

# Unit 1

Microprocessor Architecture

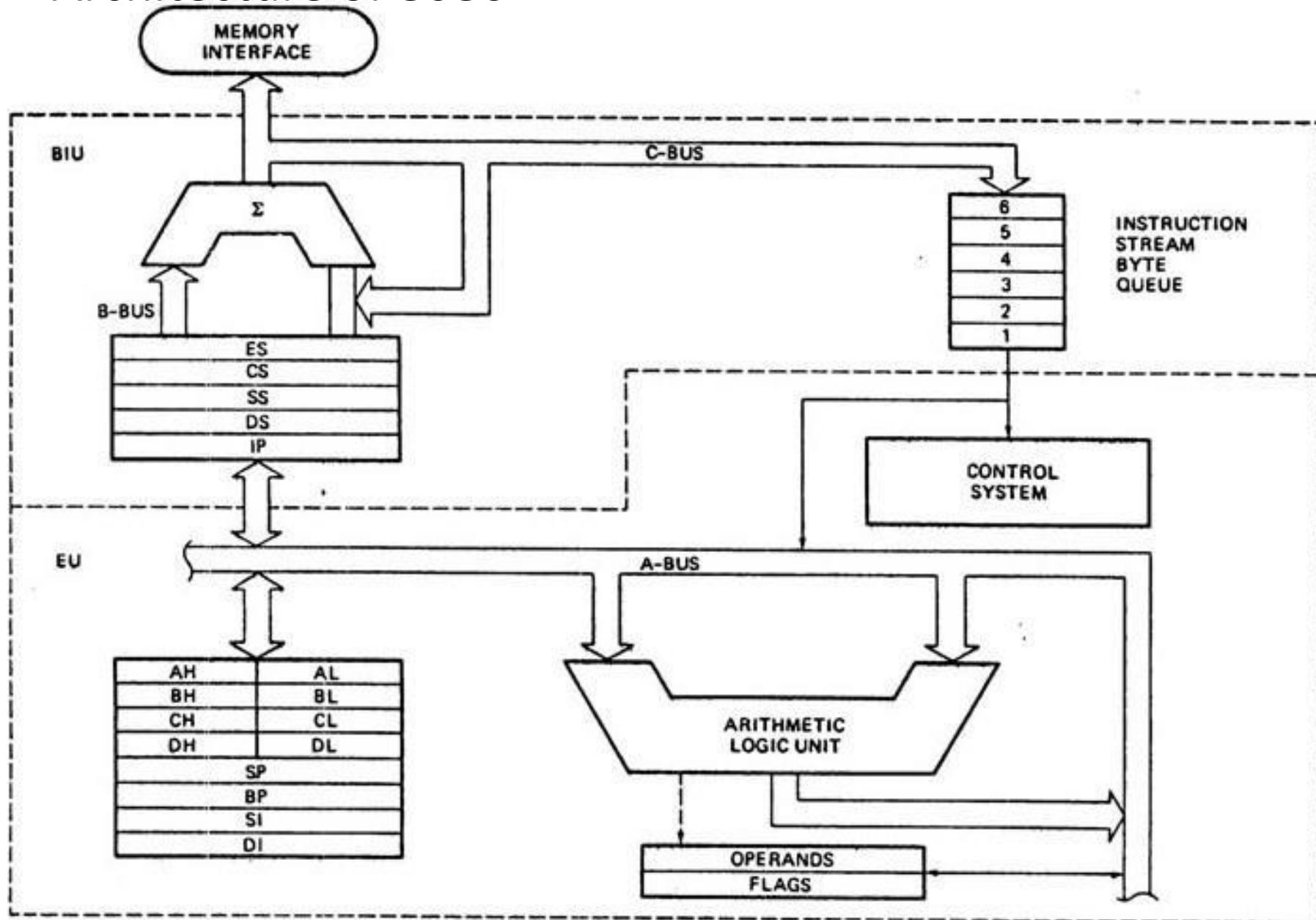
# Features of 8086:

- It requires +5v power supply.
- It is 16 bit MP
- It is available in 3 versions based on the frequency of operation –
  - 8086 → 5MHz
  - 8086-2 → 8MHz
  - 8086-1 → 10 MHz
- It has 20-bit address bus, so it can directly access  $2^{20} = 1\text{MB}$  (10,48,576) memory locations.
- It can generate 16 bit I/O address , hence it can access  $2^{16} = 65536$  I/O ports
- Operating frequency of upto 5MHz.

# Features of 8086:

- It has multiplexed address and data bus.(A<sub>D0</sub> to A<sub>D15</sub>)
- Performs bit, byte and word operations.
- Powerful instruction set with different addressing modes.
- It supports a 16 bit ALU , a set of 16 bit registers,
- Provides segmented memory addressing capability
- It has rich instruction set, powerful interrupt structure,
- It has 6 byte Instruction queue.

# Architecture of 8086



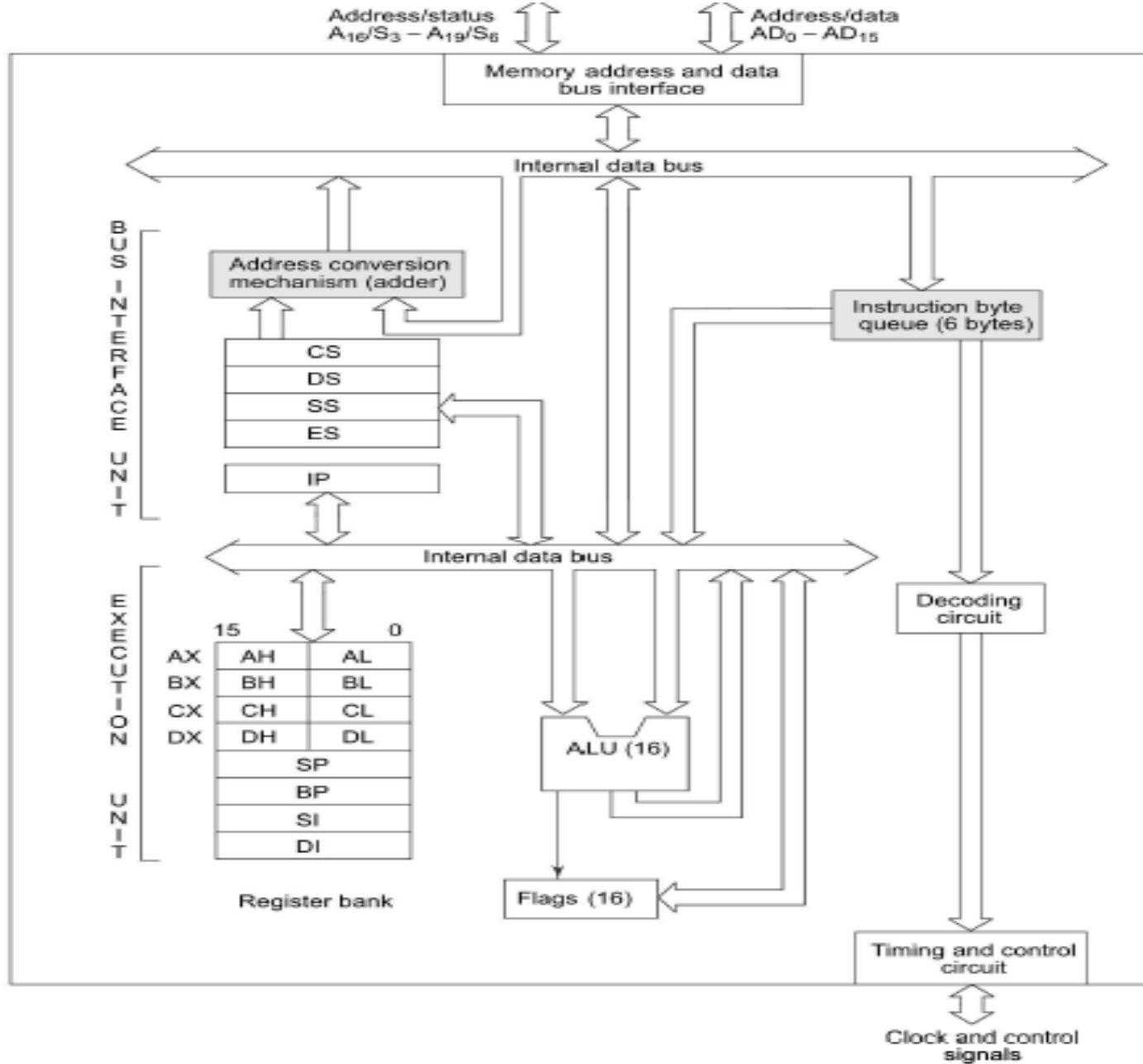


Fig. 8086 Architecture

# Register organization of 8086

## – General purpose

- Holding data , variables and intermediate results
- Counters or for storing offset address

## – Special purpose

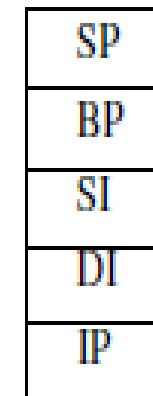
- Used as segment registers, pointers, index registers or as offset storage for particular addressing mode.

General data registers:

AX	AH	AL
BX	BH	BL
CX	CH	CL
DX	DH	DL



FLAGS/PSW



General data registers

Segment registers

Pointers and index registers

Fig. Register organization of 8086

# 8086 Architecture

- Segment Registers:
  - CS
    - is used for addressing a memory location in the code segment of the memory, where executable program is stored.
  - DS:
    - Points to the data segment of the memory, where data is resided.
  - ES:
    - Essentially another data segment of the memory
  - SS:
    - Memory used to store stack data

# Pointers and Index registers:

- pointers contain offset within particular segment.
  - IP,BP and SP contain offset within the CS
  - BP and SP within the SS.
  - **BP** – This is the base pointer. It is of 16 bits.
  - It is primarily used in accessing parameters passed by the stack.
  - It's offset address relative to stack segment.

- Index registers:
  - Are used as general purpose registers as well as offset storage in case of indexed, base indexed and relative based indexed addressing modes.
  - SI- is used to store the offset of source data in data segment
  - DI- is used to store the offset of destination in data or extra segment

# 8086 Architecture

- 2 functional units:
  - BIU(Bus Interface Unit)
  - EU(Execution Unit)
- **BIU(Bus Interface Unit)**
  - It makes systems bus signals available for external interfacing of the devices.
  - Is responsible for establishing communication with external devices and peripherals including memory.
  - BIU contains the circuit for physical address calculations and predecoding instruction byte queue(6 byte)

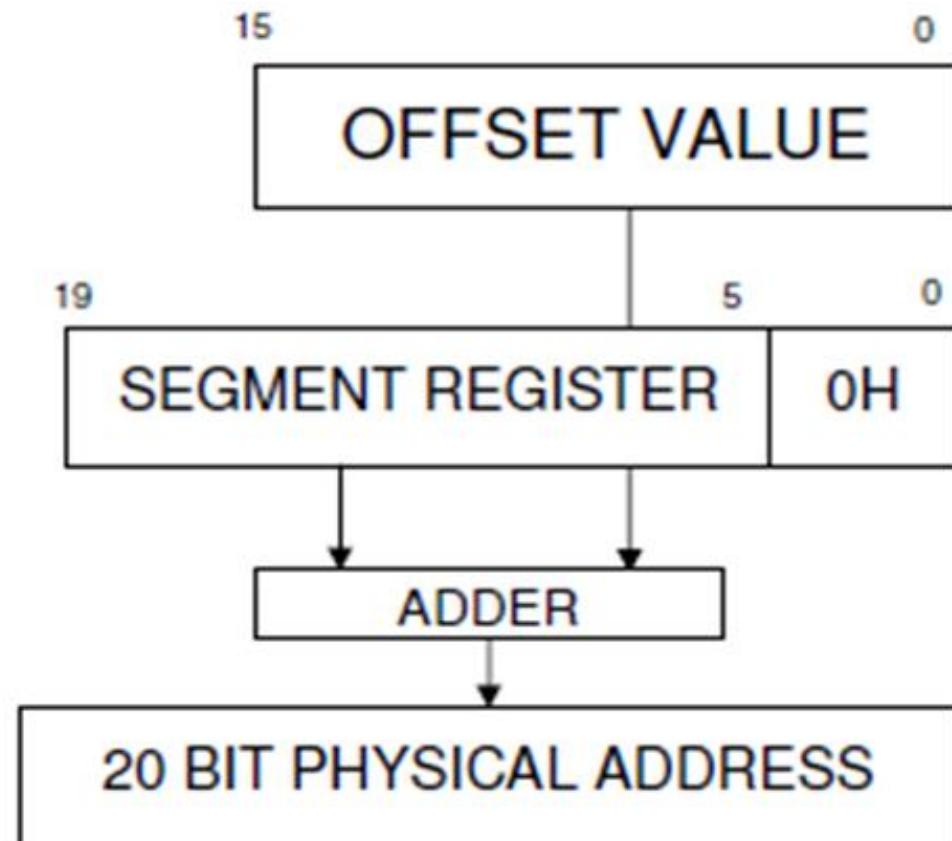
# 8086 Architecture

- It looks after the operation of address and data bus. Its functions are-
  - To generate address for memory and I/O ports
  - To fetch instructions from memory
  - To read data from memory and I/O ports
  - It supports instruction queuing
  - It writes data into port /memory.

# 8086 Architecture

- How to generate physical address:
  - Content of segment register (seg. add) is shifted left bitwise four times and offset register (offset add) is added to this result, which produces 20 bit physical address.

Fig. Physical address formation



# 8086 Architecture

e.g. seg address=

1005H

offset address=

5555H

segment address=1005= 0001 0000 0000 0101

Shifted by 4 bit

0001 0000 0000 0101 0000

+

Offset address

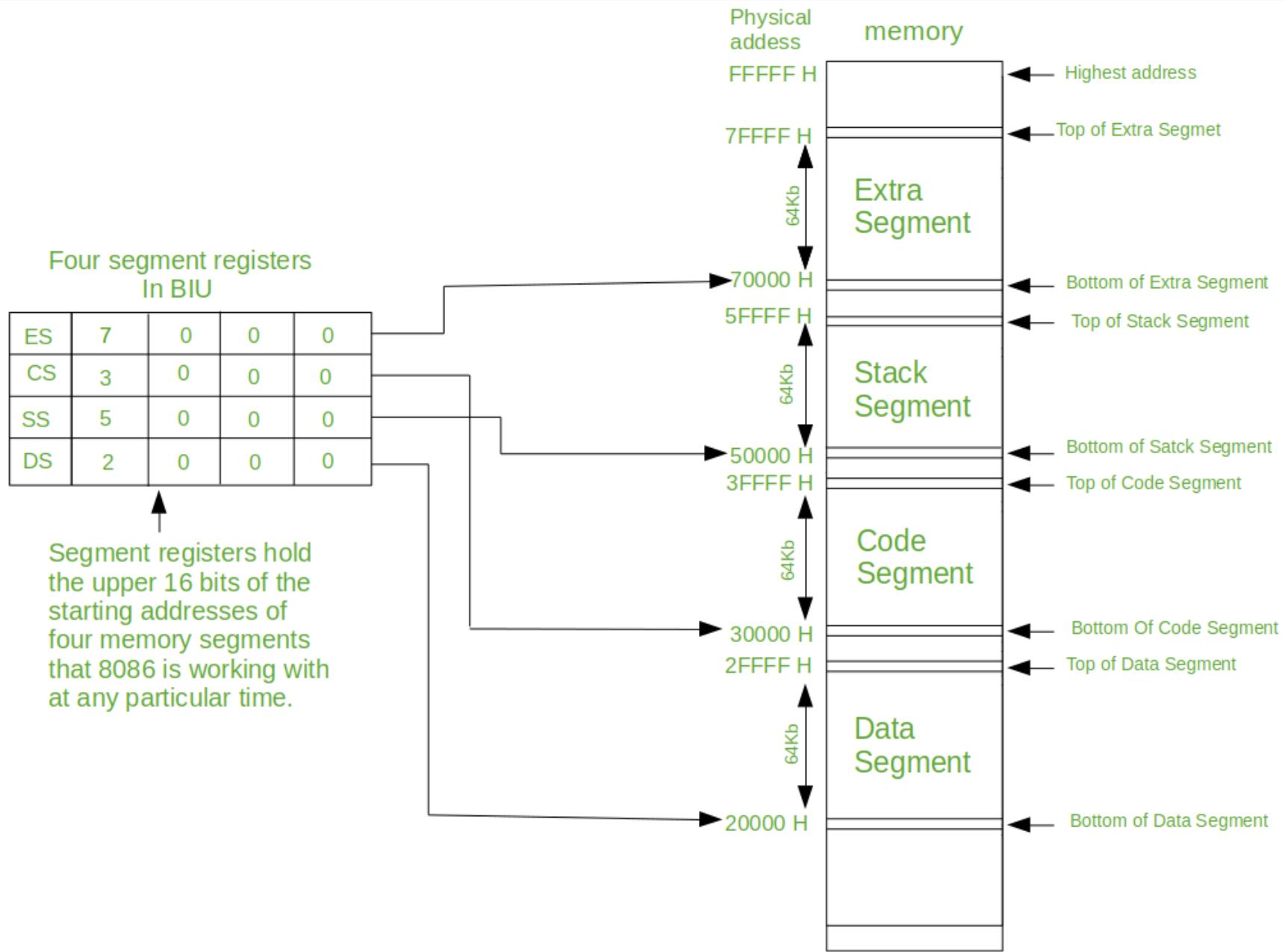
0101 0101 0101 0101

-----

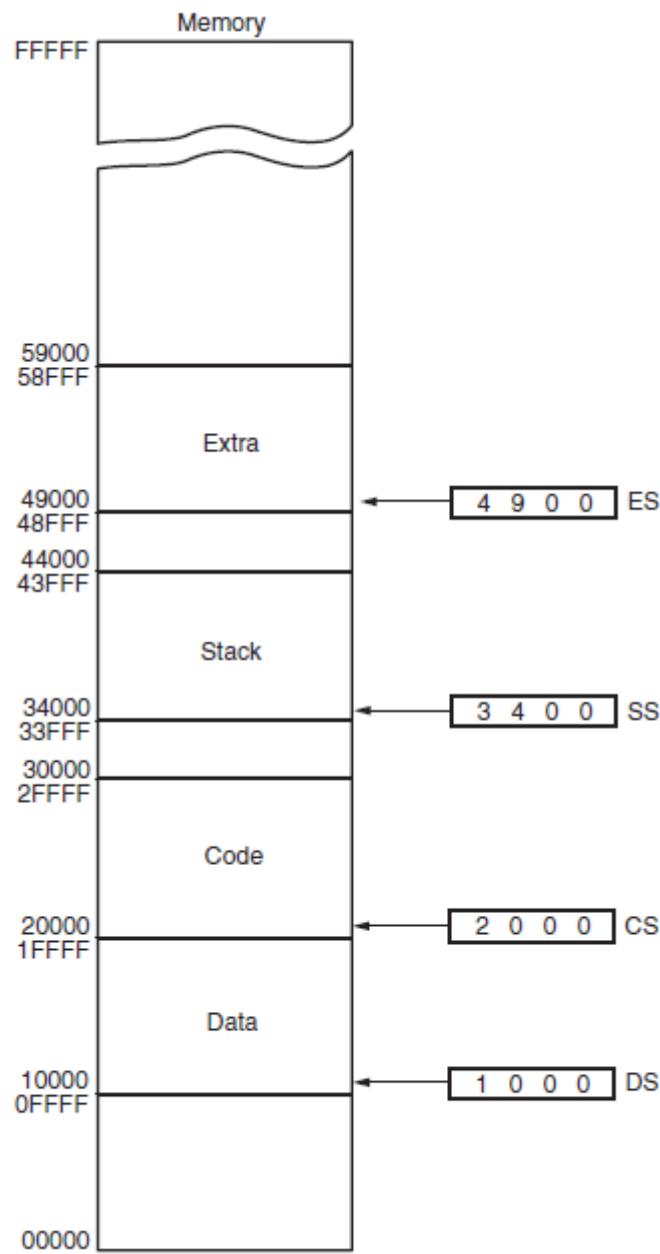
0001 0101 0101 1010 0101

=

1      5      5      A      5



**FIGURE** A memory system showing the placement of four memory segments.



- Maximum 64K locations may be accommodated in the segment.
- **Segment register** indicates the base address of a particular segment
- **offset indicates** the distance of the required memory location in the segment from the base address.
- BIU has summer (adder) to perform this procedure for obtaining physical address while addressing memory.

# 8086 Architecture

- Execution Unit:
  - Decodes and executes the instructions
- EU does:
  - To load segment registers
  - To update IP while branching a prog.
  - Execution unit gives instructions to BIU stating from where to fetch the instruction and then decode and execute those instructions
  - Instructs BIU for reading or writing data from or to a memory location or an I/O ports.

# 8086 Architecture

- **Execution unit:**
  - Contains register set of 8086 except segment registers and IP.
  - It has 16 bit ALU
  - 16 bit flag register
  - Decoding unit decodes opcode bytes received from instruction byte queue
  - Timing and control unit derives the necessary control signal to execute instruction opcode received from decoding circuit.
  - Pass this result to BIU for storing in memory.

# 8086 Architecture

- Instruction Queue:
  - It is FIFO memory

# 8086 Architecture

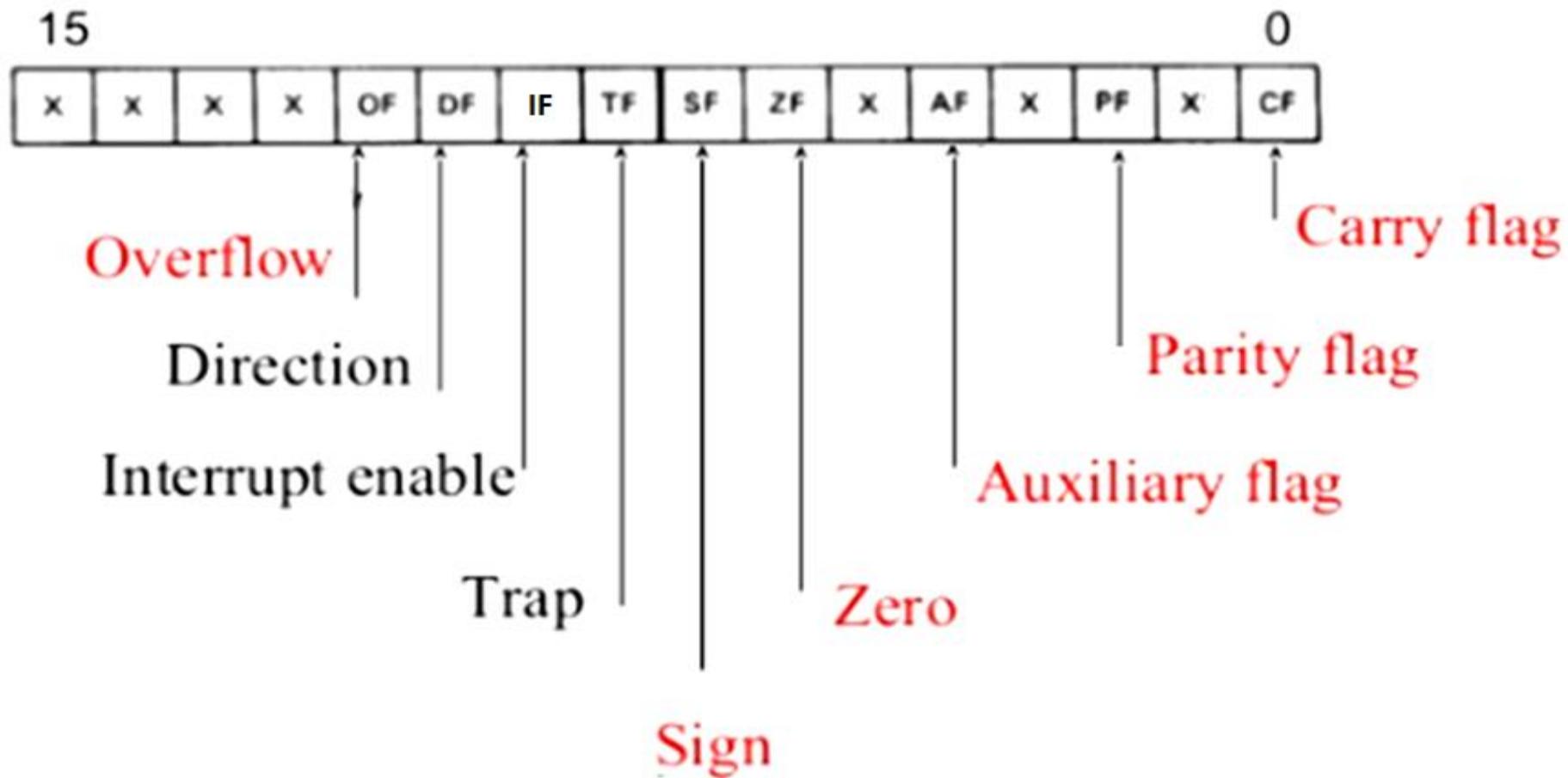


Fig 8086 Flag Register

- Conditional (status) flags=6
- Control flags=3

- TF –Trap Flag:
  - If T=1– processor enters the single step execution mode.
  - i.e. trap interrupt is generated after execution of each instruction.
- IF – Interrupt flag:
  - If this flag is set, maskable interrupts are recognized by the CPU, otherwise they are ignored.
- DF- Direction flag: used by string manipulation instructions.
  - if DF=0 –the string is processed beginning from the lowest address to the highest address i.e. auto incrementing mode.
  - if DF=1 –the string is processed from highest address towards the lowest address i.e. auto decrementing mode.

# 8086 Addressing Modes:

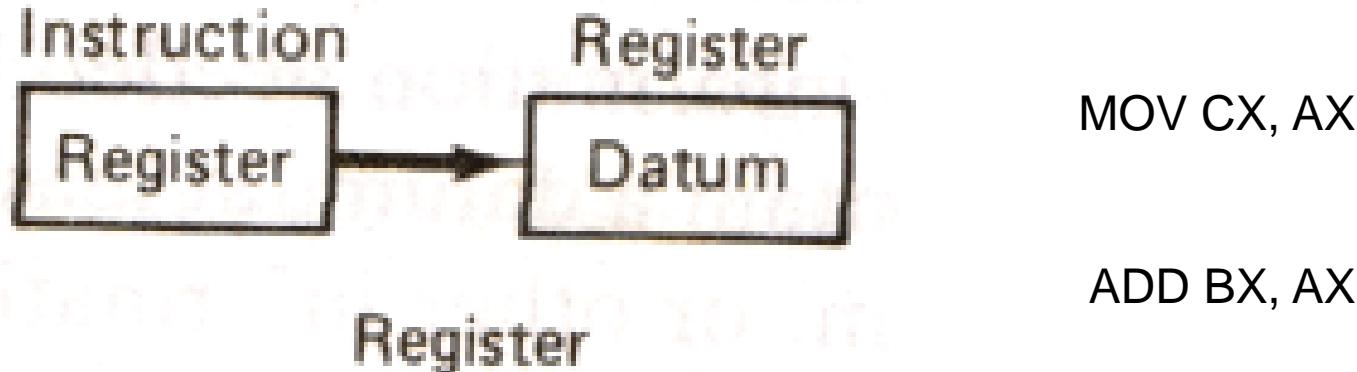
- Ways of locating data or operands.
- Describes the type of operands and the way they are accessed for executing an instruction.

# Addressing modes

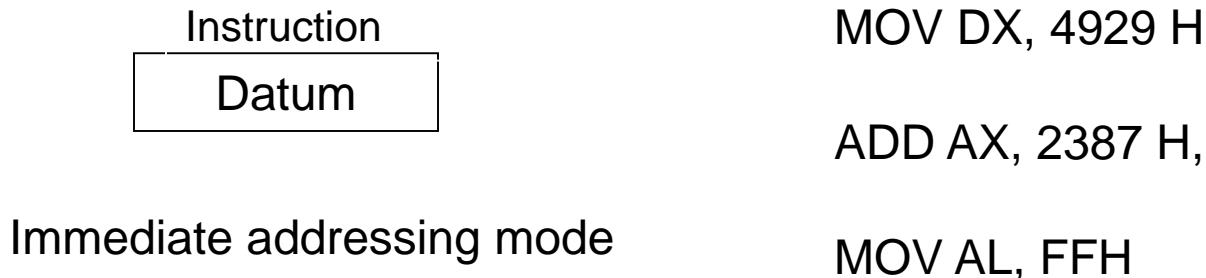
- DATA-ADDRESSING MODES
  - PROGRAM MEMORY-ADDRESSING MODES
  - STACK MEMORY-ADDRESSING MODES
- 
- **DATA-ADDRESSING MODES**
    - Register Addressing
    - Immediate addressing
    - Direct addressing
    - Register indirect addressing
    - Indexed
    - Register relative
    - Based indexed
    - Relative based indexed

# 8086 Addressing mode:

- Register addressing mode

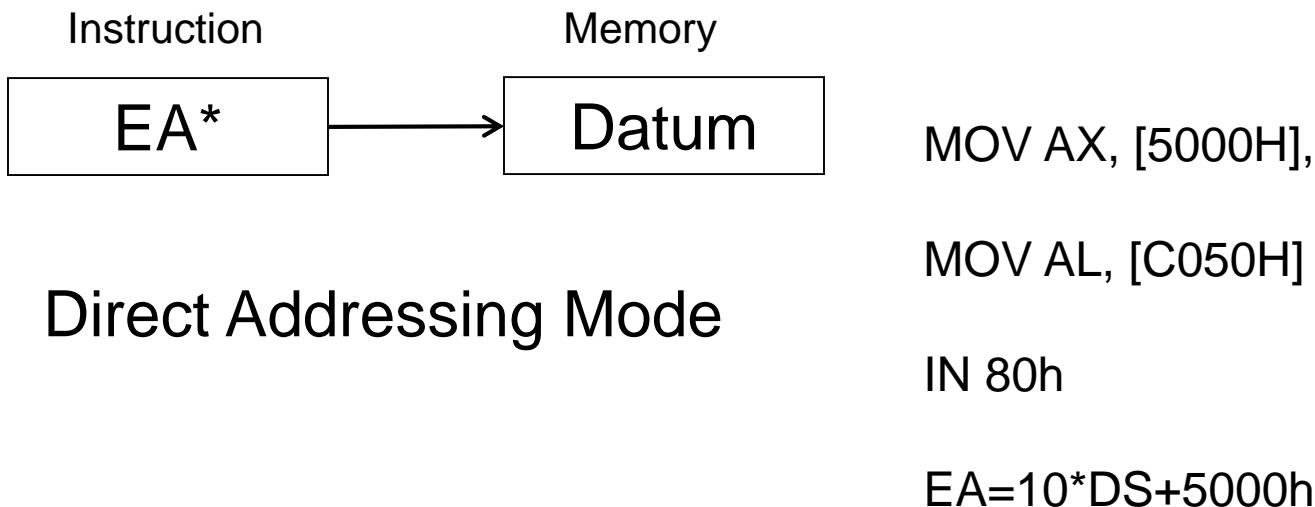


- Immediate addressing mode

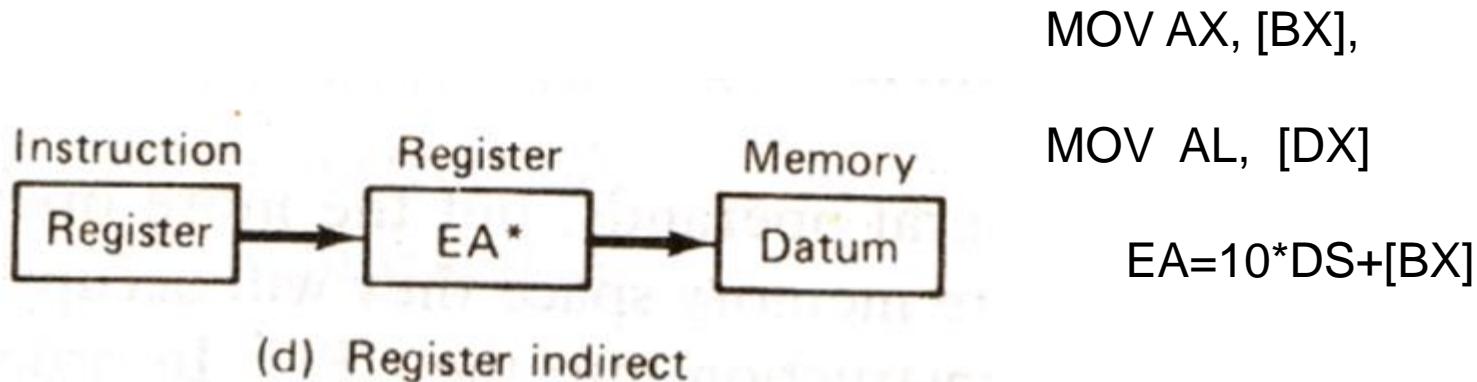


# 8086 Architecture

- Direct Addressing:
  - 16 bit memory address (offset) is specified in the instruction



- Register indirect:



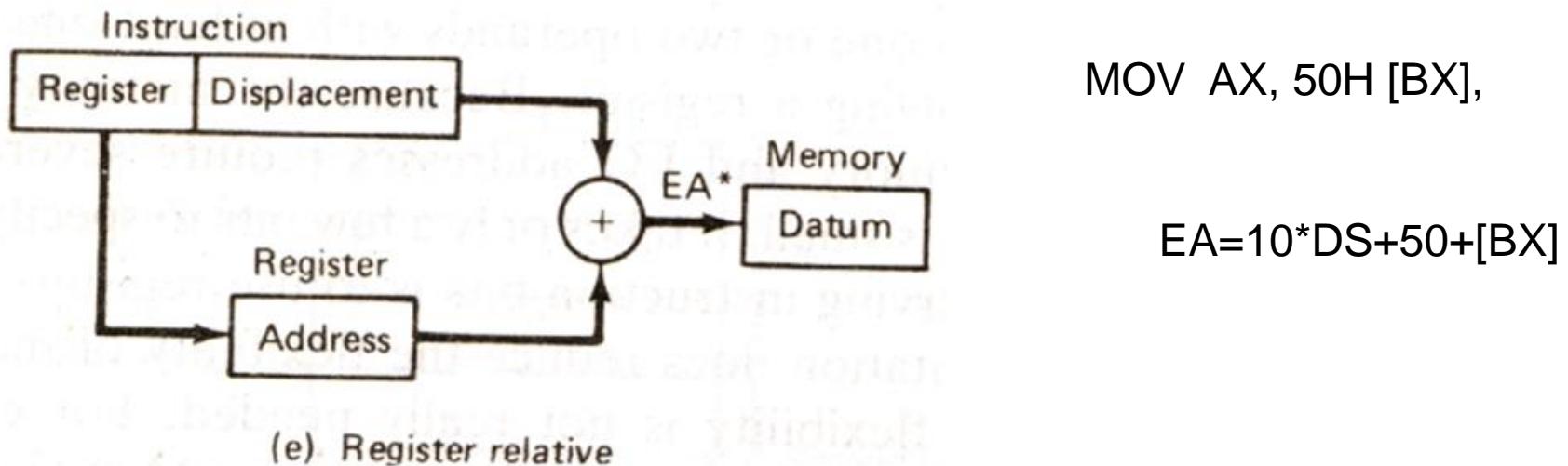
- **Indexed:** offset of the operand is stored in one of the index registers.

