Computer Architecture & Assembly Language

Chapter 5

THE BASIC CONCEPTS OF ASSEMBLY LANGUAGE

VIET NAM NATIONAL UNIVERSITY HO CHI MINH CITY
UNIVERSITY OF SCIENCE

Thái Hùng Văn
thvan@fit.hcmus.edu.vn

What is Assembly Language?

- Each personal computer has a microprocessor that manages the computer's arithmetical, logical, and control activities.
- Each family of processors has its own set of instructions for handling various operations. They are called 'machine language instructions' (MLIs).
- A processor understands only MLIs, which are series of 1's and 0's. ML is too obscure and complex for programming. So, the assembly language (AL) is designed to make programmer’s job easier.
- In AL, an instruction is an easy-to-remember form called a mnemonic.
- Assembler: Translates AL instructions into ML

Advantages of Assembly Language

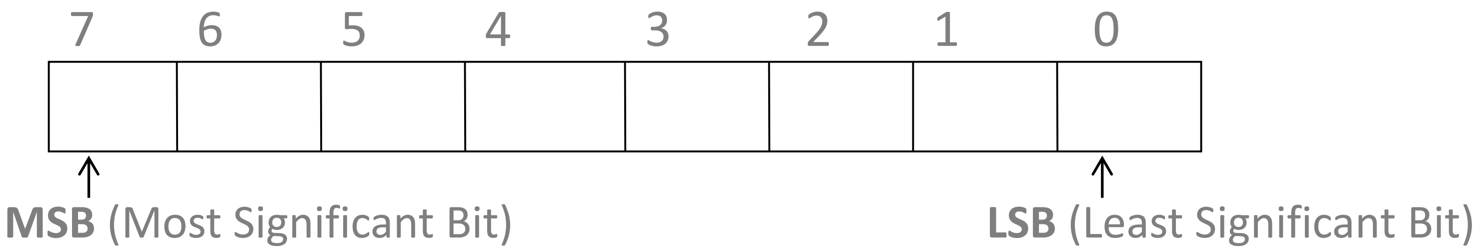
- Having an understanding of AL makes one aware of −
  - ❑ **How programs interface with OS, processor, and BIOS;**
  - ❑ **How data is represented in memory and other external devices;**
  - ❑ **How the processor accesses and executes instruction;**
  - ❑ **How instructions access and process data;**
  - ❑ **How a program accesses external devices.**
- Other advantages of using assembly language are −
  - ❑ **It requires less memory and execution time;**
  - ❑ **It allows hardware-specific complex jobs in an easier way;**
  - ❑ **It is suitable for time-critical jobs;**
  - ❑ **It is most suitable for writing interrupt service routines and other memory resident programs.**

Basic Features of PC Hardware

- The main internal hardware of a PC consists of processor, memory, and registers. Registers are processor components that hold data and address. To execute a program, the system copies it from the external device into the internal memory. The processor executes the program instructions.
- The fundamental unit of computer storage is a bit; it could be ON (1) or OFF (0). A group of nine related bits makes a byte, out of which eight bits are used for data and the last one is used for parity. According to the rule of parity, the number of bits that are ON (1) in each byte should always be odd.
- 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 been caused due to hardware fault or electrical disturbance.

Data Representation Basics

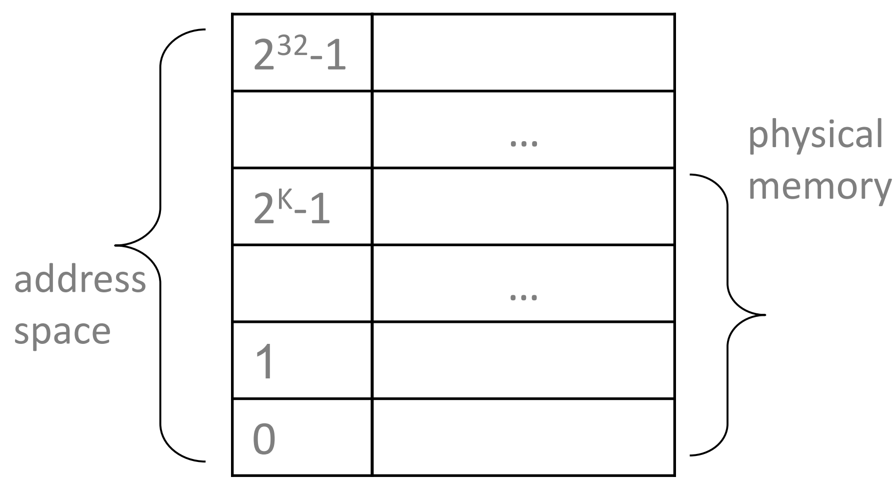
- **Bit** (Binary digit) – basic information unit: (1/0)
- **Byte** – sequence of 8 bits:



