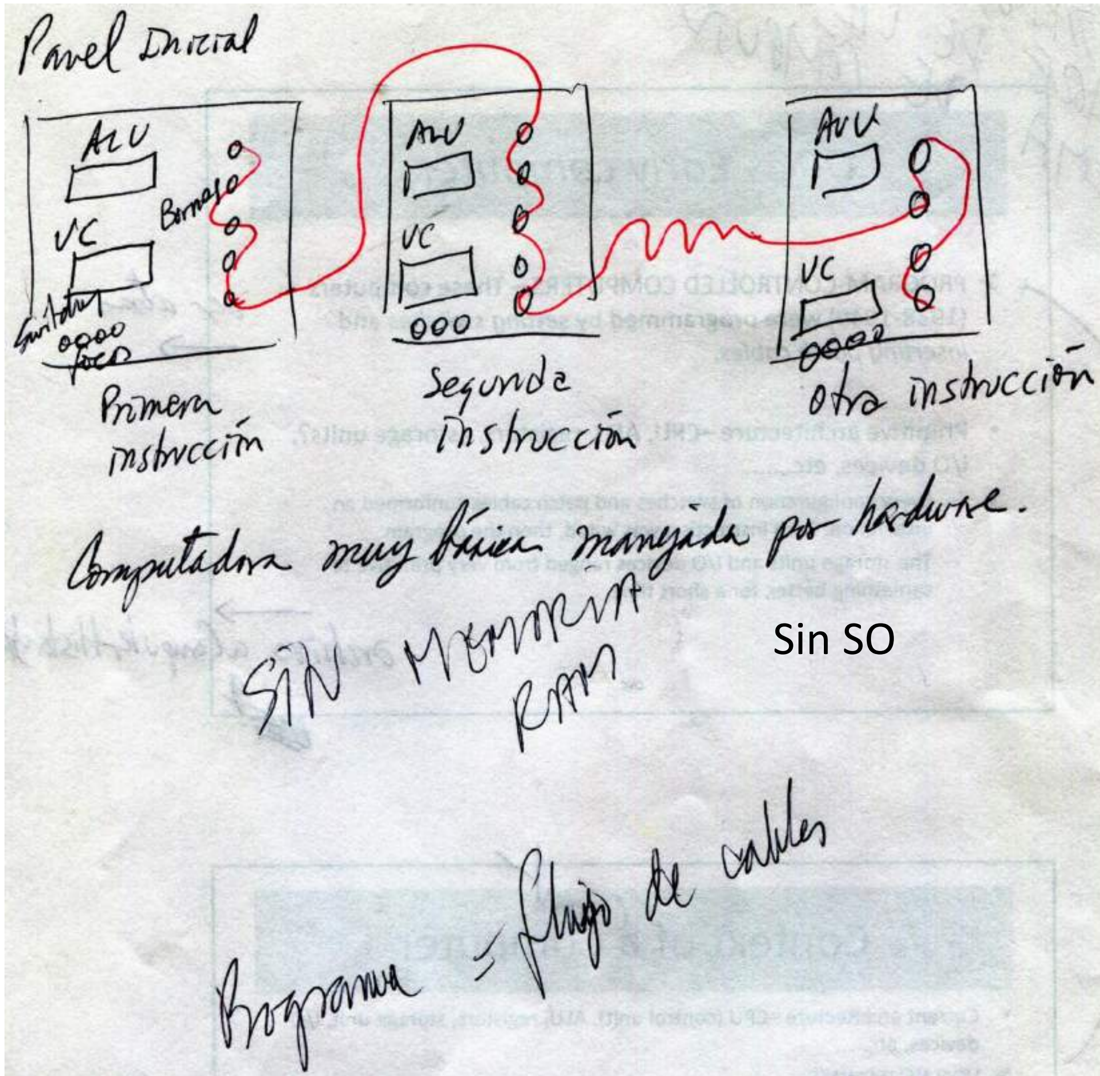
OPC - Early computers

- PROGRAM-CONTROLLED COMPUTERS – These computers (1938-1949) were programmed by *setting switches* and *inserting patch cables*.