MOV AX, [SI],

EA=10\*DS+[SI]

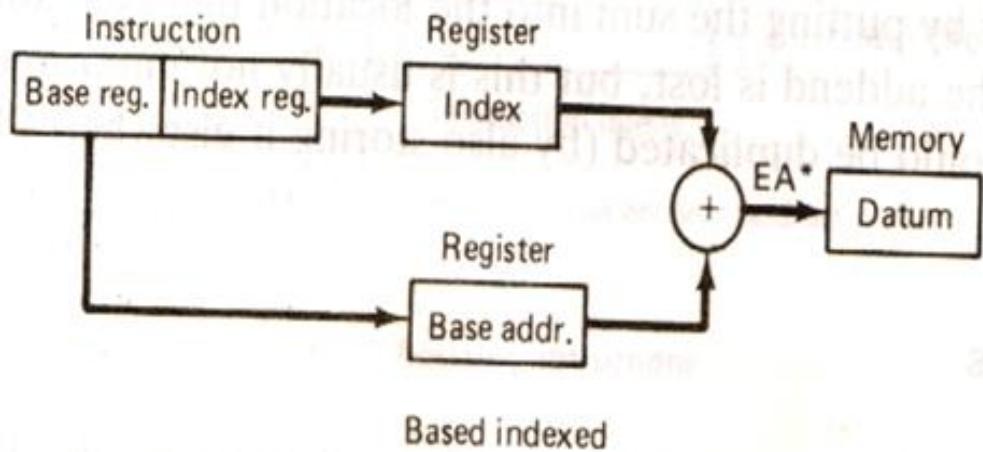
# 8086 Instruction set:

- Register relative addressing mode:
  - EA is formed by adding 8 or 16 bit displacement with the content of any one of the registers BX, BP, SI or DI in the segment(DS or ES)



# 8086 Architecture

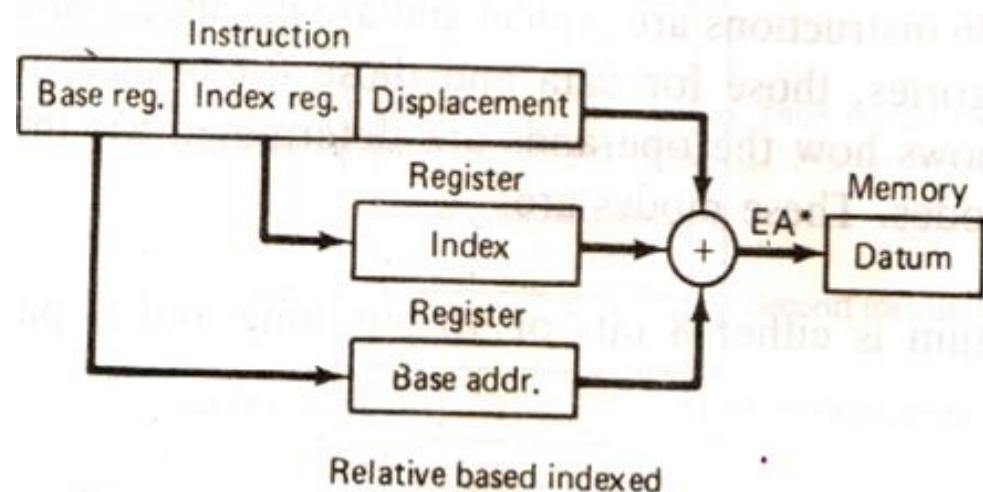
- Based Indexed:



MOV AX , [BX] [SI]

$$EA = 10^*DS + [BX] + [SI]$$

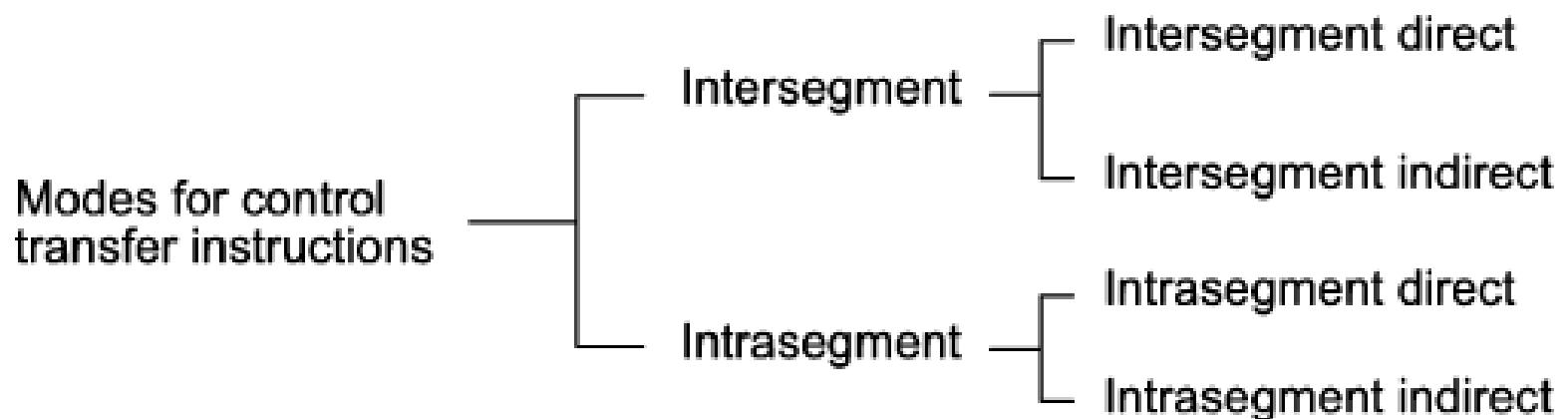
- Relative Based Indexed:



MOV AX, 50H[BX] [SI]

$$EA = 10^*DS + 50H + [BX] + [SI]$$

- For control transfer
  - Intra-segment direct Mode(within same)
  - Intra-segment indirect Mode
  - Inter-segment direct Mode (different)
  - Inter-segment indirect Mode



**Fig. Addressing Modes for Control Transfer Instructions**

# PROGRAM MEMORY-ADDRESSING MODES /control transfer instructions

- Intra-segment direct Mode
  - The displacement is computed relative to the content of the instruction pointer
  - Address to which control will be transferred is given by the sum of 8 or 16 bit displacement and current contents of IP
  - 8 bit displacement ( $-128 < d < +127$ ) –short jump
  - 16 bit displacement ( $-32768 < d < +32767$ ) –long jump
  - JMP SHORT LABEL
    - LABEL lies within  $-128 < d < +127$

# PROGRAM MEMORY-ADDRESSING MODES /control transfer instructions

- Intra-segment indirect Mode
  - JMP [BX]
  - Jump to effective address stored in BX

## PROGRAM MEMORY-ADDRESSING MODES /control transfer instructions

- Intersegment direct Mode
  - Provides branching from one code segment to another code segment.
    - JMP 5000h : 2000h
- Intersegment indirect Mode
  - The address to which control is to be transferred lies in a different segment and is passed to the instruction indirectly.
  - contents of a memory block containing four bytes, i.e. IP(LSB), IP(MSB), CS(LSB) and CS(MSB) sequentially.
  - JMP [2000]

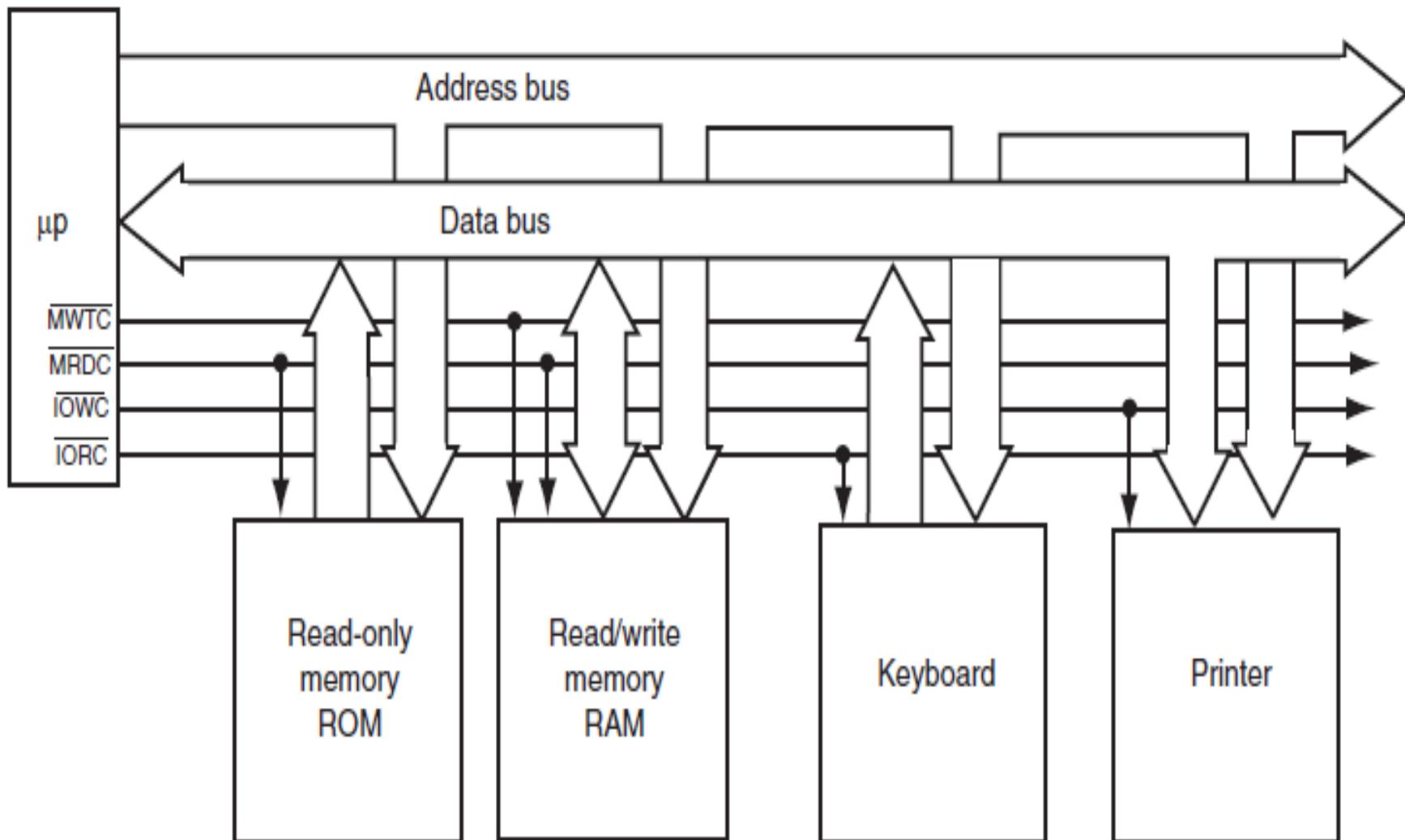
## STACK MEMORY-ADDRESSING MODES

- The stack memory is an LIFO (**last-in, first-out**) **memory**, which describes the way that data are stored and removed from the stack.
- Data are placed onto the stack with a **PUSH instruction** and removed with a **POP** instruction.

## Bus:

- A bus is a common group of wires that interconnect components in a computer system.
- The buses that interconnect the sections of a computer system transfer address, data, and control information between the microprocessor and its memory and I/O systems.
- 3 buses exist for this transfer of information: address, data, and control.
- Figure shows how these buses interconnect various system components such as the microprocessor, read/write memory (RAM), read-only memory (ROM or flash), and I/O devices.

# BUS:



**FIGURE** The block diagram of a computer system showing the address, data, and control bus structure.

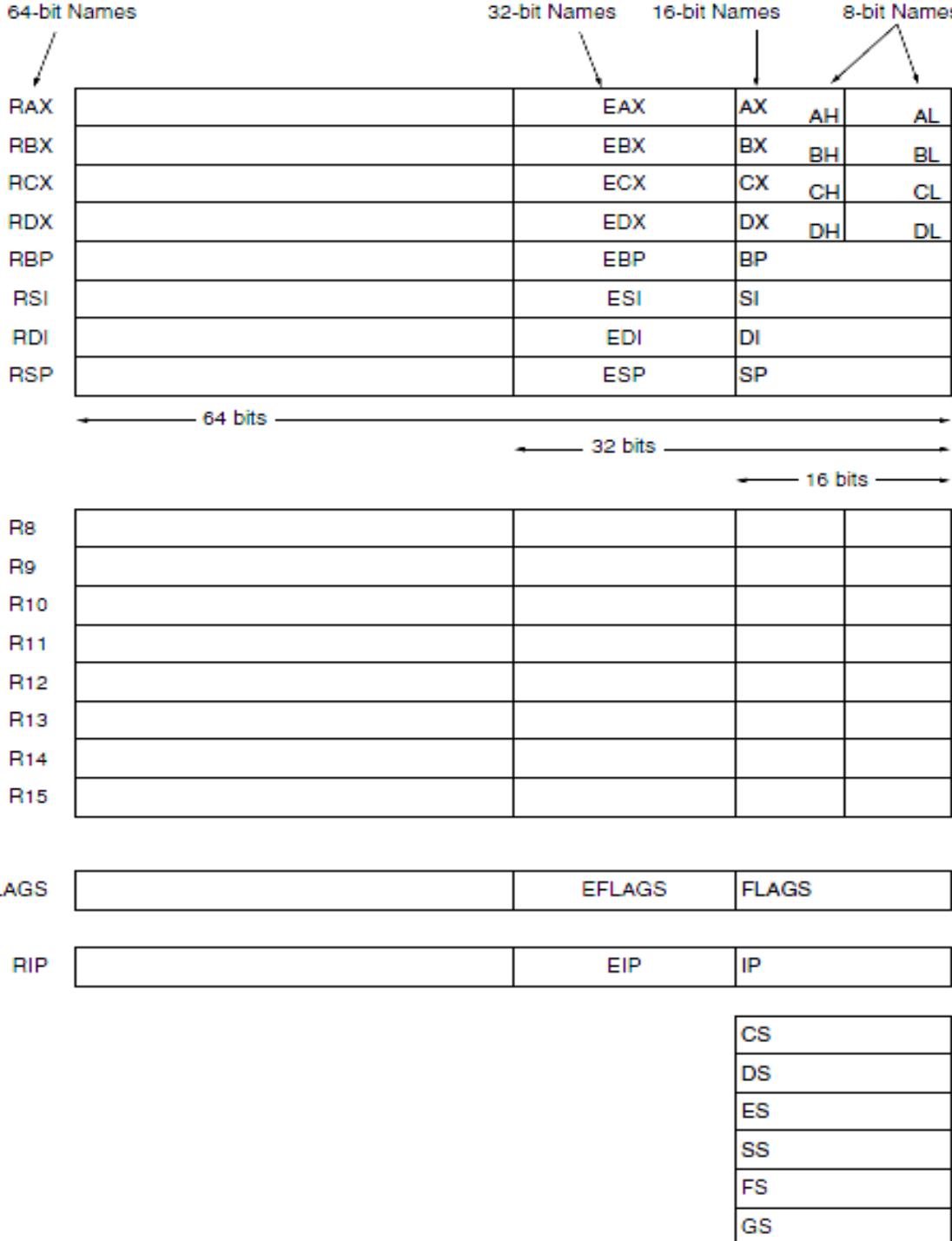
- **Address bus** requests a memory location from the memory or an I/O location from the I/O devices.
- **Data bus** transfers information between the microprocessor and its memory and I/O address space.
  - Data transfers vary in size, from 8 bits wide to 64 bits wide in various members of the Intel microprocessor family
- **Control bus** contains lines that select the memory or I/O and cause them to perform a read or write operation.
  - there are four control bus connections:(**memory read control**), (**memory write control**), (**I/O read control**), and (**I/O write control**).

- **The Programming Model.**
  - The programming model of the 8086 through the Core2 is considered to be **program visible** because its registers are used during application programming and are specified by the instructions.

# Programming model:

- The programming model contains 8-, 16-, and 32-bit registers.
- The Pentium 4 and Core2 also contain 64-bit registers when operated in the 64-bit mode as illustrated in the programming model.
  - The 8-bit registers are AH, AL, BH, BL, CH, CL, DH, and DL
  - The 16-bit registers are AX, BX, CX, DX, SP, BP, DI, SI, IP, FLAGS, CS, DS, ES, SS, FS, and GS.
  - The first 4, 16 bit registers contain a pair of 8-bit registers. e.g AX, which contains AH and AL.
  - The extended 32-bit registers are EAX, EBX, ECX, EDX, ESP, EBP, EDI, ESI, EIP (all are multipurpose) and EFLAGS. These 32-bit extended registers, and 16-bit registers FS and GS, are available only in the 80386 and above.

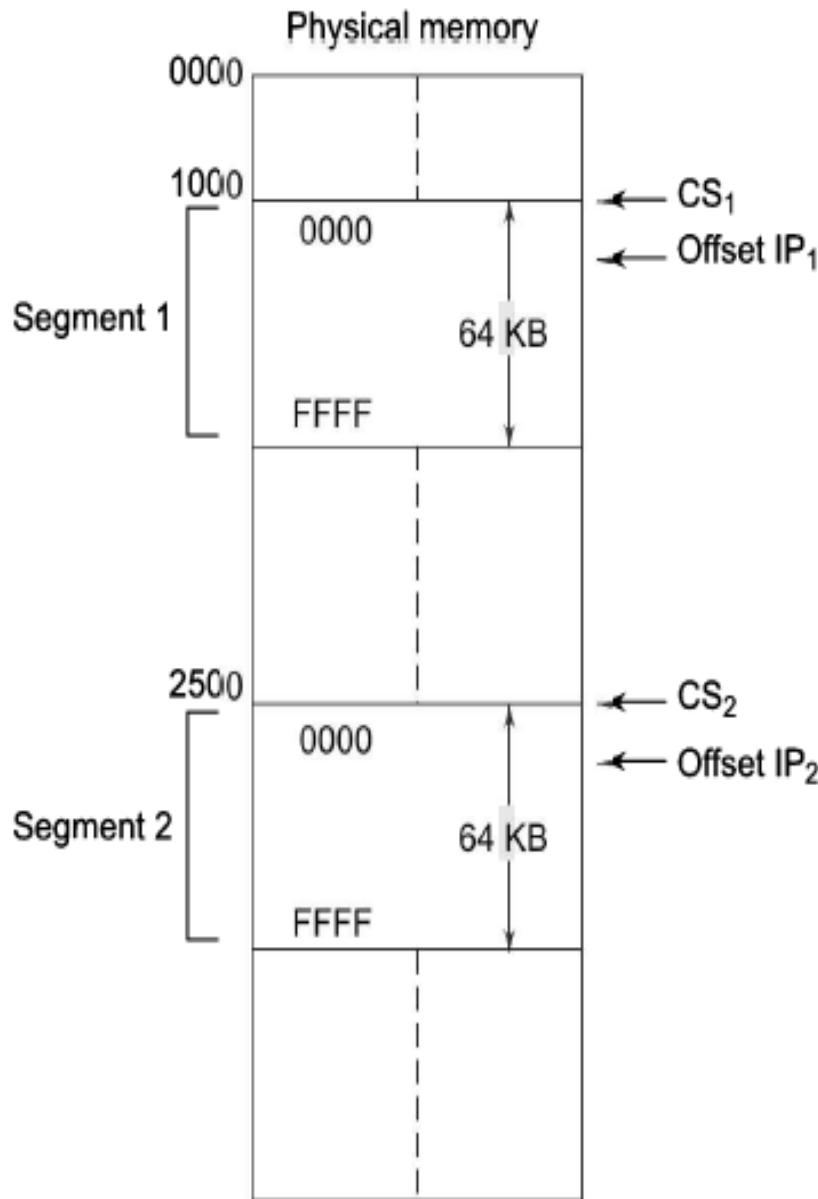
**FIGURE 2–1** The programming model of the 8086 through the Core2 microprocessor including the 64-bit extensions.



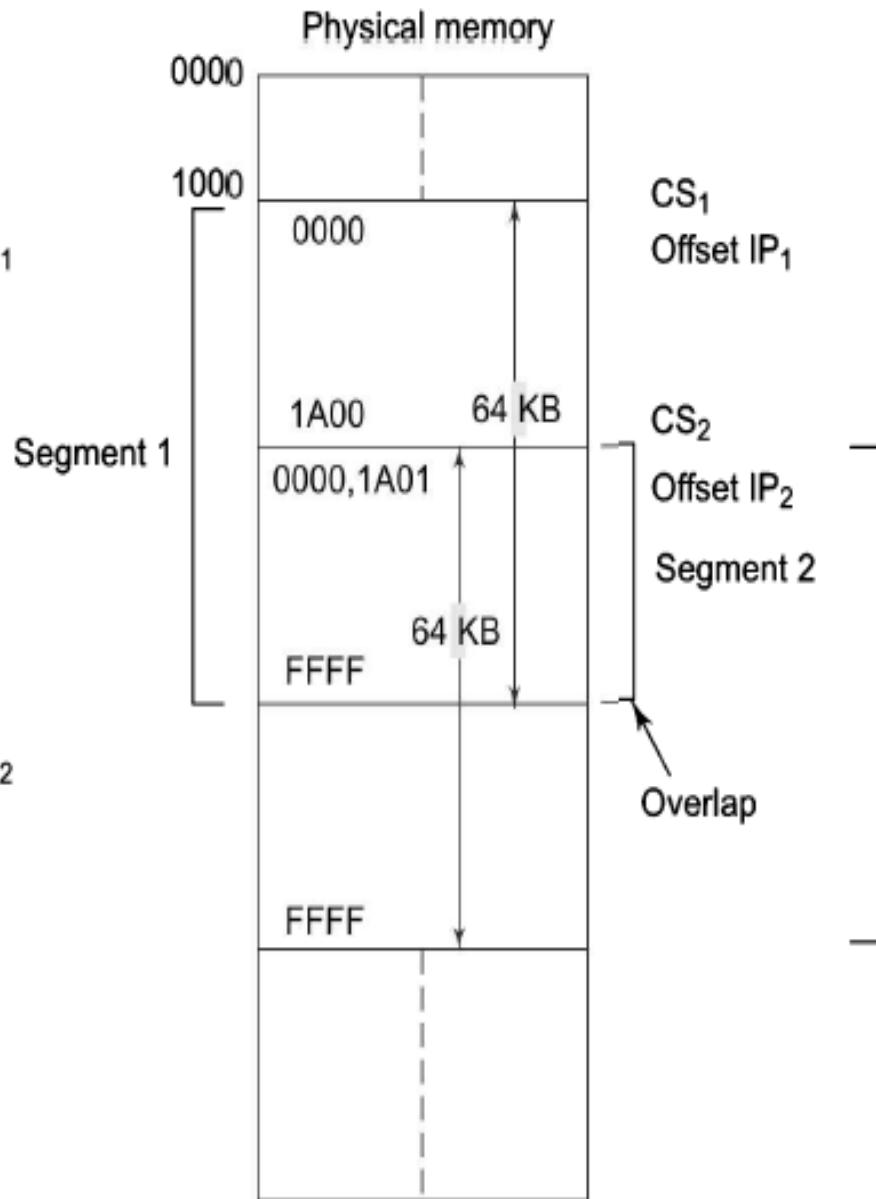
# **Memory segmentation**

- The complete physically available memory may be divided into a number of logical segments.
- Each segment is 64kb and addressed by one of the segment register.
- 8086 is able to address upto 1Mb of physical memory.
- Complete 1Mb memory is divided into 16 segments each of 64kb

- Advantages of memory segmentation are:
  - Allows memory capacity to be 1MB , although the actual address to be handled are 16 bit size.
    - It provides a powerful memory management mechanism.
  - Allows the placing of code, data and stack portions of the same program in different parts (segments) of memory for data and code protection.
    - Data related or stack related operations can be performed in different segments.
    - Code related operation can be done in separate code segments.
  - Permits a program and its data to be put into different areas of memory each time the program is executed.



**Fig. (a) Non-overlapping Segments**



**Fig. (b) Overlapping Segments**

# Features of 80286

- It is 16 bit MP but faster than 8086
- The 80286 CPU, with its 24-bit address bus is able to address  $2^{24}=16$  Mbytes of physical memory.
- The clock frequencies are 4MHz , 6MHz, 8MHz , 10 MHz and 12.5 MHz .
- 80286 is upwardly compatible with 8086 in terms of instruction set.

- Enhanced instruction set
  - It is 4 stage pipeline microprocessor .
- 
- **80286 works in 2 operating modes:**
  - **Real address mode**
    - In RM 80286 just acts as fast 8086.(addresses 1Mb memory)
    - 8086 program can be executed without modification in 80286
    - MMU and protection mechanisms are disabled in this mode.
  - **Protected virtual address mode(PVAM)**
    - 80286 can address 16Mb of memory
    - Supports multitasking –run several program at the same time.
    - Works with its MM and protection capabilities with advanced instruction set.
    - it can address upto 1GB virtual memory .

# Register Organization of 80286

- a) Eight 16-bit general purpose registers
- b) Four 16-bit segment registers
- c) Status and control registers
- d) Instruction Pointer

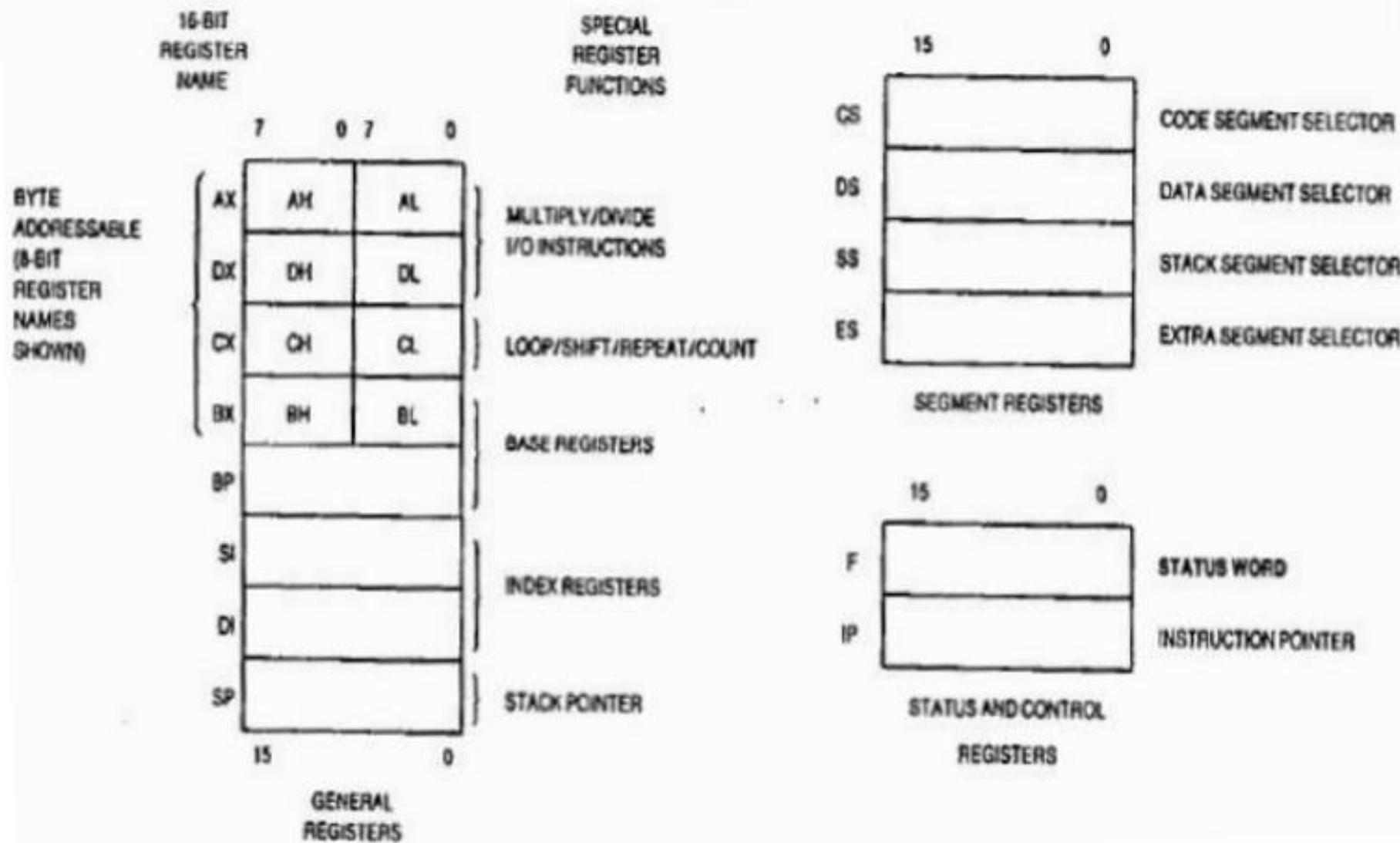


Fig.2.17 Register set of 80286

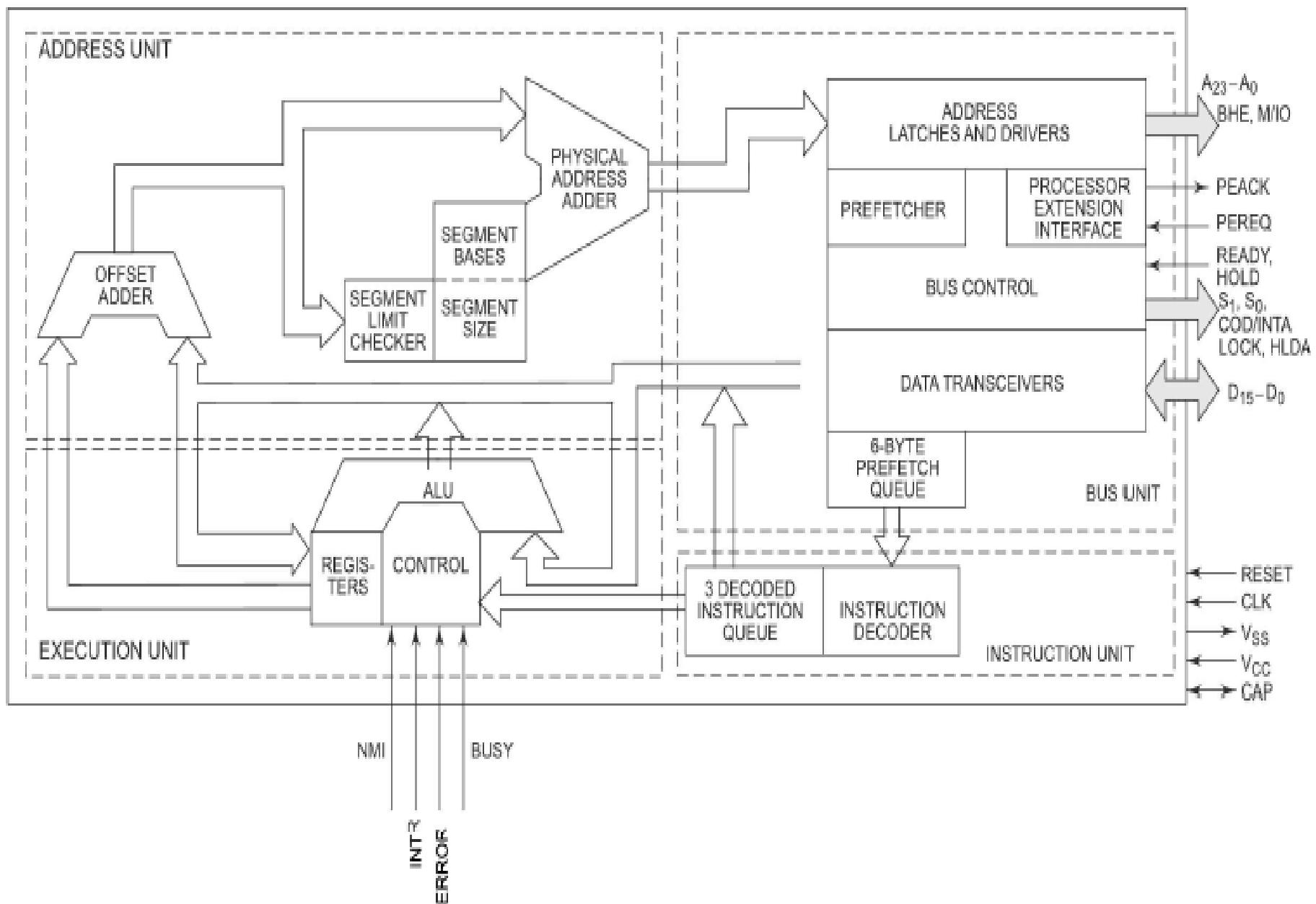


Fig. Internal Block Diagram of 80286 (Intel Corp.)

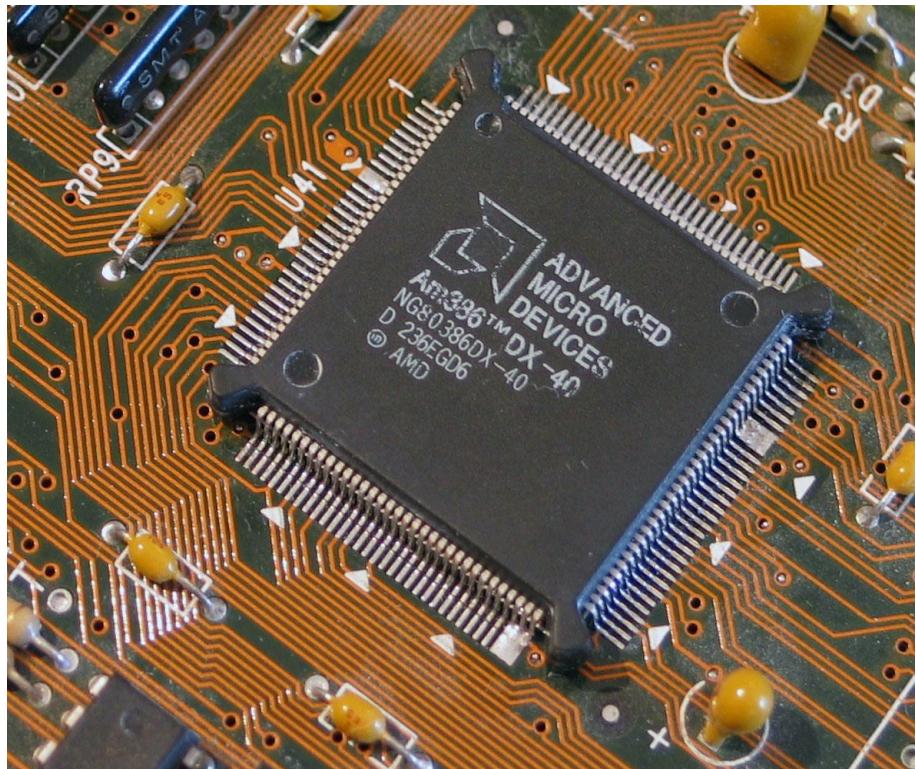
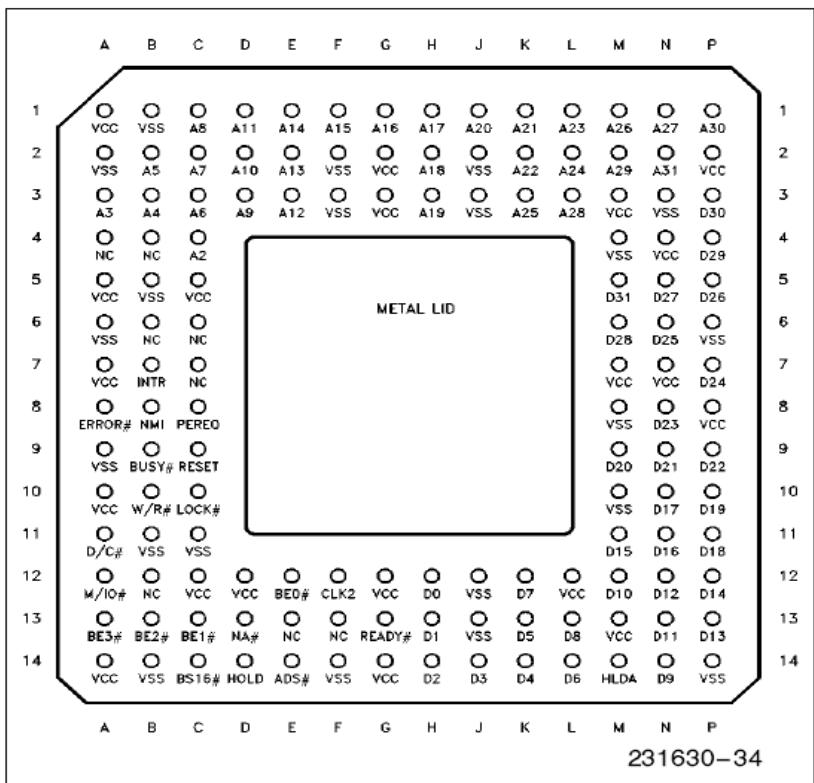
- Internal block diagram of 80286 contains 4 units :
- **Address Unit(AU)**
  - Address unit calculated the physical address of instruction and data that the CPU wants to access.
  - This address is send to BIU
- **Bus Unit(BU)**
  - Fetches instruction bytes from memory.
  - Bus Unit includes –
    - **Address latches and drivers-** transmits the physical address over the address bus A0-A23
    - **Prefetcher-** performs the task of prefetching
    - **Bus control module-** control the prefetcher module.
    - Processor extension interface module- takes care of communication between coprocessor and CPU
    - Transreceivers interface : control the internal data bus with the system bus
    - 6 byte prefetch queue: forwards the instructions to the IU.

- Instruction unit( IU)
  - It accepts instruction from prefetch queue and instruction unit decodes them one by one.
  - Output of decoding circuit drives the control circuit in the EU.
- Execution unit
  - Executes the instructions received from decoded instruction queue which sends the data part of the instructions over the data bus.
  - EU contains registers bank (for storing data) and ALU (for arithmetic and logical operations).

# 80386---32 bit processor

- Features of 80386:
- It is a 32 bit processor with 8, 16, 32-Bit Data Types
  - 8 General Purpose 32-Bit Registers
- Instruction set is compatible with all its predecessors.
- Very Large Address Space
  - With 32 address lines it can address  $2^{32} = 4$  GB Physical Memory
  - 4 GB each Segment Size
    - No of segments = 16384
  - Virtual space =  $16384 * 4\text{GB} = 64$  Terabyte

- 80386 is available in 132 PGA package and has 20MHz and 33 MHz versions
- The 80386 also includes 32-bit extended registers and a 32-bit address and data bus
- The 80386 has on chip address translation cache.
- The concept of paging is introduced in 80386 enables to organize available physical memory into page of size 4Kbytes each.