- **Word** – a sequence of bits addressed as a **single entity** by the computer

- **Main Memory** is an **array of bytes**, addressed by 0 to 2^{32}-1=0xFFFFFFFF (or to 2^{64}-1 in IA64)

2^{32} bytes = $4 \cdot 2^{10 \cdot 3}$ bytes = 4 GBs



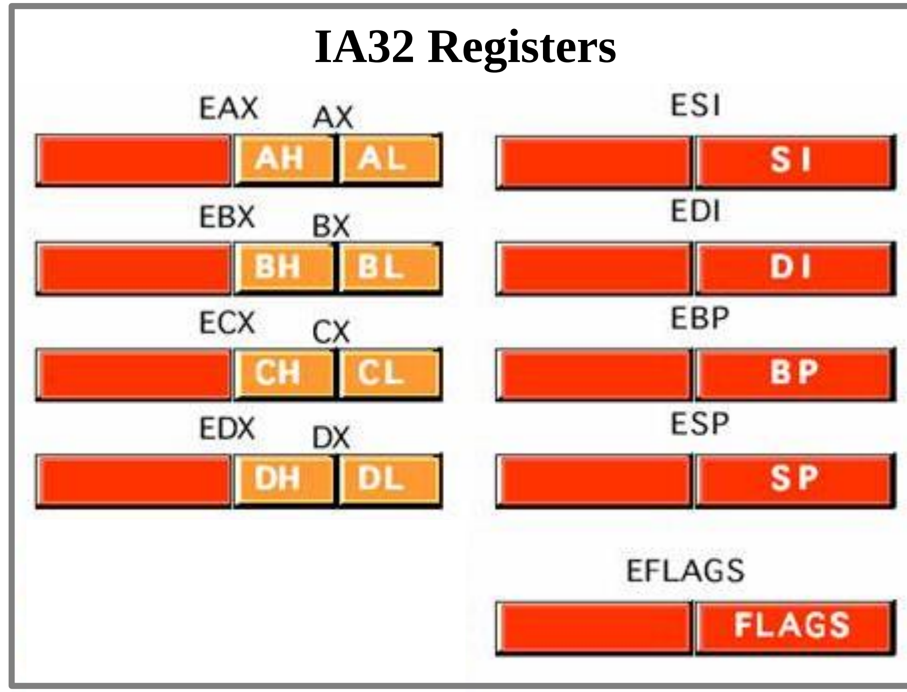
Registers in IA32

- *general purpose registers*
  EAX, EBX, ECX, EDX
  (Accumulator, Base, Counter, Data)

- *index registers*
  ESP, EBP, ESI, EDI
  (Stack pointer - contains the address of last used dword in the stack, Base pointer, Source index, Destination Index)