- Primitive architecture –CPU, ALU, registers, I/O devices, storage units?, etc.,….
  - Every configuration of switches and patch cables conformed an instruction. Each instruction was wired, then the program.
  - The electric storage units and I/O devices ranged from very primitive to something better, for a short time.
  - No OS.

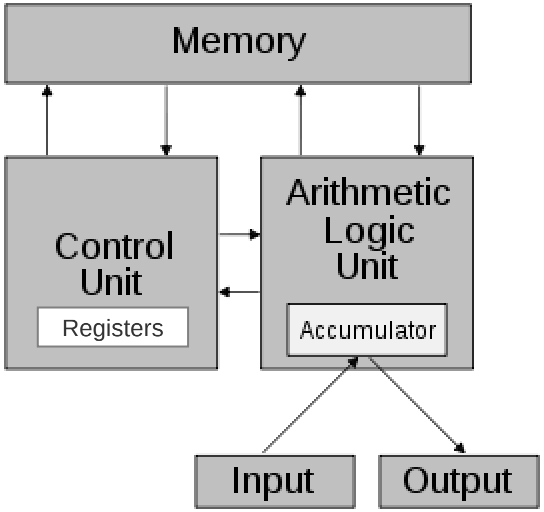
PROGRAM-CONTROLLED COMPUTERS



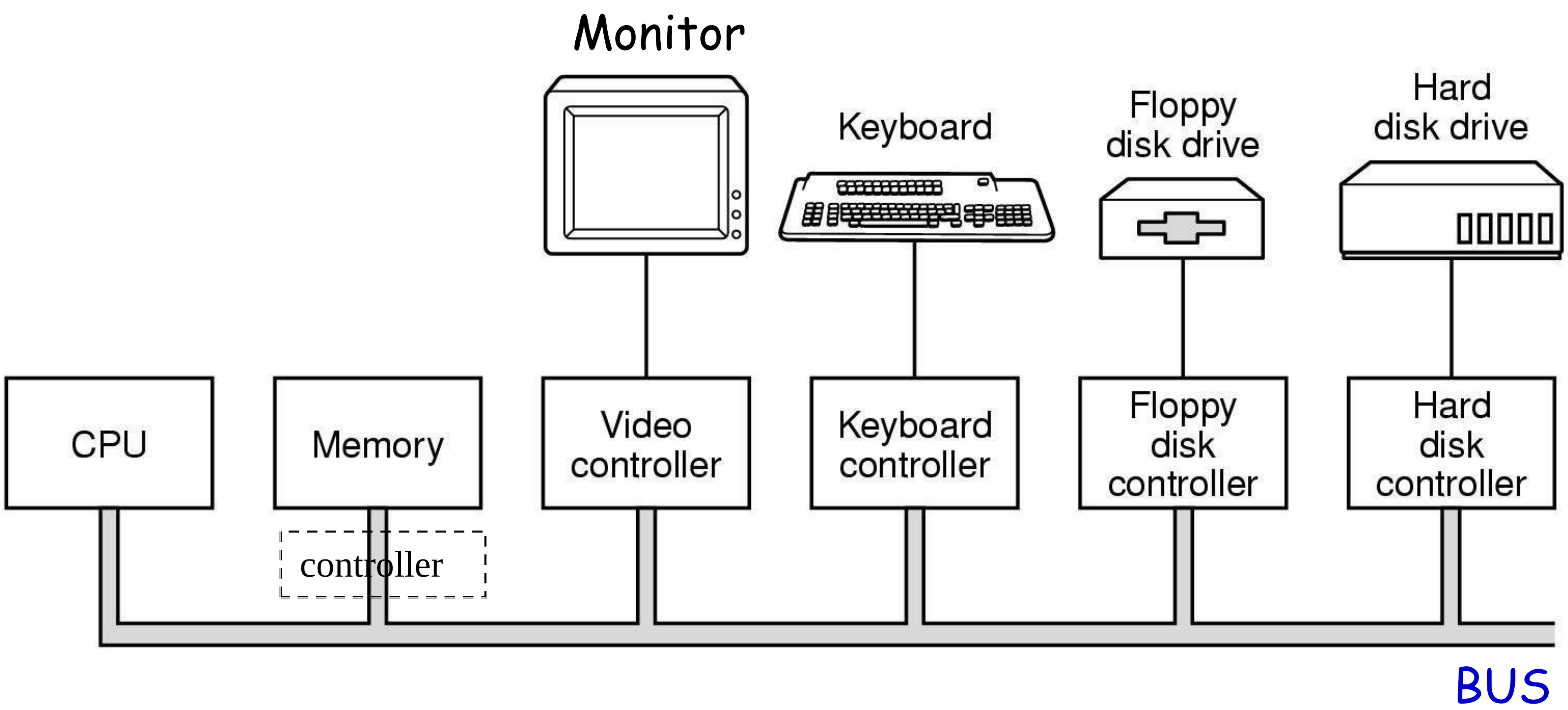
Context of a current Computer

- Current architecture –CPU (control unit), ALU, registers, storage unit, I/O devices, etc.,….

➢ VON NEUMANN'S –
  - The addition of a *stored-program* in a single separate *memory structure* that keeps both *instructions* and *data*.
  - The computers that follow the "von Neumann architecture" are also known as the "stored-program computer“.

