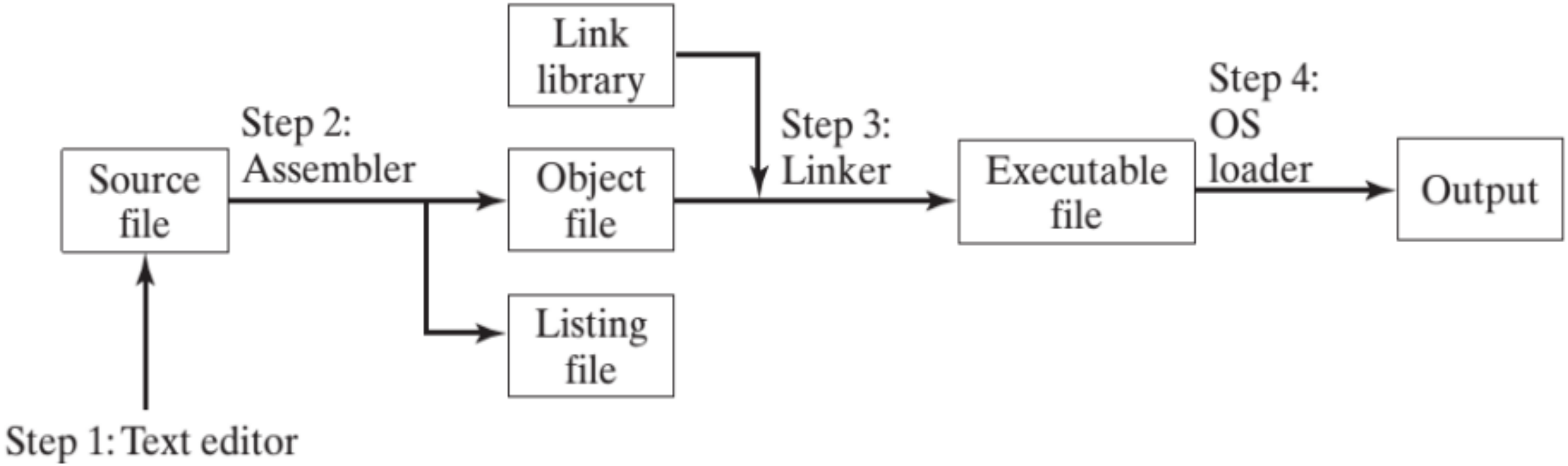
EE-2003
Computer Organization & Assembly Language

ASSEMBLING, LINKING, AND RUNNING PROGRAMS



ASSEMBLING, LINKING, AND RUNNING PROGRAMS

- **Assembler** is a utility program that converts source code programs from assembly language into an object file, a machine language translation of the program. Optionally a Listing file is also produced. We'll use MASM as our assembler.

- The linker reads the object file and checks to see if the program contains any calls to procedures in a link library. The linker copies any required procedures from the link library, combines them with the object file, and produces the executable file. Microsoft 16-bit linker is LINK.EXE and 32-bit is Linker LINK32.EXE.

- **OS Loader:** A program that loads executable files into memory, and branches the CPU to the program's starting address, (may initialize some registers (e.g. IP)) and the program begins to execute.

- **Debugger** is a utility program, that lets you step through a program while it's running and examine registers and memory
- Is called Assemble-Link-Execute Cycle.

Listing File

A listing file contains:

- a copy of the program's source code,
- with line numbers,
- the numeric address of each instruction,
- the machine code bytes of each instruction (in hexadecimal), and
- a symbol table.

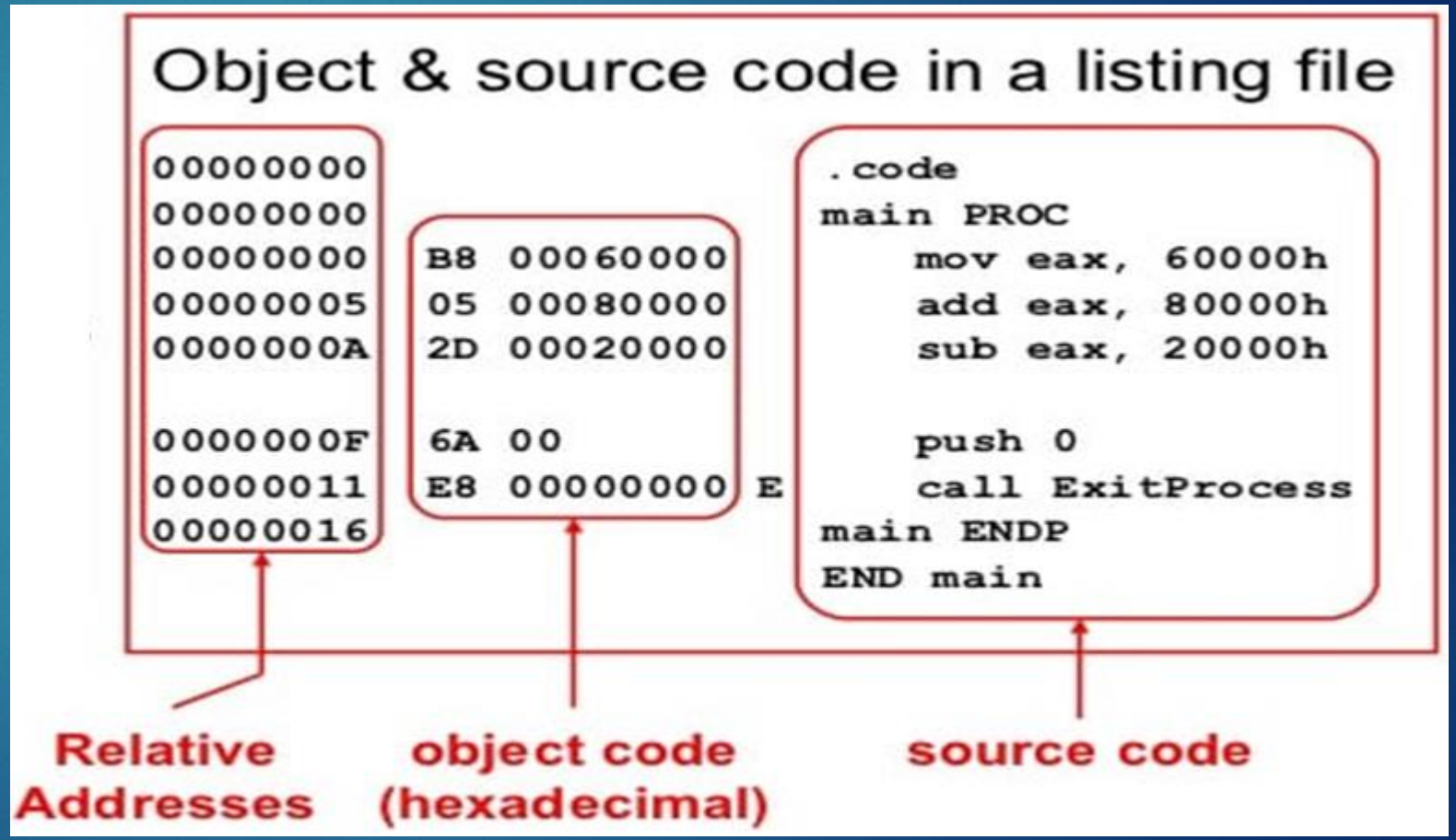
- The symbol table contains the names of all program identifiers, segments, and related information.

