	0	0	0	0	0	0	0	P L	REP		0	0	TR7

Figure 13-2. 80386 Block Diagram (# indicates active low)

(Reprinted by permission of Intel Corporation, Copyright Intel Corp. 1992)

- **CLK2:** The input pin provides the basic system clock timing for the operation of 80386.

- **D0 – D31:** These 32 lines act as bidirectional data bus during different access cycles.

- **A31 – A2:** These are upper 30 bit of the 32- bit address bus.

- **BE0# to BE3#**: The 32- bit data bus supported by 80386 and the memory system of 80386 can be viewed as a 4-byte wide memory access mechanism.

BE0,BE1,BE2 and BE3 are active low signals.

Bus Control

- **<u>ADS#: (Address Data Strobe)</u>**
  - Active when **issued a valid request.**
  - The address status output pin indicates that the address bus and **Bus Cycle Definition Pins**(W/R#, D/C#, M/IO#, BE0# to BE3#) are carrying the respective valid signals.

<u>BS16#:</u> The bus size – 16 input pin **allows the interfacing of 16 bit devices with the 32 bit wide 80386 data bus.**

- **READY#:** The ready signals indicates to the CPU that the **previous bus cycle has been terminated** and **the bus is ready for the next cycle.** So it is ready to service next request. The signal is used to insert WAIT states in a bus cycle and is useful for interfacing of slow devices with CPU.

- **NA#:** NEXT ADDRESS is used to request address pipelining.

Bus Arbitration

- **<u>HOLD</u>:** The Bus hold input pin enables the other bus masters (eg. DMA) to gain control of the system bus if it is asserted.

- **<u>HLDA:</u>** The bus hold acknowledge output indicates that a valid bus hold request has been received and the bus has been relinquished by the CPU.

Interrupts

- **INTR:** This interrupt pin is a mask-able interrupt, that can be masked using the IF of the flag register.

- **NMI:** A valid request signal at the non-mask-able interrupt request input pin internally generates a non- mask-able interrupt of Type 2.

- **RESET:** A high at this input pin suspends the current operation and restart the execution from the starting location.

Co-Processor Signaling

- **BUSY#:** The busy input signal indicates to the CPU that the **Coprocessor is Busy** with the allocated task.

- **ERROR#:** The error input pin indicates to the CPU that the **coprocessor has encountered an error** while executing its instruction.

- **PEREQ: (Processor extension request)**

  Output signal indicates to the **CPU to fetch data.**

A bus cycle definition pins

- **<u>LOCK#</u>**: BUS LOCK is a bus cycle definition pin that indicates that system have locked system bus of other peripherals.

- **<u>W/R#:</u>** WRITE/READ is a bus cycle definition pin that distinguishes write cycles from read cycles.

- **<u>D/C#:</u>** DATA/CONTROL is a bus cycle definition pin that distinguishes data cycles, either memory or I/O, from control cycles which are: Interrupt Acknowledge, Halt, and Instruction Fetching.

- **<u>M/IO#:</u>** MEMORY I/O is a bus cycle definition pin that distinguishes memory cycles from input/output cycles.

Power Connection Pins

- **VCC:** These are system power supply lines.

- **GND:**

Global Descriptor Table

- Every protected mode 80386 system contains a Global Descriptor Table.
- The **GDT holds descriptors that are available to all the tasks** in a system.
- Except for descriptors that control Interrupts or Exceptions.
- Generally, the GDT contains 2 types of descriptors:
    - **System**
    - **Non System**

- The GDT contains **descriptors for segments that are common to all tasks.**

- Generally the **GDT contains code and data segments** used by the operating systems and **Task State Segments**, and descriptors for the **LDTs in a system.**

- The first slot of the Global Descriptor Table corresponds to the **Null Selector** and is not used.

Figure 2-6. The Global Descriptor Table

Linux's GDT	*Segment Selectors*	*Linux's GDT*	*Segment Selectors*
null	0x0	TSS	0x80
reserved		LDT	0x88
reserved		PNPBIOS 32-bit code	0x90
reserved		PNPBIOS 16-bit code	0x98
not used		PNPBIOS 16-bit data	0xa0
not used		PNPBIOS 16-bit data	0xa8
TLS #1	0x33	PNPBIOS 16-bit data	0xb0
TLS #2	0x3b	APMBIOS 32-bit code	0xb8
TLS #3	0x43	APMBIOS 16-bit code	0xc0
reserved		APMBIOS data	0xc8
reserved		not used	
reserved		not used	
kernel code	0x60 (__KERNEL_CS)	not used	
kernel data	0x68 (__KERNEL_DS)	not used	
user code	0x73 (__USER_CS)	not used	
user data	0x7b (__USER_DS)	double fault TSS	0xf8

Local Descriptor Table

- Operating systems are generally designed so that **each task has a separate Local Descriptor Table.**

- LDTs **provide a way for isolating a given task's code and data segments from the rest of the operating system**, while the GDT contains descriptors for segments which are common to all tasks.

- The LDT may contain only Code, Data, Stack, Task Gate, and Call Gate Descriptors.

- Unlike the 6 byte GDT or IDT registers which contain a base address and limit, the **visible portion of the LDT register contains only a 16-bit selector.**

Interrupt Descriptor Table

- The Interrupt Descriptor Table contains the descriptors that point to the location of up to **256 Interrupt Service Routines.**

- The IDT can only contain **Trap Gates, Task Gates, and Interrupt Gates.**

- **The IDT should be at least 256 bytes in size** so it can hold descriptors for the 32 Intel-reserved interrupts.

- **Every interrupt** used by the system **must have an entry in the IDT.**

MEMORY
GATE FOR INTERRUPT #n
GATE FOR INTERRUPT #n-1
GATE FOR INTERRUPT #1
GATE FOR INTERRUPT #0
INTERRUPT DESCRIPTOR TABLE (IDT)
CPU
15
0
IDT LIMIT
IDT BASE
31
0
INCREASING MEMORY ADDRESSES
231630-58

Fundamental Data Types

Byte Integer

Word Integer

BCD

+N
+1
0
7
0
7
0
7
0
BINARY CODED
DECIMAL (BCD)
...
BCD
DIGIT N
BCD
DIGIT 1
BCD
DIGIT 0

Packed BCD

+N
+1
0
7
0
7
0
7
0
PACKED
BCD
MOST
SIGNIFICANT
DIGIT
LEAST
SIGNIFICANT
DIGIT

String

Pointers

+3
+2
+1
0
31
0
NEAR 32-BIT
POINTER
OFFSET

+5
+4
+3
+2
+1
0
48
0
FAR 48-BIT
POINTER
SELECTOR
OFFSET

Interrupt

- In system programming, an **interrupt** is a signal to the processor emitted by hardware or software indicating an **event that needs immediate attention.**
- Interrupt causes transfer of control to an **interrupt service routine (ISR).** ISR is also called a handler.
- When the ISR is completed, the original program resumes execution.
- Interrupts provide an efficient way to handle unanticipated events.