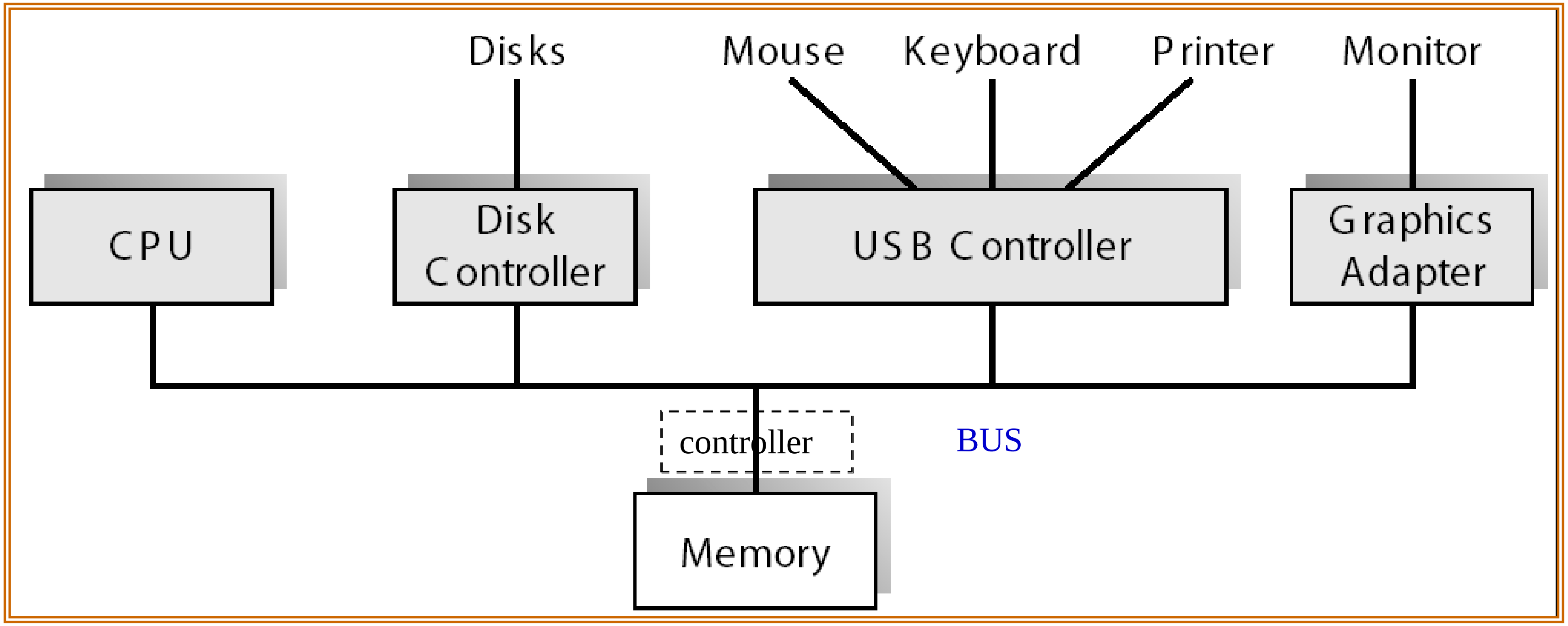


Hardware moderno con Bus



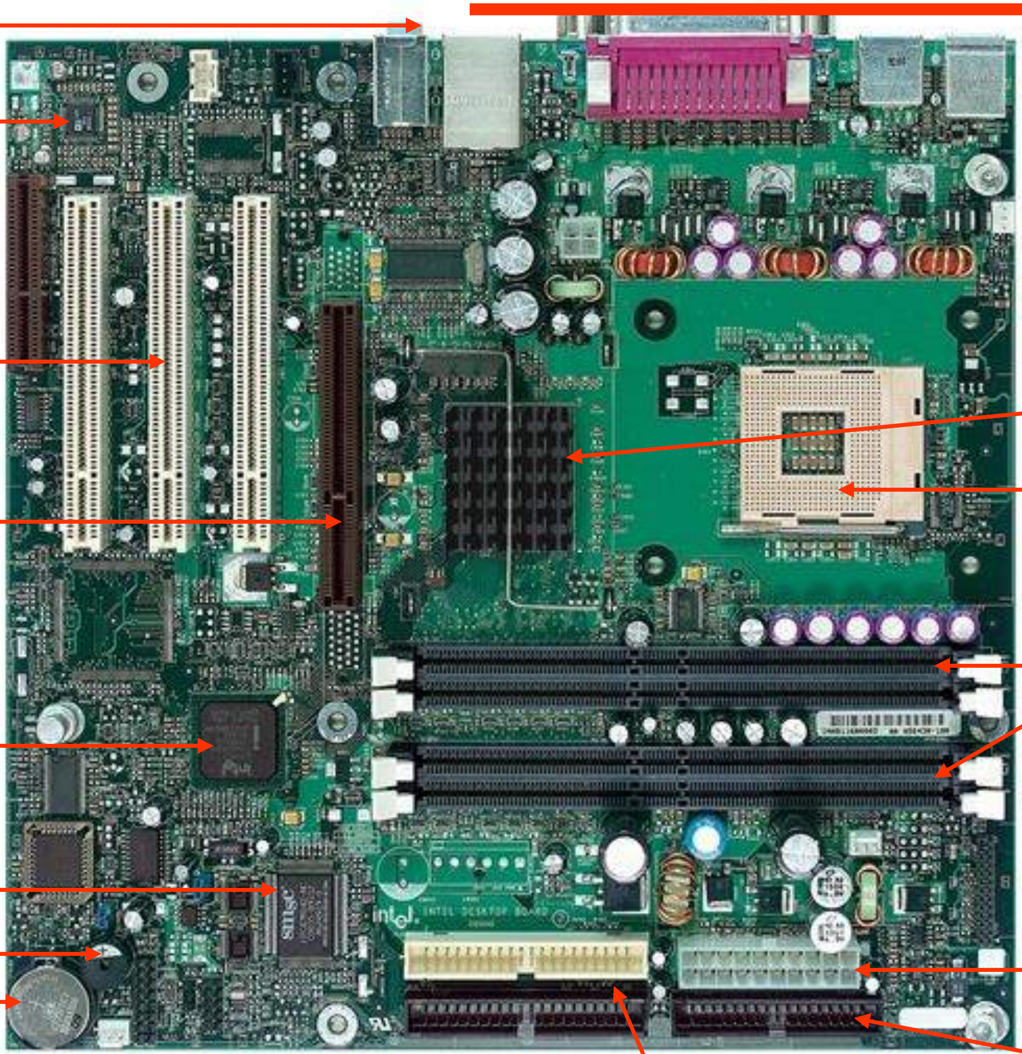
- Componentes de una computadora básica
  - Exceptuando el CPU, cada uno de los componentes tiene un controlador
  - Cada controlador posee: registros y un buffer, para realizar las entradas y / o las salidas.
  - El bus tiene tres canales: datos, direcciones de memoria y señales de control.