Listing File

Use it to see how your program is assembled

Contains:

- source code
- Object Code
- Relative addresses
- Segment names
- Symbols
  - Variables
  - Procedure
  - Constants



Listing File

```
1   TITLE My First Program (Test.asm)
2   INCLUDE Irvine32.inc
3
4   .code
5   main PROC
6   mov eax, 10h
7   mov ebx, 25h
8   call DumpRegs
9   exit
10  main ENDP
11  END main
12
13
```

```
25   00000000          .code
26   00000000          main PROC
27   00000000  B8 00000010      mov eax, 10h
28   00000005  BB 00000025      mov ebx, 25h
29   0000000A  E8 00000000 E    call DumpRegs
30                 exit
31   00000016          main ENDP
32                 END main
33
34  ♀Microsoft (R) Macro Assembler Version 14.29.30133.0        09/12/21 20:28:23
35  ←My First Program (Test.asm                  Symbols 2 - 1
36
```

Loading and Executing a Program

- The operating system (OS) searches for the program's filename in the current disk directory.
  - If it cannot find the name there, it searches a predetermined list of directories (called *paths*) for the filename.
  - If the OS fails to find the program filename, it issues an error message.
- If the program file is found, the OS retrieves basic information about the program's file from the disk directory, including the file size and its physical location on the disk drive.
- The OS determines the next available location in memory and loads the program file into memory.
  - It allocates a block of memory to the program and enters information about the program's size and location into a table (sometimes called a *descriptor table*).
  - Additionally, the OS adjust the values of pointers within the program so they contain addresses of program data.

Loading and Executing a Program

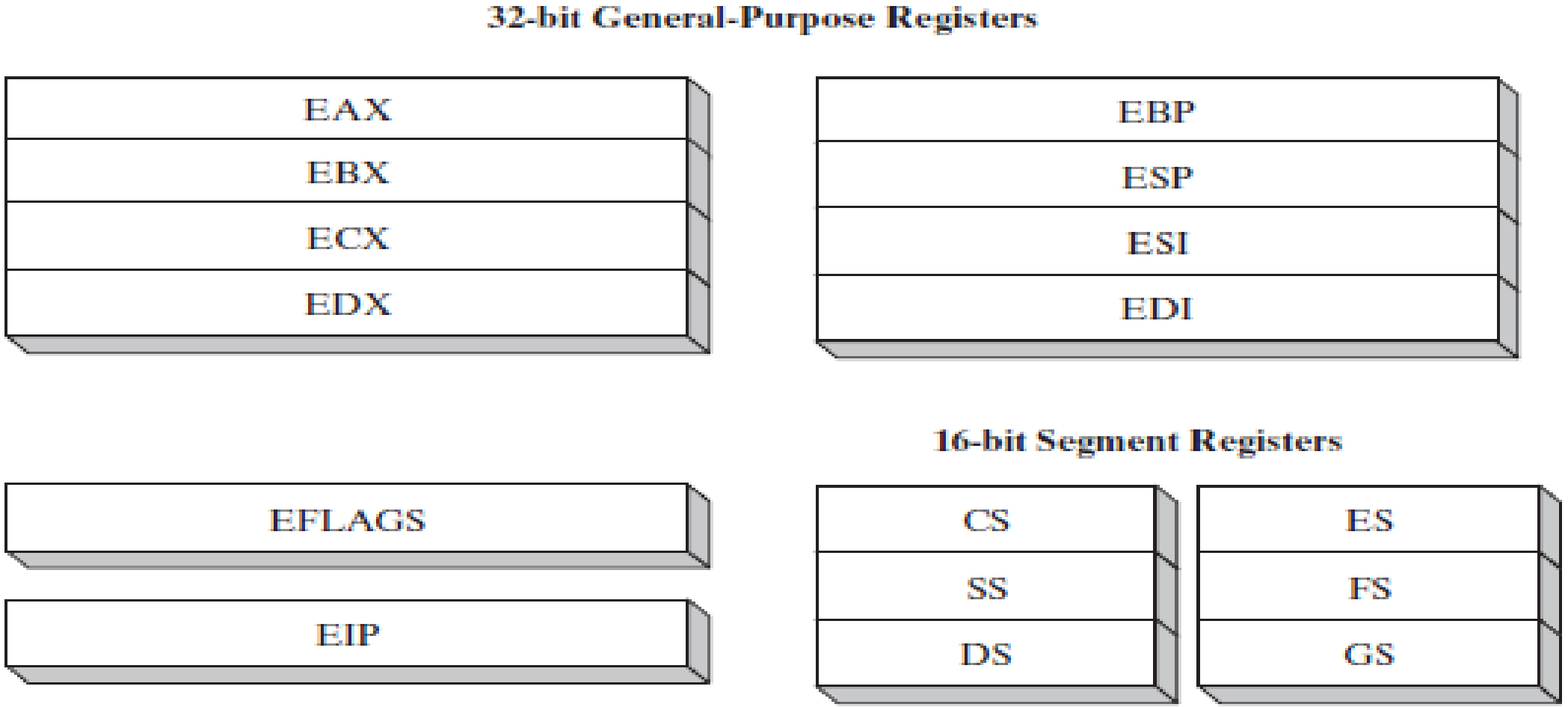
- The OS begins execution of the program's first machine instruction (its entry point).
- As soon as the program begins running, it is called a *process*.
- The OS assigns the process an *identification number* (*process ID*), which is used to keep track of it while running.
- It is the *OS's job to track the execution of the process* and *to respond to requests* for system resources.
  - Examples of resources are memory, disk files, and input-output devices.
- When the process ends, it is removed from memory.

Basic Program Execution Registers

- Registers are high-speed storage locations directly inside the CPU, designed to be accessed at much higher speed than conventional memory.
  - There are eight general-purpose registers (32-bit).
  - Six segment registers (16-bits)
  - A processor status flags register (EFLAGS), and an instruction pointer (EIP)

Basic Program Execution Registers

FIGURE 2–5 Basic Program Execution Registers.

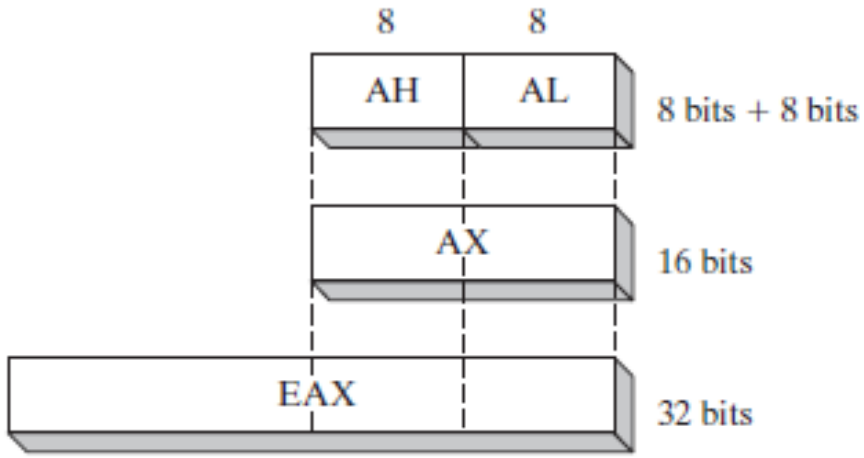


Basic Program Execution Registers

The general-purpose registers are primarily used for arithmetic and data movement.

- As shown in Figure 2–6, the lower 16 bits of the EAX register can be referenced by the name AX
- Portions of some registers can be addressed as 8-bit values
  - For example, the AX register, has an 8-bit upper half named AH and an 8-bit lower half named AL

Figure 2–6 General-Purpose Registers.



Basic Program Execution Registers

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

The remaining general-purpose registers can only be accessed using 32-bit or 16-bit names, as shown in the following table:

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

Specialized Uses of general-purpose registers

- Data Registers (EAX, EBX, ECX, EDX): These four registers are available to the programmer for general data manipulation.

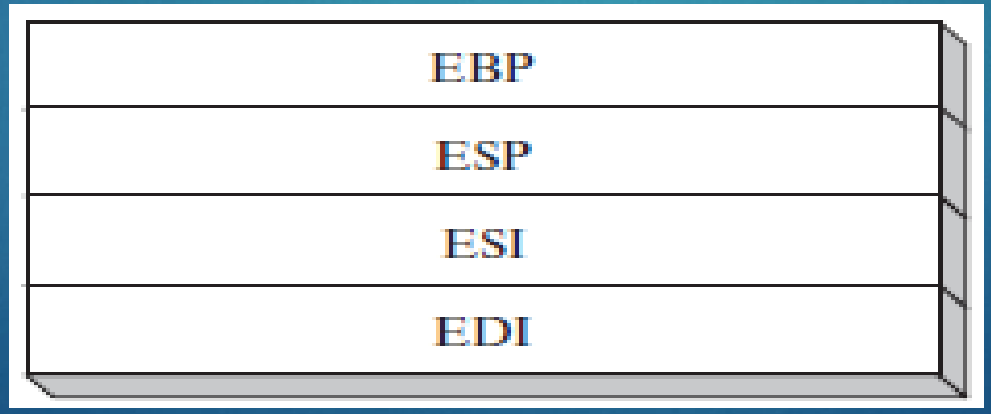
- The high and low bytes of the data registers can be accessed separately.

- EAX (Extended Accumulator Register) is preferred to use in arithmetic, logic and control instructions.
- EBX (Extended Base Register) is used to serve as an address register.
- ECX (Extended Counter Register) serves as loop counter.
- EDX (Extended Data Register) is used in multiplication and division.

