

Basics

Assembly Language Programming

Dr Emmanuel Ahene

Outline

- ❖ Welcome to Assembly
- ❖ Assembly-, Machine-, and High-Level Languages
- ❖ Assembly Language Programming Tools
- ❖ Programmer's View of a Computer System
- ❖ Basic Computer Organization

Goals and Required Background

Grading Policy

- ❖ Attendance 5%
- ❖ Quizzes 5 %
- ❖ Midsem Exam 20%
- ❖ Final Exam 70%

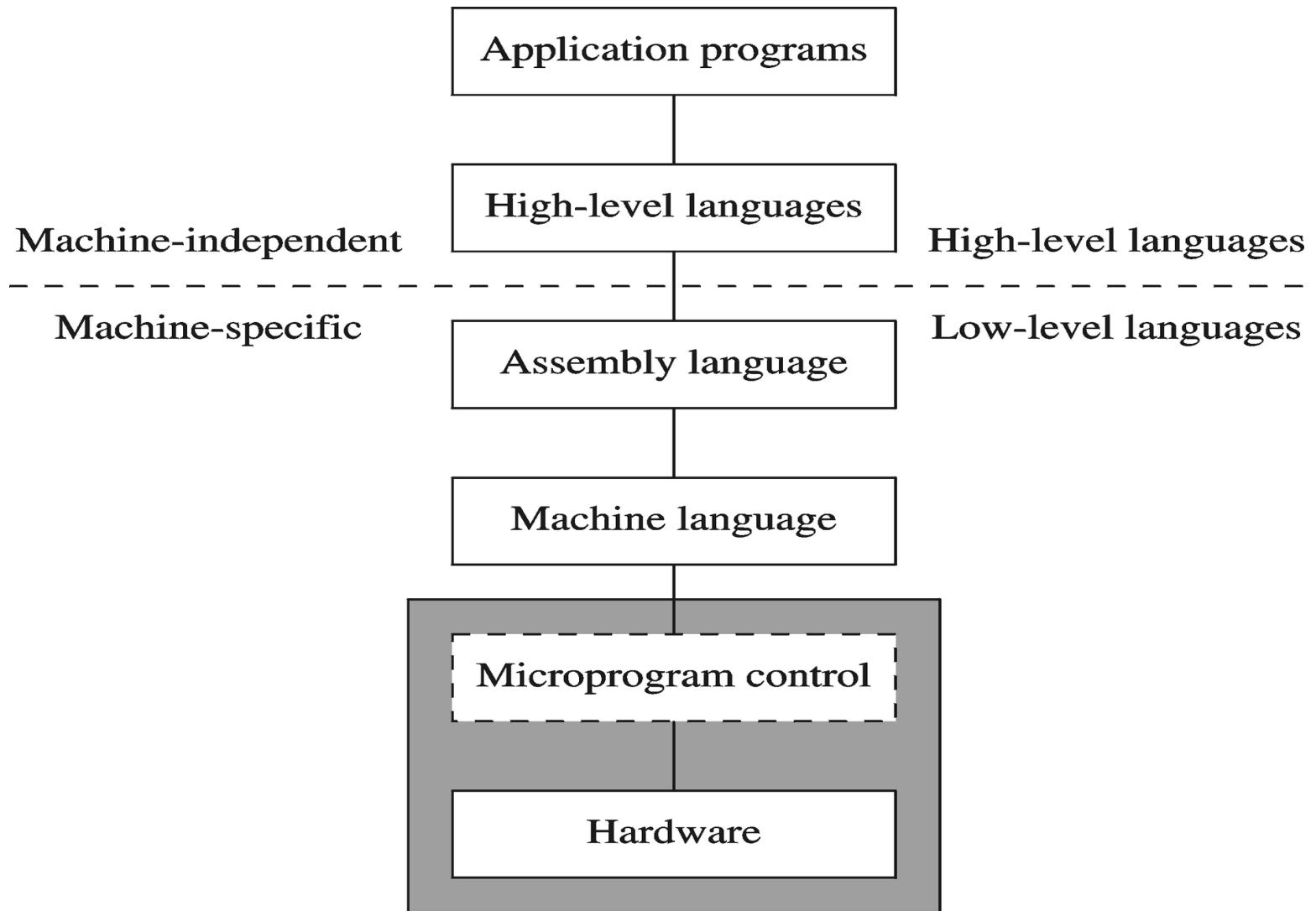
Next ...

- ❖ Welcome
- ❖ Assembly-, Machine-, and High-Level Languages
- ❖ Assembly Language Programming Tools
- ❖ Programmer's View of a Computer System
- ❖ Basic Computer Organization
- ❖ Data Representation

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?

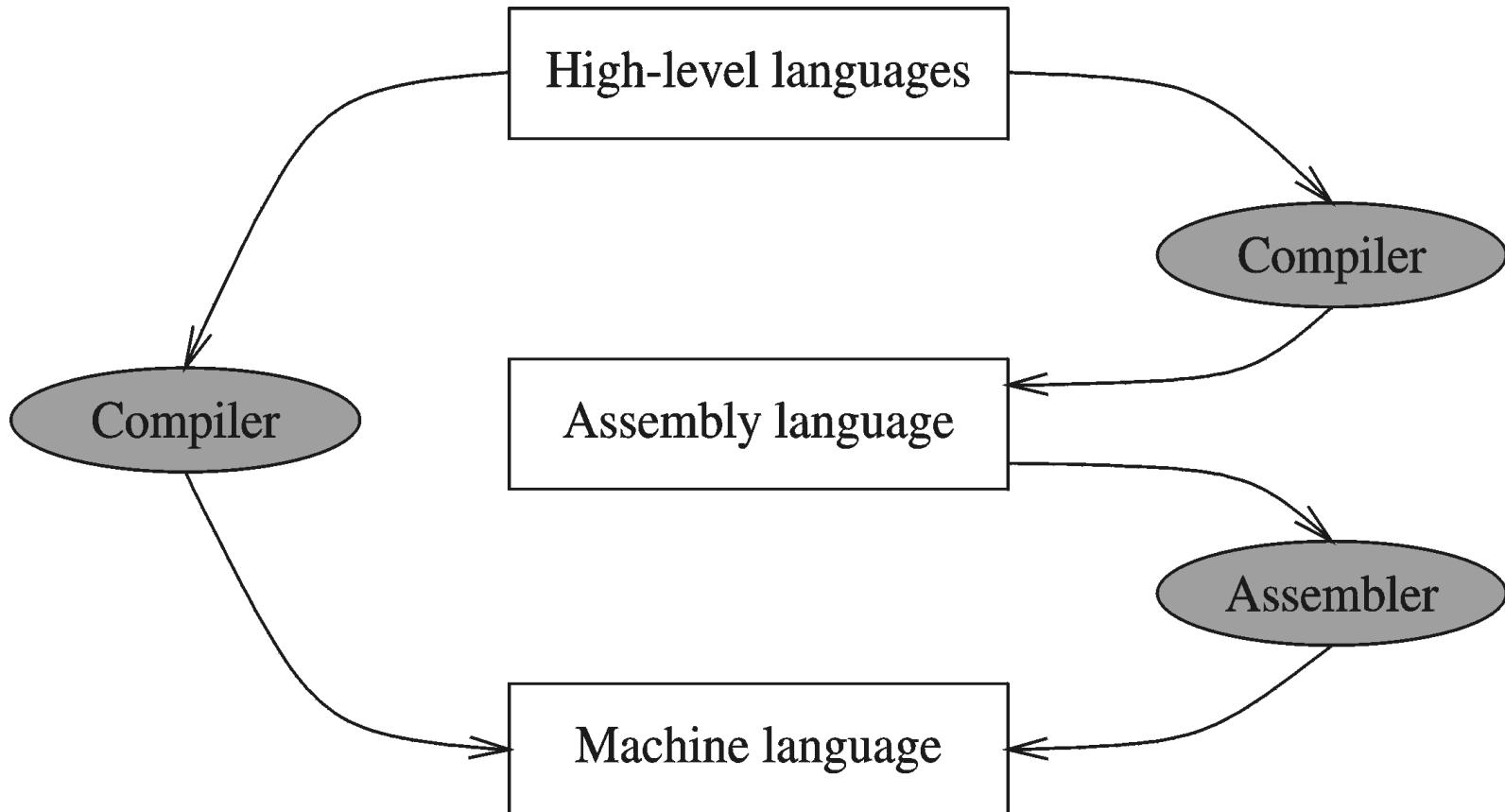
A Hierarchy of Languages



Assembly and Machine Language

- ❖ Machine language
 - ◊ Native to a processor: executed directly by hardware
 - ◊ Instructions consist of binary code: 1s and 0s
- ❖ Assembly language
 - ◊ A programming language that uses symbolic names to represent operations, registers and memory locations.
 - ◊ 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



Instructions and Machine Language

- ❖ Each command of a program is called an **instruction** (it instructs the computer what to do).
- ❖ Computers only deal with binary data, hence the instructions must be in binary format (0s and 1s) .
- ❖ The set of all instructions (in binary form) makes up the computer's **machine language**. This is also referred to as the **instruction set**.

Instruction Fields

- ❖ Machine language instructions usually are made up of several fields. Each field specifies different information for the computer. The major two fields are:
- ❖ **Opcode** field which stands for operation code and it specifies the particular operation that is to be performed.
 - ◊ Each operation has its unique opcode.
- ❖ **Operands** fields which specify where to get the source and destination operands for the operation specified by the opcode.
 - ◊ The source/destination of operands can be a constant, the memory or one of the general-purpose registers.

Assembly vs. Machine Code

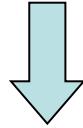
Instruction Address	Machine Code	Assembly Instruction
0005	B8 0001	MOV AX, 1
0008	B8 0002	MOV AX, 2
000B	B8 0003	MOV AX, 3
000E	B8 0004	MOV AX, 4
0011	BB 0001	MOV BX, 1
0014	B9 0001	MOV CX, 1
0017	BA 0001	MOV DX, 1
001A	8B C3	MOV AX, BX
001C	8B C1	MOV AX, CX
001E	8B C2	MOV AX, DX
0020	83 C0 01	ADD AX, 1
0023	83 C0 02	ADD AX, 2
0026	03 C3	ADD AX, BX
0028	03 C1	ADD AX, CX
002A	03 06 0000	ADD AX, i
002E	83 E8 01	SUB AX, 1
0031	2B C3	SUB AX, BX
0033	05 1234	ADD AX, 1234h

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

Mapping Between Assembly Language and HLL

- ❖ Translating HLL programs to machine language programs is not a one-to-one mapping
 - ❖ A HLL instruction (usually called a statement) will be translated to one or more machine language instructions
-

Mapping between some C instructions and 8086 assembly language

Instruction Class	C	Assembly Language
Data Movement	a = 5	MOV a, 5
Arithmetic/Logic	b = a + 5	MOV ax, a ADD ax, 5 MOV b, ax
Control Flow	goto LBL	JMP LBL

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?

- ❖ Accessibility to system hardware
 - ◊ Assembly Language is useful for implementing system software
 - ◊ Also useful for small embedded system applications
- ❖ Space and Time efficiency
 - ◊ Understanding sources of program inefficiency
 - ◊ Tuning program performance
 - ◊ Writing compact code
- ❖ Writing assembly programs gives the computer designer the needed deep understanding of the instruction set and how to design one
- ❖ To be able to write compilers for HLLs, we need to be expert with the machine language. Assembly programming provides this experience

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.

Next ...

- ❖ Welcome
- ❖ Assembly-, Machine-, and High-Level Languages
- ❖ **Assembly Language Programming Tools**
- ❖ Programmer's View of a Computer System
- ❖ Basic Computer Organization

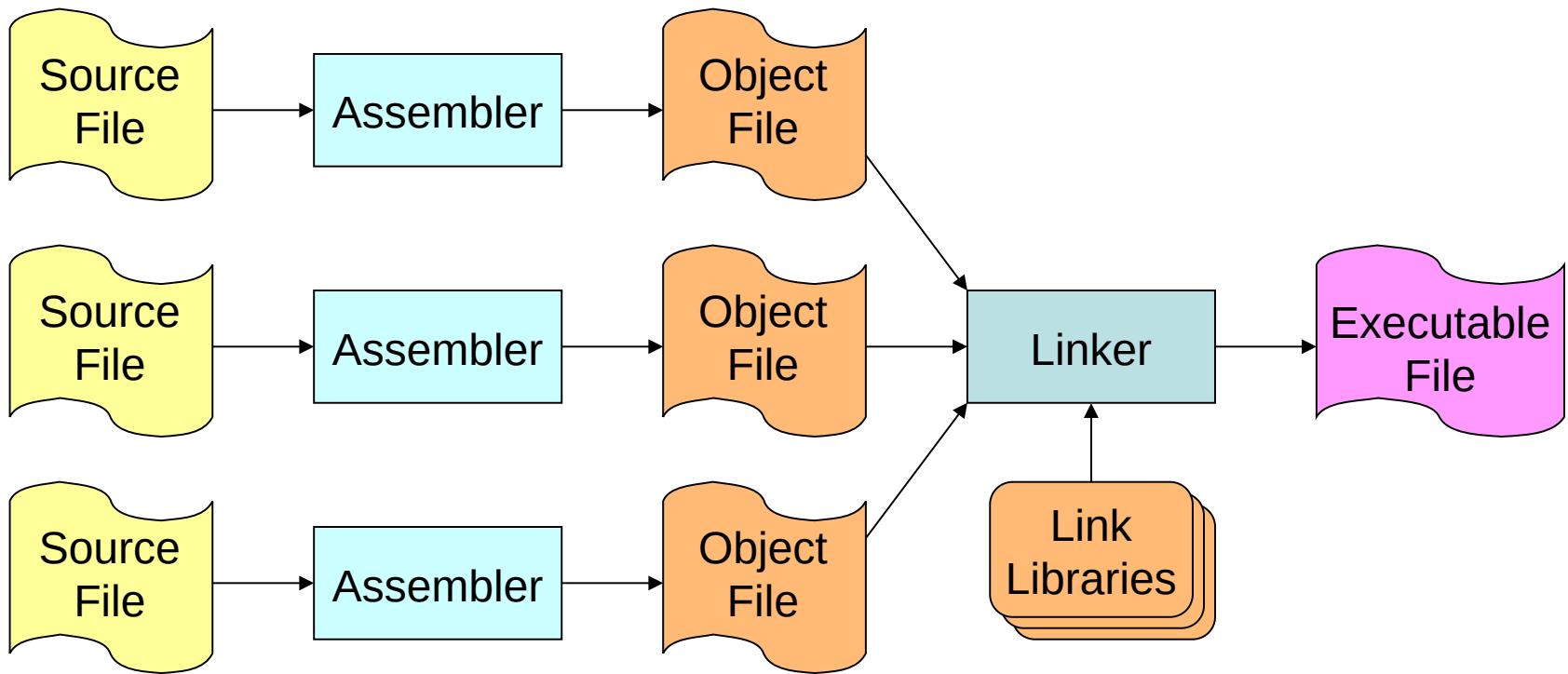
Assembler

- ❖ 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

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

Assemble and Link Process



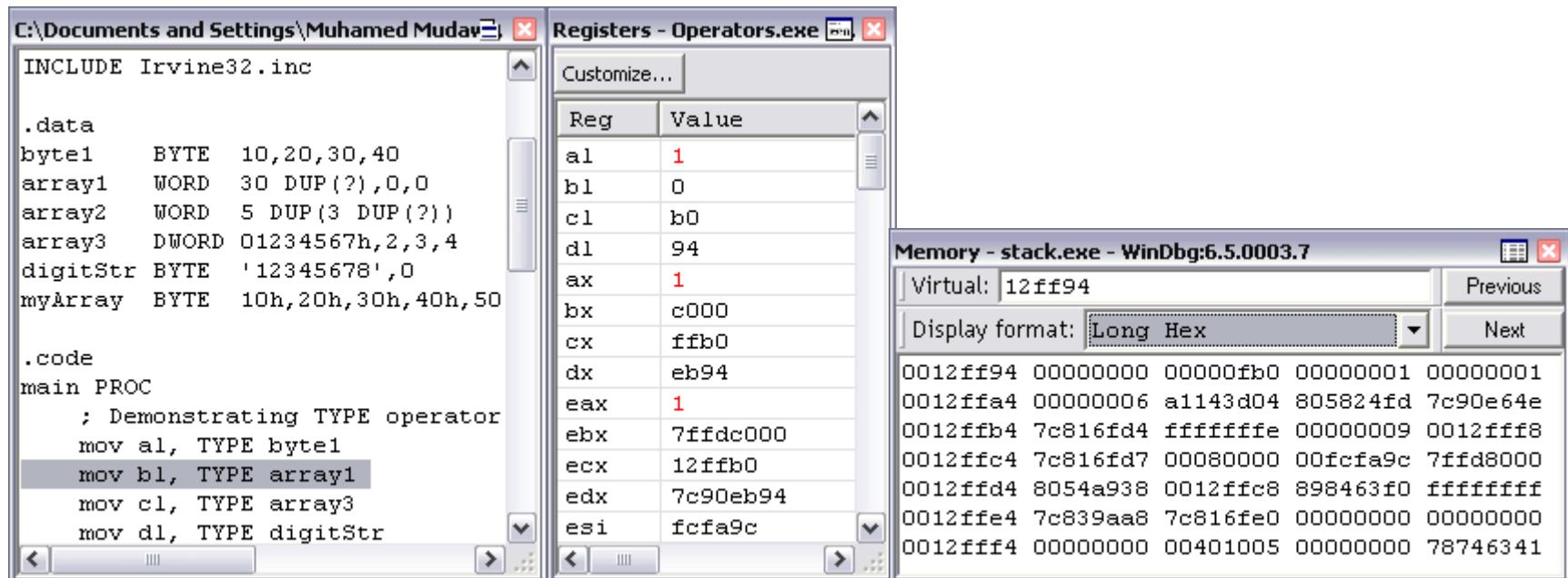
A project may consist of multiple source files

Assembler translates each source file separately into an object file

Linker links all object files together with link libraries

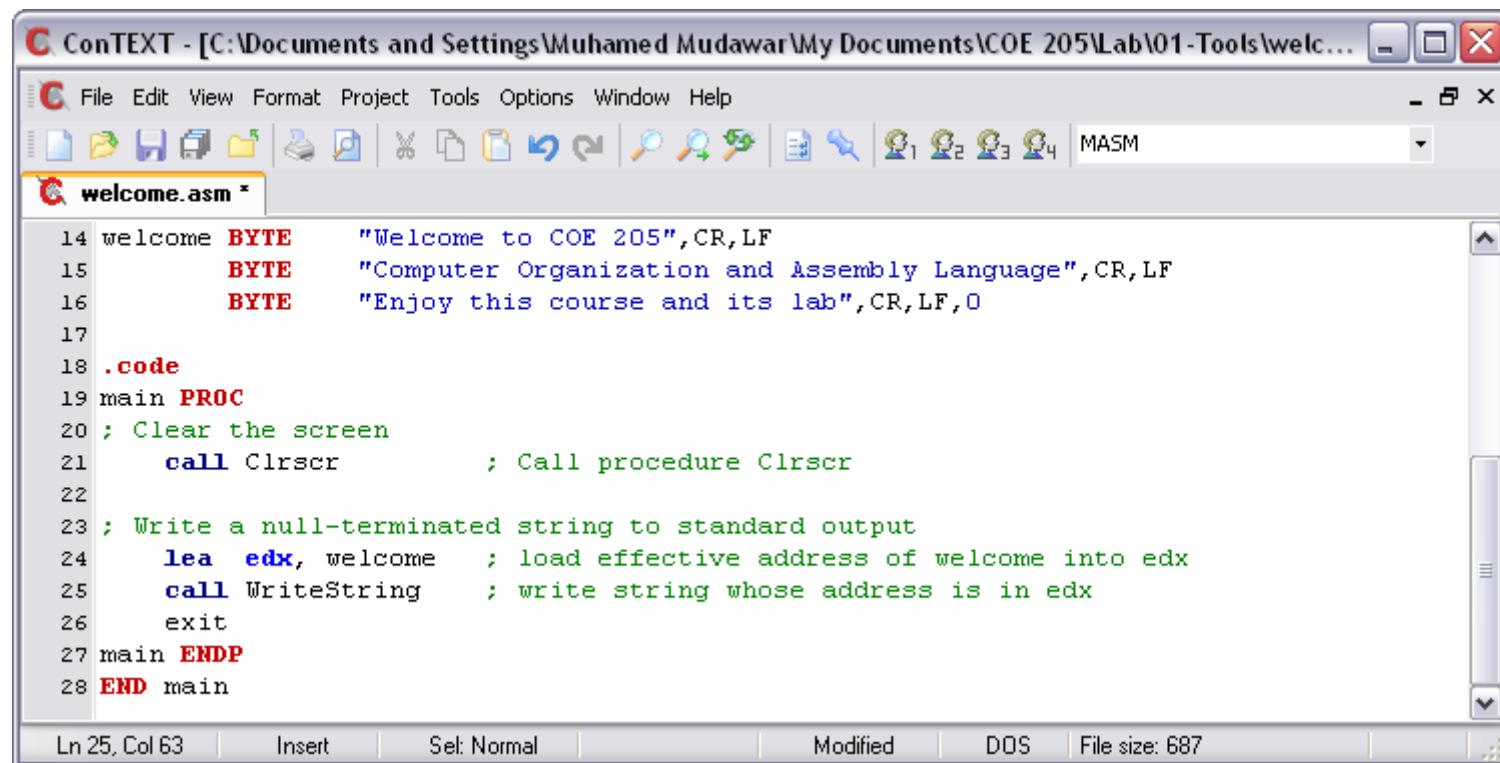
Debugger

- ❖ Allows you to trace the execution of a program
- ❖ Allows you to view code, memory, registers, etc.
- ❖ Example: **32-bit Windows debugger**