- *segment registers*
  CS, DS, ES, SS, *FS, GS*

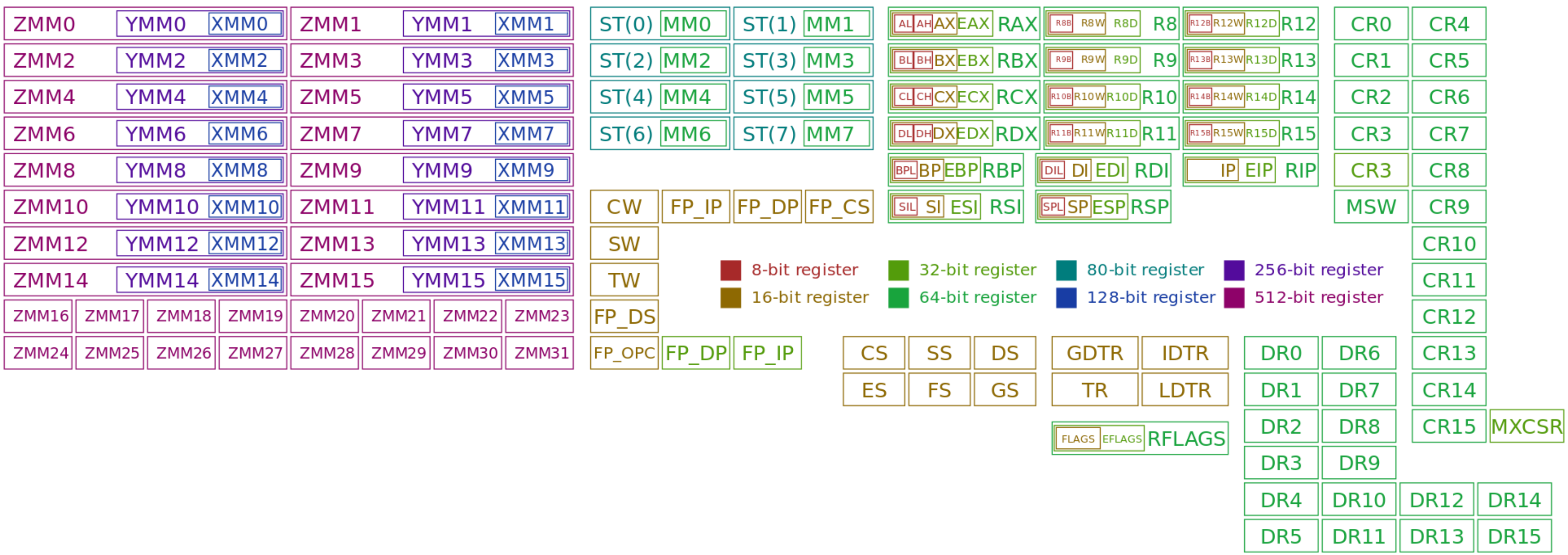
- *flag register / status register*
  EFLAGS



- *Instruction Pointer / Program Counter* EIP / EPC

- contains address (offset) of the next instruction that is going to be executed (**at run time**)
- changed by unconditional jump, conditional jump, procedure call, and return instructions

Register in all ISA



Assembly Language Program

- **Assembly Program** consist of **statement**, one per line; translated by **Assembler** into MLIs that can be loaded into memory and executed.
- **Each statement** is either an **instruction**, which the assembler translate into machine code, or assembler **directive**, which instructs the assembler to perform some spesific task, such as allocating memory space for a variable or creating a procedure.
- Both instructions and directives have up to four fields:

name operation operand(s) comment

(At least one blank or tab character must separate the fields)

Instruction and Directive

- **An example of an instruction is**

  START: MOV CX,5 ; initialize counter

  - **The name field consists of the label START:**
  - **The operation is MOV, the operands are CX and 5**
  - **And the comment is ; initialize counter**

- **An example of an assembler directive is**

  MAIN PROC

  - **MAIN is the name, and the operation field contains PROC**
  - **This particular directive creates a procedure called MAIN**

Program Statement Structure

name operation operand(s) comment

- **A Name field identifies a label, variable, or symbol. It may contain any of the following characters :**

A, B,.. , Z, a, b,.. , z, 0, 1,.. , 9, ?, _, @, $, .

- **Names are case sensitive.** (MOV or mov is same).
- **The first character may not be a digit. The period (.) may be used only as the first character.**

***A Keyword, or reserved word*, always have some predefined meaning to the assembler.**
It may be an instruction (MOV, ADD), or an assembler directive (PROC, TITLE, END)

Program Statement Structure

name operation operand(s) comment

- **Operation** field is a predefined or reserved word
  - mnemonic - symbolic **operation code**.
    - The assembler translates a symbolic opcode into a **machine language opcode**.
    - Opcode symbols often discribe the operation's function; for example, MOV, ADD, SUB
  - assemler **directive** - pseudo-operation code.
    - In an assembler directive, the operation field contains a pseudo-operation code (**pseudo-op**)
    - Pseudo-op are not translated into machine code; for example the PROC pseudo-op is used to create a procedure

Program Statement Structure

name operation operand(s) comment

- An **operand** field specifies the data that are to be acted on by the operation.
- An instruction may have zero, one, or two operands. For example:
  - **NOP** No operands; does nothing
  - **INC AX** one operand; adds 1 to the contents of **AX**
  - **ADD WORD1,2** two operands; adds 2 to **WORD1**

Program Statement Structure

name operation operand(s) comment

- The **comment** field is used by the programmer to say something about what the statement does.

- A semicolon marks the beginning of this field, and the assembler ignores anything typed after semicolon.

- Comments are optional, but because assembly language is low level, it is almost impossible to understand an assembly language program without comments.

Opcodes and Operands

- AL instructions built from 2 pieces:
  - ❑ **Opcode : What to do with the data**
  - ❑ **Operands : Where to get the data**

  <u>Ex</u>: In instruction 'add R1, R3, 3', opcode is 'add' and operand is 'R1, R3, 3'

  assembly code: MOV AL, 61h ; *load AL with 97d*

  binary code: 10110000 01100001

1011	a binary code (opcode) of instruction 'MOV'
0	specifies if data is byte (‘0’) or full size 16/32 bits (‘1’)
000	a binary identifier for a register 'AL'
01100001	a binary representation of 97d (=61h)

- Types of Opcodes:
  - ❑ **Arithmetic, logical** (add, sub, mult, and, or, cmp,..)
  - ❑ **Memory load /store** (ld, st)
  - ❑ **Control transfer** (jmp, bne)
  - ❑ **Complex** (movs, ..)

Types of Assembly Languages

- AL closely tied to processor architecture, so there are many AL.
- At least four main types:
  - ❑ **CISC: Complex Instruction-Set Computer**
  - ❑ **RISC: Reduced Instruction-Set Computer**
  - ❑ **DSP: Digital Signal Processor**
  - ❑ **VLIW: Very Long Instruction Word**
- When programing, we can use one of two forms:
  - ❑ **Inline Assembly:** some compilers allows AL instructions to be embedded within a program, such as C, C++, Pascal, Ada,... We don't need any Assembler to translate AL instructions.
  - ❑ Full **Assembly:** We must use an Assembler to translate AL instructions into MLIs (binary code). TASM (Turbo Assembler) , MASM (Macro..), NASM (Netwide..),.. - are assemblers for x86 architecture

CISC Assembly Language

- Developed when people wrote assembly language
- Complicated, often specialized instructions with many effects
- Examples from x86 architecture: *String move*, *Procedure enter, leave*
- Many, complicated addressing modes
- So complicated, often executed by a little program (microcode)
- Examples: Intel x86, 68000, PDP-11

RISC Assembly Language

- Response to growing use of compilers
- Easier-to-target, uniform instruction sets “Make the most common operations as fast as possible”
- Load-store architecture:
  - ❑ **Arithmetic only performed on registers**
  - ❑ **Memory load/store instructions for memory-register transfers**
- Designed to be pipelined
- Examples: SPARC, MIPS, HP-PA, PowerPC

DSP Assembly Language

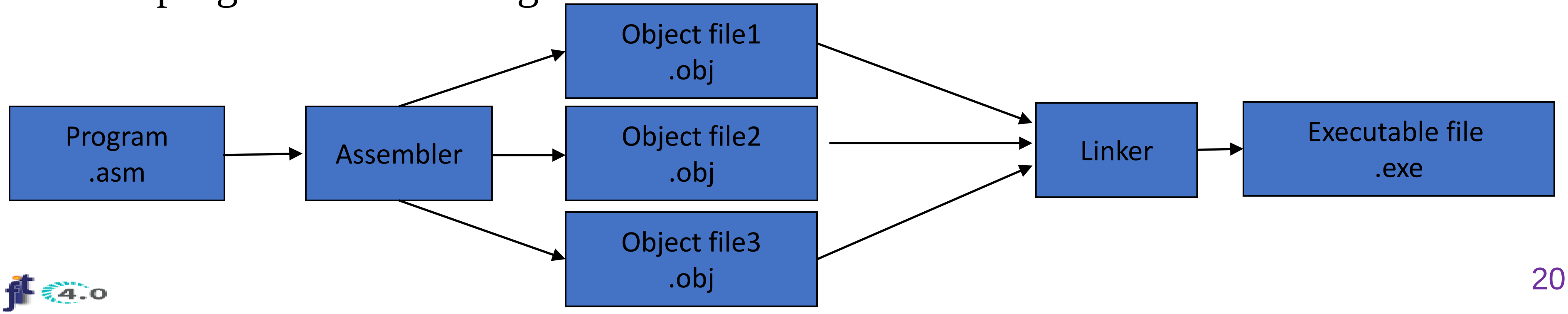
- Digital signal processors designed specifically for signal processing algorithms
- Lots of regular arithmetic on vectors
- Often written by hand
- Irregular architectures to save power, area
- Substantial instruction-level parallelism
- Examples: TI 320, Motorola 56000, Analog Devices
- Digital Signal Processor Apps
  - ❑ **Low-cost embedded systems : Modems, cellular telephones, disk drives, printers**
  - ❑ **High-throughput applications : Halftoning, base stations, 3-D sonar, tomography**
  - ❑ **PC based multimedia : Compression/decompression of audio, graphics, video**

