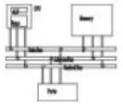
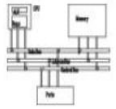
Intel IA-32 Family



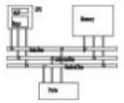
- Intel386 (1985)
  - 4 GB addressable RAM
  - 32-bit registers
  - paging (virtual memory)
  - Up to 33MHz
- Intel486 (1989)
  - instruction pipelining
  - Integrated FPU
  - 8K cache
- Pentium (1993)
  - Superscalar (two parallel pipelines)

IA32 Processors



- Totally Dominate Computer Market
- Evolutionary Design
  - Starting in 1978 with 8086
  - Added more features as time goes on
  - Still support old features, although obsolete
- Complex Instruction Set Computer (CISC)
  - Many different instructions with many different formats
    - But, only small subset encountered with Linux programs
  - Hard to match performance of Reduced Instruction Set Computers (RISC)
  - But, Intel has done just that!

General-purpose registers



32-bit General-Purpose Registers

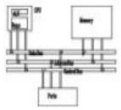
EAX	EBP
EBX	ESP
ECX	ESI
EDX	EDI

EFLAGS
EIP

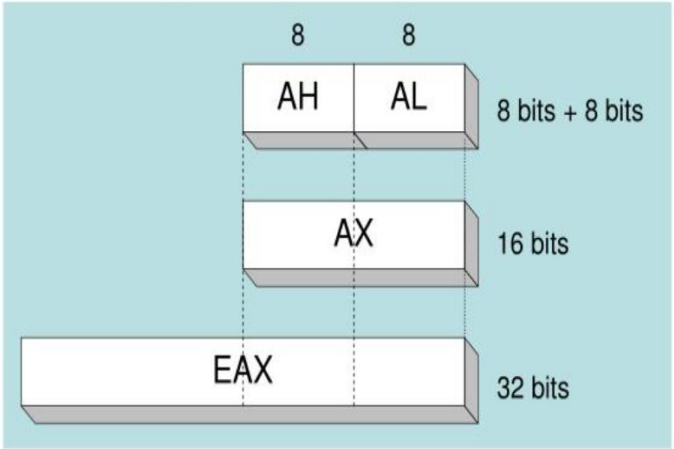
16-bit Segment Registers

CS	ES
SS	FS
DS	GS

Accessing parts of registers



- Use 8-bit name, 16-bit name, or 32-bit name
- Applies to EAX, EBX, ECX, and EDX



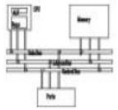
32-bit	16-bit	8-bit (high)	8-bit (low)
EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL

Index and base registers

- Some registers have only a 16-bit name fo
  their lower half (no 8-bit aliases). The 16-
  registers are usually used only in real-addr
  mode.

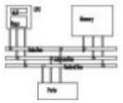
32-bit	16-bit
ESI	SI
EDI	DI
EBP	BP
ESP	SP

Some specialized register uses (1 of 2)



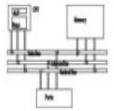
- General-Purpose
  - EAX – accumulator (automatically used by division and multiplication)
  - ECX – loop counter
  - ESP – stack pointer (should never be used for arithmetic or data transfer)
  - ESI, EDI – index registers (used for high-speed memory transfer instructions)
  - EBP – extended frame pointer (stack)

Status flags



- Carry
  - unsigned arithmetic out of range
- Overflow
  - signed arithmetic out of range
- Sign
  - result is negative
- Zero
  - result is zero
- Auxiliary Carry
  - carry from bit 3 to bit 4
- Parity
  - sum of 1 bits is an even number

Programmer's model



Basic Program Execution Registers

Eight 32-bit Registers — General-Purpose Registers

Six 16-bit Registers — Segment Registers

32-bits — EFLAGS Register

32-bits — EIP (Instruction Pointer Register)

FPU Registers

Eight 80-bit Registers — Floating-Point Data Registers

16 bits — Control Register

16 bits — Status Register

16 bits — Tag Register

Address Space*

$2^{32}-1$

0

*The address space can be flat or segmented. Using the physical address extension mechanism, a physical address space of $2^{36}-1$ can be addressed.