A Desktop Computer with USB

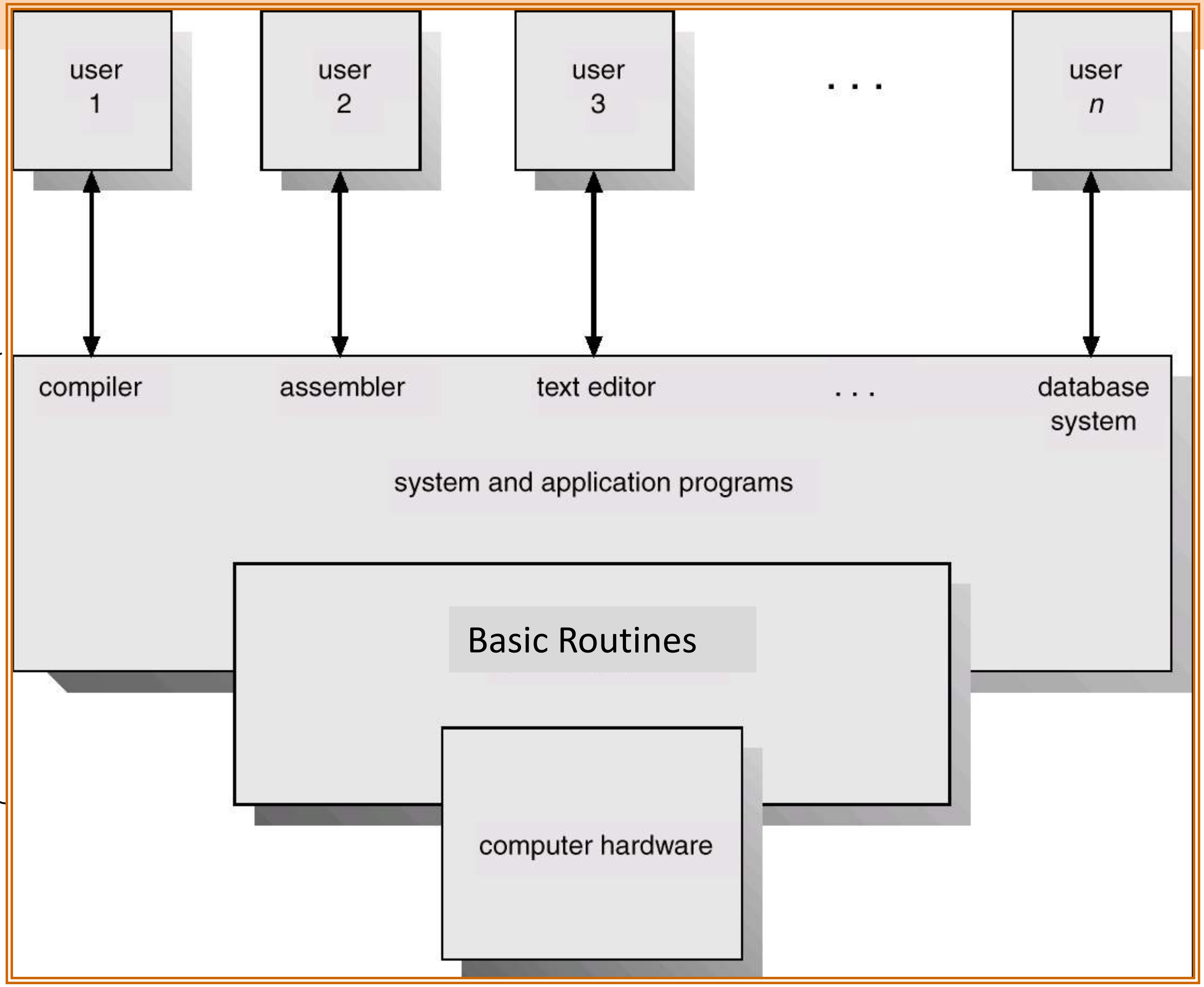


- Controlador
  - Permite hacer intercambio de información entre los distintos dispositivos.
  - El CPU se comunica con los controladores.
  - Driver: subrutina (software) del Kernel que maneja al controlador de un dispositivo.

Intel D850MD Motherboard



Computer System Components (layers)



Software

O.S.

Command Interpreter?

Kernel

Instructions?