Interrupt Process (from three potential sources)
Hardware
Processor
Software
Interrupt Request (IRQ) sent from device to processor
Exception / Trap sent from processor to processor
Software Interrupt instruction loaded by processor
Processor halts thread execution
Processor saves thread state
Processor executes interrupt handler
Processor resumes thread execution

Interrupts Vs. Procedures

Interrupts

- Initiated by both *software* and *hardware*
- Can handle *anticipated* and *unanticipated* internal as well as external events
- ISRs or interrupt handlers are memory resident
- Use numbers to identify an interrupt service
- (E)FLAGS register is saved automatically

Procedures

- Can only be initiated by *software*
- Can handle *anticipated* events that are coded into the program
- Typically loaded along with the program
- Use meaningful names to indicate their function
- Do not save the (E)FLAGS register

Interrupt Sources

- **Hardware Interrupts:** A hardware interrupt is an electronic alerting signal sent to the processor from an **external device**, either a part of the computer itself such as a Printer.

- Two types of this interrupt:
  - NMI
  - INTR: If IF flag=0.

- **Software Interrupt:** A software interrupt is caused either by an **exceptional condition in the processor itself**, or a **special instruction** in the instruction set which causes an interrupt when it is executed.
- Eg.

  DOS INT 21H, BIOS INT 10H.

  INT 00 (divide error)

  INT 01 (single step)

  INT 03 (breakpoint)

  INT 04 (signed number overflow)
- Software interrupt instructions function similarly to **subroutine calls** and are used for a variety of purposes.
- For example, computers often use software interrupt instructions to communicate with the disk controller to request data be read or written to the disk.

- **Error Conditions**: Events occurring during program execution that are exceptional enough that they cannot be handled within the program itself.

- Eg. Divide by zero error, System aborts, system faults.

- Also called as Trap or Exception.

What if two interrupts occur at same time?

When different type of interrupt (ie. software, NMI, INTR or automatic interrupts) occur at the same time, the one with the highest priority is handled first. Intel microprocessors use the following order of priority:

Interrupt Type	Priority
divide error interrupt, INT n, INT0	highest
NMI	↑
INTR	↓
TRAP flag (single step)	lowest

- **If several INTR are generated from different peripherals simultaneously, it is necessary to decide their priority** and send the INTA signal only to the highest priority peripheral.
- If more than one peripheral receive an INTA at the same time they will both try to put the interrupt number onto the data bus simultaneously.
- Two different methods can be used to establish the priority of interrupt requests from different peripherals

1. **Polling (and daisy chaining)**
2. **Interrupt priority management hardware**

Exceptions and Interrupts

- Interrupts and exceptions are special kinds of control transfer; they work somewhat like unprogrammed CALLs.
- They alter the normal program flow to handle external events or to report errors or exceptional conditions.
- **Interrupts** are used to handle asynchronous events external to the processor.
- **Exceptions** handle conditions detected by the processor itself in the course of executing instructions.

- There are two sources for external interrupts and two sources for exceptions:

1. Interrupts

- Maskable interrupts, which are signalled via the INTR pin.
- Nonmaskable interrupts, which are signalled via the NMI (Non-Maskable Interrupt) pin.

2. Exceptions

- Processor detected. These are further classified as **Faults, Traps and Aborts.**
- Programmed. The instructions **INT 0, INT 3, INT n, and BOUND** can trigger exceptions. These instructions are often called "Software Interrupts", but the processor handles them as exceptions.

Identifying Interrupts

- Each different type of **interrupt or exception have given a unique identification number.**

- The **NMI** and **the exceptions** recognized by the processor are assigned predetermined identifiers in the **range 0 through 31**.

- Not all of these numbers are currently used by the 80386; they are reserved for future.

- The **identifiers of the Maskable Interrupts are determined by external Interrupt Controllers (eg. 8259A PIC)** and communicated to the processor during the processor‘s interrupt-acknowledge sequence.

- The numbers assigned by an 8259A PIC can be specified by software. Any numbers in the **range 32 through 255 can be used.**

- **Exceptions are classified as Faults, Traps, or Aborts depending on the way they are reported** and whether restart of the instruction that caused the exception is supported.

- **Faults**
  - **Faults are exceptions that are reported "before"** the instruction causing the exception.
  - Faults are either detected before the instruction begins to execute, or during execution of the instruction.
  - If detected during the instruction, the instruction is automatically restarted by the system.
  - Eg. Page Fault.

- **Traps**
  - A trap is an exception that is **reported at the instruction boundary** immediately after the instruction in which the exception was detected.
  - Trap happens after or during the execution of instruction.
  - Eg. Debugger breakpoint, Overflow situation, Division by zero or Invalid memory access.

- **Aborts**
  - An abort is an exception that permits **neither precise location of the instruction** causing the exception nor restart of the program that caused the exception.
  - Aborts are used to report severe errors, such as hardware errors and inconsistent or illegal values in system tables.
  - Eg. Blue screen error because of RAM or ROM.

Function	Interrupt Number	Instruction Which Can Cause Exception
Divide Error	0	DIV, IDIV
Debug Exception	1	any instruction
NMI Interrupt	2	INT 2 or NMI
One Byte Interrupt	3	INT
Interrupt on Overflow	4	INTO
Array Bounds Check	5	BOUND
Invalid OP-Code	6	Any Illegal Instruction
Device Not Available	7	ESC, WAIT
Double Fault	8	Any Instruction That Can Generate an Exception
Coprocessor Segment Overrun	9	ESC
Invalid TSS	10	JMP, CALL, IRET, INT

Segment Not Present	11	Segment Register Instructions
Stack Fault	12	Stack References
General Protection Fault	13	Any Memory Reference
Intel Reserved	15	
Page Fault	14	Any Memory Access or Code Fetch
Coprocessor Error	16	ESC, WAIT
Intel Reserved	17–31	
Two Byte Interrupt	0–255	INT n

Divide Error (INT 0)

- The divide-error fault occurs during a DIV or an IDIV instruction when the **Divisor is zero**.

Debug Exceptions (INT 1)

- The processor triggers this interrupt for any of a number of conditions; whether the exception is a fault or a trap depends on the condition:
  - Instruction Address Breakpoint Fault.
  - Data Address Breakpoint Trap.
  - General Detect Fault. (attempt is made to use the debug registers at the same time that 80386 is using them) (BD)
  - Single-step Trap.
  - Task-switch Breakpoint Trap. (BT)
- The processor does not push an error code for this exception. **An exception handler can examine the debug registers to determine which condition caused the exception.**

Breakpoint (INT 3)

- Exceptions generated because of breakpoints.