Editor

- ❖ Allows you to create assembly language source files
- ❖ Some editors provide syntax highlighting features and can be customized as a programming environment



The screenshot shows the ConTEXT editor interface with the file "welcome.asm" open. The code is as follows:

```
14 welcome BYTE    "Welcome to COE 205",CR,LF
15             BYTE    "Computer Organization and Assembly Language",CR,LF
16             BYTE    "Enjoy this course and its lab",CR,LF,0
17
18 .code
19 main PROC
20 ; Clear the screen
21     call Clrscr          ; Call procedure Clrscr
22
23 ; Write a null-terminated string to standard output
24     lea   edx, welcome    ; load effective address of welcome into edx
25     call WriteString      ; write string whose address is in edx
26     exit
27 main ENDP
28 END main
```

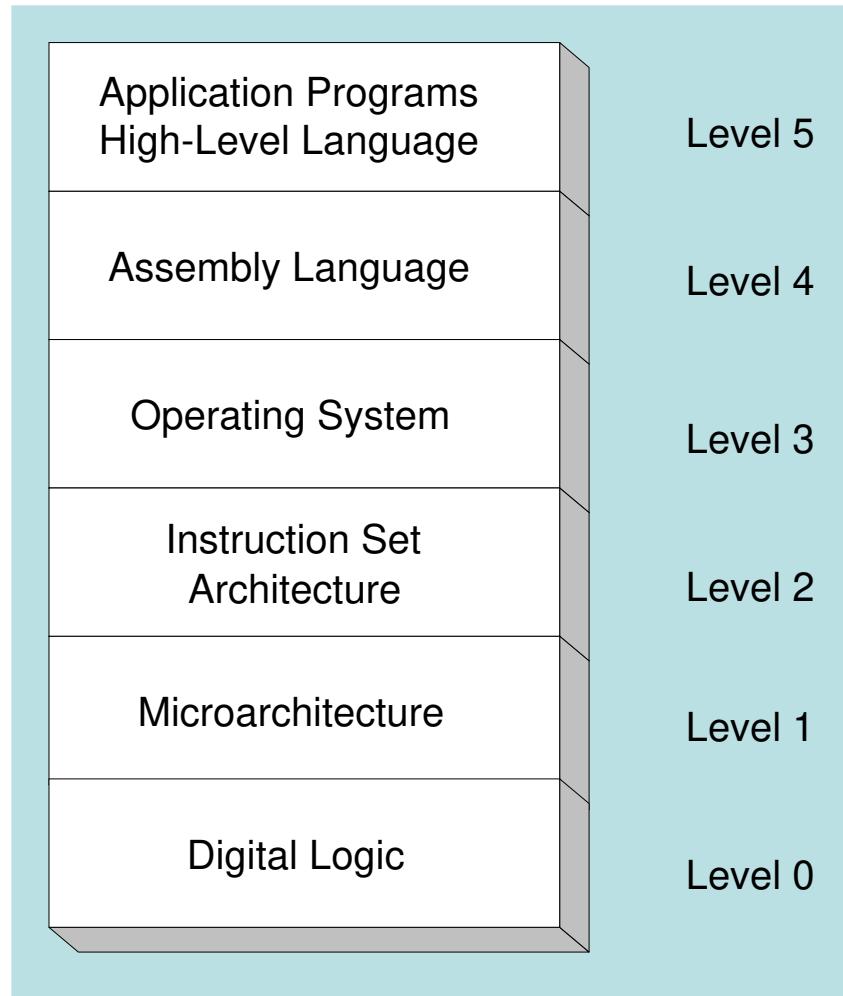
The status bar at the bottom shows "Ln 25, Col 63" and "File size: 687".

Next ...

- ❖ Welcome
- ❖ Assembly-, Machine-, and High-Level Languages
- ❖ Assembly Language Programming Tools
- ❖ Programmer's View of a Computer System
- ❖ Basic Computer Organization

Programmer's View of a Computer System

Increased level
of abstraction



Each level
hides the
details of the
level below it

Programmer's View – 2

❖ Application Programs (Level 5)

- ◊ Written in high-level programming languages
- ◊ Such as Java, C++, Pascal, Visual Basic . . .
- ◊ Programs compile into assembly language level (Level 4)

❖ Assembly Language (Level 4)

- ◊ Instruction mnemonics are used
- ◊ Have one-to-one correspondence to machine language
- ◊ Calls functions written at the operating system level (Level 3)
- ◊ Programs are translated into machine language (Level 2)

❖ Operating System (Level 3)

- ◊ Provides services to level 4 and 5 programs
- ◊ Translated to run at the machine instruction level (Level 2)

Programmer's View – 3

❖ Instruction Set Architecture (Level 2)

- ◊ Specifies how a processor functions
- ◊ Machine instructions, registers, and memory are exposed
- ◊ Machine language is executed by Level 1 (microarchitecture)

❖ Microarchitecture (Level 1)

- ◊ Controls the execution of machine instructions (Level 2)
- ◊ Implemented by digital logic (Level 0)

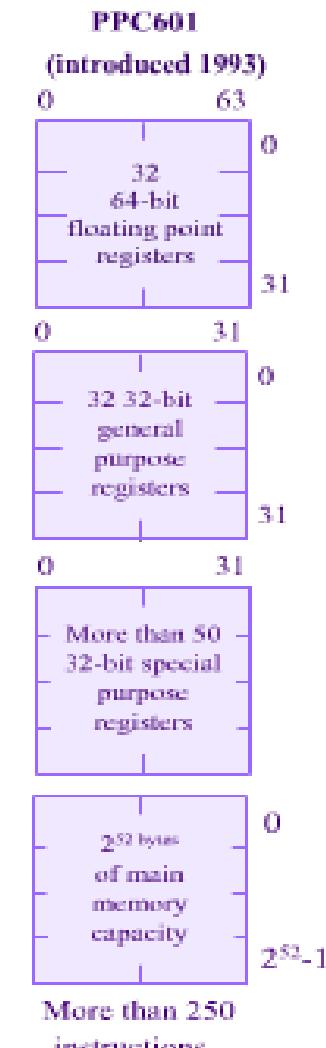
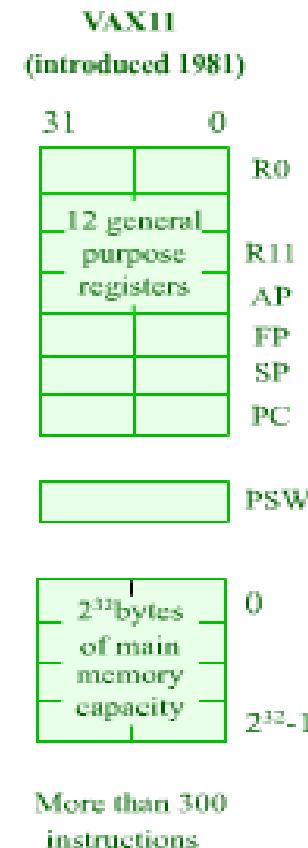
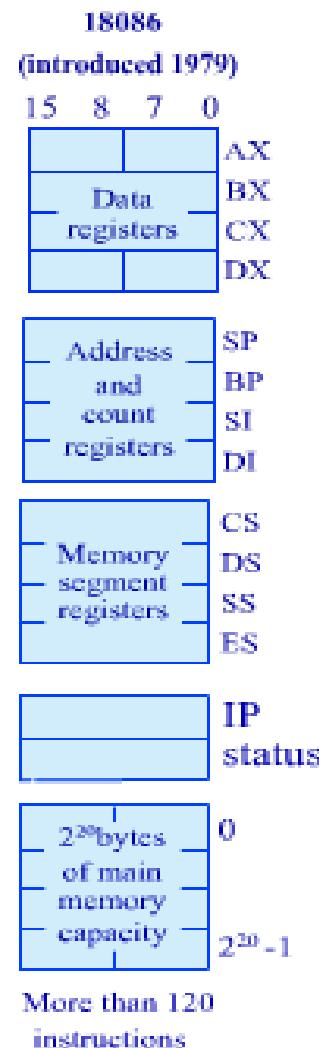
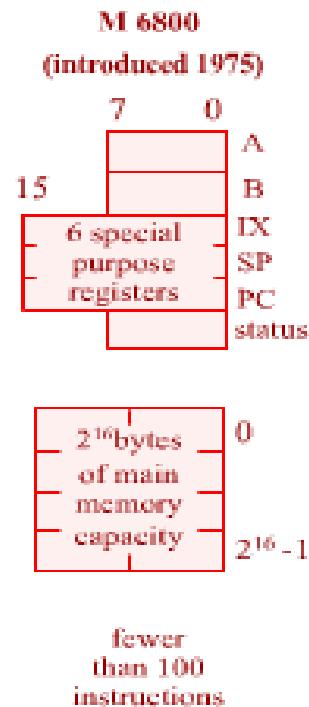
❖ Digital Logic (Level 0)

- ◊ Implements the microarchitecture
- ◊ Uses digital logic gates
- ◊ Logic gates are implemented using transistors

Instruction Set Architecture (ISA)

- ❖ Collection of assembly/machine instruction set of the machine
- ❖ Machine resources that can be managed with these instructions
 - ◊ Memory
 - ◊ Programmer-accessible registers.
- ❖ Provides a hardware/software interface

Instruction Set Architecture (ISA)

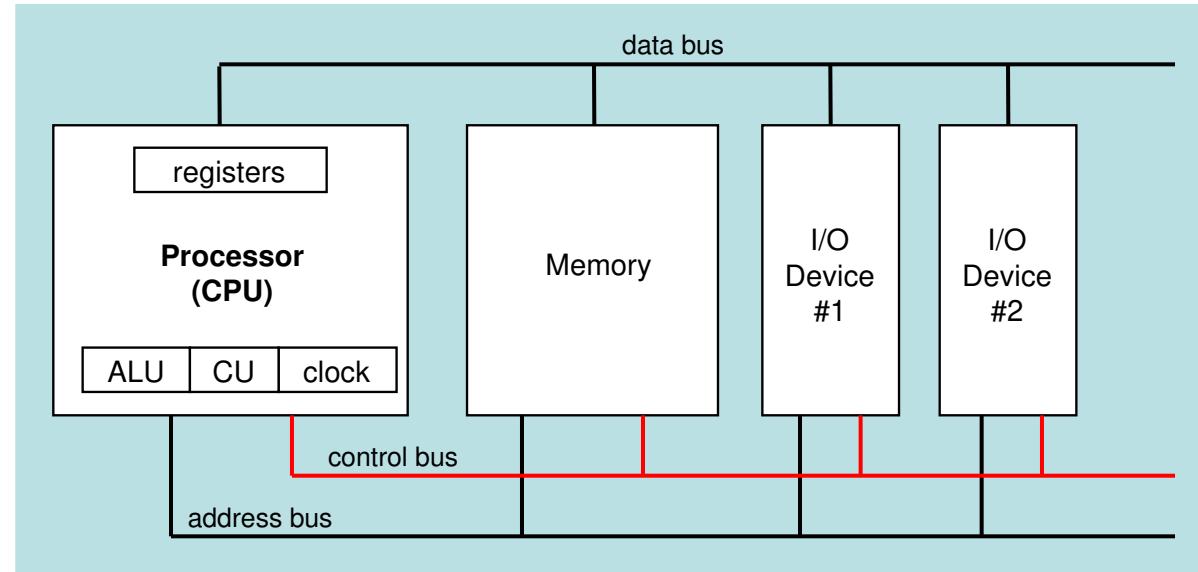


Next ...

- ❖ Welcome
- ❖ Assembly-, Machine-, and High-Level Languages
- ❖ Assembly Language Programming Tools
- ❖ Programmer's View of a Computer System
- ❖ Basic Computer Organization

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)

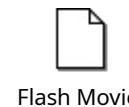
❖ Processor consists of

◊ Datapath

- ALU

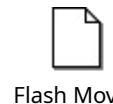
- Registers

◊ Control unit



Flash Movie

❖ ALU



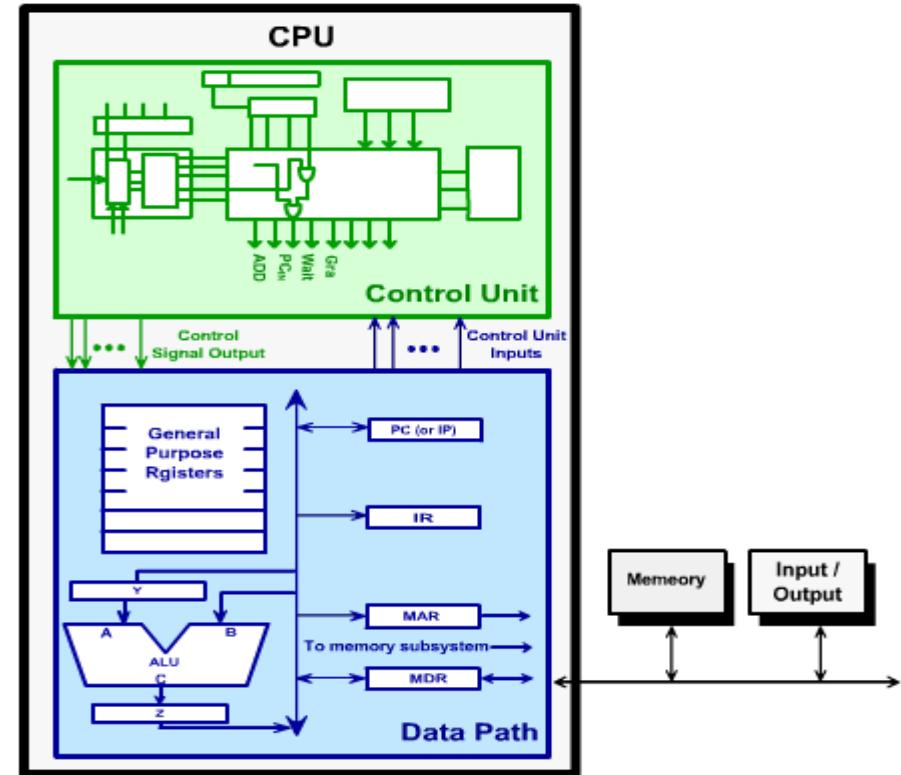
Flash Movie

◊ Performs arithmetic
and logic instructions

❖ Control unit (CU)

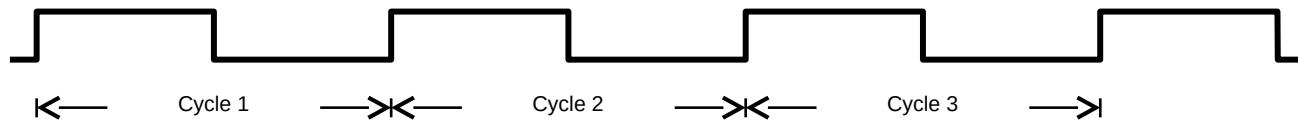
◊ Generates the control signals required to execute instructions

❖ Implementation varies from one processor to another



Clock

- ❖ Synchronizes Processor and Bus operations
- ❖ Clock cycle = Clock period = $1 / \text{Clock rate}$



- ❖ Clock rate = Clock frequency = Cycles per second
 - ◊ $1 \text{ Hz} = 1 \text{ cycle/sec}$ $1 \text{ KHz} = 10^3 \text{ cycles/sec}$
 - ◊ $1 \text{ MHz} = 10^6 \text{ cycles/sec}$ $1 \text{ GHz} = 10^9 \text{ cycles/sec}$
 - ◊ A 2 GHz clock has a cycle time = $1/(2 \times 10^9) = 0.5 \text{ nanosecond (ns)}$
- ❖ Clock cycles measure the execution of instructions

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

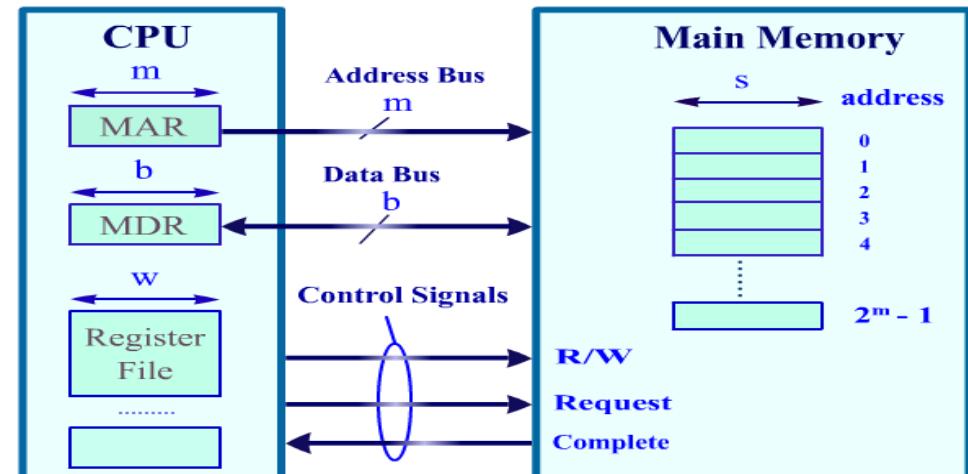
Address Space

Address (in decimal)	Address (in hex)
$2^{32}-1$	FFFFFFF
	FFFFFFE
	FFFFFFD
•	
2	00000002
1	00000001
0	00000000

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
 - ◊ Note that b is not necessarily equal to w or s . So data transfers might take more than a single cycle (if $w > b$) .
- ❖ Control Bus
 - ◊ Signals control transfer of data
 - ◊ Read request
 - ◊ Write request
 - ◊ Complete transfer



Memory Devices

❖ Random-Access Memory (RAM)

- ◊ Usually called the main memory
- ◊ It can be read and written to
- ◊ It does not store information permanently (Volatile , when it is powered off, the stored information are gone)
- ◊ Information stored in it can be accessed in any order at equal time periods (hence the name random access)
- ◊ Information is accessed by an address that specifies the exact location of the piece of information in the RAM.
- ◊ DRAM = Dynamic RAM
 - 1-Transistor cell + trench capacitor
 - Dense but slow, must be refreshed
 - Typical choice for main memory
- ◊ SRAM: Static RAM
 - 6-Transistor cell, faster but less dense than DRAM
 - Typical choice for cache memory



Memory Devices

❖ ROM (Read-Only-Memory)

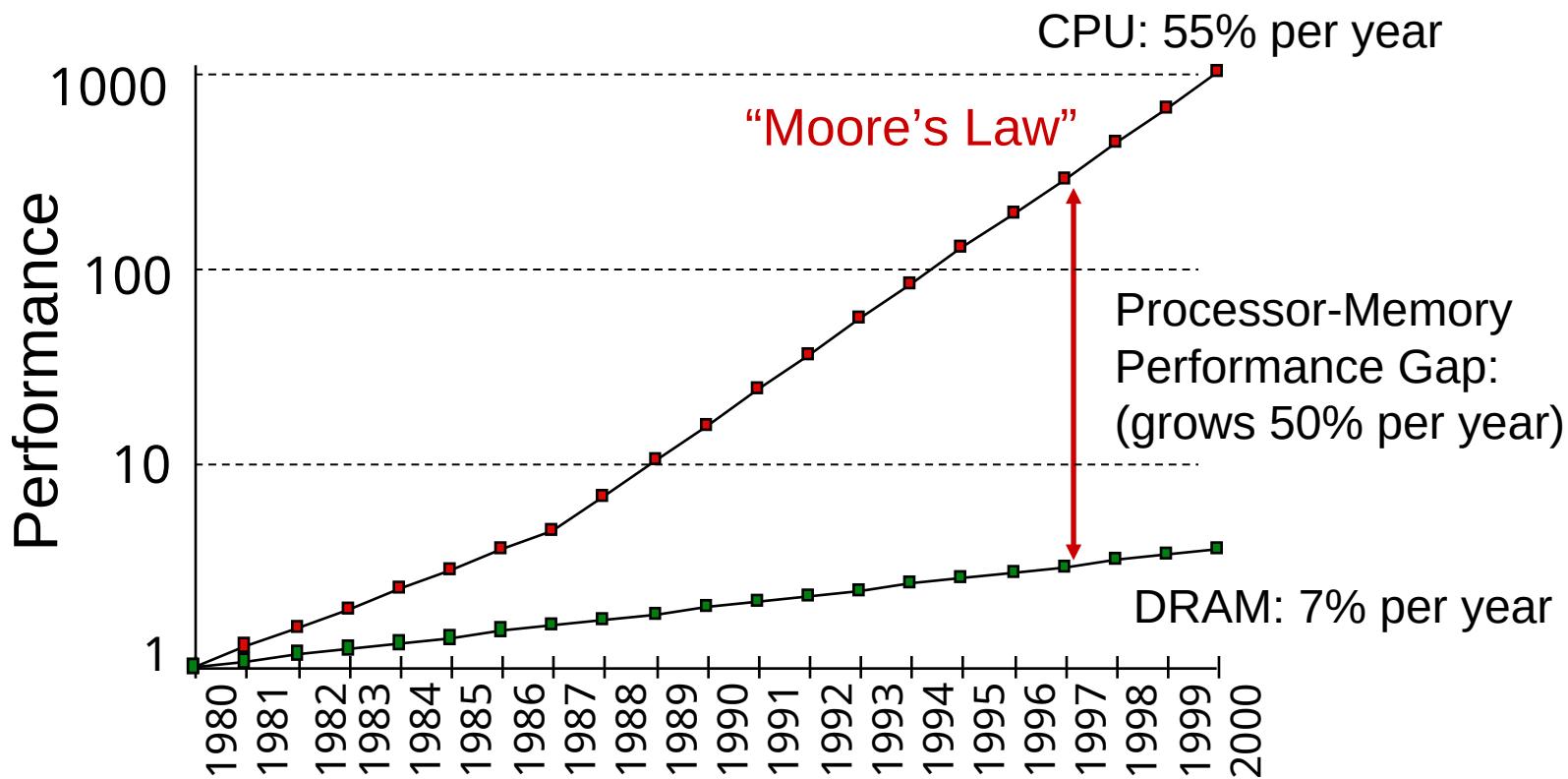
- ◊ A read-only-memory, non-volatile i.e. stores information permanently
- ◊ Has random access of stored information
- ◊ Used to store the information required to startup the computer
- ◊ Many types: ROM, EPROM, EEPROM, and FLASH
- ◊ FLASH memory can be erased electrically in blocks



❖ Cache

- ◊ A very fast type of RAM that is used to store information that is most frequently or recently used by the computer
- ◊ Recent computers have 2-levels of cache; the first level is faster but smaller in size (usually called internal cache), and the second level is slower but larger in size (external cache).