Computer System Components

4. *Users* (people, machines, other computers) captcha: distinguishes human from machine input

3. *System and Applications programs* : *System programs* – help the users to develop applications (compilers, assemblers, database systems, line text editors, etc.), Command Interpreter. *Applications programs* – help the users to solve their computing problems (spreadsheets, web explorers, video games, business programs, word processors, function libraries). -Software-.

2. *Basic Routines* – controls and coordinates the use of the hardware among the various application programs for the various users. For these tasks contains programs and subprograms (kernel) -Software-.

1. *Hardware* – provides basic computing resources (CPU, memory, I/O devices)

Computer Hardware

- Which ones are the main functions / services of the Computer Hardware?
  - ____________________
  - ____________________
  - ____________________

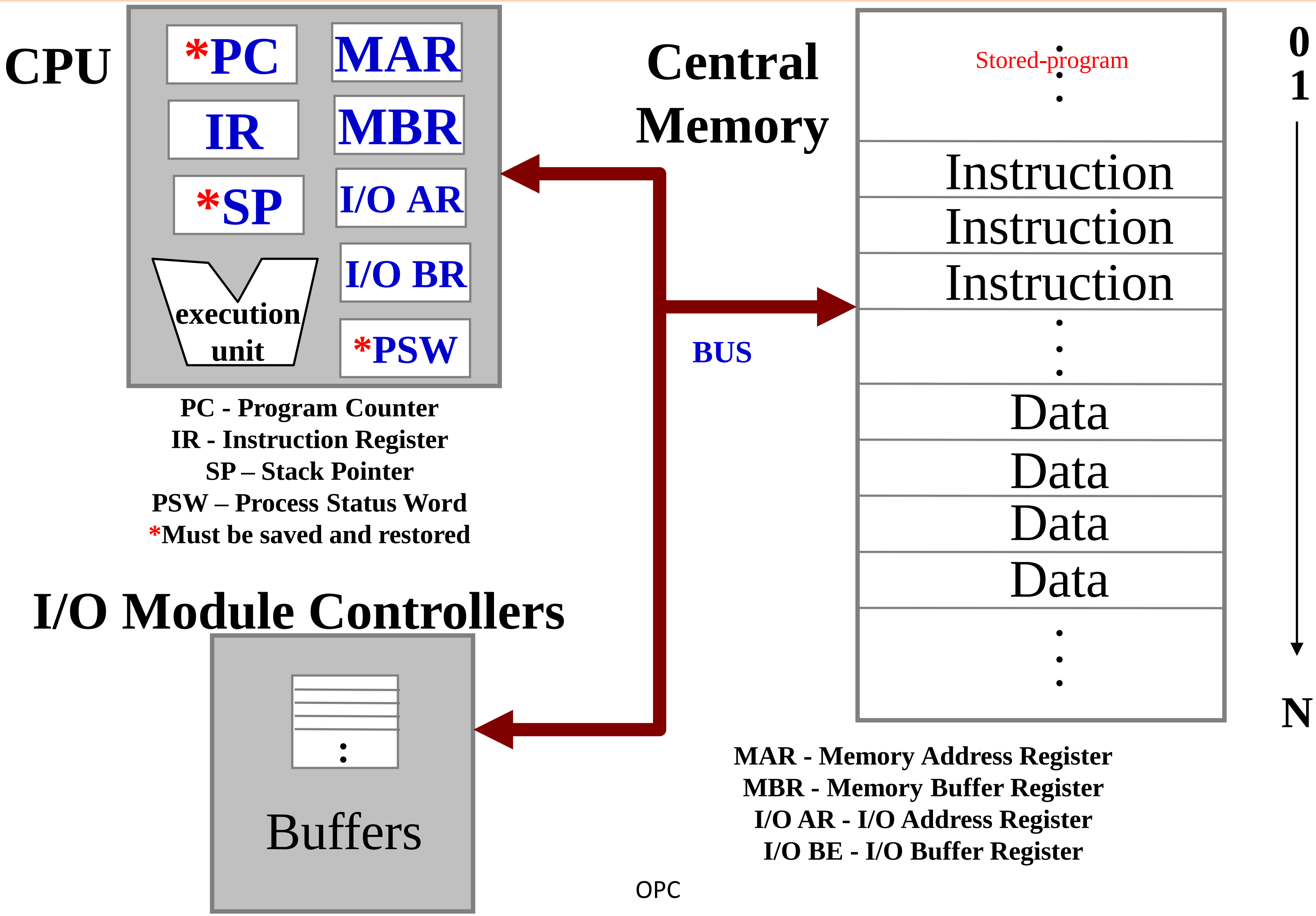
Referencias

- Chapters: Tanenbaum, Operating Systems.
- Chapters: Irvine, Kip R. Assembly Language for x86 Processors.
- Notas de Ramón Ríos.
- Ago-Dic 2023