Overflow (INT 4)

- This trap occurs when the processor encounters an INTO instruction and the OF (overflow) flag is set. {**INTO** – Used to interrupt the program during execution if OF = 1}

- Since signed arithmetic and unsigned arithmetic both use the same arithmetic instructions, the processor cannot determine which is intended and therefore does not cause overflow exceptions automatically. Instead it merely sets OF when the results.

Bounds Check (INT 5)

- This fault occurs when the processor, while **executing a BOUND instruction**, finds that the operand exceeds the specified limits.

- **BOUND ensures that a signed array index is within the limits** specified by a block of memory consisting of an upper and a lower bound.

Invalid op-code (INT 6)

- The Invalid Opcode exception occurs when the **processor tries to execute an invalid or undefined opcode**, or an instruction with invalid prefixes.

- This exception also occurs when the **type of operand is invalid for the given opcode.** Example include an intersegment JMP referencing a register operand.

No Math Unit Available (INT 7)

- It is triggered whenever a floating point instruction is being executed with the **EM** or **TS bit of MSW set.**

- It can also be generated when a WAIT instruction is detected with both MP and TS bits of MSW set.

- Wait instruction causes a processor to wait till coprocessor completes its task.

Double Fault Exception (INT 8)

- A **Double Fault** exception occurs if the processor encounters a problem while trying to service a pending interrupt or exception.

- An example situation when a double fault would occur is **when an interrupt is triggered but the segment in which the interrupt handler resides is invalid.**

- If the processor encounters a problem when calling the double fault handler, a **Triple Fault** is generated and the processor shuts down

Coprocessor Segment Overrun (INT 9)

- This exception is raised in protected mode if the 80386 detects a **Page or Segment Violation** while transferring the middle portion of a coprocessor operand to the NPX. This exception is avoidable.

- Math **Coprocessor** is known as **NPX, NDP, FPU**. Numeric Processor Extension (**NPX**), Numeric Data Processor (**NDP**), Floating Point Unit (**FPU**).

Invalid Task State Segment (INT 10)

- Its generated by the following conditions:

I. A illegal back link in a Task State Segment.
II. The TSS containing an illegal CS,DS,ES or FS value.
III. The TSS indicating an invalid privileged stack is not valid during inter-level call.
IV. The TSS is too small.
V. An invalid or not present LDT in a TSS.

Not Present (INT 11)

- It is generated by trying to load the CS,DS,ES,SS,FS,GS or a task register with an operand that is valid except for **being marked Not Present**. (P bit from descriptor is reset)

Stack Exception (INT 12)

- A stack fault occurs in either of two general conditions:
  - As a **result of a Limit Violation** in any operation that refers to the SS register. This includes stack-oriented instructions such as POP, PUSH, ENTER, and LEAVE.
  - When attempting to load the SS register with a **descriptor that is marked not-present** but is otherwise valid.

General Protection (INT 13)

- It is activated for all protection exceptions that are not specifically covered by other exceptions.

A. A jump to data segment with high(numerically) privilege level.
B. Writing to read only segment.
C. Attempting to address a memory location with an offset address that exceed the limit for the specified segment (**Invalid Offset**).
D. Putting an address into SS for a read only segment when the address come from a Task State segment.

Page Fault (INT 14)

- It is triggered only in **Protected or Virtual** mode, is generated when page fault occurs.
- The returned error code contains following information:

a) **If bit 0 is reset**, the exception was generated by a <u>Page that is not present</u>.

b) **If bit 0 set**, the exception was generated by a <u>Page Level Violation</u>.

c) **If bit 1 is reset**, the exception was caused by an illegal read access.

d) **If bit 1 is set**, the exception was caused by an illegal write access.

e) **If bit 2 is reset**, the exception was generated at user level.

f) **If bit 2 is set**, the exception was generated at supervisor level.

Math Fault (INT 16)

- Mainly generated by co-processor.
- It is activated whenever the **ERROR#** input pin detect an error.
- This test is performed at the beginning of most floating point instructions, and when a WAIT instruction is executed with the **EM (Emulate Processor Extension) bit of MSW reset to 0, which will generate INT 7.**

- Assembly language program
  - Assembly language program (.asm) file—known as source code
  - Converted to machine code by a process called assembling
  - Assembling performed by a software program—an 80x86 assembler
  - Machine (object) code that can be run is output in the executable (.exe) file
  - Source listing output in (.lst) file—printed and used during execution and debugging of program
- DEBUG—part of disk operating system (DOS) of the PC
  - Permits programs to be assembled and disassembled
  - Line-by-line assembler
  - Also permits program to be run and tested

Procedures

Although PROC_A is called hundred times in the main program, the procedure is instantiated only once.

Why Procedures?

The Procedure simplifies the debugging process in the program.

Nested Procedures

Macros

Macros
are similar to a function
are faster than executing a Procedure
can take parameters

Macros as Inline codes

Difference between Macro and Procedure

Procedures and Macros are two different constructs that reduces the number of errors in your program.

How to define macro

section .data

```
msg: db "hello",10
len: equ $-msg
```

Section .bss

```
count: resb 2
```

```
%macro print 2
Mov rax,1
Mov rdi,1
Mov rsi, %1
Mov rdx, %2
Syscall
%endmacro
```

Section .text

```
Global main
Main:
-
print msg,len
-
-
print msg,len
-
-
-
; code of addition and result stored in COUNT variable
print count,2
-
-
Mov rax,60
Mov rdi,0
syscall
```

Stack

Stack Segment
Bottom of Stack
(Initial ESP Value)
Local Variables
for Calling
Procedure
The Stack Can Be
16 or 32 Bits Wide
Parameters
Passed to
Called
Procedure
The EBP register is
typically set to point
to the return
instruction pointer.
Frame Boundary
Return Instruction
Pointer
EBP Register
ESP Register
Top of Stack
Pushes Move the
Top Of Stack to
Lower Addresses
Pops Move the
Top Of Stack to
Higher Addresses

Directives

- There are some instructions in the assembly language program **which are not a part of processor instruction set**.

- These instructions are instructions to the **assembler, linker and loader**. These are referred to as pseudo-operations or as assembler directives.