**Figure 1-2. Intel386™ DX PGA  
Pinout—View from Pin Side**

- 80386 can be supported by 80387 for mathematical data processing.
- 80386 has virtual 8086 mode which allow it to switch back and forth between real and protected mode.
- 80386 is available in different versions
  - **80386SX**
    - It has 16 bit data bus and 24 bit address bus
    - Low cost
    - Low power version used in many applications
  - **80386DX**
    - It has 32 bit address and data bus

**Thank You**

# Advanced Microprocessor (CSL211)

Unit1 Microprocessor architecture

Sandip J.Murchite

# Register organization of 8086

## 1. General data registers

- ◊ AX,BX,CX and DX are general purpose data register. Letter X is used to specify complete 16 bit register
- ◊ Register CX is used as a default counter in case of an loop instructions
- ◊ AX is used as an accumulator.

## 2. Segment registers

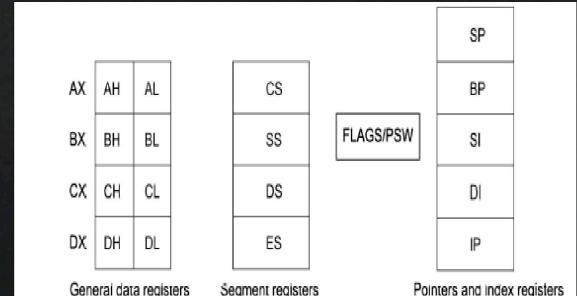
- ◊ 8086 addresses segmented memory. Complete 1MB memory is divided into 16 logical segments
- ◊ Each segment contain 64 KB of memory.
- ◊ There are four segment registers CS,DS,ES,SS
- ◊ CS: code segment of the memory where all executable program is stored.
- ◊ DS: data segment of the memory where the data is resided.
- ◊ ES: is a segment which is another data segment of the memory.
- ◊ SS: is a segment which is used to store stack data.

## 3. Flag register

- ◊ It indicate result of the computations in the ALU

## 4. Pointer and Index registers

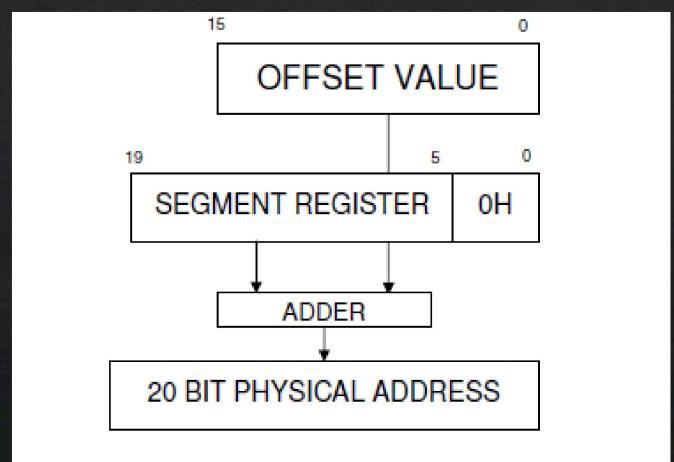
- ◊ Pointers contain offset within the particular segments. Index registers are used as general purpose registers as well as for offset storage.
- ◊ SI is used to store offset of source data in data segment while DI is used to store offset of destination data in extra segment



# Physical address formation

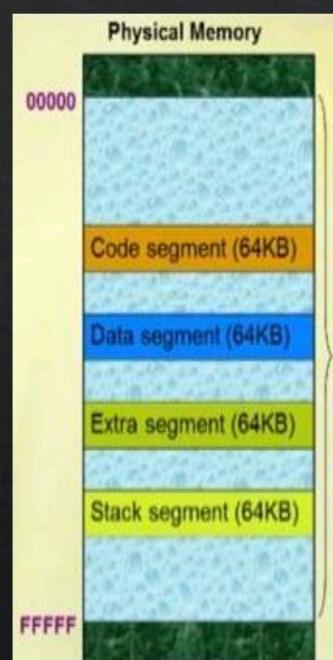
Physical address = Segment register \* 10H + Offset register

Segment address	→ 1005H
Offset address	→ 5555H
Segment address	→ 1005H → 0001 0000 0000 0101
Shifted by 4 bit positions	→ 0001 0000 0000 0101 0000
	+
Offset address	→ 0101 0101 0101 0101
Physical address	→ 0001 0101 0101 1010 0101 1    5    5    A    5



# Memory segmentation

- ❖ Memory in 8086 is organized as segmented memory.
- ❖ 16 bit contents of segment register points to starting location of a particular segment. To address a specific memory location within a segment, we need an offset. Offset address is also 16 bit ranging from 0000H to FFFFH
- ❖ Physical addresses range from 00000H to FFFFFH.
- ❖ Advantages of segmented memory scheme.
  - ❖ Allow memory capacity 1MB though actual addresses are 16 bits.
  - ❖ Allow placing of code ,data and stack portion of same program in different segments.
  - ❖ Permit program and its data to put into different area of memory each time program is executed.



# Flag register of 8086

- ◊ 8086 has 16 bit flag register divided into two parts: status flags and control flags
- ◊ **Trap flag** : IF this flag is set , the processor enters the single step execution mode. A trap is generated after execution of each instruction.
- ◊ **Interrupt flag**: if this flag is set mask able interrupts are recognized by the CPU.
- ◊ **Direction flag**: this is used for string manipulation
  - ◊ If this flag is reset string is processed from lowest to highest address.
  - ◊ If this flag is set string is processed from highest to lowest address.
- ◊ **Overflow flag**: this flag is set if an overflow occurs. If the result of signed operation is large enough to be accommodated in a destination register.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
X	X	X	X	O	D	I	T	S	Z	X	Ac	X	P	X	Cy

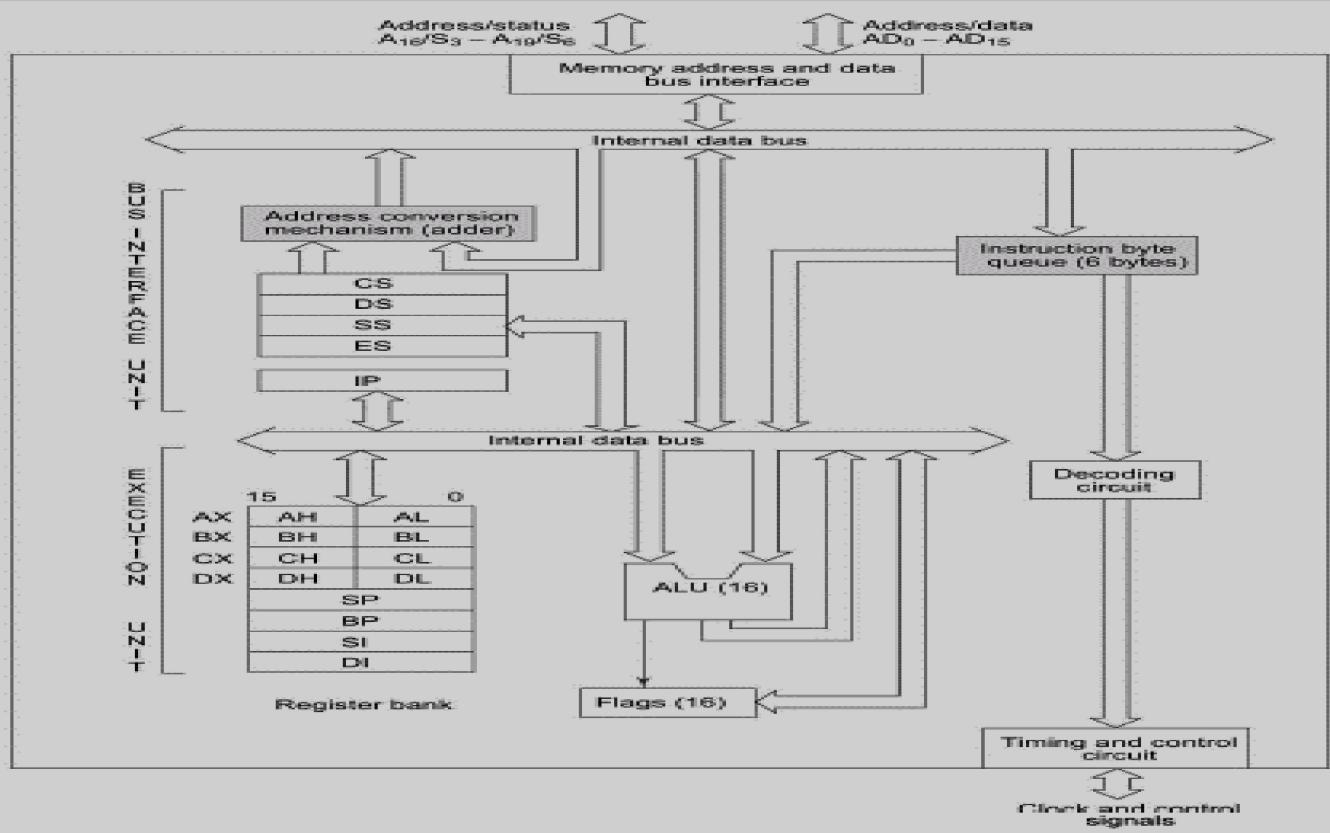


Fig. 1.2 8086 Architecture

# 8086 Architecture

Complete architecture of 8086 microprocessor is divided into two parts

## BIU

- It contain circuit for physical address calculation and pre decoding instruction byte queue.(6 Byte Queue)
- This unit is responsible for establishing communications with external devices and peripherals including memory.
- Complete physical address 20 bit is generated using segment register and offset registers, each 16 bit long.

## EU

- It contain register set of 8086 except segment registers and IP
- It has 16 bit ALU to perform Arithmetic and logical operations
- The decoding unit decodes opcode bytes issued from the instruction byte queue.
- Timing and control unit issue necessary control signals to execute the instruction opcode received from instruction byte queue.
- The EU pass the result to BIU for storing them in memory.

Hint :Size of largest instruction in 8086 is 6 byte therefore instruction byte queue is 6 byte

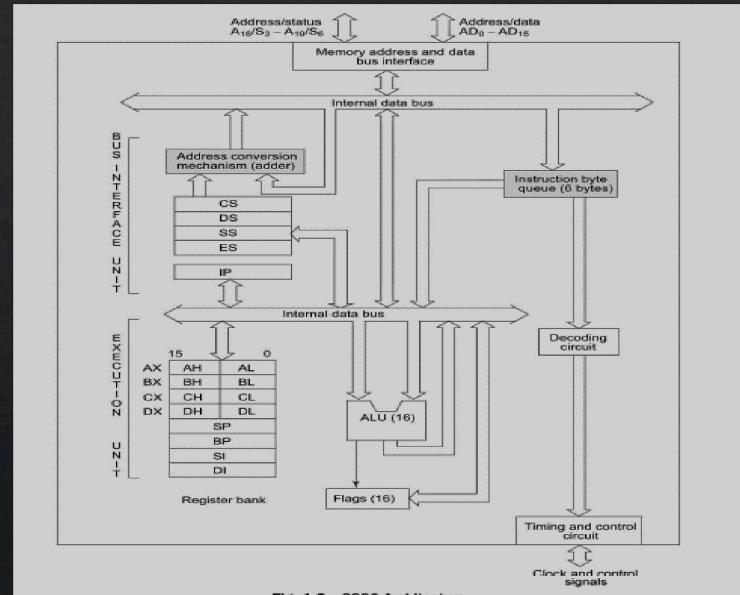
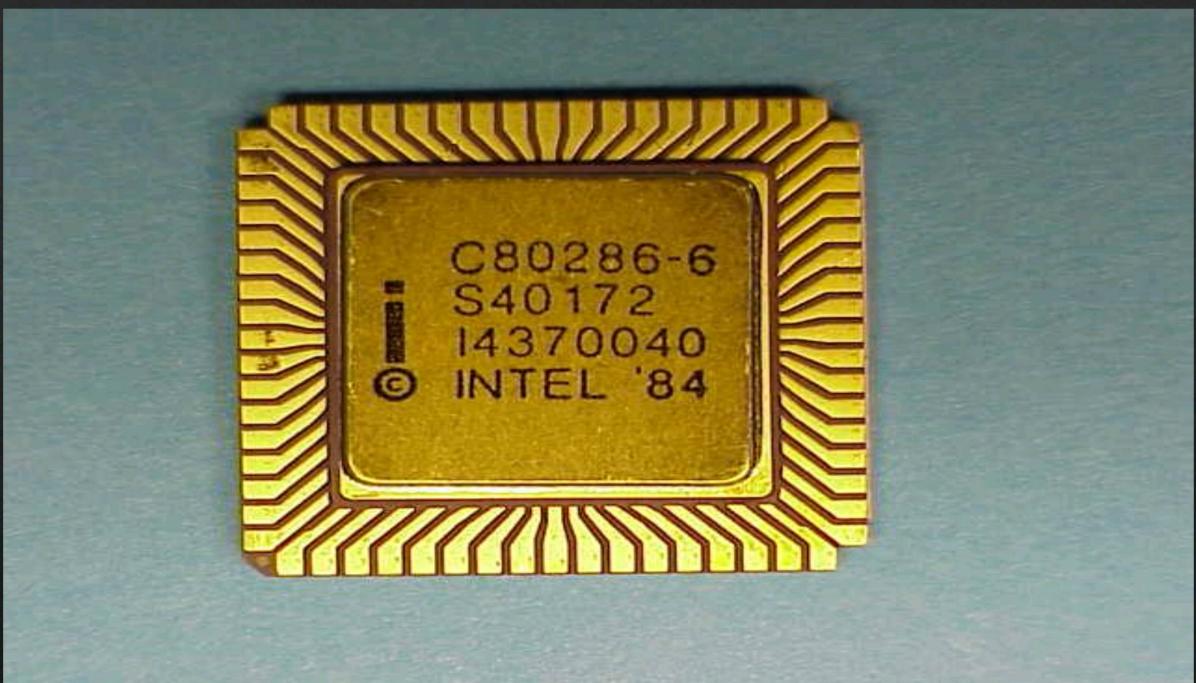


Fig. 1.2 8086 Architecture

Let's us start Intel 80286 Microprocessor.



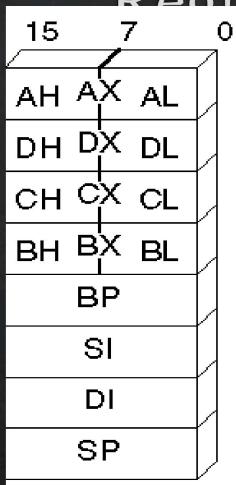
## Features of 80286

- ❖ 16 bit processor
- ❖ 24 bit address lines so access up to  $2^{24}=16$  MB of physical memory.
- ❖ Operated at different clock speeds 4 MHz,6MHz,8Mhz.
- ❖ Enhanced instruction set
- ❖ 68 pin leadless flat package
- ❖ Upward compatible with 8086.
- ❖ It supports a pipelined architecture.

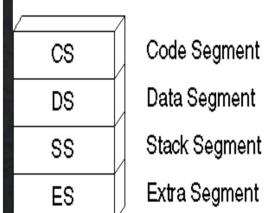
# Features of 80286

- ❖ Supports two operating modes
  - ❖ Real addressing mode
  - ❖ Protected addressing mode.
- In real mode ,80286 behaves as a fast 8086
- In protected mode ,80286 can address up to 1gb of memory.
- Provide memory management and protection circuitry
- It supports multiuser and multitasking environment
- It execute instruction in fewer clock cycles than 8086.

## Register organization of 80286



Segment Registers

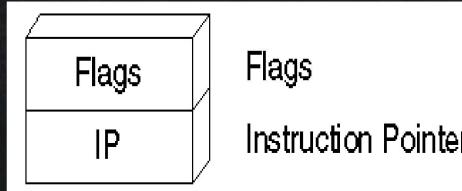


❖ 8, 16 bit general purpose registers

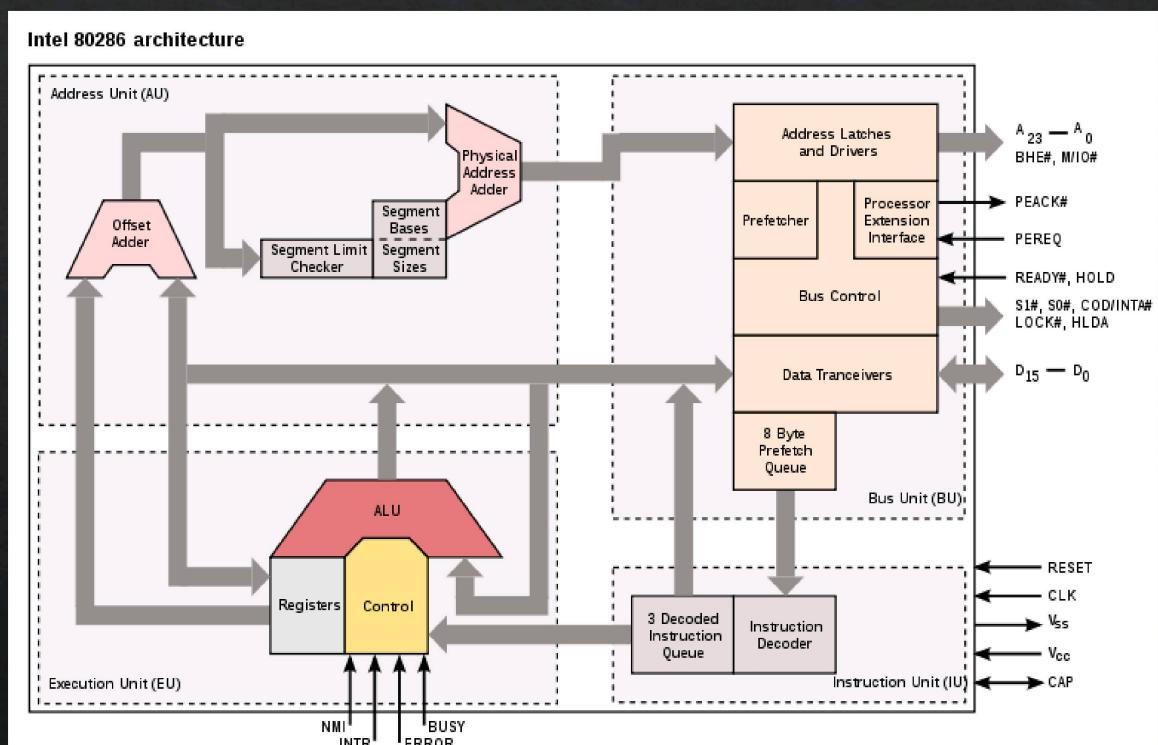
❖ 4, 16 bit segment registers

❖ Status and control register

❖ Instruction pointer.(IP)



# Internal block diagram of 80286



# Internal block diagram of 80286

❖ It contain four functional parts

1. Address unit (AU)

- ❖ The address unit (AU) computes the physical addresses that will be sent out to memory or IO .
- ❖ 80286 operate in two memory address modes
  - ❖ Real address mode. (1MB)
  - ❖ Protected virtual address mode. (1GB)

2. Bus unit (BU)

- ❖ It performs
  - ❖ All memory and IO read and writes
  - ❖ Pre fetches instruction bytes
  - ❖ Control transfer of data to and from coprocessor (80287)

3. Instruction unit (IU)

- ❖ It decodes up to three pre fetched instructions and hold them in a queue, where execution unit can access them.

4. Execution unit (EU)

- ❖ The execution unit (EU) uses its 16 bit ALU to execute instructions it receives from instruction unit.

## Limitation of 80286 microprocessor

- ❖ Only a 16 bit ALU
- ❖ Maximum segment size=64KB
- ❖ Cannot easily switch back and forth between real and protected mode.

## Features of 80386 microprocessor

- ❖ 32 bit processor that supports **8/16/32** bit data operands.
- ❖ **Compatibility** with existing processors like 80286 etc
- ❖ **32 bit ALU**
- ❖ No of segments=**16384**, each segment size=**4GB** therefore virtual address space= $16384 \times 4\text{gb} = \mathbf{64\text{TeraBytes}}$
- ❖ Physical memory size= $2^{32} = 4\text{GB}$
- ❖ The 80386 can be supported by 80387 for mathematical data processing.
- ❖ 80386 has on chip address translation cache.

# Features of 80386 microprocessor

- ❖ The concept of paging is introduced in 80386, size of each page is 4KBs under the segmented memory.
- ❖ 80386 has “virtual 8086” mode, which allow it to easily switch back and forth between real and protected mode.
- ❖ 80386 is available in versions
  - ❖ **80386SX-**
    - ❖ It has only a16 bit data bus and 24 bit address bus.
    - ❖ Low cost
    - ❖ Low power version of 80386
  - ❖ **80386DX**
    - ❖ IT has 32 bit address and data bus.

# Assignment no. 1 CSL211: AMP

- ❖ Q1. Draw block diagram of computer system showing address, data and control bus structure.
- ❖ Q2. Write Short note on
  - ❖ A) Programming Model
  - ❖ B) Register organization of 8086
  - ❖ C) memory segmentation of 8086
  - ❖ D) 8086 Flag register
- ❖ Q3. Describe internal architecture of 8086 microprocessor with neat diagram.
- ❖ Q4. Describe internal architecture of 80286 Microprocessor with neat diagram.
- ❖ Q5. List the features of 80386 Microprocessor.

# Unit2: Data and Control transfer operations

Sandip J.Murchite

# Data transfer instructions

- ❖ 1) Bswap R32
- ❖ Example: bswap %eax
- ❖ Description : In computer network, the data may arrive in big endian order while the IA 32 processor may need to operate on the data in little endian order.

**SYBTECH CSE\_CSL211(AMP)\_SJM**

Terminal

Open Save

```
1 .section .data
2
3
4 value:
5     .int 0xffffdeecc
6
7
8
9
10 .section .text
11
12 .globl _start
13 _start: nop
14
15     movl value,%eax
16     bswap %eax
17
18
19     movl $1,%eax
20     movl $0,%ebx
21     int $0x80
22
```

student@student-OptiPlex-3020:~

To run a command as administrator (user "root"), use "sudo <command>".  
See "man sudo\_root" for details.

```
student@student-OptiPlex-3020:~$ as -gstabs -o abc.o abc.s
student@student-OptiPlex-3020:~$ ld -o abc abc.o
student@student-OptiPlex-3020:~$ gdb -q abc
Reading symbols from abc...done.
(gdb) break*_start+1
Breakpoint 1 at 0x8048075: file abc.s, line 15.
(gdb) run
Starting program: /home/student/abc
SYBTech_CSE_211(AMP)_SJM
```

Breakpoint 1, \_start () at abc.s:15

```
15             movl value,%eax
(gdb) s
16             bswap %eax
(gdb) print/x $eax
$1 = 0xffffdeecc
(gdb) s
19             movl $1,%eax
(gdb) print/x $eax
$2 = 0xccceedff
(gdb)
```

Plain Text Tab Width: 8 Ln 12, Col 14

# Data transfer instructions

- ❖ 2) XCHG SRC,DEST
  - Values stored in source register and destination register get exchanged.
- ❖ Example: xchg %eax,%ebx
- ❖ Before execution : %eax=0xff and %ebx=0xaa
- ❖ After execution : %eax=0xaa and %ebx=0xff
- ❖ 3) Cmovcc src,dest
- ❖ Example: cmovae src,dest
- ❖ Description: it move content from source to destination only when destination operand is above or equal than the source.
- ❖ Destination operand $\geq$  Source operand.
- ❖ Source operand can be register or memory, but destination operand can only be register.
- ❖ This instruction is used immediately after compare instruction.

Terminal

Open Save

```
1 .section .data
2
3
4 value:
5     .int 60
6 value1:
7     .int 12
8
9
10
11
12 .section .text
13
14 .globl _start
15 _start: nop
16
17     movl value,%eax
18     movl value1,%ebx
19     cmpl %ebx,%eax
20     cmovae %eax,%ebx
21     xchg %eax,value1
22
23
24     movl $1,%eax
25     movl $0,%ebx
26     int $0x80
27
```

student@student-OptiPlex-3020: ~

(gdb) run

Starting program: /home/student/abc

Breakpoint 1, start () at abc.s:17

17 movl value,%eax

(gdb) s

18 movl value1,%ebx

(gdb) s

19 cmpl %ebx,%eax

(gdb) s

20 cmovae %eax,%ebx

(gdb) s

21 xchg %eax,value1

(gdb) print/d \$ebx

\$1 = 60

(gdb) x/d &value1

0x804909b: 12

(gdb) s

24 movl \$1,%eax

(gdb) x/d &value1

0x804909b: 60

(gdb) print/d \$eax

\$2 = 12

(gdb)

Plain Text Tab Width: 8 Ln 21, Col 25

02 March,2021 (Tuesday) 10:40 AM

## Data transfer instructions

- ❖ Machine stack
- ❖ Pushl src
- ❖ Description: The source operand of push instruction can be an immediate data,register or memory variable.
- ❖ Example: Pushl %eax ,pushw %ax SYBTECH CSE\_CSL211(AMP)\_SJM
- ❖ Popl dest
- ❖ Description: the Destination operand of pop instruction can only register or memory variable.
- ❖ Example: Popl,%eax , popw %ax

Terminal

student@student-OptiPlex-3020: ~

```
1 .section .data
2
3
4
5 value:
6     .int 0x11224499
7
8
9 .section .text
10
11 .globl _start
12 _start: nop
13     movl $0xffffdeecc,%eax
14     movl $0xaabbccdd,%ebx
15     movl value,%ecx
16     pushl %ebx
17     pushw %cx
18     inc %esp
19     popw %bx
20     popl %edx
21
22
23
24     movl $1,%eax
25     movl $0,%ebx
26     int $0x80

Breakpoint 1, _start () at stack.s:13
13     movl $0xffffdeecc,%eax
(gdb) s
14     movl $0xaabbccdd,%ebx
(gdb) s
15     movl value,%ecx
(gdb) s
16     pushl %ebx
(gdb) print/x $esp
$1 = 0xbffff0e0
(gdb) s
17     pushw %cx
(gdb) print/x $esp
$2 = 0xbffff0dc
(gdb) s
18     inc %esp
(gdb) print/x $esp
$3 = 0xbffff0da
(gdb) s
19     popw %bx
(gdb) print/x $esp
$4 = 0xbffff0db
(gdb) s
20     popl %edx
(gdb) print/x $esp
$5 = 0xbffff0dd
(gdb) s
24     movl $1,%eax
(gdb) print/x $esp
$6 = 0xbffff0e1
(gdb) 
```

SYBTEC\_CSC\_SPL211(AMP)\_SJM

Plain Text ▾ Tab Width: 8 ▾ Ln 21, Col 5 ▾ INS

# Data transfer instructions

- ❖ Handling signed and unsigned numbers
- ❖ Double the size of the source operand by means of sign extension
- ❖ cbtw
  - ❖ The CBW (convert byte to word) instruction copies the sign (bit 7) in the source operand into every bit in the AH register.
- ❖ Cwtl
  - ❖ The CWDE (convert word to double word) instruction copies the sign (bit 15) of the word in the AX register into the higher 16 bits of the EAX register.
- ❖ Cltd
  - ❖ Input 32 bit number in register eax and the resultant 64 bit number is made available in registers edx and eax.

Terminal

The screenshot shows a terminal window with a dark theme. On the left is a dock with various icons. The main area contains two panes. The left pane displays assembly code:

```
1 .section .data
2
3 value:
4     .byte 0xff
5 value1:
6     .byte 0x7f
7
8
9
10
11 .section .text
12
13 .globl _start
14 _start:    nop
15
16     movb value,%al
17     cbtw
18     cwtl
19     cltd
20     movb value1,%al
21     cbtw
22     cwtl
23     cltd
24
25     movl $1,%eax
26     movl $0,%ebx
27     int $0x80
28
```

The right pane shows a GDB session:

```
student@student-OptiPlex-3020: ~
Starting program: /home/student/loop

Breakpoint 1, _start () at loop.s:16
16          movb value,%al
(gdb) s
17          cbtw
(gdb) print/x $al
$1 = 0xff
(gdb) s
18          cwtl
(gdb) print/x $ax
$2 = 0xffff
(gdb) s
19          cltd
(gdb) print/x $eax
$3 = 0xffffffff
(gdb) s
20          movb value1,%al
(gdb) print/x $edx
$4 = 0xffffffff
(gdb) print/x $eax
$5 = 0xffffffff
(gdb) s
21          cbtw
(gdb) print/x $al
$6 = 0x7f
(gdb) s
22          cwtl
(gdb) print/x $ax
$7 = 0x7f
(gdb) s
23          cltd
(gdb) print/x $eax
$8 = 0x7f
(gdb) ■
```

At the top of the right pane, there is a watermark: SYBTECH\_CSE\_CSL211(AMP)\_SJM.

Bottom status bar: Plain Text ▾ Tab Width: 8 ▾ Ln 5, Col 8 ▾ INS

# Data Transfer instructions

- ❖ Direct addressing mode
- ❖ base address (offset address, index, size)
  
- ❖ values:  
❖ .int 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60
- ❖ Example: To reference the value 20 from the values array shown, you would use the following
- ❖ instructions:  
❖ Movl \$2,%edi  
❖ Movl values(%edi,4)

# Data transfer instruction

- ❖ indirect addressing with registers
  - ❖ you can get the memory location address of the data value by placing a dollar sign (\$) in front of the label in the instruction.
  - ❖ movl \$values, %esi
- SYBTECH CSE\_CSL211(AMP)\_SJM**
- ❖ This instruction places the value contained in the memory location 8 bytes after the location pointed to by the esi register to register eax register.
  - ❖ Movl 8(%esi),%eax

Terminal

student@student-OptiPlex-3020:~

Breakpoint 1 at 0x8048075: file large.s, line 13.

(gdb) run

Starting program: /home/student/large

Breakpoint 1, \_start () at large.s:13

13 movl \$1,%edi

(gdb) s

14 loop: movl value(%edi,4),%ebx

(gdb) s

15 movl \$value,%esi

(gdb) print/d \$ebx

\$1 = 11

(gdb) s

16 movl 8(%esi),%eax

(gdb) s

17 movl \$0,16(%esi)

(gdb) print/d \$eax

\$2 = 33

(gdb) s

19 movl \$1,%eax

(gdb) x/5d &value

0x804909c: 110 11 33 44

0x80490ac: 0

(gdb) s

SYBTECH\_ESE\_CSL211(AMP)\_SJM

Plain Text ▾ Tab Width: 8 ▾ Ln 18, Col 5 ▾ INS

```
1 .section .data
2
3
4
5 value:
6     .int 110,11,33,44,550
7
8 .section .text
9
10 .globl _start
11 _start: nop
12
13     movl $1,%edi
14 loop: movl value(%edi,4),%ebx
15     movl $value,%esi
16     movl 8(%esi),%eax
17     movl $0,16(%esi)
18
19     movl $1,%eax
20     movl $0,%ebx
21     int $0x80
22
```

## Part II: Control Transfer Instructions

- ❖ IA 32 processors, the target address in a control transfer instruction can be specified in one of the following ways.
  - 1. Immediate constant
    - ❖ Jmp swapbyte
    - ❖ Jmp 0x100
    - ❖ Target address = A+0x100H where A= address of the next instruction.
    - ❖ Absolute address: jmp \*0x100
    - ❖ Target address=0x100h

## Part II: Control Transfer Instructions

- ◊ IA 32 processors, the target address in a control transfer instruction can be specified in one of the following ways.

### 2. Register addressing

- ◊ `Jmp %eax`  
SYBTECH CSE\_CSL211(AMP)\_SJM  
(Target address may be specified using a register)

### 2. Memory addressing mode:

the address of the target instruction is read from memory.

- `Jmp (%eax)`
- Address of target instruction is stored in memory location whose address is available in register eax

Terminal

Open Save

```
1
2
3
4
5
6
7
8
9 .section .text
10
11 .globl _start
12 _start: nop
13
14     movl $1,%eax
15     jmp end
16     movl $30,%ebx
17     int $0x80
18
19
20 end:  movl $20,%ebx
21     int $0x80
22
```

student@student-OptiPlex-3020:~\$ as -gstabs -o abc.o abc.s  
student@student-OptiPlex-3020:~\$ ld -o abc abc.o  
student@student-OptiPlex-3020:~\$ ./abc  
student@student-OptiPlex-3020:~\$ echo \$?  
20  
student@student-OptiPlex-3020:~\$ objdump -D abc  
  
abc: file format elf32-i386  
  
Disassembly of section .text:  
  
08048054 <\_start>:  
 8048054: 90 nop  
 8048055: b8 01 00 00 00 mov \$0x1,%eax  
 804805a: eb 07 jmp 8048063 <end>  
 804805c: bb 1e 00 00 00 mov \$0x1e,%ebx  
 8048061: cd 80 int \$0x80  
  
08048063 <end>:  
 8048063: bb 14 00 00 00 mov \$0x14,%ebx  
 8048068: cd 80 int \$0x80  
  
Disassembly of section .stab:

Plain Text Tab Width: 8 Ln 1, Col 1 INS

# Conditional Control transfer instructions

- ◆ IA32 architectures support conditional control transfer instruction
  - ◆ jcc target
  - ◆ Examples: je, jz,jne,ja,jnbe,jnc,jc

$$F(x) = (x+x) \quad \text{for } x > 2$$

$$(X+3) \quad \text{for } X \leq 2$$

- ◆ Assembly code for Function f(X)

Assume value of X=5 which is loaded in register %ebx

Movl \$5,%ebx

%ebx=5

Movl \$3,%eax

%eax=3

Cmpl \$2,%ebx

%ebx-2

Jle next

Movl %ebx,%eax

**Next:** addl %ebx,%eax

Terminal

Open Save

80386 ALP to evaluates the sum of first n integer by iteration

student@student-OptiPlex-3020: ~

```
(gdb) break*_start+1
Breakpoint 1 at 0x8048075: file loop.s, line 14.
(gdb) run
Starting program: /home/student/loop

Breakpoint 1, _start () at loop.s:14
14      movl $0,%eax
(gdb) s
15      movl value,%ecx
(gdb) s
17      back: addl %ecx,%eax
(gdb) s
18      loop back
(gdb) s
17      back: addl %ecx,%eax
(gdb) s
18      loop back
(gdb) s
17      back: addl %ecx,%eax
(gdb) s
18      loop back
(gdb) s
17      back: addl %ecx,%eax
(gdb) s
18      loop back
(gdb) s
21      movl $1,%eax
(gdb) print/d $eax
$1 = 10
(gdb) 
```

SYBTECH\_CSE\_COL21(AMP)\_SJM

Plain Text Tab Width: 8 Ln 12, Col 14 INS

# Function call and Return

- ❖ Functions are subprograms to which the control of execution is passed temporarily. After execution of the called function is completed , the control is transferred back to the calling program.
- ❖ The most important reason is to reuse the code
- ❖ Example: if in a program several strings are to be displayed on the screen at different time , it is better to write a function to display one string. Each time a string is to be displayed , this function can be called.
- ❖ Writing function in a program is to make it readable and understandable.

## CALL FUNCTION

0X100C PUSH %EAX

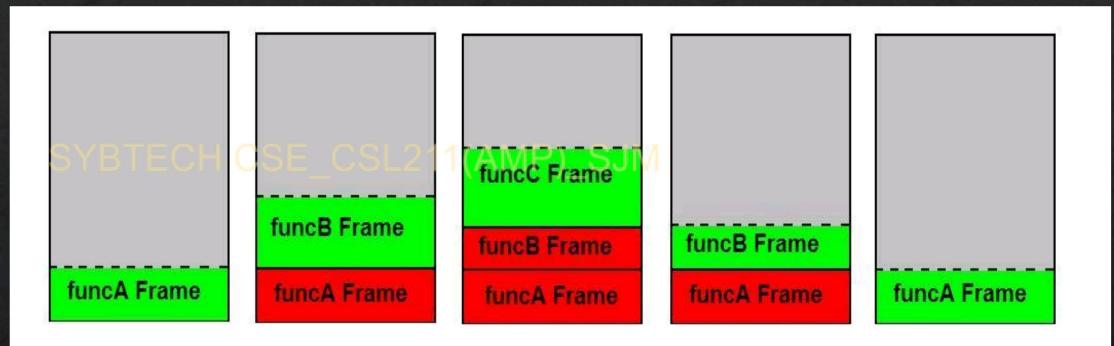
**SYBTECH CSE\_CSL211(AMP)\_SJM**

0X1100 **FUNCTION:**

- ❖ During execution of CALL instruction ,address of next instruction is saved on the stack. Stack pointer decremented by 4
- ❖ The register EIP changed to 0x00001100,thereby control transfer to function **FUNCTION**
- ❖ IA32 architectures provide an instruction called RET to return control to the caller after execution of the called function is completed.
- ❖ The RET instruction pops a 32 bit offset from top of the stack into EIP register, thereby causing a control transfer.

