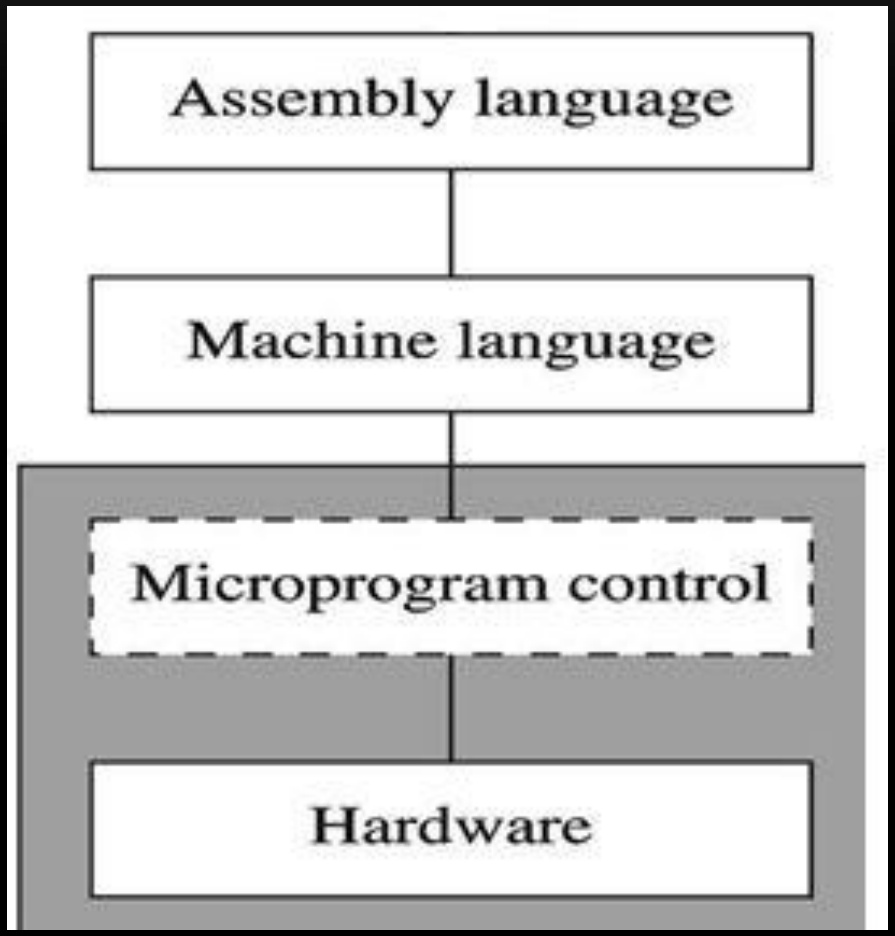
ASSEMBLY LANGUAGE PROGRAMMING -LECTURE #1

