EE-213 Computer Organization and Assembly language

Instructor

- Miss. Mahwish Amjad (mehwish.amjad@nu.edu.pk)
- Sir Nadeem Kafi (nadeem.kafi@nu.edu.pk) Sir Danish
- (m.danish@nu.edu.pk)

- Office: Room No: 18 Opposite to the Department , CS Block
- Office hours: Tuesday 2:00 – 3:00
- Lectures available:
  - Slate

Laboratory instructors

Miss Sehrish, Sir Zain, Miss Sumaiyah

Email: sehrish.saeed@nu.edu.pk
Zain.hassan@nu.edu.pk
sumaiyah@nu.edu.pk

Office hours: ask

Course Objectives

Programming Methodology of low-level languages
How to access computer hardware directly
Overview of a user-visible architecture (of Intel 80x86 processors)
Intel 80x86 instruction set, assembler directives, macro, etc.
How programs interact with the operating system for various services including memory management and input/output services - Device handlers
How is it possible to interface high-level language and low-level language modules

Required Skills

- Proficiency C programming and debugging
- Digital logic design

Course Material

- Lecture notes (posted at the class Slate)
- Textbooks:
  - Computer Organization And Embedded Systems(Sixth Edition) By Hamacher
  - Assembly Language For x86 Processor by Kip R. Irvine

QUIZZES AND ASSIGNMENTS

- 3/2 GRAND QUIZZES
  – 4th Week , 9th Week , 15th Week
- 2-3 Assignments

Course Grading

- Final Exam(50%)
- class participipition(5%)
- Term exams (30%)
- Project/ Assignment/quizzes(15%)

- Be aware:
  - To really understand COAL (or the concepts in COAL), you will need to read the book/lecture notes **repeatedly**.

Your Responsibilities

- Understand lecture and reading materials
- Attend office hours for extra help, as needed
- Uphold academic honesty
- Turn in your assignments on time.
- Check class discussion platform(piazza) and your email account and regularly.

Dos and Don’ts

- Do share knowledge of tools
- Do acknowledge help from others.
- Do acknowledge sources of information from books and web pages

Dos and *Don'ts*

- Don't cheat
- Don't copy code from others
- Don't *paraphrase* code from others either
  - E.g., changing variable names & indentations
- Don't leak your code to any place
  - There is no difference in terms of penalty between copying and being copied.
- All honor code violation will be reported to and resolved through the Office of the Dean and the Faculties.
  - .

Course Policies

- Attendance mandatory
- There are no make-up exams for missed exams unless one (1) has a really good excuse AND (2) notifies the instructor before the exam.

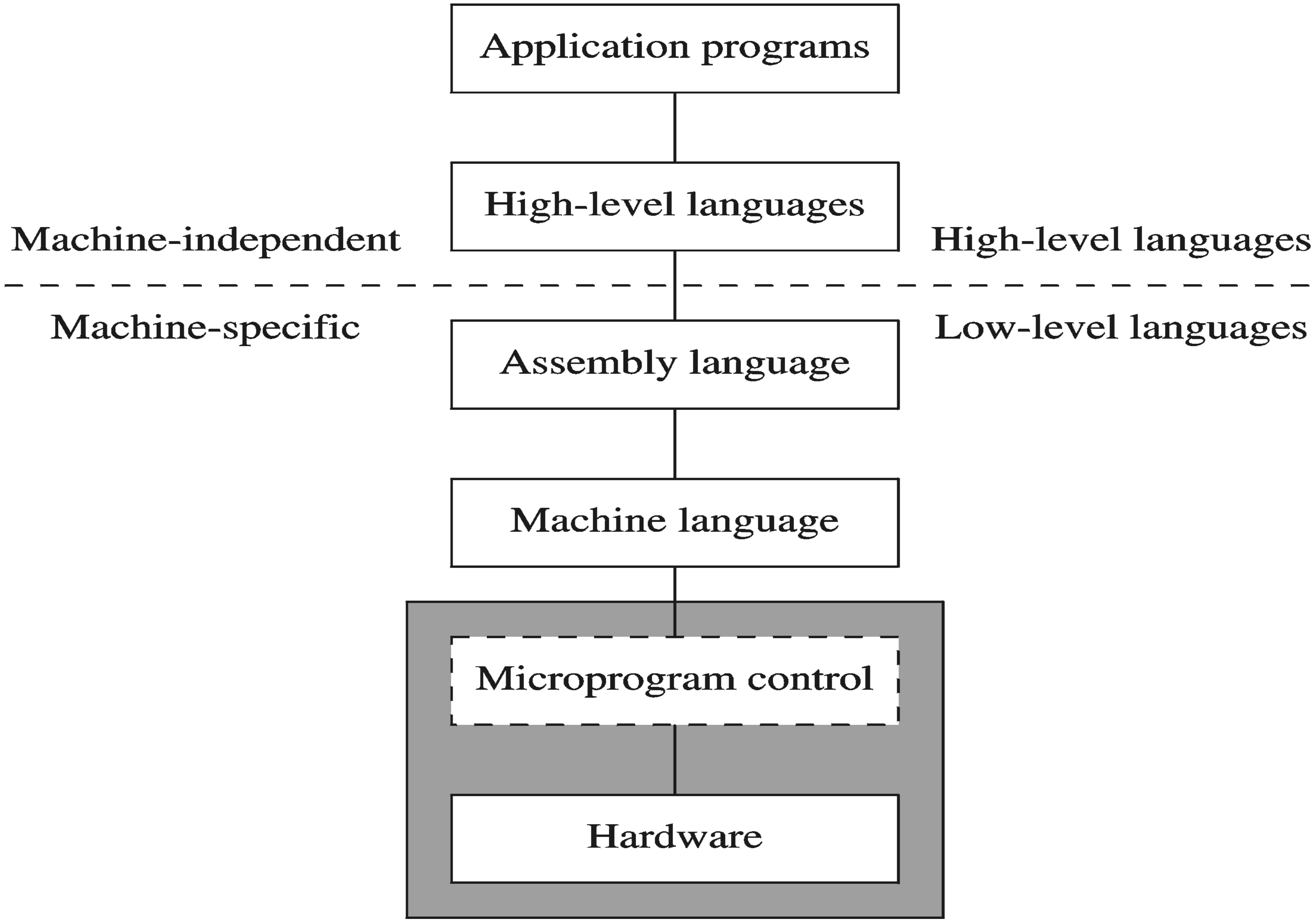
To see or not to see me

- Student feedbacks of ANY KIND are always very welcome for a serious teacher
- We are not psychics
  - We know the materials, but we may not know the most effective way to pass the knowledge to you.
- Please let us know if…
  - Class is too hard
  - You don’t have the background
  - Class can be improved in certain ways
- When in doubt, come knocking…

Some Important Questions to Ask

- What is Assembly Language?
- Why Learn Assembly Language?
- What is Machine Language?
- How is Assembly related to Machine Language?
- What is an Assembler?
- How is Assembly related to High-Level Language?
- Is Assembly Language portable?