- **Different Data Types in ALP**
  - DB – Define Byte
  - DW – Define Word
  - DD – Define Doubleword
  - DQ – Define Quadword
  - DT – Define Ten Bytes

- **Different sections in ALP**
  - Section .data
  - Section .bss
  - Section .text

- General structure of an assembly language statement

```
LABEL:     INSTRUCTION      ;COMMENT
```

  - Label—address identifier for the statement
  - Instruction—the operation to be performed
  - Comment—documents the purpose of the statement
  - Example:

```
START:   MOV  AX, BX   ; Copy BX into AX
```

- Other examples:

```
INC SI     ;Update pointer
ADD  AX, BX
```

  - Few instructions have a label—usually marks a jump to point
  - Not all instructions need a comment

- Each instruction is represented by a mnemonic that describes its operation—called its operation code (opcode)
  - MOV = move → data transfer
  - ADD = add → arithmetic
  - AND = logical AND → logic
  - JMP = unconditional jump → control transfer
- Operands are the other parts of an assembly language Instructions
  - Identify whether the elements of data to be processed are in registers or memory
    - Source operand– location of one operand to be process
    - Destination operand—location of the other operand to be processed and the location of the result

Register File

ALP Constructs

1. Basic Data Types

Data Types:

- Byte (8-bit)
- Word (16-bit)
- Double word (32-bit)
- Quadword (64-bit)
- Ten bytes (80-bit)

2. Data Types

1. Definition directives
   - **db** (define byte)
   - **dw** (define word)
   - **dd** (define double word)
   - **dq** (define quad word)
   - **dt** (define ten bytes)

2. Declaration directives
   - **resb** (reserve byte)
   - **resw** (reserve word)
   - **resd** (reserve double word)
   - **resq** (reserve quad word)

3. Memory addressing directives
   - byte
   - word
   - dword
   - qword

Byte Ordering in Computer Memory (Data definition)

1. Little endian machine
   - Stores data **little-end first**
   - **Least significant byte** at smallest address
   - Example: Intel processors (all x86 processors)

2. Big endian machine
   - Stores data **big-end first**
   - **Most significant byte** at smallest address
   - Example: IBM processors (Power PC)

Byte Ordering in Computer Memory – Data Definition (Continued…)

Little endian

Memory location	Data
1000000A h	
10000009 h	12
10000008 h	34
10000007 h	56
10000006 h	78
10000005 h	A9
10000004 h	5C
10000003 h	CD
10000002 h	FE
10000001 h	
10000000 h	

Big endian

Memory location	Data
1000000A h	
10000009 h	FE
10000008 h	CD
10000007 h	5C
10000006 h	A9
10000005 h	78
10000004 h	56
10000003 h	34
10000002 h	12
10000001 h	
10000000 h	

Qnumber dq 12345678A95CCDFE h

Memory addressing

section .data

num: dq 9828919849096878h

section .bss

name: resb 8

<u>Memory addressing:</u>

mov al, byte[num] ; al = 78

mov ax , word [num] ; ax = 6878

mov eax , dword [num] ; eax = 49096878

mov rax , qword [num]

; rax = 9828919849096878

Memory location	Data
1000000A h	
10000009 h	98
10000008 h	28
10000007 h	91
10000006 h	98
10000005 h	49
10000004 h	09
10000003 h	68
10000002 h	78
10000001 h	
10000000 h	

A. Data Movement Instructions

- These instructions **provide convenient methods for moving bytes, words, or doublewords of data** between memory and the registers of the base architecture. They fall into the following classes:

1. **General-purpose data movement instructions.**
2. **Stack manipulation instructions.**
3. **Type-conversion instructions.**

1. General-purpose Data Movement Instructions

- **MOV (Move)** transfers a byte, word, or doubleword from the source operand to the destination operand. **The MOV instruction is useful for transferring data along any of these paths.** There are also variants of MOV that operate on segment registers.

  - To a register from memory
  - To memory from a register
  - Between general registers
  - Immediate data to a register
  - Immediate data to a memory

- **XCHG (Exchange)** swaps the contents of two operands.

- This instruction takes the place of **three MOV instructions.**

- XCHG is especially useful for implementing semaphores or similar data structures for process synchronization.

2. Stack Manipulation Instructions

- **PUSH (Push)** decrements the stack pointer (ESP), then transfers the source operand to the top of stack indicated by ESP.

- **PUSH is often used to place parameters on the stack before calling a procedure;** it is also the basic means of storing temporary variables on the stack.

- The PUSH instruction operates on memory operands, immediate operands, and register operands (including segment registers).

DIRECTION OF EXPANSION
BEFORE PUSH
31
0
ESP
AFTER PUSH
31
0
OPERAND
ESP

- **PUSHA (Push All Registers)** saves the contents of the eight general registers on the stack.

- This instruction simplifies procedure calls by reducing the number of instructions required to retain the contents of the general registers for use in a procedure.

- The processor pushes the general registers on the stack in the following order: **EAX, ECX, EDX, EBX, the initial value of ESP before EAX was pushed, EBP, ESI, and EDI**. PUSHA is complemented by the POPA instruction.

BEFORE PUSHA
AFTER PUSHA
31
0
31
0
DIRECTION OF EXPANSION
ESP
EAX
ECX
EDX
EBX
OLD ESP
EBP
ESI
EDI
ESP

- **POP (Pop)** transfers the word or doubleword at the current top of stack (indicated by ESP) to the destination operand, and then increments ESP to point to the new top of stack.

- POP moves information from the stack to a general register, or to memory There are also a variant of POP that operates

DIRECTION OF EXPANSION
BEFORE POP
31
0
OPERAND
ESP
AFTER POP
31
0
ESP

- **POPA (Pop All Registers)** restores the registers saved on the stack by PUSHA, except **that it ignores the saved value of ESP.**

BEFORE POPA
AFTER POPA
31
0
31
0
DIRECTION OF EXPANSION
EAX
ECX
EDX
EBX
ESP
EPB
ESI
EDI
ESP
ESP

3. Type Conversion Instructions

- The type conversion **instructions convert bytes into words, words into doublewords, and doublewords into 64-bit items (quad-words).**

- There are two classes of type conversion instructions:

1. The forms **CWD, CDQ, CBW, and CWDE** which operate only on data in the EAX register.

2. The forms **MOVSX and MOVZX**, which permit one operand to be in any general register while permitting the other operand to be in memory or in a register.

- **CWD (Convert Word to Doubleword)** and **CDQ (Convert Doubleword to Quad-Word)** double the size of the source operand. CWD extends the sign of the word in register AX throughout register DX.

- CDQ extends the sign of the doubleword in EAX throughout EDX.

- **CBW (Convert Byte to Word)** extends the sign of the byte in register AL throughout AX.

- **CWDE (Convert Word to Doubleword Extended)** extends the sign of the word in register AX throughout EAX.