Course Teacher-
Shaila Rahman

Assembly language interaction to hardware



Example of machine-language

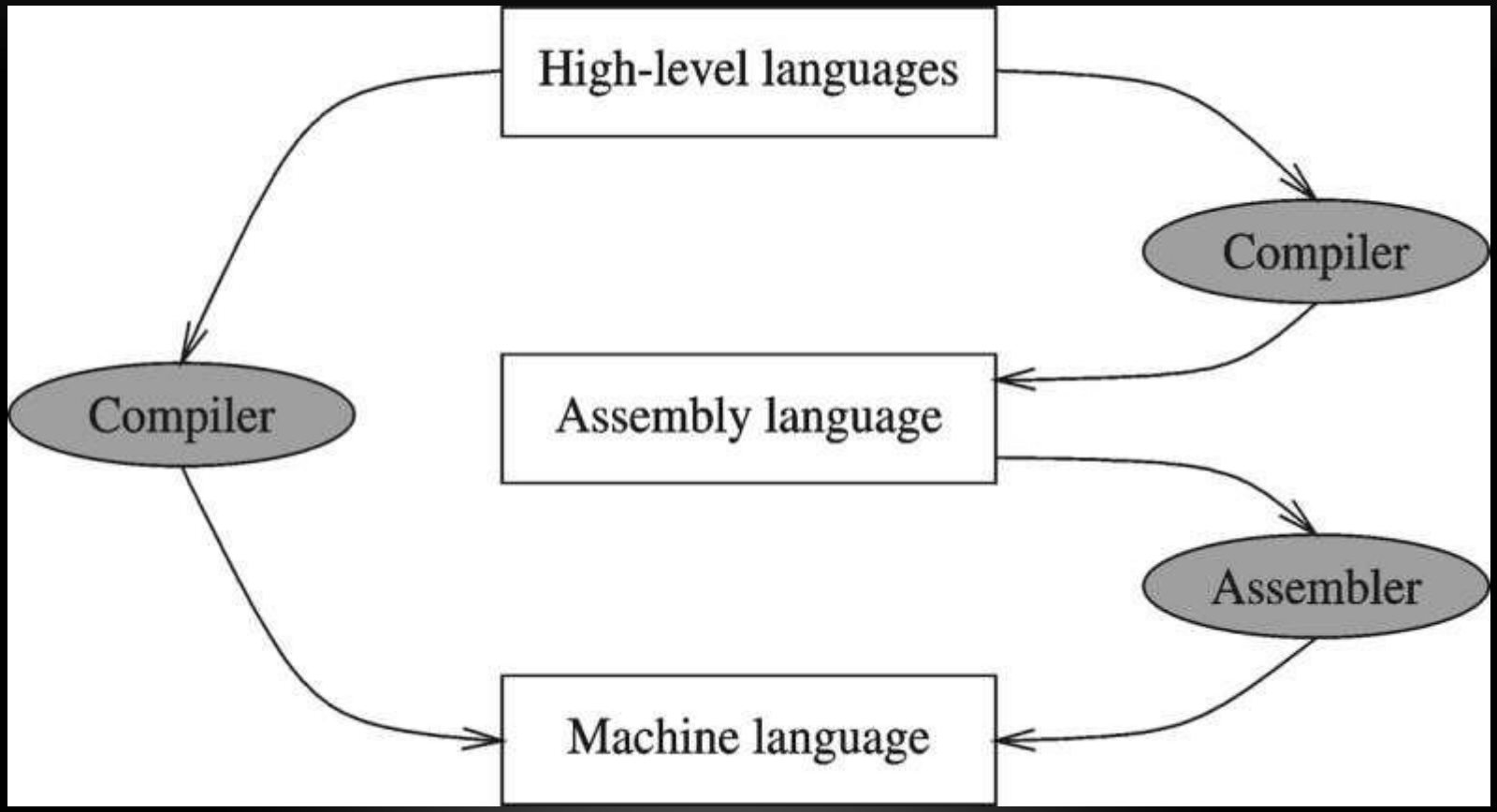
Machine language is composed of binary data:

10100001 10111100 10010011 00000100
00001000 00000011 00000101 11000000

Assembly and Machine Language

- Machine language
  - Friendly 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, memory locations and I/O ports.
  - Slightly higher-level language, that's why called mid-level language.
  - A bit user friendly than machine language to the programmer.
  - Programmer needs detail knowledge about hardware.
- Assemblers translate assembly to machine code
- Compilers translate high-level programs to machine code
  - Either directly, or
  - Indirectly via an assembler

Compiler and Assembler



Assembler

An assembler is a program that converts source-code programs written in assembly language into object files in machine language

Input File: name.asm ; programmer written text file

output: File: name.obj ; machine code but not executable

Then further this .obj file is linked with library functions and the output is: name.exe which is the executable file