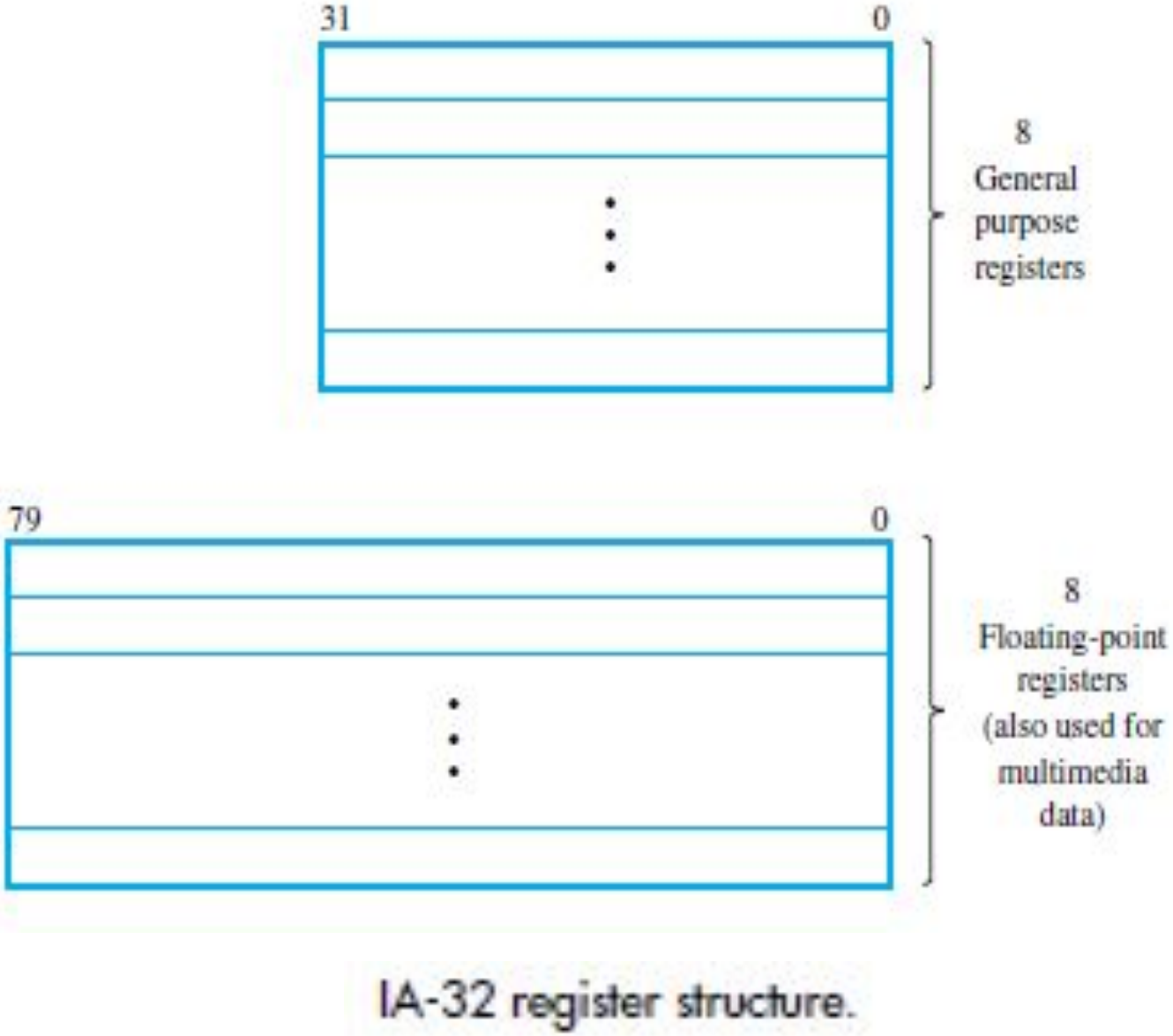
Intel IA-32 Architecture

Memory Organization

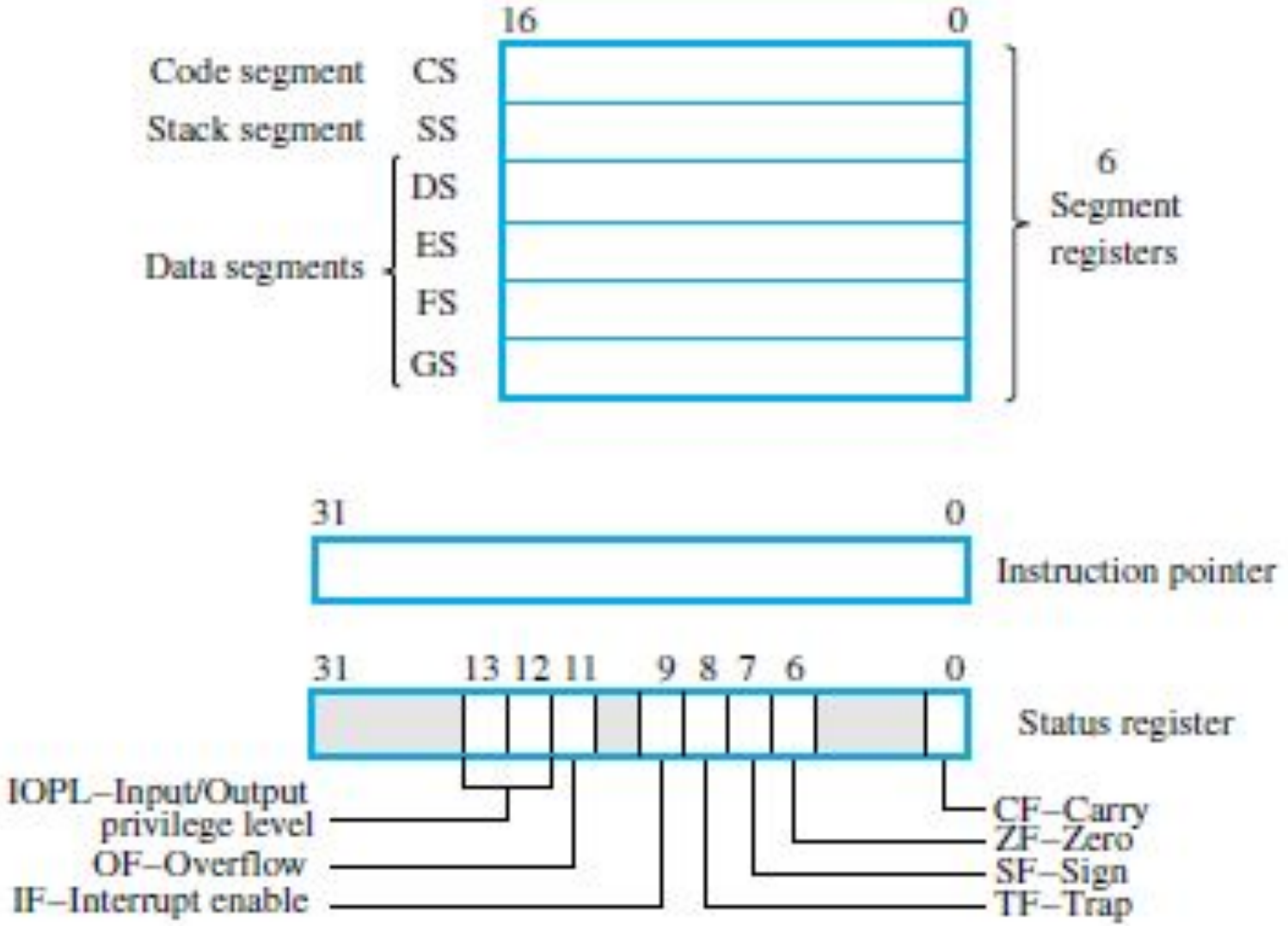
- IA-32 architecture, memory is byte-addressable using 32-bit addresses, and instructions typically operate on data operands of 8 or 32 bits.
- Operand sizes are called byte and doubleword in Intel terminology.
- 16-bit operand was called a word in earlier 16-bit Intel processors
- A larger 64-bit operand size called a quadword for double-precision floating-point numbers and packed integer data.
- Multiple-byte data operands may start at any byte address location.
- They need not be aligned with any particular address boundaries in the memory.