Processor-Memory Performance Gap



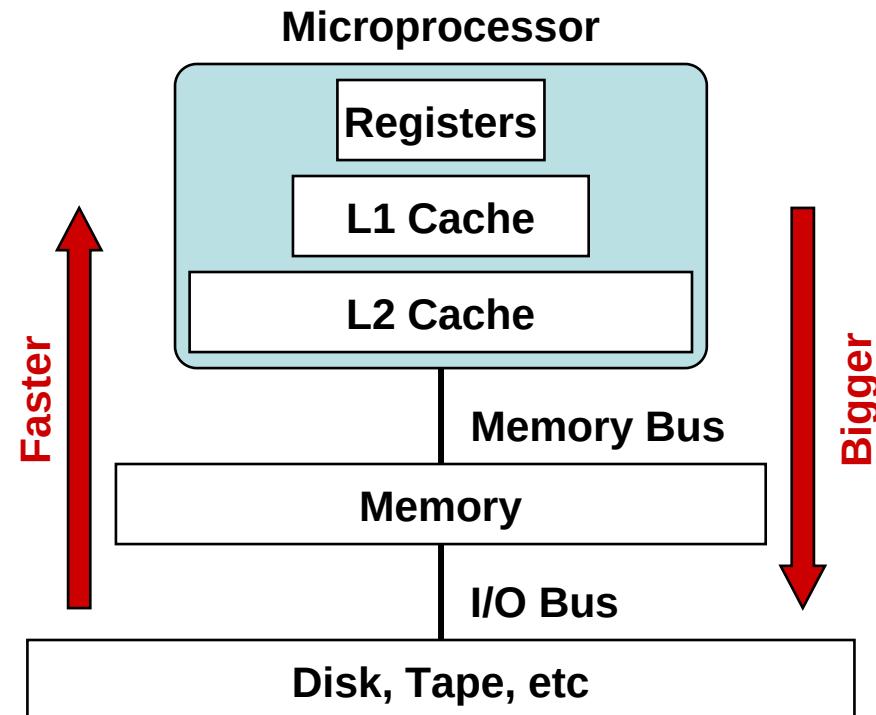
- ❖ 1980 – No cache in microprocessor
- ❖ 1995 – Two-level cache on microprocessor

The Need for a Memory Hierarchy

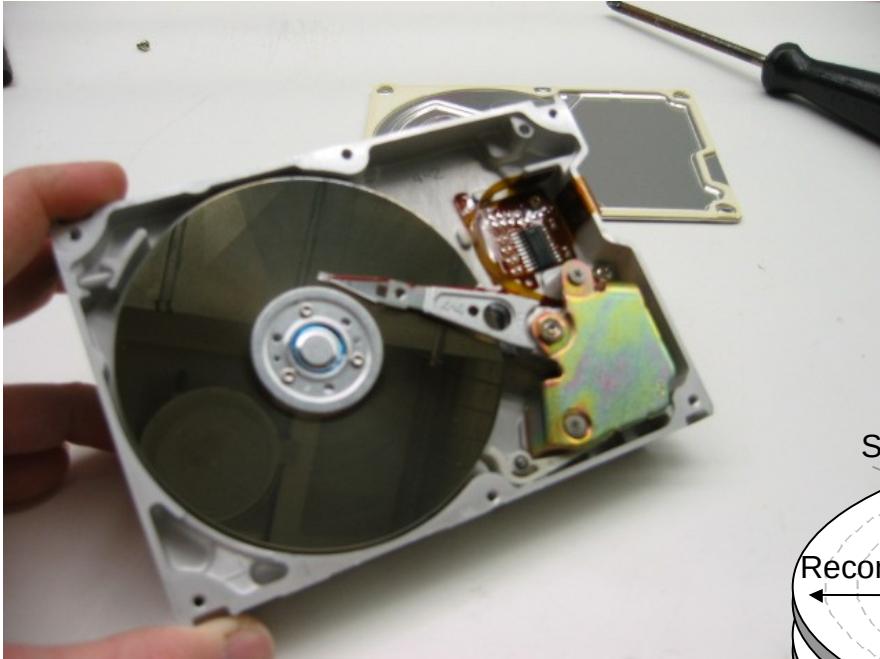
- ❖ Widening (expand) speed gap between CPU and main memory
 - ◊ Processor operation takes less than 1 ns
 - ◊ Main memory requires more than 50 ns to access
- ❖ Each instruction involves at least one memory access
 - ◊ One memory access to fetch the instruction
 - ◊ Additional memory accesses for instructions 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

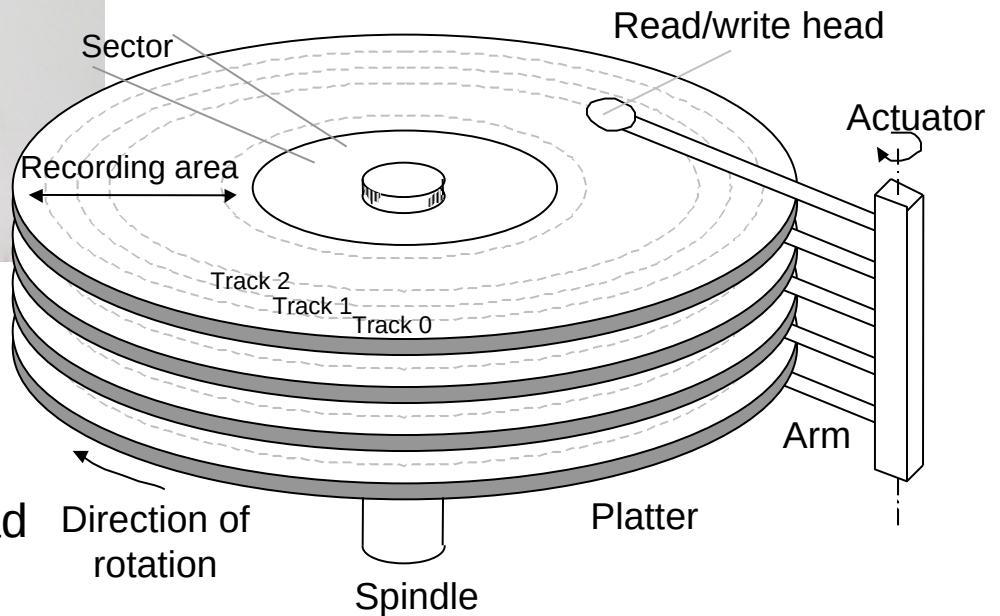
- ❖ Registers are at the top of the hierarchy
 - ◊ Typical size < 1 KB
 - ◊ Access time < 0.5 ns
- ❖ Level 1 Cache (8 – 64 KB)
 - ◊ Access time: 0.5 – 1 ns
- ❖ L2 Cache (512KB – 8MB)
 - ◊ Access time: 2 – 10 ns
- ❖ Main Memory (1 – 2 GB)
 - ◊ Access time: 50 – 70 ns
- ❖ Disk Storage (> 200 GB)
 - ◊ Access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



Magnetic Disk Storage



Disk Access Time =
Seek Time +
Rotation Latency +
Transfer Time



Seek Time: head movement to the desired track (milliseconds)

Rotation Latency: disk rotation until desired sector arrives under the head

Transfer Time: to transfer data

Example on Disk Access Time

- ❖ Given a magnetic disk with the following properties
 - ◊ Rotation speed = 7200 RPM (rotations per minute)
 - ◊ Average seek = 8 ms, Sector = 512 bytes, Track = 200 sectors
- ❖ Calculate
 - ◊ Time of one rotation (in milliseconds)
 - ◊ Average time to access a block of 32 consecutive sectors
- ❖ Answer
 - ◊ Rotations per second = $7200/60 = 120 \text{ RPS}$
 - ◊ Rotation time in milliseconds = $1000/120 = 8.33 \text{ ms}$
 - ◊ Average rotational latency = time of half rotation = 4.17 ms
 - ◊ Time to transfer 32 sectors = $(32/200) * 8.33 = 1.33$
 - ◊ Average access time = $8 + 4.17 + 1.33 = 13.5 \text{ ms}$

Beyond basics – L2

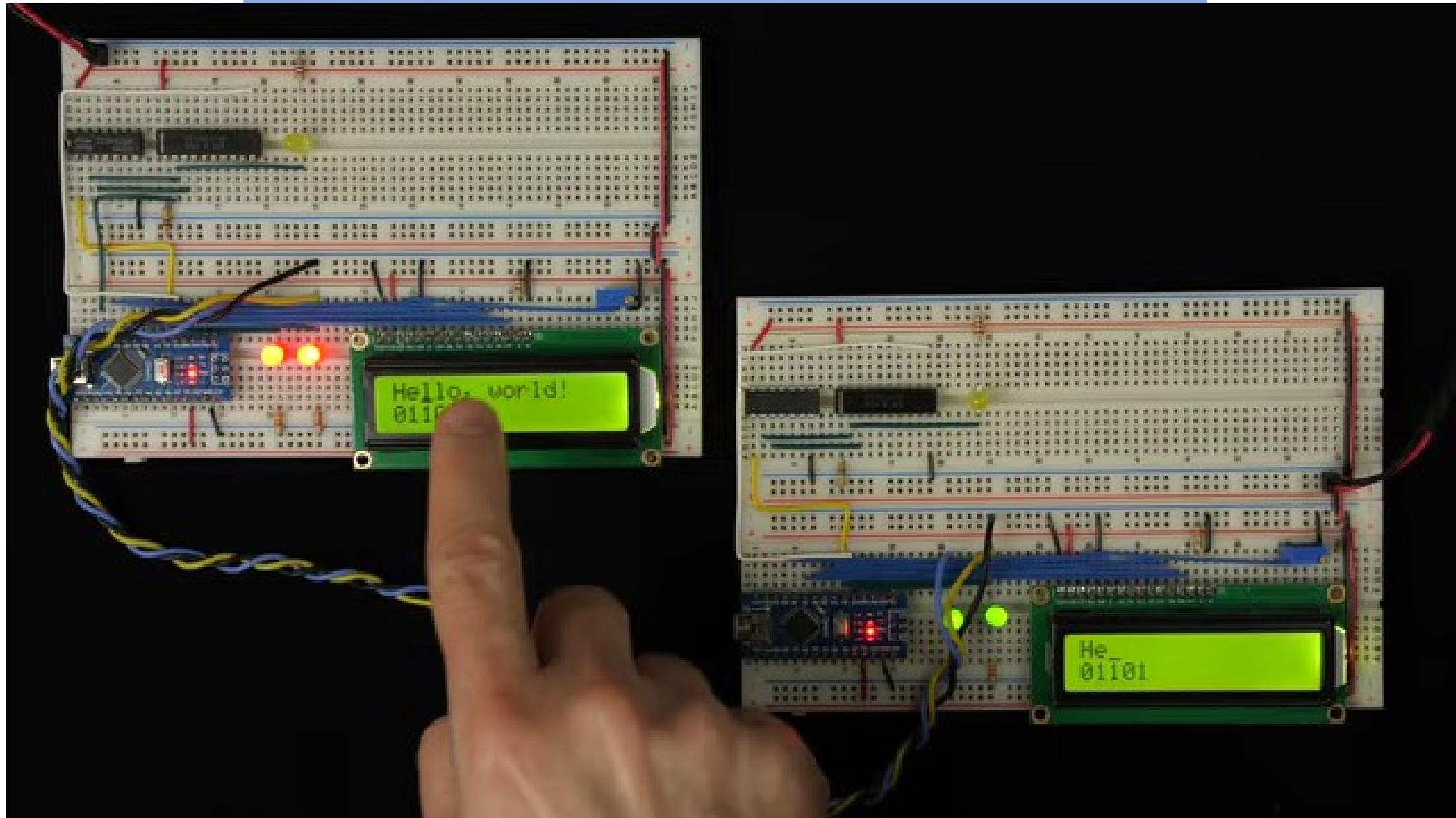
Assembly Language Programming

Dr Emmanuel Ahene

PARITY

- The fundamental unit of computer storage is a bit; it could be on (1) or off (0). A group of eight bits makes a byte. Seven bits are used for data and the last one is used for parity. According to the rule of parity, number of bits **that are on (1) in each byte** should always **be odd**

DUMMY DEMO



- So the parity bit is used to make the number of bits in a byte odd. If the parity is even, the system assumes that there had been a parity error (though rare) which might have caused due to hardware fault or electrical disturbance.
- The processor supports the following data sizes:
 - Word: a 2-byte data item
 - Doubleword: a 4-byte (32 bit) data item
 - Quadword: an 8-byte (64 bit) data item
 - Paragraph: a 16-byte (128 bit) area
 - Quadword: an 8-byte (64 bit) data item
 - Paragraph: a 16-byte (128 bit) area

The Binary Number System

- Every number system uses positional notation i.e., each position in which a digit is written has a different positional value. Each position is power of the base, which is 2 for binary number system, and these powers begin at 0 and increase by 1

Bit value	1	1	1	1	1	1	1	1	1
Position value as a power of base 2	128	64	32	16	8	4	2	1	1
Bit number	7	6	5	4	3	2	1	0	

- The value of a binary number is based on the presence of 1 bits and their positional value. So the value of the given binary number is: $1 + 2 + 4 + 8 + 16 + 32 + 64 + 128 = 255$, which is same as $2^8 - 1$.

The Hexadecimal Number System

- Hexadecimal number system uses **base 16**. The digits range from 0 to 15. By convention, the letters A through F is used to represent the hexadecimal digits corresponding to decimal values 10 through 15.
- Main use of hexadecimal numbers in computing is for abbreviating lengthy binary representations. Basically hexadecimal number system represents a binary data by dividing each byte in half and expressing the value of each half-byte. The following table provides the decimal, binary and hexadecimal equivalents:

Decimal number	Binary representation	Hexadecimal representation
0	0	0
1	1	1
2	10	2
3	11	3
4	100	4
5	101	5
6	110	6
7	111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

[Check ASCII CHART](#)
<http://www.aboutmyip.com/AboutMyXApp/AsciiChart.jsp>

- To convert a binary number to its hexadecimal equivalent, break it into groups of **4 consecutive groups** each, starting from **the right**, and write those groups over the corresponding digits of the hexadecimal number.
- Example: Binary number **1000 1100 1101 0001** is equivalent to hexadecimal - **8CD1**

- To convert a hexadecimal number to binary just write each hexadecimal digit into its 4-digit binary equivalent.
- Example: Hexadecimal number **FAD8** is equivalent to binary - **1111 1010 1101 1000**

BINARY ARITHMETIC

- The following table illustrates four simple rules for binary addition:

(i)	(ii)	(iii)	(iv)
			1
0	1	1	1
+0	+0	+1	+1
=0	=1	=10	=11

- Rules (iii) and (iv) shows a carry of a 1-bit into the next left position.

Decimal	Binary
60	00111100
+42	00101010
102	01100110

A negative binary value is expressed in **two's complement notation**. According to this rule, to convert a binary number to its negative value is to **reverse its bit values and add 1**

Example:

Number 53	00110101
Reverse the bits	11001010
Add 1	1
Number -53	11001011

To subtract one value from another, convert the number being subtracted to two's complement format and add the numbers.

Example: Subtract 42 from 53

Number 53	00110101
Number 42	00101010
Reverse the bits of 42	11010101
Add 1	1
Number -42	11010110
$53 - 42 = 11$	00001011

Overflow of the last 1 bit is lost.

Addressing Data in Memory

- The process through which the processor controls the execution of instructions is referred as the fetch-decode execute cycle, or the execution cycle. It consists of three continuous steps:
 - Fetching the instruction from memory
 - Decoding or identifying the instruction
 - Executing the instruction

- The processor may access one or more bytes of memory at a time. Let us consider a hexadecimal number **0725H**.
- This number will require two bytes of memory.
- The high-order byte or most significant byte is **07** and the low order byte is **25**.
- The processor stores data in reverse-byte sequence i.e., the low-order byte is stored in low memory address and high-order byte in high memory address.
- So if processor brings the value 0725H from register to memory, it will transfer 25 first to the lower memory address and 07 to the next memory address.

Register

07	25
----	----

Memory

25	07
----	----

x

$x+1$

x: memory
address

- When the processor gets the numeric data from memory to register, it again reverses the bytes.
- There are two kinds of memory addresses:
 - **An absolute address** - a direct reference of specific location.
 - **The segment address** (or offset) - starting address of a memory segment with the offset value
- an **offset** usually denotes the number of address locations added to a base address in order to get to a specific absolute address.

Assembly Basic Syntax

- An assembly program can be divided into three sections:
- The **data** section
- The **bss** section
- The **text** section

The data Section

The data section is used for declaring initialized data or constants. This data does not change at runtime.

You can declare various constant values, file names or buffer size etc. in this section.

The syntax for declaring data section is:

```
section .dat
```

The bss Section

- The bss section is used for declaring variables. The syntax for declaring bss section is:
- `section .bss`

The text section

- The text section is used for keeping the actual code.
- This section must begin with the declaration global main, which tells the kernel where the program execution begins.
- The syntax for declaring text section is:

```
section .text
    global main
main:
```

Comments

- Assembly language comment begins with a semicolon (;). It may contain any printable character including blank. It can appear on a line by itself, like:
- ; This program displays a message on screen
- or, on the same line along with an instruction, like:
- add eax ,ebx ; adds ebx to eax

Assembly Language Statements

- Assembly language programs consist of three types of statements:
- Executable instructions or instructions
- Assembler directives or pseudo-ops
- Macros

- The **executable instructions** or simply **instructions** tell the processor what to do.
- Each instruction consists of an **operation code** (opcode). Each executable instruction generates one machine language instruction.
- The **assembler directives** or **pseudo-ops** tell the assembler about the various aspects of the assembly process. These are non-executable and do not generate machine language instructions.
- **Macros** are basically a text substitution mechanism.

Syntax of Assembly Language Statements

- Assembly language statements are entered one statement per line. Each statement follows the following format:
- [label] mnemonic [operands] [;comment]
- The fields in the square brackets are optional. A basic instruction has two parts, the first one is the name of the instruction (or the mnemonic) which is to be executed, and the second are the operands or the parameters of the command.