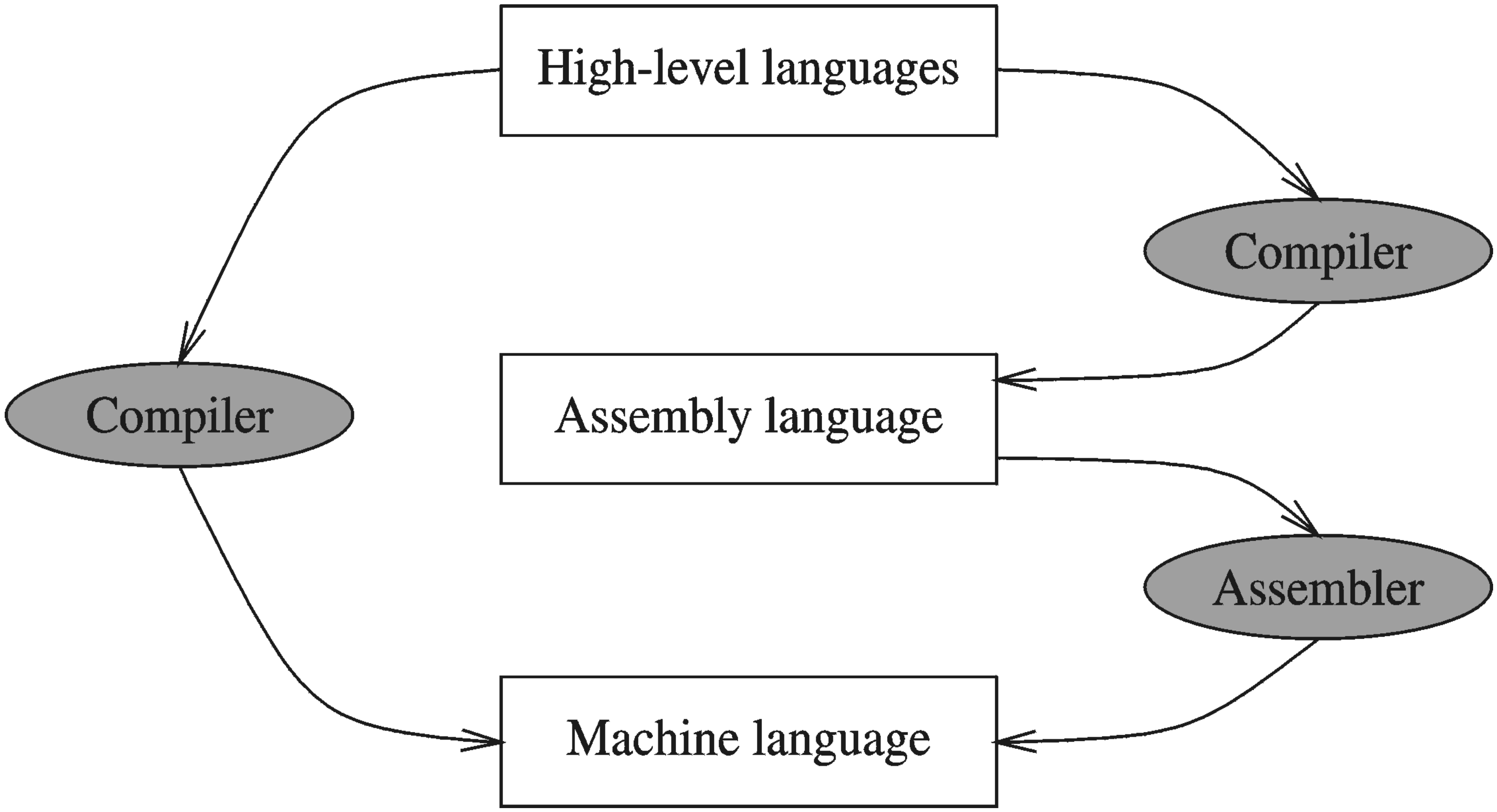
A Hierarchy of Languages



Assembly and Machine Language

- Machine language
  - Native to a processor: executed directly by hardware
  - Instructions consist of binary code: 1s and 0s
- Assembly language
  - Slightly higher-level language
  - Readability of instructions is better than machine language
  - One-to-one correspondence with machine language instructions
- Assemblers translate assembly to machine code
- Compilers translate high-level programs to machine code
  - Either directly, or
  - Indirectly via an assembler

Compiler and Assembler



Translating Languages

English: D is assigned the sum of A times B plus 10.

High-Level Language: D = A * B + 10

A statement in a high-level language is translated typically into several machine-level instructions

Intel Assembly Language:

```
mov   eax, A
mul   B
add   eax, 10
mov   D, eax
```

Intel Machine Language:

```
A1 00404000
F7 25 00404004
83 C0 0A
A3 00404008
```

Advantages of High-Level Languages

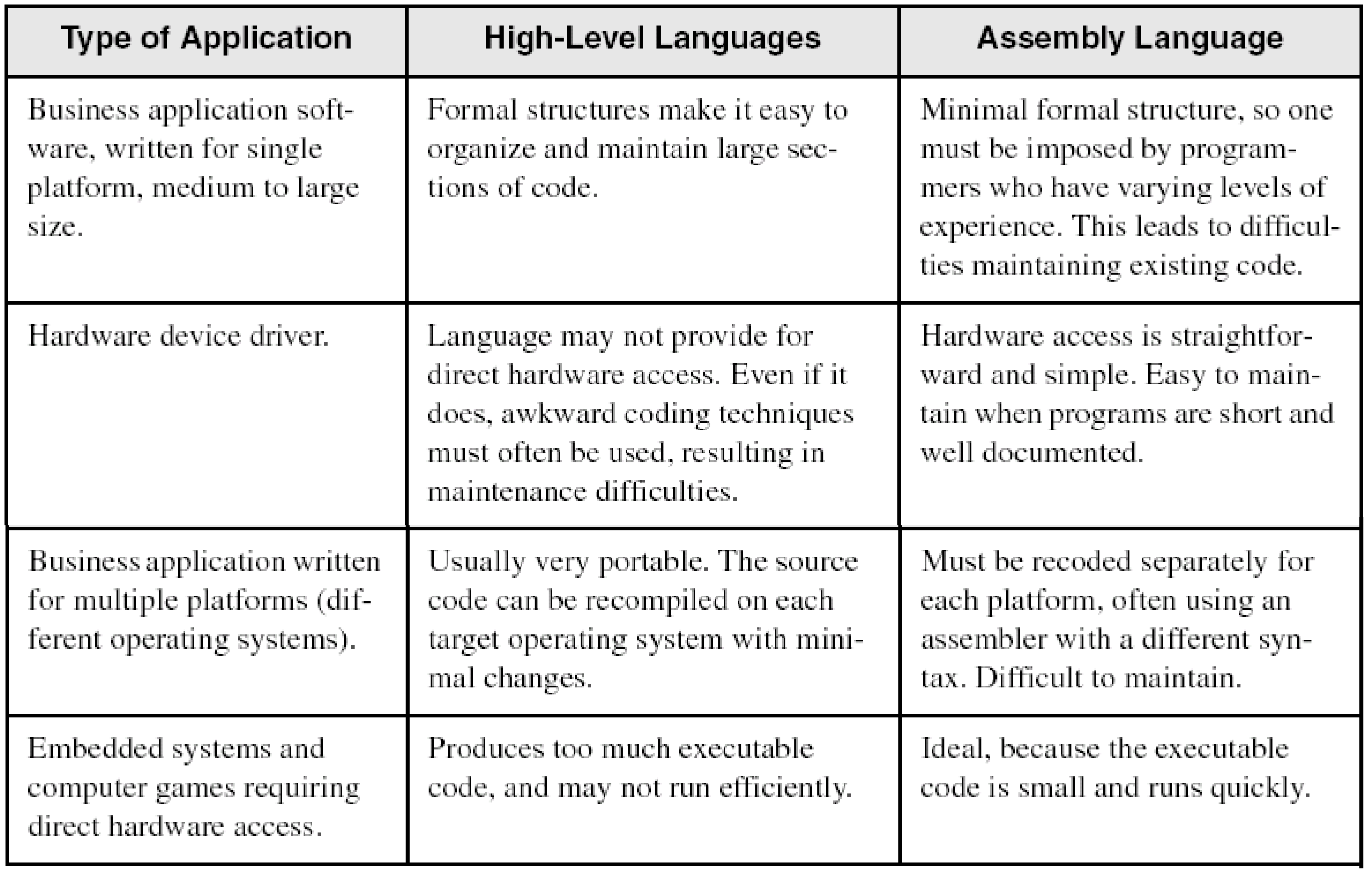
- Program development is faster
  - High-level statements: fewer instructions to code
- Program maintenance is easier
  - For the same above reasons
- Programs are portable
  - Contain few machine-dependent details
    - Can be used with little or no modifications on different machines
  - Compiler translates to the target machine language
  - However, Assembly language programs are not portable

Why Learn Assembly Language?

- Two main reasons:
  - Accessibility to system hardware
  - Space and time efficiency
- Accessibility to system hardware
  - Assembly Language is useful for implementing system software
  - Also useful for small embedded system applications

Assembly vs High-Level Languages

- Some representative types of applications:



Type of Application	High-Level Languages	Assembly Language
Business application software, written for single platform, medium to large size.	Formal structures make it easy to organize and maintain large sections of code.	Minimal formal structure, so one must be imposed by programmers who have varying levels of experience. This leads to difficulties maintaining existing code.
Hardware device driver.	Language may not provide for direct hardware access. Even if it does, awkward coding techniques must often be used, resulting in maintenance difficulties.	Hardware access is straightforward and simple. Easy to maintain when programs are short and well documented.
Business application written for multiple platforms (different operating systems).	Usually very portable. The source code can be recompiled on each target operating system with minimal changes.	Must be recoded separately for each platform, often using an assembler with a different syntax. Difficult to maintain.
Embedded systems and computer games requiring direct hardware access.	Produces too much executable code, and may not run efficiently.	Ideal, because the executable code is small and runs quickly.