Register Structure

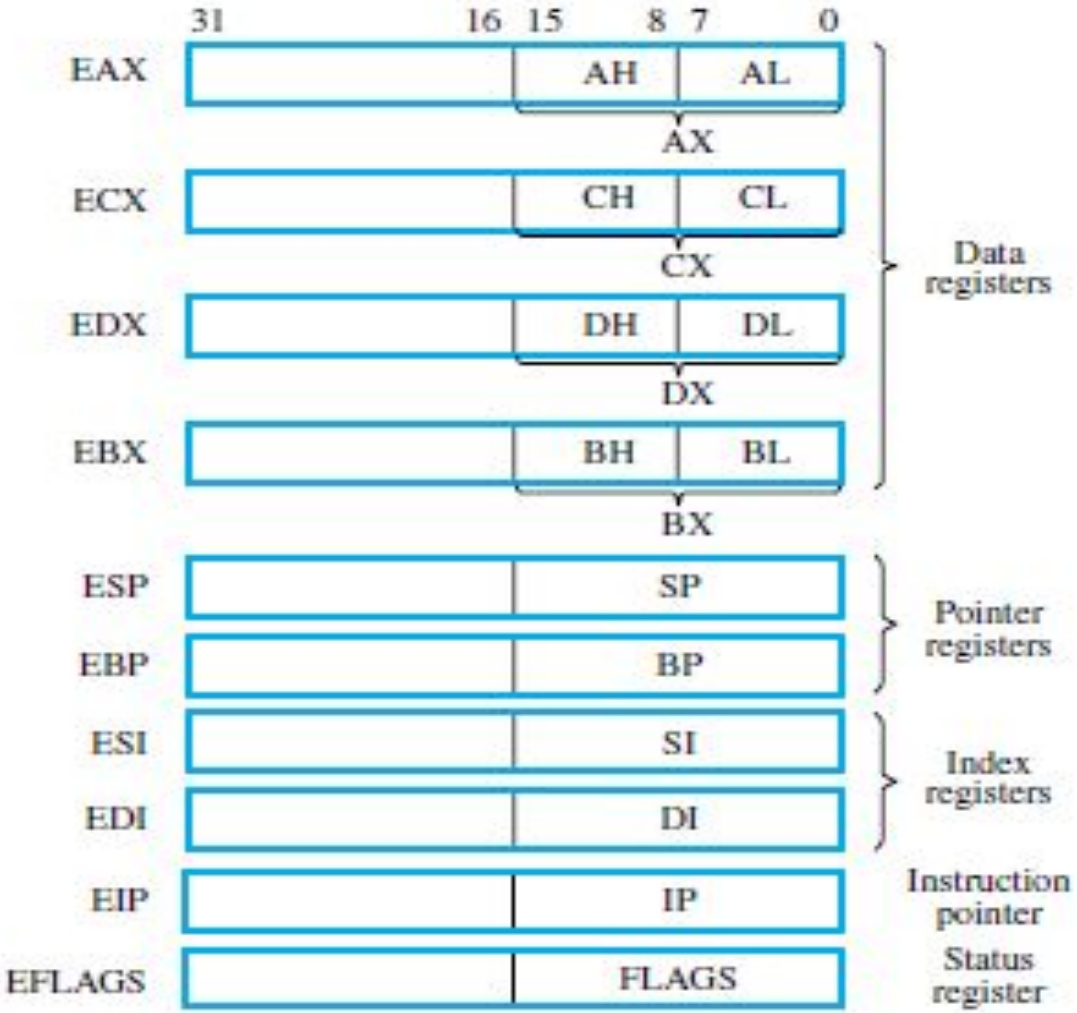
- IA-32 architecture has different models for accessing the memory.
- Segmented memory model associates different areas of the memory, called segments, with different usages.
- Code segment holds the instructions of a program.
- Stack segment contains the processor stack, and four data segments are provided for holding data operands. (DS, ES, FS, GS)
- Six segment registers contain selector values that identify where these segments begin in the memory address space.



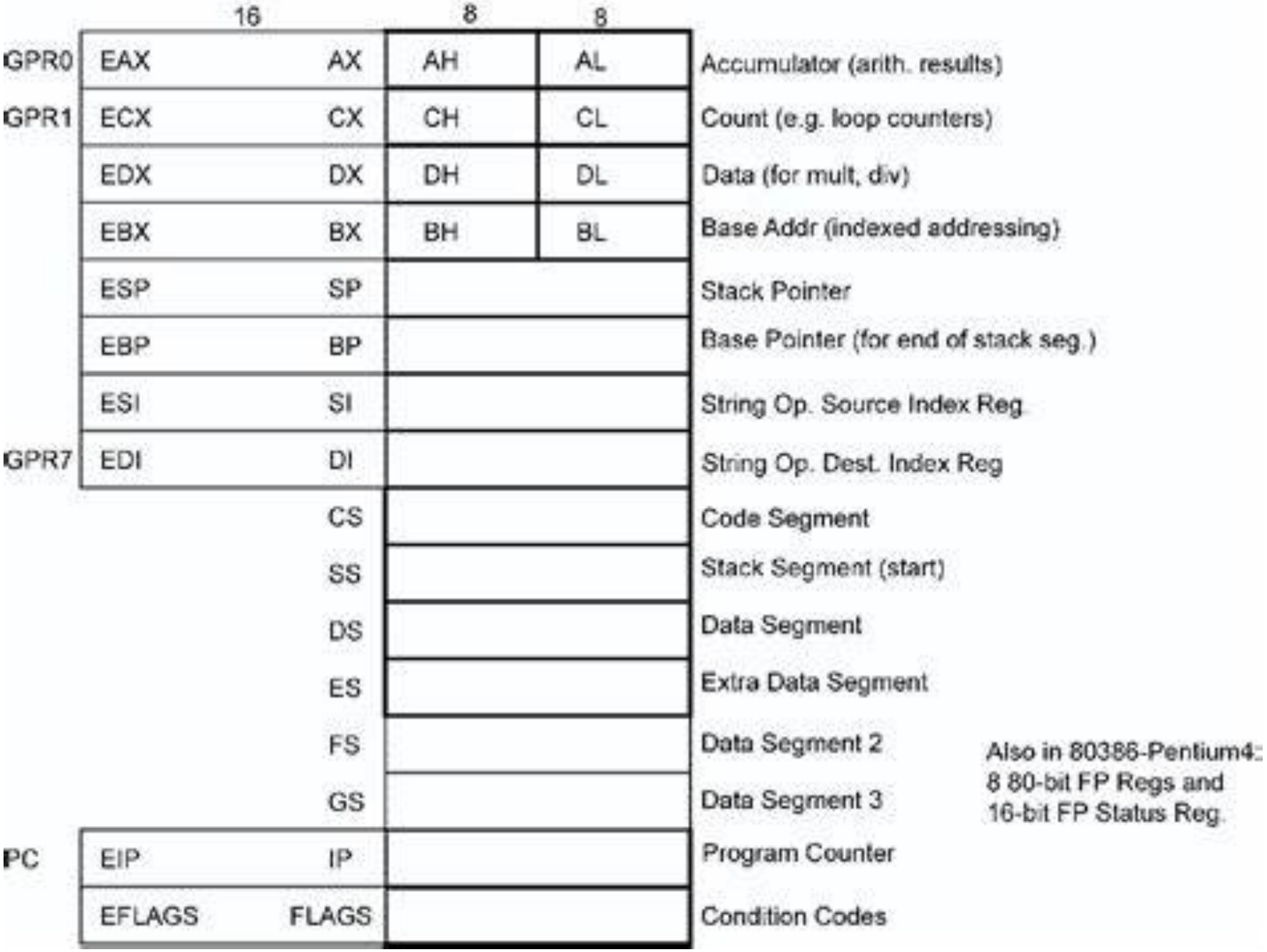
IA-32 register structure.



IA-32 register structure.



Compatibility of the IA-32 register structure with earlier Intel processor register structures.


16
8
8
GPR0
EAX
AX
AH
AL
Accumulator (arith. results)
GPR1
ECX
CX
CH
CL
Count (e.g. loop counters)
EDX
DX
DH
DL
Data (for mult, div)
EBX
BX
BH
BL
Base Addr (indexed addressing)
ESP
SP
Stack Pointer
EBP
BP
Base Pointer (for end of stack seg.)
ESI
SI
String Op. Source Index Reg
GPR7
EDI
DI
String Op. Dest. Index Reg
CS
Code Segment
SS
Stack Segment (start)
DS
Data Segment
ES
Extra Data Segment
FS
Data Segment 2
GS
Data Segment 3
PC
EIP
IP
Program Counter
EFLAGS
FLAGS
Condition Codes
Also in 80386-Pentium4:
8 80-bit FP Regs and
16-bit FP Status Reg.