15
7
0
BEFORE SIGN EXTENSION
S N N N N N N N N N N N N N N N
AFTER SIGN EXTENSION
31
23
15
7
0
S S S S S S S S S S S S S S S S N N N N N N N N N N N N N N N N

- **MOVSX (Move with Sign Extension)** sign-extends an 8-bit value to a 16-bit value and a 8- or 16-bit value to 32-bit value.

- **MOVZX (Move with Zero Extension)** extends an 8-bit value to a 16-bit value and an 8- or 16-bit value to 32-bit value by inserting high-order zeros.

Sign and Zero Extend Instructions

- **MOVZX (move with zero-extend)**
- **MOVSX (move with sign-extend)**
- **Both move the source into a destination of larger size (valid only for 386 and later processors)**
- **imm operands are not allowed**

```
mov bl, 07h
mov bh, 80h
movzx ax,bh     ;AX = 0080h
movsx dx,bh     ;DX = FF80h
movsx ax, bl    ;AX = 0007h
movzx ecx,dx    ;ECX = 0000FF80h
movsx ecx,dx    ;ECX = FFFFFF80h
movsx ecx,ax    ;ECX = 00000007h
```

69

B. Binary Arithmetic Instructions

- Addition and Subtraction Instructions
- Comparison and Sign Change Instruction
- Multiplication Instructions
- Division Instructions

Addition and Subtraction Instructions

- **ADD (Add Integers)** replaces the destination operand with the sum of the source and destination operands. Sets CF if overflow.

- **ADC (Add Integers with Carry)** sums the operands, adds one if CF is set, and replaces the destination operand with the result. If CF is cleared, ADC performs the same operation as the ADD instruction.

- **INC (Increment)** adds one to the destination operand. INC does not affect CF. Use ADD with an immediate value of 1 if an increment that updates carry (CF) is needed.

- **SUB (Subtract Integers)** subtracts the source operand from the destination operand and replaces the destination operand with the result. If a borrow is required, the CF is set. The operands may be signed or unsigned bytes, words, or doublewords.

- **SBB (Subtract Integers with Borrow)** subtracts the source operand from the destination operand, subtracts 1 if CF is set, and returns the result to the destination operand. If CF is cleared, SBB performs the same operation as SUB.

- **DEC (Decrement)** subtracts 1 from the destination operand. DEC does not update CF. Use SUB with an immediate value of 1 to perform a decrement that affects carry.

2. Comparison and Sign Change Instruction

- **CMP (Compare)** subtracts the source operand from the destination operand. It updates OF, SF, ZF, AF, PF, and CF but does not alter the source and destination operands. A subsequent Jcc instruction can test the appropriate flags.

- **NEG (Negate)** subtracts a signed integer operand from zero. The effect of NEG is to reverse the sign of the operand from positive to negative or from negative to positive.

3. Multiplication Instructions

- **MUL (Unsigned Integer Multiply)** performs an unsigned multiplication of the source operand and the accumulator.
  - **If the source is a byte**, the processor multiplies it by the contents of AL and returns the double-length **result to AH and AL.**
  - **If the source operand is a word**, the processor multiplies it by the contents of AX and returns the double-length **result to DX and AX.**
  - **If the source operand is a doubleword**, the processor multiplies it by the contents of EAX and returns the 64-bit **result in EDX and EAX.**
  - **MUL sets CF and OF** when the upper half of the result is nonzero; otherwise, they are cleared.

- **IMUL (Signed Integer Multiply)** performs a signed multiplication operation.

- IMUL has three variations:
  1. **A one-operand form**. The operand may be a byte, word, or doubleword located in memory or in a general register. This instruction uses EAX and EDX as implicit operands in the same way as the MUL instruction.

  2. **A two-operand form**. One of the source operands may be in any general register while the other may be either in memory or in a general register. The product replaces the general-register operand.

  3. **A three-operand form**; two are source and one is the destination operand. One of the source operands is an immediate value stored in the instruction; the second may be in memory or in any general register. The product may be stored in any general register.

4. Division Instruction

- **DIV (Unsigned Integer Divide)** performs an unsigned division of the accumulator by the source operand. The Dividend (the accumulator) is twice the size of the Divisor (the source operand); the quotient and remainder have the same size as the divisor, as the following table shows.

Size of Source Operand (divisor)	Dividend	Quotient	Remainder
Byte	AX	AL	AH
Word	DX:AX	AX	DX
Doubleword	EDX:EAX	EAX	EDX

- **IDIV (Signed Integer Divide)** performs a signed division of the accumulator by the source operand. IDIV uses the same registers as the DIV instruction.

C. Decimal Arithmetic Instructions

- **Packed BCD Adjustment Instructions**
- **Unpacked BCD Adjustment Instructions**

1. Packed BCD Adjustment Instructions

- **DAA (Decimal Adjust after Addition)** adjusts the result of adding two valid packed decimal operands in AL. DAA must always follow the addition of two pairs of packed decimal numbers (one digit in each half-byte) to obtain a pair of valid packed decimal digits as results. The carry flag is set if carry was needed.

- **DAS (Decimal Adjust after Subtraction)** adjusts the result of subtracting two valid packed decimal operands in AL. DAS must always follow the subtraction of one pair of packed decimal numbers (one digit in each halfbyte) from another to obtain a pair of valid packed decimal digits as results. The carry flag is set if a borrow was needed.

DAA – Decimal Adjust After Addition

Used after ADD or ADC instruction to adjust the result back into packed BCD format.

```
MOV   AL,24H        (24H is BCD no.)
ADD   AL,28H        - Result = 4CH
DAA                 - Converts it into 52H
```

DAS – Decimal Adjust After Subtraction

Used after SUB or SBB instruction to adjust result back to BCD (packed format).

```
MOV   AL, 91H
SUB   AL, 56H       ; Result = 3BH
DAS                 ; Converts into 35H
```

2. Unpacked BCD Adjustment Instructions

- **AAA (ASCII Adjust after Addition)** changes the contents of register AL to a valid unpacked decimal number, and zeros the top 4 bits. AAA must always follow the addition of two unpacked decimal operands in AL. The carry flag is set and AH is incremented if a carry is necessary.

- **AAS (ASCII Adjust after Subtraction)** changes the contents of register AL to a valid unpacked decimal number, and zeros the top 4 bits. AAS must always follow the subtraction of one unpacked decimal operand from another in AL. The carry flag is set and AH decremented if a borrow is necessary.

ASCII Adjustment Instruction

ASCII coded numbers - 30H for 0 to 39H for 9.
Instructions use Register AX for source and destination.