# Stack frame

- ❖ Each function call results in a creation of a stack frame.



# Function prologue and epilogue

**Function:**

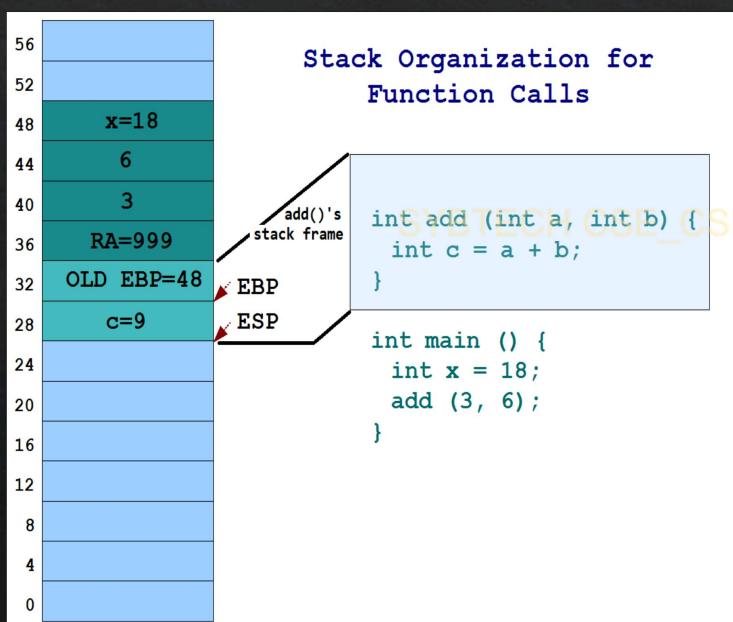
```
pushl %ebp  
movl %esp, %ebp
```

```
.  
. .
```

```
movl %ebp, %esp  
popl %ebp  
ret
```

- ❖ The first two instructions at the top of the function code
  - 1) save the original value of EBP to the top of the stack
  - 2) copy the current ESP stack pointer to the EBP register.
- ❖ SYBTECH CSE CSL211(AMP) SJM
  - ❖ After the function processing completes
- ❖ the last two instructions in the function
  - 1. Retrieve the original value in the ESP register that was stored in the EBP register.
  - 2. Restore the original EBP register value.

# Stack organization for function call



Function: Pushl %ebp  
movl %esp,%ebp

....

....

Movl %ebp,%esp  
Pop %ebp  
Ret

In GNU environment, **ebp** is used to store address of current frame.

When function is called, it must save frame pointer and must restore it before returning the control to the caller.

## Assembly language program on Function call

```
.section .data  
Output:  
    .asciz "This is section %d\n"  
.section .text  
.globl _start  
_start: pushl $1  
        pushl $output  
        call printf  
        addl $8,%esp  
        call function  
        pushl $3  
        pushl $output  
        call printf  
        addl $8,%esp  
        movl $1,%eax  
        movl $0,%ebx  
        int $0x80
```

Function: pushl %ebp  
movl %esp,%ebp

Entry Code

pushl \$2  
pushl \$output  
call printf  
addl \$8,%esp

Movl %ebp,%esp  
Popl %ebp  
ret

Exit Code

This is section 1  
This is section 2  
This is section 3

Terminal

The screenshot shows a Linux desktop environment with a terminal window open in the background and a file editor window in the foreground.

File icons in the dock:

- Terminal
- Open
- File
- Recycle Bin
- Firefox
- File Manager
- Terminal
- Amazon
- System Settings
- Text Editor
- Terminal
- Terminal

Terminal window content:

```
student@student-OptiPlex-3020:~$ clear
student@student-OptiPlex-3020:~$ as -gstabs -o loop.o loop.s
student@student-OptiPlex-3020:~$ ld -dynamic-linker /lib/ld-linux.so.2 -lc -o loop loop.o
student@student-OptiPlex-3020:~$ ./loop
This is section 1
This is section 2
This is section 3
```

File Editor window content:

```
4 .asciz "This is section %d\n"
5
6
7 .section .text
8
9 .globl _start
10 _start: nop
11     pushl $1
12     pushl $output
13     call printf
14     addl $8,%esp
15     call overhere
16     pushl $3
17     pushl $output
18     call printf
19     addl $8,%esp
20     movl $1,%eax
21     movl $0,%ebx
22     int $0x80
23 overhere: pushl %ebp
24     movl %esp,%ebp
25     pushl $2
26     pushl $output
27     call printf
28     addl $8,%esp
29     movl %ebp,%esp
30     popl %ebp
31     ret
```

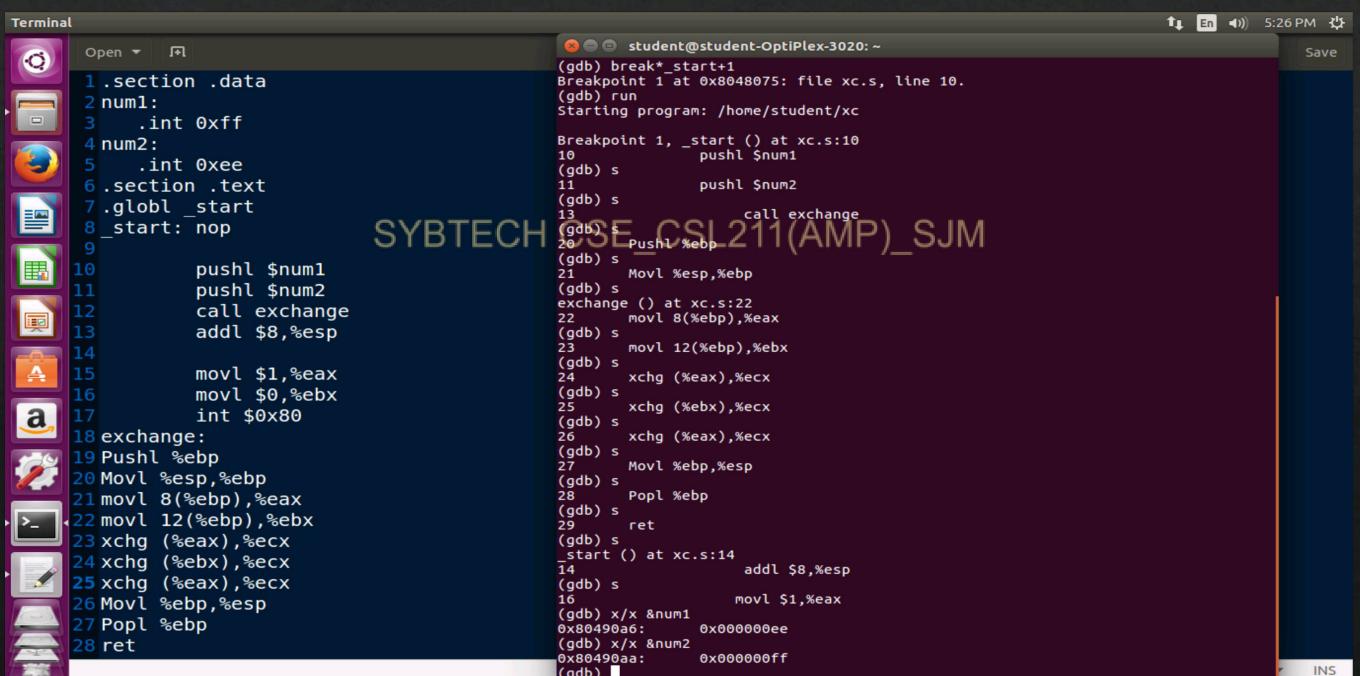
File Editor status bar:

- Plain Text
- Tab Width: 8
- Ln 31, Col 12
- INS

# Parameter Passing Conventions

- ❖ There are two parameter passing conventions that are used by most of programming languages.
- ❖ Parameter passing by value
- ❖ **Drawback:** If called function changes the value of the parameter, the original value remains unchanged and ~~only copy on stack is changed.~~ **SYSTECH\_CSE\_CS11(AMP)\_SJM**
- ❖ Parameter passing by address or reference
- ❖ It provide mechanism to pass addresses of the parameters (instead of their values) to the called function.

# Parameter passing by Address OR Reference



The screenshot shows a terminal window with two panes. The left pane displays the assembly code for a program named 'xc.s'. The right pane shows a GDB session running on the same program.

```
Terminal Open student@student-OptiPlex-3020: ~
1 .section .data
2 num1:
3     .int 0xff
4 num2:
5     .int 0xee
6 .section .text
7 .globl _start
8 _start: nop
9
10    pushl $num1
11    pushl $num2
12    call exchange
13    addl $8,%esp
14
15    movl $1,%eax
16    movl $0,%ebx
17    int $0x80
18 exchange:
19 Pushl %ebp
20 Movl %esp,%ebp
21 movl 8(%ebp),%eax
22 movl 12(%ebp),%ebx
23 xchg (%eax),%ecx
24 xchg (%ebx),%ecx
25 xchg (%eax),%ecx
26 Movl %ebp,%esp
27 Popl %ebp
28 ret
29
30 _start () at xc.s:14
31     addl $8,%esp
32
33    movl $1,%eax
34    x/x &num1
35 0x80490a6: 0x000000ee
36    x/x &num2
37 0x80490aa: 0x000000ff
38
39
(gdb) break*_start+1
Breakpoint 1 at 0x8048075: file xc.s, line 10.
(gdb) run
Starting program: /home/student/xc
Breakpoint 1, _start () at xc.s:10
10      pushl $num1
11      pushl $num2
12      (gdb) s
13      call exchange
14      (gdb) s
15      Pushl %ebp
16      (gdb) s
17      Movl %esp,%ebp
18      (gdb) s
19      exchange () at xc.s:22
20      movl 8(%ebp),%eax
21      (gdb) s
22      movl 12(%ebp),%ebx
23      (gdb) s
24      xchg (%eax),%ecx
25      (gdb) s
26      xchg (%ebx),%ecx
27      (gdb) s
28      xchg (%eax),%ecx
29      (gdb) s
30      Movl %ebp,%esp
31      (gdb) s
32      Popl %ebp
33      (gdb) s
34      ret
35      (gdb) s
36      _start () at xc.s:14
37      addl $8,%esp
38      (gdb) s
39      movl $1,%eax
(gdb) x/x &num1
0x80490a6: 0x000000ee
(gdb) x/x &num2
0x80490aa: 0x000000ff
(gdb) 
```

The assembly code defines two integer variables, num1 and num2, both initialized to 0xff. It then calls a function named 'exchange'. Inside the 'exchange' function, it saves the current stack pointer to the base pointer register (%ebp), pushes the current stack pointer onto the stack, and then swaps the values of %eax and %ebx using the xchg instruction. Finally, it restores the stack pointer from %ebp and returns. The GDB session shows the breakpoints being set and the execution flow through the 'exchange' function, where the values of num1 and num2 are printed as 0xee and 0xff respectively.



A blue pen is shown writing the text "Thank you!" in cursive on a white surface. The pen is angled downwards from the top right towards the bottom left where the text is written.

SYBTECH CSE\_CSE211(MP)\_SJM

Thank you!

# Unit3

Data and Control transfer  
instructions

# Data Movement instructions

**movx src, dest**

# **xchg**

- It can exchange data values between two general purpose registers, or between a register and a memory location.
- Format of the instruction is as follows:  
**xchg operand1(src), operand2(dest)**
- Either operand1 or operand2 can be a general-purpose register or a memory location (but both cannot be a memory location).
- Two operands must be the same size. i.e. 8-, 16-, or 32-bit

# **xchg**

- None of the operand can be specified using immediate addressing.

# bswap

- The BSWAP instruction reverses the order of the bytes in a register.
- Bits 0 through 7 are swapped with bits 24 through 31, while bits 8 through 15 are swapped with bits 16 through 23.

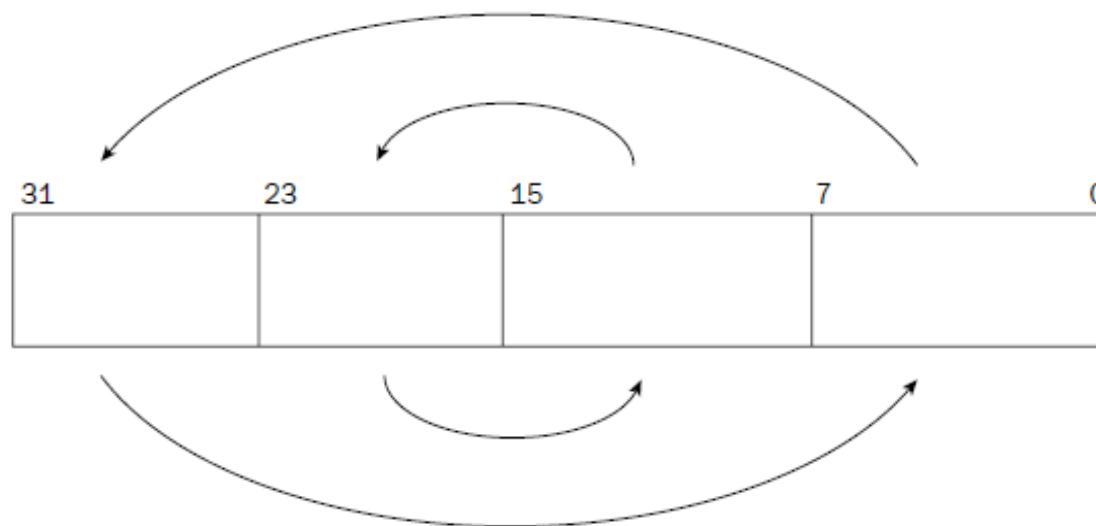


Figure 5-3

# bswap

- bits are not reversed; but rather, the individual bytes contained within the register are reversed.

```
movl $0x12345678, %ebx  
bswap %ebx
```

# **XADD**

- xadd is used to exchange the values between two registers, or a memory location and a register, add the values, and then store them in the destination location (either a register or a memory location).
- format of the XADD instruction is  
**xadd source, destination**

# **CMPXCHG**

- The CMPXCHG instruction compares the destination operand with the value in the EAX, AX, or AL registers.
- If the values are equal, the value of the source operand value is loaded into the destination operand.
- If the values are not equal, the destination operand value is loaded into the EAX, AX, or AL registers.
- The
- format of the CMPXCHG instruction is

**cmpxchg source, destination**

## example of the cmpxchg instruction

```
.section .data
    data:
        .int 10

.section .text
.globl _start
_start:
    nop
    movl $10, %eax
    movl $5, %ebx
    cmpxchg %ebx, data
    movl $1, %eax
    int $0x80
```

Result:

11 cmpxchg %ebx, data

(gdb) x/d &data

0x8049090 <data>: 10

(gdb) s

12 movl \$1, %eax

(gdb) x/d &data

0x8049090 <data>: 5

(gdb)

```
        movl $105, %ebx
        movl $235, %eax
        cmp %ebx, %eax          -----eax-ebx
        cmova %eax, %ebx
```

```
15  cmp %ebx, %eax
    (gdb) print $eax
$1 = 235
    (gdb) print $ebx
$2 = 105
(gdb) s
16 cmova %eax, %ebx
(gdb) s
(gdb) print $ebx
$3 = 235
(gdb)
```

# Data movement with conditional move instruction

## **cmovx/cc src , dest**

Instruction Pair	Description	EFLAGS Condition
CMOVA/CMOVNBE	Above/not below or equal	(CF or ZF) = 0
CMOVAE/CMOVNB	Above or equal/not below	CF=0
CMOVNC	Not carry	CF=0
CMOVB/CMOVNAE	Below/not above or equal	CF=1
CMOVC	Carry	CF=1
CMOVBE/CMOVNA	Below or equal/not above	(CF or ZF) = 1
CMOVE/CMOVZ	Equal/zero	ZF=1
CMOVNE/CMOVNZ	Not equal/not zero	ZF=0
CMOVP/CMOVPE	Parity/parity even	PF=1
CMOVNP/CMOVPO	Not parity/parity odd	PF=0

Fig unsigned conditional move instructions

# Data movement with conditional move instruction

Instruction Pair	Description	EFLAGS Condition
CMOVGE/CMOVNL	Greater or equal/not less	(SF,OF)=0
CMOVL/CMOVNGE	Less/not greater or equal	(SF,OF)=1
CMOVLE/CMOVNG	Less or equal/not greater	SF, OF or ZF =1
CMOVO	Overflow	OF=1
CMOVNO	Not overflow	OF=0
CMOVS	Sign (negative)	SF=1
CMOVNS	Not sign (non-negative)	SF=0
cmovg/cmovnle	Greater/not less or equal	ZF=0 and SF=OF

Fig Signed conditional move instructions

```
.section .data
    value:
        .int 5

.section .text
    .globl _start
    _start:
        nop
        movl $3, %ebx
        movl value, %ecx

        cmp %ebx, %ecx
        cmova %ecx, %ebx

        movl $7,%edx
        cmp %edx, %ecx
        cmova %ecx, %edx

        movl $0, %ebx
        movl $1, %eax
        int $0x80
```

```
.section .data
data:
.int 5
.section .text
.globl _start
_start:
nop
movl $0, %eax
movl $-1, %ebx

cmpl $2, data
cmova %ebx, %eax

cmpl $7 , data
cmovb %ebx, %eax

movl $0, %ebx
movl $1, %eax
int $0x80
```

- EBX: The integer file descriptor
  - ECX: The pointer (memory address) of the string to display
  - EDX: The size of the string to display
- 
- 0 (STDIN): The standard input for the terminal device (normally the keyboard)
  - 1 (STDOUT): The standard output for the terminal device (normally the terminal screen)
  - 2 (STDERR): The standard error output for the terminal device (normally the terminal screen)

# Machine stack:

- Stack is a data structure in which data items are added and removed in the LIFO order.

e.g.

esp=0x000011C0

eax=0x13579BDF

PUSH eax

# Machine stack:

- Machine stack can be used in several ways:
  - To keep the return address for the function calls
  - To temporarily save the contents of registers and recover them.
  - Local variables of high level language function
  - Execution control for recursive functions is implemented using stack.

```
.section .data
data:
.int 125
.section .text
.globl _start
_start:
Nop
movl $24420, %ecx
movw $350, %bx
movb $100, %eax
pushl %ecx
pushw %bx
pushl %eax
pushl data
pushl $data          #puts the 32-bit memory address referenced by the data label
popl %eax
popl %eax
popl %eax
popw %ax
popl %eax
movl $0, %ebx
movl $1, %eax
int $0x80
```

```
.section .data
    data:
        .int 125
.section .text
.globl _start
_start:
    nop
    movw $0x1122, %ax
    movl $0xaabbccdd, %edx
    movb $0x88, %bl
    pushw %ax
    pushl %edx
    inc %esp
    popl %ebx
    dec %esp
    popw %cx
    movl $0, %ebx
    movl $1, %eax
    int $0x80
```

# Control transfer instructions:

- Target address can be specified by --

## 1. Immediate constant

- Relative addressing

JMP swapbyte(label)

JMP 0x100 ----target address= a+0x100

Where a = address of next instruction immediately after jmp.

0x100= offset from the next instruction of jmp

## 2. Register addressing:

- Target address of control instruction may be specified using the register.

JMP %eax

### 3. Memory addressing:

- Address of target instruction is read from memory.  
JMP (**memvar**)
- 32 bit memory address is stored in **memvar**.
- During execution 32 bit value is read from memory location **memvar** and copied to **eip register** and transfer the control

- Unconditional control transfer instruction:  
JMP target
- Conditional control transfer instruction:  
JCC target

# Control transfer instructions:

Instruction	Description	EFLAGS
JA	Jump if above	CF=0 and ZF=0
JAE	Jump if above or equal	CF=0
JB	Jump if below	CF=1
JBE	Jump if below or equal	CF=1 or ZF=1
JC	Jump if carry	CF=1
JCXZ	Jump if CX register is 0	
JECXZ	Jump if ECX register is 0	
JE	Jump if equal	ZF=1
JG	Jump if greater	ZF=0 and SF=OF
JGE	Jump if greater or equal	SF=OF
JL	Jump if less	SF<>OF
JLE	Jump if less or equal	ZF=1 or SF<>OF
JNA	Jump if not above	CF=1 or ZF=1
JNAE	Jump if not above or equal	CF=1
JNB	Jump if not below	CF=0
JNBE	Jump if not below or equal	CF=0 and ZF=0
JNC	Jump if not carry	CF=0
JNE	Jump if not equal	ZF=0
JNG	Jump if not greater	ZF=1 or SF<>OF

## Control transfer instructions continued....:

JNGE	Jump if not greater or equal	SF<OF
JNL	Jump if not less	SF=OF
JNLE	Jump if not less or equal	ZF=0 and SF=OF
JNO	Jump if not overflow	OF=0
JNP	Jump if not parity	PF=0
JNS	Jump if not sign	SF=0
JNZ	Jump if not zero	ZF=0
JO	Jump if overflow	OF=1
JP	Jump if parity	PF=1
JPE	Jump if parity even	PF=1
JPO	Jump if parity odd	PF=0
JS	Jump if sign	SF=1
JZ	Jump if zero	ZF=1

# #Example of using the CMP and JGE instructions

```
.section .text
.globl _start
_start:
    nop
    movl $15, %eax
    movl $20, %ebx
    cmp %eax, %ebx
    jge next
    movl $1, %eax
    int $0x80
next:
    movl $10, %ebx
    movl $1, %eax
    int $0x80
```

# **Building loops in program:**

- Loop sections in the program that are iterated over again and again.
- Loops have 3 main parts-
  - Initialization code
  - Condition testing code
  - Body
- WAP to perform sum of all integers from 1 to n.

# *The loop instructions:*

- **Loop:**
  - Loops are another way of altering the instruction path within the program.
  - Loops enable us to code repetitive tasks with a single loop function.
  - The loop operations are performed repeatedly until a specific condition is met.

Instruction	Description
LOOP	Loop until the ECX register is zero
LOOPE/LOOPZ	Loop until either the ECX register is zero / ZF is SET
LOOPNE/LOOPNZ	Loop until either the ECX register is zero / ZF is NOT SET

# ***Using indexed memory locations***

values:

.int 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60

- When referencing data in the array, we must use an index system to determine which value we are accessing.
- The memory location is determined by the following:
  - A base address
  - An offset address to add to the base address
  - The size of the data element
  - An index to determine which data element to select

## The format of the expression is-

base\_address(offset\_address, index, size)

- To reference the value 20 from the values array shown, use----  
`movl $2, %edi`  
`movl values(, %edi, 4), %eax`
- This instruction loads the **second index value (3<sup>rd</sup> value)** of 4 bytes from the values label to the EAX register

- ***Using indirect addressing with registers:***
  - The memory location address of the data value by placing a **dollar sign (\$)** in front of the label in the instruction.

```
movl $values, %edi
```

- Is used to move the memory address the values label references to the EDI register.

```
movl %ebx, (%edi)
```

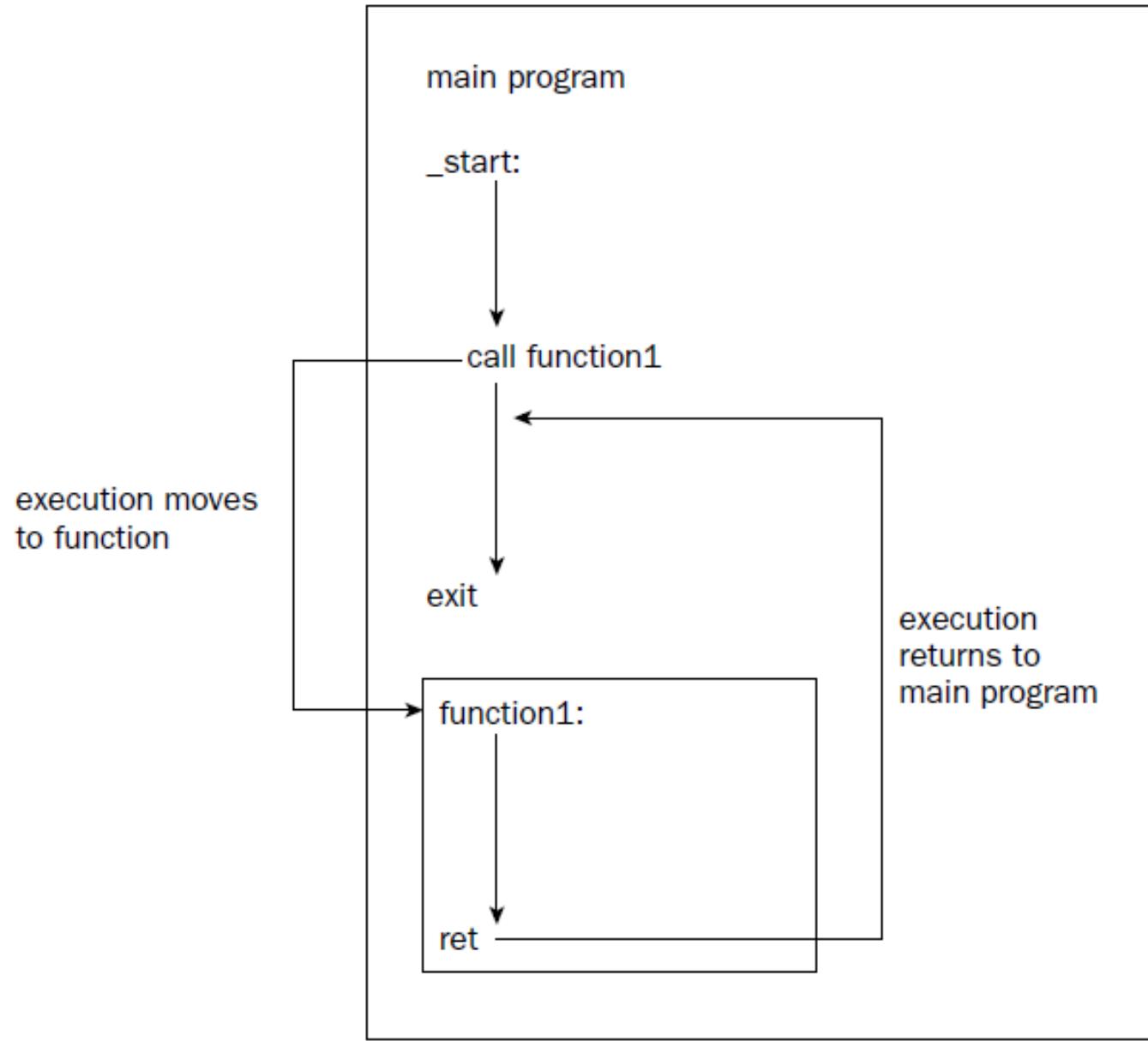
- moves the value in the EBX register to the memory location contained in the EDI register.

`movl %edx, 4(%edi)`

- This instruction places the value contained in the EDX register in the memory location 4 bytes **after** the location pointed to by the EDI register.

`movl %edx, -4(&edi)`

- This instruction places the value in the memory location 4 bytes before the location pointed to by the EDI register.



# Function call standard template:

```
function_label:
```

```
    pushl %ebp
```

```
    movl %esp, %ebp
```

```
.
```

```
.
```

```
.
```

```
.
```

```
.
```

```
    movl %ebp, %esp
```

```
    popl %ebp
```

```
    ret
```

#4 Assembly language program to Demonstrate Function calls and return.

```
.section .data
```

```
output:
```

```
.asciz "This is section %d\n"
```

```
.section .text
```

```
.globl _start
```

```
_start:
```

```
pushl $1
```

```
pushl $output
```

```
call printf
```

```
add $8, %esp
```

# should clear up stack

```
call overhere
```

```
pushl $3
```

```
pushl $output
```

```
call printf
```

```
add $8, %esp          # should clear up stack
pushl $0
call exit
overhere:
    pushl %ebp
    movl %esp, %ebp
    pushl $2
    pushl $output
    call printf
    add $8, %esp          # should clear up stack
    movl %ebp, %esp
    popl %ebp
    ret
```

```
as -gstabs -o test1.o test1.s  
ld -dynamic-linker /lib/ld-linux.so.2 -lc -o test1 test1.o  
. ./test1
```

Output:

This is section 1

This is section 2

This is section 3

- ***Passing function parameters on the stack***
  - Before a function call is made, the main program places the required input parameters for the function on the **top of the stack**.
  - When the CALL instruction is executed, it places the **return address from the calling program onto the top of the stack** as well, so the function knows where to return.

# Passing parameters:

- Passing parameters through registers:

```
#function that converts an 8 bit signed no to 32 bit signed number
```

Convert:

```
    movsx %al,%ebx  
    ret
```

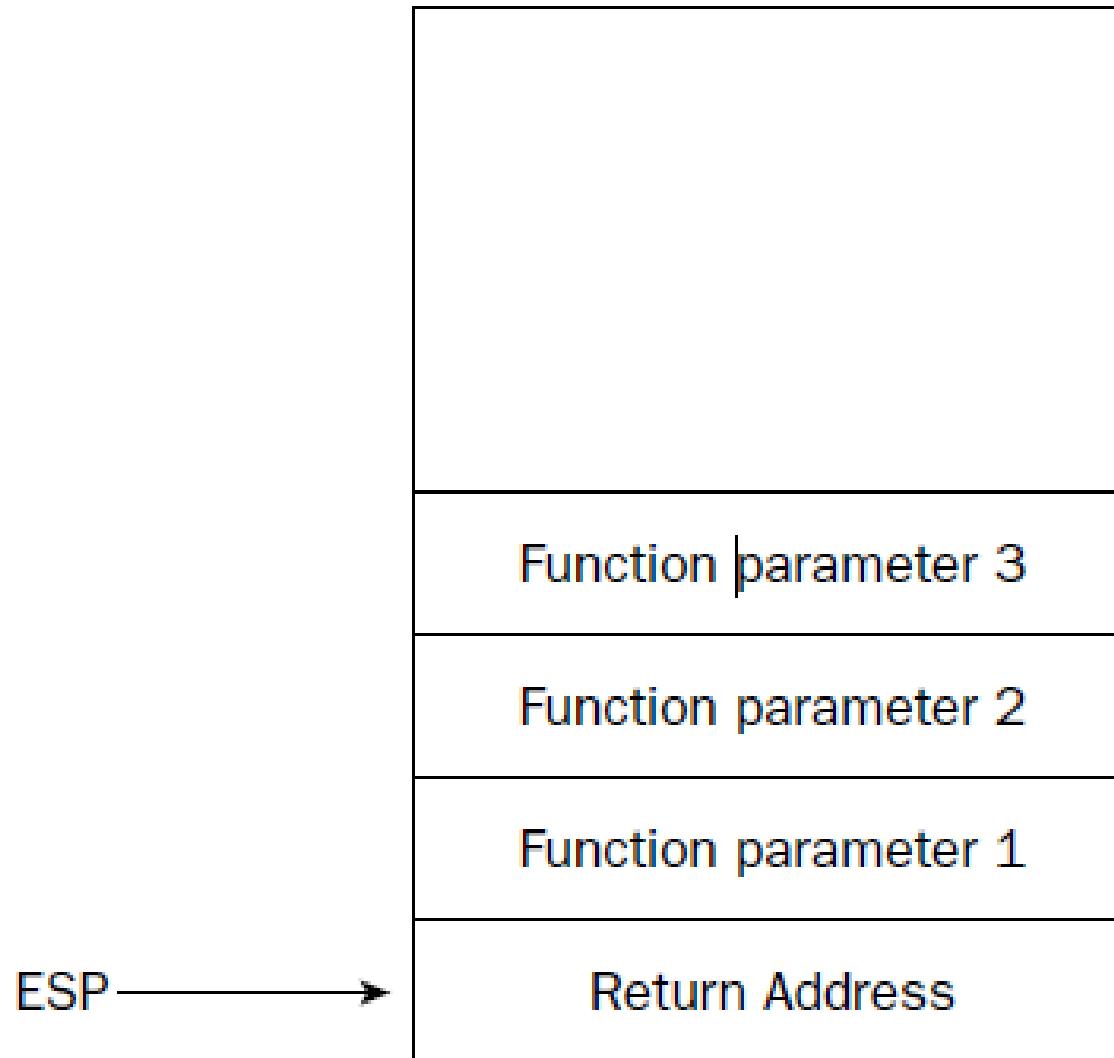
```
#use the function with parameter passing using registers
```

```
    mov memvar8 ,%al  
    call convert  
    mov %ebx, memvar32
```

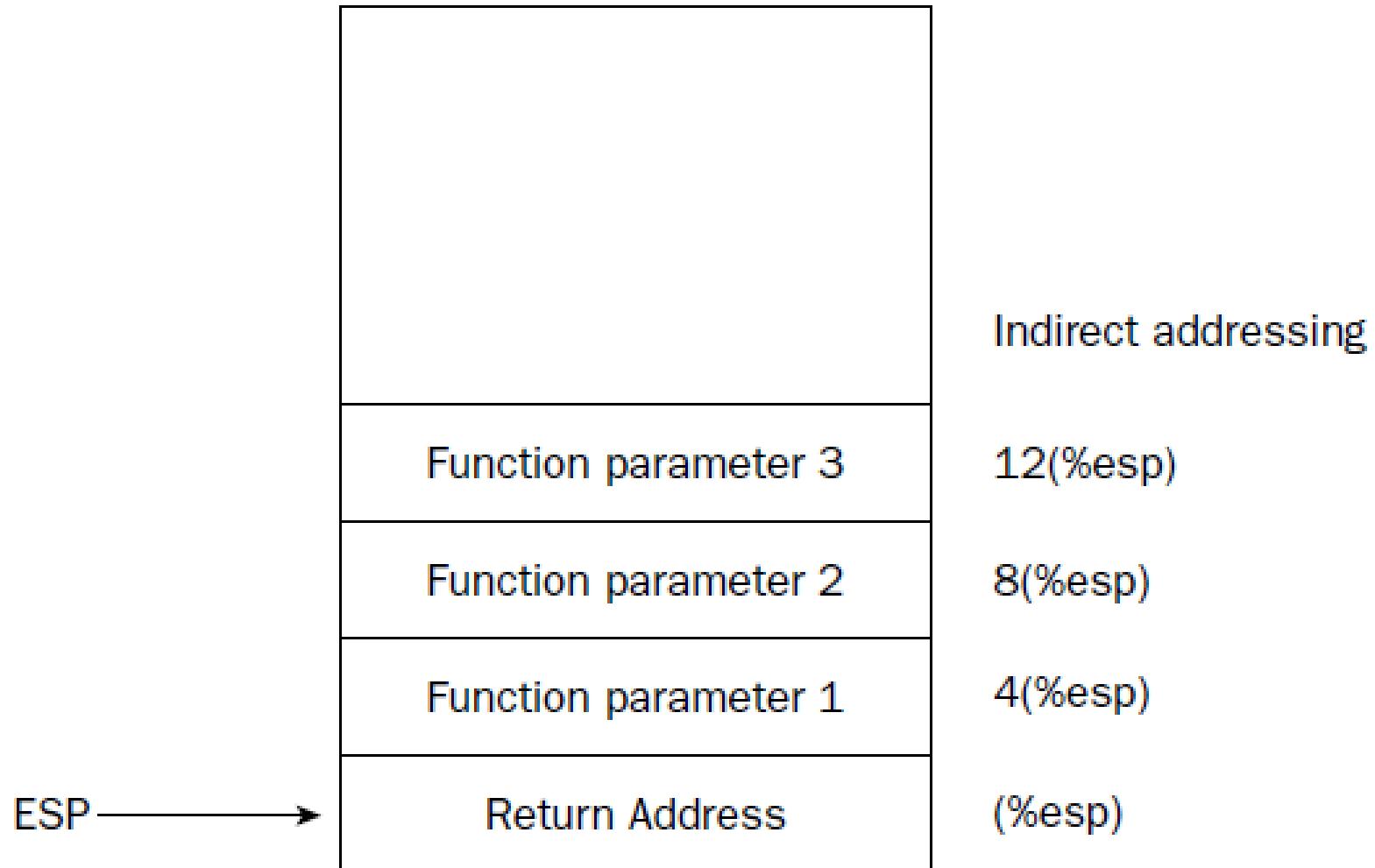
# Parameter Passing conventions:

- Parameters passing by value:
  - Parameters passing by address or reference:
- 
- Parameters are passed to the called function by copying their values to registers, memory or stack.
  - If the called function changes the value of the parameter , the original value remains unchanged and only the copy is changed.
  - In such a case parameters are said to be **passed by value**