- The IA-32 general-purpose registers allow for compatibility with the registers of earlier 8-bit and 16-bit Intel processors.

- In those processors, there are some restrictions regarding the use of certain registers

- The eight general-purpose registers are grouped into three different types:
  - data registers for holding operands,
  - pointer registers for holding addresses, and
  - index registers for holding address indices.

- The pointer and index registers are used to determine the effective address of a memory operand.

Addressing Modes

- IA-32 architecture has a large and flexible set of addressing modes.

- They are designed to access individual data items, or data items that are members of an ordered list that begins at a specified memory address.

- Effective address of the operand,

  EA, is calculated as

  EA = [Ri] + [Rj] + X

  where Ri and Rj are general-purpose registers and X is a constant..

- Instructions have zero, one, or two operands.

- In two-operand instructions, the source(src) and destination (dst) operands are specified in assembly language in the order

  OP dst, src

Addressing Modes

Table E.1 IA-32 addressing modes.

Name	Assembler syntax	Addressing function
Immediate	Value	Operand = Value
Direct	Location	EA = Location
Register	Reg	EA = Reg that is, Operand = [Reg]
Register indirect	[Reg]	EA = [Reg]
Base with displacement	[Reg + Disp]	EA = [Reg] + Disp
Index with displacement	[Reg * S + Disp]	EA = [Reg] × S + Disp
Base with index	[Reg1 + Reg2 * S]	EA = [Reg1] + [Reg2] × S
Base with index and displacement	[Reg1 + Reg2 * S + Disp]	EA = [Reg1] + [Reg2] × S + Disp

Value = an 8- or 32-bit signed number
Location = a 32-bit address
Reg, Reg1, Reg2 = one of the general purpose registers EAX, EBX, ECX, EDX, ESP, EBP, ESI, EDI, with the exception that ESP cannot be used as an index register.
Disp = an 8- or 32-bit signed number, except that in the Index with displacement mode it can only be 32 bits.
S = a scale factor of 1, 2, 4, or 8

Immediate addressing mode

It is convenient to use the Move instruction to illustrate the IA-32 addressing modes and their notation in assembly language.

The instruction

MOV EAX, 25

uses the Immediate addressing mode for the source operand to move the decimal value 25 into the destination register EAX.

Direct addressing mode

Symbolic names may also be used as operands. If the name LOCATION has been defined as an address label, the instruction

```
MOV EAX, LOCATION
```

implicitly uses the Direct addressing mode to move the doubleword at memory address LOCATION into register EAX.

The Direct addressing mode can also be made explicit. The instruction

```
MOV EAX, DWORD PTR LOCATION
```

uses the keywords DWORDPTR to indicate that the label LOCATION should be interpreted as the address of a 32-bit operand.

Register addressing mode

MOV EAX, EBX

moves the value of the address label EBX into the EAX register using the Register addressing mode.

Register indirect addressing mode

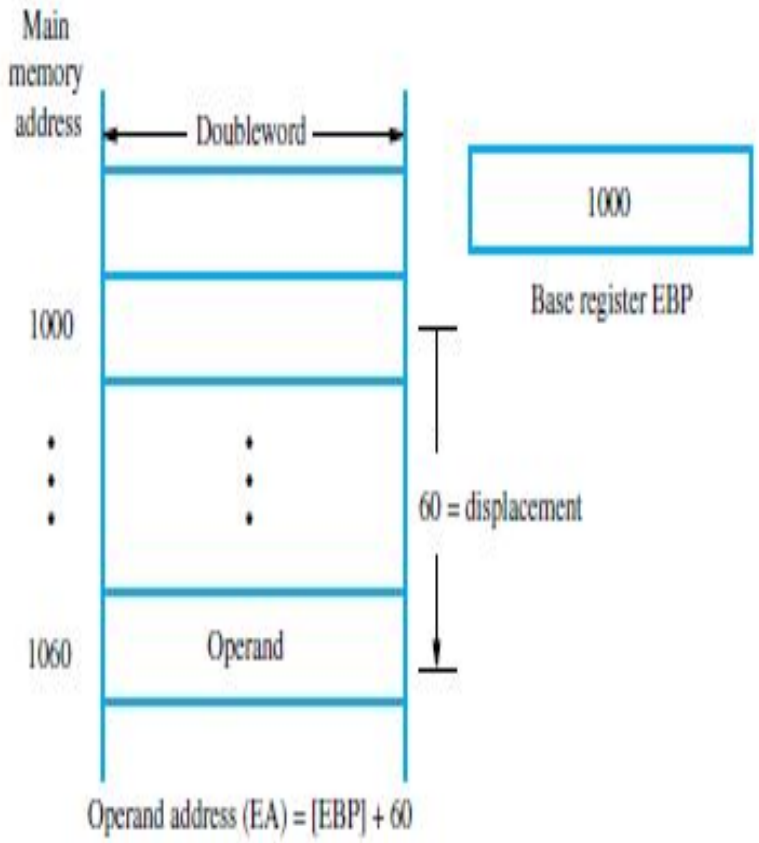
Once an address is loaded into a register, the Register indirect mode can be used to access the operand in memory.
The instruction

MOV EAX, [EBX]

moves the contents of the memory location whose address is contained in register EBX into register EAX.

Base with displacement mode

- Register EBP is used as the base register.
- A doubleword operand at address 1060, which is 60 byte locations away from the base address of 1000, can be moved into register EAX by the instruction
- MOV EAX, [EBP + 60]



Base with displacement mode, expressed as [EBP + 60]