- Following are some examples of typical assembly language statements:

```
INC COUNT          ; Increment the memory variable COUNT
MOV TOTAL, 48     ; Transfer the value 48 in the
                  ; memory variable TOTAL
ADD AH, BH        ; Add the content of the
                  ; BH register into the AH register
AND MASK1, 128    ; Perform AND operation on the
                  ; variable MASK1 and 128
ADD MARKS, 10     ; Add 10 to the variable MARKS
MOV AL, 10         ; Transfer the value 10 to the AL register
```

The Hello World Program in Assembly

- The following assembly language code displays the string 'Hello World' on the screen:

```
section .text
    global main          ;must be declared for linker (ld)
main:
    mov  edx,len        ;tells linker entry point
    mov  ecx,msg        ;message length
    mov  ebx,1           ;message to write
    mov  eax,4           ;file descriptor (stdout)
    mov  eax,4           ;system call number (sys_write)
    int  0x80            ;call kernel
```

```
mov eax,1          ;system call number (sys_exit)
int 0x80          ;call kernel

section .data
msg db 'Hello, world!', 0xa ;our dear string
len equ $ - msg        ;length of our dear string
```

When the above code is compiled and executed, it produces
following result.

```
Hello, world!
```

Compiling and Linking an Assembly Program in NASM

- Make sure you have set the path of **nasm** and **ld** binaries in your PATH environment variable. Now take the following steps for compiling and linking the above program:
- Type the above code using a text editor and save it as hello.asm. □ Make sure that you are in the same directory as where you saved hello.asm.

- To assemble the program, type nasm -f elf hello.asm
- If there is any error, you will be prompted about that at this stage. Otherwise an object file of your program named **hello.o** will be created.
- To link the object file and create an executable file named hello, type
- **ld -m elf_i386 -s -o hello hello.o**
- Execute the program by typing **./hello**
- If you have done everything correctly, it will display Hello, world! on the screen

Beyond basics – L3

Assembly Language Programming

Dr Emmanuel Ahene

Assembly Basic Syntax/structure

- An assembly program can be divided into three sections:
- The **data** section
- The **bss** section
- The **text** section

Memory Segments

- A segmented memory model divides the system memory into groups of independent segments, referenced by pointers located in the segment registers. Each segment is used to contain a specific type of data. One segment is used to contain instruction codes, another segment stores the data elements, and a third segment keeps the program stack.

Memory Segments

- In the light of the above discussion, we can specify various memory segments as:
- **Data segment** - it is represented by .data section and the .bss. The **.data section** is used to declare the memory region where data elements are stored for the program.
- This section cannot be expanded after the data elements are declared, and it remains static throughout the program.

Memory Segments

- The **.bss section** is also a static memory section that contains buffers for data to be declared later in the program. This buffer memory is zero-filled.
- **Code segment** - it is represented by .text section. This defines an area in memory that stores the instruction codes. This is also a fixed area.
 - Stack - this segment contains data values passed to functions and procedures within the program

ASSEMBLY REGISTERS

- Processor operations mostly involve processing data. This data can be stored in memory and accessed from thereon.
- However, reading data from and storing data into memory slows down the processor, as it involves complicated processes of sending the data request across the control bus, and into the memory storage unit and getting the data through the same channel.

ASSEMBLY REGISTERS

- To speed up the processor operations, the processor includes some internal memory storage locations, called **registers**.
- The registers stores data elements for processing without having to access the memory. A limited number of registers are built into the processor chip.

PROCESSOR REGISTERS

- In IA-32 architecture, registers hold 32 bits. This means each register can hold the values:

Unsigned : 0 to 4294967295

Signed: -2147483648 to 2147483647

- In the x86_64 architecture, registers hold 64 bits. This means each register can hold the values :

Unsigned: 0 to 18,446,744,073,709,551,616

Signed: -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

Registers

Registers (32/64Bits) are grouped into three categories:

General registers

Control registers

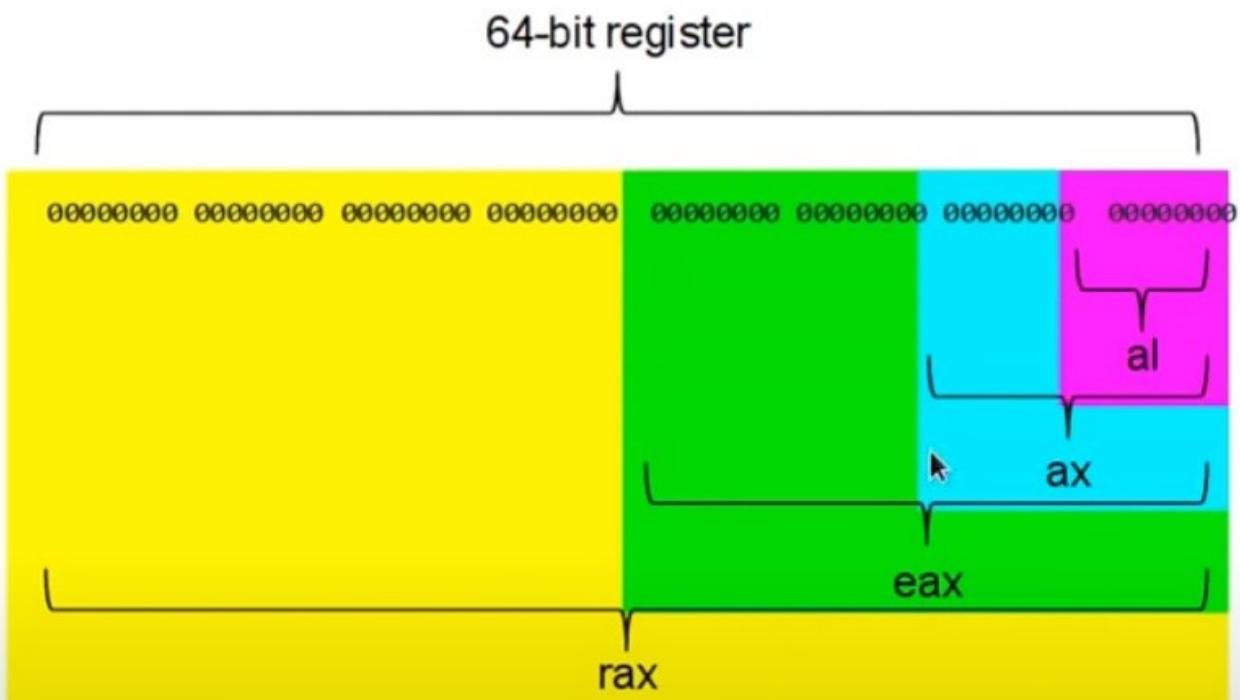
Segment register

PROCESSOR REGISTERS (32 & 64 bits)

- The general registers are further divided into the following groups:
 - Data registers
 - Pointer registers
 - Index registers

Register Table

Registers



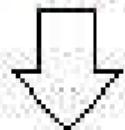
8-bit	16-bit	32-bit	64-bit
al	ax	eax	rax
bl	bx	ebx	rbx
cl	cx	ecx	rcx
dl	dx	edx	rdx
sil	si	esi	rsi
dil	di	edi	rdi
bpl	bp	ebp	rbp
spl	sp	esp	rsp
r8b	r8w	r8d	r8
r9b	r9w	r9d	r9
r10b	r10w	r10d	r10
r11b	r11w	r11d	r11
r12b	r12w	r12d	r12
r13b	r13w	r13d	r13
r14b	r14w	r14d	r14
r15b	r15w	r15d	r15

Data Registers (32 bits)

- Four 32-bit data registers are used for arithmetic, logical and other operations. These 32-bit registers can be used in three ways:
 1. As complete 32-bit data registers: EAX, EBX, ECX, EDX.
 2. Lower halves of the 32-bit registers can be used as four 16-bit data registers: AX, BX, CX and DX.
 3. Lower and higher halves of the above-mentioned four 16-bit registers can be used as eight 8-bit data registers: AH, AL, BH, BL, CH, CL, DH, and DL

Data Registers (32 bits)

32-bit registers



31

16 15

8 7

16-bit registers



0

AH

AL

BH

BL

CH

CL

DH

DL

AX Accumulator

BX Base

CX Counter

DX Data

EAX

EBX

ECX

EDX

Data Registers

- Some of these data registers has specific used in arithmetical operations.
- **AX** is the primary accumulator; it is used in input/output and most arithmetic instructions.
- For example, in multiplication operation, one operand is stored in EAX, or AX or AL register according to the size of the operand.
- **BX** is known as the base register as it could be used in indexed addressing.

Data Registers

- **CX** is known as the count register as the ECX, CX registers store the loop count in iterative operations. DX is known as the data register. It is also used in input/output operations.
- It is also used with AX register along with DX for multiply and divide operations involving large values

Hello World Source codes

```
section .text
    global main          ;must be declared for linker (ld)
main:
    mov  edx,len        ;tells linker entry point
    mov  ecx,msg        ;message length
    mov  ebx,1           ;message to write
    mov  eax,4           ;file descriptor (stdout)
    mov  eax,4           ;system call number (sys_write)
    int  0x80            ;call kernel

    mov  eax,1           ;system call number (sys_exit)
    int  0x80            ;call kernel

section .data
msg db 'Hello, world!', 0xa ;our dear string
len equ $ - msg             ;length of our dear string
```

```
section .data
    text db "Hello, World!",10

section .text
    global _start
_start:
    mov  rax, 1
    mov  rdi, 1
    mov  rsi, text
    mov  rdx, 14
    syscall

    mov  rax, 60
    mov  rdi, 0
    syscall
```

Any differences?

Hello World into details

```
section .data
    text db "Hello, World!",10

section .text
    global _start

_start:
    mov rax, 1
    mov rdi, 1
    mov rsi, text
    mov rdx, 14
    syscall

    mov rax, 60
    mov rdi, 0
    syscall
```



Defines bytes
"Hello, Word!\n" and
labels the memory
address "text".

And....

```
section .data
    text db "Hello, World!",10

section .text
    global _start
_start:
    mov rax, 1
    mov rdi, 1
    mov rsi, text
    mov rdx, 14
    syscall

    mov rax, 60
    mov rdi, 0
    syscall
```

The diagram illustrates the assembly code for a system call. A brace on the left side of the assembly code groups the four instructions: mov rax, 1; mov rdi, 1; mov rsi, text; and mov rdx, 14. This grouped code is connected by a line to a rectangular box containing the text "sys_write(1, text, 14)".

And....

```
section .data
    text db "Hello, World!",10

section .text
    global _start
_start:
    mov rax, 1
    mov rdi, 1
    mov rsi, text
    mov rdx, 14
    syscall

    mov rax, 60
    mov rdi, 0
    syscall
```

The diagram illustrates the assembly code for a C program. It shows the .data and .text sections. The .data section contains a string 'Hello, World!'. The .text section defines the entry point '_start' and contains assembly instructions for printing the string and exiting. A brace groups the final three instructions (mov rax, 60; mov rdi, 0; syscall) and points to a callout box labeled 'sys_exit(0)'.

Sections onward....

All x86_64 assembly files have three sections, the “.data” section, the “.bss” section, and the “.text” section.

The data section is where all data is defined before compilation.

The bss section is where data is allocated for future use.

The text section is where the actual code goes.

```
section .data
    text db "Hello, World!",10

section .text
global _start
_start:
    mov rax, 1
    mov rdi, 1
    mov rsi, text
    mov rdx, 14
    syscall

    mov rax, 60
    mov rdi, 0
    syscall
```

Labels Onward.....

A “label” is used to *label* a part of code.

Upon compilation, the compiler will calculate the location in which the label will sit in memory.

Any time the name of the label is used afterwards, that name is replaced by the location in memory by compiler.

```
section .data
    text db "Hello, World!",10

section .text
    global _start

_start:
    mov rax, 1
    mov rdi, 1
    mov rsi, text
    mov rdx, 14
    syscall

    mov rax, 60
    mov rdi, 0
    syscall
```

The “_start” label

The “_start” label is essential for all programs.

When your program is compiled and later executed, it is executed first at the location of “_start”.

If the linker cannot find “_start”, it will throw an error.

```
section .data
    text db "Hello, World!",10

section .text
    global _start
_start:
    mov rax, 1
    mov rdi, 1
    mov rsi, text
    mov rdx, 14
    syscall

    mov rax, 60
    mov rdi, 0
    syscall
```

Global

The word “global” is used when you want the linker to be able to know the address of some a label.

The object file generated will contain a link to every label declared “global”.

In this case, we have to declare “`_start`” as global since it is required for the code to be properly linked.

```
section .data
    text db "Hello, World!",10

section .text
    global _start
_start:
    mov rax, 1
    mov rdi, 1
    mov rsi, text
    mov rdx, 14
    syscall

    mov rax, 60
    mov rdi, 0
    syscall
```

Other important ones...

```
text db "Hello, World!",10
```



"db" stands for "define bytes".
It essentially means that we are going
to *define* some raw *bytes* of data to
insert into our code.

...System Call

A **system call**, or a **syscall**, is when a program requests a service from the **kernel**.

System calls will differ by operating system because different operating systems use different kernels.

All syscalls have an ID associated with them (a number).

Syscalls also take **arguments**, meaning, a list of inputs.

System call inputs by registers

Argument	Registers
ID	rax
1	rdi
2	rsi
3	rdx
4	r10
5	r8
6	r9

System Call List

syscall	ID	ARG1	ARG2	ARG3	ARG4	ARG5	ARG6
sys_read	0	#filedescriptor	\$buffer	#count			
sys_write	1	#filedescriptor	\$buffer	#count			
sys_open	2	\$filename	#flags	#mode			
sys_close	3	#filedescriptor					
...
pwritev2	328

Sys_write

Argument Type	Argument Description
File Descriptor	0 (Standard Input), 1 (Standard Output), 2 (Standard Error)
Buffer	Location of string to write
Count	Length of string

syscall	ID	ARG1	ARG2	ARG3	ARG4	ARG5	ARG6
sys_write	1	#filedescriptor	\$buffer	#count			

Sys_write

sys_write

Argument	Registers
ID	rax
1	rdi
2	rsi
3	rdx
4	r10
5	r8
6	r9

syscall	ID	ARG1	ARG2	ARG3	ARG4	ARG5	ARG6
sys_write	1	#filedescriptor	\$buffer	#count			

Sys_write

sys_write

Argument	Registers
ID	rax
1	rdi
2	rsi
3	rdx
4	r10
5	r8
6	r9

syscall	rax	rdi	rsi	rdx	r10	r8	r9
sys_write	1	#filedescriptor	\$buffer	#count			

Sys_write

Suppose we want to write “Hello, World!\n” to the screen...

Argument Type	Argument Description
File Descriptor	0 (Standard Input), 1 (Standard Output), 2 (Standard Error)
Buffer	Location of string to write
Count	Length of string

syscall	rax	rdi	rsi	rdx	r10	r8	r9
sys_write	1	#filedescriptor	\$buffer	#count			

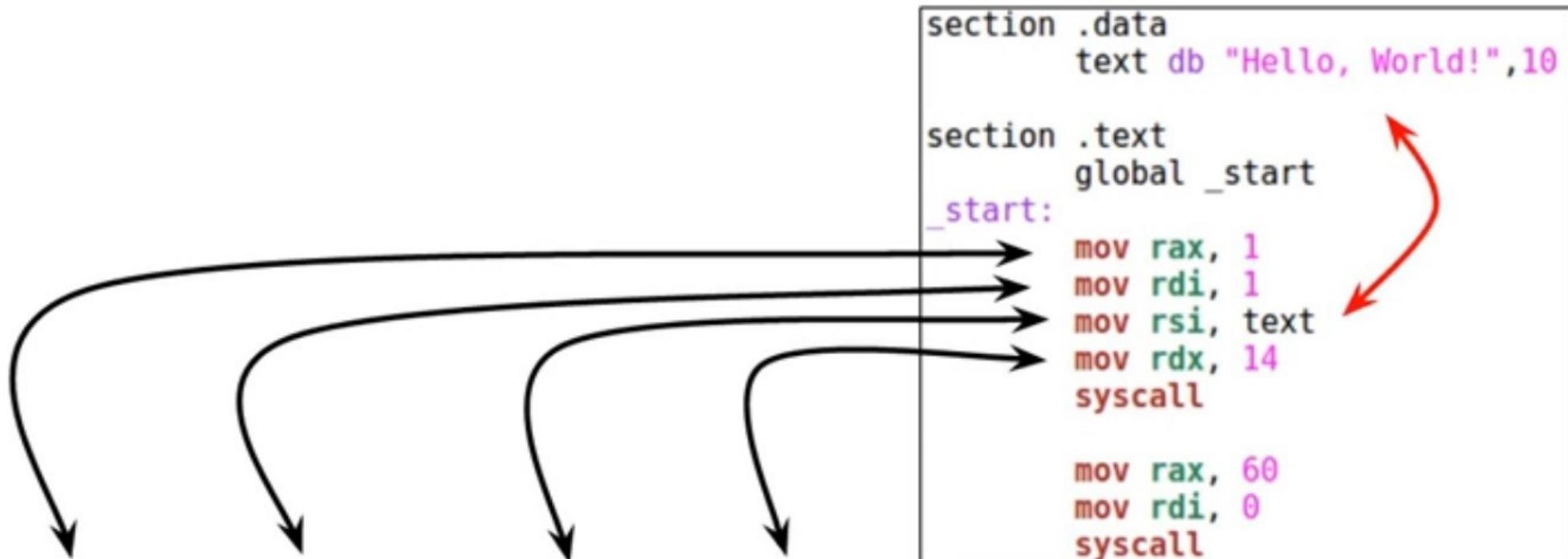
Sys_write

Suppose we want to write “Hello, World!\n” to the screen...

Argument Type	Argument Description
File Descriptor	0 (Standard Input), 1 (Standard Output), 2 (Standard Error)
Buffer	Location of string to write
Count	Length of string

syscall	rax	rdi	rsi	rdx	r10	r8	r9
sys_write	1	1	ADDR	14			

Sys_write



syscall	rax	rdi	rsi	rdx	r10	r8	r9
sys_write	1	1	ADDR	14			

Sys_exit

```
section .data
    text db "Hello, World!",10

section .text
    global _start
_start:
    mov rax, 1
    mov rdi, 1
    mov rsi, text
    mov rdx, 14
    syscall

    mov rax, 60
    mov rdi, 0
    syscall
```



syscall	rax	rdi	rsi	rdx	r10	r8	r9
sys_exit	60	0					

Pointer Registers

- The pointer registers are 32-bit EIP, ESP and EBP registers and corresponding 16-bit right portions IP, SP and BP. There are three categories of pointer registers:
- **Instruction Pointer (IP)** - the 16-bit IP register stores the offset address of the next instruction to be executed. IP in association with the CS register (as CS:IP) gives the complete address of the current instruction in the code segment.

Sample programs done:

- **Hello_world.asm** ;
display hello world on the screen
 - **Addition.asm / Subtraction.asm** ;
add/subtract two numbers and display the answer to screen
 - **Compare.asm** ;
compare 3 numbers and display the greatest on screen
- See WhatsApp group for sample codes
-
- **Assignment:**
Write the compare.asm code in 64 bits.
- To be submitted on 3/06/2021

Beyond basics – L4

Assembly Language Programming

Dr Emmanuel Ahene

Sample programs done:

- **Hello_world.asm** ;
displays hello world
 - **Addition.asm / Subtraction.asm** ;
add/subtract two numbers and display the answer to screen
 - **Compare.asm** ;
 - compares 3 numbers for the greatest
- See WhatsApp group for sample codes
-
- **Assignment:**
Write the compare.asm code in 64 bits.

To be submitted on 3/06/2021

Continuing Assembly Registers

Registers (32/64Bits) are grouped into three categories:

General registers

Control registers

Segment register

PROCESSOR REGISTERS (32 & 64 bits)

- The **general registers** are further divided into the following groups:
 - Data registers
 - Pointer registers
 - Index registers

Pointer Registers

- **Stack Pointer (SP)** - SP in association with the SS register (SS:SP) refers to be current position of data or address within the program stack.
- the 16-bit SP register provides the offset value within the program stack.
- SS – stack segment (procedures) CS: code segment (_text)
- DS – data segment (.bss, .data)

Pointer Registers

- **Base Pointer (BP)** - The address in SS register is combined with the offset in BP to get the location of the parameter.
 - BP can also be combined with DI and SI as base register for special addressing.
 - The 16-bit BP register mainly helps in referencing the parameter variables passed to a subroutine.
-
- **DI** - destination index **SI** – source index

Pointer Registers

Pointer registers

31 16 15 0

ESP

SP

Stack Pointer

EBP

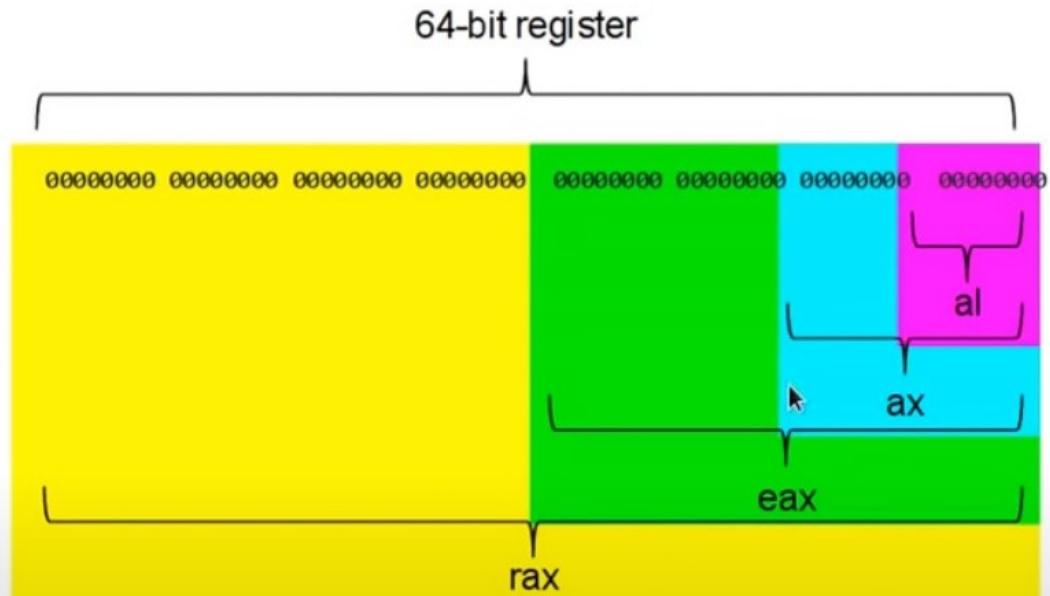
BP

Base Pointer



Have a look.....