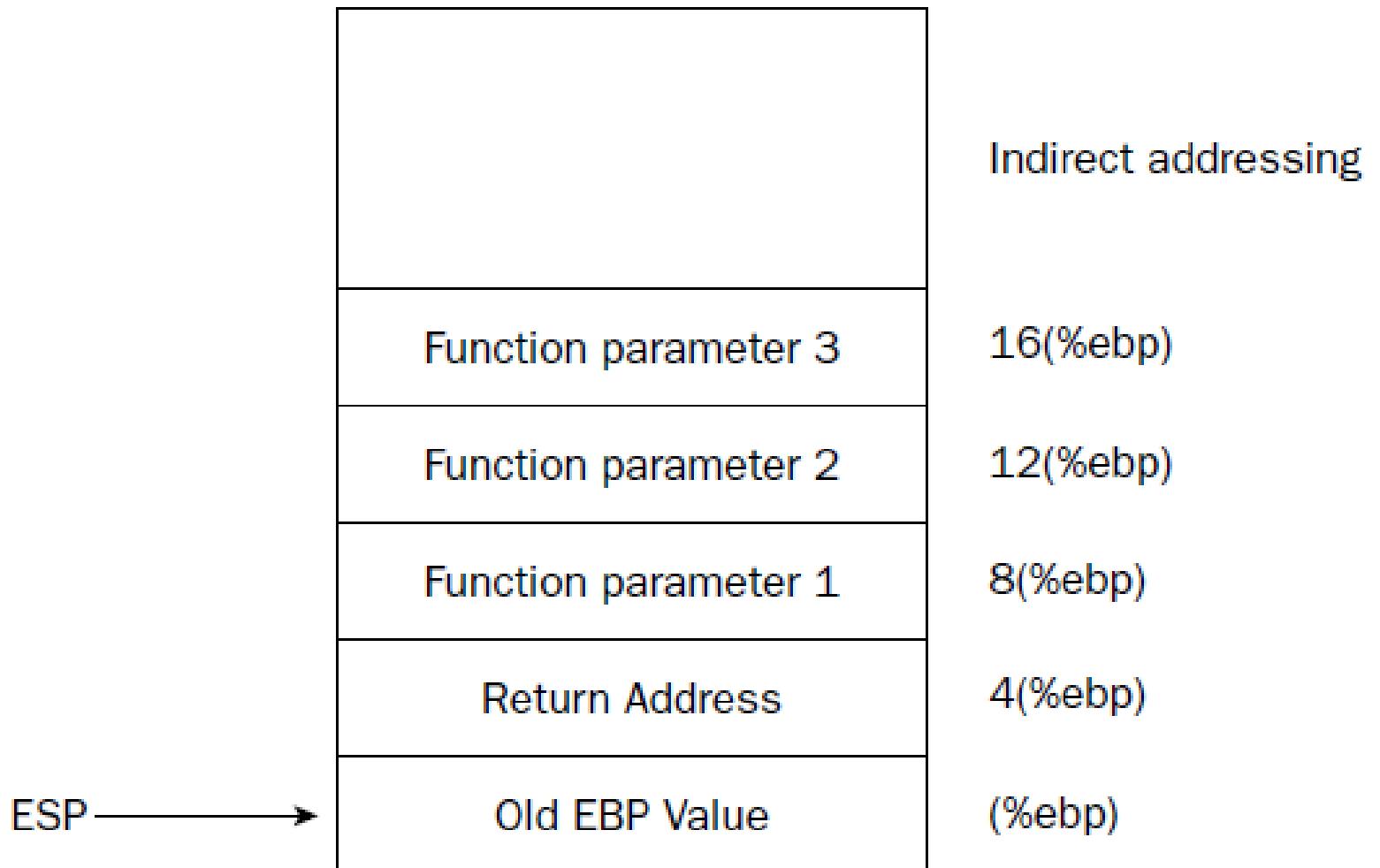
## Program Stack



## Program Stack



## Program Stack



EBP register is often used as a base pointer to the stack

```
.  
. .  
Pushl NumB  
Pushl NumA  
Call exchangeNums  
Addl $8, %esp
```

### exchangeNums:

```
    mov 4(%esp), %eax  
    xchg %eax, 8(%esp)  
    mov %eax, 4(%esp)  
    ret
```

	⋮
esp-4	
esp	0x1016
esp+4	0x9805
esp+8	
	⋮
	⋮
NumA	0x1016
NumB	0x9805
	⋮

(a) After pushing NumA and NumB on stack

	⋮
esp	Ret Addr
esp+4	0x1016
esp+8	0x9805
esp+12	
	⋮
	⋮
NumA	0x1016
NumB	0x9805
	⋮

(b) After making a call to function exchangeNums

	⋮
esp	Ret Addr
esp+4	0x1016
esp+8	0x9805
esp+12	
	⋮
	⋮
NumA	0x1016
NumB	0x9805
	⋮

(c) After execution of `mov -4(%esp), %eax`

	⋮
esp	Ret Addr
esp+4	0x1016
esp+8	0x1016
esp+12	
	⋮
	⋮
NumA	0x1016
NumB	0x9805
	⋮

(d) After execution of `xchq %eax, -8(%esp)`

Register eax  
0x00009805

	⋮
esp	Ret Addr
esp+4	0x9805
esp+8	0x1016
esp+12	
	⋮
	⋮
NumA	0x1016
NumB	0x9805
	⋮

(e) After execution of `mov %eax, -4(%esp)`

Register eax  
0x00001016

**Figure :** Memory contents while executing exchangeNums.

# Parameter passing by address or reference:

```
.  
Pushl $NumB  
Pushl $NumA  
Call exchangeNums  
Addl $8, %esp
```

```
.  
. 
```

## exchangeNums:

```
    mov 8(%esp), %eax    #addresses of two 32 bit nos as parameters  
    mov 12(%esp), %ebx  
    xchg (%eax), %ecx  
    xchg (%ebx), %ecx  
    xchg (%eax), %ecx  
    ret
```

#program for exchanging 2, 32 bit numbers through parameter passing by address /reference

.section .data

numA:

    .int 0xAABBCCDD

numB:

    .int 0x11223344

output:

    .asciz "The exchanged values are numA=%d ,numB=%d\n"

.section .text.

globl \_start

\_start:

    nop

    pushl \$numB

    pushl \$numA

    call exchnum

    addl \$8,%esp

    movl \$0, %ebx

    movl \$1, %eax

    int \$0x80

exchnum:

```
pushl %ebp
mov %esp, %ebp
movl 8(%esp),%eax
movl 12(%esp),%ebx
xchg (%eax),%ecx
xchg (%ebx),%ecx
xchg (%eax),%ecx
pushl (%ebx)
pushl (%eax)
pushl $output
call printf
add $12,%esp
movl %ebp,%esp
pop %ebp
ret
```

## Interfacing with GNU C compilers;

- ALP can be interfaced to C functions
- Programs written in AL functions can call C functions and vice versa.
- Functions maintain a frame on stack that contains parameters, return address, reference to older frame and local operands.
- Parameters and return addresses are put on stack by the **caller** (by using push and call instructions)

# Interfacing with GNU C compilers;

- reference to older frame and space for local variables are maintained by the called function.
- Use frame pointer instead of stack pointer
- ebp register as the frame pointer
- In ALP, the **function prologue** is a few lines of code at the beginning of a function, which prepare the stack and registers for use within the function.
- **function epilogue** appears at the end of the function, and restores the stack and registers to the state they were in before the function was called.

# Interfacing with GNU C compilers;

- Prologue code for functions:

```
Pushl %ebp  
movl %esp, %ebp
```

.

.

- ebp was set to stack pointer before creating space for local variables.
- Old frame pointer can be restored by popping off the stack.

# Interfacing with GNU C compilers;

- Epilogue code for functions:

```
    movl %ebp, %esp  
    Popl %ebp  
    ret
```

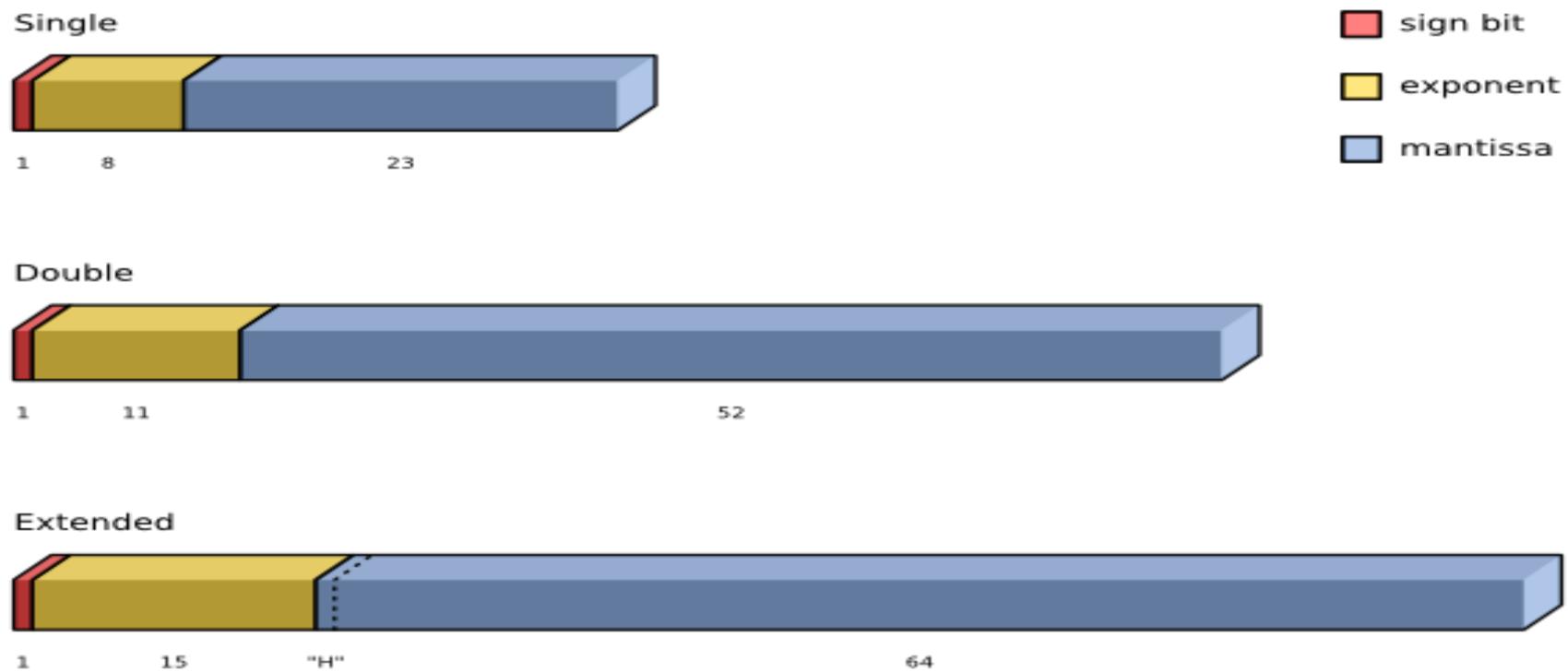
Thank You

# Unit5

Floating point and SIMD instructions

# Floating point Representation

- There are three kinds of floating point numbers in IA32 architectures.
- IEEE754 Single precision floating point format
- IEEE754 Double precision floating point format
- Double extended precision floating point number



# Floating point Representation

- IA32 architectures support IEEE754 number representation schemes of floating point numbers. There are three kinds of floating point numbers in IA32 architecture.
  - **IEEE754 Single precision floating point format**
  - These are 32 bit wide numbers and have three parts-a sign, a mantissa and an exponent.
  - Sign is 1 bit wide and represents the sign of the number. Mantissa is 23 bit wide and represents the fractional part of the floating point number in binary. A leading one bit integer is not stored and is assumed as 1 in the normalized numbers.
  - Exponents are stored in excess-127 representation.
  - **IEEE754 Double precision floating point format**
  - These are 64 bit wide number with sign ,mantissa and exponent in excess 1023 representation.
  - **Double extended precision floating point number**
  - Double extended precision floating point numbers are 80 bit wide.
  - This number format is used to perform computations internally in IA32 processors when instructions in x87 floating point instruction set are used.

## Floating point conversion in single precision

- Conversion of decimal number to IEEE single precision floating point format
- 24.75

Sign bit(S) =1 (for negative number)

$$24.75 = 11000.11 \times 2^0$$

$$1.100011 \times 2^4$$

— Exponent is excess-127 in single precision format

bias exponent = actual exponent + bias value

$$= 127 + 4 = 131$$

Mantissa (M) = (1.  
100011)



$$-24.75 = 0xC1C60000$$

$$+7.5 = ?$$



# First Floating point program

```
.section .data
value1:
.float -24.75
value2:
.double -24.75
.section .bss
.lcomm data ,4
.lcomm data1, 8
.section .text
.globl _start
_start : nop
finit
flds value1
fldl value2
fst data
fstl data1
movl $1,%eax
movl $0,%ebx
int $0x80
```

## Single and double precision Number in IEEE

```
Terminal student@student-OptiPlex-3020: ~
student@student-OptiPlex-3020:~$ as -gstabs -o float1.o float1.s
student@student-OptiPlex-3020:~$ ld -o float1 float1.o
student@student-OptiPlex-3020:~$ gdb -q float1
Reading symbols from float1...done.
(gdb) break*_start+1
Breakpoint 1 at 0x8048075: file float1.s, line 11.
(gdb) run
Starting program: /home/student/float1

Breakpoint 1, _start () at float1.s:11
11      flds value1
(gdb) x/fx &value1
0x8049093:    0xc1c60000
(gdb) s
12      fild value2
(gdb) x/gfx &value2
0x8049097:    0xc038c00000000000
(gdb) s
13      fstl data
(gdb) s
14      movl $1,%eax
(gdb) x/gfx &data
0x80490a0 <data>:    0xc038c00000000000
(gdb)
```

## Double extended precision Number in IEEE

Terminal

↑ En 9:48 PM



```
student@student-OptiPlex-3020: ~
eax          0x0      0
ecx          0x0      0
edx          0x0      0
ebx          0x0      0
esp          0xbfffff110    0xbfffff110
ebp          0x0      0x0
esi          0x0      0
edi          0x0      0
eip          0x804807b    0x804807b <_start+7>
eflags        0x202    [ IF ]
cs           0x73     115
ss           0x7b     123
ds           0x7b     123
es           0x7b     123
fs           0x0      0
gs           0x0      0
st0          -24.75   ( raw 0xc003c6000000000000000000 )
st1          0         ( raw 0x000000000000000000000000 )
st2          0         ( raw 0x000000000000000000000000 )
st3          0         ( raw 0x000000000000000000000000 )
st4          0         ( raw 0x000000000000000000000000 )
st5          0         ( raw 0x000000000000000000000000 )
st6          0         ( raw 0x000000000000000000000000 )
---Type <return> to continue, or q <return> to quit---
```

Plain Text ▾ Tab Width: 8 ▾

Ln 10, Col 14 ▾

INS

# Architecture of floating point processor

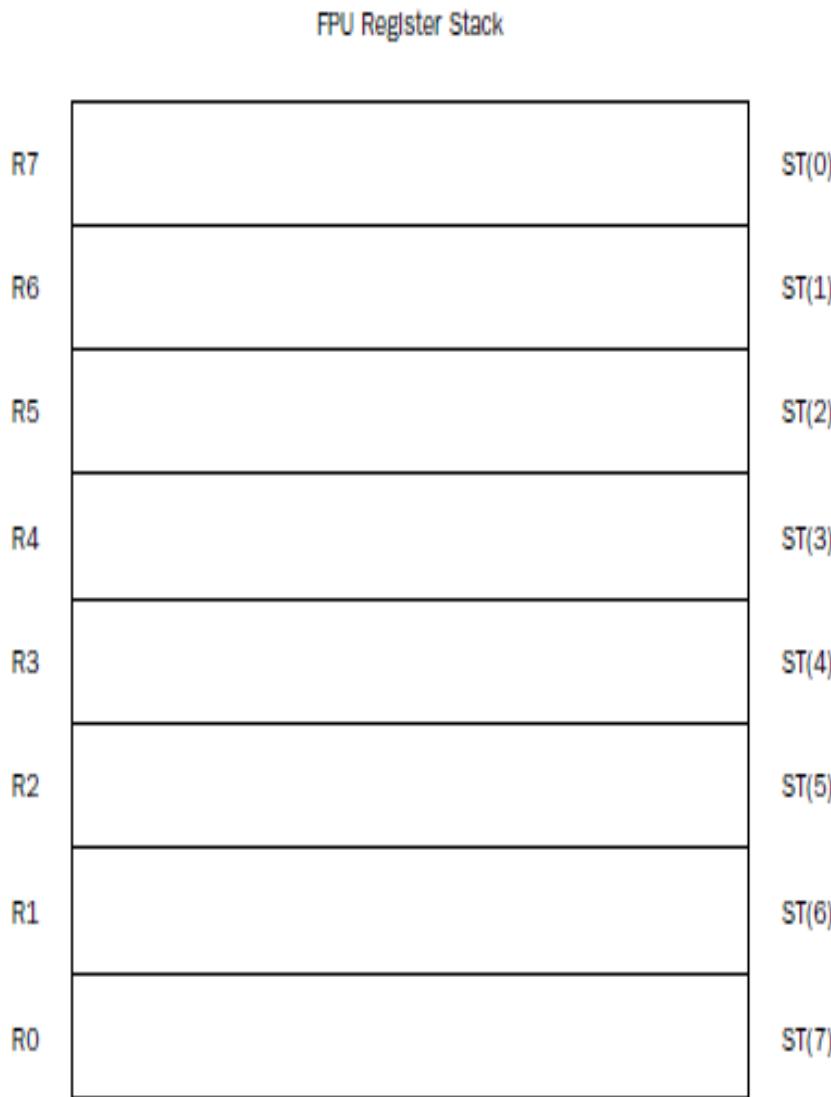


- **X87 general purpose Data registers in IA32 processor**

It include 8 general purpose data registers, each 80 bit wide and capable of storing a real number in double extended precision format.

- As data is loaded into the FPU stack, the stack top moves downward in the registers.
- When eight values have been loaded into the stack, all eight FPU data registers have been utilized.
- If a ninth value is loaded into the stack, the stack pointer wraps around to the first register and replaces the value in that register with the new value

# Architecture of floating point processor



- When an integer ,BCD, single and double precision operand is loaded into one of these data registers, the operand is implicitly converted to double extended precision format.
- In addition to data registers, x87 also includes three 16 bit registers which are used in controlling the computations.
- These registers are the following
  - X87 control register
  - X87 status register
  - X87 tag register

# Top of stack

Stack register	Data register when TOP=5	Data register when TOP=2
St(0)	R5	R2
St(1)	R4	R1
St(2)	R3	R0
St(3)	R2	R7
St(4)	R1	R6
St(5)	R0	R5
St(6)	R7	R4
St(7)	R6	R3

The register stack used registers in a circular buffer.

- 1) When TOP=5, then st (0) refers to x87 data register R5.

# Floating point exceptions

- Invalid operation exception (IE)
  - It is raised when operands of the instruction contain invalid values such as NaN.
  - Example: division of 0 by 0 should lead to NaN
- Divide by zero exception (ZE)
  - Occurs when divisor contains a zero and dividend is a finite number other than 0
- De-normalized operand exception (DE)
  - It is raised when one of the operands of an instruction is in denormalized form.
  - De-normalized numbers can be used to represent numbers with magnitudes too small to normalize (i.e. below  $1.0 \times 2^{-126}$ )
- Numeric overflow exception (OE)
  - Occurred when rounding result would not fit into destination operand.
  - *When a double precision floating point number is to be stored in single precision floating point format, a numeric overflow is set to have occurred.*
- Numeric underflow exception (UE)
  - Occur when the instruction generate result whose magnitude is less than the smallest possible normalized number.
- Precision (Inexact result) exception (PE)
  - Division of (1.0/12.0) results in recurring infinite sequence of bits.
  - This number cannot be stored in any format without loss of precision.

# X87 control register

During computation ,several floating point error conditions such as divide by zero occur

In case of such errors , x87 FPU can be programmed to raise floating point exception

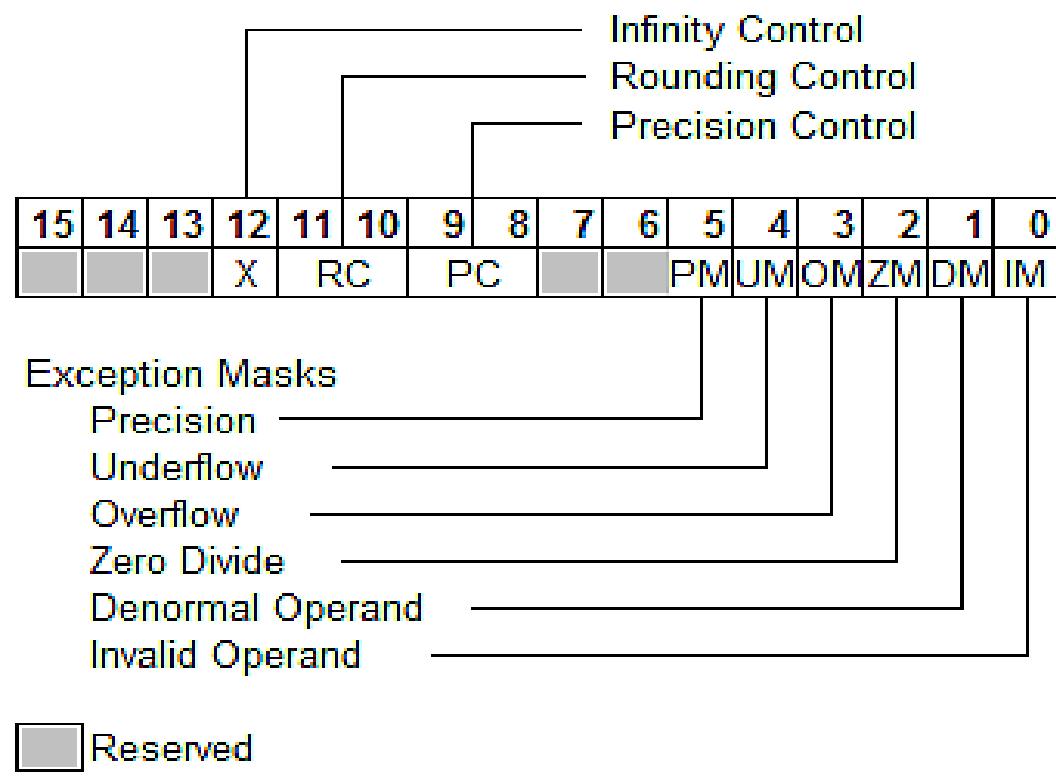
Bit 0 to bit 5

0 indicate : unmask exception

1 indicate: Mask exception

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
			X	RC	PC			PM	UM	OM	ZM	DM	IM		

# X87 control register



Used to handle the precision and rounding of operands  
Infinity bit always set to 0  
Precision control  
00- single precision  
01 unused  
10-Double precision  
11-Double extended precision  
Rounding control  
00-Round to nearest  
01 Round down  
10-Round up  
11 Round towards zero(Truncate)  
Exception Mask  
0-Unmask  
1-Mask

# Rounding Controls in IA 32 architectures

- +1.000100100011010011101 **0011**
  1. Rounding towards 0 +1.000100100011010011101
  2. Rounding towards  $+\infty$  (Rounding up) +1.00010010001101001110
- Rounding towards plus infinity control used 0011 would be removed but a 1 will be added to the LSB of remanding bits because the removed bit pattern is other than a zero.
- +1.000100100011010011101 **1011**
  3. Rounding towards int +1.00010010001101001110
  4. Rounding towards  $-\infty$  (Rounding down) +1.000100100011010011101

Note:1) Rounding towards 0 identical as rounding towards  $-\infty$

Note:1) Rounding towards int identical as rounding towards  $+\infty$

# X87 Status register

1) **TOP**-it is 3 bit field contain index to register assumed to be at the top of the stack.

2) **Four conditional code flags C0 to C3**

A floating point number can be compared in different ways with other floating point number.

The result of comparison is stored in either eflags or in floating point conditional code c0-c3

3) **SF**-The stack fault bit in status register indicates errors due to stack overflow or underflow conditions

Example: when one data on stack and addition operation is performed which takes two data from the stack, a stack fault occurs.

Stack overflow; when all 8 registers are in use and an attempt is made to load an data , stack overflow occurs.

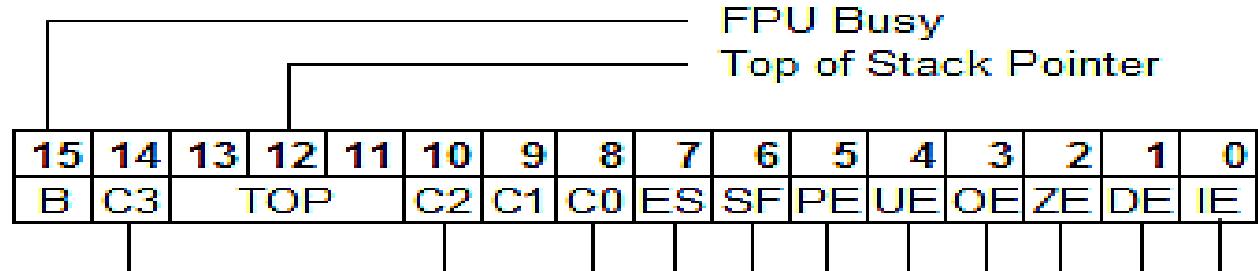
## ES- Error status

1- Error occurred 0- No error

## B bit

FPU Busy bit B=0 FPU free

FPU Busy bit B=1 FPU busy.



# X87 Tag register



Each data register of x87 FPU has an associated 2 bit tag contained in a tag register.

The code in the tag register indicates the type of the value of the corresponding data register.

Tag is associated with data register

00- R has a valid value

01-R has a zero

10-R has a special value (Nan, +/- infinity, renormalized number)

11-R is empty.

## Program to Illustrate Status Register, Tag register and Control register.

```
1 .section .bss
2
3 .lcomm status,2
4 .section .text
5
6 .globl _start
7 _start : nop
8
9
10    fstsw status
11
12    movl $1,%eax
13    movl $0,%ebx
14    int $0x80
15
16
```

The screenshot shows a terminal window titled "student@student-OptiPlex-3020: ~". The window displays assembly code and a GDB session. The assembly code is as follows:

```
0x00000000000000000000000000000000, 0x000000
ymm2          {v8_float = {0x0, 0x0, 0x0, 0x0,
v4_double = {0x0, 0x0, 0x0, 0x0}, v32_int8 =
v16_int16 = {0x0 <repeats 16 times>}, v8_int3
---Type <return> to continue, or q <return> to
[1]+ Stopped                      gdb -q fs
student@student-OptiPlex-3020:~$ gdb -q fs
Reading symbols from fs...done.
(gdb) break*_start+1
Breakpoint 1 at 0x8048075: file fs.s, line 11.
(gdb) run
Starting program: /home/student/fs

Breakpoint 1, _start () at fs.s:11
11          fstsw status
(gdb) s
13          movl $1,%eax
(gdb) print/x $ftag
$1 = 0xffff
(gdb) print/x $fctrl
$2 = 0x37f
(gdb) print/x $fstat
$3 = 0x0
(gdb) █
```

## **Fadd**

Add two floating point number available in stack registers

Example fadd %st(1),%st\*0)

Here %st(0) is added with %st(1) and result stored in **destination** stack %st(0)

## **Fiadd**

It is used to add add an integer stored in memory location to the floating point number on the top of stack.

## **Faddr**

Add two floating point number available in stack registers

Example faddr %st(1),%st(0)

Here %st(0) is added with %st(1) and result stored in **source** stack register %st(0)

## **Faddp**

Add two floating point number available in stack registers and pop the top of stack.

Example faddp %st(1),%st(0)

here %st(0) is added with %st(1) and remove stack registers and put result in top of stack.

$$((43.65 / 22) + (76.34 * 3.1)) / ((12.43 * 6) - (140.2 / 94.21))$$

```
.section .data
value1:
    .float 43.65
value2:
    .int 22
value3:
    .float 76.34
value4:
    .float 3.1
value5:
    .float 12.43
value6:
    .int 6
value7:
    .float 140.2
value8:
    .float 94.21
.section .text
```

```
.globl _start
_start:
nop
finit
flds value1
fidiv value2
flds value3
flds value4
fmul %st(1), %st(0)
fadd %st(2), %st(0)
flds value5
fimul value6
flds value7
flds value8
fdivrp
fsubr %st(1), %st(0)
fdivr %st(2), %st(0)
movl $1,%eax
movl $0,%ebx
int $0x80
```

1 43.65	2 1.98409	3 76.34 1.98409	4 3.1 76.34 1.98409	5 236.654 76.34 1.98409	6 238.63809
7 12.43	8 74.58 238.63809	9 140.2 74.58 238.63809	10 94.21 140.2 74.58 238.63809	11 1.48816 74.58 238.63809	12 73.09184 74.58 238.63809
13 3.264907					

$$((43.65 / 22) + (76.34 * 3.1)) / ((12.43 * 6) - (140.2 / 94.21))$$

1

**2**  
**1.98409**

3
76.34
1.98409

4
3.1
76.34
1.98409

5
236.654
76.34
1.98409

**6**  
238.63809

7
12.43
238.63809

8
74.58
238.63809

9
140.2
74.58
238.63809

10
94.21
140.2
74.58
238.63809

11
1.48816
74.58
238.63809

12
73.09184
74.58
238.63809

**13**  
**3.264907**

# Advanced Floating-Point Maths

- .section .data
- value1:
- .float 395.21
- value2:
- .float -9145.290
- value3:
- .float 64.0
- .section .text
- .globl \_start
- \_start:
- nop
- finit
- flds value1
- fchs
- flds value2
- fabs
- flds value3
- fsqrt
- movl \$1, %eax
- movl \$0, %ebx
- int \$0x80

**Fchs** this instruction is used to change the sign of the input argument on the top of stack i.e st(0)

**Fabs** this instruction is used to compute the absolute value of register st(0)

**Fsqrt** this instruction is used to compute square root of a number on top of stack st(0) and returns the result in register st(0) overwriting the value stored previously

$$x_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$$

$$x_1^2 - 4x_1 + 3$$

## Roots of a quadratic equation

```
.section .text
.globl _start
_start : nop

flds a
fimul val1
flds c
fmul %st(1),%st(0)
flds b
fmul %st(0),%st(0)
fsubp %st(1),%st(0)
fsqrt
flds b
fchs
fadd %st(1),%st(0)
flds b
fchs
fsub %st(2),%st(0)
flds a
fimul val2
fstps val5
fdivr %st(2),%st(0)
fstps val3
flds val5
(gdb) x/f &val3
0x80490f8 <val3>:      3
(gdb) s
44          fdivr %st(2),%st(0)
(gdb) s
45          fstps val4
(gdb) s
47          movl $1,%eax
(gdb) x/f &val4
0x80490fc <val4>:      1
(gdb) 
```

## constant loading x87 FPU instructions

in the x87 instruction set, there are certain instructions that can load commonly used constants on the stack top. These instructions are the following

**Fld1**  $\text{st}(0)=1.0$

**Fldz**  $\text{st}(0)=0.0$

**Fldpi**  $\text{st}(0)=3.14$

**Fldl2e**  $\log_2 e$

**Fldln2**  $\log_e 2$

**Fldl2t**  $\log_2 10$

**Fldlg2**  $\log_{10} 2$

## Trigonometric x87 FPU instructions

In the x87 FPU instruction set, following instructions are available and can compute trigonometric functions for arguments stored on the stack top

- Fsin It compute sin of angle in radians provided in st(0)
- Fcos It compute cosine of angle in radians provided in st(0)
- Fsincos It compute and leaves two values in the register stack.
- Fptan It removes the angle provided on top of the register stack and computers its tangent.

$$\text{radians} = (\text{degrees} * \pi) / 180$$

- .section .data
  - degree1:
  - .float 90.0
  - val180:
    - .int 180
- .section .bss
  - .lcomm radian1, 4
  - .lcomm result1, 4
  - .lcomm result2, 4
- .section .text
  - .globl \_start
  - \_start:
    - nop
    - finit
    - flds degree1
    - fidivs val180
    - fldpi
    - fmul %st(1), %st(0)
    - fstts radian1
    - fsin
    - fstts result1
    - flds radian1
    - fcos
    - fstts result2
    - movl \$1, %eax
    - movl \$0, %ebx
    - int \$0x80

## logarithmic x87 FPU instructions

there are two instructions in x87 FPU instruction set to compute logarithm

Fyl2x

it compute  $y \log_2 x$  with y and x being in register st(1) and st(0) respectively.

fyl2xpl

it compute  $y \log_2 (x + 1)$  with y and x being in register st(1) and st(0) respectively

Both the instructions take two implied arguments x and y on the register stack in registers st(0) and st(1) respectively.

After successful execution of this instruction ,both operands are removed and result is pushed on the register stack.

$$\log_b X = (1/\log_2 b) * \log_2 X$$

- .section .data
- value:
- .float 12.0
- base:
- .float 10.0
- .section .bss
- .lcomm result, 4
- .section .text
- .globl \_start
- \_start:
- nop
- finit
- fld1
- flds base
- fyl2x
- fld1
- fdivp
- flds value
- fyl2x
- fst<sub>s</sub> result
- movl \$1, %eax
- movl \$0, %ebx
- int \$0x80

# SIMD Technology

- **Introduction**
- **MMX instruction set**
- Intel introduced multimedia extension (MMX) instruction set with Pentium processors.
- Instruction operate simultaneously on **multiple data values**.
- Application include image processing, voice and data communication.
- **XMM instruction set**
- In Pentium III, intel introduced **streaming SIMD extension (SSE)** instruction set.
- While MMX instructions operate on **integer** data, SSE instructions operate on **floating** point data.
- The SSE instruction set is targeted at applications that operate on large arrays of floating point numbers such as 3D graphics, video encoding and decoding etc.

# SIMD Environment

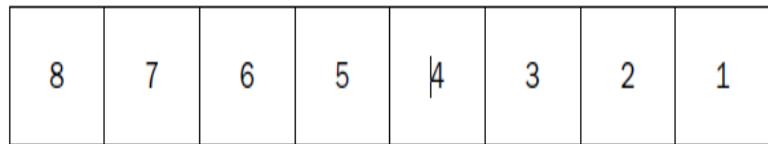
- SIMD instruction set provide additional registers called MMX registers and XMM registers.
- MMX registers includes 8, **64** bit registers named mm0-mm7.
- XMM registers includes 8, **128** bit registers named xmm0-xmm7.
- SIMD architecture uses packed data type
- It Perform arithmetic operations on a group of multiple integers simultaneously

# SIMD Environment

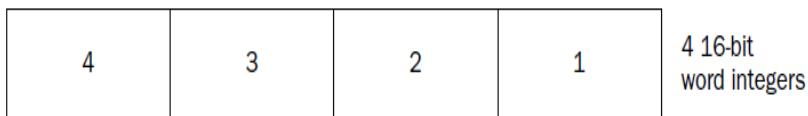
- MMX registers are aliased to x87 floating point data registers R0-R7.
- Data register in x87 are 80 bit wide, out of which only 64 bits are used by instructions in the SIMD instruction set.
- It is recommended to use finit instruction to initialize x87 FPU at the time of switching from SIMD to x87 environment.
- XMM registers support integer and floating point data types.

# SIMD Environment

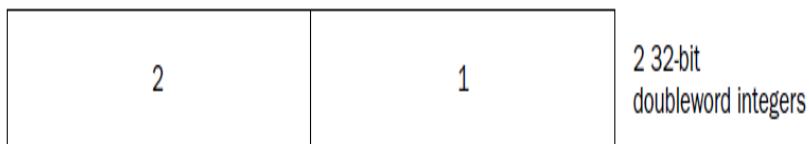
SIMD instructions perform the same operations on multiple data items as per the data types.



8 8-bit  
byte integers



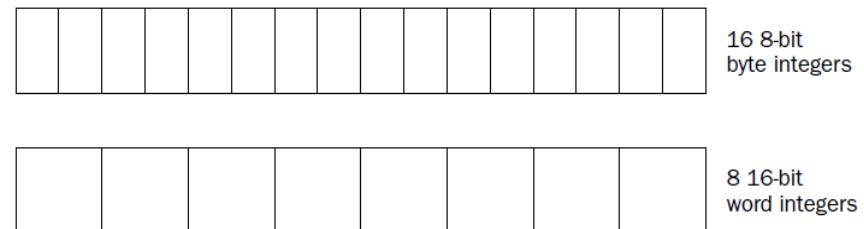
4 16-bit  
word integers



2 32-bit  
doubleword integers



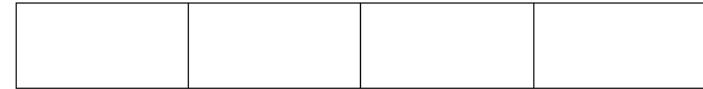
MMX data type for  
integer



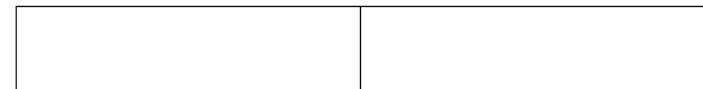
16 8-bit  
byte integers



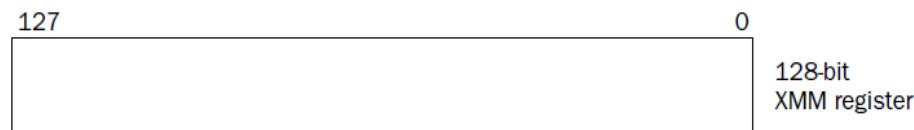
8 16-bit  
word integers



4 32-bit  
doubleword integers



2 64-bit  
quadword integers



XMM data type for  
integer

# SIMD Environment



4 packed single precision floating point number

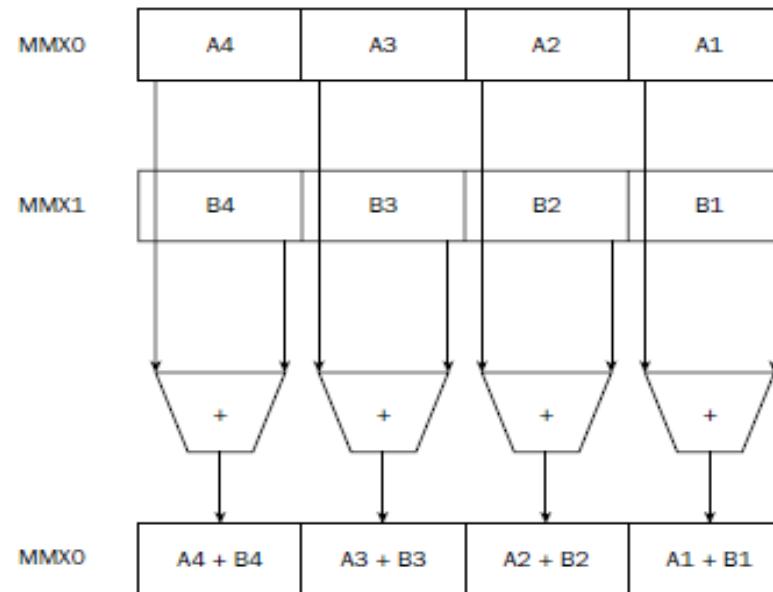


2 packed double precision floating point number

XMM data type for floating point

# SIMD Environment

- An addition operation that operates on packed 16 bit words. It takes two, 4 packed words, add them and provide result as 4 packed words.



# Loading and retrieving packed integer values

```
.section .data
packedvalue1:
.byte 10, 20, -30, 40, 50, 60, -70, 80
packedvalue2:
.short 10, 20, 30, 40
packedvalue3:
.int 10, 20
.section .text
.globl _start
_start:
    movq packedvalue1, %mm0
    movq packedvalue2, %mm1
    movq packedvalue3, %mm2
```

```
Movl $1,%eax
Movl $0,%ebx
Int $0x80
```

```
(gdb) print $mm0
(gdb) print $mm1
(gdb) print $mm2
```

# MMX addition and subtraction instructions

- With normal addition and subtraction with general-purpose registers, if an overflow condition exists from the operation, the EFLAGS register is set to indicate the overflow condition
- when using MMX addition or subtraction, you must decide ahead of time what the processor should do in case of overflow conditions within the operation.
- You can choose from three overflow methods for performing the mathematical operations:
  1. Wraparound arithmetic
  2. Signed saturation arithmetic
  3. Unsigned saturation arithmetic

# Examples of SIMD byte operations with saturations

A	B	Operation	Wrap around	Signed saturation	Unsigned saturation
0xA3	0xC2	A+B	0x65	0x80	0xFF
0x57	0xB2	A+B	0x09	0x09	0xFF
0x78	0x40	A+B	0xB8	0x7F	0xB8
0x40	0x72	A-B	0xCE	0xCE	0x00

**Wrap around-** Whenever the result is larger than 0xFF, Extra bit is dropped. Result is treated as module 256.

**Signed saturation-** when -93 (0xA3) and -62(0xC2) in decimal added. The summation of two numbers will be -155 which is below the minimum possible (-128) So result is treated as (-128) in decimal and in hex (0x80)

**Unsigned saturation-** When 163 and 194 in decimal added the result is 357, which again cannot represent in 8 bit unsigned number format. So the result is 0xFF

# MMX addition

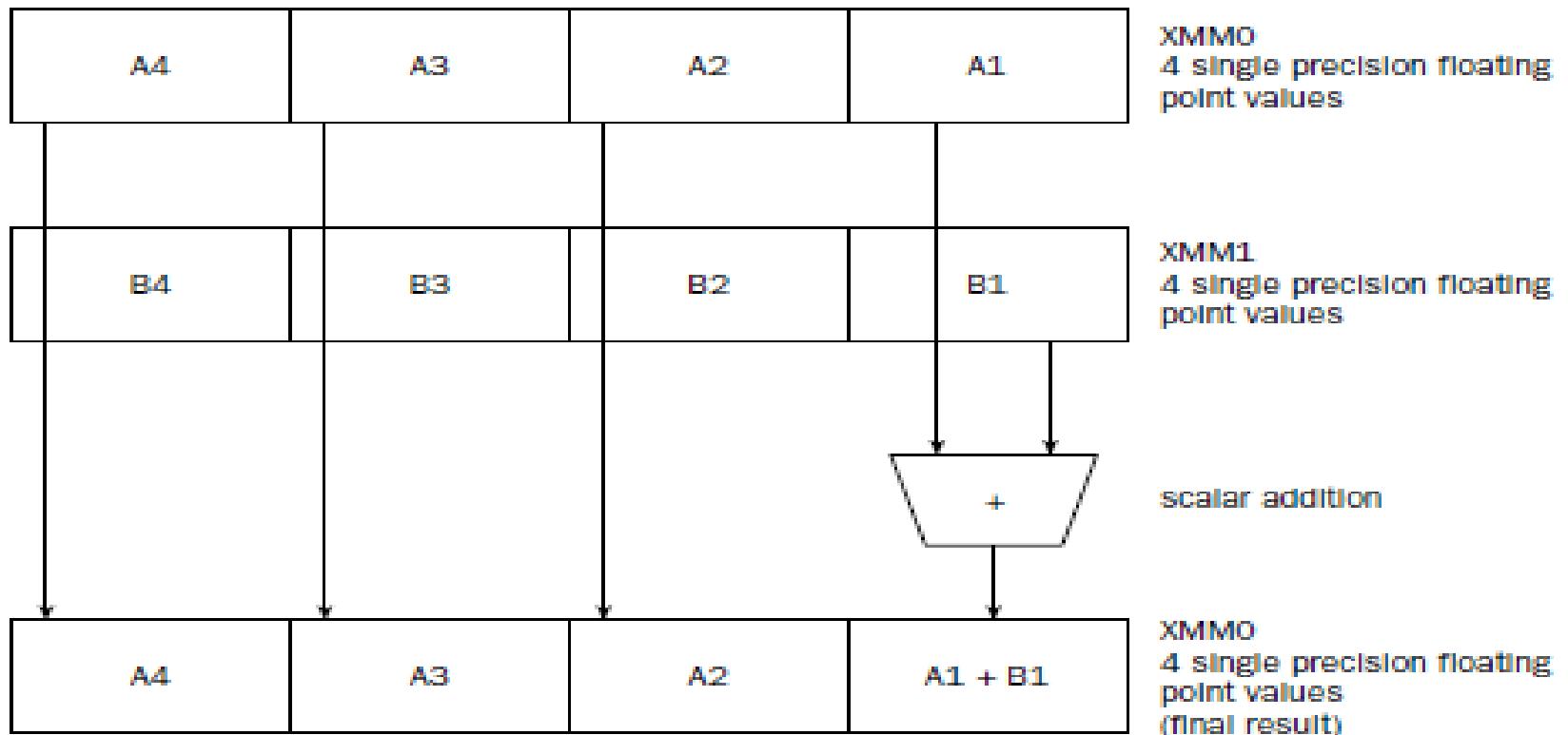
```
.section .data
value1:
.int 10, 20
value2:
.int 30, 40
.section .bss
.lcomm result, 8
.section .text
.globl _start
_start: nop
```

```
    movq value1, %mm0
    movq value2, %mm1
    paddd %mm1, %mm0
    movq %mm0, result
    movl $1, %eax
    movl $0, %ebx
    int $0x80
```

PADDD Add packed double-word integers with  
wraparound  
PADDSD.....with sign saturation  
PADDUSD.....with unsigned saturation

# SSE Instructions

- Main purpose of the SSE technology is to perform SIMD operations on floating point data.



# DATA TRANSFER IN SSE TECHNOLOGY

```
.section .data
.align 16
value1:
.float 12.34, 2345.543, -3493.2, 0.4491
.section .text
.globl _start
_start:
movaps value1, %xmm0
Movl $1,%eax
Movl $0,%ebx
Int $0x80
```

```
(gdb) print $xmm0
```

**.align directive instructs the gas assembler to align the data on a specific memory boundary.**

It takes a single operand, the size of the memory boundary on which to align the data

# Addition Example of using SSE arithmetic instructions

- .section .data
- .align 16
- value1:
- .float 12.34, 2345., -93.2, 10.44
- value2:
- .float 39.234, 21.4, 100.94, 10.56
- .section .bss
- .lcomm result, 16
- .section .text
- .globl \_start
- \_start:
- nop
- movaps value1, %xmm0
- movaps value2, %xmm1
- addps %xmm1, %xmm0
- movaps %xmm0, result
- movl \$1, %eax
- movl \$0, %ebx
- int \$0x80

```
(gdb) print $xmm0
```

Thank you

# **Unit 5**

## **Floating point and SIMD instructions**

Prepared By- Mrs . J. D. Pakhare

# Floating point numbers in IA32 Architecture:

- IEEE754 Single precision floating point format
- IEEE754 Double precision floating point format
- Double extended precision floating point number

(a) Single precision floating point number

Sign	Exp	Fraction	0
1	8	23	

31 30 23 22 0

(b) Double precision floating point number

Sign	Exp	Fraction	0
1	11	52	

63 62 52 51 0

(c) Double extended-precision floating point number

Sign	Exp	Frac	Fraction	0
1	15	1	64	

79 78 64 63 62 0

Floating point data types in IA32 processors.

# Floating point conversion in single & double precision

- **single precision:**
- -24.75 --- convert to binary and normalize  
=  $-1.100011 \times 2^4$

$$\text{Biased exponent} = 127 + 4 = 131 = (10000011)_2$$

- **Double precision:**
- -24.75 --- convert to binary and normalize  
=  $-1.100011 \times 2^{14}$
- Biased exponent =  $1023 + 4 = 1027 = (10000000011)_2$
- Represent in single and double precision floating point format--

# Floating point conversion in single & double precision

IEEE single precision	Sign	Exp	Mantissa
	1	10000011	1000110.....00
IEEE754 no in Hex	0xC1C60000		
IEEE double precision	Sign	Exp	Mantissa
	1	10000000011	10001100.....0000
IEEE754 no in Hex	0xC038 C000 0000 0000		

- 0.0625--- convert to binary and normalize  
 $= 0.0001 * 2^0 = 1.0 \times 2^{-4}$
- Biased exponent =  $127 - 4 = 123 = (01111011)_2$

IEEE single precision	Sign	Exp	Mantissa
	0	01111011	00000.....00
IEEE754 no in Hex	0x3D80 0000		
IEEE double precision	Sign	Exp	Mantissa
	0	01111111011	00000000.....0000
IEEE754 no in Hex	0x3FB0 0000 0000 0000		

- Convert +7.5 into single and double precision FP format?

# ***Moving floating-point values/data transfer***

- FLD ----instruction is used to move floating-point values into and out of the FPU registers.

**fld source**

- source can be a 32, 64, or 80-bit memory location.

flds---for single precision

fldl---- for double precision

- FST ---instruction is used for retrieving the top value on the FPU register stack and placing the value in a memory location.

**fst dest**

fststs---for single precision

fstl---- for double precision

- fstp memvar
  - fstp src
- 
- Fild memvr
  - Fist memvar
  - Fistp memvar

# ***Defining floating-point values--Program***

```
.section .data
    value1:
        .float -24.75
    value2:
        .double -24.75

.section .bss
    .lcomm data,4
    .lcomm data1,8
```

```
.section .text
.globl _start
_start: nop
flds value1
fldl value2
fstps data
fstl data1
movl $1,%eax
movl $0,%ebx
int $0x80
```

```
student@student@OptiPlex-3020:~  
student@student-OptiPlex-3020:~$ as -gstabs -o float1.o float1.s  
student@student-OptiPlex-3020:~$ ld -o float1 float1.o  
student@student-OptiPlex-3020:~$ gdb -q float1  
Reading symbols from float1...done.  
(gdb) break*_start+1  
Breakpoint 1 at 0x8048075: file float1.s, line 11.  
(gdb) run  
Starting program: /home/student/float1  
  
Breakpoint 1, _start () at float1.s:11  
11      flds value1  
(gdb) x/fx &value1  
0x8049093:    0xc1c60000  
(gdb) s  
12      fild value2  
(gdb) x/gfx &value2  
0x8049097:    0xc038c00000000000  
(gdb) s  
13      fstl data  
(gdb) s  
14      movl $1,%eax  
(gdb) x/gfx &data  
0x80490a0 <data>:    0xc038c00000000000  
(gdb)
```

# Architecture of floating point processor

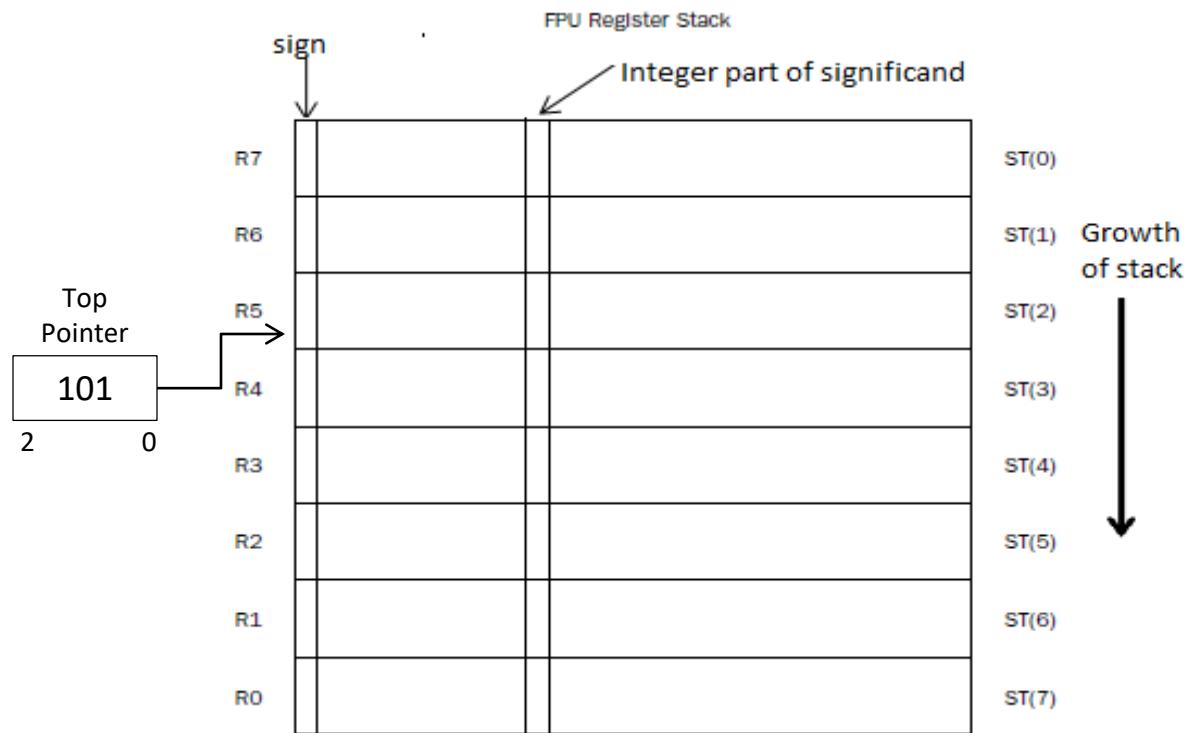


Fig. x87 Data registers in IA32 processor

- **Data registers in IA32 processor**

- It include 8 general purpose data registers, each 80 bit wide and capable of storing a real number in double extended precision format.
- As data is loaded into the FPU stack, the stack top moves downward in the registers.
- When eight values have been loaded into the stack, all eight FPU data registers have been utilized. If a ninth value is loaded into the stack, the stack pointer wraps around to the first register and replaces the value in that register with the new value

# Architecture of floating point processor

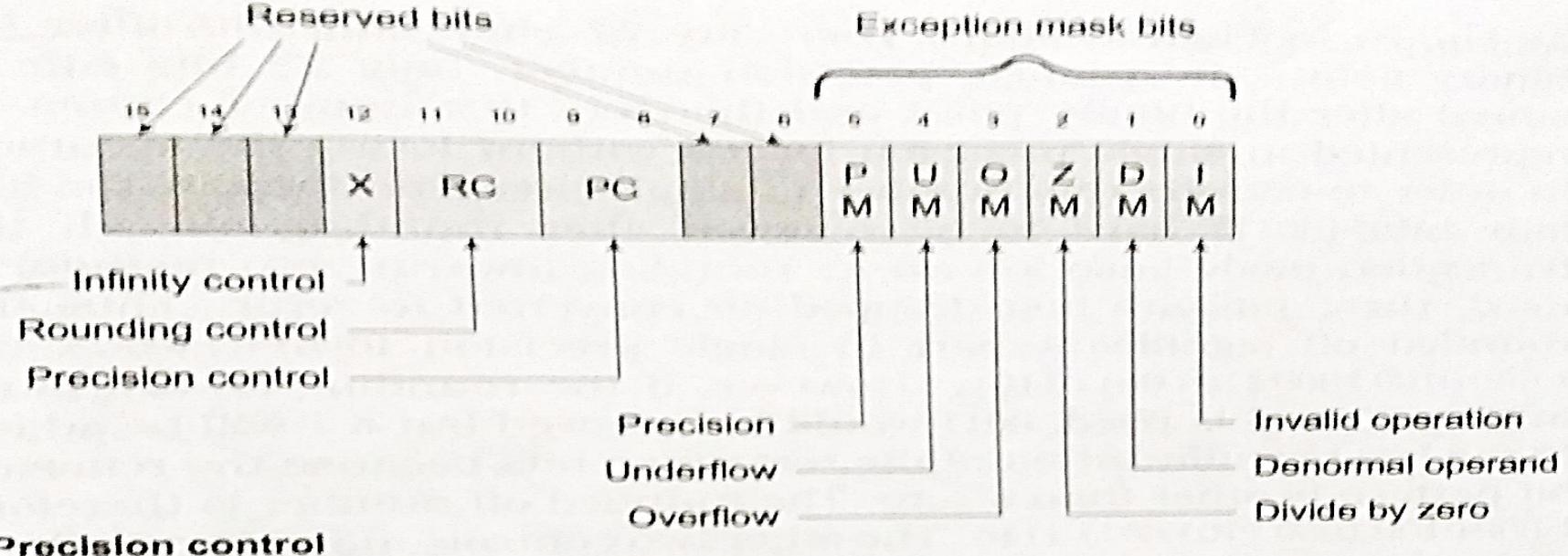
- IA32 environment includes 8 general purpose data registers, each of 80 bits and capable to store real no in double extended precision format.
- In addition to general purpose data register, the x87 , operating environment includes three 16 bit registers .These registers are the following
  - X87 control register
  - X87 status register
  - X87 tag register

# Architecture of floating point processor

- Control register:
  - Is used to control the way x87 instructions handle the precision and rounding off operands.
  - Bits for exception masking
  - Bits for computation control
  - 6 different floating point exceptions
  - Occurrence of these exceptions can be masked by setting the corresponding mask bit in the control register to 1

- Computation control bits are used to control precision and rounding .
- Infinity control bit for compatibility with the x87 FPUs. In GNU/Linux based computation it is always 0
- Rounding is needed when
  - When no in register is to be converted to a lower precision i.e. single or double

# Architecture of floating point processor



## Precision control

- 00: Single precision (24 bits)
- 01: Unused, Reserved.
- 10: Double precision (53 bits)
- 11: Double extended-precision (64 bits).  
(Default)

## Exception mask

- 0: Unmask corresponding exception  
(cause exception to occur)
- 1: Mask corresponding exception  
(cause exception to not occur)

## Rounding control

- 00: Round to nearest integer
- 01: Round down (towards  $-\infty$ )
- 10: Round up (towards  $+\infty$ )
- 11: Round towards zero (truncate)

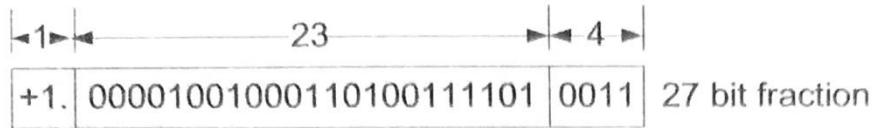
Figure 9.3: x87 control register.

- Used to control the precision and rounding of operands
- Infinity control bit is always set to 0

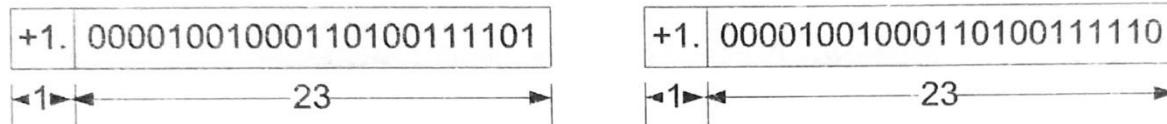
# Control register

- X87 FPU provides 6 different kinds of FP exceptions
- Occurrence of these exceptions can be masked by **setting** the corresponding mask bit in the control register.
- **PC** bits specify precision(single or double or double extended) of FP computations
- **RC** bits used to define way of rounding is performed. (shown in fig 9.4)

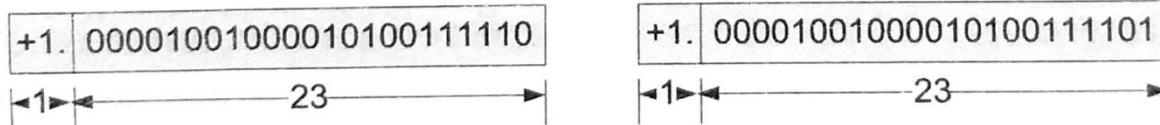
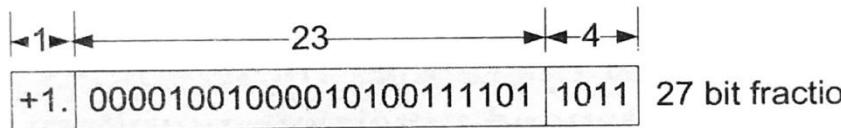
# Rounding control in IA32 :



Used to handle the precision and rounding of operands  
Infinity bit always set to 0



(a) Rounding controls (towards 0 and  $+\infty$ )



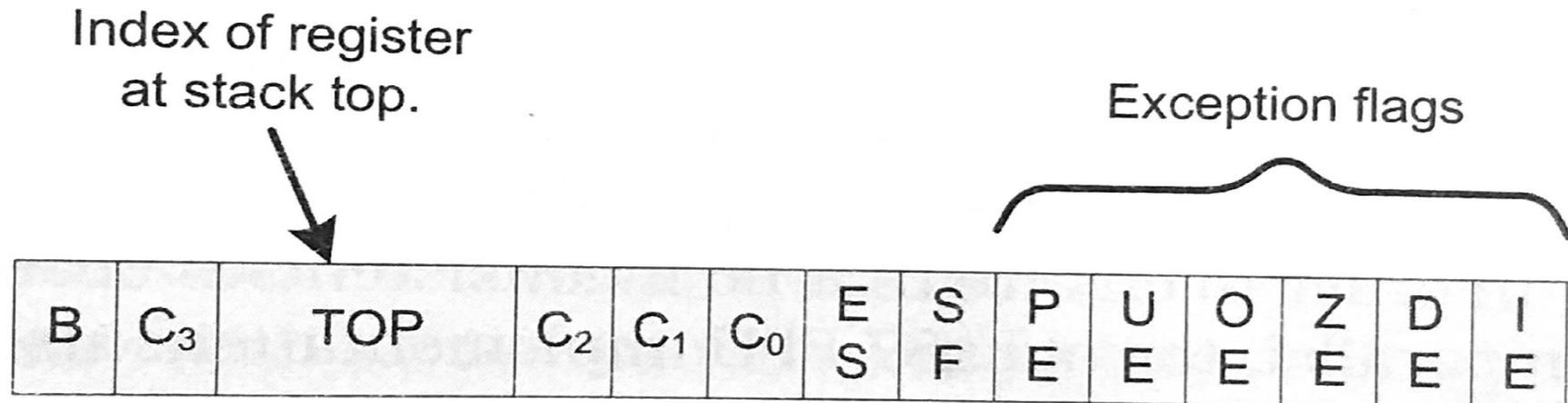
(b) Rounding controls (towards nearest and  $-\infty$ )

**Figure 9.4:** Rounding controls in IA32 architectures.

# X87 status register:

- It indicates various exception flags, condition codes and stack top details.

# X87 status register:



**B: Busy flag**

0: FPU free

1: FPU busy

**C<sub>3..C<sub>0</sub></sub>: Condition codes**

**ES: Error status**

1: Error occurred

0: No error

**SF: Stack fault**

**Exception flags**

PE: Precision exception

UE: Underflow exception

OE: Overflow exception

ZE: Divide-by-zero exception

DE: Denormal operand exception

IE: Invalid operation exception

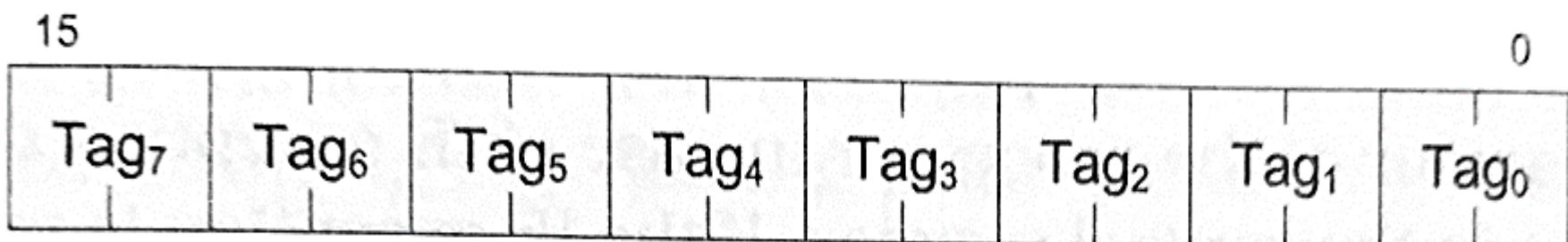
**Figure 9.5:** x87 status register.

- Status register can be saved in memory and can be taken into ax register.
- 4 condition code flags  $C_0$  to  $C_3$  –generated during execution of instruction
  - These indicates the result of floating point comparison and arithmetic operations.
  - Integer compare and branch instructions may be used to operate according to condition codes.
- **Exceptions flags** indicates various FP error conditions that may occur during execution of X87 instruction.
- If FP exception occurs during computation , ES (Error status) bit is set in the status register.
  - e. g. if an exception causes divide by zero FP exception , **ZE flag is set to 1.**

- **SF- Stack fault---** indicates errors due to stack overflow or underflow conditions.
- **B-Busy bit** ---for compatibility and most programs do not use this bit.

X87 data register:

- Each data register of X87 FPU has 2 bit tag contained in tag register.
- Code in the tag register indicates the type of the value of the **corresponding data register**.
- e.g. If Data register contain valid normalized value ,corresponding tag code is **00**



Tag<sub>i</sub> is associated with data register R<sub>i</sub>

00 → R<sub>i</sub> has a valid value

01 → R<sub>i</sub> has a zero

10 → R<sub>i</sub> has a special value (NaN,  $\pm\infty$  or denormalized number)

11 → R<sub>i</sub> is empty (has not been loaded yet)

**Figure 9.6:** x87 tag register.

# Floating point exceptions

## 1. Invalid operation exception (IE)

- It is raised when operands of the instruction contain invalid values such as NaN.
- $0 * \pm\infty$  i.e. multiplication of 0 by  $\pm\infty$ , division of  $\pm\infty$  by  $\pm\infty$

## 2. Divide by zero exception (ZE)

- Occurs when divisor contains a zero and dividend is a finite number other than 0

## 3. De-normalized operand exception (DE)

- It is raised when one of the operands of an instruction is in de-normalized form.
- De-normalized numbers can be used to represent numbers with magnitudes too small to normalize (i.e. below  $1.0 \times 2^{-126}$ )

## 4. Numeric overflow exception (OE)

- Occurred when rounding result would not fit into destination operand.
- When a double precision floating point number is to be stored in single precision floating point format, a numeric overflow is set to have occurred.

## 5. Numeric underflow exception (UE)

- Occur when the instruction generate result whose magnitude is less than the smallest possible normalized number.

## 6. Precision (inexact result)exception (PE)

- Division of  $(1.0 / 12.0)$  results in recurring infinite sequence of bits.
- This number cannot be stored in any format without loss of precision.

# Architecture of floating point processor

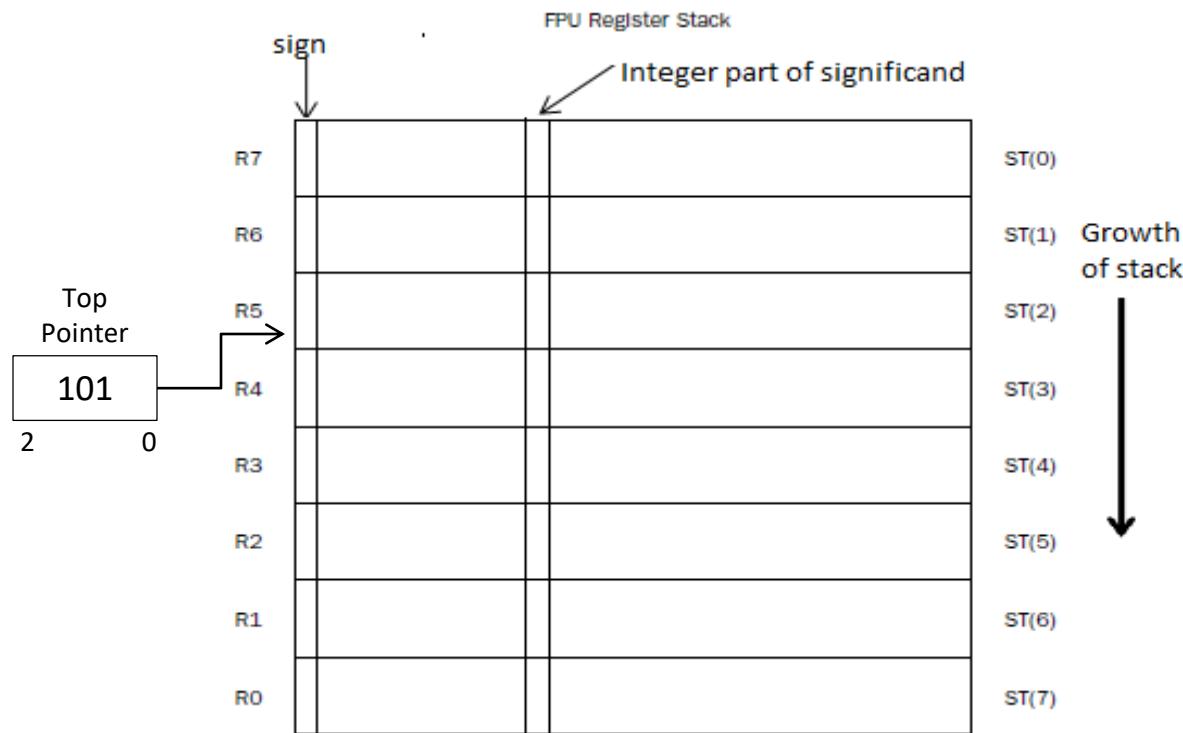


Fig. x87 Data registers in IA32 processor

- **Data registers in IA32 processor**

- It include 8 general purpose data registers, each 80 bit wide and capable of storing a real number in double extended precision format.
- As data is loaded into the FPU stack, the stack top moves downward in the registers.
- When eight values have been loaded into the stack, all eight FPU data registers have been utilized. If a ninth value is loaded into the stack, the stack pointer wraps around to the first register and replaces the value in that register with the new value

- **X87 register addressing:**

- Register stack uses registers in a circular buffer
- Data register are organized as stack of registers
- e.g. if the top field in the x87 status register has a value 5 then **st(0), refers to x87 data register R5 as shown in Table 9.7**

**Table 9.7:** Register stack addressing in x87 FPU

<i>Stack register</i>	<i>Data register when TOP = 5</i>	<i>Data register when TOP = 2</i>
st(0)	R5	R2
st(1)	R4	R1
st(2)	R3	R0
st(3)	R2	R7
st(4)	R1	R6
st(5)	R0	R5
st(6)	R7	R4
st(7)	R6	R3

# Floating point instructions:

- Basic arithmetic instruction
- Constant loading instructions
- Trigonometric, logarithmic and exponentiation instructions
- Data comparison instructions
- Data transfer instructions
- FPU control instructions

## Basic arithmetic instruction

- Addition instructions:

**fadd memVar**

**faddl value1**

: adds double precision FP no stored at  
memory location value1 to the stack top

**fadd src, dest**

**fadd %st, %st(3)**

: adds FP no in register st to register st(3),  
result is in st(3)

**faddp**

faddp

$st(1)=st(1)+st(0)$ , pop at  $st(0)$

**faddp dest**

faddp %st(3)

$st(3)=st(3)+st(0)$ , pop the stack

,result will be available at  $st(2)$ (new TOS)

**fiadd memVar** --- Add a 16- or 32-bit integer value to  
 $st(0)$  and store result at ST(0)

## **–Fiadds Value1**

**-add 16 bit integer(short int -suffix s)**

stored at memory location value1 to a floating point number in st

## **–Fiaddl Value2**

**-add 32 bit (long int- suffix l) integer**

stored at memory location value2 to a floating point number in st

# Subtraction instruction:

**fsubs/ fsubl no1**----subtract single/double precision floating point (32 bit/64 bit) number **no1** from st(0) and store result at st(0)

$$st(0) = st(0) - no1$$

**fsub st(3), st(0)** -----  $st(0) = st(0) - st(3)$

**fsubp st(3)** -----  $st(3) = st(3) - st(0)$

:after the operation ,one item is removed from stack (pop)and hence  $st(3)$  becomes  $st(2)$

**fisubs/ fisubl val1** ---- Subtract a 16 or 32 bit integer value of memory from st(0) and store result at st(0)

**st(0)= st(0) – 16/32 bit integer from memory location  
val1**

**Fisubs val1**

**st(0)= st(0) –val1**

**fsubr st(3),st(0)** ----- **st(0)= st(3)- st(0)**

**fsubrp st(2)**-----  $st(2) = st(0) - st(2)$

:after the operation ,one item is removed from stack (pop)and hence  $st(2)$  becomes  $st(1)$

**fisubrl val1** ----- subtract  $st(0)$  from 32 bit int from memory location  $val1$

$$st(0) = val1 - st(0)$$

# Multiplication and division instructions:

Instruction	Description
<b>fmuls val1</b>	st(0)= st(0) * single precision floating point no stored at memory location val1
<b>fmul st(3),st(0)</b>	st(0)= st(0) *st(3)
<b>fmulp st(6)</b>	st(6)= st(6) *st(0)—after operation, stack is incremented (pop) by 1 and hence st(6) becomes st(5)
<b>fimuls val1</b>	st(0)= st(0) *16 bit integer from memory location val1

# Multiplication and division instructions:

Instruction	Description
<b>fdivl val1</b>	$st(0) = st(0) / \text{double precision floating point number stored at memory location val1}$
<b>fdiv st(0),st(3)</b>	$st(3) = st(3) * st(0)$
<b>fdivp</b>	$st(1) = st(1) / st(0)$ —after operation, stack is incremented (pop) by 1 and hence $st(1)$ becomes $st(0)$
<b>fidivl val32</b>	$st(0) = st(0) / 32 \text{ bit integer from memory location val32}$
<b>fdivrl val1</b>	double precision floating point number stored at memory location <b>val1/st(0)</b>
<b>fdivr st(3),st(0)</b>	$st(0) = st(3) / st(0)$

# Remainder computation:

Instruction	Description
fprem	<ul style="list-style-type: none"><li>➤ Compute the remainder when <math>st(0)</math> is divided by <math>st(1)</math> and return remainder in <math>st(0)</math></li><li>➤ Truncating the Division result to an integer</li></ul>
fprem1	<ul style="list-style-type: none"><li>➤ Compute the remainder when <math>st(0)</math> is divided by <math>st(1)</math> and return remainder in <math>st(0)</math></li><li>➤ Rounding the Division result to the nearest integer</li></ul>

#An example of basic FPU math to compute  
 $((43.65 / 22) + (76.34 * 3.1)) / ((12.43 * 6) - (140.2 / 94.21))$