AAA – ASCII Adjustment After Addition

Used after ADD or ADC to adjust result back in (AX)

```
MOV   AX, 39H          ; ASCII code for 9
ADD   AL, 35H          ; ASCII code for 5
                       ; Result = 6EH
AAA     - Adjust result as 0104H in AX
ADD   AX, 3030H        ; Result in ASCII
; 3134H is
AH – 31H               ; ASCII code for 1
AL – 34H               ; ASCII code for 4
```

Example :

d1 contains 34h , d2 contains 38h of byte type

	Ax	AF
Mov AL, d1 ;	0034	
Sub AL, d2;	00fc	1
AAS ;	ff06	1

- since the rightmost digit of AL is c , subtract 6 from AL
- Subtract 1 from ah, set AF and CF flags
- The answer is -4, is ff06 in 10's complement

11

- **AAM (ASCII Adjust after Multiplication)** corrects the result of a multiplication of two valid unpacked decimal numbers. AAM must always follow the multiplication of two decimal numbers to produce a valid decimal result. The high order digit is left in AH, the low order digit in AL.

- **AAD (ASCII Adjust before Division)** modifies the numerator in AH and AL to prepare for the division of two valid unpacked decimal operands so that the quotient produced by the division will be a valid unpacked decimal number. AH should contain the high-order digit and AL the low-order digit. This instruction adjusts the value and places the result in AL. AH will contain zero.

Example:

AL contains 35H and CL contains 39H

Instruction	comment	AX	CL
and CL, 0Fh;	Convert CL to 09	0035	39
and AL,0Fh;	Convert AL to 05	0005	09
mul CL;	Multiply AL by CL	002D	
AAM ;	Convert to unpacked BCD	0405	
Or AX,3030h;	covert to ASCII	3435	

14

Example:

AX contains 3238 (28) - Dividend

CL contains 37 (07) – Divisor

Instruction	comment	AX	CL
and CL,0Fh;	convert to unpacked BCD	3238	07
and AX,0F0Fh;	convert to unpacked BCD	0208	07
AAD;	convert to binary	001C	
div CL;	divide by 7	0004	

18

D. Logical Instructions

- The group of logical instructions includes:

  - The Boolean operation instructions
  - Bit test and modify instructions
  - Bit scan instructions
  - Rotate and shift instructions
  - TEST instructions

1. Boolean Operation Instructions

- **NOT (Not)** inverts the bits in the specified operand to form a **one‘s complement of the operand.**

- The NOT instruction is a unary operation that **uses a single operand** in a register or memory.

- NOT has **no effect on the flags.**

- The **AND, OR, and XOR** instructions perform the standard logical operations "and", "(inclusive) or", and "exclusive or". These instructions can use the following combinations of operands:

  - Two register operands.
  - A general register operand with a memory operand.
  - An immediate operand with either a general register operand or a memory operand.
  - **AND, OR, and XOR clear OF and CF, leave AF undefined, and update SF, ZF, and PF.**

2. Bit Test and Modify Instructions

Instruction	Effect on CF	Effect on Selected Bit
Bit (Bit Test)	CF ← BIT	(none)
BTS (Bit Test and Set)	CF ← BIT	BIT ← 1
BTR (Bit Test and Reset)	CF ← BIT	BIT ← 0
BTC (Bit Test and Complement)	CF ← BIT	BIT ← NOT(BIT)

3. Bit Scan Instructions

- **These instructions scan a word or doubleword for a one-bit and store the index of the first set bit into a register.**

- The bit string being scanned may be either in a register or in memory.

- **The ZF flag is set if the entire word is zero** (no set bits are found); **ZF is cleared if a one-bit is found**. If no set bit is found, the value of the destination register is undefined.

- **BSF (Bit Scan Forward)** scans from low-order to high-order (starting from bit index zero).

- **BSR (Bit Scan Reverse)** scans from high-order to low-order (starting from bit index 15 of a word or index 31 of a doubleword).

4. Shift and Rotate Instructions

Logical Shift

Fills the newly created bit with zero.

Arithmetic Shift

Fills the newly created bit with a copy of the number's sign bit.

How it works?

Instruction	Operand on which action will be performed	Number of times the operation will be performed
SAL or SHL or SAR or SHR	• A value in a register (Value can be shifted either right or left; can be either arithmetic right or logic right.) • A byte (8-bit) in memory • A word (16-bit) in memory • A double-word (32-bit) in memory **A shift can be left or right, arithmetic or logical, in memory or in register.**	• Blank (once) • The value in the CL register (E.g., If a CL register is loaded with value 4, it will shift either right or left, arithmetic or logical by 4.) • A defined number

Rotate Instructions

Rotate without Carry

The bits are rotated right or left depending upon whether ROR or ROL instruction is used.

The bit rotated out gets copied into carry as well as into the extreme bit.

Rotate through Carry

The bits are rotated right or left through carry depending upon whether RCR or RCL instruction is used. The diagram below shows RCR.

TEST Instruction

- **TEST (Test) performs the logical "and"** of the two operands, clears OF and CF, leaves AF undefined, and **updates SF, ZF, and PF**.

- The flags can be tested by conditional control transfer instructions. The operands may be doublewords, words, or bytes.

- The difference between TEST and AND is that **TEST does not alter the destination operand.**

- TEST differs from BT in that TEST is useful **for testing the value of multiple bits** in one operations, whereas BT tests a single bit.

Control Transfer Instructions

- **Unconditional Transfer Instructions**
- **Conditional Transfer Instructions**
- **Software-Generated Interrupts**

1.Unconditional Transfer Instructions

- **JMP (Jump)** unconditionally transfers control to the target location. JMP is a one-way transfer of execution; it does not save a return address on the stack.

- **CALL (Call Procedure)** activates an out-of-line procedure, saving on the stack the address of the instruction following the CALL for later use by a RET (Return) instruction.
- **CALL places the current value of EIP on the stack**. The RET instruction in the called procedure uses this address to transfer control back to the calling program.

- **RET (Return From Procedure)** terminates the execution of a procedure and transfers control through a back-link on the stack to the program that originally invoked the procedure.

- RET restores the value of EIP that was saved on the stack by the previous CALL instruction.

- **IRET (Return From Interrupt)** returns control to an interrupted procedure.

- IRET differs from RET in that it also pops the flags from the stack into the flags register. The flags are stored on the stack by the interrupt mechanism.