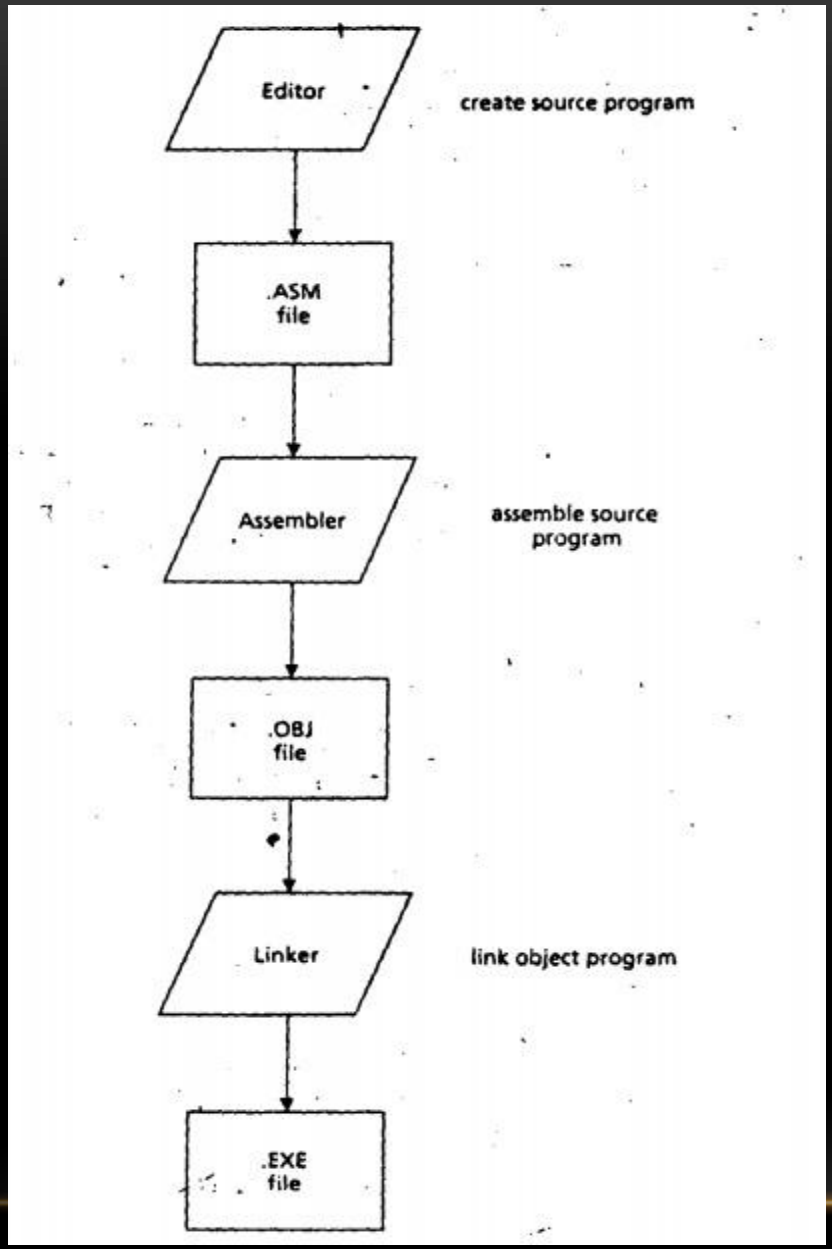
Popular assemblers have emerged over the years for the Intel family of processors. These include …

- ✧ TASM (Turbo Assembler from Borland)
- ✧ NASM (Netwide Assembler for both Windows and Linux)

Creating and Running a Program

The four steps are:

I. Use a text editor or word processor to create a source programing file.
2. Use an assembler to create a machine language object file.
3. Use the LINK program to link one or more object files to create a run file.
4. Execute the run file.