Index Registers

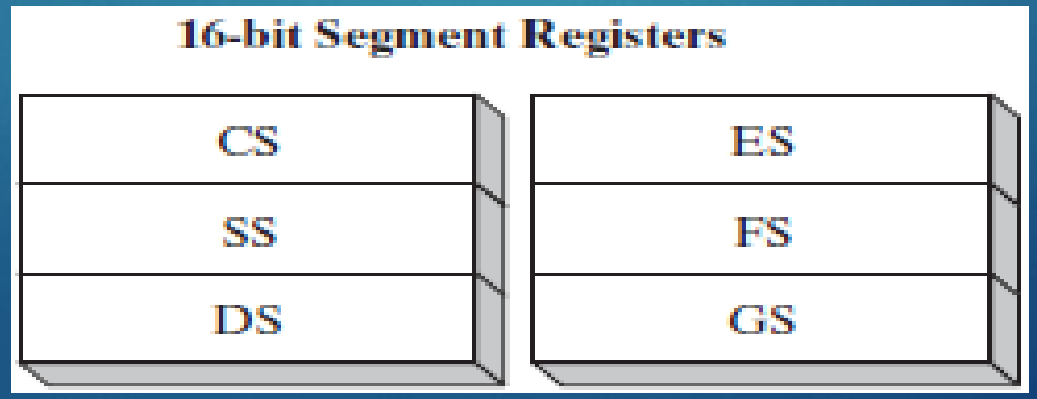
- Index Registers contain the offsets for data and instructions.
- Offset- distance (in bytes) from the base address of the segment.

- ESP (extended stack pointer register) contains the offset for the top of the stack to addresses data on the stack (a system memory structure)
- ESI and EDI (extended source index and extended destination index) points to the source and destination string respectively in the string move instructions
- EBP is used to reference function parameters and local variables on the stack

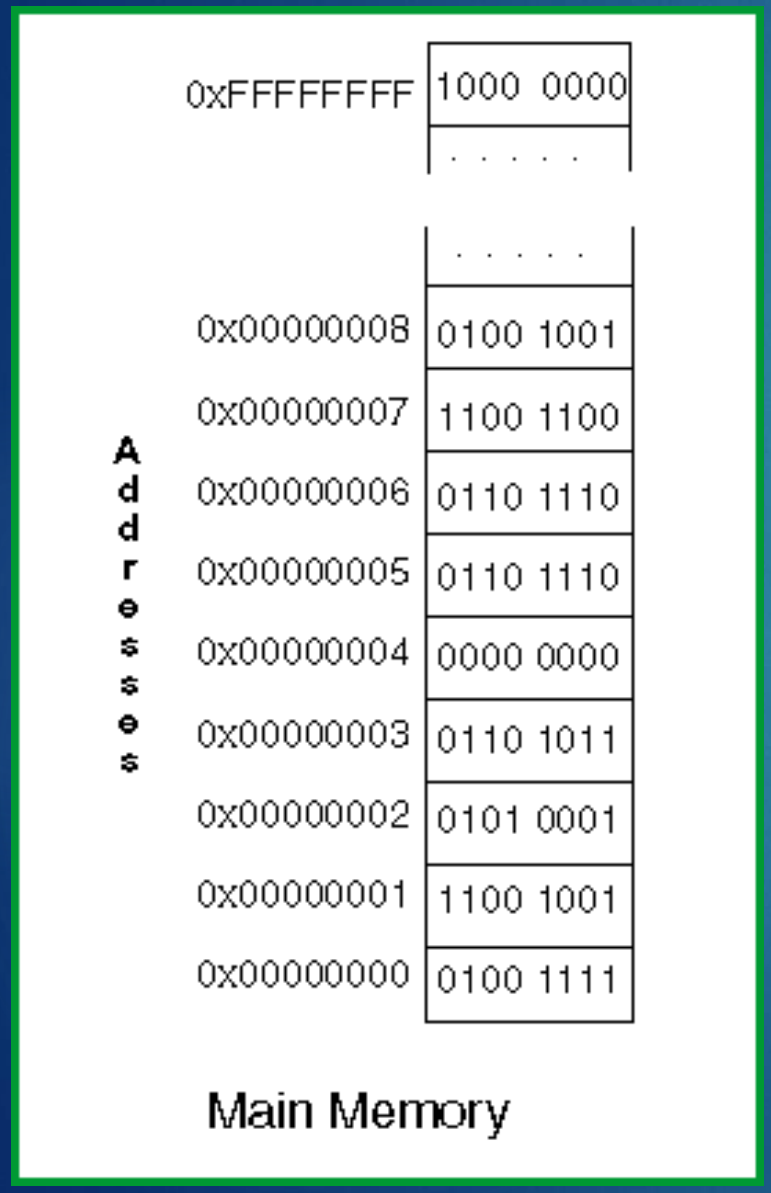


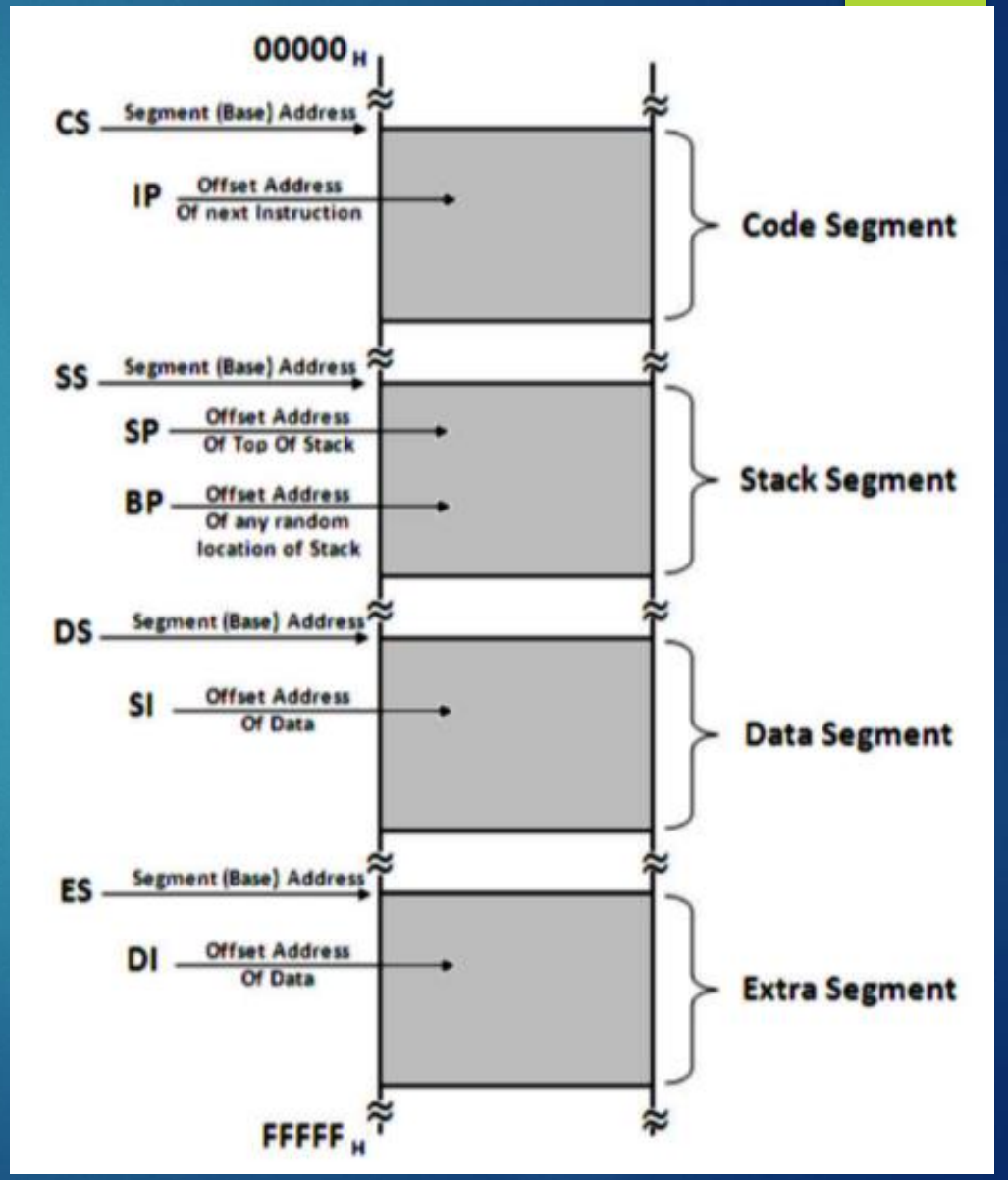
Segment Registers

- In real-address mode, 16-bit segment registers indicate base addresses of pre-assigned memory areas named segments.
- In protected mode, segment registers hold pointers to segment descriptor tables (The descriptor describes the location, length and access rights of the memory segment).
- Some segments hold program instructions (code), others hold variables (data), and another segment named the stack segment holds local function variables and function parameters.



Memory


0xFFFFFFFF
1000 0000
. . . . . .
. . . . . .
0x00000008
0100 1001
0x00000007
1100 1100
0x00000006
0110 1110
0x00000005
0110 1110
0x00000004
0000 0000
0x00000003
0110 1011
0x00000002
0101 0001
0x00000001
1100 1001
0x00000000
0100 1111
Address
Main Memory


00000 H
CS
Segment (Base) Address
IP
Offset Address
Of next Instruction
Code Segment
SS
Segment (Base) Address
SP
Offset Address
Of Top Of Stack
BP
Offset Address
Of any random
location of Stack
Stack Segment
DS
Segment (Base) Address
SI
Offset Address
Of Data
Data Segment
ES
Segment (Base) Address
DI
Offset Address
Of Data
Extra Segment
FFFFF H

Segment Registers

- Memory Segment: A memory segment is a block of consecutive memory bytes. Each segment is identified by a segment number, staring with 0.
- Within a segment, a memory location is specified by giving an off- set. This is the number of bytes from the beginning of the segment.
- A memory location may be specified by providing a segment number and an offset, written in the format segment:offset.
- E.g. A4FB:4872h means offset 4872h within segment A4FBh.

Segment Registers

- The program's code, data, and stack are loaded into different memory segments, we call them the code segment, data segment, and stack segment.
- Stack segment holds local function variables and function parameters.
- To keep track of the various program segments, segment register are used.
- The ECS (Extended Code Segment), EDS (Extended Data Segment), and ESS (Extended Stack Segment) registers contain the code, data, and segment numbers respectively.
- If a program needs to access a second data segment, it can use the EES (Extended Extra segment) register.

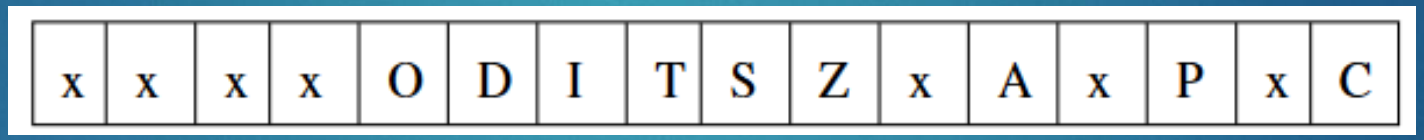
Instruction Pointer

- The EIP, or instruction pointer, register contains the address of the next instruction to be executed.

- Certain machine instructions manipulate EIP, causing the program to branch to a new location.

EFLAGS Register

- The EFLAGS register consists of individual binary bits that control the operation of the CPU or reflect the outcome of some CPU operation.
  - A flag is set when it equals 1; it is clear (or reset) when it equals 0.
- Programs can set individual bits in the EFLAGS register to control the CPU's operation
- For example: Interrupt when arithmetic overflow is detected



O = Overflow	S = Sign
D = Direction	Z = Zero
I = Interrupt	A = Auxiliary Carry
T = Trap	P = Parity
x = undefined	C = Carry

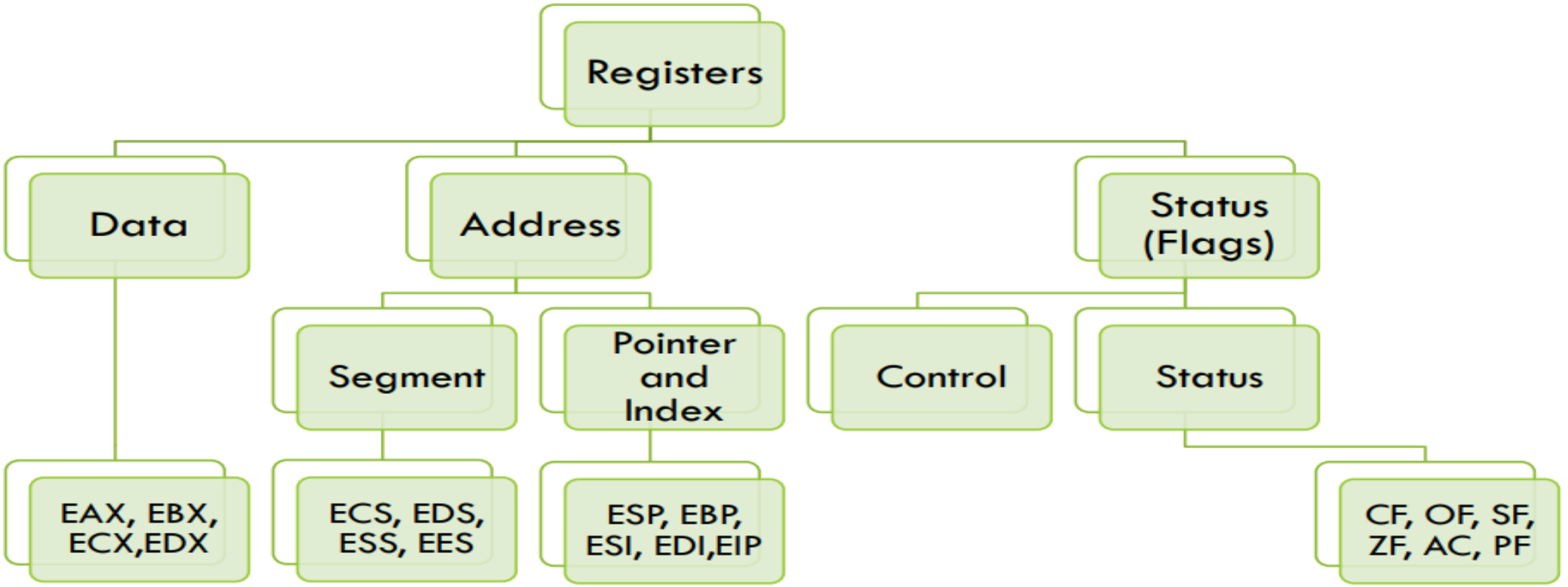
EFLAGS Register

- There are two types of flags:
  - Control flags: which determine how instructions are carried out
  - Status flag: which report on the result operation
- Control flags include:
  - Direction Flag (DF): affects the direction of block data transfers (like long character string) 1=up; 0= down.
  - Interrupt Flag (IF): determines whether interrupts can occur (whether devices like keyboard, disk drives and system clock can get the CPU's attention to get their needs attended to.
  - Trap Flag (TF): determines whether the CPU is halted after every instruction. Used for debugging purposes

EFLAGS Register

- Status flags include:
  - Carry Flag (CF): set when the result of unsigned arithmetic is too large to fit in the destination, 1=carry; 0=no carry.
  - Overflow Flag (OF): set when the result of signed arithmetic is too large to fit in the destination, 1=overflow; 0=no overflow.
  - Sign Flag (SF): set when an arithmetic or logical operation generates a negative result. 1=negative, 0=positive.
  - Zero Flag (ZF): set when an arithmetic or logical operation generates a result of zero. Used primarily in jump and loop operations, 1=zero; 0=not zero.
  - Auxiliary-carry Flag (AF): set when an operation causes a carry from bit3 to 4 or borrow (from bit 4 to 3), 1=carry; 0=no carry.
  - Parity Flag (PF): is set if the least significant byte in the result contains an even number of 1 bits. It is used to verify memory integrity.

Basic Program Execution Registers



Mode of Operations

- x86 processors have three primary modes of operation:
  - protected mode,
  - real-address mode, and
  - system management mode.

Mode of Operations

- Real Address mode (original mode provided by 8086)
  - Only 1 MB of memory can be addressed from 0 to FFFFF (hex)
  - Programs can access any part of main memory
  - MS-DOS runs in real address mode
- Implements the programming environment of the Intel 8086 processor
- This mode is available in Windows 98, can be used to run MS-DOS program that requires direct access to system memory and hardware devices
- Programs running in real-address mode can cause the operating system to crash (stop responding to commands)

Mode of Operations

- Protected mode (introduced with the 80386)
  - Each Program can address a maximum of 4 GB of memory
  - The operating system assigns memory to each running program
  - Programs are prevented from accessing each other's memory (segments)
  - Native mode used by Windows NT, 2000, XP and Linux

Mode of Operations

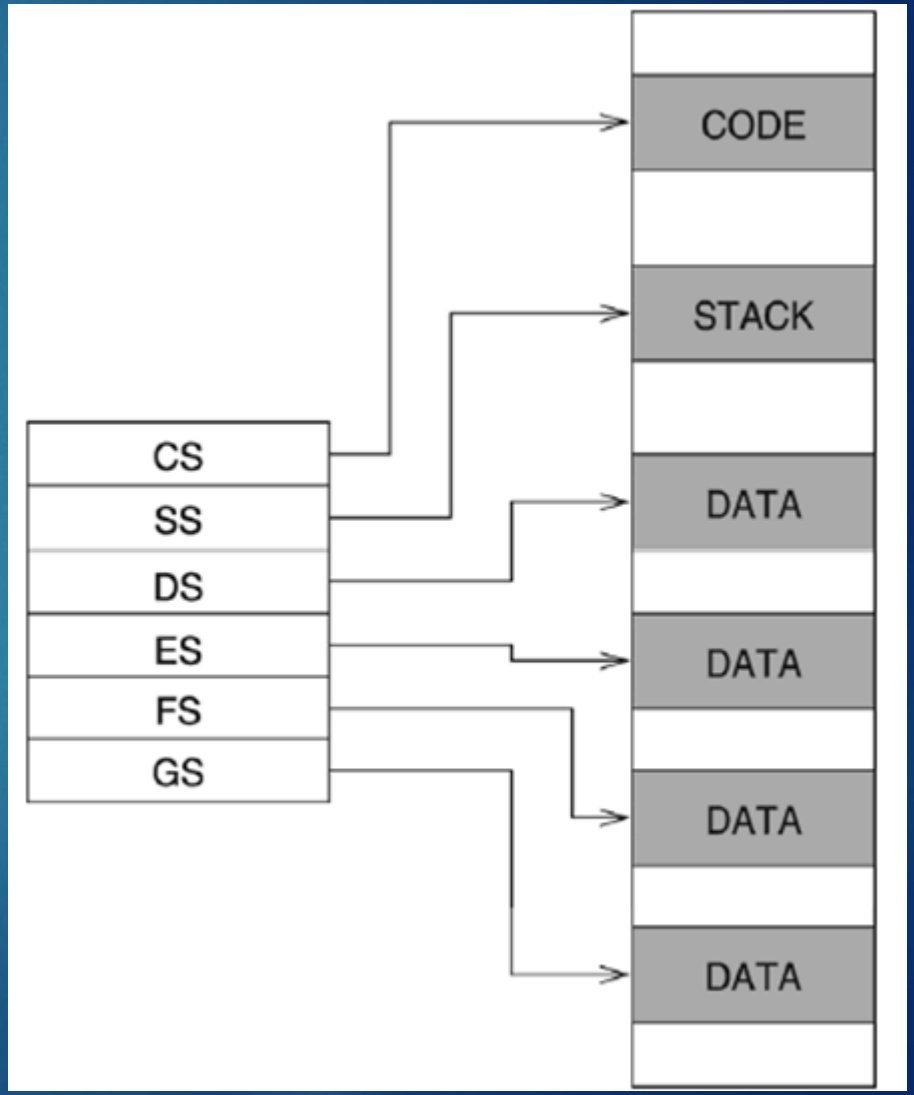
- Virtual 8086 mode: A sub-mode, virtual-8086, is a special case of protected mode
- Processor runs in protected mode, and creates a virtual 8086 machine with 1 MB of address space for each running Program, such as: MS-DOS
  - If an MS-DOS program crashes or attempts to write data into the system memory area, it will not affect other programs running at the same time
  - Windows XP can execute multiple separate virtual-8086 sessions at the same time

Mode of Operations

- System Management Mode:
  - Provides a mechanism for implementation power management and system security
  - Manage system safety functions, such as shutdown on high CPU temperature and turning the fans on and off
  - Handle system events like memory or chipset errors

Real Address Mode

- A Program can access up to six segments at any time
  - Code segment
  - Stack segment
  - Data Segment
  - Extra segments (up to 3)
- Each segment is 64 KB
- Logical address
  - Segment = 16 bits
  - Offset = 16 bits
- Linear (physical) address = 20 bits



Real Address Mode

- Program Segments and Segments Registers: Segment registers are used to hold base addresses for the program code, data and stack.
  - The code segment holds the base address for all executable instructions in the program
  - The data segment holds the base address for variables. This segment stores data for the program
  - The extra segment is an extra data segment (often used for shared data)
  - The stack segment holds the base address for the stack. The segment is also to store interrupt and subroutine return addresses

Real Address Mode

- Linear address = Segments x 10 (hex) + Offset

Example:

- Given:

Segment = A1F0 (hex)

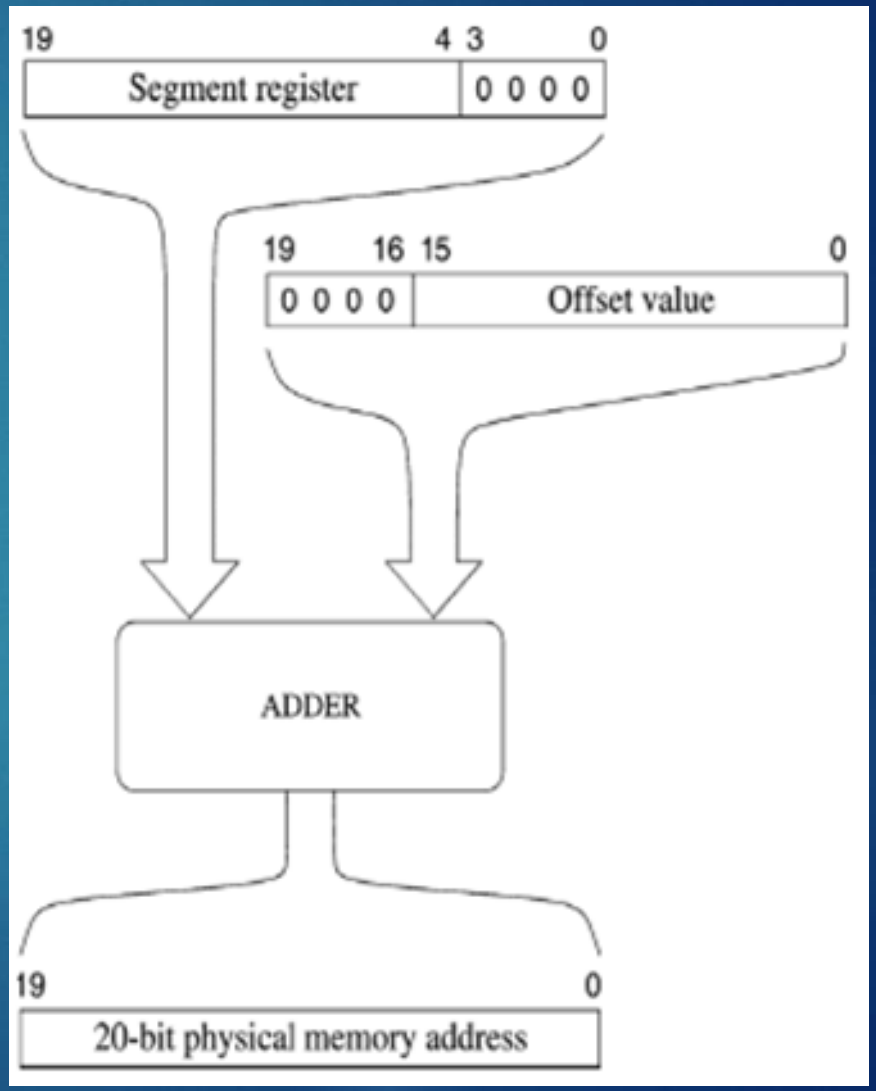
Offset = 04C0 (hex)