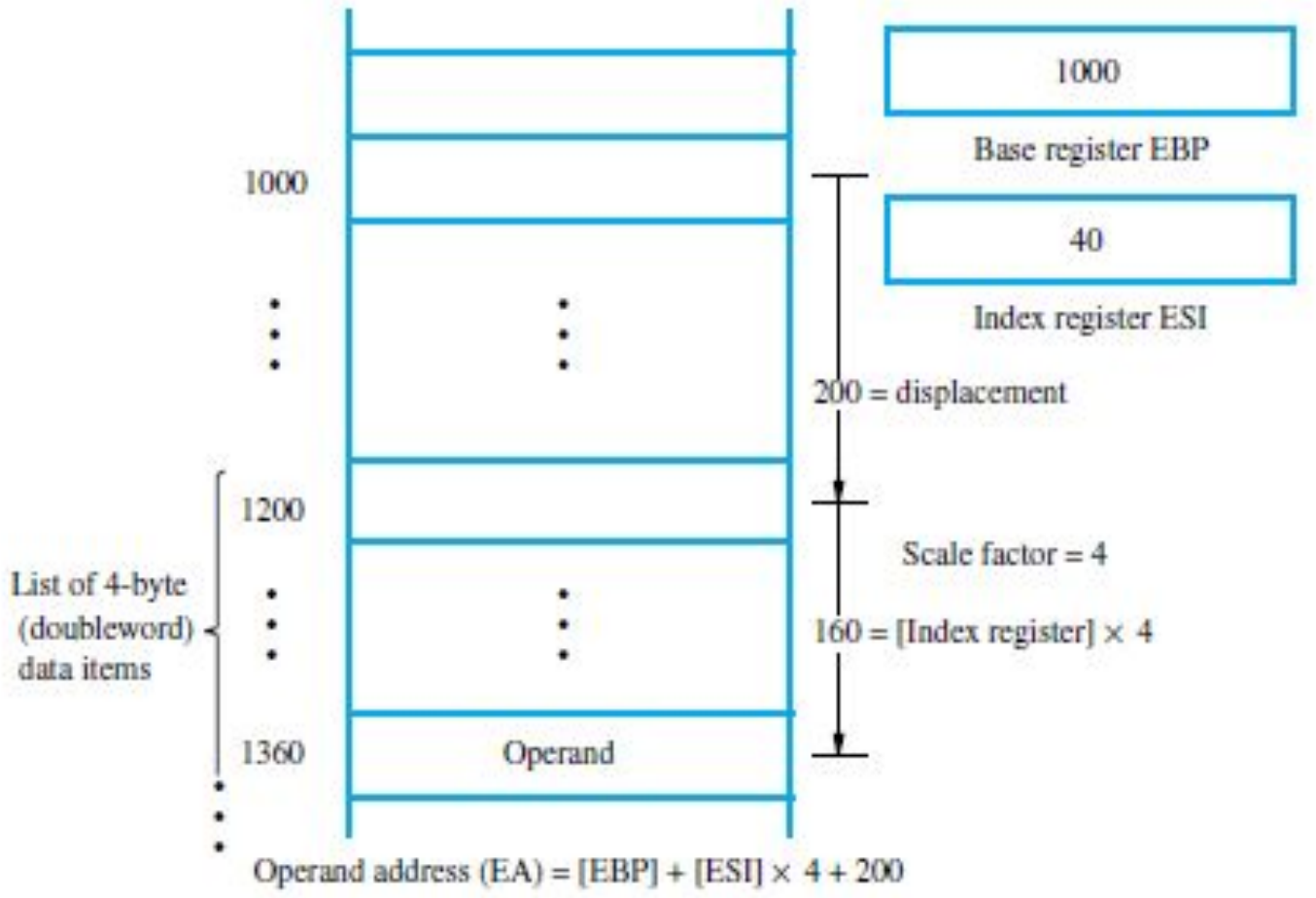
Base with index and displacement mode

- An 8-bit or 32-bit signed displacement, two of the eight general-purpose registers, and a scale factor of 1, 2, 4, or 8, are specified in the instruction. The registers are used as base and index registers.

- Effective address of the operand is determined by first multiplying the contents of the index register by the scale factor and then adding the result to the contents of the base register and the displacement.

- The addressing mode that provides the most flexibility

MOV EAX, [EBP + ESI * 4 + 200]



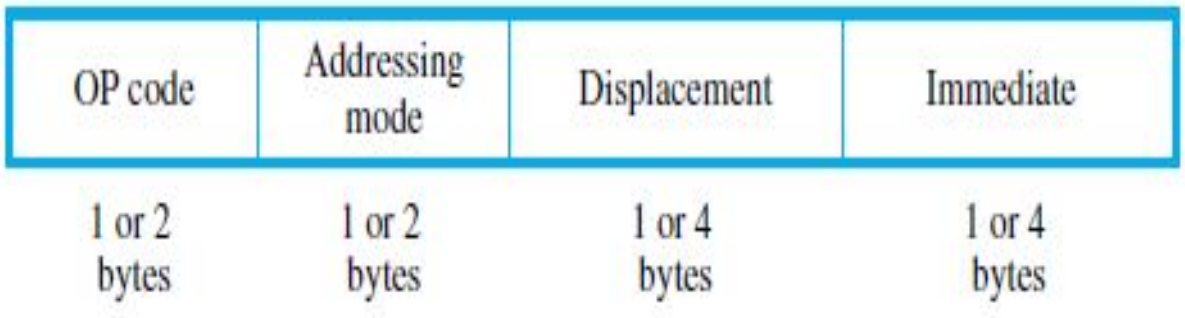
Base with displacement and index mode, expressed as [EBP + ESI * 4 + 200]

Instructions

- IA-32 instruction set is extensive.
- Instructions are provided for moving data between the memory and the processor registers, performing arithmetic operations, and performing logical and shift/rotate operations.
- In the two-operand case, only one of the operands can be in the memory.
- The other must either be in a processor register or be an immediate value in the instruction.

- Jump instructions and subroutine call/return instructions are included.

- Push and pop operations for manipulating the processor stack are also directly supported in the instruction set.



IA-32 instruction format.

Machine Instruction Format

- Instructions are variable in length, ranging from 1 to 12 bytes and consisting of up to four fields.

- The OP-code field consists of one or two bytes, with most instructions requiring only one byte.

- Addressing mode information is contained in one or two bytes immediately following the OP code.

- For instructions that involve the use of only one register in generating the effective address of an operand in memory, only one byte is needed in the addressing mode field.

- Some simple instructions, such as those that increment or decrement a register, occupy only one byte. For example, the instruction

INC EDI

Move Instruction

- Register contents may also be transferred to memory or to another register. The instruction

MOV LOCATION, ECX

moves the doubleword in register ECX into the memory location at address LOCATION.

- The instruction

MOV EBP, EDI

moves the doubleword in register EDI to register EBP. The contents in register EDI are not changed.

The MOV instruction cannot be used with two memory operands, but it can be used to move an immediate value into a memory location, as in

MOV DWORD PTR [EAX + 16], 100

Note that the assembler requires the keywords DWORD PTR (or BYTE PTR) to specify the operand size in this instruction.

Load-Effective-Address Instruction

MOV EAX, OFFSET LOCATION

MOV instruction can be used to load an address into a register by using the keyword OFFSET. Alternatively, the LEA (Load-effective-address) instruction may be used

LEA EAX, LOCATION

The LEA instruction can be used to load an effective address that is computed at execution time.

The desired operand is an element of an array, located at an offset of 12 bytes from the start of the array. If register EBP contains the starting address of the array, the instruction

LEA EBX, [EBP + 12]

computes the desired effective address and places it in register EBX. The operand can then be accessed by a Move or other instruction using the Register indirect mode with EBX.

Arithmetic Instructions
Addition, Subtraction, Comparison, and Negation

Two-operand arithmetic instructions are:

- ADD (Add)
- ADC (Add with carry; for multiple-precision arithmetic)
- SUB (Subtract)
- SBB (Subtract with borrow; for multiple-precision arithmetic)
- CMP (Compare; value of destination operand remains unchanged)

One-operand arithmetic instructions are:

- INC (Increment)
- DEC (Decrement)
- NEG (Negate)

The NEG instruction affects all condition code flags, but the INC and DEC instructions do not affect the CF flag. These instructions must include keywords to specify the operand size unless the Register mode is used for the operand.

These instructions affect all of the condition code flags based on the result of the operation that is performed.

The instruction

ADD EAX, EBX

performs the 32-bit operation

EAX←[EAX] + [EBX]

The instruction

CMP [EBX + 10], AL

performs the 8-bit operation

[[EBX] + 10] − [AL]

Using register AL implies an operand size of one byte

The instruction

INC DWORD PTR [EDX]

increments the doubleword at the memory location whose address is contained in register EDX.

Multiplication

The signed integer multiplication instruction, IMUL, performs 32-bit multiplication. Depending on the form of the instruction that is used, the destination may be implicit and the 64-bit product may be truncated to 32 bits.

One form of this instruction is

IMUL src

which implicitly uses the EAX register as the multiplicand.

A second form of this instruction is

IMUL REG, src

The destination operand, REG, must be one of the eight general-purpose registers. The source operand can be in a register or in the memory. The product is truncated to 32 bits before it is placed in the destination register REG.

Division

The integer divide instruction, IDIV, operates on a 64-bit dividend and a 32-bit divisor to generate a 32-bit quotient and a 32-bit remainder.

The format of the instruction is

IDIV src

The source operand is the divisor. The 64-bit dividend is formed by the contents of register EDX (high-order half) and register EAX (low-order half).

After performing the division, the quotient is placed in EAX and the remainder is placed in EDX. All of the condition code flags are undefined. Division by zero causes an exception.

Jump and Loop Instructions

Conditional Jump Instructions and Condition Code Flags

Table E.2 IA-32 conditional jump instructions.

Mnemonic	Condition name	Condition test
JS	Sign (negative)	SF = 1
JNS	No sign (positive or zero)	SF = 0
JE/JZ	Equal/Zero	ZF = 1
JNE/JNZ	Not equal/Not zero	ZF = 0
JO	Overflow	OF = 1
JNO	No overflow	OF = 0
JC/JB	Carry/Unsigned below	CF = 1
JNC/JAE	No carry/Unsigned above or equal	CF = 0
JA	Unsigned above	CF ∨ ZF = 0
JBE	Unsigned below or equal	CF ∨ ZF = 1
JGE	Signed greater than or equal	SF ⊕ OF = 0
JL	Signed less than	SF ⊕ OF = 1
JG	Signed greater than	ZF ∨ (SF ⊕ OF) = 0
JLE	Signed less than or equal	ZF ∨ (SF ⊕ OF) = 1

Unconditional Jump Instruction

- An unconditional Jump instruction, JMP, causes a branch to the instruction at the target address.

- In addition to using short (one-byte) or long (four-byte) relative signed offsets to determine the target address, as is done in conditional Jump instructions, the JMP instruction also allows the use of other addressing modes.

- This flexibility in generating the target address can be very useful.

- At execution time, the index of the selected case is loaded into index register ESI.

- A jump to the routine for the selected case is performed by executing the instruction

JMP [JUMPTABLE + ESI * 4]

which uses the Index with displacement addressing mode.

Loop Instruction

A loop can be implemented as

```
        MOV ECX, NUM_PASSES
START:
        ...
        DEC ECX
        JG START
```

Loops of this form can be expressed in a more compact manner by using the LOOP instruction.

It combines the functionality of the DEC and JG instructions, and it also implicitly uses register ECX for the counter variable. Using this instruction, the loop can be implemented as

```
        MOV ECX, NUM_PASSES
START:
        ...
        LOOP START
```

Condition code flags are not affected by the LOOP instruction.

Logic Instructions

- The IA-32 architecture has instructions that perform the logic operations AND, OR, and XOR.

- The operation is performed bitwise on two operands, and the result is placed in the destination location.

- For example, suppose register EAX contains the hexadecimal pattern 0000FFFF and register EBX contains the pattern 02FA62CA. The instruction

- AND EBX, EAX

clears the left half of EBX to all zeroes, and leaves the right half unchanged. The result in EBX will be 000062CA.

- There is also a NOT instruction which generates the logical complement of all bits of the operand, that is, it changes all 1s to 0s and all 0s to 1s.

Shift and Rotate Instructions

An operand can be shifted right or left, using either logical or arithmetic shifts, by a number of bit positions determined by a specified count. The format of the shift instructions is

OP dst, count

There are four shift instructions:

- SHL (Shift left logical)
- SHR (Shift right logical)
- SAL (Shift left arithmetic; operation is identical to SHL)
- SAR (Shift right arithmetic)

There are four rotate instructions:

- ROL (Rotate left without the carry flag CF)
- ROR (Rotate right without the carry flag CF)
- RCL (Rotate left including the carry flag CF)
- RCR (Rotate right including the carry flag CF)

Subroutine Linkage Instructions

- The stack grows toward lower numbered addresses. The width of the stack is 32 bits, that is, all stack entries are doublewords.

- There are two instructions for pushing and popping individual elements onto and off the stack. The instruction

PUSH src

decrements ESP by 4, and then stores the doubleword at location src into the memory location pointed to by ESP.

- The instruction

POP dst

reverses this process by retrieving the TOS doubleword from the location pointed to by ESP, storing it at location dst, and then incrementing ESP by 4. These instructions implicitly use ESP as the stack pointer.

There are also two more instructions that push or pop the contents of multiple registers. The instruction

PUSHAD

POPAD

These two instructions are used to efficiently save and restore the contents of all registers as part of implementing subroutines

The subroutine is called by the instruction

CALL LISTADD

which first pushes the return address onto the stack and then jumps to ISTADD. Thereturn address is the address of the MOV instruction that immediately follows the CALL instruction.

The subroutine saves the contents of register EDI on the stack.

Calling program

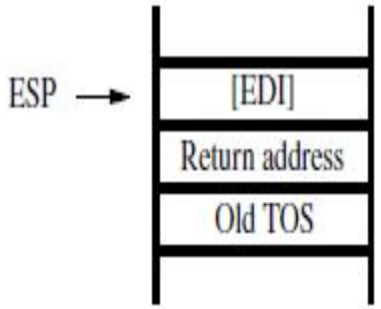
```
                ⋮
        LEA     EBX, NUM1       Load parameters
        MOV     ECX, N            into EBX, ECX.
        CALL    LISTADD         Branch to subroutine.
        MOV     SUM, EAX        Store sum into memory.
                ⋮
```

Subroutine

```
LISTADD:    PUSH    EDI                 Save EDI.
            MOV     EDI, 0              Use EDI as index register.
            MOV     EAX, 0              Use EAX as accumulator register.

STARTADD:   ADD     EAX, [EBX + EDI * 4]  Add next number.
            INC     EDI                 Increment index.
            DEC     ECX                 Decrement counter.
            JG      STARTADD            Branch back if [ECX] > 0.
            POP     EDI                 Restore EDI.
            RET                         Return to calling program.
```

Calling program and subroutine



Stack contents after saving EDI in subroutine

Operations on Large Numbers

```
MOV   EAX, 0A72C10F8H   EAX contains A72C10F8.
MOV   EBX, 10H          EBX contains 10.
MOV   ECX, 5C00FE04H    ECX contains 5C00FE04.
MOV   EDX, 4AH          EDX contains 4A.
ADD   EAX, ECX          Add low-order 32 bits; carry-out sets CF flag.
ADC   EBX, EDX          Add high-order bits with CF flag as carry-in bit.
```

Addition of numbers larger than 32 bits using the ADC instruction.

Interrupts and Exceptions

- Processors implementing the IA-32 architecture use two interrupt-request lines, a nonmaskable interrupt, NMI, and a maskable interrupt, also called the user interrupt request, INTR.

- Interrupt requests on NMI are always accepted by the processor. Requests on INTR are accepted only if they have a higher privilege level than the program currently running.

- INTR interrupts can be enabled or disabled by setting an interrupt-enable bit, IF, in the status register.

- The status register, contains the Interrupt Enable Flag (IF), the Trap flag (TF) and the I/O Privilege Level (IOPL).

- When IF = 1, INTR interrupts are accepted. The Trap flag enables trace interrupts after every instruction.

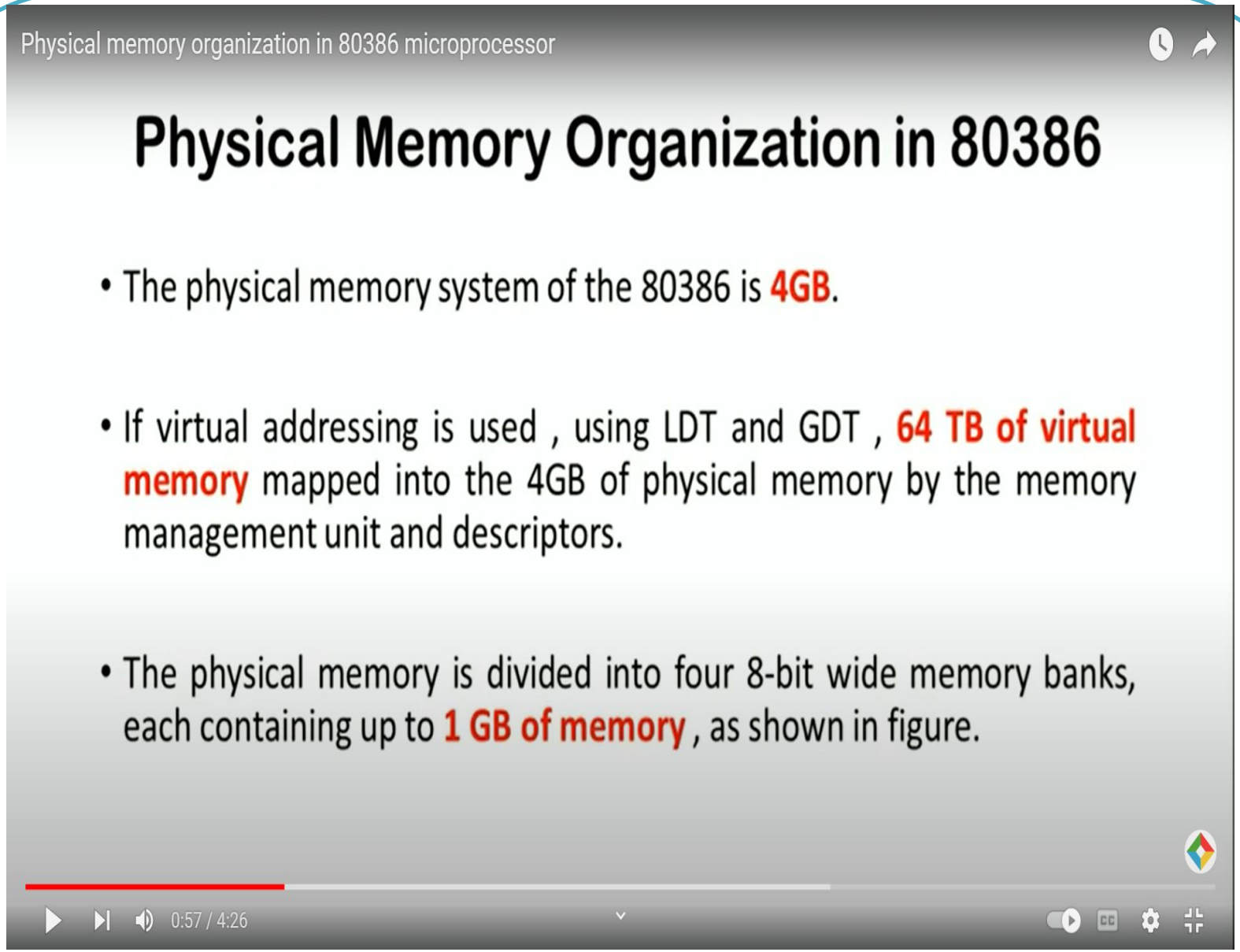
When an interrupt request is received or when an exception occurs, the processor
performs the following actions:

1. It pushes the status register, the Code Segment register (CS), and the Instruction Pointer (EIP) onto the processor stack.

2. In the case of an exception resulting from an abnormal execution condition, it pushes a code on the stack describing the cause of the exception.

3. It clears the corresponding interrupt-enable flag so that further interrupts from the same source are disabled.

4. It fetches the starting address of the interrupt-service routine from the Interrupt Descriptor Table based on the vector number of the interrupt, loads this value into EIP, and then resumes execution.


Physical memory organization in 80386 microprocessor
Physical Memory Organization in 80386
• The physical memory system of the 80386 is 4GB.
• If virtual addressing is used , using LDT and GDT , 64 TB of virtual memory mapped into the 4GB of physical memory by the memory management unit and descriptors.
• The physical memory is divided into four 8-bit wide memory banks, each containing up to 1 GB of memory , as shown in figure.
0:57 / 4:26