Editor
create source program
.ASM
file
Assembler
assemble source
program
.OBJ
file
Linker
link object program
.EXE
file

8086Emulator 8086EMU

EMU8086 - MICROPROCESSOR EMULATOR is a free emulator for multiple platforms. It provides its user with the ability to emulate 8086 processors, which were used in Macintosh and Windows computers from the 1980s and early 1990s. It can emulate a large amount of code on these microprocessors, a user can program their own assembly code to run on it.

This shows the register contents, memory locations and the corresponding machine code of the assembly program.

This executes and also give a virtual processor expression.

Emulator Vs. Simulator

An emulator works by duplicating every aspect of the original device's behavior. It basically simulates all of the hardware the real device uses, allowing the exact same software to run on it unmodified. You could copy an app off the original device, put it in the emulator, and it wouldn't even notice the difference.

A simulator, on the other hand, sets up a similar environment to the original device's OS, but doesn't attempt to simulate the real device's hardware. Some programs may run a little differently, and it may require other changes (like that the program be compiled for the computer's CPU instead of the device's), but it's a close enough match that you can do most of your development against the simulator.

mnemonics

Mnemonics are used in assembly language, to specify an opcode that represents a complete and operational machine language instruction.

This is later translated by the assembler to generate the object code.

For example, the mnemonic MOV is used in assembly language for copying and moving data between registers and memory locations.

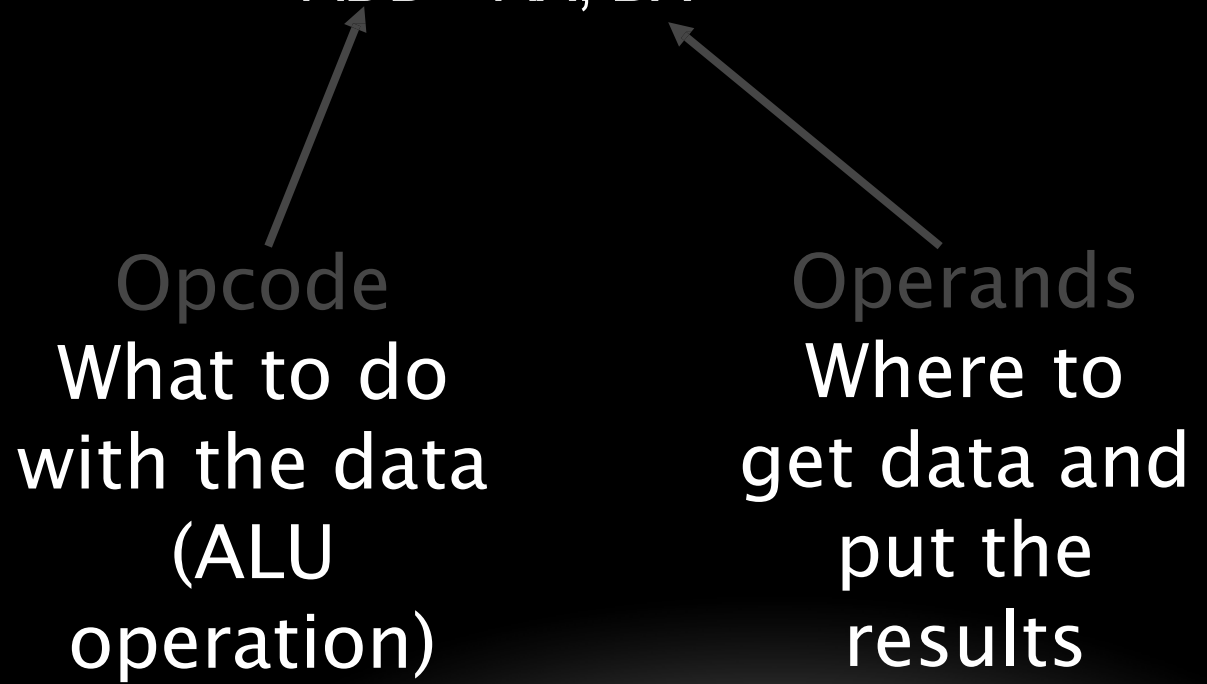
Instructions and Machine Language

- Each command of a program is called an instruction (it instructs the computer what to do).
- The set of all instructions (in binary form) for a specific processor 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.