Logical Address = A1F0:04C0 (hex)

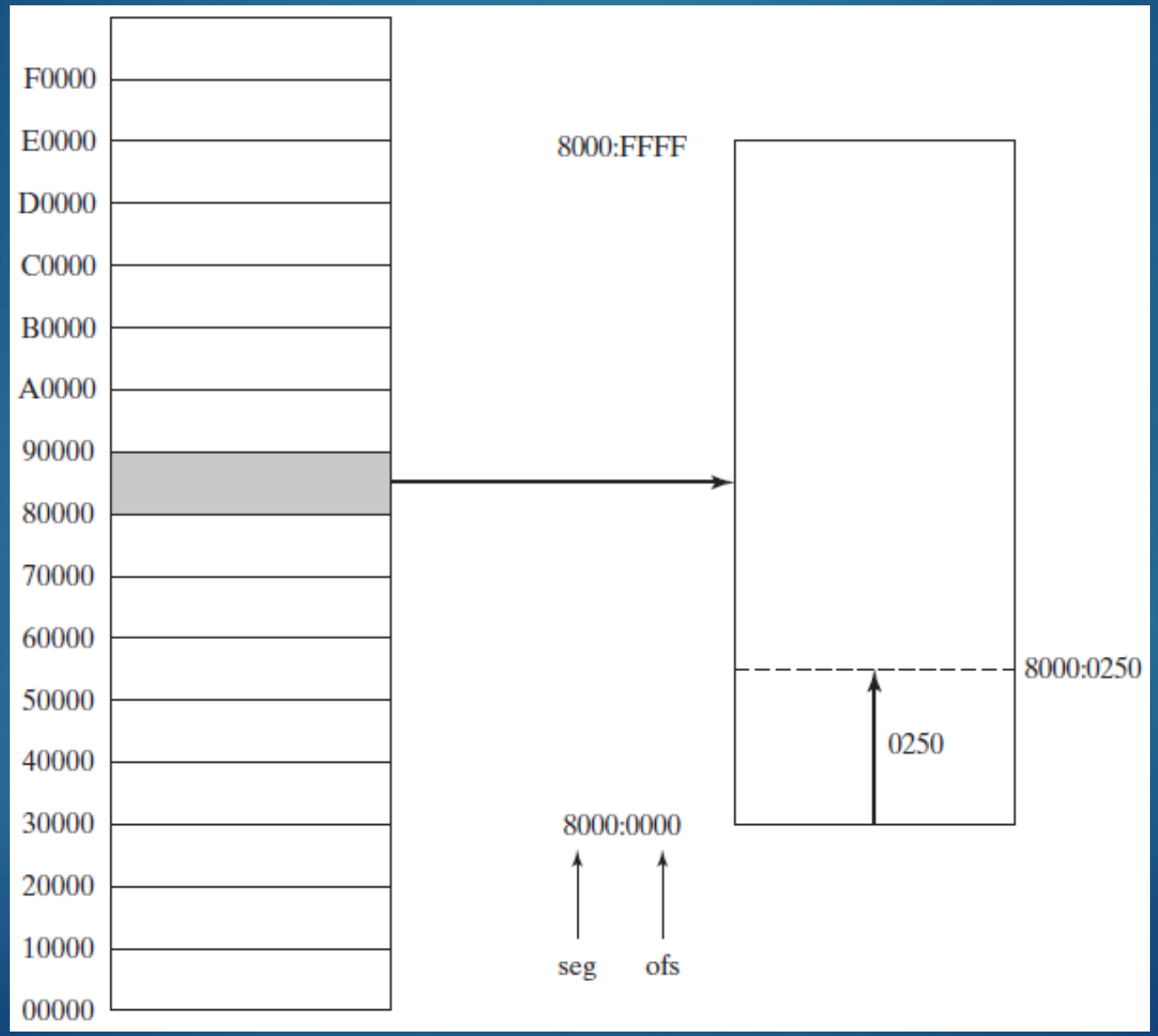
What is linear address?

- Solution:

```
  A1F00  (add 0 to segment in hex)
+  04C0  (offset in hex)
-------
  A23C0  (20-bit linear address in hex)
```



Real Address Mode



Address Space

- In 32-bit protected mode, a task or program can address a linear address space of up to 4 GB
  - Extended Physical Addressing allows a total of 64 GB of physical memory to be addressed
- Real-address mode programs, on the other hand, can only address a range of 1 MB
- If the processor is in protected mode and running multiple programs in virtual-8086 mode, each program has its own 1-MByte memory area