2. Conditional Transfer Instructions

Unsigned Conditional Transfers

Mnemonic	Condition Tested	"Jump If..."
JA/JNBE	(CF or ZF) = 0	above/not below nor equal
JAE/JNB	CF = 0	above or equal/not below
JB/JNAE	CF = 1	below/not above nor equal
JBE/JNA	(CF or ZF) = 1	below or equal/not above
JC	CF = 1	carry
JE/JZ	ZF = 1	equal/zero
JNC	CF = 0	not carry
JNE/JNZ	ZF = 0	not equal/not zero
JNP/JPO	PF = 0	not parity/parity odd
JP/JPE	PF = 1	parity/parity even

Signed Conditional Transfers

Mnemonic	Condition Tested	"Jump If..."
JG/JNLE	((SF xor OF) or ZF) = 0	greater/not less nor equal
JGE/JNL	(SF xor OF) = 0	greater or equal/not less
JL/JNGE	(SF xor OF) = 1	less/not greater nor equal
JLE/JNG	((SF xor OF) or ZF) = 1	less or equal/not greater
JNO	OF = 0	not overflow
JNS	SF = 0	not sign (positive, including 0)
JO	OF = 1	overflow
JS	SF = 1	sign (negative)

- **LOOP (Loop While ECX Not Zero)** is a conditional transfer that automatically decrements the ECX register before testing ECX for the branch condition.

- If ECX is non-zero, the program branches to the target label specified in the instruction.

- The LOOP instruction causes the repetition of a code section, until the operation of the LOOP instruction decrements ECX to a value of zero.

- **LOOPE (Loop While Equal) and LOOPZ (Loop While Zero)** are synonyms for the same instruction.
- These instructions automatically decrement the ECX register before testing ECX and ZF for the branch conditions.
- **If ECX is non-zero and ZF=1, the program branches to the target** label specified in the instruction.
- If LOOPE or LOOPZ finds that ECX=0 or ZF=0, control transfers to the instruction immediately following the LOOPE or LOOPZ instruction.

- LOOPE/LOOPZ Label
  - Decrements CX by 1 (without modifying the flag).
  - Branches to label if zero flag is set and CX ≠ 0.

```
   MOV   CX, -
KK :
   CMP   AL, DL
   LOOPE   KK        ; LOOP will execute till AL = DL and CX ≠ 0.
```

Do – while Loop

i.e. (CX) ← (CX) – 1

If [(CX) ≠ 0 and (ZF) = 1]

then (IP) ← (IP) + disp 8

- **LOOPNE (Loop While Not Equal) and LOOPNZ (Loop While Not Zero)** are synonyms for the same instruction.
- These instructions automatically decrement the ECX register before testing ECX and ZF for the branch conditions.
- **If ECX is non-zero and ZF=0, the program branches to the target** label specified in the instruction.
- If LOOPNE or LOOPNZ finds that ECX=0 or ZF=1, control transfers to the instruction immediately following the LOOPNE or LOOPNZ instruction.

3. Software-Generated Interrupts

- **INT n (Software Interrupt)** activates the interrupt service routine that corresponds to the number coded within the instruction.

- The interrupt service routine terminates with an IRET instruction that returns control to the instruction that follows INT.

- **INTO (Interrupt on Overflow)** invokes interrupt 4 if OF is set. Interrupt 4 is reserved for this purpose. OF is set by several arithmetic, logical, and string instructions.

- **BOUND (Detect Value Out of Range)** verifies that the signed value contained in the specified register lies within specified limits.

- An interrupt (INT 5) occurs if the value contained in the register is less than the lower bound or greater than the upper bound.

String and Character Translation Instructions

MOVSB DST, SRC

MOVSW DST, SRC

MOVS Instructions

- MOVS – **move** string from one memory location to other

Assembly Language	*Operation*
MOVSB	ES:[DI] = DS:[SI]; DI = DI ± 1; SI = SI ± 1 (byte transferred)
MOVSW	ES:[DI] = DS:[SI]; DI = DI ± 2; SI = SI ± 2 (word transferred)
MOVSD	ES:[DI] = DS:[SI]; DI = DI ± 4; SI = SI ± 4 (doubleword transferred)
MOVS BYTE1,BYTE2	ES:[DI] = DS:[SI]; DI = DI ± 1; SI = SI ± 1 (if BYTE1 and BYTE2 are bytes)
MOVS WORD1,WORD2	ES:[DI] = DS:[SI]; DI = DI ± 2, SI = SI ± 2 (if WORD1 and WORD2 are words)
MOVS DWORD1, DWORD2	ES:[DI] = DS:[SI]; DI = DI ± 4; SI = SI ± 4 (if DWORD1 and DWORD2 are doublewords)

Table 4-13 : From Brey's Book

LOD SB
Loads a byte from a String in memory into AL.
Automatically increments/decrements SI by 1.

LOD SW
Loads a word from a String in memory into AX.
Automatically increments/decrements SI by 2.

Moving from String to Accumulator

STO SB
Stores a byte from AL into a String location in memory.
Automatically increments/decrements DI by 1.

STO SW
Stores a word from AX into a String location in memory.
Automatically increments/decrements DI by 2.

Moving from Accumulator to String

CMPSB or CMPSW – compares either Byte or Word Strings

- The CX Register holds length of STRINGs to be compared.
- STRING1 is pointed to by [DS:SI], STRING2 by [ES:DI].
- If STRING1 = STRING2; then Zero Flag is Set.

SCASB or SCASW – scans either Byte or Word Strings

- The CX Register holds length of STRING to be scanned.
- STRING is pointed to by [DS:SI].
- If the STRING contains value, Zero Flag is Set.

REP Instruction

- **These instructions are used along with string instructions only.**

	Condition 1	Condition 2	Instructions
REPE/REPZ	CX != ZERO	ZF = ONE	CMPSB, CMPSW, SCASB, SCASW
REPNE/REPNZ	CX != ZERO	ZF = ZERO	
REP	CX != ZERO	ZF = don't care	MOVSB, MOVSW, LODSB, LODSW, STOSB, STOSW

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

Instructions for Block-Structured Languages

- **ENTER**

- **LEAVE**

- **ENTER**: ENTER (Enter Procedure) creates a stack frame that may be used to implement the scope rules of block-structured high-level languages.

- The ENTER instruction includes two parameters.
  - The first parameter specifies the **number of bytes of dynamic storage** to be allocated on the stack for the routine being entered.
  - The second parameter corresponds to the **lexical nesting level (0-31)** of the routine.

- Example: **ENTER 2048,3**
  - Allocates 2048 bytes of dynamic storage on the stack and sets up pointers to two previous stack frames in the stack frame that ENTER creates for this procedure.

- **LEAVE:** LEAVE (Leave Procedure) reverses the action of the previous ENTER instruction.

- The LEAVE instruction does not include any operands.

- LEAVE copies EBP to ESP to release all stack space allocated to the procedure by the most recent ENTER instruction.