VLIW Assembly Language

- Response to growing desire for instruction-level parallelism
- Using more transistors cheaper than running them faster
- Many parallel ALUs
- Objective: keep them all busy all the time
- Heavily pipelined
- More regular instruction set
- Very difficult to program by hand
- Looks like parallel RISC instructions
- Examples: Itanium, TI 320C6000 56000, Analog Devices

Modular Programming

- Many programs are too large to be developed by one person. This means that programs are routinely developed by a teams.
- The **Assembler** converts a **source module** (file) into a **object file**.
- The **Linker** program is used so that modules can be linked together into a complete program. It reads the object files that are created by the assembler and links them together into a single execution file.
- An **execution file** is created with the file name extension EXE.
  - ❑ In DOS /Windows: If a file is short enough (less than 64K bytes long), it can be converted from an execution file to a command file (.COM).
  - ❑ The command file is slightly different from an execution file in that the program must be originated at location 0100H before it can execute.



Ascii table

32 20 (spc)	33 21 !	34 22 "	35 23 #	36 24 $	37 25 %	38 26 &	39 27 '
40 28 (	41 29)	42 2A *	43 2B +	44 2C ,	45 2D -	46 2E .	47 2F /
48 30 0	49 31 1	50 32 2	51 33 3	52 34 4	53 35 5	54 36 6	55 37 7
56 38 8	57 39 9	58 3A :	59 3B ;	60 3C <	61 3D =	62 3E >	63 3F ?
64 40 @	65 41 A	66 42 B	67 43 C	68 44 D	69 45 E	70 46 F	71 47 G
72 48 H	73 49 I	74 4A J	75 4B K	76 4C L	77 4D M	78 4E N	79 4F O
80 50 P	81 51 Q	82 52 R	83 53 S	84 54 T	85 55 U	86 56 V	87 57 W
88 58 X	89 59 Y	90 5A Z	91 5B [	92 5C \	93 5D]	94 5E ^	95 5F _
96 60 `	97 61 a	98 62 b	99 63 c	100 64 d	101 65 e	102 66 f	103 67 g
104 68 h	105 69 i	106 6A j	107 6B k	108 6C l	109 6D m	110 6E n	111 6F o
112 70 p	113 71 q	114 72 r	115 73 s	116 74 t	117 75 u	118 76 v	119 77 w
120 78 x	121 79 y	122 7A z	123 7B {	124 7C |	125 7D }	126 7E ~	127 7F □

	0	1	2	3	4	5	6	7
0	NUL	DLE	space	0	@	P	`	p
1	SOH	DC1 XON	!	1	A	Q	a	q
2	STX	DC2	"	2	B	R	b	r
3	ETX	DC3 XOFF	#	3	C	S	c	s
4	EOT	DC4	$	4	D	T	d	t
5	ENQ	NAK	%	5	E	U	e	u
6	ACK	SYN	&	6	F	V	f	v
7	BEL	ETB	'	7	G	W	g	w
8	BS	CAN	(	8	H	X	h	x
9	HT	EM	)	9	I	Y	i	y
A	LF	SUB	*	:	J	Z	j	z
B	VT	ESC	+	;	K	[	k	{
C	FF	FS	,	<	L	\	l	|
D	CR	GS	-	=	M	]	m	}
E	SO	RS	.	>	N	^	n	~
F	SI	US	/	?	O	_	o	del

= Decimal numbers

= Hexidecimal numbers