Registers



8-bit	16-bit	32-bit	64-bit
al	ax	eax	rax
bl	bx	ebx	rbx
cl	cx	ecx	rcx
dl	dx	edx	rdx
sil	si	esi	rsi
dl	di	edi	rdi
bpl	bp	ebp	rbp
spl	sp	esp	rsp
r8b	r8w	r8d	r8
r9b	r9w	r9d	r9
r10b	r10w	r10d	r10
r11b	r11w	r11d	r11
r12b	r12w	r12d	r12
r13b	r13w	r13d	r13
r14b	r14w	r14d	r14
r15b	r15w	r15d	r15

and even more . . .

Pointers

Pointers are also registers that hold data.

They “point to” data, meaning, they hold its memory address.

Pointer Name	Meaning	Description
rip (eip, ip)	Index pointer	Points to next address to be executed in the control flow.
rsp (esp, sp)	Stack pointer	Points to the top address of the stack.
rbp (ebp, bp)	Stack base pointer	Points to the bottom of the stack.
...

Registers as Pointers

The default registers can be treated as pointers.

To treat a register as a pointer, surround the register name with square brackets, such as, “rax” becomes “[rax]”.

```
mov rax, rbx
```

Loads the value in the *rbx* register into the *rax* register.

```
mov rax, [rbx]
```

Loads the value the *rbx* register is *pointing to* into the *rax* register.

Index Registers

- The 32-bit index registers ESI and EDI and their 16-bit rightmost portions SI and DI are used for indexed addressing and sometimes used in addition and subtraction.
- There are two sets of index pointers:
 - **Source Index (SI)** - it is used as source index for string operations
 - **Destination Index (DI)** - it is used as destination index for string operations.

Index Registers

Index registers

31 16 15 0



Control Registers

- The 32-bit/64 bit instruction pointer register and 32-bit/64 bit flags register combined are considered as the **control registers**.
- Many instructions involve comparisons and mathematical calculations and change the status of the flags and some other conditional instructions test the value of these status flags to take the control flow to other location.

Flags

Flags, like registers, hold data.

Flags only hold 1 bit each. They are either *true* or *false*.

Individual flags are part of a larger register.

Flag Symbol	Description
CF	Carry
PF	Parity
ZF	Zero
SF	Sign
OF	Overflow
AF	Adjust
IF	Interrupt Enabled

Control Registers

The common flag bits are:

- **Overflow Flag (OF):** indicates the overflow of a high-order bit (leftmost bit) of data after a signed arithmetic operation.
- **Direction Flag (DF):** determines left or right direction for moving or comparing string data. When the DF value is 0, the string operation takes left-to-right direction and when the value is set to 1, the string operation takes right-to-left direction

Control Registers

- **Interrupt Flag (IF):** determines whether the external interrupts like, keyboard entry etc. are to be ignored or processed. It disables the external interrupt when the value is 0 and enables interrupts when set to 1.
- **Trap Flag (TF):** allows setting the operation of the processor in single-step mode. The DEBUG program we used sets the trap flag, so we could step through the execution one instruction at a time.

Control Registers

- **Sign Flag (SF):** shows the sign of the result of an arithmetic operation. This flag is set according to the sign of a data item following the arithmetic operation. The sign is indicated by the high-order of leftmost bit. A positive result clears the value of SF to 0 and negative result sets it to 1.
- **Zero Flag (ZF):** indicates the result of an arithmetic or comparison operation. A nonzero result clears the zero flag to 0, and a zero result sets it to 1

Control Registers

- **Auxiliary Carry Flag (AF):** contains the carry from bit 3 to bit 4 following an arithmetic operation; used for specialized arithmetic. The AF is set when a 1-byte arithmetic operation causes a carry from bit 3 into bit 4.
- **Parity Flag (PF):** indicates the total number of 1-bits in the result obtained from an arithmetic operation. An even number of 1-bits clears the parity flag to 0 and an odd number of 1-bits sets the parity flag to 1.

Control Registers

- **Carry Flag (CF):** contains the carry of 0 or 1 from a high-order bit (leftmost) after an arithmetic operation. It also stores the contents of last bit of a shift or rotate operation.
- The following table indicates the position of flag bits in the 16-bit Flags register:

Flag:					O	D	I	T	S	Z		A		P		C
Bit no:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Control Flow

All code runs from top to bottom by default. The direction a program flows is called the *control flow*.

The *rip* register holds the address of the next instruction to be executed. After each instruction, it is incremented by 1, making the control flow naturally flow from top to bottom.

```
section .data
    text db "Hello, world!", 10

section .text
    global _start

_start:
    mov rax, 1          ; rip = x
    mov rdi, 1          ; rip = x+1
    mov rsi, text       ; rip = x+2
    mov rdx, 14         ; rip = x+3
    syscall            ; rip = x+4

    mov rax, 60          ; rip = x+5
    mov rdi, 0           ; rip = x+6
    syscall            ; rip = x+7
```



Jumps

Jumps can be used to *jump* to different parts of code based on *labels*. They are used to divert program flow.

The general format of the jump is:

jmp label



Loads the value “label” into the rip register

Example:

```
_start:  
    jmp _start
```



Comparisons

Comparisons allow programs to be able to take different paths based on certain conditions.

Comparisons are done on *registers*.

The general format of a comparison is...

cmp register, register/value

Example:

```
cmp rax, 23  
cmp rax, rbx
```

Conditional Jumps

After a comparison is made, a *conditional jump* can be made.

Conditional jumps are based on the status of the flags.

Conditional jumps in code are written just like unconditional jumps, however “jmp” is replaced by the symbol for the conditional jump.

Jump symbol (signed)	Jump symbol (unsigned)	Results of cmp a,b
je	-	a = b
jne	-	a ≠ b
jg	ja	a > b
jge	jae	a ≥ b
jl	jb	a < b
jle	jbe	a ≤ b
jz	-	a = 0
jnz	-	a ≠ 0
jo	-	Overflow occurred
jno	-	Overflow did not occur
js	-	Jump if signed
jns	-	Jump if not signed

Conditional Jump Examples

This code will jump to the address of label “_doThis” *if and only if* the value in the *rax* register equals 23.

```
cmp rax, 23  
je _doThis
```

This code will jump to the address of label “_doThis” *if and only if* the value in the *rax* register is greater than the value in the *rbx* register.

```
cmp rax, rbx  
jg _doThis
```

Compare.asm

```
|section .text
    global _start

_start:
    mov ecx, [num1]
    cmp ecx, [num2]
    jg check_third
    mov ecx, [num2]

check_third:
    cmp ecx, [num3]
    jg _exit
    mov ecx, [num3]

_exit:
    mov [large], ecx
    mov ecx, msg
    mov edx, len
    mov ebx, 1
    mov eax, 4
    int 0x80
```

Compare.asm

```
        mov ecx, large
        mov edx, 2
        mov ebx, 1
        mov eax, 4
        int 0x80

        mov eax, 1
        int 0x80

section .data
    msg db "The largest is: ", 0xA, 0xD
    len equ $-msg
    num1 dd '47'
    num2 dd '22'
    num3 dd '31'

segment .bss
    large resb 2
```

Calls

Calls and jumps are essentially the same.

However, when “call” is used, the original position the call was made can be returned to using “ret”.

In this modification of the “Hello, World!” code, the part of code that prints “Hello, World!” was moved into its own section, and that section was *called*.

This is called a *subroutine*.

```
section .data
    text db "Hello, world!",10

section .text
    global _start

_start:
    call _printHello

    mov rax, 60
    mov rdi, 0
    syscall

_printHello:
    mov rax, 1
    mov rdi, 1
    mov rsi, text
    mov rdx, 14
    syscall
    ret
```

Segment Registers

- Segments are specific areas defined in a program for containing data, code and stack. There are three main segments:
- **Code Segment:** it contains all the instructions to be executed. A 16 - bit Code Segment register or CS register stores the starting address of the code segment.
- *Yeah ! Same thing you know about segments....*

Segment Registers

- **Data Segment:** it contains data, constants and work areas. A 16 - bit Data Segment register or DS register stores the starting address of the data segment.
- **Stack Segment:** it contains data and return addresses of procedures or subroutines. It is implemented as a 'stack' data structure. The Stack Segment register or SS register stores the starting address of the stack

Segment Registers

- Apart from the DS, CS and SS registers, there are other extra segment registers - ES (extra segment), FS and GS, which provides additional segments for storing data.

Segment Registers

- In assembly programming, a program needs to access the memory locations.
- All memory locations within a segment are relative to the starting address of the segment.
- The segment registers stores the starting addresses of a segment.
- To get the exact location of data or instruction within a segment, an offset value (or displacement) is required.

Segment Registers

- To reference any memory location in a segment, the processor combines the segment address in the segment register with the offset value of the location.

9stars.asm

```
section .text
    global main ;must be declared for linker (gcc)
main:    ;tell linker entry point
        mov     edx,len          ;message length
        mov     ecx,msg          ;message to write
        mov     ebx,1              ;file descriptor (stdout)
        mov     eax,4              ;system call number (sys_write)
        int     0x80              ;call kernel

        mov     edx,9              ;message length
        mov     ecx,s2             ;message to write
        mov     ebx,1              ;file descriptor (stdout)
        mov     eax,4              ;system call number (sys_write)
        int     0x80              ;call kernel
        mov     eax,1              ;system call number (sys_exit)
        int     0x80              ;call kernel

section .data
msg db 'Displaying 9 stars',0xa ;a message
len equ $ - msg                 ;length of message
s2 times 9 db '*'               
```

When the above code is compiled and executed, it produces following result:

```
Displaying 9 stars
*****
```

Assembly System Calls

32/64 bits, Done already. Refer to beyond basics L3

Details for only 32bits are in subsequent slides

Assembly System Calls

- System calls are APIs for the interface between user space and kernel space. We have already used the system calls `sys_write` and `sys_exit` for writing into the screen and exiting from the program respectively.

Linux System Calls

- You can make use of Linux system calls in your assembly programs. You need to take the following steps for using Linux system calls in your program:
 - Put the system call number in the EAX register.
 - Store the arguments to the system call in the registers EBX, ECX, etc.
 - Call the relevant interrupt (80h)
 - The result is usually returned in the EAX register

Linux System Calls

- There are six registers that stores the arguments of the system call used. These are the EBX, ECX, EDX, ESI, EDI, and EBP. These registers take the consecutive arguments, starting with the EBX register. If there are more than six arguments then the memory location of the first argument is stored in the EBX register.

Linux System Calls

The following code snippet shows the use of the system call `sys_exit`:

```
mov eax,1    ; system call number (sys_exit)
int 0x80      ; call kernel
```

Linux System Calls

The following code snippet shows the use of the system call `sys_write`:

```
mov edx,4 ; message length
mov ecx,msg ; message to write
mov ebx,1 ; file descriptor (stdout)
mov eax,4 ; system call number (sys_write)
int 0x80 ; call kernel
```

Linux System Calls

All the syscalls are listed in `/usr/include/asm/unistd.h`, together with their numbers (the value to put in EAX before you call int 80h).

Linux System Calls

The following table shows some of the system calls used in this tutorial:

%eax	Name	%ebx	%ecx	%edx	%esx	%edi
1	sys_exit	int	-	-	-	-
2	sys_fork	struct pt_regs	-	-	-	-
3	sys_read	unsigned int	char *	size_t	-	-
4	sys_write	unsigned int	const char *	size_t	-	-
5	sys_open	const char *	int	int	-	-
6	sys_close	unsigned int	-	-	-	-

Input.asm

```
section .data ;Data segment
    userMsg db 'Please enter a number: ' ;Ask the user to enter a number
    lenUserMsg equ $-userMsg ;The length of the message
    dispMsg db 'You have entered: '
    lenDispMsg equ $-dispMsg

section .bss ;Uninitialized data
    num resb 5
section .text ;Code Segment
    global main
    main:
        ;User prompt
        mov eax, 4
        mov ebx, 1
        mov ecx, userMsg
        mov edx, lenUserMsg
        int 80h

        ;Read and store the user input
        mov eax, 3
        mov ebx, 2
        mov ecx, num
        mov edx, 5 ;5 bytes (numeric, 1 for sign) of that information
        int 80h
```

```
;Output the message 'The entered number is: '
mov eax, 4
mov ebx, 1
mov ecx, dispMsg
mov edx, lenDispMsg
int 80h

;Output the number entered
mov eax, 4
mov ebx, 1
mov ecx, num
mov edx, 5
int 80h

; Exit code
mov eax, 1
mov ebx, 0
int 80h
```

When the above code is compiled and executed, it produces following result

```
Please enter a number:  
1234  
You have entered:1234
```

Beyond basics – L5

Assembly Language Programming

Dr Emmanuel Ahene

Course Outlook...

- Done so far --- Basic Concepts, Beyond basics L2, L3, L4
- Looking up to L5, L6, L7 & L8
 - Addressing modes
 - Assembly Variables & Constants
 - Arithmetic instructions
 - Logical Instructions
 - Assembly Conditions
 - Procedures
 - Recursions
 - Macros
 - File management
 - Memory Management

ADDRESSING MODES

Addressing Modes

Most assembly language instructions require **operands** to be processed.

An **operand address** provides the **location** where the data to be processed is **stored**.

Some instructions do not require an operand, whereas some other instructions may require one, two or three operands.

Addressing Modes

When an instruction requires two operands, the first operand is generally the destination, which contains data in a register or memory location and the second operand is the source.

Source contains either the data to be delivered (immediate addressing) or the address (in register or memory) of the data.

Generally the source data remains unaltered after the operation.

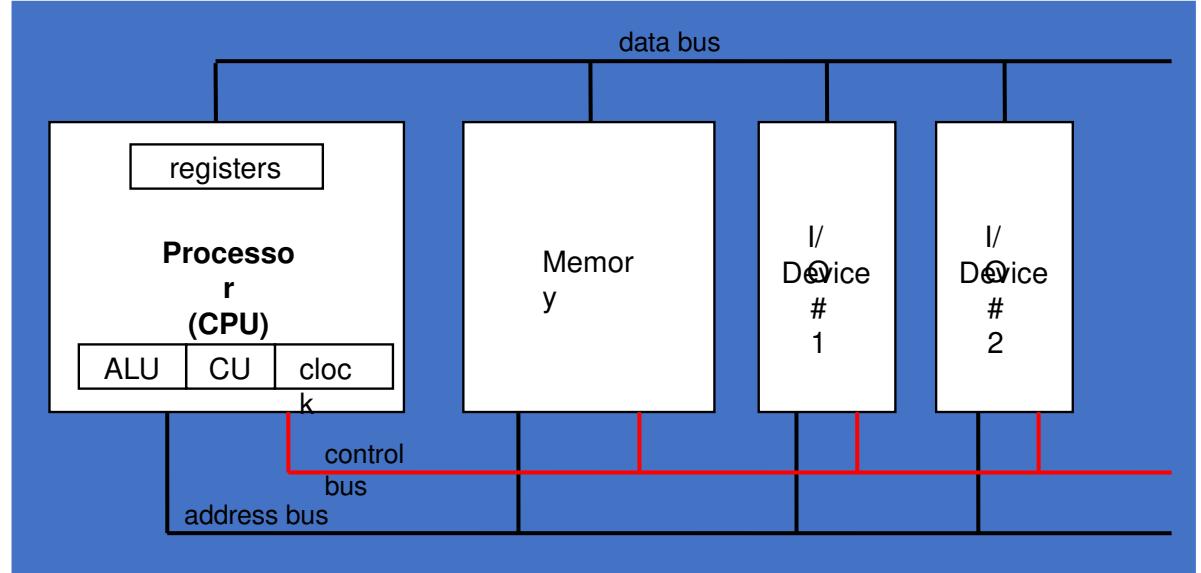
Addressing Modes

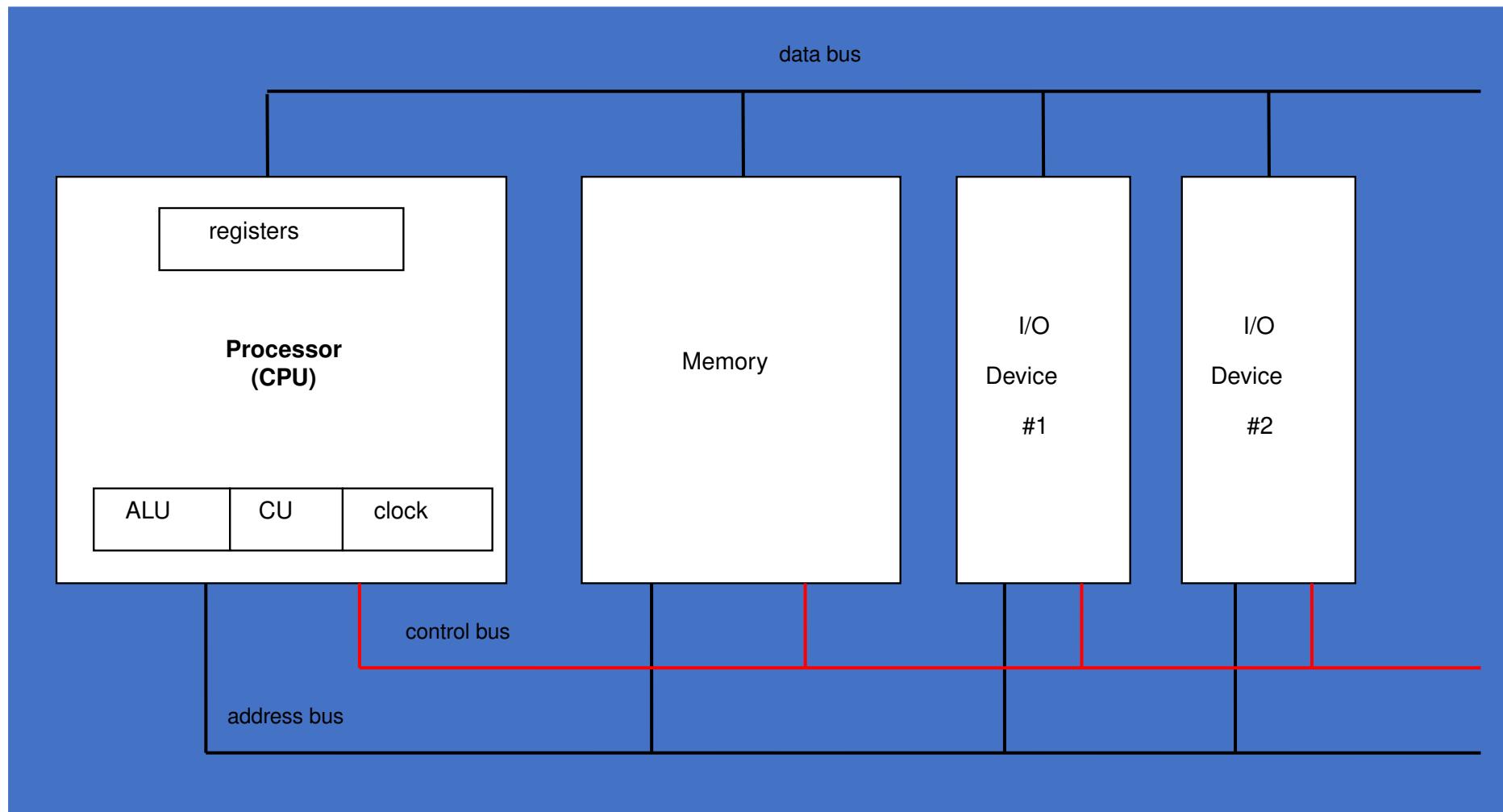
The three basic modes of addressing are:

- Register addressing
- Immediate addressing
- Memory addressing

Basic Computer Organization

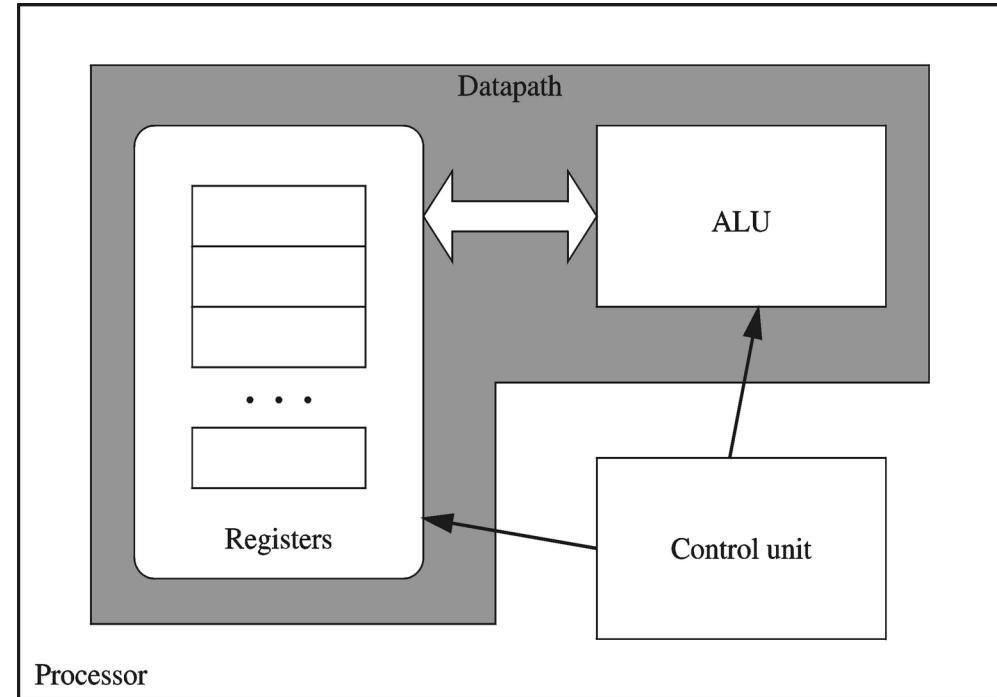
- Since the 1940's, computers have 3 classic components:
 - Processor, called also the CPU (Central Processing Unit)
 - Memory and Storage Devices
 - I/O Devices
- Interconnected with one or more buses
- Bus consists of
 - Data Bus
 - Address Bus
 - Control Bus





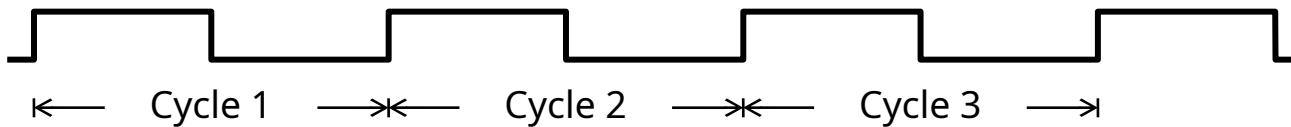
Processor

- ❖ Processor consists of
 - ◊ Datapath
 - ALU
 - Registers
 - ◊ Control unit
- ❖ ALU
 - ◊ Performs arithmetic and logic instructions
- ❖ Control unit (CU)
 - ◊ Generates the control signals required to execute instructions
- ❖ Implementation varies from one processor to another



Clock

- Synchronizes Processor and Bus operations
- Clock cycle = Clock period = $1 / \text{Clock rate}$



- Clock rate = Clock frequency = Cycles per second
 - 1 Hz = 1 cycle/sec 1 KHz = 10^3 cycles/sec
 - 1 MHz = 10^6 cycles/sec 1 GHz = 10^9 cycles/sec
 - 2 GHz clock has a cycle time = $1/(2 \times 10^9) = 0.5$ nanosecond (ns)
- Clock cycles measure the execution of instructions

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

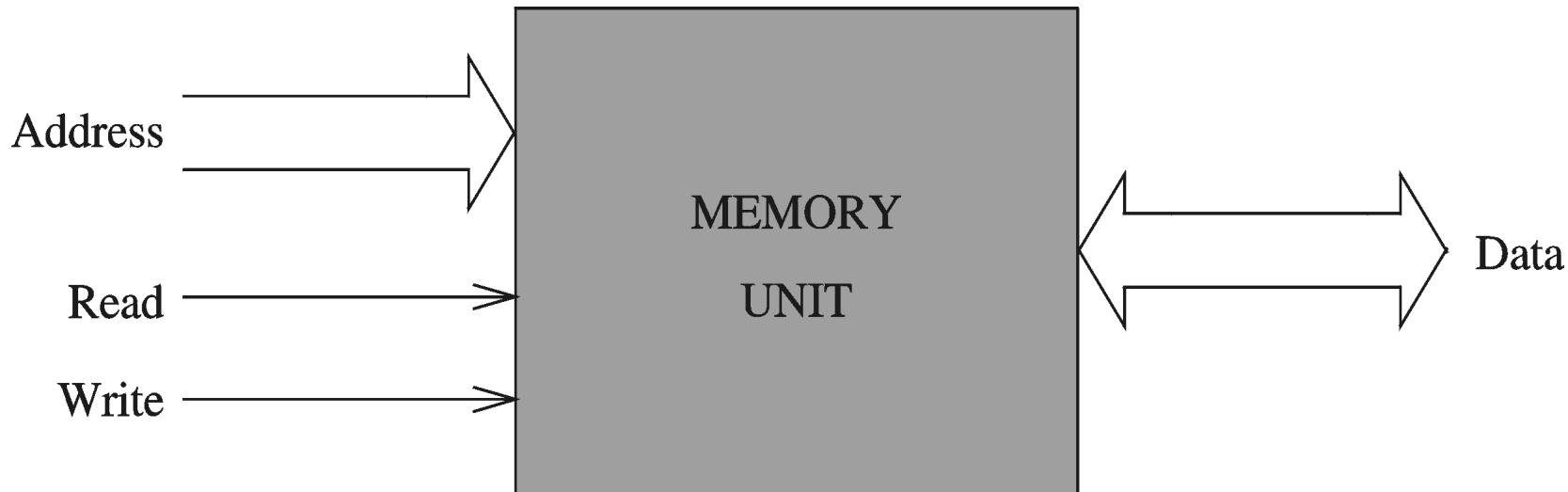
Address Space

Address (in decimal)	Address (in hex)
$2^{32}-1$	FFFFFFF
	FFFFFFE
	FFFFFFD
•	
2	0000002
1	0000001
0	0000000

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

Memory Unit

- Address Bus
 - Address is placed on the address bus ◊ Read
 - Address of location to be read/written ◊ Write
 - Data Bus
 - Data is placed on the data bus
- ❖ Two Control Signals
- ◊ Control whether memory should be read or written

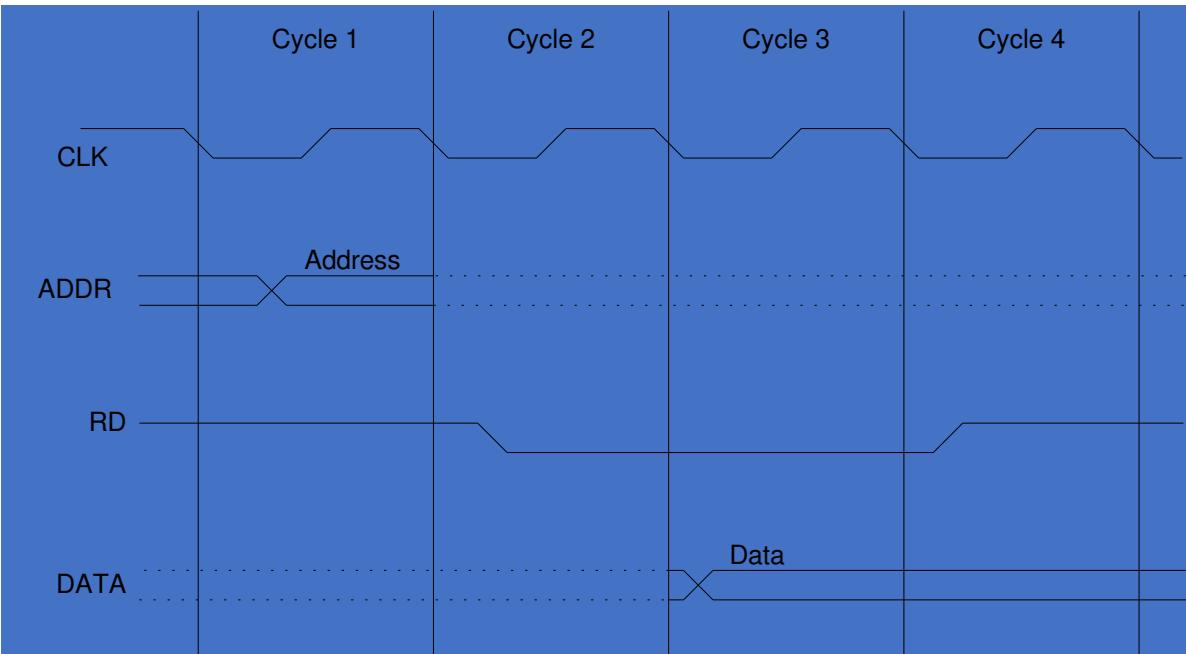


Memory Read and Write Cycles

- Read cycle
 1. Processor places **address** on the address bus
 2. Processor asserts the memory **read** control signal
 3. Processor waits for memory to place the data on the data bus
 4. Processor reads the **data** from the data bus
 5. Processor drops the memory read signal
- Write cycle
 1. Processor places **address** on the address bus
 2. Processor asserts the memory **write** control signal
 3. Processor places the data on the data bus
 4. Wait for memory to **store** the data (**wait states** for slow memory)
 5. Processor drops the memory write signal

Reading from Memory

- Multiple clock cycles are required
- Memory responds much more slowly than the CPU
 - Address is placed on address bus
 - Read Line (RD) goes low, indicating that processor wants to read
 - CPU waits (one or more cycles) for memory to respond
 - Read Line (RD) goes high, indicating that data is on the data bus

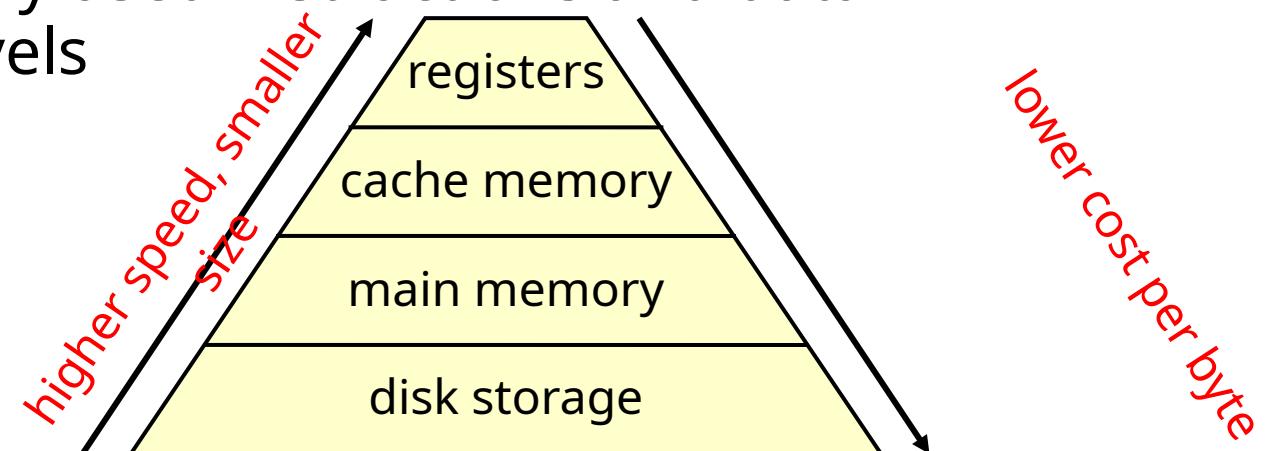


Memory Devices

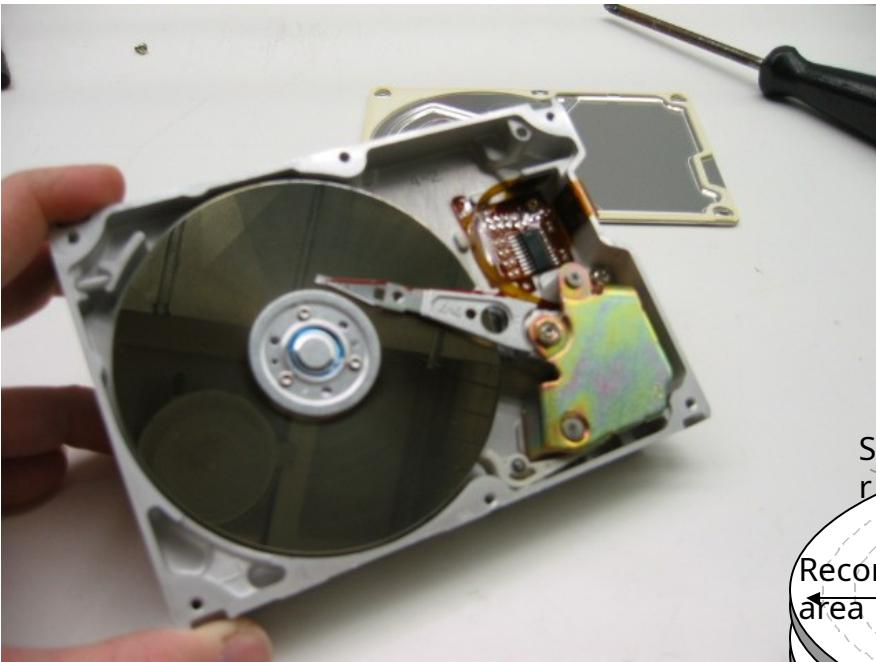
- ROM = Read-Only Memory
 - Stores information permanently (non-volatile)
 - Used to store the information required to startup the computer
 - Many types: ROM, EPROM, EEPROM, and FLASH
 - FLASH memory can be erased electrically in blocks
- RAM = Random Access Memory
 - Volatile memory: data is lost when device is powered off
 - Dynamic RAM (DRAM)
 - Inexpensive, used for main memory, must be refreshed constantly
 - Static RAM (SRAM)
 - Expensive, used for cache memory, faster access, no refresh
 - Video RAM (VRAM)
 - Dual ported: read port to refresh the display, write port for updates

Memory Hierarchy

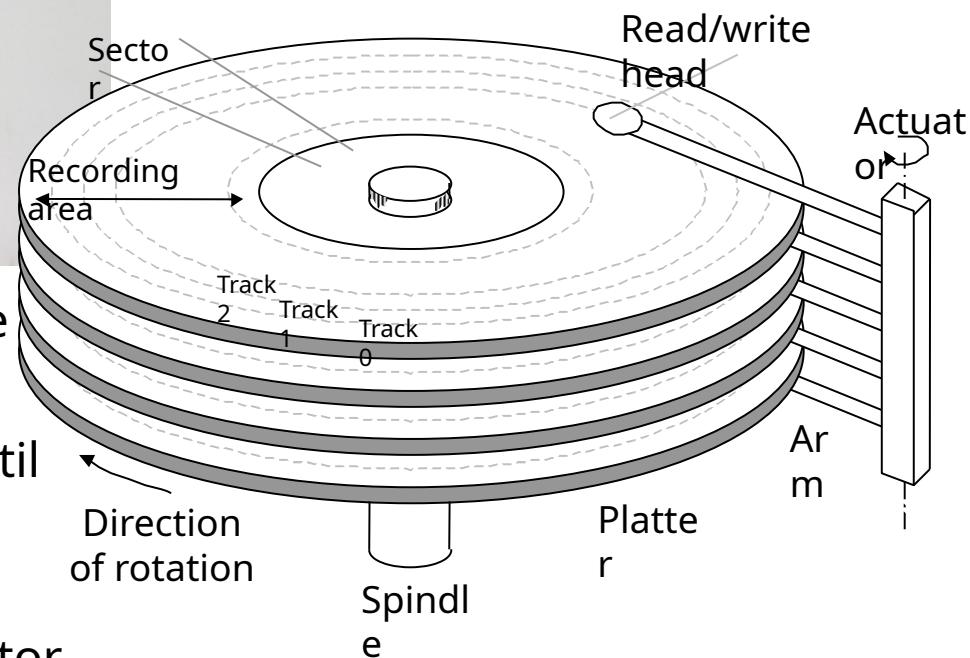
- Registers
 - Fastest storage elements, stores most frequently used data
 - General-purpose registers: accessible to the programmer
 - Special-purpose registers: used internally by the microprocessor
- Cache Memory
 - Fast SRAM that stores recently used instructions and data
 - Recent processors have 2 levels
- Main Memory (DRAM)
- Disk Storage
 - Permanent magnetic storage for files



Magnetic Disk Storage



Disk Access Time =
Seek Time +
Rotation Latency +
Transfer Time



Seek Time: head movement to the desired track (milliseconds)

Rotation Latency: disk rotation until desired sector arrives under the head

Transfer Time: to transfer one sector

Example on Disk Access Time

- Given a magnetic disk with the following properties
 - Rotation speed = 7200 RPM (rotations per minute)
 - Average seek = 8 ms, Sector = 512 bytes, Track = 200 sectors
- Calculate
 - Time of one rotation (in milliseconds)
 - Average time to access a block of 32 consecutive sectors
- Answer
 - Rotations per second = $7200/60 =$
 - Rotation time in milliseconds = $120 \text{ RPS} = 1000/120 = 8.33$
 - Average rotational latency = $\frac{1}{2} \text{ time of half rotation} =$
 - Time to transfer 32 sectors = $\frac{4.17 \text{ ms}}{(32/200)} * 8.33 = 1.33$
 - Average access time = $8 + 4.17 + 1.33 = 13.5$ ms

Register Addressing

In this addressing mode, a register contains the operand. Depending upon the instruction, the register may be the first operand, the second operand or both.

```
MOV DX, TAX_RATE ; Register in first operand  
MOV COUNT, CX ; Register in second operand  
MOV EAX, EBX ; Both the operands are in registers
```

As processing data between registers does not involve memory, it provides fastest processing of data.

Immediate Addressing

An immediate operand has a constant value or an expression. When an instruction with two operands uses immediate addressing, the first operand may be a register or memory location, and the second operand is an immediate constant. The first operand defines the length of the data.

For example:

```
BYTE_VALUE DB 150 ; A byte value is defined
```

```
WORD_VALUE DW 300 ; A word value is defined
```

```
ADD BYTE_VALUE, 65 ; An immediate operand 65 is added
```

```
MOV AX, 45H ; Immediate constant 45H is transferred to AX
```

Direct Memory Addressing

When operands are specified in memory addressing mode, direct access to main memory, usually to the data segment, is required.

This way of addressing results in slower processing of data. To locate the exact location of data in memory, we need the segment start address, which is typically found in the DS register and an offset value.

This offset value is also called effective address.

Direct Memory Addressing

In direct addressing mode, the offset value is specified directly as part of the instruction, usually indicated by the variable name.

The assembler calculates the offset value and maintains a symbol table, which stores the offset values of all the variables used in the program.

Direct-Offset Addressing

This addressing mode uses the arithmetic operators to modify an address. For example, look at the following definitions that define tables of data:

```
BYTE_TABLE DB 14, 15, 22, 45 ; Tables of bytes
```

```
WORD_TABLE DW 134, 345, 564, 123 ; Tables of words
```

Direct Memory Addressing

In direct memory addressing, one of the operands refers to a memory location and the other operand references a register.

For example:

ADD BYTE_VALUE, DL ; Adds the register in the memory location

MOV BX, WORD_VALUE ; Operand from the memory is added to register

Direct Memory Addressing

The following operations access data from the tables in the memory into registers:

```
MOV CL, BYTE_TABLE[2] ; Gets the 3rd element of the BYTE_TABLE  
MOV CL, BYTE_TABLE + 2 ; Gets the 3rd element of the BYTE_TABLE  
MOV CX, WORD_TABLE[3] ; Gets the 4th element of the  
WORD_TABLE MOV CX, WORD_TABLE + 3 ; Gets the 4th element of  
the WORD_TABLE
```

Indirect Memory Addressing

This addressing mode utilizes the computer's ability of Segment:Offset addressing. Generally the base registers EBX, EBP (or BX, BP) and the index registers (DI, SI), coded within square brackets for memory references, are used for this purpose.

Indirect addressing is generally used for variables containing several elements like, arrays. Starting address of the array is stored in, say, the EBX register

Indirect Memory Addressing

The following code snippet shows how to access different elements of the variable.

```
MY_TABLE TIMES 10 DW 0 ; Allocates 10 words (2 bytes) each initialized to 0  
MOV EBX, [MY_TABLE] ; Effective Address of MY_TABLE in EBX  
MOV [EBX], 110 ; MY_TABLE[0] = 110  
ADD EBX, 2 ; EBX = EBX +2  
MOV [EBX], 123 ; MY_TABLE[1] = 123
```

The MOV Instruction

We have already used the MOV instruction that is used for moving data from one storage space to another. The MOV instruction takes two operands.