Flag Control Instructions

- The Flag manipulation instructions directly modify some of the Flags of 8086.
-

i. CLC – Clear Carry Flag.
ii. CMC – Complement Carry Flag.
iii. STC – Set Carry Flag.
iv. CLD – Clear Direction Flag.
v. STD – Set Direction Flag.
vi. CLI – Clear Interrupt Flag.
vii. STI – Set Interrupt Flag.

Flag Transfer Instructions

- **LAHF (Load AH from Flags)** copies SF, ZF, AF, PF, and CF to AH bits 7, 6, 4, 2, and 0, respectively. The contents of the remaining bits (5, 3, and 1) are undefined. The flags remain unaffected.

- **SAHF (Store AH into Flags)** transfers bits 7, 6, 4, 2, and 0 from AH into SF, ZF, AF, PF, and CF, respectively (see Figure 3-22).

7	6	5	4	3	2	1	0
SF	ZF	UU	AF	UU	PF	UU	CF

LAHF LOADS FIVE FLAGS FROM THE FLAG REGISTER INTO REGISTER AH. SAHF STORES THESE SAME FIVE FLAGS FROM AH INTO THE FLAG REGISTER. THE BIT POSITION OF EACH FLAG IS THE SAME IN AH AS IT IS IN THE FLAG REGISTER. THE REMAINING BITS (MARKED UU) ARE RESERVED; DO NOT DEFINE.

- The **PUSHF** and **POPF** instructions are not only useful for storing the flags in memory where they can be examined and modified but are also useful for preserving the state of the flags register while executing a procedure.

- **PUSHF (Push Flags)** decrements ESP by two and then transfers the low-order word of the flags register to the word at the top of stack pointed to by ESP.
- The variant PUSHFD decrements ESP by four, then transfers both words of the extended flags register to the top of the stack pointed to by ESP (the VM and RF flags are not moved, however).

- **POPF (Pop Flags)** transfers specific bits from the word at the top of stack into the low-order byte of the flag register, then increments ESP by two.
- The variant POPFD transfers specific bits from the doubleword at the top of the stack into the extended flags register (the RF and VM flags are not changed, however), then increments ESP by four.

BITS MARKED 0 AND 1 ARE RESERVED BY INTEL. DO NOT DEFINE.

SYSTEMS FLAGS (INCLUDING THE IOPL FIELD, AND THE VM, RF, AND IF FLAGS) ARE PUSHED AND ARE VISIBLE TO APPLICATIONS PROGRAMS. HOWEVER, WHEN AN APPLICATIONS PROGRAM POPS THE FLAGS, THESE ITEMS ARE NOT CHANGED, REGARDLESS OF THE VALUES POPPED INTO THEM.

Coprocessor Interface Instructions

- **ESC (Escape)** is a 5-bit sequence that begins the opcodes that identify floating point numeric instructions.

- The ESC pattern tells the 80386 to send the opcode and addresses of operands to the numeric coprocessor.

- **WAIT (Wait)** is an 80386 instruction that suspends program execution until the 80386 CPU detects that the BUSY pin is inactive.

- BUSY condition indicates that the coprocessor has completed its processing task and that the CPU may obtain the results.

Segment Register Instructions

1. Segment-register transfer instructions.

MOV SegReg, ...

MOV ..., SegReg

PUSH SegReg

POP SegReg

2. Control transfers to another executable segment.

JMP far ; direct and indirect

CALL far

RET far

3. Data pointer instructions.

LDS

LES

LFS

LGS

LSS

- **LDS (Load Pointer Using DS)** transfers a pointer variable from the source operand to DS and the destination register.

- The source operand must be a memory operand, and the destination operand must be a general register.

- DS receives the segment-selector of the pointer. The destination register receives the offset part of the pointer, which points to a specific location within the segment.

- Example: **LDS ESI, STRING_X**

- Loads DS with the selector identifying the segment pointed to by a STRING_X, and loads the offset of STRING_X into ESI.
- Specifying ESI as the destination operand is a convenient way to prepare for a string operation on a source string that is not in the current data segment.

- **LES (Load Pointer Using ES)** operates identically to LDS except that ES receives the segment selector rather than DS.

- Example: **LES EDI, DESTINATION_X**

- Loads ES with the selector identifying the segment pointed to by DESTINATION_X, and loads the offset of DESTINATION_X into EDI.
- This instruction provides a convenient way to select a destination for a string operation if the desired location is not in the current extra segment.

- **LFS (Load Pointer Using FS)** operates identically to LDS except that FS receives the segment selector rather than DS.

- **LGS (Load Pointer Using GS)** operates identically to LDS except that GS receives the segment selector rather than DS.

- **LSS (Load Pointer Using SS)** operates identically to LDS except that SS receives the segment selector rather than DS.

Miscellaneous Instructions

- **Address Calculation Instruction**
- **No-Operation Instruction**
- **Translate Instruction**

Address Calculation Instruction

- **LEA (Load Effective Address)** transfers the offset of the source operand (rather than its value) to the destination operand. The source operand must be a memory operand, and the destination operand must be a general register. This instruction is especially useful for initializing registers before the execution of the string primitives (ESI, EDI) or the XLAT instruction (EBX). The LEA can perform any indexing or scaling that may be needed.

- Example: **LEA EBX, EBCDIC_TABLE**

- Causes the processor to place the address of the starting location of the table labeled EBCDIC_TABLE into EBX.

No-Operation Instruction

- **NOP (No Operation)** occupies a byte of storage but affects nothing but the instruction pointer, EIP.

Translate Instruction

- **XLAT (Translate)** replaced a byte in the AL register with a byte from a user-coded translation table.

- When XLAT is executed, AL should have the unsigned index to the table addressed by EBX. XLAT changes the contents of AL from table index to table entry. EBX is unchanged.
- The XLAT instruction is useful for translating from one coding system to another such as from ASCII to EBCDIC. The translate table may be up to 256 bytes long. The value placed in the AL register serves as an index to the location of the corresponding translation value.

XLAT – Translate

Replace the byte in AL with byte from users table, addressed by BX. Original value of AL is index into translate table.

```
MOV   BX, OFFSET TABLE
MOV   AL, 00H
XLAT
```

(AL) ← 5

AL← [BX+AL]

ASCII to Hex Table

Dec	Hx	Oct	Char		Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL	(null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH	(start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX	(start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX	(end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT	(end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ	(enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK	(acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL	(bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS	(backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB	(horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF	(NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT	(vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF	(NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR	(carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO	(shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI	(shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE	(data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1	(device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2	(device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3	(device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4	(device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK	(negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN	(synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB	(end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN	(cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM	(end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB	(substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC	(escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS	(file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS	(group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS	(record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US	(unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL