



CS202 – System Software


Dr. Manish Khare



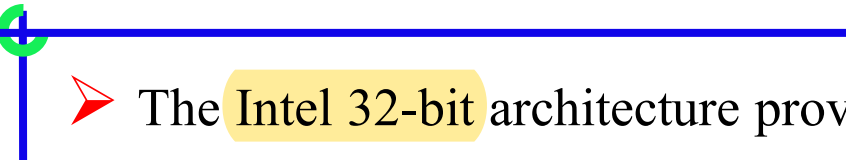
Assembly Programming Language

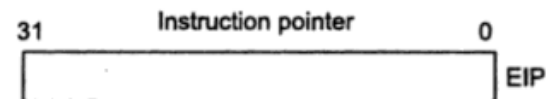
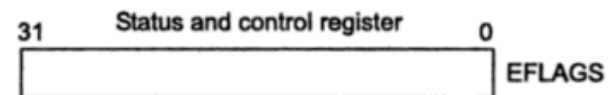
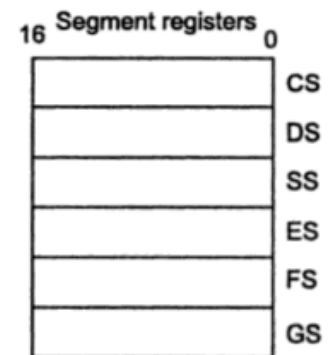
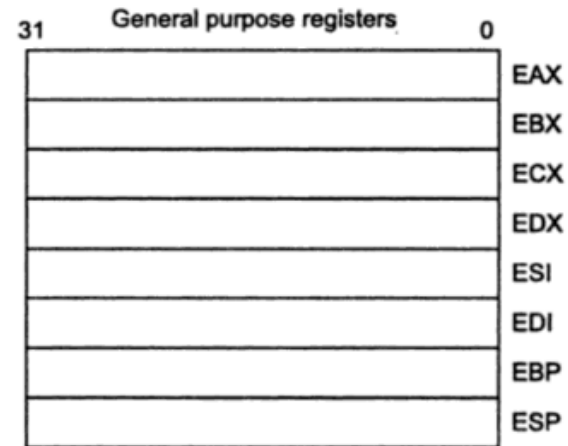
- The design of an assembler needs a detailed knowledge about the underlying processor and its instruction set.
- To make the concepts about system softwares clear, first we will discuss an overview of Pentium assembly language.
- The instruction set of Pentium is really huge; hence, we will be discussing only a subset of it.
- The program written in the assembly language needs to be converted into machine code by means of an assembler.

Pentium Assembly Language

- 
- Registers
 - Memory Models
 - Addressing Modes
 - Instruction Set
 - Instruction format

Registers

- 
- The Intel 32-bit architecture provides 16 basic program execution registers for use in general system and application programming.
 - General purpose registers: Eight general purpose registers are available for storing operands and pointers
 - Segment registers: Six segment registers hold up to six segment selectors.
 - EFLAGS registers: It holds the program status and allows limited control of the processor to the application program.
 - EIP: It is the 32-bit instruction pointer pointing to the next instruction to be executed



General Purpose Registers

➤ The eight 32-bit general purpose registers EAX, EBX, ECX, EDX, ESI, EDI, EBP, and ESP are provided for the following operations:

- To hold operands for operations
- To hold operands for address calculations
- For memory pointers



➤ The special uses of general purpose registers by instructions are as follows:


- EAX - Accumulator for operands and results data.
- EBX - Pointer to data in the data segment.
- ECX - Counter for string and loop operations.
- EDX - I/O pointer.
- ESI - Pointer to data in the segment pointed to by the DS segment register; source pointer for string operations.
- EDI - Pointer to data (or destination) in the segment pointed to by the segment register ES; destination pointer for string operations.
- EBP - Pointer to data on the stack (in the SS segment).
- ESP - Stack pointer in the segment pointed to by the segment register SS.

- The lower order 16 bits of the general purpose registers map directly to the register set found in the earlier Intel processors, like 8086.
- These 16-bit registers can be referenced with the names AX, BX, CX, DX, BP, SP, SI, and DI. The 8-bit portions of AX, BX, CX, and DX can be accessed as a collection of 8-bit registers.
- The higher order 8-bits of AX is referred to as AH, while the lower order 8-bits can be accessed as AL. Similarly, BX, CX, and DX can be accessed as BH and BL, CH and CL, and DH and DL registers.

31	16	15	8	7	0	16 bit	32 bit
	AH		AL			AX	EAX
	BH		BL			BX	EBX
	CH		CL			CX	ECX
	DH		DL			DX	EDX
							EBP
							ESI
							EDI
							ESP

Segment Register

- The segment registers (CS, DS, SS, ES, FS, and GS) hold 16-bit segment selectors.
- A segment selector is a special pointer that identifies a segment in memory.
- To access a particular segment in memory, the segment selector for that segment must be present in the appropriate segment register.
- Each of the segment registers is associated with one of the three types of storage: code, data, or stack.

- 
- The DS, ES, FS, and GS registers point to four data segments. The availability of four data segments permits efficient and secure access to different types of data. To access additional data segments, the application program must load segment selectors for these segments into the corresponding segment register.
 - The SS register contains the segment selector for a stack segment. Unlike CS register, SS register can be loaded explicitly, permitting application programs to set up multiple stacks and switch between them.

EFLAGS register

- The 32-bit EFLAGS register contains a group of status flags, a control flag, and a group of system flags. Some of the flags in the EFLAGS register can be modified directly, using special purpose instructions. However, there are no instructions that allow the whole register to be examined or modified directly.

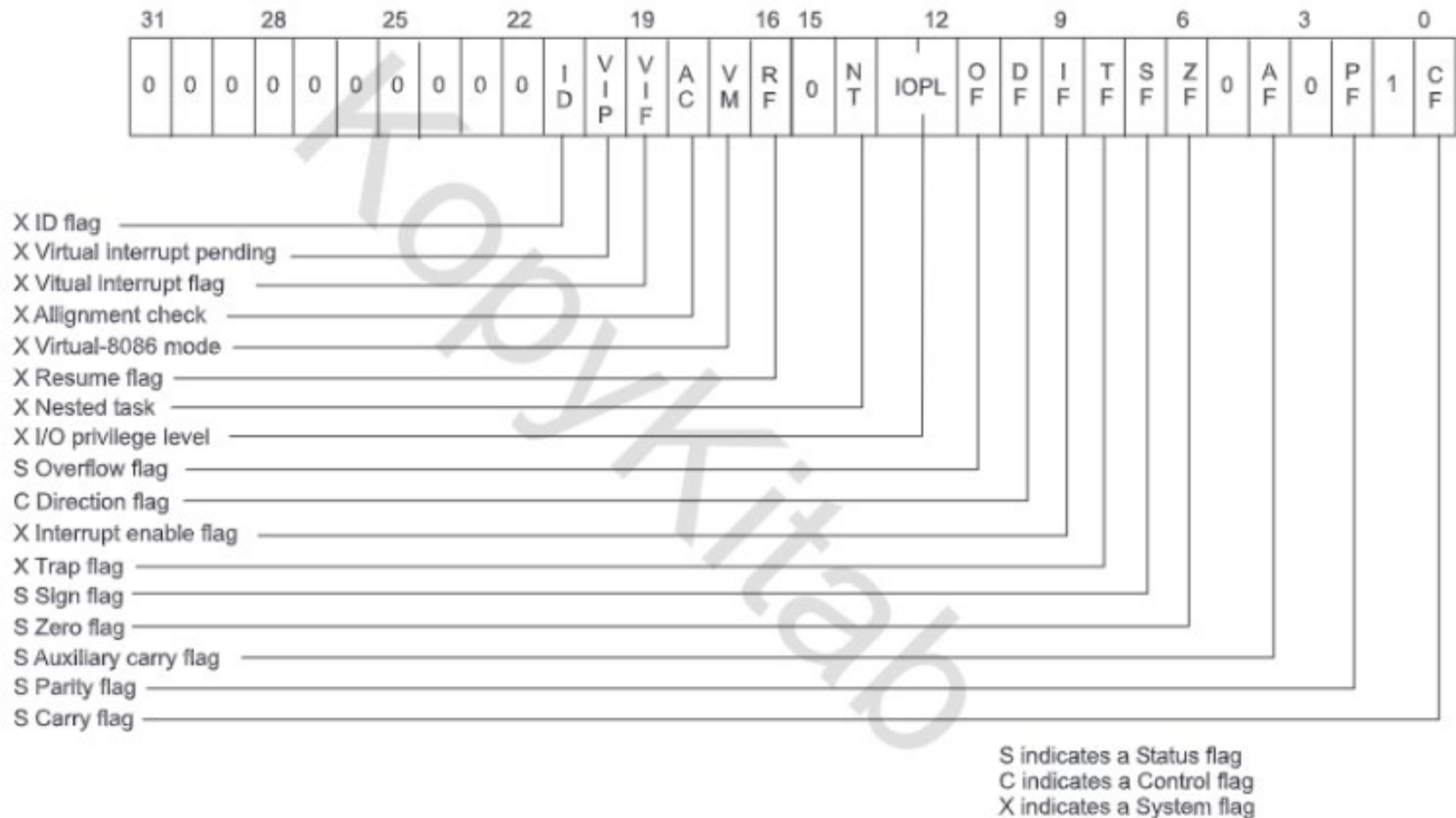


FIGURE 2.3 EFLAGS register.

Memory Model

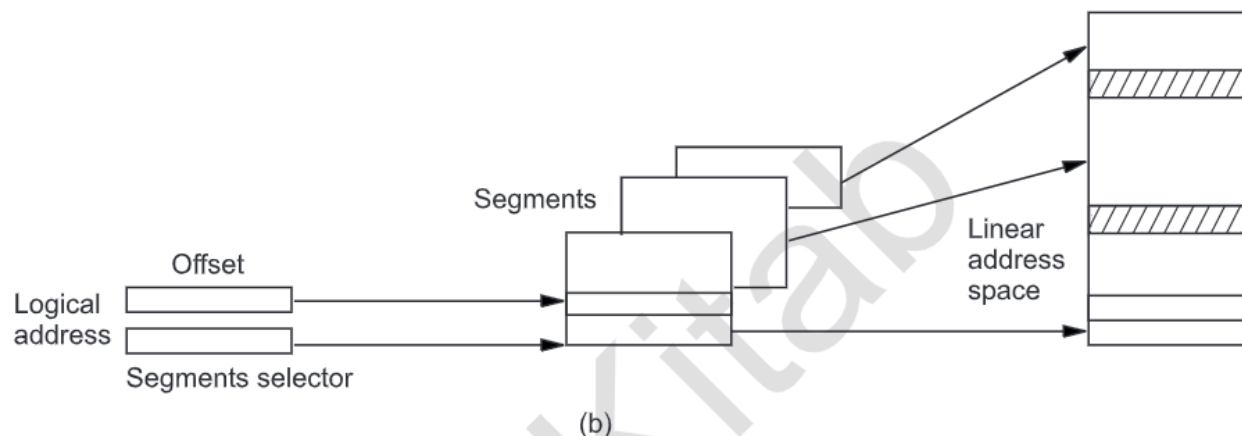
- The physical memory is organized as a sequence of bytes. The physical address space ranges from zero to $2^{36} - 1$ (64 GBytes).
- Intel 32-bit (IA32) processors provide memory management facilities such as segmentation and paging. Thus, the programs do not address the physical memory directly. Memory is accessed using any of the three memory models:
 - Flat mode
 - Segmented mode
 - Real address mode

- In flat memory model, memory appears as a continuous address space. This linear address space is byte addressable with addresses running continuously from 0 to $2^{32} - 1$



(a)

- With the segmented memory model, memory appears to a program as a group of independent address spaces called segments. When using this model, code, data and stack are typically contained in separate segments.
- To address a byte in a segment, the program must issue a logical address, which consists of a segment selector and an offset.
- The segment selector identifies the segment to be accessed and the offset identifies a byte in the address space of the segment.
- The programs can address up to 16,383 segments of different sizes and types, and each segment can be as large as 2^{32} bytes.



- The real address mode memory model uses the memory model for Intel 8086 processor. Individual segments can be up to 64 KBytes in size, and maximum size of the linear address space is 2^{20} bytes. The linear address is obtained by multiplying the segment register by 16 and adding offset to it.



- Modern operating systems and applications use the flat (that is, unsegmented) memory mode: all the segment registers are loaded with the same segment selector (zero) so that all memory references a program makes are to a single linear-address space.

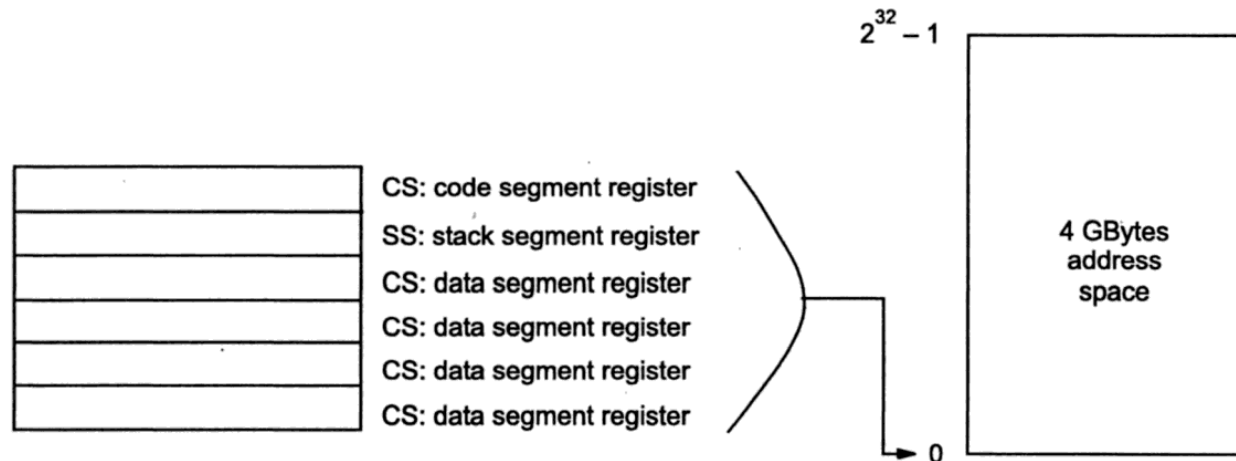


FIGURE 2.5 Segment registers in flat memory model.

Segmented Memory Model

- In this mode, the segment registers receive the name selectors. Although the segment registers are still 16-bit in size, their interpretation by the processor is different. The structure of a selector has been shown below.



RPL : Requestor privilege level
TI : Table indicator
Index : Index into descriptor table

- Instead of using the segment registers to extend it to 20 bits, the processor treats each selector as an index to a Descriptor Table.

Descriptor Tables

- Descriptor tables reside in system memory and are used by the processor to perform address translation. Each entry in a descriptor table is 8 bytes long and represents a single segment in memory.
- It contains a pointer to the first byte in the associated segment and a 20-bit value representing the size of the segment in memory.

Base 24–31	G	D / B	0	AVL	Segment limit 16–19	P	D P L	S	Type	Base 16–23
Base address 0–15						Segment limit 0–15				

AVL: Available for use by operating system

Base: Segment base address

D/B: Default segment size (16/32 bits)

DPL: Descriptor privilege level

G: Granularity

Limit: Segment limit

P: Present bit

S; Descriptor type (system/application)

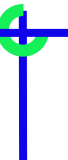
Type: Segment type


FIGURE 2.7 Descriptor entry.



➤ The individual fields are as follows:

- 1. Base: This 32-bit segment base address field points to the segment's starting location in the 4GB linear address space.
- 2. D/B: It is called segment size bit. When the descriptor entry describes a code segment, this bit is used to specify the default length of operands and addresses. If the bit is set, the processor assumes a 32-bit segment, else a 16-bit segment is assumed.
- 3. DPL: Descriptor privilege level (2-bit) field defines the segment privilege level. It is used by the protection mechanism built into the processor to restrict access to the segment.
- 4. G: Granularity bit controls the resolution of the segment limit field. When this bit is clear, the resolution is set to one byte. When this bit is set, the resolution is 4KB.

- 
- 5. Limit: The segment limit field (20-bit) determines the size of the segment in units of one byte or 4KB (depending on the G-bit).
 - 6. P: The segment present bit specifies whether the segment is present in memory.
 - 7. S: The descriptor type bit determines whether this is a normal segment or a system segment
 - 8. Type: The segment type bit (4-bit) determines the type of the segment, for example, execute-only, execute-read, read-only, read-write, and so on.



➤ Two types of descriptor tables are used by the processor when working in the protected mode.

- The first one is known as the Global descriptor table (GDT) and is used mainly for holding descriptor entries of operating system segments.
- The second type is known as the Local descriptor table (LDT). It contains entries of normal application segments. During initialization, the kernel creates a single GDT which is kept in memory until either the operating system terminates or the processor is switched to the real-mode.

- Whenever the user starts an application, the operating system creates a new LDT to hold the descriptor entries, which represent the segments used by the new task.
- When looking into a specific descriptor entry, the addressing unit in the processor uses the TI bit to decide which descriptor table be used—
 - the GDT or the currently active LDT. The linear address and size of the GDT are stored in a special processor register called GDTR. Similarly, LDTR register contains the size and position of the currently active LDT in memory.

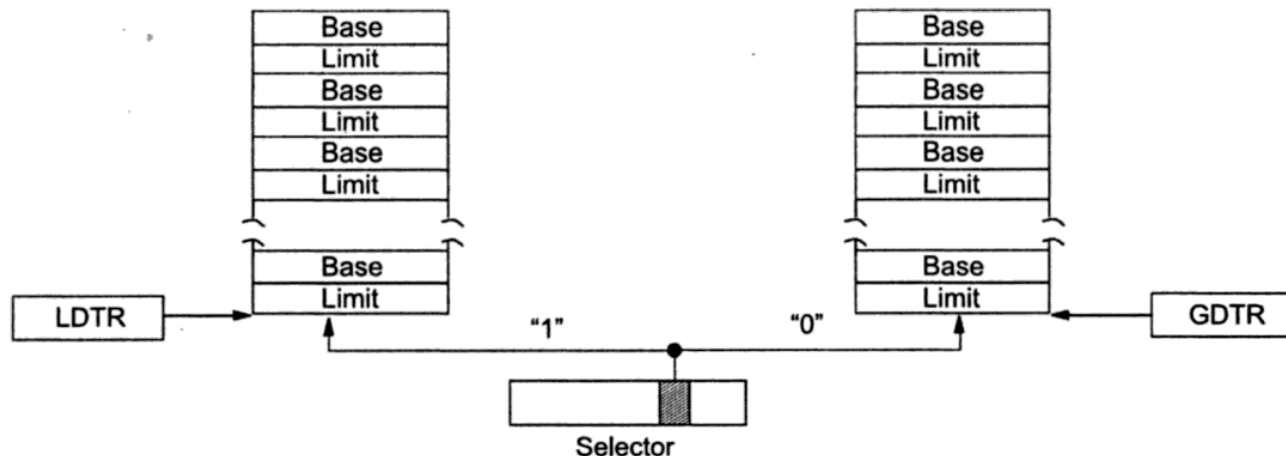


FIGURE 2.8 Table indicator bit.

- The logical address generated by the CPU consists of two parts
 - a selector part, and an offset part.
- The segment selector points to a segment descriptor, which contains the base address of a memory segment.
- The 32-bit offset from the logical address is added to the segment's base address, generating 32-bit linear address.
- It has been illustrated here.

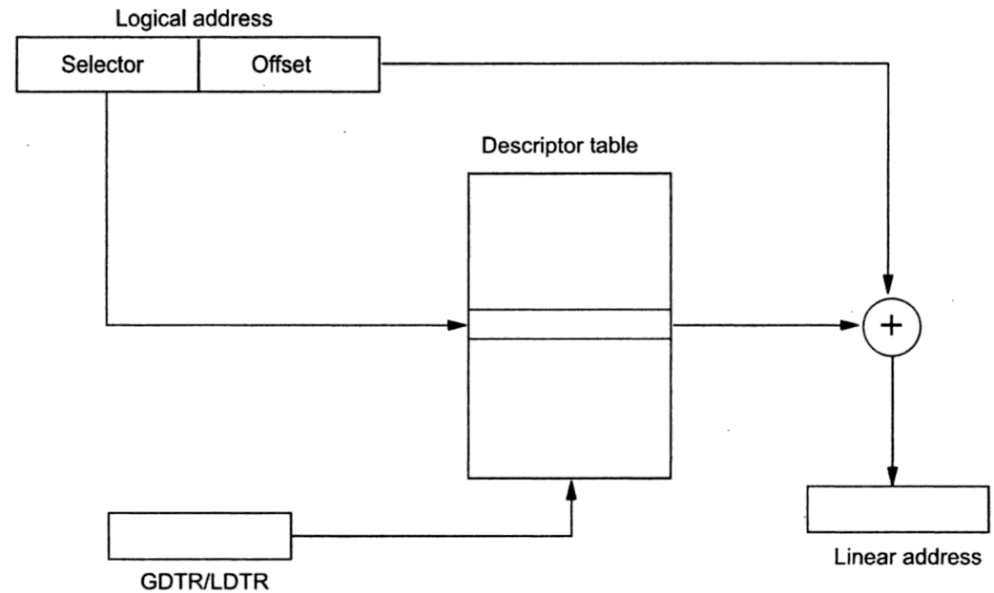


FIGURE 2.9 Logical to linear address translation.

Addressing Mode

➤ Intel architecture provides a total of seven distinct addressing modes

- Register
- Immediate
- Direct
- Register indirect
- Based relative
- Indexed relative
- Based indexed relative

Register addressing mode

- It involves the usage of two registers to hold the data to be manipulated.
- Since the registers reside on the same chip as the processor, register addressing provides the fastest means of data manipulation.
- No memory access is involved in the process.
- For example,
 - MOV BX, DX ; copy contents of register BX into register DX
 - MOV ES, AX ; copy contents of register AX into segment register ES

Immediate addressing mode

- In this mode, the source operand is a constant. After assembly, the operand comes immediately after the opcode. This addressing mode can be used to load into any of the registers excepting the segment registers and the flag register.
- For example,
 - `MOV AX, 2550h` ; move hexadecimal value 2550 into register AX.
- It may be noted that the segment registers can be loaded by transfer from general purpose registers using the register addressing mode. The following are two examples of it.
- Real-address mode:
 - `MOV AX, 2550h` ; AX contains the segment address
 - `MOV DS, AX` ; load DS segment register with AX
- Protected mode:
 - `MOV AX, 08h` ; AX contains the segment selector
 - `MOV DS, AX` ; load DS segment register with AX

Direct addressing mode

- In this case, the data is located in the memory. The address of the memory location comes immediately after the instruction. It may be noted that in immediate addressing, operand is provided itself with the instruction, whereas in direct addressing, the address of the operand is found in the instruction.
- The following are two examples of direct addressing used in real and protected mode respectively.
- Real-address mode:
 - `MOV DL, [2400h]` ; moves content of memory location $16 \times DS + 2400h$ to DL
- Protected mode:
 - `MOV EAX, [12345678h]`; segment selector in DS indexes into the descriptor table to get the base of the data segment to which 12345678h is added to obtain the logical address content of this location is transferred to EAX

Register indirect addressing mode

- The address of the memory location where the operand resides is held in a register.
- In real-address mode, the register SI, DI, and BX can be used for this purpose.
- In the protected mode, any of the registers EAX, EBX, ECX, EDX, ESI, or EDI can be used.
- These register values are combined with DS segment register to generate the physical address. For example,
- Real-address mode:
 - `MOV AL, [BX]` ;moves into AL contents of memory location pointed to by DS:BX
- Protected mode:
 - `MOV AL, [EAX]`; moves into AL the contents of the memory location pointed to by DS:EAX

Based relative addressing mode

- In this case, in real-address mode, the base registers BX and BP, as well as a displacement value are used to compute an effective address, which, combined with the segment register value, gives the address of the operand.
- The default segment registers used for the calculation of physical address are DS for BX and SS for BP.
- For Example,
 - `MOV CX, [BX + 10h];` moves 16-bit word at memory location DS:BX + 10h into CX register
 - `MOV [BP + 22h], CX;` moves contents of CX into memory location SS:BP + 22h
- In case of protected mode, registers EAX, EBX, ECX, EDX, ESI, EDI, EBP, and ESP can be used as base registers. The default segment registers used for the calculation of physical address are DS for EAX, EBX, ECX, EDX, ESI, and EDI and SS for EBP and ESP.
- For example,
 - `MOV EAX, [EBX + 10h];` moves 32-bit quantity at memory location DS:EBX + 10h into EAX register
 - `MOV [EBP + 22h], EDX;` moves contents of EDX into memory location SS:EBP + 22h

Indexed relative addressing mode

- In real-address mode, index registers SI and DI, as well as a displacement value are used to calculate effective address. The default segment register is DS.
- For example,
 - `MOV CX, [SI+10h]` ; moves 16-bit word at memory location DS:SI + 10h into CX register
- In case of protected mode, registers EAX, EBX, ECX, EDX, ESI, EDI, EBP, and ESP can be used as index registers. The default segment registers used are DS for EAX, EBX, ECX, EDX, ESI and EDI, and SS for EBP and ESP.
- For example,
 - `MOV EAX, [ESI+10h]` ; moves 32-bit quantity at memory location DS:ESI + 10h into EAX register

Based indexed relative addressing mode

- It is the combination of based and indexed addressing modes.
- In this mode, one base register and one index register are used.
- For example,
- Real-address mode:
 - `MOV AL, [BX + SI + 8h]` ; moves into AL contents of memory location pointed to by `DS:BX + SI + 8h`
 - `MOV AL, [BP + SI + 8h]` ; moves into AL contents of memory location pointed to by `SS:BP + SI + 8h`
- Protected mode:
 - `MOV AL, [EAX + EBX + 10h]` ; moves into AL the contents of the memory location pointed to by `DS:EAX + EBX + 10h`
 - `MOV AL, [EBP + EBX + 10h]` ; moves into AL the contents of the memory location pointed to by `SS:EBP + EBX + 10h`

Segment overrides



- Each of the addressing modes has a default segment register associated with it. However, it is possible to explicitly mention in the statement some other segment register to be used to access the operands.
- It is called segment overrides.
- For example, the following instruction asks to use ES as the segment register, though the default segment register is DS.
 - `MOV AL, ES:[SI + 8h]` ; moves into AL contents of memory location pointed to by `ES:SI + 8h`

Scaling factors

- For 32-bit instructions, a scaling factor can be specified in any of the based, indexed, or based indexed addressing modes.
- The scaling factor is 1, 2, 4, or 8.
- For example,
 - `MOV EAX, [ESI + EAX*4]` ; moves into EAX the 32-bit quantity at $DS:ESI + EAX * 4$
- Thus in the protected mode, the effective address can be viewed as,
 - $\text{effective address} = \text{base register} + (\text{index register} \times \text{scaling factor}) + \text{displacement}$

Instruction Set

- Most of the instructions have exactly two operands, in which case one of the operands must be a register, while there is no restriction on the other one – it can be another register, memory location, or immediate value.
- Most of the instructions are available in 8-, 16-, and 32-bit operands.
- Some most common abbreviations in instruction set.
 - reg : register mode operand, 32-bit register
 - reg8 : register mode operand, 8-bit register
 - r/m : general addressing mode, 32-bit
 - r/m8 : general addressing mode, 8-bit
 - immmed : 32-bit immediate along with instruction
 - immmed8 : 8-bit immediate along with instruction
 - m : symbol (label) in the instruction is the effective address



The set of instructions can be classified into a number of groups based on their functionality.

- Data movement
- Integer arithmetic
- Logical
- Floating point
- Control transfer

Data Movement

- **MOV** reg, r/m ; copy data
 r/m, reg
 reg, immed
 r/m, immed
- **MOVSX** reg, r/m8 ; sign extend and copy data
- **MOVZX** reg, r/m8 ; #zero extend and copy data
- **LEA** reg, m ; get effective address

Example

- places 32-bit 2's complement immediate 23 into register EAX.
 - `MOV EAX, 23`

- sign extends the 8-bit quantity in AL to 32-bits and places it in ECX.
 - `MOVSX ECX, AL`

- places -1 into memory, address given by contents of ESP.
 - `MOV [ESP], -1`

- put the address assigned to label `loop_top` into register EBX.
 - `LEA EBX, loop_top`

Integer Arithmetic

- **ADD**
reg, r/m ; 2's complement addition
r/m, Teg
reg, immmed
r/m, immmed
- **INC**
reg ; add 1 to operand
r/m
- **SUB**
reg, r/m ; 2's complement subtraction
r/m, reg
reg, immmed
r/m, immmed
- **DEC**
reg ; subtract 1 from operand
r/m
- **NEG**
r/m ; get additive inverse of operand

➤	MUL	EAX, t/m	; unsigned multiplication, $EDX \parallel EAX \leftarrow EAX * r/m$; 64-bit results stored in the EDX, EAX pair
➤	IMUL	r/m	; 2's complement multiplication ; $EDX \parallel EAX \leftarrow EAX * r/m$
		reg, r/m	; $reg \leftarrow reg * r/m$
		reg, imm8	; $reg \leftarrow reg * imm8$
➤	DIV	r/m	; unsigned division, does $EDX \parallel EAX / r/m$; $EAX \leftarrow$ quotients, $EDX \leftarrow$ remainder
➤	IDIV	r/m	; 2's complement division, does $EDX \parallel EAX / r/m$; $EAX \leftarrow$ quotients, $EDX \leftarrow$ remainder
➤	CMP	reg, r/m	; sets EFLAGS based on
		r/m, imm8	; second operand — first operand
		r/m8, imm8	
		r/m, imm8	; sign extends imm8 before subtract

Example

- takes the double word at the address $EAX + 4$ and finds its additive inverse, then planes the additive inverse back at that address
 - `NEG [BAX + 4]`
- adds one to the contents of register ECX, and result is stored in ECX.
 - `INC ECX`

Logical Instruction

➤ NOT r/m ; logical NOT

➤ AND reg, r/m ; logical AND

reg8, r/m8

r/m, reg

r/m8, reg8

r/m, imm8

r/m8, imm8

➤ OR reg, r/m ; logical OR.

reg8, r/m8

r/m, reg

r/m8, reg8

r/m, imm8

r/m8, imm8



XOR reg, r/m ; logical exclusive OR

reg8, r/m8

r/m, reg

r/m8, reg8

r/m, immed

r/m8, immed8



TEST r/m, reg ; logical AND to set EFLAGS

r/m8, reg8

r/m, immed

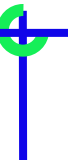
r/m8, immed8

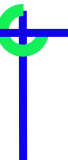
Control Instructions

➤ JMP	m	; unconditional jump
➤ JG	m	; Jump if greater than 0
➤ JGE	m	; Jump if greater than or equal to 0
➤ JL	m	; Jump if less than 0
➤ JLE	m	; Jump if less than or equal to 0

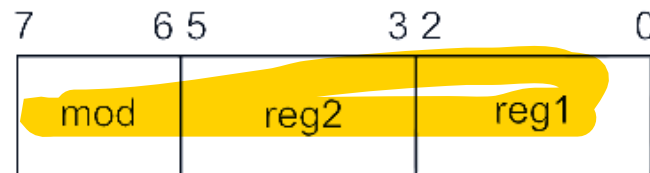
Instruction Format

- The instructions in an Intel processor vary in size from one byte up to fourteen bytes. However, all these instructions follow a six-part structure noted below.
 - Prefix – 0 to 4 bytes
 - Opcode – 1 to 2 bytes
 - ModR/M – 1 byte
 - SIB – 1 byte
 - Displacement – 1 byte or word
 - Immediate – 1 byte or word
- Except for the opcode, all other parts are optional.
- Simple instructions such as NOP (No Operation) require Just the opcode.
- Complicated instructions, such as `ADD [GS:mem_data+EBX+ESI*8], WORD 1003H`, require all of the fields.