Physical memory organization in 80386 microprocessor

Physical Memory Organization in 80386

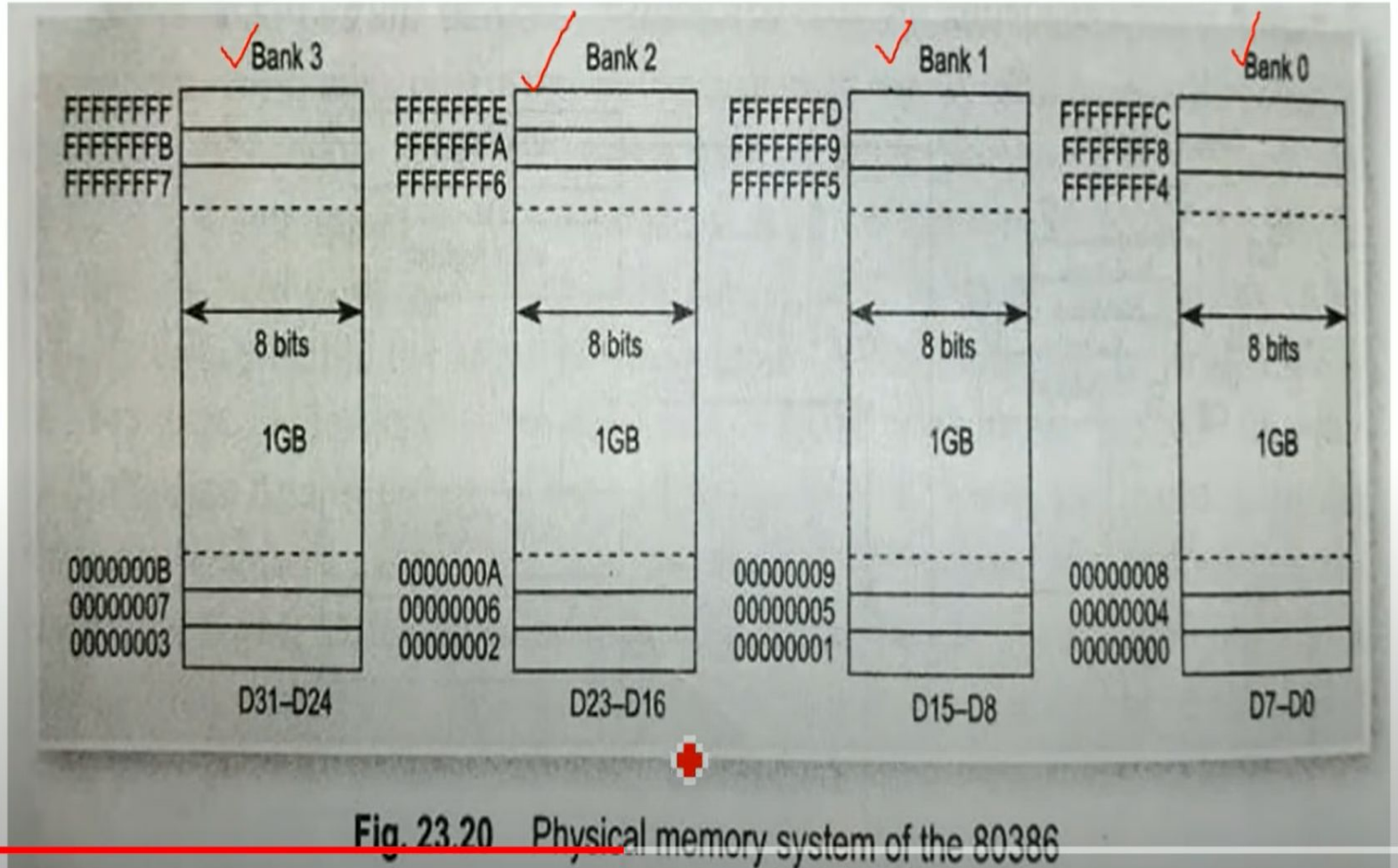
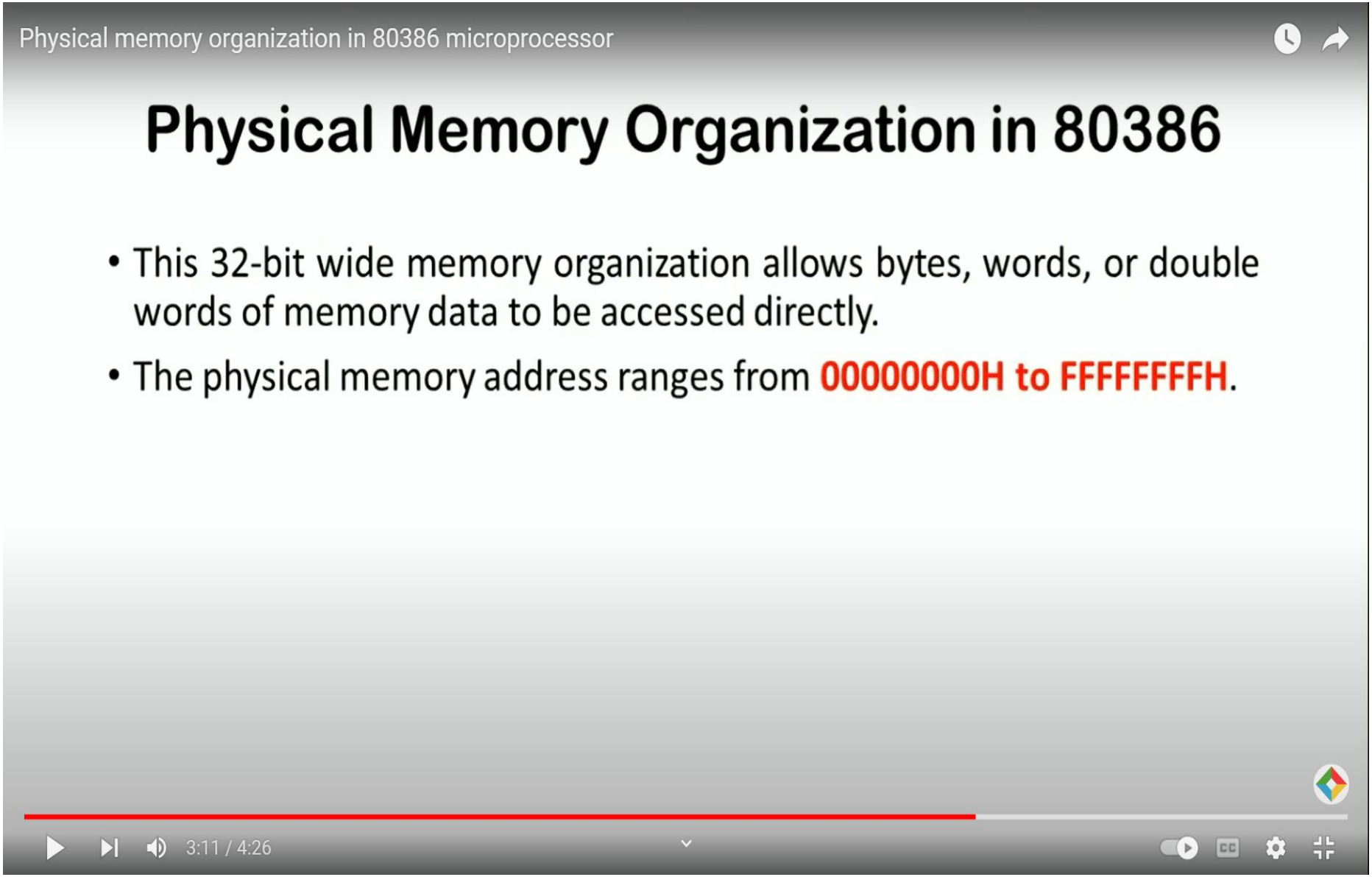


Fig. 23.20 Physical memory system of the 80386

2:04 / 4:26


Physical memory organization in 80386 microprocessor
Physical Memory Organization in 80386
• This 32-bit wide memory organization allows bytes, words, or double words of memory data to be accessed directly.
• The physical memory address ranges from 00000000H to FFFFFFFFH.
3:11 / 4:26

Physical Memory Organization in 80386

- The physical memory location with address 00000000H is in Bank 0 , 00000001H in Bank 1 , 00000002H in Bank 2 , 00000003H in Bank 3, etc.,
- The memory banks 3,2,1, and 0 are accessed via four bank enable signals - **BE3* , BE2* , BE1* , and BE0*** respectively.
- In most cases , a word is addressed in banks 0 and 1 , or in banks 2 and 3.