Assembler

- Software tools are needed for editing, assembling, linking, and debugging assembly language programs
- An assembler is a program that converts source-code programs written in assembly language into object files in machine language
- Popular assemblers have emerged over the years for the Intel family of processors. These include …
  - TASM (Turbo Assembler from Borland)
  - NASM (Netwide Assembler for both Windows and Linux), and
  - GNU assembler distributed by the free software foundation
- You will use MASM (Macro Assembler from Microsoft)

Linker and Link Libraries

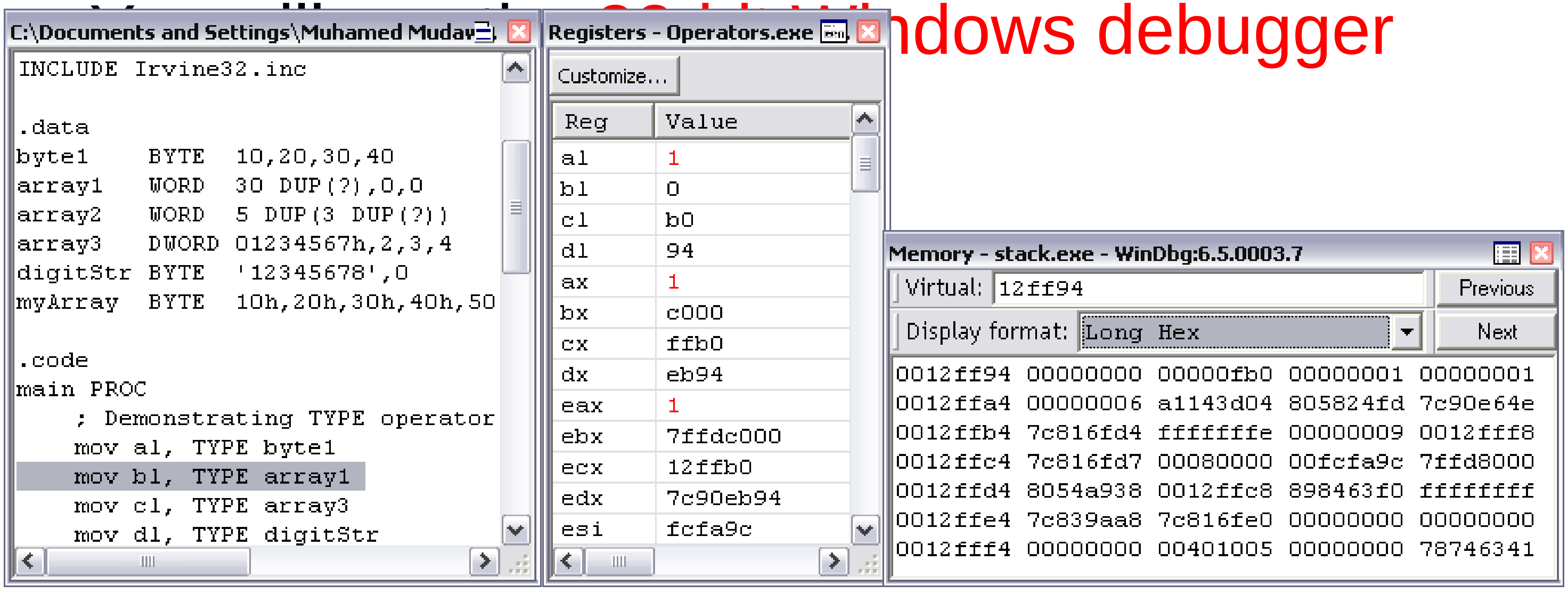
- You need a linker program to produce executable files
- It combines your program's object file created by the assembler with other object files and link libraries, and produces a single executable program

❖ LINK32.EXE is the linker program provided with the MASM distribution for linking 32-bit programs

- We will also use a link library for input and output
- Called Irvine32.lib developed by Kip Irvine
  - Works in Win32 console mode under MS-Windows

Debugger

- Allows you to trace the execution of a program
- Allows you to view code, memory, registers, etc.
- ndows debugger



C:\Documents and Settings\Muhamed Mudav

```
INCLUDE Irvine32.inc

.data
byte1    BYTE  10,20,30,40
array1   WORD  30 DUP(?),0,0
array2   WORD  5 DUP(3 DUP(?))
array3   DWORD 01234567h,2,3,4
digitStr BYTE  '12345678',0
myArray  BYTE  10h,20h,30h,40h,50

.code
main PROC
    ; Demonstrating TYPE operator
    mov al, TYPE byte1
    mov bl, TYPE array1
    mov cl, TYPE array3
    mov dl, TYPE digitStr
```

Registers - Operators.exe

Reg	Value
al	1
bl	0
cl	b0
dl	94
ax	1
bx	c000
cx	ffb0
dx	eb94
eax	1
ebx	7ffdc000
ecx	12ffb0
edx	7c90eb94
esi	fcfa9c

Memory - stack.exe - WinDbg:6.5.0003.7

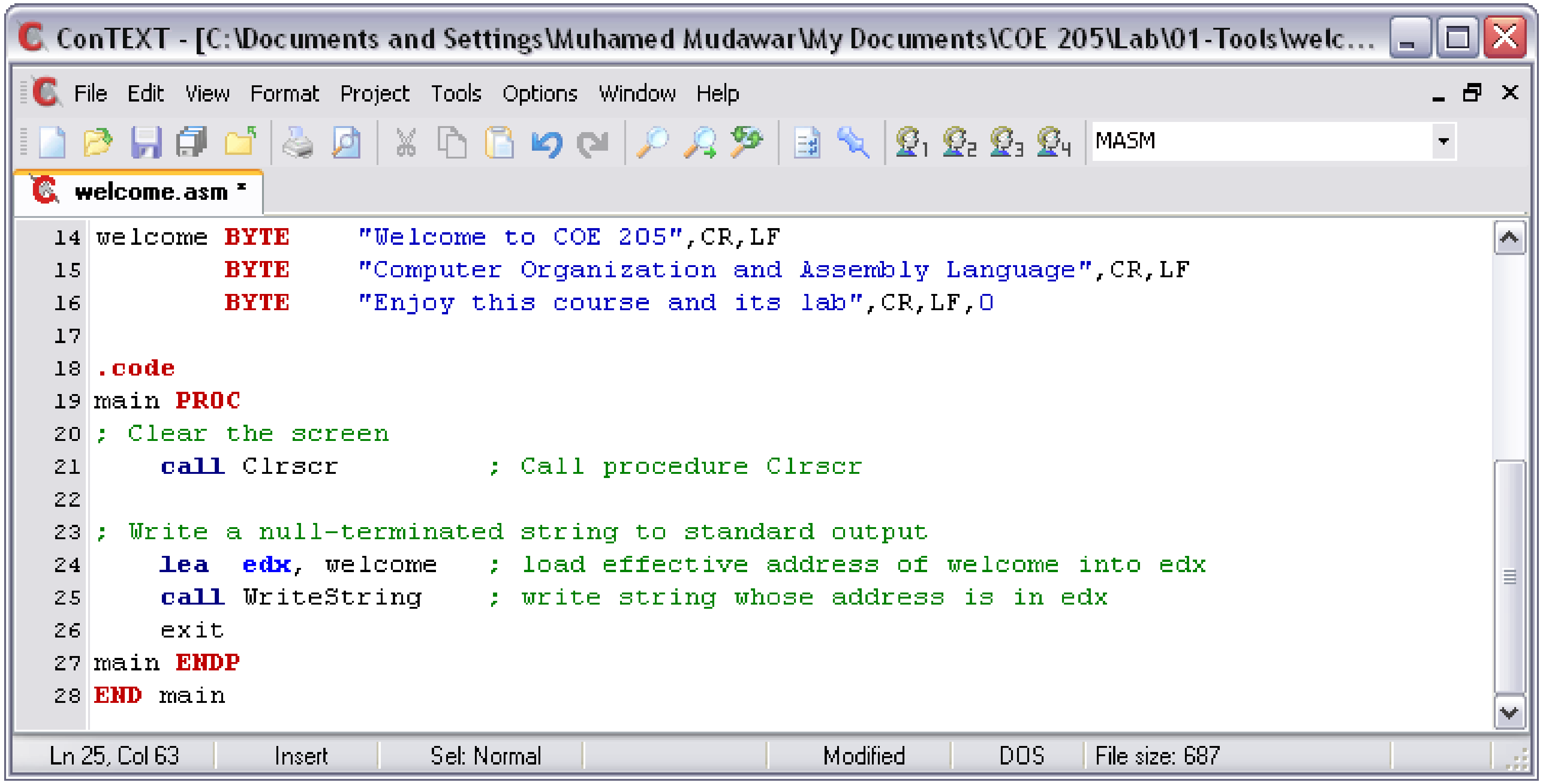
Virtual: 12ff94

Display format: Long Hex

```
0012ff94 00000000 00000fb0 00000001 00000001
0012ffa4 00000006 a1143d04 805824fd 7c90e64e
0012ffb4 7c816fd4 fffffffe 00000009 0012fff8
0012ffc4 7c816fd7 00080000 00fcfa9c 7ffd8000
0012ffd4 8054a938 0012ffc8 898463f0 ffffffff
0012ffe4 7c839aa8 7c816fe0 00000000 00000000
0012fff4 00000000 00401005 00000000 78746341
```

Editor

- Allows you to create assembly language source files



ConTEXT - [C:\Documents and Settings\Muhamed Mudawar\My Documents\COE 205\Lab\01-Tools\welc...

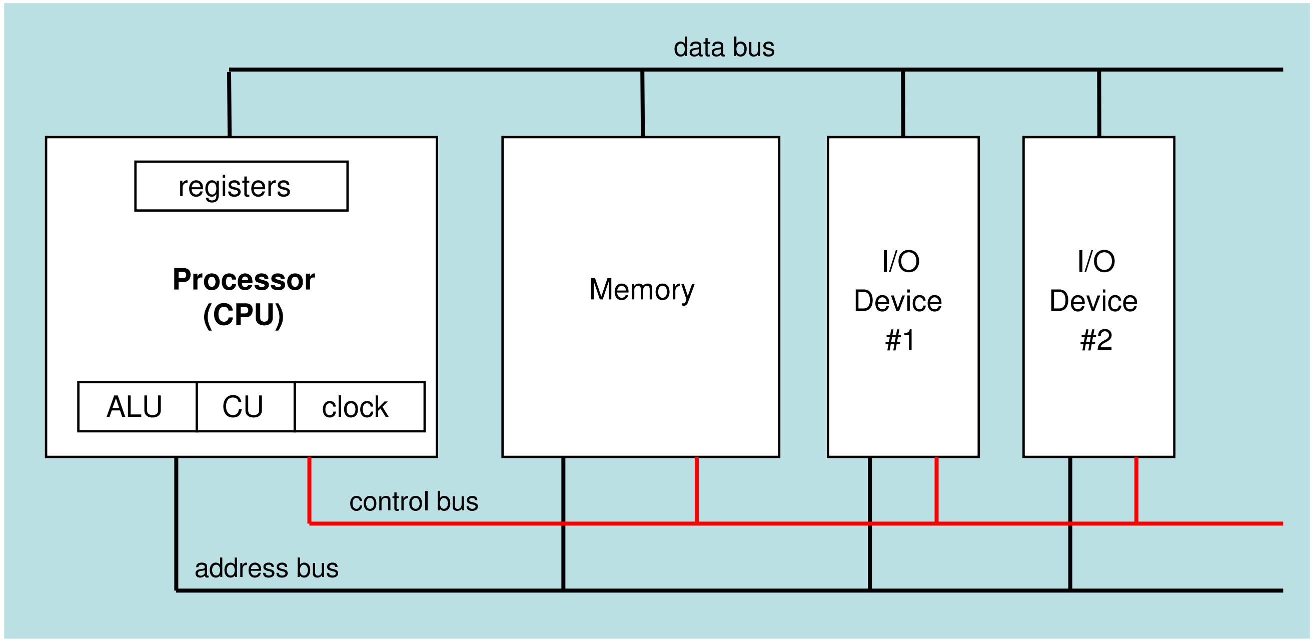
```
welcome BYTE    "Welcome to COE 205",CR,LF
        BYTE    "Computer Organization and Assembly Language",CR,LF
        BYTE    "Enjoy this course and its lab",CR,LF,0

.code
main PROC
; Clear the screen
    call Clrscr          ; Call procedure Clrscr

; Write a null-terminated string to standard output
    lea  edx, welcome    ; load effective address of welcome into edx
    call WriteString     ; write string whose address is in edx
    exit
main ENDP
END main
```

Ln 25, Col 63 | Insert | Sel: Normal | Modified | DOS | File size: 687

BASIC MICROCOMPUTER DESIGN

Basic Computer Organization

- Since the 1940's, computers have 3 classic components:
  - Processor, called also the CPU (Central Processing Unit)
  - Memory and Storage Devices
  - I/O Devices
- Interconnected with one or more buses
- Bus consists of
  - Data Bus
  - Address Bus
  - Control Bus



Processor (CPU)

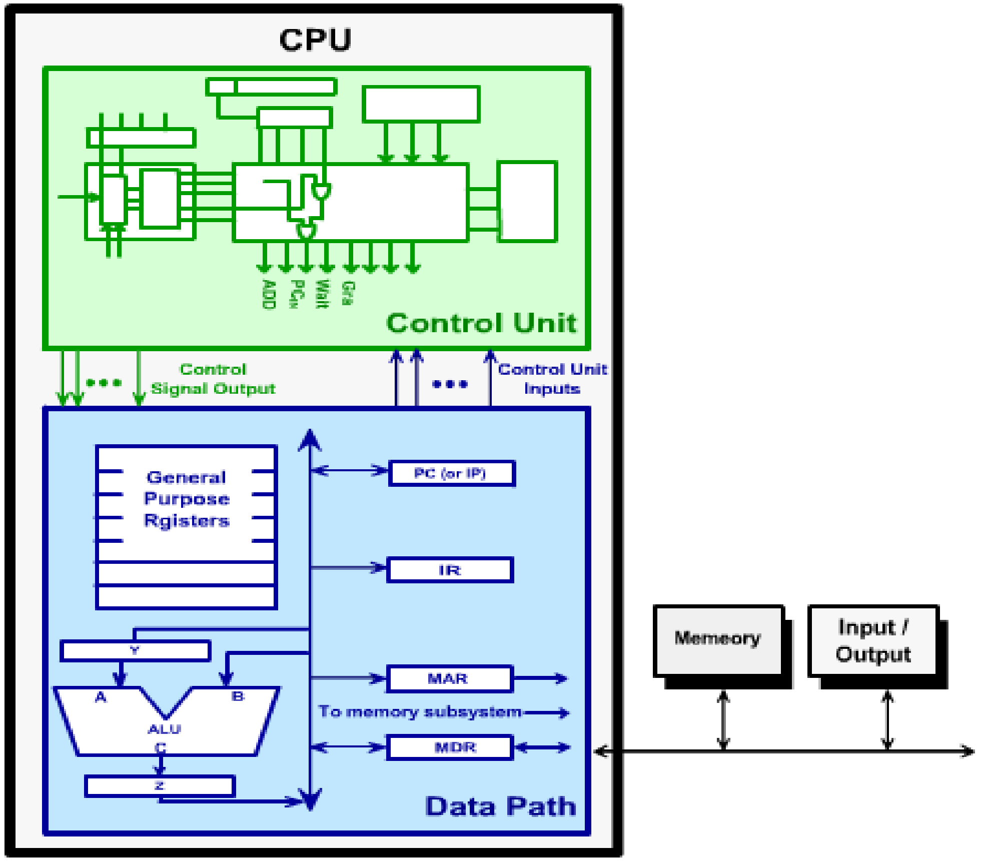
- Processor consists of
  - Datapath
    - ALU
    - Registers
  - Control unit

Flash Movie

- ALU
  - Performs arithmetic and logic instructions

Flash Movie

- Control unit (CU)
  - Generates the control signals

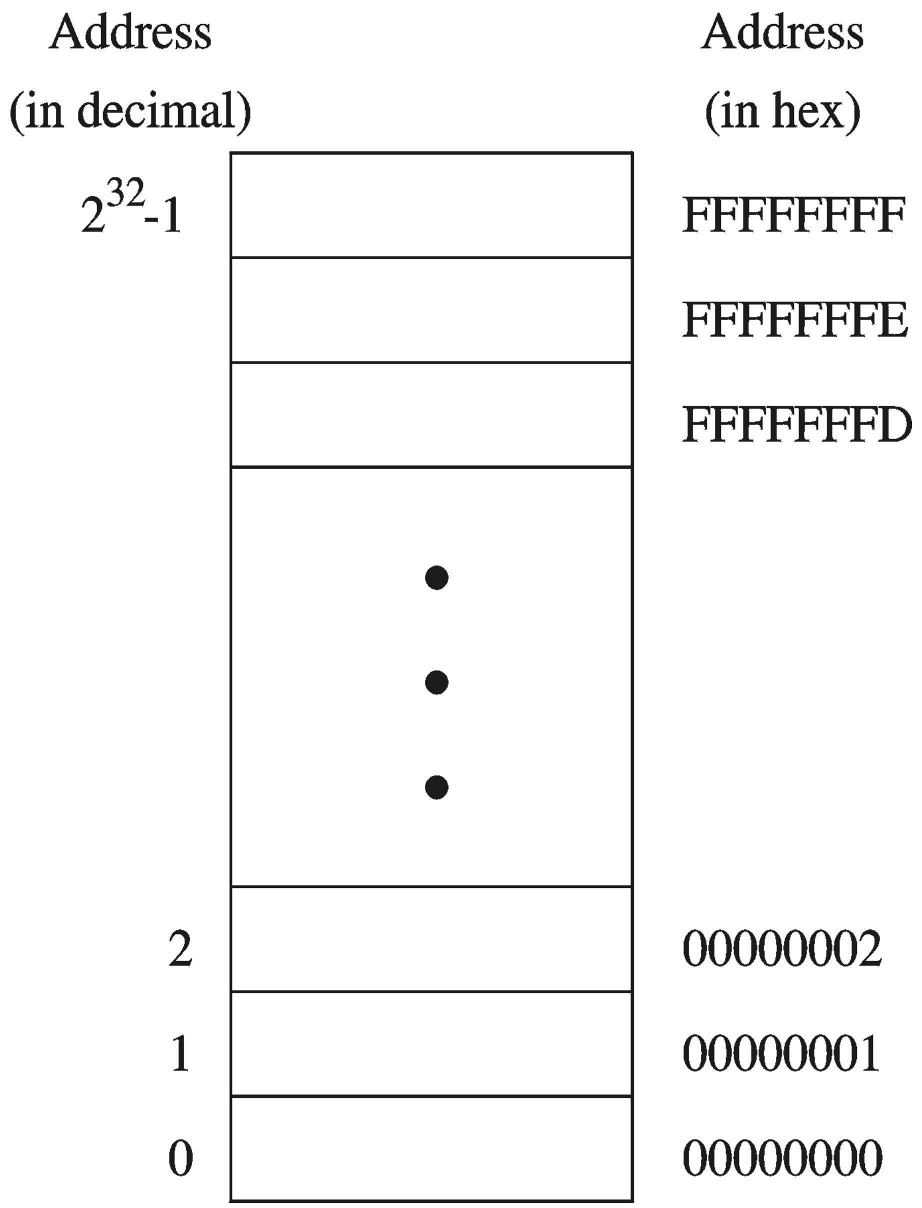


- Implementation varies from one processor to another

Memory

- Ordered sequence of bytes
  - The sequence number is called the memory address
- Byte addressable memory
  - Each byte has a unique address
  - Supported by almost all processors
- Physical address space
  - Determined by the address bus width
  - Pentium has a 32-bit address bus
    - Physical address space = **4GB = 2^{32} bytes**
  - Itanium with a 64-bit address bus can support
    - Up to **2^{64} bytes** of physical address space

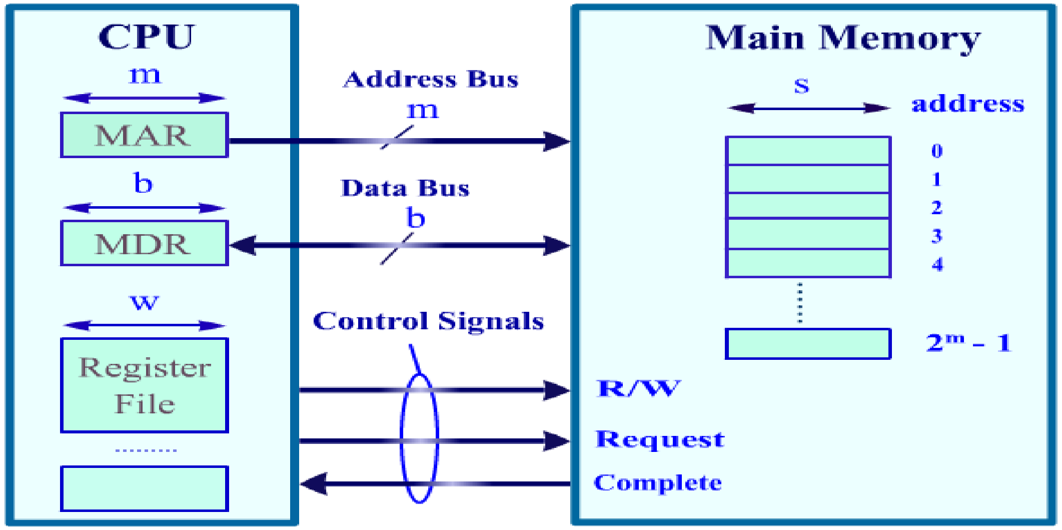
Address Space



Address Space is the set of memory locations (bytes) that can be addressed

CPU Memory Interface

- Address Bus
  - Memory address is put on address bus
  - If memory address = *m* bits then 2^m locations are addressed
- Data Bus: b-bit bi-directional bus
  - Data can be transferred in both directions on the data bus
- Control Bus
  - Signals control transfer of data
  - Read request
  - Write request
  - Complete transfer
-

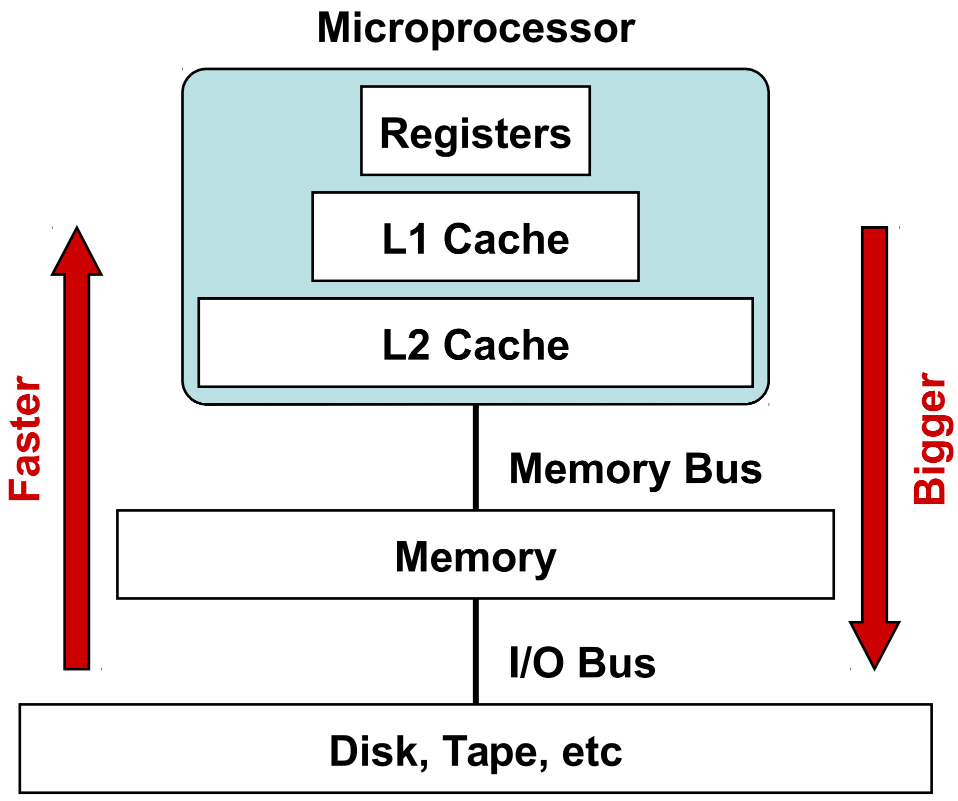


The Need for a Memory Hierarchy

- Widening speed gap between CPU and main memory
  - Processor operation takes less than 1 ns
  - Main memory requires more than 50 ns to access
- Each instruction involves at least one memory access
  - One memory access to fetch the instruction
  - Additional memory accesses for instructions involving memory data access
- Memory bandwidth limits the instruction execution rate
- Cache memory can help bridge the CPU-memory gap
- Cache memory is small in size but fast

Typical Memory Hierarchy

- Registers are at the top of the hierarchy
  - Typical size < 1 KB
  - Access time < 0.5 ns
- Level 1 Cache (8 – 64 KB)
  - Access time: 0.5 – 1 ns
- L2 Cache (512KB – 8MB)
  - Access time: 2 – 10 ns
- Main Memory (1 – 2 GB)
  - Access time: 50 – 70 ns
- Disk Storage (> 200 GB)
  - Access time: milliseconds



Intel Microprocessors

- Intel introduced the 8086 microprocessor in 1979
- 8086, 8087, 8088, and 80186 processors
  - 16-bit processors with 16-bit registers
  - 16-bit data bus and 20-bit address bus
    - Physical address space = 2^{20} bytes = 1 MB
  - 8087 Floating-Point co-processor
  - Uses segmentation and real-address mode to address memory
    - Each segment can address 2^{16} bytes = 64 KB

Intel 80286 and 80386 Processors

- 80286 was introduced in 1982
  - 24-bit address bus $\Rightarrow 2^{24}$ bytes = 16 MB address space
  - Introduced protected mode
    - Segmentation in protected mode is different from the real mode
- 80386 was introduced in 1985
  - First 32-bit processor with 32-bit general-purpose registers
  - First processor to define the IA-32 architecture
  - 32-bit data bus and 32-bit address bus
  - 2^{32} bytes $\Rightarrow$ 4 GB address space
  - Introduced paging, virtual memory, and the flat memory model
    - Segmentation can be turned off

Intel 80486 and Pentium Processors

- 80486 was introduced 1989
  - Improved version of Intel 80386
  - On-chip Floating-Point unit (DX versions)
  - On-chip unified Instruction/Data Cache (8 KB)
  - Uses Pipelining: can execute up to 1 instruction per clock cycle
- Pentium (80586) was introduced in 1993
  - Wider 64-bit data bus, but address bus is still 32 bits
  - Two execution pipelines: U-pipe and V-pipe
    - Superscalar performance: can execute 2 instructions per clock cycle
  - Separate 8 KB instruction and 8 KB data caches
  - ◇ MMX instructions (later models) for multimedia applications

CISC and RISC

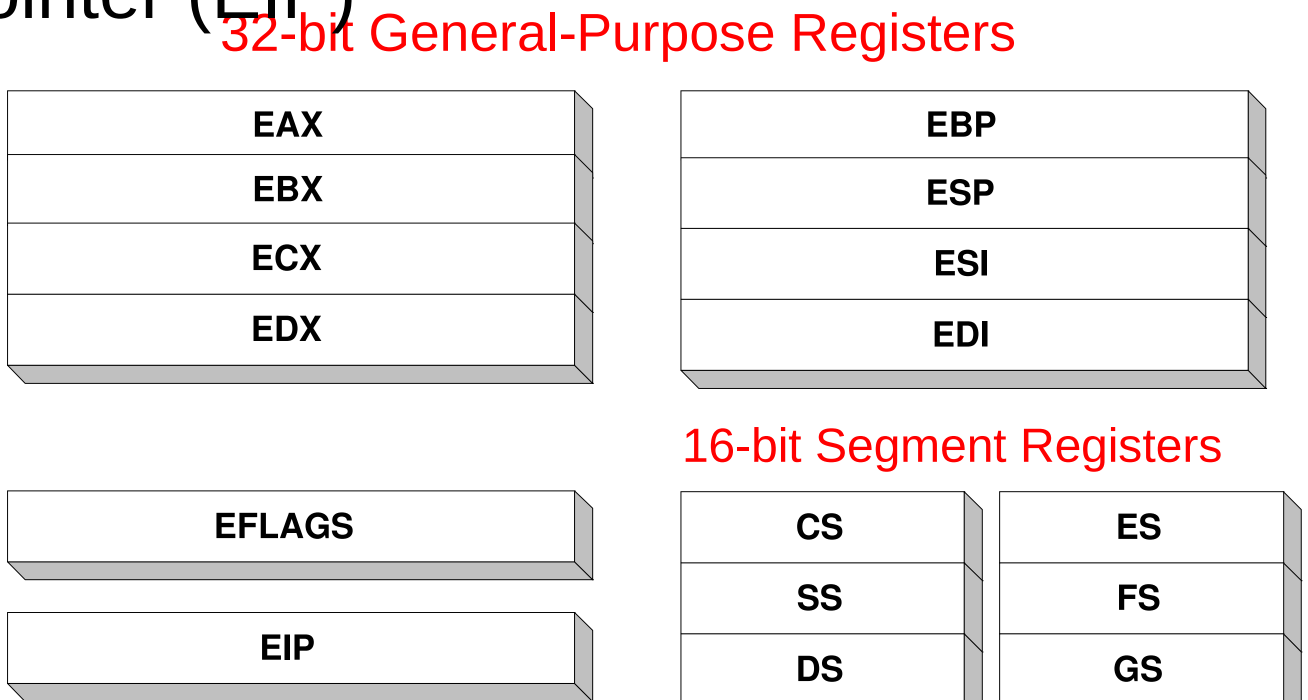
- CISC – Complex Instruction Set Computer
  - Large and complex instruction set
  - Variable width instructions
  - Requires microcode interpreter
    - Each instruction is decoded into a sequence of micro-operations
  - Example: Intel x86 family
- RISC – Reduced Instruction Set Computer
  - Small and simple instruction set
  - All instructions have the same width
  - Simpler instruction formats and addressing modes
  - Examples: ARM, MIPS, PowerPC, SPARC, etc.

Next ...

- Intel Microprocessors
- ❖IA-32 Registers
- Instruction Execution Cycle
- IA-32 Memory Management

Basic Program Execution Registers

- Registers are high speed memory inside the CPU
  - Eight 32-bit general-purpose registers
  - Six 16-bit segment registers
  - Processor Status Flags (EFLAGS) and Instruction Pointer (EIP)

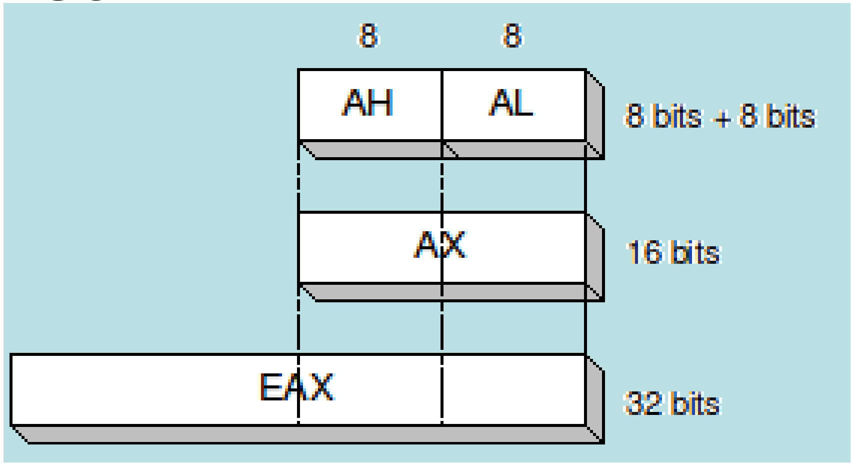


General-Purpose Registers

- Used primarily for arithmetic and data movement
  - **`mov eax, 10`** move constant 10 into register eax
- Specialized uses of Registers
  - EAX – Accumulator register
    - Automatically used by multiplication and division instructions
  - ECX – Counter register
    - Automatically used by LOOP instructions
  - ESP – Stack Pointer register
    - Used by PUSH and POP instructions, points to top of stack
  - ESI and EDI – Source Index and Destination Index register
    - Used by string instructions
  - EBP – Base Pointer register
    - Used to reference parameters and local variables on the stack

Accessing Parts of Registers

- EAX, EBX, ECX, and EDX are 32-bit Extended registers
  - Programmers can access their 16-bit and 8-bit parts
  - Lower 16-bit of EAX is named AX
  - AX is further divided into
    - AL = lower 8 bits
    - AH = upper 8 bits
- ESI, EDI, EBP, ESP have only 16-bit names for lower half

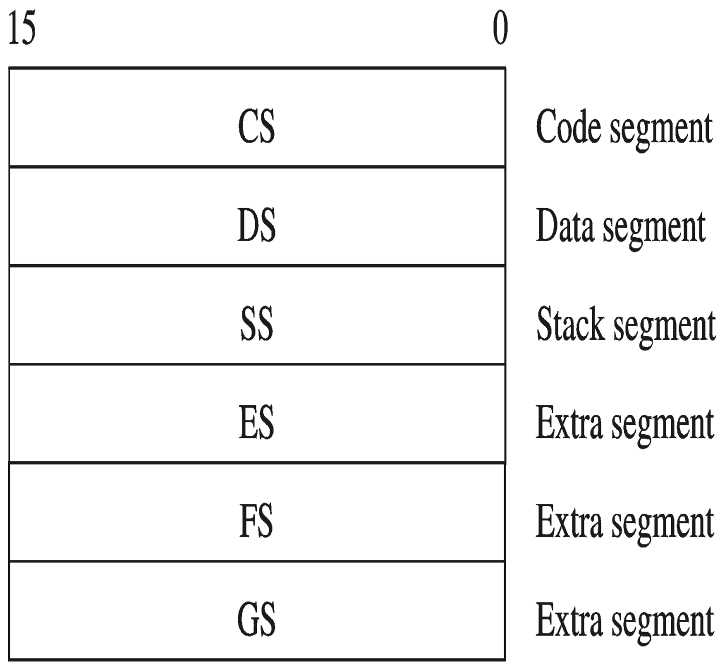


32-bit	16-bit	8-bit (high)	8-bit (low)
EAX	AX	AH	AL
EBX	BX	BH	BL
ECX	CX	CH	CL
EDX	DX	DH	DL

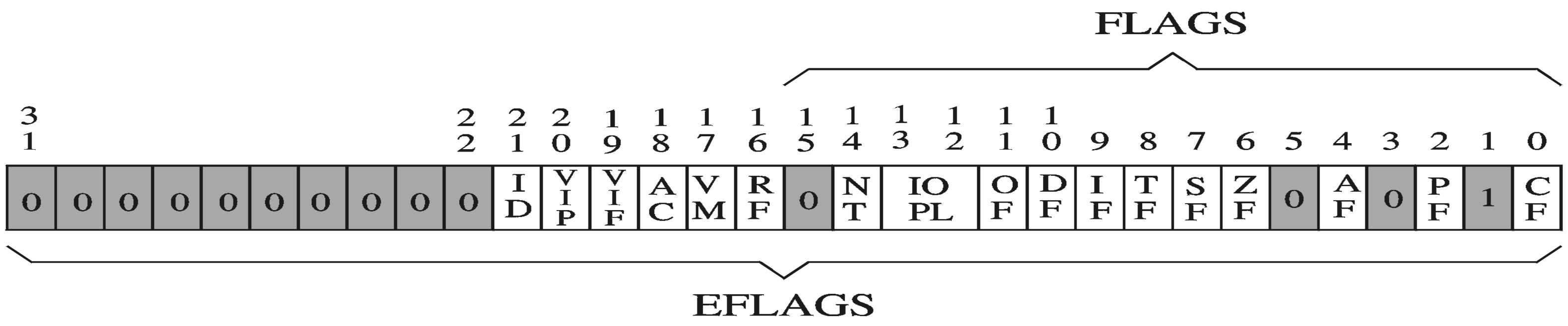
32-bit	16-bit
ESI	SI
EDI	DI
EBP	BP
ESP	SP