#step-by-step analysis of what to perform the calculation

1. Load 43.65 into ST0.
2. Divide ST0 by 22, saving the results in ST0.
3. Load 76.34 in ST0 (the answer from step 2 moves to ST1).
4. Load 3.1 in ST0 (the value in step 3 moves to ST1, and the answer from Step 2 moves to ST2).
5. Multiply ST0 and ST1, leaving the answer in ST0.
6. Add ST0 and ST2, leaving the answer in ST0 (this is the left side of the equation).
7. Load 12.43 into ST0 (the answer from Step 6 moves to ST1).
8. Multiply ST0 by 6, leaving the answer in ST0.
9. Load 140.2 into ST0 (the answer from Step 8 moves to ST1, and from Step 6 to ST2).
10. Load 94.21 into ST0 (the answer from Step 8 moves to ST2, and from Step 6 to ST3).
11. Divide ST1 by ST0, popping the stack and saving the results in ST0 (the answer from Step 8 moves to ST1, and from Step 6 to ST2).
12. Subtract ST0 from ST1, storing the result in ST0 (this is the right side of the equation).
13. Divide ST2 by ST0, storing the result in ST0 (this is the final answer)

```

.section .data
value1:
    .float 43.65
value2:
    .int 22
value3:
    .float 76.34
value4:
    .float 3.1
value5:
    .float 12.43
value6:
    .int 6
value7:
    .float 140.2
value8:
    .float 94.21
.section .text
    .globl _start
    _start:
        nop
        finit
        flds value1
        fidiv value2
        flds value3
        flds value4
        fmul %st(1), %st(0)
        fadd %st(2), %st(0)
        flds value5
        fimul value6
        flds value7
        flds value8
        fdivrp
        fsubr %st(1), %st(0)
        fdivr %st(2), %st(0)
        movl $1,%eax
        movl $0,%ebx
        int $0x80

```

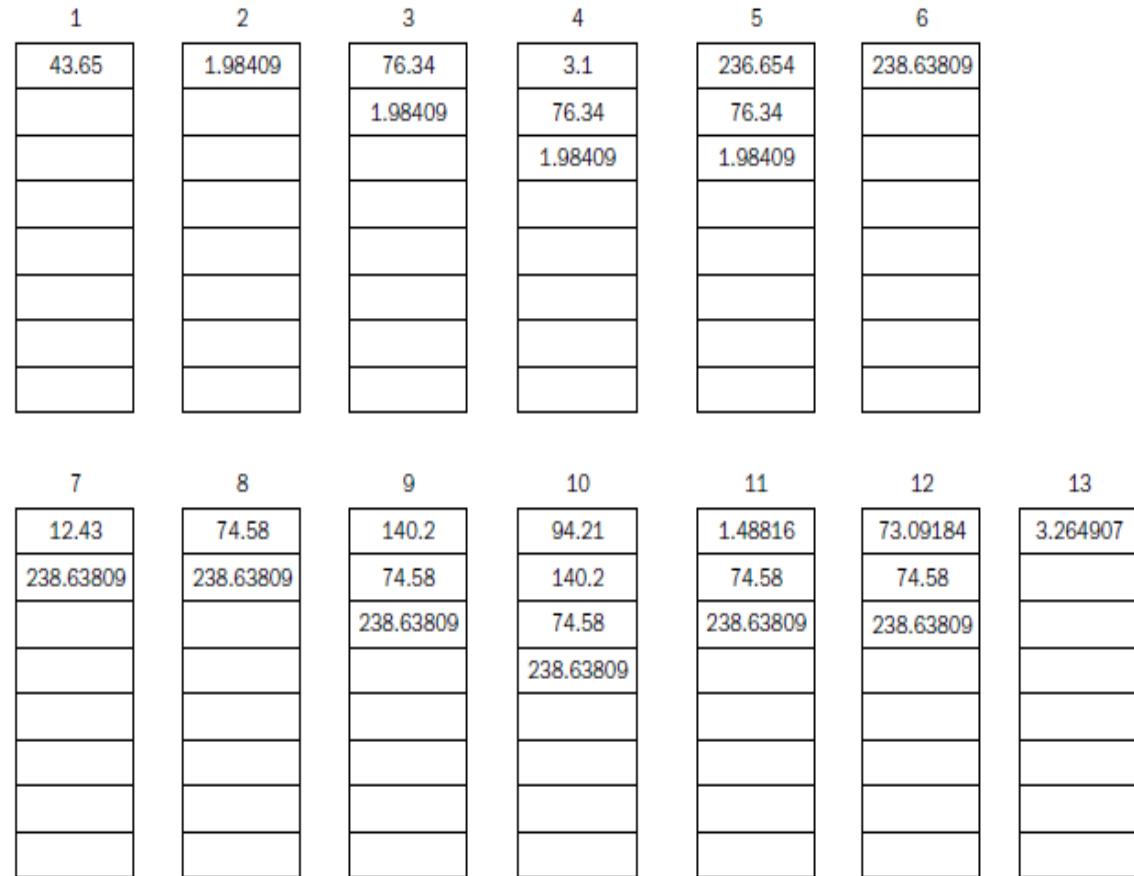


Fig. Calculation sequence

# Other instructions:

Instruction	Description
<b>Square root computation</b>	
fsqrt	<ul style="list-style-type: none"><li>➤ Square root of a number on top of stack st(0) is computed using <b>fsqrt</b></li><li>➤ result is in st(0)</li></ul>
<b>Miscellaneous computations</b>	
fchs	<ul style="list-style-type: none"><li>➤ Changes the sign of the input on the top of stack st(0)</li><li>➤ After execution +ve no become –ve and –ve no becomes +ve.</li></ul>
fscale	<ul style="list-style-type: none"><li>➤ takes 2 operands on the stack (i.e. st(0) and st(1) and replaces the TOS by the result of computation - <math display="block">st(0)= st(0)*2^{st(1)}</math></li></ul>
fabs	Compute the absolute value of register st(0)

# #Example of the FABS, FCHS, and FSQRT instructions

```
.section .data
value1:
    .float 395.21
value2:
    .float -9145.290
value3:
    .float 64.0

.section .text
.globl _start
_start:
    nop          #After execution result is :
    finit        (gdb) info all
    fld value1   .
    fchs         .
    fld value2   .
    fabs         st0  8
    fld value3   st1  9145.2900390625
    fabs         st2  -395.209991455078125
    fsqrt        (gdb)
    movl $1, %eax
    movl $0, %ebx
    int $0x80
```

# #An example of the FSCALE instruction

```
.section .data  
value:  
    .float 10.0  
scale1:  
    .float 2.0  
scale2:  
    .float -2.0  
.section .bss  
.lcomm result1, 4  
.lcomm result2, 4
```

```
.section .text  
.globl _start  
_start:  
Nop  
finit  
flds scale1  
flds value  

```

**After execution result is:**  
(gdb) x/f &result1  
0x80490b8 <result1>: 40  
(gdb) x/f &result2  
0x80490bc <result2>: 2.5  
(gdb)

# #finding Roots of a quadratic equation

Val1=4 ---int

val2=2 --int

a=1 --float

b= -4 --float

C=3 --float

Val3=0.0 and val4=0.0 ---float

$$x_1, x_2 = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

# #finding Roots of a quadratic equation

```
.section .data      .section .text
a:                   .globl _start
                     _start:
                     nop
                     flds a
                     flds c
                     fmul %st(1),%st(0)
                     fmul val1
                     fsts val3
                     flds b
                     fmul %st(0),%st(0)
                     fsub %st(1),%st(0)
                     fsqrt
                     fsts val4
                     flds b
                     fadd %st(1),%st(0)      print $st0
                     flds val2
                     fmul %st(4),%st(0)      or
                                         Info all
```

# Constant loading instructions:

- These instructions are used to load constants on the top of the stack(TOS)

Instruction	Description
fld1	Loads constant +1.0
fldz	Loads constant +0.0
fldpi	Loads constant $\pi$ (pi)
fldl2e	Loads constant $\log_2 e$
fldln2	Loads constant $\log_e 2$
fldl2t	Loads constant $\log_2 10$
fldlg2	Loads constant $\log_{10} 2$

# Trigonometric log and exponentiation instructions:

- These instructions compute Trigonometric functions for arguments stored on the top of the stack(TOS)

Instructions	Description
fsin	Compute sine of an angle in radians provided in st(0) and result is in st(0)
fcos	Compute cosine of an angle in radians provided in st(0) and result is in st(0)
fsincos	Compute sine and cosine of an angle in radians provided in st(0) --st(0) contains cosine and st(1) contains sine of the angle
fptan	Computes tangent of the angle
fptan	Computes arctangent of the angle

- #10 Write assembly language program in 80386 to convert degree to radian and compute its sin, cosine and tan value.

#degree to radian conversion and calculating the angle  
radians = (degrees \* pi) / 180

.section .data	finit
degree1:	flds degree1
.float 60.0	fdivl val180
val180:	fldpi
.int 180	fmul %st(1), %st(0)
.section .bss	fstst rad1
.lcomm rad1, 4	fsin
.lcomm out1, 4	fstst out1 ---- (0.8660)
.lcomm out2, 4	flds rad1
.section .text	fcos
.globl _start	fstst out2
_start:	movl \$1, %eax
nop	movl \$0, %ebx
	int \$0x80

# Logarithmic instructions:

- These instructions set to compute logarithm.

Instructions	Description
fyl2x	<ul style="list-style-type: none"><li>➤ It takes 2 arguments x and y on the stack st(0) and st(1) resp.</li><li>➤ Compute the <math>y \cdot \log_2 x</math></li></ul>
fyl2xp1	<ul style="list-style-type: none"><li>➤ It takes 2 arguments x and y on the stack st(0) and st(1) resp.</li><li>➤ Compute the <math>y \cdot \log_2(x+1)</math></li></ul>

# #example of using the FYL2X instruction

To find a logarithm of another base using base 2 logarithms, we can use the equation:

$$\log_b X = (1/\log_2 b) * \log_2 X$$

```
.section .text
.globl _start
_start:
nop
finit
fld1          #after execution result is-
flds base    (gdb) x/f &result
fyl2x        0x80490a8 <result>: 1.07918119
fld1          (gdb)
fdivp
flds value
fyl2x
fstp result
movl $1, %eax
movl $0, %ebx
int $0x80
```

# Exponentiation instruction:

- This instruction computes  $2^x - 1$

Instruction	Description
f2xm1	Result is at top of the stack st(0)

# FPU control instructions:

Instruction	Description
finit	initialize the x87 FPU
fninit	It must be executed when the MMX computing environment is changed to x87 FPU computing environment

- These instructions initialize the x87 FPU state to the following:
  - **Control register** is initialized to 0x37f , which masks all FP exceptions ,
  - set **rounding control** to round to nearest and
  - Precision control to double extended precision.
  - **status word** is set to 0 , meaning that no exceptions are set,
  - All condition flags are set to 0 and
  - Register stack top s initialized to 0
  - **Tag word** is set to 0xFFFF , which makes all data registers marked as empty

# program to get the FPU Status register contents

```
.section .bss
.lcomm status, 2
.section .text
.globl _start
_start:
nop
#fstsw %ax
fstsw status
movl $1, %eax
movl $0, %ebx
int $0x80
```

**After execution**

```
(gdb) x/x &status
0x804908c <status>: 0x00000000
```

**Or**

```
(gdb) info all
```

.

.

.

```
fctrl 0x37f 895
```

```
fstat 0x0 0
```

```
ftag 0x55555 349525 #shows current values of 3 regis.
```

```
(gdb)
```

**Or**

```
(gdb) print/x $fstat
```

```
$1 = 0x0
```

```
(gdb) print/x $ftag
```

```
$2 = 0xffff
```

```
(gdb) print/x $fctrl
```

```
$3 = 0x37f
```

# Data comparison instructions

- The FCOM family of instruction is used to compare two floating-point values in the FPU.
- The instructions compare the value loaded in the ST0 FPU register with either another FPU register or a floating point value in memory.

Instruction	Description
FCOM	Compare the ST0 register with the ST1 register.
FCOM ST(x)	Compare the ST0 register with another FPU register.
FCOM source	Compare the ST0 register with a 32- or 64-bit memory value.
FCOMP	Compare the ST0 register with the ST1 register value and pop the stack.
FCOMP ST(x)	Compare the ST0 register with another FPU register value and pop the stack.
FCOMP source	Compare the ST0 register with a 32 or 64-bit memory value and pop the stack.
FCOMPP	Compare the ST0 register with the ST1 register and pop the stack twice.
FTST	Compare the ST0 register with the value 0.0.

The result of the comparison is set in the C0, C2, and C3 condition code bits of the status register.

Condition	C3	C2	C0
ST0 > source	0	0	0
ST0 < source	0	0	1
ST0 = source	1	0	0

- For single precision floating point no stored in memory then
  - Use fcoms and fcomps
- For double precision floating point no stored in memory then
  - Use fcoml and fcompl
  - Fcom %st(3) ----- compares stack top with st(3)

# ***FCOMI instruction***

- The FCOMI family of instructions performs the floating-point comparisons and places the results in the **EFLAGS registers** using the carry, parity, and zero flags.

Instruction	Description
FCOMI	Compare the ST0 register with the ST(x) register.
FCOMIP	Compare the ST0 register with the ST(x) register and pop the stack.
FUCOMI	Check for unordered values before the comparison.
FUCOMIP	Check for unordered values before the comparison and pop the stack afterward.

Condition	ZF	PF	CF
$ST0 > ST(x)$	0	0	0
$ST0 < ST(x)$	0	0	1
$ST0 = ST(x)$	1	0	0

FCOMI src, %st

FCOMIP src, %st

FUCOMI src, %st

FUCOMIP src, %st

# Exchanging values

- Fxchg src
  - Top of stack exchanged with any of src i.e st(i)
- Fxchg
  - Top of stack exchanged with st(1)

# SIMD Introduction:

- Intel introduced multimedia extension (MMX) instruction set with Pentium processors.
- MMX was the first technology to support the **Intel Single Instruction, Multiple Data (SIMD) execution model**.
- Most Instructions operate simultaneously on multiple data values.
- The SIMD model was developed to process larger numbers, commonly found in applications such as **image processing , multimedia applications including voice and data communication**.

- With Pentium III, Intel introduced **streaming SIMD extension (SSE)** instruction set.
- **MMX instructions operate on integer data, and SSE instructions operate on floating point data.**
- SSE enhances performance for complex floating-point arithmetic, often used in 3-D graphics, motion video, and video conferencing.
- SSE2 – second enhancement to the SSE with 128 bit wide registers and operations on multiple integers and floating point nos.

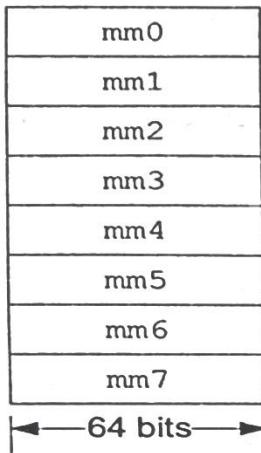
# SIMD environment

- The Intel Single Instruction Multiple Data (SIMD) technology provides additional ways to define integers.
- These integer perform arithmetic operations on a group of multiple integers simultaneously.
- The SIMD architecture uses the packed integer data type.
- A packed integer is a series of bytes that can represent more than one integer value.
- SIMD instruction set (i.e. MMX,SSE and SSE2) provide a few additional register called **MMX registers and XMM registers**.
- **MMX registers includes 8,64 bit registers named mm0-mm7.**

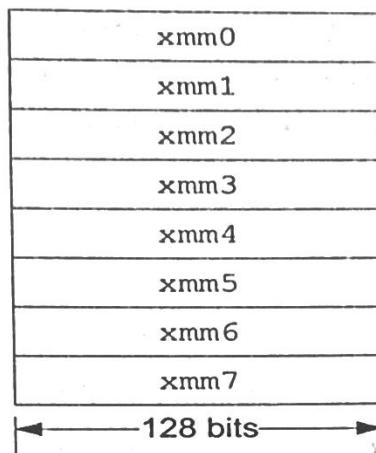
# SIMD Integers and SIMD environment

- Data register of x87 FPU are **80 bit wide** and capable of storing double extended precision FP numbers. Out of these 80 bits only 64 bits are used **by instructions in the SIMD instruction set**.
- Use **fninit or finit** instruction to initialize x87 FPU at the time of switching from SIMD to x87 environment.
- **XMM registers includes 8,128 bit registers named xmm0-xmm7.**
  - i.e. XMM registers stores four, single precision floating point nos or
  - two, single precision floating point nos or
  - Multiple integers in byte, word, long word or single 128 bit integer format.

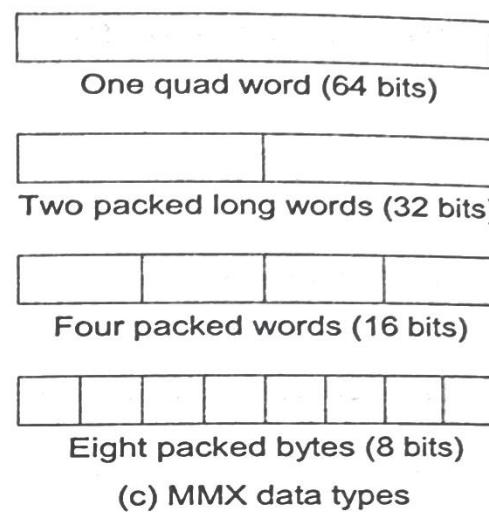
- XMM registers are independent registers and are not shared with x87 FPU data registers unlike MMX registers.



(a) MMX registers



(b) XMM registers



(c) MMX data types

One double quad word (128 bits)

Two packed quad words (64 bits)

Four packed long words (32 bits)

Eight packed words (16 bits)

Sixteen packed bytes (8 bits)

(d) XMM integer data types

Two packed double precision floating point numbers (64 bits)

Four packed single precision floating point numbers (32 bits)

(e) XMM floating point data types

**Figure 10.1:** MMX registers and supported data types.

# SIMD Integers:

- The Intel Single Instruction Multiple Data (SIMD) technology provides additional ways to define integers
- These new integer types enable the processor to perform arithmetic operations on **a group of multiple integers simultaneously.**
- The SIMD architecture uses the packed integer data type.
- A packed integer is a series of bytes that can represent more than one integer value.
  - Mathematical operations can be performed on the series of bytes as a whole

- **movq source, destination** ---move data into MMX register
  - source and destination can be an MMX register, an SSE register, or a 64-bit memory location

## **#Loading and retrieving packed integer values**

```
.section .data
packedvalue1:
    .byte 10, 20, -30, 40, 50, 60, -70, 80          #8-byte integer values
packedvalue2:
    .short 10, 20, 30, 40                            #four word integer values
packedvalue3:
    .int 10, 20                                      #two double word integer values.

.section .text
.globl _start
_start:
    movq packedvalue1, %mm0
    movq packedvalue2, %mm1
    movq packedvalue3, %mm2
    movl $1, %eax
    movl $0, %ebx
    int $0x80

    print $mm0
    print $mm1
    print $mm2

    Or reserve memory
    in the .bss section
```

# ***Performing MMX operations***

- Once the data is loaded into the MMX register, parallel operations can be performed on the packed data using a single instruction.
- The operations are performed on each packed integer value in the register, utilizing the same placed packed integer values. As shown in following **fig a**
- ***MMX addition and subtraction instructions:***
- With normal addition and subtraction with general-purpose registers, if an overflow condition exists from the operation, the EFLAGS register is set to indicate the overflow condition.
- With packed integer values, multiple result values are computed simultaneously. This means that a single set of flags cannot indicate the result of the operation, as in normal integer math.

- when using MMX addition or subtraction, we must decide ahead of time what the processor should do in case of overflow or underflow conditions within the operation.
- There are three ways to handle the conditions of result overflow or underflow so the result will be meaningful:
  - Normal operation (Wraparound arithmetic )
  - Operation with Signed saturation
  - Operation with Unsigned saturation

- Overflow flag – to represent signed number overflow
- carry flag – to represent unsigned number overflow
- To indicate that the result cannot be represented in designed no of bits.

Data Type	Positive Overflow Value	Negative Overflow Value
Signed byte	127	-128
Signed word	32,767	-32,768
Unsigned byte	255	0
Unsigned word	65,535	0

# Examples of SIMD byte operations with saturations

A	B	Operation	Wrap around	Signed saturation	Unsigned saturation
0xA3	0xC2	A+B	0x65	0x80	0xFF
0x57	0xB2	A+B	0x09	0x09	0xFF
0x78	0x40	A+B	0xB8	0x7F	0xB8
0x40	0x72	A-B	0xCE	0xCE	0x00
0x44	0x23	A-B	0x21	0x21	0x21

**Wrap around-** Whenever the result is larger than 0xFF, Extra bit is dropped. And wrap around results is 0x65

- **Saturation modes** ensures that results are not outside the range of minimum and maximum possible values.
  - i.e. while using **signed saturation**
    - when the true result of an operation is smaller than the minimum negative number , the result is set to the minimum negative number.
    - Similarly the result is set to the maximum positive number when the true result is more than the maximum positive number.

- While using unsigned saturation
  - The results are **set to 0** if the true result is **negative** and
  - **set to maximum positive value** if the true **result is larger than the maximum possible value.**

- **Signed saturation-** when -93 and -62 in decimal added. The summation of two numbers will be -155 which is below the minimum possible and cannot be represented in 8 bits. And provides results as 0x80 (i.e. -128)
- **Unsigned saturation-** When 163 and 194 in decimal added the result is 357, which again cannot represent in 8 bit unsigned number format. So the result is 0xFF (i.e. 255)

## **SIMD integer arithmetic instructions:**

- Operations that use MMX registers were introduced with MMX instruction set.
- Operations that use XMM registers were introduced with SSE2 instruction set.

<i>Instruction</i>	<i>Operation</i>	<i>Inst. sets</i>
<i>Addition instructions</i>		
paddb src, dest	Add packed bytes	MMX, SSE2
paddw src, dest	Add packed words	MMX, SSE2
paddd src, dest	Add packed longs (or doublewords)	MMX, SSE2
paddq src, dest	Add packed quadwords	SSE2
paddsb src, dest	Add packed bytes with signed saturation	MMX, SSE2
paddsw src, dest	Add packed words with signed saturation	MMX, SSE2
paddusb src, dest	Add packed bytes with unsigned saturation	MMX, SSE2
paddusw src, dest	Add packed words with unsigned saturation	MMX, SSE2

These instructions operate on 64 bit packed data stored in MMX registers or on 128 bit packed data stored in XMM registers.

# MMX addition

```
.section .data
value1:
.int 10, 20
value2:
.int 30, 40
.section .bss
.lcomm result, 8
.section .text
.globl _start
_start: nop
```

```
(gdb) x/2d &value1
0x804909c <value1>: 10      20
(gdb) x/2d &value2
0x80490a4 <value2>: 30      40
(gdb) x/2d &result
0x80490b0 <result>: 40      60
(gdb)
```

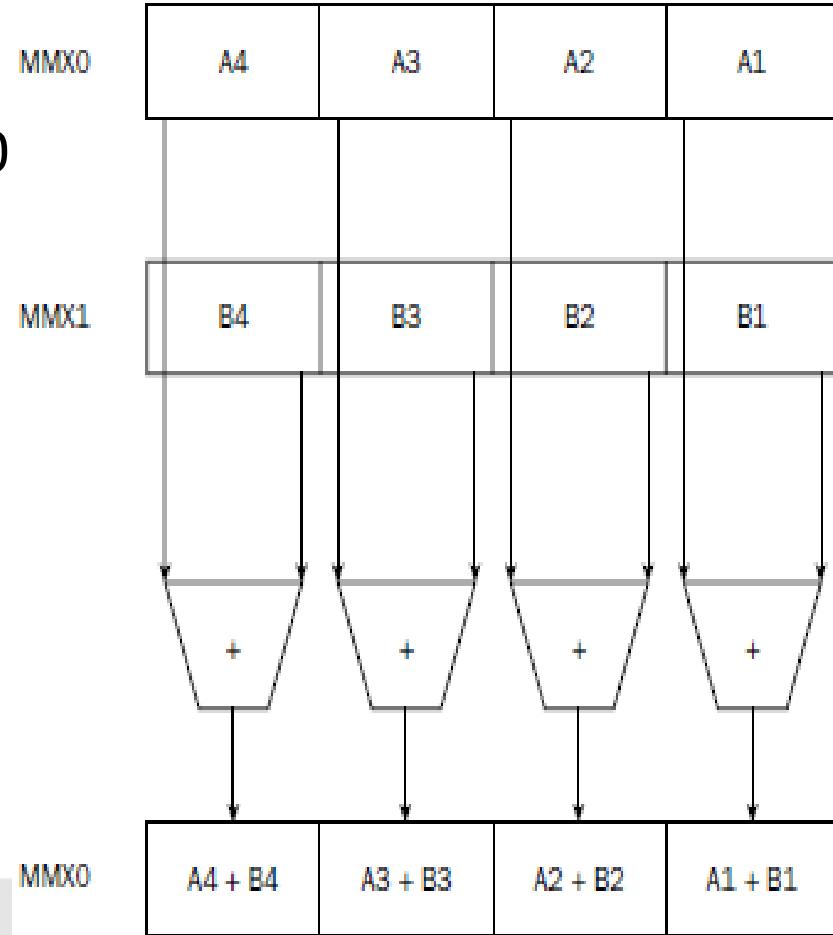


Fig a

# MMX math operations

MMX Instruction	Description
PADDB	Add packed byte integers with wraparound
PADDW	Add packed word integers with wraparound
PADDD	Add packed doubleword integers with wraparound
PADDSB	Add packed byte integers with signed saturation
PADDSW	Add packed word integers with signed saturation
PADDUSB	Add packed byte integers with unsigned saturation
PADDUSW	Add packed word integers with unsigned saturation
PSUBB	Subtract packed byte integers with wraparound
PSUBW	Subtract packed word integers with wraparound
PSUBD	Subtract packed doubleword integers with wraparound
PSUBSB	Subtract packed byte integers with signed saturation
PSUBSW	Subtract packed word integers with signed saturation
PSUBUSB	Subtract packed byte integers with unsigned saturation
PSUBUSW	Subtract packed word integers with unsigned saturation

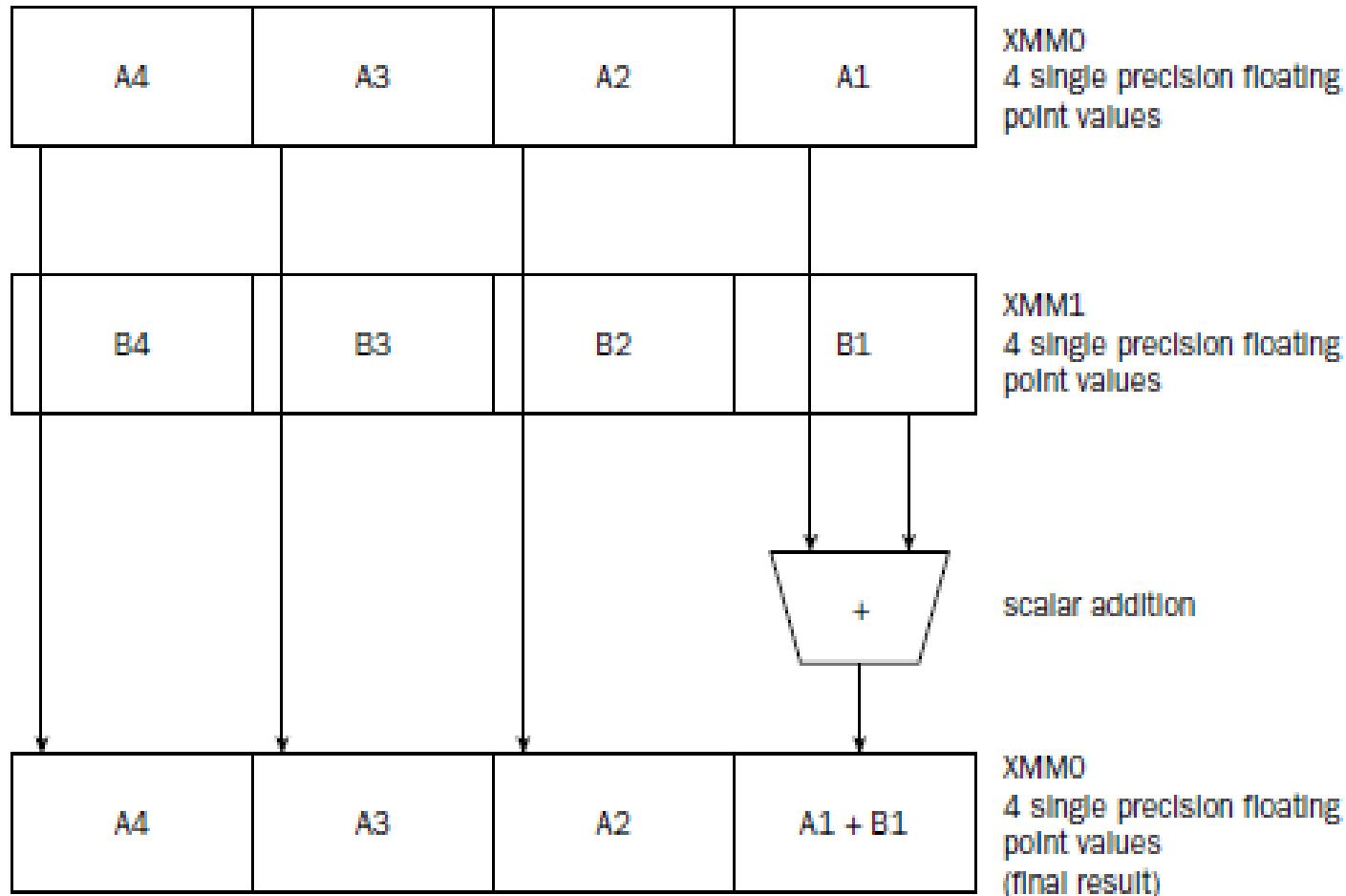
# Using SSE Instructions:

- The SSE architecture provides SIMD support for packed single-precision floating-point values.
- SSE provides new instructions for moving data into XMM registers, processing mathematical operations on the SSE data, and retrieving data from the XMM registers.
- SSE instruction has two versions.
- The first version uses a PS suffix.
  - These instructions perform the arithmetic operation on the **packed single-precision floating-point value similarly to how MMX operates**.

- The second version of the arithmetic instruction **uses an SS suffix**.
  - These instructions perform the arithmetic operation on a scalar single-precision floating-point value.
  - Instead of performing the operation on all of the floating-point values in the packed value, it is performed on only the low doubleword in the packed value. The remaining three values from the source operand are carried through to the result, as shown in following fig.

# SSE Instructions

- Main purpose of the SSE technology is to perform SIMD operations on floating point data.



- The scalar operations enable normal FPU-type arithmetic operations to be performed on one single precision floating-point value in the XMM registers.

# Moving Data in SSE technology

```
.section .data
.align 16
value1:
.float 12.34, 2345.543, -3493.2, 0.4491
.section .text
.globl _start
_start:
    movaps value1, %xmm0
```

**.align directive** instructs the gas assembler to align the data on a specific memory boundary.

- It takes a single operand, the size of the memory boundary on which to align the data.
- **MOVAPS : This instruction** move four aligned, single-precision values to XMM registers or memory.
- The SSE MOVAPS instruction expects the data to be located on a 16-byte memory boundary  
This makes it easier for the processor to read the data in a single operation.

# *SSE Arithmetic instructions*

Instruction	Description
ADDPS	Add two packed values.
SUBPS	Subtract two packed values.
MULPS	Multiply two packed values.
DIVPS	Divide two packed values.
RCPFPS	Compute the reciprocal of a packed value.
SQRTFPS	Compute the square root of a packed value.
RSQRTFPS	Compute the reciprocal square root of a packed value.
MAXFPS	Compute the maximum values in two packed values.
MINFPS	Compute the minimum values in two packed values.
ANDFPS	Compute the bitwise logical AND of two packed values.
ANDNFPS	Compute the bitwise logical AND NOT of two packed values.
ORFPS	Compute the bitwise logical OR of two packed values.
XORFPS	Compute the bitwise logical exclusive-OR of two packed values.

# Example of using SSE arithmetic instructions

```
.section .data
.align 16
value1:
.float 12.34, 2345., -93.2, 10.44
value2:
.float 39.234, 21.4, 100.94, 10.56
.section .bss
.lcomm result, 16
.section .text
.globl _start
_start:
nop
movaps value1, %xmm0
movaps value2, %xmm1
addps %xmm1, %xmm0
movaps %xmm0, result
movl $1, %eax
movl $0, %ebx
int $0x80
```

```
Breakpoint 1, _start () at ssemath.s:14
14          movaps value1, %xmm0
Current language: auto; currently asm
(gdb) s
15          movaps value2, %xmm1
(gdb) s
17          addps %xmm1, %xmm0
(gdb) print $xmm0
$1 = {f = {12.3400002, 2345, -93.1999969, 10.4399996}}
(gdb) print $xmm1
$2 = {f = {39.2340012, 21.3999996, 100.940002, 10.5600004}}
```

```
(gdb) s
|
(gdb) print $xmm0
$3 = {f = {51.5740013, 2366.3999, 7.74000549, 21}}
```

or  
x/4f &result

Thank You

# Interrupt and System calls for Linux

Prepared by Mrs. J. D. Pakhare

# Interrupts and Exceptions

- ***Interrupts***
- An interrupt is a way for the processor to “interrupt” the current instruction code path and switch to a different path.
- Interrupts come in two varieties i.e. sources of interrupts:
  - **Software interrupts**
    - Programs generate software interrupts.
    - They are a signal to hand off control to another program.
    - The INT N instruction permits interrupts to be generated within software by supplying an interrupt vector number as an operand.
    - Interrupts generated in software with the INT N instruction cannot be masked by the IF flag in the EFLAGS register.

# Interrupts and Exceptions

## – **Hardware interrupts**

- Hardware devices generate hardware interrupts
- External interrupts are received through pins on the processor
- They are used to signal events happening at the hardware level (such as when an I/O port receives an incoming signal)

# Difference: Interrupts and Exceptions

- **Interrupts** are used to handle asynchronous events external to the processor
- Two sources for external interrupts:
  - Maskable interrupts, which are signalled via the INTR pin.
  - Nonmaskable interrupts, which are signalled via the NMI (Non-Maskable Interrupt) pin.
- When an interrupt is received or an exception is detected, the currently running procedure or task is suspended while the processor executes an interrupt or exception handler.
- When execution of the handler is complete, the processor resumes execution of the interrupted procedure or task

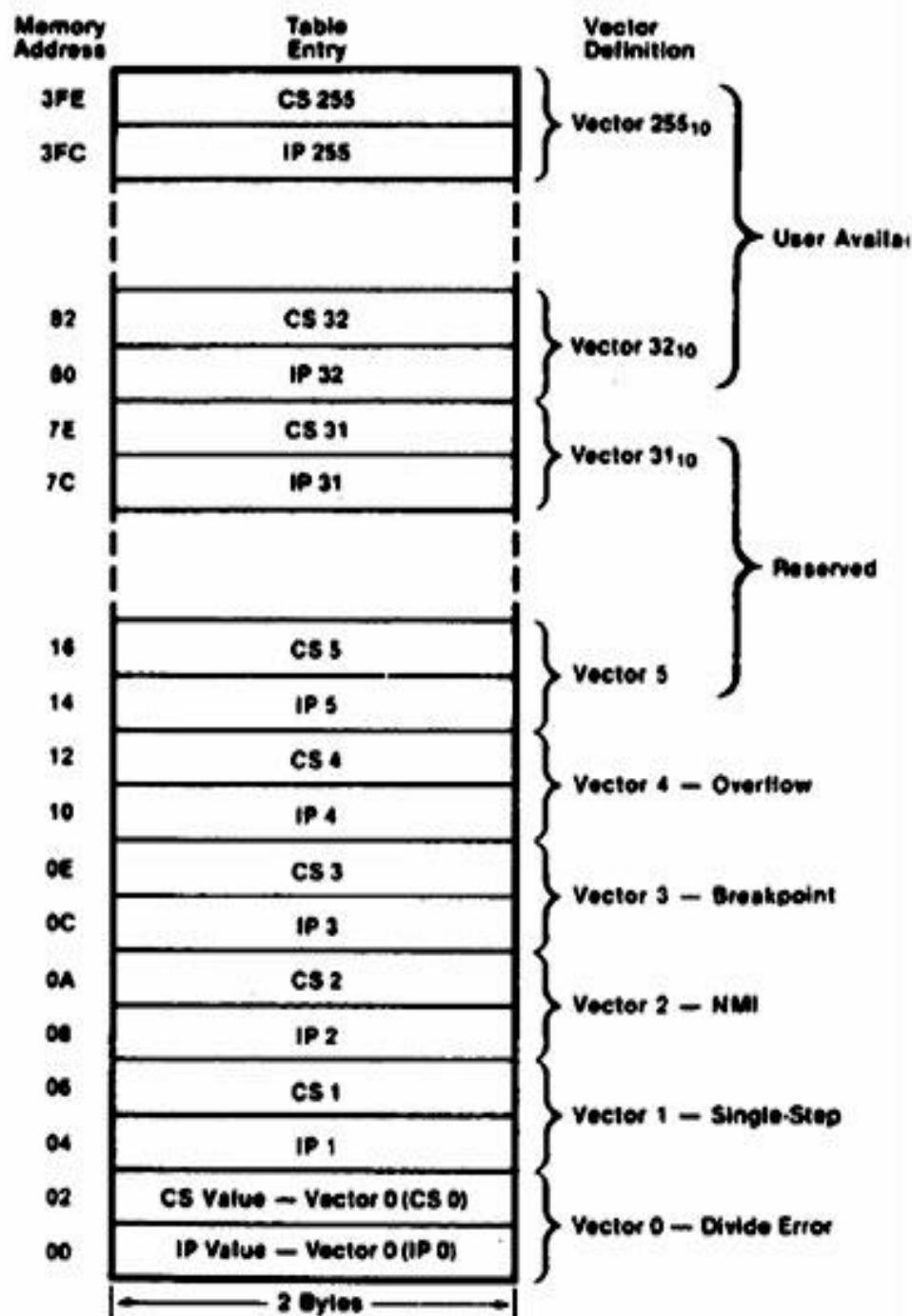
# Difference: Interrupts and Exceptions

- **Exceptions** handle conditions detected by the processor itself while executing instructions. E.g. divide by zero.
- Two sources for exceptions:
  - Processor detected. These are further classified as faults, traps, and aborts.
  - Programmed. The instructions INTO, INT 3, INT can trigger exceptions.
  - These instructions are often called "software interrupts", but the processor handles them as exceptions.

- Interrupts and exceptions vector to interrupt procedures via an interrupt table.
- The entries of the interrupt table are pointers to the entry points of interrupt or exception handler procedures.
- When an interrupt occurs, the processor pushes the current values of CS:IP onto the stack, disables interrupts, clears TF (the single-step flag), then transfers control to the location specified in the interrupt table.

# Interrupt

- Internal or external interrupt is assigned with type (N ) or specified with instruction INT N (00 to FFh)
- Intel has reserved 1024 locations for storing interrupt vector table.
- Total 256 types of interrupts from (00 to FFh)
- Each interrupt requires 4 bytes, 2 bytes for IP and 2 bytes for CS.
- interrupt vector table (IVT) starts at 0000:0000 and ends at 0000:03FFh
- (IVT) contains IP and CS of all interrupt types from 0000:0000 to 0000:03FFh
- The processor uses the vector number assigned to an exception or interrupt as an index into the interrupt descriptor table (IDT).
- The allowable range for vector numbers is 0 to 255. Vector numbers in the range 5 through 31 are reserved by the Intel



# system call format:

- To initiate a system call, the INT instruction is used.
- The Linux system calls are located at interrupt 0x80.
- When the INT instruction is performed, all operations transfer to the system call handler in the kernel.
- When the system call is complete, execution transfers back to the next instruction after the INT instruction (otherwise the exit system call is performed).

- ***System call values:***

- Each system call is assigned a unique number to identify it.
- EAX register is used to hold the system call value.
- This value defines which system call is used from the list of system calls supported by the kernel.
- e.g exit system call

```
MOVL $1, %eax
```

```
int $0x80
```

# **Linux Kernel/system call interface**

- Linux supports multiple processes running simultaneously and time sharing the same CPU.
- So that memory images of multiple processes exists simultaneously within the same physical memory.
- OS ensures that a process gets operate on its own memory image.

# **Linux Kernel /system call interface**

- Other resources such as I/O devices used by a process in a mutually exclusive manner with other processes.
- In such a mechanism, resource is used only by one process at a time and other processes wait for this process to release the resource before they can use it.
- To implement Protection of a process by other processes, Linux do not permit a process to do the direct I/O .

# Linux Kernel /system call interface

- Process makes a call to a function within the OS to perform I/O operations.
  - e.g. file can be read only if the process has a permission to read that file.
- Such functionality within the OS is called a mechanism of making **system calls**.
- OS do not permit any part of OS code to be executed by a process in the normal user mode.
- Calls to the OS function are not made by normal call instructions.

# Linux Kernel /system call interface

- Processors provide a mechanism for making system calls by using an instruction.
  - e.g. IA32 provides a trap kind instructions for handling Operating system calls.
  - Execution of this trap kind of instructions causes the CPU to execute predetermined program within the OS.

e.g. int type

- type –immediate constant value between 0 to 255.
- In Linux, **type 0x80** is used for making system call (exit) to the OS.

# System call identification

- Linux on IA32 provides only one mechanism for entering into a system call by using **int 0x80 instruction**.
- After executing this instruction the control of the program is transferred to the predetermined location within the OS kernel program. This code is **known as system call handler**.
- This system call distinguishes other system calls such as `read` , `write`, `open` etc.
- After identifying system call , appropriate function call is made within the OS to serve the system call.

- Each system call in Linux has unique identification number. This number is passed as an argument.
- The register **EAX** is used to pass the system call number.
- Many OS provide core functions that application programs can access i.e. access files, determine user and group permissions, access network resources, and retrieve and display data. These functions are called **system calls**.

## System call identification:

- EAX register is used to hold the system call value.
- This value defines which system call is used from the list of system calls supported by the kernel.
- e.g exit system call

```
MOVL $1, %eax
```

```
int $0x80
```

# *Finding system calls*

## `/usr/include/asm/unistd.h`

The unistd.h file contains definitions for each of the system calls available in the kernel.

```
#define __NR_exit      1
#define __NR_fork      2
#define __NR_read      3
#define __NR_write     4
#define __NR_open      5
#define __NR_close     6
#define __NR_waitpid   7
#define __NR_creat     8
#define __NR_link      9
#define __NR_unlink    10
#define __NR_execve    11
#define __NR_chdir     12
#define __NR_time      13
#define __NR_mknod    14
#define __NR_chmod    15
#define __NR_lchown   16
```

Each system call is defined as a name (preceded by `__NR_`), and its system call number.

# Parameter passing for System call

- In GNU/Linux parameters are passed to a system call through registers .
- All system call take less than or equal to six paramters.
- Parameters are passed using registers ebx , ecx , edx , esi, edi and ebp
- In addition to the parameter of the system call, a system call identification no is passed in register eax.
- In GNU/Linux all system call take only integer parameter or address of memory variables.

- e.g.
- Write system call takes 3 parameters other than system call identification no in register eax.
  - First parameter is an integer known as file descriptor is in ebx register.
  - Second parameter is the address of the buffer from where the data is written to the file identified by file descriptor and is in ecx register.
  - Third parameter is an integer that provides the size of the data to be written in the file and is in edx register.
  - All parameters are given below--

- The input values would be assigned to the following registers:
  - **EBX**: The integer file descriptor
  - **ECX**: The pointer (memory address) of the string to display
  - **EDX**: The size of the string to display
- The file descriptor value for the output location is placed in the **EBX**. Linux systems contain three special file descriptors:
  - **0 (STDIN)**: The standard input for the terminal device normally the keyboard and is used by the read system call to read data from the keyboard.
  - **1 (STDOUT)**: The standard output for the terminal device normally the terminal screen . The output of this file by means of write system call and appears on screen.
  - **2 (STDERR)**: The standard error output for the terminal device. Output of this file appears on the terminal or output screen

## Example of passing parameters /input values to write system call to display message

```
.section .data
output:
.ascii "Computer Science.\n"
output_end:
.equ len, output_end - output
.section .text
.globl _start
_start:
    movl $4, %eax
    movl $1, %ebx
    movl $output, %ecx
    movl $len, %edx
    int $0x80
    movl $1, %eax
    movl $0, %ebx
    int $0x80
```

- **EAX register is used to hold the system call value.**
  - The system call value for the write() system call (4) is placed in the EAX register
- **EBX: The integer file descriptor**
- The file descriptor value for the output location is placed in the EBX. Linux systems contain three special file descriptors:
  - **0 (STDIN): The standard input for the terminal device (normally the keyboard)**
  - **1 (STDOUT): The standard output for the terminal device (normally the terminal screen)**
  - **2 (STDERR): The standard error output for the terminal device (normally the terminal screen)**
- **ECX: The pointer (memory address) of the string to display**
- **EDX: The size of the string to display.**

- After assembling and linking the program —

```
$ ./syscall1test
```

Computer Science.

```
$
```

- **.equ** directive is used to define the length value by subtracting the two labels:
- The .equ directive is used to set a constant value to a symbol that can be used in the text section.
- Once set, the data symbol value cannot be changed within the program.
- The .equ directive can appear anywhere in the data section,

# *Return values from System calls :*

- Return value mechanism is implemented in a way similar to that in function calls.
- Many system calls return a value after they complete.
- System calls return a 32 bit value as an integer.
- The return value from a system call is placed in the EAX register.
- E.g. exit system call never returns and therefore there is no return value of the exit system call.
- It is your job to check the value in the EAX register, especially for failure conditions.

- All non-negative values returned by the system call represent successful execution of the system call.
- All negative values represent error condition .
- e.g.
  - In the case of the write system call, it returns the size of the string written to the file descriptor, or a negative value if the call fails.
  - The exit system call never returns and therefore there is no return value of the exit system call.

# Starting a process in GNU/Linux

- in GNU/Linux a program is executed by creating a process and then loading a program in the newly created process.
- A process is created using **fork system call** while the program is loaded using the **execve system call**.
- When a program is loaded it is made to run from a memory location whose address is specified in the executable file.
- This address is known as the start address of the program.

- By default the start address of the program is indicated by a symbolic label `_start`.
- Programs written in Assembly Language can define any address as the start address of the program.
- When a program is loaded in the memory upon execution of `execve` system call, the OS performs several operations.
  - Like it initializes the stack and various CPU registers.

- **Process** is assigned a **process ID**, or **PID**. This is how the operating system identifies and keeps track of **processes**.
- Operating systems identify a **user** by a value called a **user identifier (UID)**
- and Identify group by a group identifier (**GID**), are used to determine which system resources a **user** or group can access.

## An example to display return values of system calls related to process

```
.section .bss
.lcomm pid, 4
.lcomm uid, 4
.lcomm gid, 4
.section .text
.globl _start
_start:
    movl $20, %eax
    int $0x80
    movl %eax, pid

    movl $24, %eax
    int $0x80
    movl %eax, uid
```

After moving each system call value to the EAX register and executing the INT instruction, the return value in EAX is placed in the appropriate memory location.

# An example of getting a return value from a system call

System Call Value	System Call	Description
20	getpid	Retrieves the process ID of the running program
24	getuid	Retrieves the user ID of the person running the program
47	getgid	Retrieves the group ID of the person running the program

```
$ gdb -q syscalltest2
(gdb) break *end
Breakpoint 1 at 0x8048098: file syscalltest2.s, line 21.
(gdb) run
Starting program: /home/rich/palp/chap12/syscalltest2

Breakpoint 1, end () at syscalltest2.s:21
21          movl $1, %eax
Current language: auto; currently asm
(gdb) x/d &pid
0x80490a4 <pid>:        4758
(gdb) x/d &uid
0x80490a8 <uid>:        501
(gdb) x/d &gid
0x80490ac <gid>:        501
```

- The values in the pid, uid, and gid memory locations can be displayed as integer values using the x/d debugger command.
- While the process ID is unique to the running program, you can check the uid and gid values using the id shell command:

```
$ id  
uid=501(rich) gid=501(rich) groups=501(rich), 22(cdrom), 43(usb), 80(cdwriter),  
81(audio), 503(xgrp)  
$
```

## System calls related to process management:

- A process in GNU/Linux has 3 user ids- real user ID, effective user ID and saved user ID.
- Also it has 3 group ids –real group ID, effective group ID and saved group ID.
- Processes have parent child relationship
- For job control reasons , the processes may be grouped together and given a unique process group ID.

- Various system calls in GNU/Linux that handle the process and process related information.
- **System call:** getuid
- Input:
  - eax: SYS.getuid
    - It returns the Real user ID of the process.
- **System call:** getgid
- Input:
  - eax: SYS.getgid
    - It returns the group ID of the process.

- **System call:** getpid
- Input:
  - eax: SYS.getpid
    - It returns the process ID of the calling process.

## Process management:

- **System call:** fork
- Input:
  - eax: SYS.fork
    - It returns child process ID in the parent process and 0 in the child process.

- Fork system **creates a new process**
- Calling process becomes the parent of the newly created process .
- Child process so created has its own address space which is initialized to the values from the parent process.
- The newly created process belongs to the same session and process group as the parent.
- This system call has 2 returns. One in parent and one in child process.
  - To the parent process the return value is the process ID of the child while to the child process the return value is 0.

- **System call:** execve
- Input:
  - eax: SYS.execve
  - ebx: address of pathname string
  - ecx: address of an array of multiple command line arguments
  - edx: address of an array of multiple environment strings.
- Does not return.
- This system call initializes the memory map of the calling process by a new program loaded from an executable file given in register ebx.

- **System call:** exit
- Input:
  - eax: SYS.exit
    - Does not return.
- The exit system call is used to terminate the calling process.
  
- **System call:** kill
- Input:
  - eax: SYS.kill
  - ebx:ID
  - ecx:signal
    - The kill system call is used to send any signal to any process or processes.
    - The ID in ebx register will determine how signal is sent.

- **System call:** nice
- Input:
  - eax: SYS.nice
  - ebx: increment
    - The nice system call can change the priority of the calling process.
    - A normal process can only provide a positive increment.

**Thank You**

# Unit 6: Interrupt and System calls

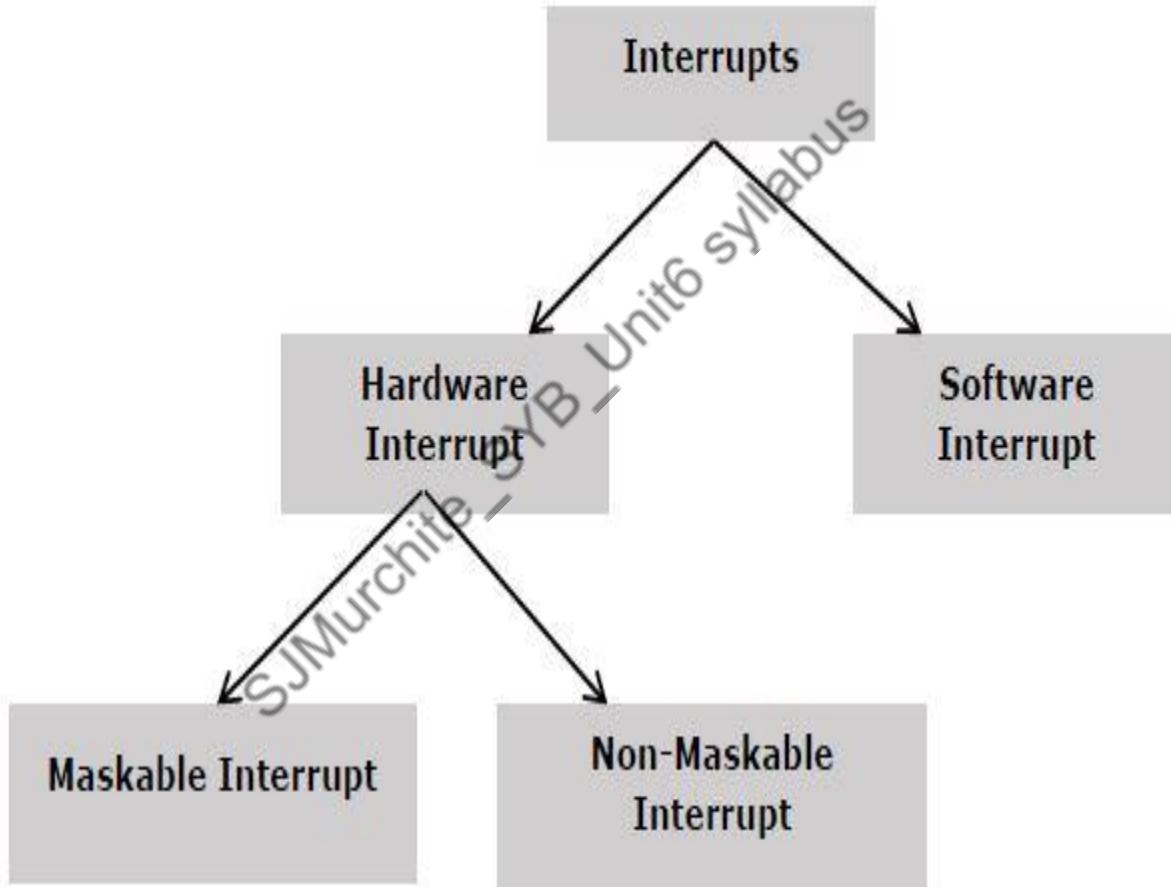
Sandip J.Murchite

An interrupt is an event that changes the sequence of instructions executed by the processor.

There are two different kinds of interrupts:

**Synchronous interrupt (Exception)** produced by the CPU while processing instructions  
**Asynchronous interrupt (Interrupt)** issued by other hardware devices

# Overview of interrupt and exception

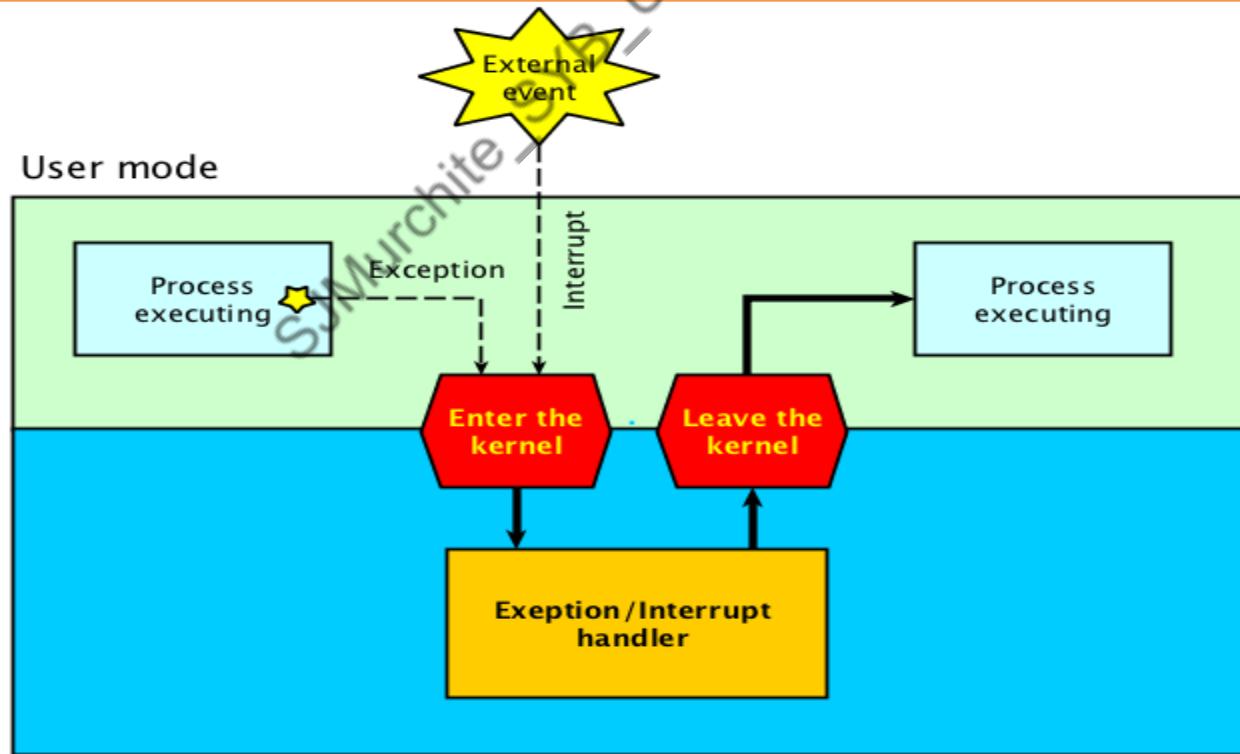


# Overview of interrupt and exception

- **Interrupts** occur at random times during the execution of a program, in response to signals from hardware
- Software can also generate interrupts by executing the INT *n* instruction.
- **Exceptions** occur when the processor detects an error condition while executing an instruction, such as division by zero.

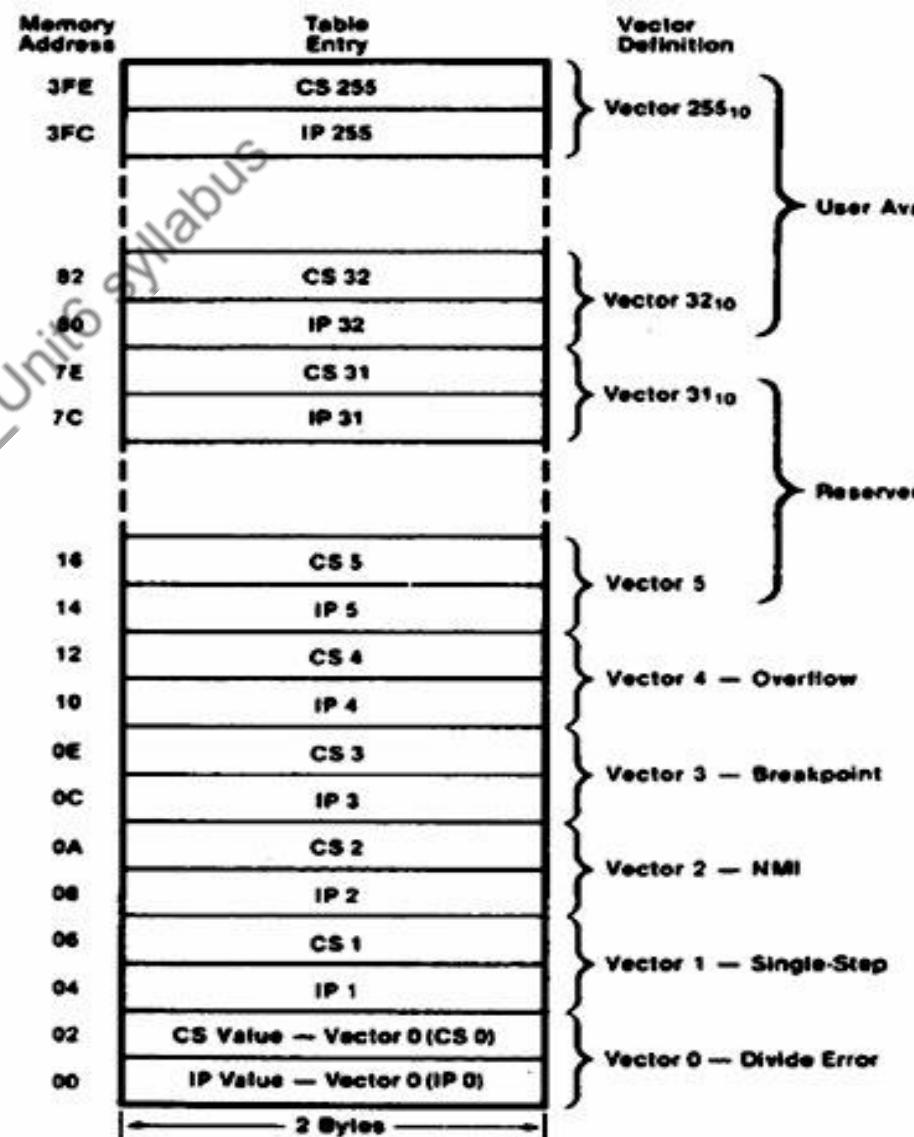
# Overview of interrupt and exception

- 1) When an interrupt is received or an exception is detected, the currently running procedure or task is suspended while the processor executes an interrupt or exception handler.
- 2) When execution of the handler is complete, the processor resumes execution of the interrupted procedure or task



# Exception and interrupt vector

- The processor uses the vector number assigned to an exception or interrupt as an index into the interrupt descriptor table (IDT).
- The allowable range for vector numbers is 0 to 255.
- Vector numbers in the range 5 through 31 are reserved by the Intel 64 and IA-32 architectures
- Vector numbers in the range 32 to 255 are designated as user-defined interrupts
- These interrupts are generally assigned to external I/O devices to enable those devices to send interrupts to the processor through one of the external hardware interrupt



# Source of interrupts

- The processor receives interrupts from two sources:
  - **External (hardware generated) interrupts.**
- External interrupts are received through pins on the processor or through the local APIC
- **Maskable Hardware Interrupts**
- Any external interrupt that is delivered to the processor by means of the INTR pin or through the local APIC is called a maskable hardware interrupt.
- **Software-generated interrupts.**
- The INT n instruction permits interrupts to be generated from within software by supplying an interrupt vector number as an operand.
- **For example**, the INT 35 instruction forces an implicit call to the interrupt handler for interrupt 35.
- Interrupts generated in software with the INT n instruction cannot be masked by the IF flag in the EFLAGS register.

# ENABLING AND DISABLING INTERRUPTS

- IE and ID are machine instructions used to set in processor status register.
- To avoid interruption by same device during execution of ISR (interrupt service routine)

First instruction of ISR can be interrupt disable

Last instruction of ISR can be interrupt enable

- The IF flag does not affect non- mask able interrupts (NMIs) delivered to the NMI pin

# Unit6: Part B- System call

Definition of system call

It is a method by which program makes a request to OS.

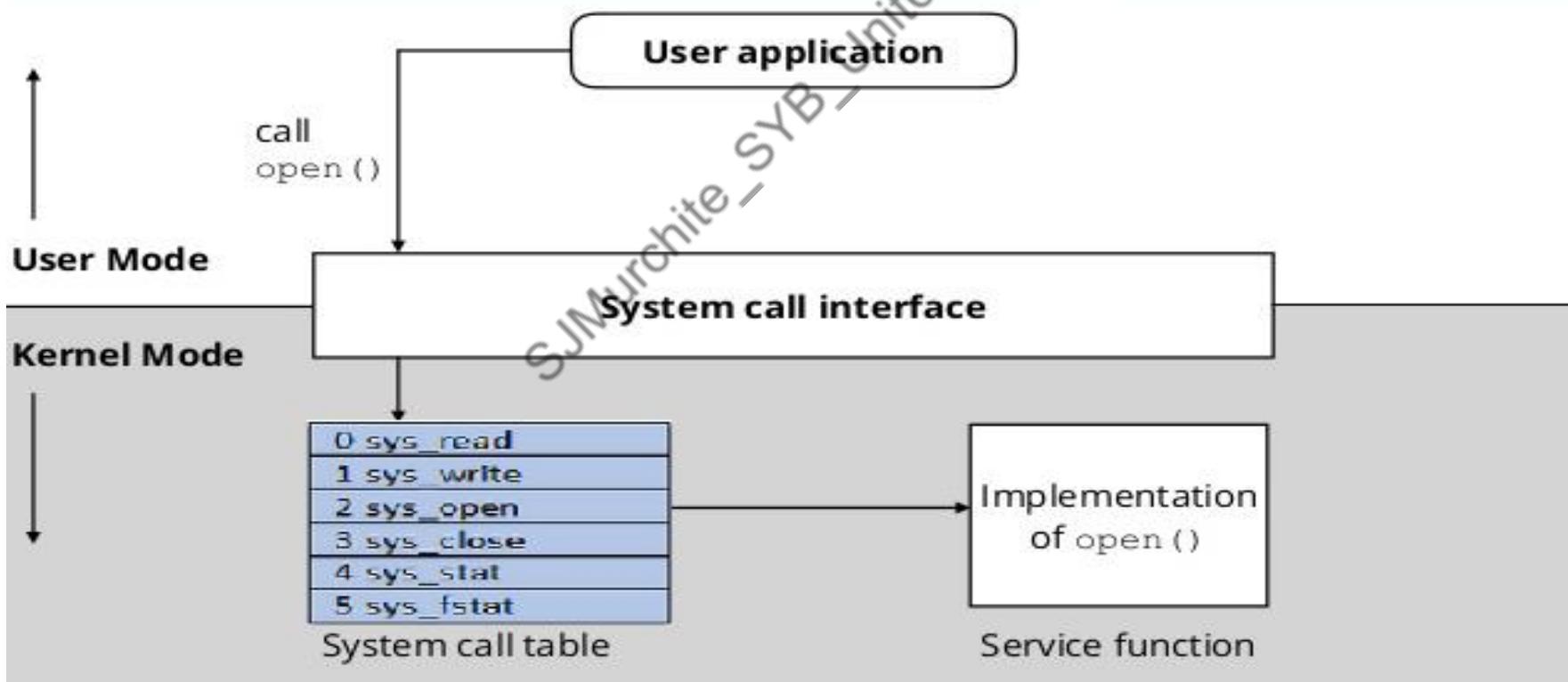
Many OS provide core functions that application programs can access called system calls.

# Linux kernel interface

- Linux support multiple processes running simultaneously
- At any time, memory images of multiple processes exit simultaneously with the same physical memory.
- Multiple resources (IO devices) are used by a process in mutually exclusive manner with other processes.
- A recourse is used only by one process at a time and other processes wait for this process to release the resource before they can use it.
- To protect a process by other processes, Linux do not permit a process to do direct IO i.e. The OS does not permit OS code to be executed by a process in user mode.
- A process makes a call to a function within an OS to perform such operations. Such functionality within os is called system call.
- IA32 processors provide instruction for handling operating system calls.
- INT N     N is an immediate constant ranging from 0 to 255
- In linux , Instruction int \$0x80 is used for making system call to the operating system.

# Example- open system call

System call execution requires context switching



# System call identification

- Linux provide a mechanism for entering into system call by using int \$0x80 instruction.
- After executing int \$0x80, control of program transfers to a predefined location within os. This code is called as system call handler.
- System call handler handle various system call such as read , write, open etc as per the system call identification process.
- Each system cal in Linux is given a unique number for identification
- In Linux register EAX is used to pass the system call number.

# System call values

- System call values
  - EAX register hold the system call value. This value defines which system call is used from list of system calls supported by kernel.
  - MOVL \$1, %eax
  - Int \$0x80

# Finding system calls

- **/usr/include/asm/unistd.h**

- #define \_\_NR\_exit 1
- #define \_\_NR\_fork 2
- #define \_\_NR\_read 3
- #define \_\_NR\_write 4
- #define \_\_NR\_open 5
- #define \_\_NR\_close 6
- #define \_\_NR\_waitpid 7
- #define \_\_NR\_creat 8
- #define \_\_NR\_link 9
- #define \_\_NR\_unlink 10
- #define \_\_NR\_execve 11
- #define \_\_NR\_chdir 12
- #define \_\_NR\_time 13
- #define \_\_NR\_mknod 14
- #define \_\_NR\_chmod 15
- #define \_\_NR\_lchown 16

```
 / arch / arm / include / asm / unistd.h
 24  /*
 25   * This file contains the system call numbers.
 26   */
 27
 28 #define __NR_restart_syscall      (__NR_SYSCALL_BASE+ 0)
 29 #define __NR_exit                (__NR_SYSCALL_BASE+ 1)
 30 #define __NR_fork                (__NR_SYSCALL_BASE+ 2)
 31 #define __NR_read                (__NR_SYSCALL_BASE+ 3)
 32 #define __NR_write               (__NR_SYSCALL_BASE+ 4)
 33 #define __NR_open                (__NR_SYSCALL_BASE+ 5)
 34 #define __NR_close               (__NR_SYSCALL_BASE+ 6)
 35 /* 7 was sys_waitpid */
 36 #define __NR_creat               (__NR_SYSCALL_BASE+ 8)
 37 #define __NR_link                (__NR_SYSCALL_BASE+ 9)
 38 #define __NR_unlink              (__NR_SYSCALL_BASE+ 10)
 39 #define __NR_execve              (__NR_SYSCALL_BASE+ 11)
 40 #define __NR_chdir               (__NR_SYSCALL_BASE+ 12)
 41 #define __NR_time                (__NR_SYSCALL_BASE+ 13)
 42 #define __NR_mknod               (__NR_SYSCALL_BASE+ 14)
 43 #define __NR_chmod               (__NR_SYSCALL_BASE+ 15)
 44 #define __NR_lchown              (__NR_SYSCALL_BASE+ 16)
 45 /* 17 was sys_break */
 46 /* 18 was sys_stat */
 47 #define __NR_lseek               (__NR_SYSCALL_BASE+ 19)
 48 #define __NR_getpid              (__NR_SYSCALL_BASE+ 20)
 49 #define __NR_mount               (__NR_SYSCALL_BASE+ 21)
 50 #define __NR_umount              (__NR_SYSCALL_BASE+ 22)
 51 #define __NR_setuid              (__NR_SYSCALL_BASE+ 23)
 52 #define __NR_getuid              (__NR_SYSCALL_BASE+ 24)
 53 #define __NR_stime               (__NR_SYSCALL_BASE+ 25)
 54 #define __NR_ptrace              (__NR_SYSCALL_BASE+ 26)
 55 #define __NR_alarm               (__NR_SYSCALL_BASE+ 27)
```

# Parameter passing for system calls

- In linux, parameters are passed to a system call through registers.
- All system calls in linux takes less than or equal to 6 parameters
- The parameters are passed using registers ebx,ecx,edx,esi,edi,ebp.

# Parameter passing for system calls

- **Example1:** write system call takes four parameters (including system call number)
  - Movl \$4,%eax
  - Movl \$1,%ebx
  - Movl \$output,%ecx
  - Movl \$len,%edx
  - Int \$0x80
  - File descriptor
    - 0- it normally bound to keyboard and used by read system call
    - 1-it normally appear on console and used by write system call to write the data
- **Example2:** exit system call is used to terminate the process
  - Movl \$1,%eax
  - Movl \$0,%ebx
  - Int \$0x80

# An Example of passing input values to a system call

```
.section .data
output:
.ascii "This is a test message.\n"
output_end:
.equ len, output_end - output
.section .text
.globl _start
_start:
    movl $4, %eax
    movl $1, %ebx
    movl $output, %ecx
    movl $len, %edx
    int $0x80
    movl $1, %eax
    movl $0, %ebx
    int $0x80
```

- **EAX register is used to hold the system call value.**
- **EBX: The integer file descriptor**
- The file descriptor value for the output location is placed in the EBX. Linux systems contain three special file descriptors:
  - **0 (STDIN): The standard input for the terminal device (normally the keyboard)**
  - **1 (STDOUT): The standard output for the terminal device (normally the terminal screen)**
  - **2 (STDERR): The standard error output for the terminal device (normally the terminal screen)**
- **ECX: The pointer (memory address) of the string to display**
- **EDX: The size of the string to display.**

# Return values from system calls

- Return value of system calls is implemented similar to function calls.
- System call return their values in eax register.
- There are two kinds of values
  - A) non negative
  - B) negative
- All non negative values returned by the system call represents successful execution of system call, whereas negative values represents error conditions.
- For write system call, it returns the size of the string written to the file descriptor.

# Return values from system calls

System Call Value	System Call	Description
20	getpid	Retrieves the process ID of the running program
24	getuid	Retrieves the user ID of the person running the program
47	getgid	Retrieves the group ID of the person running the program

# An example of getting a return value from a system call

```
.section .bss
.lcomm pid, 4
.lcomm uid, 4
.lcomm gid, 4

.section .text
.globl _start
_start:
    movl $20, %eax
    int $0x80
    movl %eax, pid
    movl $24, %eax
    int $0x80
    movl %eax, uid

    movl $47, %eax
    int $0x80
    movl %eax, gid
end:
    movl $1, %eax
    movl $0, %ebx
    int $0x80
```

```
(gdb) x/d &pid
0x80490a4 <pid>:      4758
(gdb) x/d &uid
0x80490a8 <uid>:      501
(gdb) x/d &gid
0x80490ac <gid>:      501
```

# Starting a process in linux

- In Linux, a program is executed by creating a process and loading a program in newly created process.
- A **fork** system call is used to create a process and a program is loaded using **execve** system call.
- When a program is loaded , it run from memory location whose address is specified in executable file.
- By default starting address of the program is indicated by label **\_start**

# System call related to process information

System call name	System call value	description
Getuid	24	Retrieves user id of a person running the program
Getpid	20	Retrieves process id of running program.
Getgid	47	Retrieves group id of a person running the program.

# Strace program

- The **strace** program intercepts system calls made by a program and displays them for you.

```
$ strace ./syscalltest2
execve("./syscalltest2", ["./syscalltest2"], /* 38 vars */) = 0
getpid()                      = 7616
getuid()                       = 501
getgid()                       = 501
_exit(0)                      =
$
```

- This output shows all of the system calls the syscalltest2 program made, in the order in which they were performed by the application.
- The left side shows the system call names, and the right side shows the return values produced by the system calls

# System call related to process management

System call name	System call value	description
Fork		Create a child process
Execve		Execute a program
Exit		Terminate the current process
Nice		Change the process priority.
Kill		Send a signal to kill a process.

# Thank you

SJMurchite \_ SYB\_JUnit6 syllabus