SYNTAX:

Syntax of the MOV instruction is:

MOV destination, source

The MOV Instruction

The MOV instruction may have one of the following five forms:

- MOV register, register
- MOV register, immediate
- MOV memory, immediate
- MOV register, memory
- MOV memory, register

- Please note that:
- Both the operands in MOV operation should be of same size
- The value of source operand remains unchanged

The MOV instruction causes ambiguity at times. For example, look at the statements.

```
MOV EBX, [MY_TABLE] ; Effective Address of MY_TABLE in EBX  
MOV [EBX], 110 ; MY_TABLE[0] = 110
```

- It is not clear whether you want to move a byte equivalent or word equivalent of the number 110. In such cases, it is wise to use a type specifier.
- Following table shows some of the common type

Type Specifier	Bytes addressed
BYTE	1
WORD	2
DWORD	4
QWORD	8
TBYTE	10

- **EXAMPLE:**

The following program illustrates some of the concepts discussed above. It stores a name 'Zara Ali' in the data section of the memory. Then changes its value to another name 'Nuha Ali' programmatically and displays both the names.

```
section .text
    global main      ;must be declared for linker (ld)
main:   ;tell linker entry point

;writing the name 'Zara Ali'
    mov     edx,9          ;message length
    mov     ecx, name      ;message to write
    mov     ebx,1          ;file descriptor (stdout)
    mov     eax,4          ;system call number (sys_write)
    int     0x80          ;call kernel

    mov     [name], dword 'Nuha'      ; Changed the name to Nuha Ali
;writing the name 'Nuha Ali'
    mov     edx,8          ;message length
    mov     ecx, name      ;message to write
    mov     ebx,1          ;file descriptor (stdout)
    mov     eax,4          ;system call number (sys_write)
```

```
int      0x80          ;call kernel
mov      eax,1         ;system call number (sys_exit)
int      0x80          ;call kernel

section .data
name db 'Zara Ali '
```

When the above code is compiled and executed, it produces following result:

```
Zara Ali Nuha Ali
```

ASSEMBLY VARIABLES

Assembly Variables

- NASM provides various define directives for reserving storage space for variables. The define assembler directive is used for allocation of storage space. It can be used to reserve as well as initialize one or more bytes.

Allocating Storage Space for Initialized Data

- The syntax for storage allocation statement for initialized data is:
- [variable-name] define-directive initial-value [,initial-value]...
• Where, variable-name is the identifier for each storage space. The assembler associates an offset value for each variable name defined in the data segment.

Allocating Storage Space for Initialized Data

- There are five basic forms of the define directive:

Directive	Purpose	Storage Space
DB	Define Byte	allocates 1 byte
DW	Define Word	allocates 2 bytes
DD	Define Doubleword	allocates 4 bytes
DQ	Define Quadword	allocates 8 bytes
DT	Define Ten Bytes	allocates 10 bytes

Allocating Storage Space for Initialized Data

- Following are some examples of using define directives:

• choice	DB	'y'
• Number	DW	12345
• neg_number	DW	-12345
• big_number	DQ	123456789
• real_number1	DD	1.234
• real_number2	DQ	123.456

Allocating Storage Space for Initialized Data

- Please note that:

- Each byte of character is stored as its ASCII value in hexadecimal
- Each decimal value is automatically converted to its 16-bit binary equivalent and stored as a hexadecimal number
- Processor uses the little-endian byte ordering
- Negative numbers are converted to its 2's complement representation
- Short and long floating-point numbers are represented using 32 or 64 bits, respectively

Allocating Storage Space for Uninitialized Data

- The reserve directives are used for reserving space for uninitialized data. The reserve directives take a single operand that specifies the number of units of space to be reserved. Each define directive has a related reserve directive.
- There are five basic forms of the reserve directive:

Directive	Purpose
RESB	Reserve a Byte
RESW	Reserve a Word
RESD	Reserve a Doubleword
RESQ	Reserve a Quadword
REST	Reserve a Ten Bytes

Multiple Definitions

- You can have multiple data definition statements in a program. For example:

```
choice      DB      'Y'          ;ASCII of y = 79H
number1     DW      12345        ;12345D = 3039H
number2     DD      12345679    ;123456789D = 75BCD15H
```

Multiple Initializations

- The TIMES directive allows multiple initializations to the same value. For example, an array named marks of size 9 can be defined and initialized to zero using the following

```
marks    TIMES  9    DW  0
```

The TIMES directive is useful in defining arrays and tables. The following program displays 9 asterisks on the screen:

```
section .text
    global main      ;must be declared for linker (ld)
main:   ;tell linker entry point
        mov     edx,9          ;message length
        mov     ecx, stars      ;message to write
        mov     ebx,1           ;file descriptor (stdout)
        mov     eax,4           ;system call number (sys_write)
        int     0x80            ;call kernel

        mov     eax,1           ;system call number (sys_exit)
        int     0x80            ;call kernel

section .data
stars  times 9 db '*'
```

ASSEMBLY CONSTANTS

Assembly Variables

There are several directives provided by NASM that define constants. We have already used the EQU directive in previous chapters. We will particularly discuss three directives:

- EQU
- %assign
- %define

The EQU Directive

- The EQU directive is used for defining constants. The syntax of the EQU directive is as follows:
- **CONSTANT_NAME EQU expression**
- For example,
- **TOTAL_STUDENTS equ 50**
- You can then use this constant value in your code, like:
 - **mov ecx, TOTAL_STUDENTS**
 - **cmp eax, TOTAL_STUDENTS**

The EQU Directive

- The operand of an EQU statement can be an expression:
 - LENGTH equ 20
 - WIDTH equ 10
- AREA equ length * width
- Above code segment would define AREA as 200

The EQU Directive

- Example:
- The following example illustrates the use of the EQU directive:

```
SYS_EXIT equ 1
SYS_WRITE equ 4
```
- `SYS_EXIT equ 1`
- `SYS_WRITE equ 4`

```
section .text
    global main      ;must be declared for using gcc
main:   ;tell linker entry point
        mov eax, SYS_WRITE
        mov ebx, STDOUT
        mov ecx, msg1
        mov edx, len1
        int 0x80

        mov eax, SYS_WRITE
        mov ebx, STDOUT
        mov ecx, msg2
        mov edx, len2
        int 0x80

        mov eax, SYS_WRITE
        mov ebx, STDOUT
        mov ecx, msg3
        mov edx, len3
        int 0x80
        mov eax, SYS_EXIT      ;system call number (sys_exit)
        int 0x80              ;call kernel

section .data
msg1 db 'Hello, programmers!', 0xA, 0xD
len1 equ $ - msg1
msg2 db 'Welcome to the world of, ', 0xA, 0xD
len2 equ $ - msg2
msg3 db 'Linux assembly programming! '
len3 equ $- msg3
```

- When the above code is compiled and executed, it produces following result:

- Hello, programmers!
- Welcome to the world of,
- Linux assembly programming!

The %assign Directive

- The %assign directive can be used to define numeric constants like the EQU directive. This directive allows redefinition. For example, you may define the constant TOTAL as:

```
%assign TOTAL 10
```

- %assign TOTAL 10
- Later in the code you can redefine it as:
- %assign TOTAL 20
- This directive is case-sensitive.

The %define Directive

- The %define directive allows defining both numeric and string constants. This directive is similar to the #define in C. For example, you may define the constant PTR as:
- `%define PTR [EBP+4]`
- The above code replaces PTR by [EBP+4]. This directive also allows redefinition and it is case sensitive

Arithmetic Instructions

Arithmetic Instructions

- The INC instruction is used for incrementing an operand by one. It works on a single operand that can be either in a register or in memory.
- SYNTAX:
- The INC instruction has the following syntax:
INC destination
- The operand destination could be an 8-bit, 16-bit or 32-bit operand.

Arithmetic Instructions

- EXAMPLE:
- INC EBX ; Increments 32-bit register
- INC DL ; Increments 8-bit register
- INC [count] ; Increments the count variable
-

The DEC Instruction

- The DEC instruction is used for decrementing an operand by one. It works on a single operand that can be either in a register or in memory.
- SYNTAX:
- The DEC instruction has the following syntax:
- **DEC destination**
- The operand destination could be an 8-bit, 16-bit or 32-bit operand.

The DEC Instruction

- EXAMPLE:
- segment .data
 - count dw 0
 - value db 15
- segment .text
 - inc [count]

The DEC Instruction

- dec [value]
- mov ebx, count
- inc word [ebx]
- mov esi, value
- dec byte [esi]

The ADD and SUB Instructions

- The ADD and SUB instructions are used for performing simple addition/subtraction of binary data in byte, word and doubleword size, i.e., for adding or subtracting 8-bit, 16-bit or 32-bit operands respectively.
- SYNTAX:
- The ADD and SUB instructions have the following syntax
- ADD/SUB destination, source

- The ADD/SUB instruction can take place between:
 - Register to register
 - Memory to register
 - Register to memory
 - Register to constant data
 - Memory to constant data
-
- However, like other instructions, memory-to-memory operations are not possible using ADD/SUB instructions. An ADD or SUB operation sets or clears the overflow and carry flags.

- EXAMPLE:
- The following example asks two digits from the user, stores the digits in the EAX and EBX register respectively, adds the values, stores the result in a memory location 'res' and finally displays the result.

```
SYS_EXIT    equ 1
SYS_READ    equ 3
SYS_WRITE   equ 4
STDIN       equ 0
STDOUT      equ 1

segment .data

msg1 db "Enter a digit ", 0xA, 0xD
len1 equ $- msg1

msg2 db "Please enter a second digit", 0xA, 0xD
len2 equ $- msg2

msg3 db "The sum is: "
len3 equ $- msg3

segment .bss
```

```
num1 resb 2
num2 resb 2
res resb 1

section .text
    global main      ;must be declared for using gcc
main:   ;tell linker entry point
    mov eax, SYS_WRITE
    mov ebx, STDOUT
    mov ecx, msg1
    mov edx, len1
    int 0x80

    mov eax, SYS_READ
    mov ebx, STDIN
    mov ecx, num1
    mov edx, 2
    int 0x80

    mov eax, SYS_WRITE
    mov ebx, STDOUT
    mov ecx, msg2
    mov edx, len2
    int 0x80

    mov eax, SYS_READ
    mov ebx, STDIN
    mov ecx, num2
    mov edx, 2
    int 0x80

    mov eax, SYS_WRITE
    mov ebx, STDOUT
    mov ecx, msg3
    mov edx, len3
    int 0x80

; moving the first number to eax register and second number to ebx
; and subtracting ascii '0' to convert it into a decimal number
    mov eax, [number1]
    sub eax, '0'
    mov ebx, [number2]
    sub ebx, '0'
```

```
; add eax and ebx
add eax, ebx
; add '0' to to convert the sum from decimal to ASCII
add eax, '0'

; storing the sum in memory location res
mov [res], eax

; print the sum
mov eax, SYS_WRITE
mov ebx, STDOUT
mov ecx, res
mov edx, 1
int 0x80
exit:
    mov eax, SYS_EXIT
    xor ebx, ebx
```

- When the above code is compiled and executed, it produces following result:

```
Enter a digit:  
3  
Please enter a second digit:  
4  
The sum is:  
7
```

The MUL/IMUL Instruction

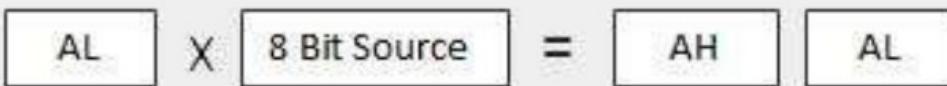
- There are two instructions for multiplying binary data. The MUL (Multiply) instruction handles unsigned data and the IMUL (Integer Multiply) handles signed data. Both instructions affect the Carry and Overflow flag.

- SYNTAX:
- The syntax for the MUL/IMUL instructions is as follows:
 - **MUL/IMUL multiplier**
- Multiplicand in both cases will be in an accumulator, depending upon the size of the multiplicand and the multiplier and the generated product is also stored in two registers depending upon the size of the operands.
Following section explains MULL instructions with three different cases:

When two bytes are multiplied

The multiplicand is in the AL register, and the multiplier is a byte in the memory or in another register. The product is in AX. High order 8 bits of the product is stored in AH and the low order 8 bits are stored in AL

1

**When two one-word values are multiplied**

The multiplicand should be in the AX register, and the multiplier is a word in memory or another register. For example, for an instruction like MUL DX, you must store the multiplier in DX and the multiplicand in AX.

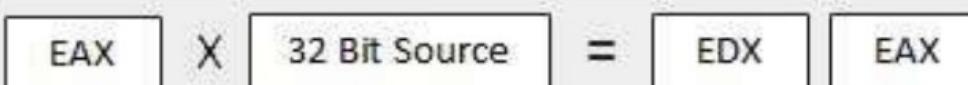
2

The resultant product is a double word, which will need two registers. The High order (leftmost) portion gets stored in DX and the lower-order (rightmost) portion gets stored in AX.

**When two doubleword values are multiplied**

When two doubleword values are multiplied, the multiplicand should be in EAX and the multiplier is a doubleword value stored in memory or in another register. The product generated is stored in the EDX:EAX registers, i.e., the high order 32 bits gets stored in the EDX register and the low order 32-bits are stored in the EAX register.

3



EXAMPLE:

```
MOV AL, 10
MOV DL, 25
MUL DL
...
MOV DL, OFFH      ; DL= -1
MOV AL, 0BEH      ; AL = -66
IMUL DL
```

- EXAMPLE:
- The following example multiplies 3 with 2, and displays the result.

```
section .text
```

```
    global main      ;must be declared for using gcc
main:   ;tell linker entry point

        mov     al,'3'
        sub     al, '0'
        mov     bl, '2'
        sub     bl, '0'
        mul
        add     al, '0'
        mov     [res], al
        mov     ecx,msg
        mov     edx, len
        mov     ebx,1    ;file descriptor (stdout)
        mov     eax,4    ;system call number (sys_write)
        int     0x80    ;call kernel
```

```
nwln
    mov     ecx, res
    mov     edx, 1
    mov     ebx, 1      ;file descriptor (stdout)
    mov     eax, 4      ;system call number (sys_write)
    int    0x80        ;call kernel
    mov     eax, 1      ;system call number (sys_exit)
    int    0x80        ;call kernel

section .data
msg db "The result is:", 0xA, 0xD
len equ $- msg
segment .bss
res resb 1
```

- When the above code is compiled and executed, it produces following result:

```
The result is:
```

```
6
```

The DIV/IDIV Instructions

- The division operation generates two elements - a quotient and a remainder.
- In case of multiplication, overflow does not occur because double-length registers are used to keep the product.
- However, in case of division, overflow may occur. The processor generates an interrupt if overflow occurs.
- The DIV (Divide) instruction is used for unsigned data and the IDIV (Integer Divide) is used for signed data

- SYNTAX:
 - The format for the DIV/IDIV instruction:
 - DIV/IDIV divisor
-
- The dividend is in an accumulator. Both the instructions can work with 8-bit, 16-bit or 32-bit operands. The operation affects all six status flags. Following section explains three cases of division with different operand size:

SN	Scenarios
	When the divisor is 1 byte
1	The dividend is assumed to be in the AX register (16 bits). After division, the quotient goes to the AL register and the remainder goes to the AH register.

16 bit dividend

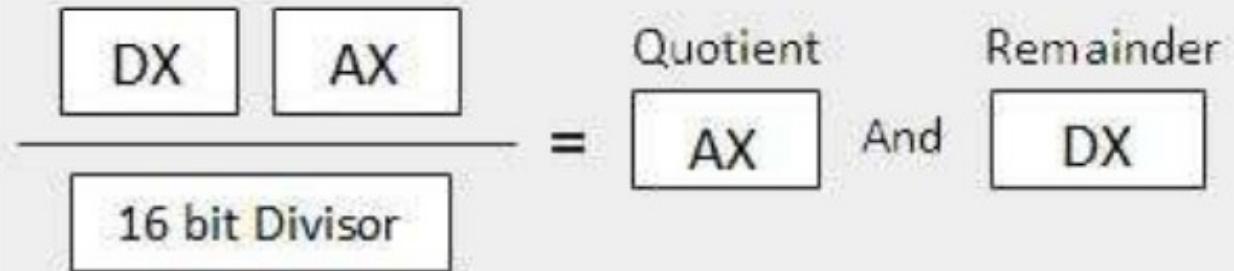


When the divisor is 1 word

The dividend is assumed to be 32 bits long and in the DX:AX registers. The high order 16 bits are in DX and the low order 16 bits are in AX. After division, the 16 bit quotient goes to the AX register and the 16 bit remainder goes to the DX register.

2

32 bit dividend



- EXAMPLE:
- The following example divides 8 with 2. The dividend 8 is stored in the 16 bit AX register and the divisor 2 is stored in the 8 bit BL register

```
section .text
    global main      ;must be declared for using gcc
main:   ;tell linker entry point
        mov     ax, '8'
        sub     ax, '0'
        mov     bl, '2'
        sub     bl, '0'
        div     bl
        add     ax, '0'
```

```
mov      [res], ax
mov      ecx, msg
mov      edx, len
mov      ebx, 1      ;file descriptor (stdout)
mov      eax, 4      ;system call number (sys_write)
int     0x80      ;call kernel
nlwln
mov      ecx, res
mov      edx, 1
mov      ebx, 1      ;file descriptor (stdout)
mov      eax, 4      ;system call number (sys_write)
int     0x80      ;call kernel
mov      eax, 1      ;system call number (sys_exit)
int     0x80      ;call kernel

section .data
msg db "The result is:", 0xA, 0xD
len equ $- msg
segment .bss
res resb 1
```

When the above code is compiled and executed, it produces following result:

The result is:

Beyond basics – L6

Assembly Language Programming

Dr Emmanuel Ahene

Course Outlook...

- Done so far --- Basic Concepts, Beyond basics L2, L3, L4, L5
- Looking up to L6, L7 & L8
 - ❑ Logical Instructions
 - ❑ Assembly Conditions
 - ❑ Procedures
 - ❑ Recursions
 - ❑ Macros
 - ❑ File management
 - ❑ Memory Management

Logical Instructions

- The **processor instruction set** provides the instructions **AND**, **OR**, **XOR**, **TEST** and **NOT Boolean logic**, which tests, sets and clears the bits according to the need of the program.
- The format for these instructions:

SN	Instruction	Format
1	AND	AND operand1, operand2
2	OR	OR operand1, operand2
3	XOR	XOR operand1, operand2
4	TEST	TEST operand1, operand2
5	NOT	NOT operand1

Logical Instructions

- The first operand in all the cases could be either in register or in memory.
- The second operand could be either in register/memory or an immediate (constant) value.

However, memory to memory operations are not possible.

- These instructions compare or match bits of the operands and set the CF, OF, PF, SF and ZF flags.

The AND Instruction

- The AND instruction is used for supporting logical expressions by performing **bitwise AND operation**.
- The **bitwise AND operation** returns 1, if the matching bits from both the operands are 1, otherwise it returns 0. For example:

Operand1 :	0101
Operand2 :	0011

After AND -> Operand1 :	0001

- The **AND operation** can be **used for clearing one or more bits**. For example, say, the BL register contains 0011 1010. If you need to clear the high order bits to zero, you **AND it with 0FH**

- AND BL, 0FH ; This sets BL to 0000 1010
- Let's take up another example. If you want to check whether a given number is odd or even, a simple test would be to check the least significant bit of the number. If this is 1, the number is odd, else the number is even.
- Assuming the number is in AL register, we can write:

```
AND      AL, 01H      ; ANDing with 0000 0001
JZ      EVEN_NUMBER
```

The following program illustrates this:

Example:

```
section .text
    global main
main:
    mov ax, 8h
    and ax, 1
    jz evnn
    mov eax, 4
    mov ebx, 1
    mov ecx, odd_msg
    mov edx, len2
    int 0x80
    jmp outprog
evnn:
    mov ah, 09h
    mov eax, 4
    mov ebx, 1
    mov ecx, even_msg
    mov edx, len1
    int 0x80
outprog:
    mov eax, 1
    int 0x80
section .data
even_msg db 'Even Number!' ;message showing even number
len1 equ $ - even msg
```

```
    mov    edx, len1           ;length of message
    int    0x80                ;call kernel
outprog:
    mov    eax,1               ;system call number (sys_exit)
    int    0x80                ;call kernel
section .data
even_msg db  'Even Number!' ;message showing even number
len1 equ $ - even_msg
odd_msg db  'Odd Number!'  ;message showing odd number
len2 equ $ - odd_msg
```

When the above code is compiled and executed, it produces following result:

```
Even Number!
```

Change the value in the ax register with an odd digit, like:

```
mov  ax, 9h                 ; getting 9 in the ax
```

The program would display:

```
Odd Number!
```

Similarly to clear the entire register you can AND it with 00H.

The OR Instruction

- The OR instruction is used for supporting logical expression by performing bitwise OR operation.
- The bitwise OR operator returns 1, if the matching bits from either or both operands are one. It returns 0, if both the bits are zero.
- For example,
- Operand1: 0101
- Operand2: 0011

- After OR -> Operand1: 0111
- The OR operation can be used for setting one or more bits. For example, let us assume the AL register contains 0011 1010, you need to set the four low order bits, you can OR it with a value 0000 1111, i.e., FH.
- OR BL, 0FH ; This sets BL to 0011 1111

Example:

- The following example demonstrates the OR instruction. Let us store the value 5 and 3 in the AL and the BL register respectively. Then the instruction,
- OR AL, BL
- should store 7 in the AL register:

```
section      .text
    global main
main:
    mov    al, 5           ;must be declared for using gcc
    mov    bl, 3           ;tell linker entry point
    or     al, bl          ;getting 5 in the al
    add    al, byte '0'    ;getting 3 in the bl
    mov    [result], al    ;or al and bl registers, result should be 7
    mov    eax, 4
    mov    ebx, 1
    mov    ecx, result
    mov    edx, 1
    int    0x80

outprog:
    mov    eax, 1           ;system call number (sys_exit)
    int    0x80             ;call kernel

section      .bss
result resb 1
```

- When the above code is compiled and executed, it produces following result:

7

The XOR Instruction

- The XOR instruction implements the bitwise XOR operation.
- The XOR operation sets the resultant bit to 1, if and only if the bits from the operands are different. If the bits from the operands are same (both 0 or both 1), the resultant bit is cleared to 0.
- For example,

Operand1: 0101

Operand2: 0011

After XOR -> Operand1: 0110

- XORing an operand with itself changes the operand to 0. This is used to clear a register.

```
XOR EAX, EAX
```

The TEST Instruction

- The TEST instruction works same as the AND operation, but unlike AND instruction, it does not change the first operand.
- So, if we need to check whether a number in a register is even or odd, we can also do this using the TEST instruction without changing the original number.

```
TEST AL, 01H  
JZ EVEN_NUMBER
```

The NOT Instruction

- The NOT instruction implements the bitwise NOT operation. NOT operation reverses the bits in an operand. The operand could be either in a register or in the memory.
- For example,

Operand1: 0101 0011

After NOT -> Operand1: 1010 1100

Assembly Conditions

- Conditional execution in assembly language is accomplished by several looping and branching instructions.
- These instructions can change the flow of control in a program. Conditional execution is observed in two scenarios:

SN	Conditional Instructions
1	Unconditional jump This is performed by the JMP instruction. Conditional execution often involves a transfer of control to the address of an instruction that does not follow the currently executing instruction. Transfer of control may be forward to execute a new set of instructions, or backward to re-execute the same steps.
2	Conditional jump This is performed by a set of jump instructions <code>j<condition></code> depending upon the condition. The conditional instructions transfer the control by breaking the sequential flow and they do it by changing the offset value in IP.

Let us discuss the CMP instruction before discussing the conditional instructions.

The CMP Instruction

- The CMP instruction compares two operands. It is generally used in conditional execution.
- This instruction basically subtracts one operand from the other for comparing whether the operands are equal or not.
- It does not disturb the destination or source operands.
- It is used along with the conditional jump instruction for decision making.

SYNTAX

CMP destination, source

- CMP compares two numeric data fields.
- The destination operand could be either in register or in memory.
- The source operand could be a constant (immediate) data, register or memory.

EXAMPLE:

```
CMP DX, 00 ; Compare the DX value with zero  
JE L7 ; If yes, then jump to label L7
```

.

.

L7 : . . .

CMP is often used for comparing whether a counter value has reached the number of time a loop needs to be run. Consider the following typical condition:

```
INC EDX  
CMP EDX, 10 ; Compares whether the counter has reached 10  
JLE LP1 ; If it is less than or equal to 10, then jump to LP1
```

Unconditional Jump

- As mentioned earlier this is performed by the JMP instruction.
- Conditional execution often involves a transfer of control to the address of an instruction that does not follow the currently executing instruction.
- Transfer of control may be forward to execute a new set of instructions, or backward to re-execute the same steps.

SYNTAX:

- The JMP instruction provides a label name where the flow of control is transferred immediately. The syntax of the JMP instruction is:

JMP label

EXAMPLE:

- The following code snippet illustrates the JMP instruction:

```
MOV AX, 00      ; Initializing AX to 0
MOV BX, 00      ; Initializing BX to 0
MOV CX, 01      ; Initializing CX to 1
L20:
ADD AX, 01      ; Increment AX
ADD BX, AX      ; Add AX to BX
SHL CX, 1       ; shift left CX, this in turn doubles the CX value
JMP L20         ; repeats the statements
```

Conditional Jump

- If some specified condition is satisfied in conditional jump, the control flow is transferred to a target instruction.
- There are numerous conditional jump instructions, depending upon the condition and data.
- Following are the conditional jump instructions used on signed data used for arithmetic operations:

Instruction	Description	Flags tested
JE/JZ	Jump Equal or Jump Zero	ZF
JNE/JNZ	Jump not Equal or Jump Not Zero	ZF
JG/JNLE	Jump Greater or Jump Not Less/Equal	OF, SF, ZF
JGE/JNL	Jump Greater or Jump Not Less	OF, SF
JL/JNGE	Jump Less or Jump Not Greater/Equal	OF, SF
JLE/JNG	Jump Less/Equal or Jump Not Greater	OF, SF, ZF

Following are the conditional jump instructions used on unsigned data used for logical operations:

Instruction	Description	Flags tested
JE/JZ	Jump Equal or Jump Zero	ZF
JNE/JNZ	Jump not Equal or Jump Not Zero	ZF
JA/JNBE	Jump Above or Jump Not Below/Equal	CF, ZF
JAE/JNB	Jump Above/Equal or Jump Not Below	CF
JB/JNAE	Jump Below or Jump Not Above/Equal	CF
JBE/JNA	Jump Below/Equal or Jump Not Above	AF, CF

The following conditional jump instructions have special uses and check the value of flags:

Instruction	Description	Flags tested
JXCZ	Jump if CX is Zero	none
JC	Jump If Carry	CF
JNC	Jump If No Carry	CF
JO	Jump If Overflow	OF
JNO	Jump If No Overflow	OF
JP/JPE	Jump Parity or Jump Parity Even	PF
JNP/JPO	Jump No Parity or Jump Parity Odd	PF
JS	Jump Sign (negative value)	SF
JNS	Jump No Sign (positive value)	SF

*The syntax for the J set of instructions:
Example,*

CMP	AL, BL
JE	EQUAL
CMP	AL, BH
JE	EQUAL
CMP	AL, CL
JE	EQUAL
NON_EQUAL:	...
EQUAL:	...

Example:

- The following program displays the largest of three variables. The variables are double-digit variables. The three variables num1, num2 and num3 have values 47, 72 and 31 respectively:

```
section .text
    global main          ;must be declared for using gcc

main:   ; tell linker entry point
        mov    ecx, [num1]
        cmp    ecx, [num2]
        jg     check_third_num
        mov    ecx, [num3]
check_third_num:
        cmp    ecx, [num3]
```

```
jg    _exit
    mov  ecx, [num3]
_exit:
    mov  [largest], word ecx
    mov  ecx, msg
    mov  edx, len
    mov  ebx, 1      ;file descriptor (stdout)
    mov  eax, 4      ;system call number (sys_write)
    int  0x80        ;call kernel
    nwln
    mov  ecx, largest
    mov  edx, 2
    mov  ebx, 1      ;file descriptor (stdout)
    mov  eax, 4      ;system call number (sys_write)
    int  0x80        ;call kernel

    mov  eax, 1
    int  80h

section .data
msg db "The largest digit is: ", 0xA, 0xD
len equ $- msg
num1 dd '47'
num2 dd '22'
num3 dd '31'

segment .bss
largest resb 2
```

- When the above code is compiled and executed, it produces following result:
- The largest digit is: 47

Assembly Loops

Assembly Loops

- The JMP instruction can be used for implementing loops.
- For example, the following code snippet can be used for executing the loop-body 10 times.

```
MOV      CL, 10
L1:
<LOOP-BODY>
DEC      CL
JNZ      L1
```

- The processor instruction set however includes a group of loop instructions for implementing iteration. The basic LOOP instruction has the following syntax:

- **LOOP label**
- Where, label is the target label that identifies the target instruction as in the jump instructions.
- The LOOP instruction assumes that the ECX register contains the loop count.
- When the loop instruction is executed, the ECX register is decremented and the control jumps to the target label, until the ECX register value, i.e., the counter reaches the value zero.

- The above code snippet could be written as:

```
mov ECX, 10
11:
<loop body>
loop 11
```

- Example: The following program prints the number 1 to 9 on the screen:

```
section .text
    global main          ;must be declared for using gcc
main:           ;tell linker entry point
    mov ecx,10
    mov eax, '1'

l1:
    mov [num], eax
    mov eax, 4
    mov ebx, 1
    push ecx
```

```
    mov ecx, num
    mov edx, 1
    int 0x80
    mov eax, [num]
    sub eax, '0'
    inc eax
    add eax, '0'
    pop ecx
    loop 11
    mov eax, 1      ; system call number (sys_exit)
    int 0x80      ; call kernel
section .bss
num resb 1
```

When the above code is compiled and executed, it produces following result:

123456789

Assembly Numbers

- Numerical data is generally represented in binary system. Arithmetic instructions operate on binary data.
- When numbers are displayed on screen or entered from keyboard, they are in ASCII form.
- So far, we have converted input data in ASCII form to binary for arithmetic calculations and converted the result back to binary.

Hexadecimal (base 16)

Hex	Bits ("nibble")	Decimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7

Hex	Bits ("nibble")	Decimal
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

Hexadecimal (base 16)

- 0x10

- $0x10 = 16 * 1 + 0 = 16$

- $0x10 = 0001\ 0000$

- 0xAF

- $0xAF = 16 * A + F = 16 * 10 + 15 = 175$

- $0xAF = 1010\ 1111$

ASCII

- Characters (often) represented in ASCII
 - 1 byte/char = 2 hex digits/char

ASCII Table

Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char	Dec	Hex	Oct	Char
0	0	0		32	20	40	[space]	64	40	100	@	96	60	140	'
1	1	1		33	21	41	!	65	41	101	A	97	61	141	a
2	2	2		34	22	42	"	66	42	102	B	98	62	142	b
3	3	3		35	23	43	#	67	43	103	C	99	63	143	c
4	4	4		36	24	44	\$	68	44	104	D	100	64	144	d
5	5	5		37	25	45	%	69	45	105	E	101	65	145	e
6	6	6		38	26	46	&	70	46	106	F	102	66	146	f
7	7	7		39	27	47	'	71	47	107	G	103	67	147	g
8	8	10		40	28	50	(72	48	110	H	104	68	150	h
9	9	11		41	29	51)	73	49	111	I	105	69	151	i
10	A	12		42	2A	52	*	74	4A	112	J	106	6A	152	j
11	B	13		43	2B	53	+	75	4B	113	K	107	6B	153	k
12	C	14		44	2C	54	,	76	4C	114	L	108	6C	154	l
13	D	15		45	2D	55	-	77	4D	115	M	109	6D	155	m
14	E	16		46	2E	56	.	78	4E	116	N	110	6E	156	n
15	F	17		47	2F	57	/	79	4F	117	O	111	6F	157	o
16	10	20		48	30	60	0	80	50	120	P	112	70	160	p
17	11	21		49	31	61	1	81	51	121	Q	113	71	161	q
18	12	22		50	32	62	2	82	52	122	R	114	72	162	r
19	13	23		51	33	63	3	83	53	123	S	115	73	163	s
20	14	24		52	34	64	4	84	54	124	T	116	74	164	t
21	15	25		53	35	65	5	85	55	125	U	117	75	165	u
22	16	26		54	36	66	6	86	56	126	V	118	76	166	v
23	17	27		55	37	67	7	87	57	127	W	119	77	167	w
24	18	30		56	38	70	8	88	58	130	X	120	78	170	x
25	19	31		57	39	71	9	89	59	131	Y	121	79	171	y
26	1A	32		58	3A	72	:	90	5A	132	Z	122	7A	172	z
27	1B	33		59	3B	73	;	91	5B	133	[123	7B	173	{
28	1C	34		60	3C	74	<	92	5C	134	\	124	7C	174	
29	1D	35		61	3D	75	=	93	5D	135]	125	7D	175	}
30	1E	36		62	3E	76	>	94	5E	136	^	126	7E	176	~
31	1F	37		63	3F	77	?	95	5F	137	-	127	7F	177	

example

```
section .text
    global main          ;must be declared for using gcc
main:   ;tell linker entry point
        mov    eax,'3'
        sub    eax, '0'
        mov    ebx, '4'
        sub    ebx, '0'
        add    eax, ebx
        add    eax, '0'
        mov    [sum], eax
        mov    ecx, msg
        mov    edx, len
        mov    ebx,1      ;file descriptor (stdout)
        mov    eax,4      ;system call number (sys_write)
        int    0x80      ;call kernel
```

```
nwln
    mov     ecx, sum
    mov     edx, 1
    mov     ebx, 1      ; file descriptor (stdout)
    mov     eax, 4      ; system call number (sys_write)
    int    0x80        ; call kernel
    mov     eax, 1      ; system call number (sys_exit)
    int    0x80        ; call kernel

section .data
msg db "The sum is:", 0xA, 0xD
len equ $ - msg
segment .bss
sum resb 1
```

- When the above code is compiled and executed, it produces following result:

```
The sum is:  
7
```

- Such conversions are however, has an overhead and assembly language programming allows processing numbers in a more efficient way, in the binary form.
- Decimal numbers can be represented in two forms:
 - ASCII form
 - BCD or Binary Coded Decimal form

ASCII Representation

- In ASCII representation, decimal numbers are stored as string of ASCII characters. For example, the decimal value 1234 is stored as:
- **31 32 33 34H**
- Where, 31H is ASCII value for 1, 32H is ASCII value for 2, and so on. There are the following four instructions for processing numbers in ASCII representation:

- **AAA** - ASCII Adjust After Addition
- **AAS** - ASCII Adjust After Subtraction
- **AAM** - ASCII Adjust After Multiplication
- **AAD** - ASCII Adjust Before Division

- These instructions do not take any operands and assumes the required operand to be in the AL register.
- The following example uses the AAS instruction to demonstrate the concept:

```
section .text
    global main          ;must be declared for using gcc
main:   ;tell linker entry point
        sub    ah, ah
        mov    al, '9'
        sub    al, '3'
        aas
        or     al, 30h
        mov    [res], ax

        mov    edx, len ;message length
        mov    ecx, msg ;message to write
        mov    ebx, 1  ;file descriptor (stdout)
        mov    eax, 4  ;system call number (sys_write)
        int    0x80    ;call kernel
```

```
        mov     edx, 1      ; message length
        mov     ecx, res    ; message to write
        mov     ebx, 1      ; file descriptor (stdout)
        mov     eax, 4      ; system call number (sys_write)
        int     0x80       ; call kernel
        mov     eax, 1      ; system call number (sys_exit)
        int     0x80       ; call kernel

section .data
msg db 'The Result is:',0xa
len equ $ - msg
section .bss
res resb 1
```

- When the above code is compiled and executed, it produces following result:
- The Result is: 6



BCD Representation

- There are two types of BCD representation:
 - Unpacked BCD representation
 - Packed BCD representation

In unpacked BCD representation, each byte stores the binary equivalent of a decimal digit. For example, the number 1234 is stored as:

01

02

03

04H

- There are two instructions for processing these numbers:
 - AAM - ASCII Adjust After Multiplication
 - AAD - ASCII Adjust Before Division

The four ASCII adjust instructions, AAA, AAS, AAM and AAD can also be used with unpacked BCD representation. In packed BCD representation, each digit is stored using four bits. Two decimal digits are packed into a byte. For example, the number 1234 is stored as:

12

34H

- There are two instructions for processing these numbers:
 - DAA - Decimal Adjust After Addition
 - DAS - decimal Adjust After Subtraction
- There is no support for multiplication and division in packed BCD representation.

- The program adds up two 5-digit decimal numbers and displays the sum.
Example:

```
section .text
    global main          ;must be declared for using gcc

main:     ;tell linker entry point

        mov     esi, 4  ;pointing to the rightmost digit
        mov     ecx, 5  ;num of digits
        clc

add_loop:
        mov     al, [num1 + esi]
        adc     al, [num2 + esi]
        aaa
        pushf
        or      al, 30h
        popf
        mov     [sum + esi], al
        dec     esi
        loop   add_loop
        mov     edx, len ;message length
```

```
mov      ecx, msg ; message to write
mov      ebx, 1   ; file descriptor (stdout)
mov      eax, 4   ; system call number (sys_write)
int     0x80    ; call kernel
```

```
mov      edx, 5   ; message length
mov      ecx, sum ; message to write
mov      ebx, 1   ; file descriptor (stdout)
mov      eax, 4   ; system call number (sys_write)
int     0x80    ; call kernel
```

```
mov      eax, 1   ; system call number (sys_exit)
int     0x80    ; call kernel
```

```
section .data
msg db 'The Sum is:',0xa
len equ $ - msg
num1 db '12345'
num2 db '23456'
sum db ''
```

Result:

The Sum is:
35801

Assembly strings ---done

Now Arrays:

- We have already discussed that the data definition directives to the assembler are used for allocating storage for variables. The variable could also be initialized with some specific value. The initialized value could be specified in hexadecimal, decimal or binary form.

For example, we can define a word variable months in either of the following way:

```
MONTHS    DW      12  
MONTHS    DW      0CH  
MONTHS    DW      0110B
```

The data definition directives can also be used for defining a one dimensional array. Let us define a one dimensional array of numbers.

```
NUMBERS  DW  34, 45, 56, 67, 75, 89
```

THAT WAS

an array of six words each initialized with the numbers 34, 45, 56, 67, 75, 89. This

allocates $2 \times 6 = 12$ bytes of consecutive memory space.

The TIMES directive can also be used for multiple initializations to the same value. Using TIMES, the INVENTORY array can be defined as

```
INVENTORY TIMES 8 DW 0
```

NEXT ASSEMBLY PROCEDURES ---

DONE

STACKS DATA STRUCTURE

- Push
- Pop
- Assembly language provides two instructions for stack operations: PUSH and POP. These instructions have syntaxes like:

PUSH
POP

operand
address/*register*

Push enters an item on the stack, and pop retrieves an item, moving the rest of the items in the stack up one level.

Assembly recursion

- A recursion procedure is one that calls itself. There are two kinds of recursion: direct and indirect. In direct recursion, the procedure calls itself and in indirect recursion, the first procedure calls a second procedure, which in turn, calls the first procedure.

Assembly MACROS

- A macro is a sequence of instructions, assigned by a name and could be used anywhere in the program.

In NASM, macros are defined with **%macro** and **%endmacro** directives.

The macro begins with the %macro directive and ends with the %endmacro directive.

The Syntax for macro definition:

```
%macro macro_name    number_of_params  
<macro body>  
%endmacro
```

ASSEMBLY FILE MANAGEMENT

- The system considers any input or output data as stream of bytes. There are three standard file streams:
 - Standard input (stdin)
 - Standard output (stdout)
 - Standard error (stderr)

Creating and Opening a File

- For creating and opening a file, perform the following tasks:
 - Put the system call `sys_creat()` number 8, in the EAX register
 - Put the filename in the EBX register
 - Put the file permissions in the ECX registerThe system call returns the file descriptor of the created file in the EAX register, in case of error, the error code is in the EAX register.

Opening an Existing File

- For opening an existing file, perform the following tasks:

Put the system call `sys_open()` number 5, in the EAX register

Put the filename in the EBX register

Put the file access mode in the ECX register

Put the file permissions in the EDX register

The system call returns the file descriptor of the created file in the EAX register, in case of error, the error code is in the EAX register.

Among the file access modes, most commonly used are: read-only (0), write-only (1), and read-write (2).

Reading from a File

For reading from a file, perform the following tasks:

- Put the system call `sys_read()` number 3, in the EAX register
- Put the file descriptor in the EBX register
- Put the pointer to the input buffer in the ECX register
- Put the buffer size, i.e., the number of bytes to read, in the EDX register.

The system call returns the number of bytes read in the EAX register, in case of error, the error code is in the EAX register.

Writing to a File

- For writing to a file, perform the following tasks:
 - Put the system call `sys_write()` number 4, in the EAX register
 - Put the file descriptor in the EBX register
 - Put the pointer to the output buffer in the ECX register
 - Put the buffer size, i.e., the number of bytes to write, in the EDX registerThe system call returns the actual number of bytes written in the EAX register, in case of error, the error code is in the EAX register.

Closing a File

For closing a file, perform the following tasks:

- Put the system call `sys_close()` number 6, in the EAX register

- Put the file descriptor in the EBX register

The system call returns, in case of error, the error code in the EAX register.

Updating a File

For updating a file, perform the following tasks:

Put the system call sys_lseek () number 19, in the EAX register

Put the file descriptor in the EBX register

Put the offset value in the ECX register

Put the reference position for the offset in the EDX register

The reference position could be:

Beginning of file - value 0

Current position - value 1

End of file - value 2

The system call returns, in case of error, the error code in the EAX register

Memory management

the **sys_brk()** system call is provided by the kernel, to allocate memory without the need of moving it later. This call allocates memory right behind application image in memory. This system function allows you to set the highest available address in the data section. This system call takes one parameter, which is the highest memory address need to be set. This value is stored in the EBX register.

In case of any error, **sys_brk()** returns -1 or returns the negative error code itself. The following example demonstrates dynamic memory allocation.