Special-Purpose & Segment Registers

- EIP = Extended Instruction Pointer
  - Contains address of next instruction to be executed
- EFLAGS = Extended Flags Register
  - Contains status and control flags
  - Each flag is a single binary bit
- Six 16-bit Segment Registers
  - Support segmented memory
  - Six segments accessible at a time
  - Segments contain distinct contents
    - Code
    - Data
    - Stack



EFLAGS Register



Status flags

CF = Carry flag
PF = Parity flag
AF = Auxiliary carry flag
ZF = Zero flag
SF = Sign flag
OF = Overflow flag

Control flags

DF = Direction flag

System flags

TF = Trap flag
IF = Interrupt flag
IOPL = I/O privilege level
NT = Nested task
RF = Resume flag
VM = Virtual 8086 mode
AC = Alignment check
VIF = Virtual interrupt flag
VIP = Virtual interrupt pending
ID = ID flag

- Status Flags
  - Status of arithmetic and logical operations
- Control and System flags
  - Control the CPU operation
- Programs can set and clear individual bits in the EFLAGS register

Status Flags

- Carry Flag
  - Set when unsigned arithmetic result is out of range
- Overflow Flag
  - Set when signed arithmetic result is out of range
- Sign Flag
  - Copy of sign bit, set when result is negative
- Zero Flag
  - Set when result is zero
- Auxiliary Carry Flag
  - Set when there is a carry from bit 3 to bit 4
- Parity Flag
  - Set when parity is even
  - Least-significant byte in result contains even number of 1s

Next ...

- Intel Microprocessors
- IA-32 Registers

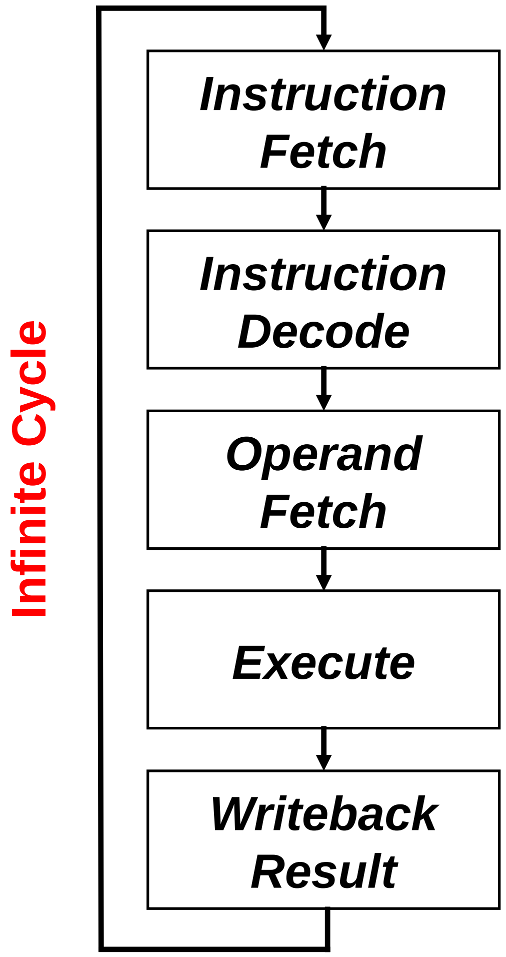
❖ Instruction Execution Cycle

- IA-32 Memory Management

Fetch-Execute Cycle

- Each machine language instruction is first fetched from the memory and stored in an **Instruction Register** (IR).
- The address of the instruction to be fetched is stored in a register called **Program Counter** or simply PC. In some computers this register is called the **Instruction Pointer** or IP.
- After the instruction is fetched, the PC (or IP) is incremented to point to the address of the next instruction.
- The fetched instruction is decoded (to determine what needs to be done) and executed by the CPU.

Instruction Execute Cycle



Obtain instruction from program storage

Determine required actions and instruction size

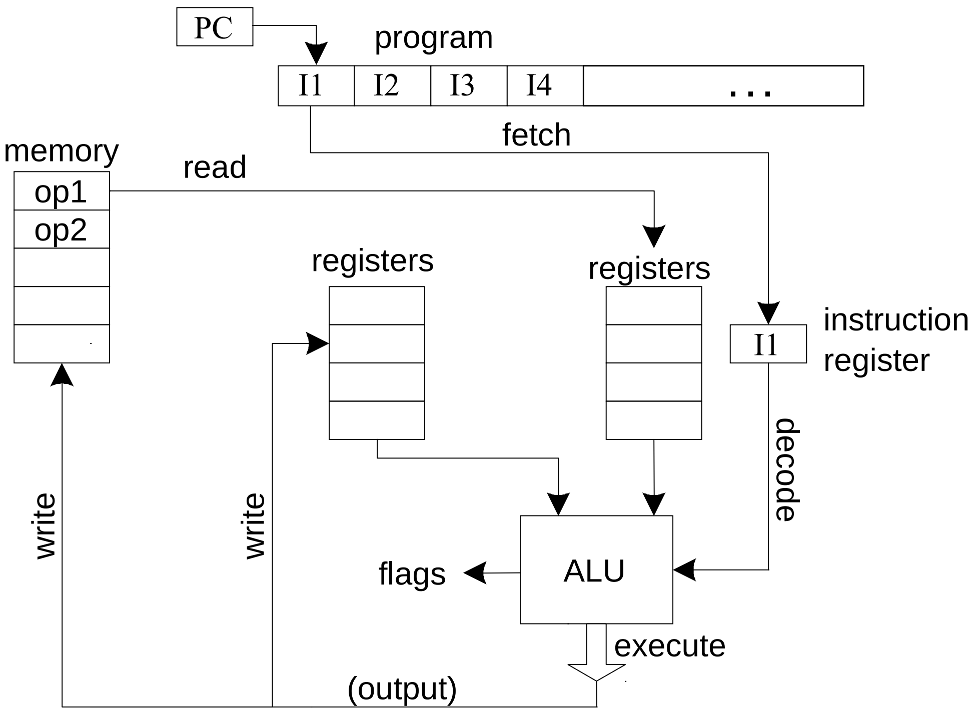
Locate and obtain operand data

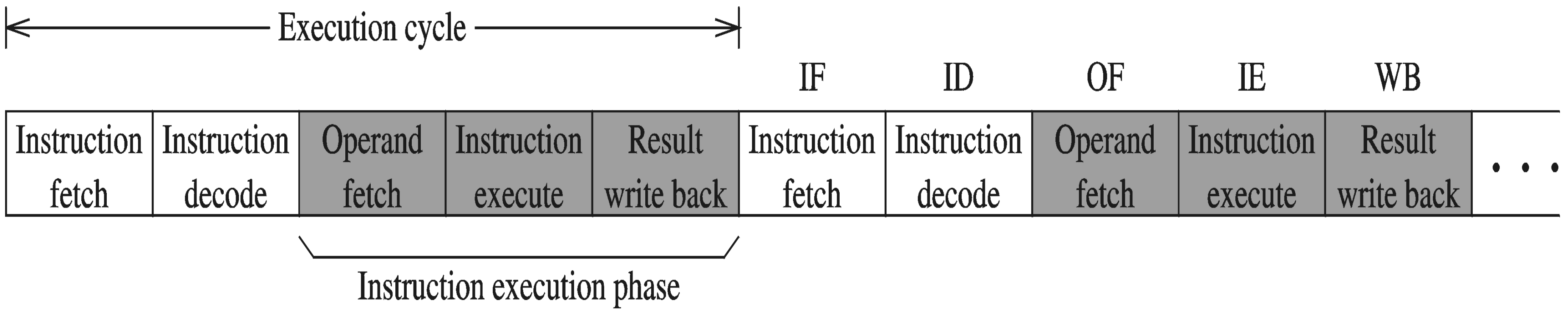
Compute result value and status

Deposit results in storage for later use

Instruction Execution Cycle – cont'd

- Instruction Fetch
- Instruction Decode
- Operand Fetch
- Execute
- Result Writeback





Next ...

- Intel Microprocessors
- IA-32 Registers
- Instruction Execution Cycle

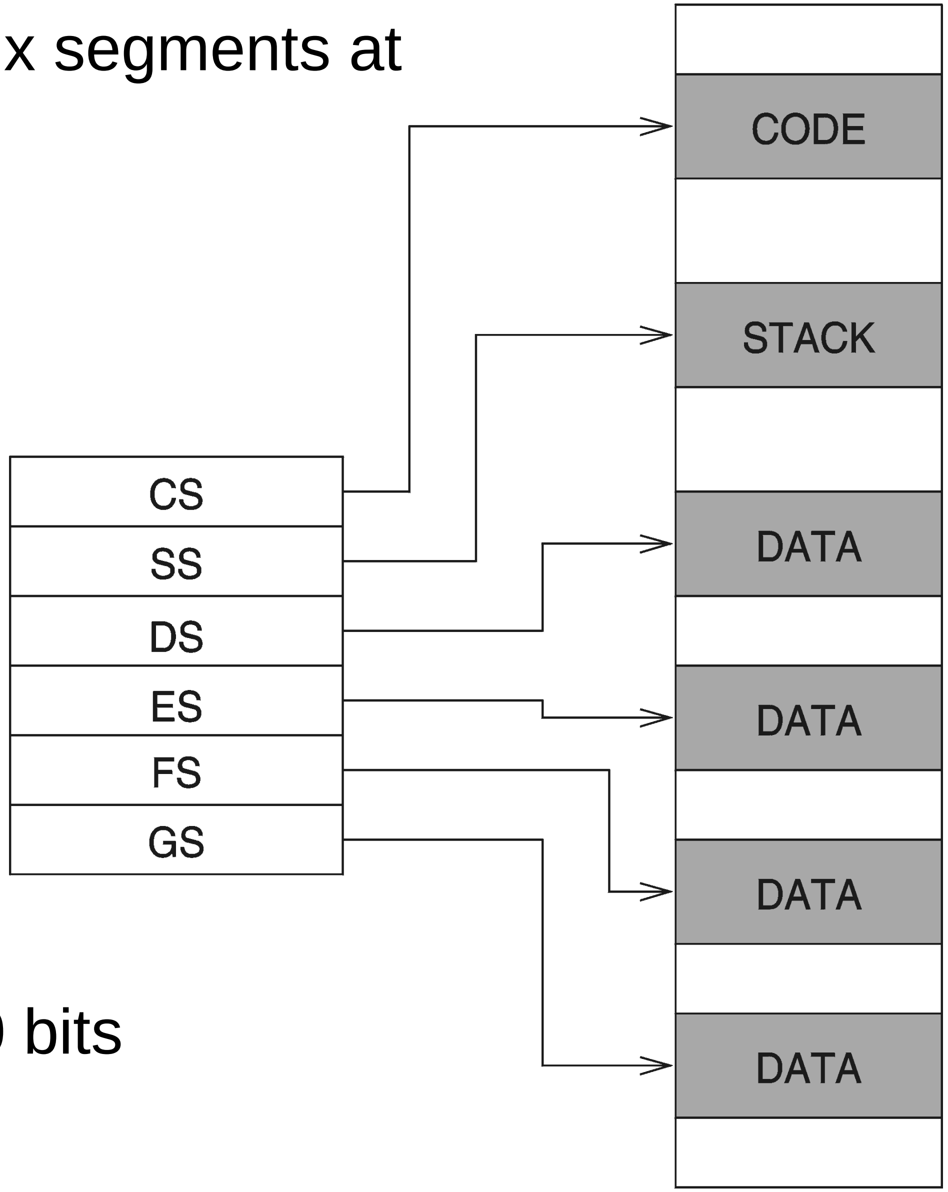
❖IA-32 Memory Management

Modes of Operation

- 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
- Protected mode (introduced with the 80386 processor)
  - 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
  - Native mode used by Windows NT, 2000, XP, and Linux
- Virtual 8086 mode
  - Processor runs in protected mode, and creates a virtual 8086 machine with 1 MB of address space for each running program

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



Logical to Linear Address Translation

Linear address = Segment × 10 (hex) + Offset

Example:

segment = A1F0 (hex)

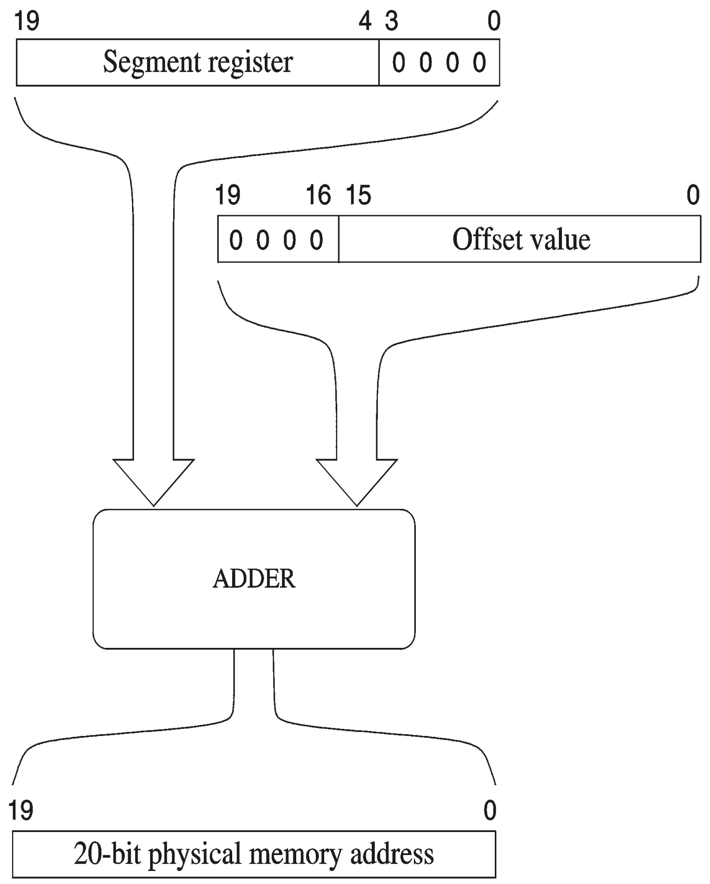
offset = 04C0 (hex)

logical address = A1F0:04C0 (hex)

what is the linear address?

Solution:

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



Your turn . . .

What linear address corresponds to logical address 028F:0030?

Solution: 028F0 + 0030 = 02920 (hex)

Always use hexadecimal notation for addresses

YOUR TASK

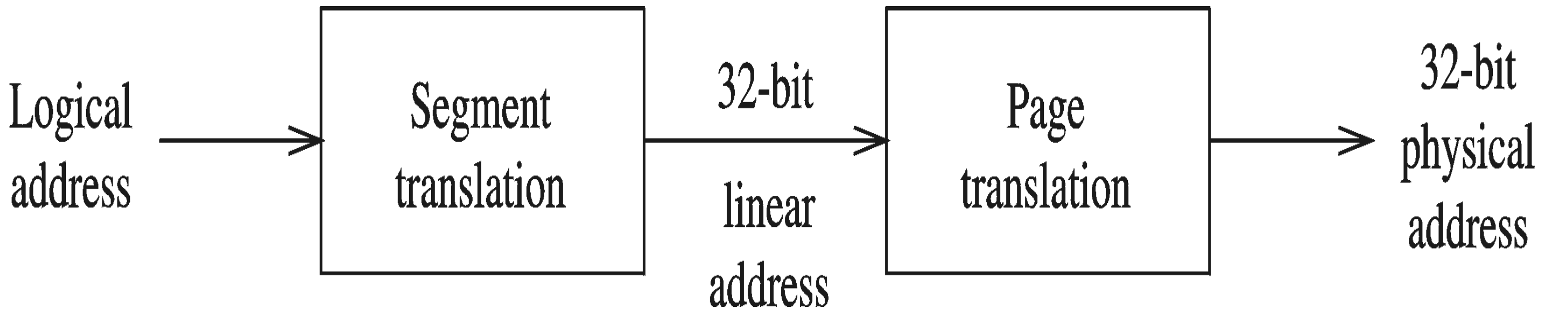
- CALCULATE PHYSICAL ADDRESS FOR FOLLOWING LOGICAL ADDRESS 8860:1238
- CALCULATE PHYSICAL ADDRESS FOR FOLLOWING LOGICAL ADDRESS 2160:1120

Flat Memory Model

- Modern operating systems turn segmentation off
- Each program uses one 32-bit linear address space
  - Up to 2^{32} = 4 GB of memory can be addressed
  - Segment registers are defined by the operating system
  - All segments are mapped to the same linear address space
- In assembly language, we use .MODEL flat directive
  - To indicate the Flat memory model
- A linear address is also called a virtual address
  - Operating system maps virtual address onto physical addresses
  - Using a technique called paging

Protected Mode Architecture

- Logical address consists of
  - 16-bit segment selector (CS, SS, DS, ES, FS, GS)
  - 32-bit offset (EIP, ESP, EBP, ESI ,EDI, EAX, EBX, ECX, EDX)
- Segment unit translates logical address to linear address
  - Using a segment descriptor table
  - Linear address is 32 bits (called also a virtual address)
- Paging unit translates linear address to physical address
  - Using a page directory and a page table


Logical address
Segment translation
32-bit
linear address
Page translation
32-bit physical address