- Built from two pieces



Types of Opcodes

- Arithmetic, logical
  - ADD, SUB, MUL, DIV
  - AND, OR, XOR, TEST
  - CMP
- Memory load/store and Interface
  - MOV
  - IN/OUT
- Control transfer
  - JMP- unconditional
  - JNC/JC- conditional
  - INT
- Complex String
  - MOVSB
  - STC

BCD (Binary Coded Decimal) Code

Decimal	Binay (BCD) 8 4 2 1
0	0 0 0 0
1	0 0 0 1
2	0 0 1 0
3	0 0 1 1
4	0 1 0 0
5	0 1 0 1
6	0 1 1 0
7	0 1 1 1
8	1 0 0 0
9	1 0 0 1

Binary-coded decimal (BCD) is a class of binary encodings of decimal numbers where each digit is represented of decimal number is represented by a fixed number of bits, usually four.

ASCII

- ASCII, American Standard Code for Information Interchange, is a scheme used for assigning numeric values to punctuation marks, spaces, numbers and other characters.
- ASCII uses 7 bits to represent characters.
- The values 000 0000 through 111 1111 or 00 through 7F are used giving ASCII the ability to represent 128 different characters. An extended version of ASCII assigns characters from 80 through FF.

Examples of ASCII CODE

Character	ASCII Code (hex)	ASCII Code (binary)
R	52	0101 0010
G	47	0100 0111
space	20	0010 0000
2	32	0011 0010
z	7A	0111 1010

A snap Shot of ASCII Table

Table 2.5 ASCII Code

Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	00	<CC>	32	20	SP	64	40	@	96	60	`
1	01	<CC>	33	21	!	65	41	A	97	61	a
2	02	<CC>	34	22	"	66	42	B	98	62	b
3	03	<CC>	35	23	#	67	43	C	99	63	c
4	04	<CC>	36	24	$	68	44	D	100	64	d
5	05	<CC>	37	25	%	69	45	E	101	65	e
6	06	<CC>	38	26	&	70	46	F	102	66	f
7	07	<CC>	39	27	'	71	47	G	103	67	g
8	08	<CC>	40	28	(	72	48	H	104	68	h
9	09	<CC>	41	29	)	73	49	I	105	69	i
10	0A	<CC>	42	2A	*	74	4A	J	106	6A	j
11	0B	<CC>	43	2B	+	75	4B	K	107	6B	k
12	0C	<CC>	44	2C	,	76	4C	L	108	6C	l
13	0D	<CC>	45	2D	-	77	4D	M	109	6D	m
14	0E	<CC>	46	2E	.	78	4E	N	110	6E	n
15	0F	<CC>	47	2F	/	79	4F	O	111	6F	o
16	10	<CC>	48	30	0	80	50	P	112	70	p
17	11	<CC>	49	31	1	81	51	Q	113	71	q
18	12	<CC>	50	32	2	82	52	R	114	72	r
19	13	<CC>	51	33	3	83	53	S	115	73	s
20	14	<CC>	52	34	4	84	54	T	116	74	t
21	15	<CC>	53	35	5	85	55	U	117	75	u
22	16	<CC>	54	36	6	86	56	V	118	76	v
23	17	<CC>	55	37	7	87	57	W	119	77	w
24	18	<CC>	56	38	8	88	58	X	120	78	x

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

Advantages of Assembly Language

1. Shows how program interfaces with the processor, operating system, and BIOS.
2. Shows how data is represented and stored in memory and on external devices.
3. Clarifies how processor accesses and executes instructions and how instructions access and process data.
4. Clarifies how a program accesses external devices.