- 
- Prefixes. The optional prefixes modify the behavior of instructions in several ways. Each prefix adds one byte to the instruction. An instruction can have one prefix from each of the four prefix groups, for a maximum of four prefix bytes.
 - Group 1: LOCK, REPE/REPZ, REP, REPNE/REPZ—the prefix bytes for these are FoH through F3H. The REP prefixes are used only with string instructions.
 - Group 2: Segment overrides prefixes. The hexadecimal codes 2E, 36, 3E, 26, 64, and 65 are used to override the default segment registers by CS, SS, DS, ES, FS, and GS respectively.
 - Group 3: Operand-size override (16-bit vs. 32-bit) the prefix byte is 66H.
 - Group 4: Address-size override (16-bit vs. 32-bit) the prefix byte is 67H.

- 
- Opcode. The operation code or opcode comes after at the most four number of prefix bytes.
 - The opcode field is one or two bytes. The code tells the processor which instruction to execute. Apart from the operation, it also contains bit-fields identifying the type and size of operands expected.
 - For example, the NOT operation has an opcode 1111011w.
 - Here, the 'w' bit determines whether the operand is a byte or a word.
 - Similarly, the OR instruction has the format 000010dw. Here the 'd' bit determines the direction of dataflow—that is, which operand is the source and which one is the destination.

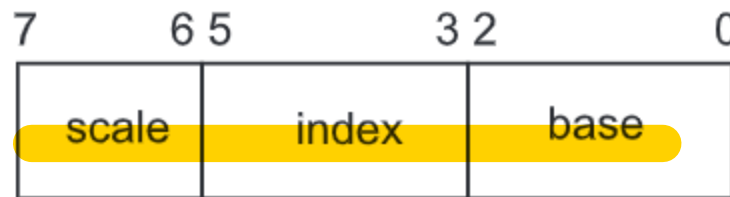
- ModR/M: It is 0 or 1 byte long and is put if the instruction requires it. If present, the ModR/M field comes just after the opcode. This byte tells the processor which registers or memory locations to use as the instruction's operands. The byte has the structure shown in shown below.



- Both reg1 and reg2 fields take three-bit register codes, indicating which registers to be used as operands. By default, reg1 is the source operand and reg2 is the destination. However, it may be overridden by some instructions like OR, using the direction bit there. If an instruction requires only one operand, the unused reg2 field holds extra opcode bits.
- The mod field determines the meaning of the reg1 field.

Code	Meaning
00	Operand's memory address is in reg1
01	Operand's memory address is reg1 + a byte-sized displacement
10	Operand's memory address is reg1 + a word-sized displacement
11	Operand is reg1 itself

- SIB. In 32-bit addressing, for mod = 00, 01, or 10, when reg1 field indicates ESP register, an additional byte follows the ModR/M byte—known as SIB byte.
- SIB is the acronym for Scale % Index + Base.
- It provides a powerful addressing mode in 32-bit that uses a combination of two registers and a scaling factor instead of reg1 for the operand address.
- The SIB byte structure is shown as



- In SIB bytes, both index and base are three bit registers codes, and scale is a two-bit number. To compute the SIB value, the processor uses the formula: $(\text{index} * 2^{\text{scale}}) + \text{base}$




➤ This value is used instead of reg1 field to access the memory.

➤ For example, the ModR/M and SIB bytes for the memory address such as “[EBX*4 + ESI + displ]” is as follows.

- ModR/M.mod = 10 (that is [reg1 + word1])
- ModR/M.reg2 = the destination register
- ModR/M.reg1 = ESP
- SIB.scale = 2
- SIB.index = EBX
- SIB.base = ESI

- When $\text{mod} = 01$ or 10 , a displacement is a part of the operand's address.
- The displacement may be a byte or a word.
- For example, the full machine code for the 32-bit instruction `OR EAX, [ECX + EDX*2 + 406080A0h]` is shown below

Opcode	ModR/M	SIB	Displacement
00001011	10 000 100	01 010 001	10100000 10000000 01100000 01000000

- 
- Immediate: If an instruction uses immediate value as an operand, the immediate value is the last part of the instruction.
 - The machine code for 16-bit instruction, AND SI, 0420h

Opcode	ModR/M	Immediate
10000001	11 100 110	00100000 00000100

Assembler Design

- Assembler is the tool to convert an assembly language program into machine language one, understandable by the processor that executes it.
- The complexity of the process depends upon various factors, including size of the instruction set, different addressing modes supported, length of program being translated, etc.