OPC - Basic Concepts: Overview

- Computer Structure - *hardware*
- Assembly Language

General Computer Structure



PC - Program Counter
IR - Instruction Register
SP – Stack Pointer
PSW – Process Status Word
*Must be saved and restored

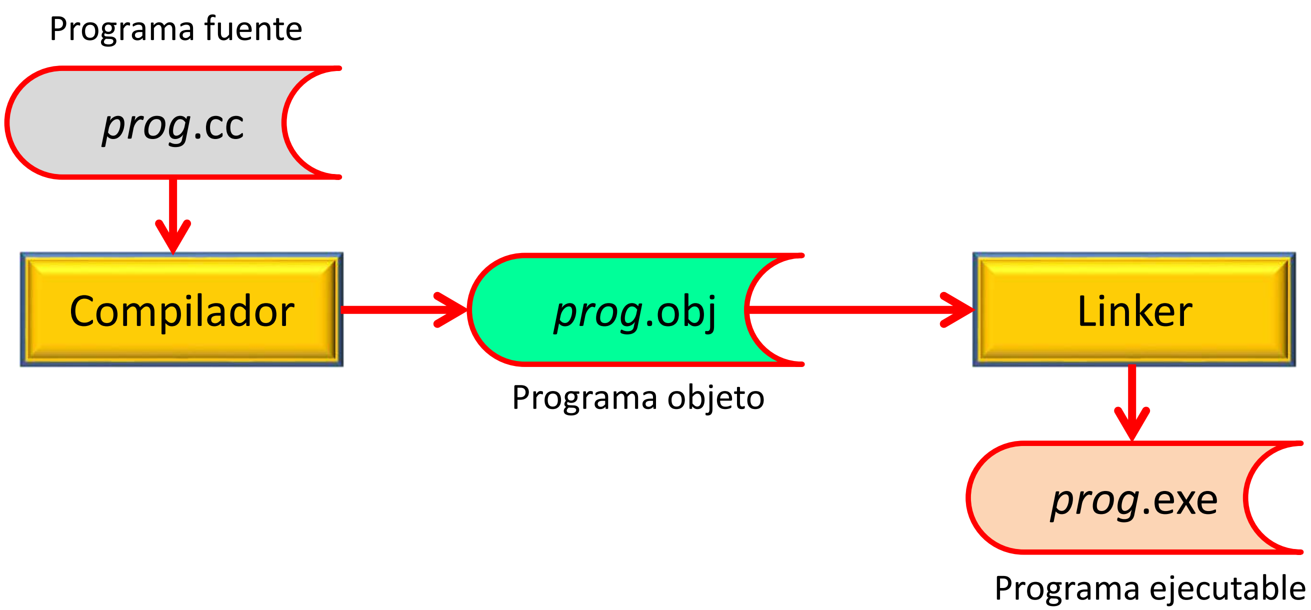
MAR - Memory Address Register
MBR - Memory Buffer Register
I/O AR - I/O Address Register
I/O BE - I/O Buffer Register

Central Memory

- Conformed by RAM
- Ordered collection of bytes (like a *vector*)
- Memory address as index of a byte
- Each byte of the memory has a memory address for access

Ejecución de Programas en Leng. Alto Nivel

- Flujo para ejecutar programas en C / C++:
  - *Compilar* con el **Compilador**, *prog.cc* a *prog.obj*
  - *Ligado o vinculado* (linking) con el **Linker**, *prog.obj* a *prog.exe*
  - Ejecución de programa *prog.exe*



Assembly Language (AL)

- Assembly Language?

Translating Languages

English: Display the sum of A times B plus C.

C++: F = A * B + C;

cout << (F);

Assembly Language:

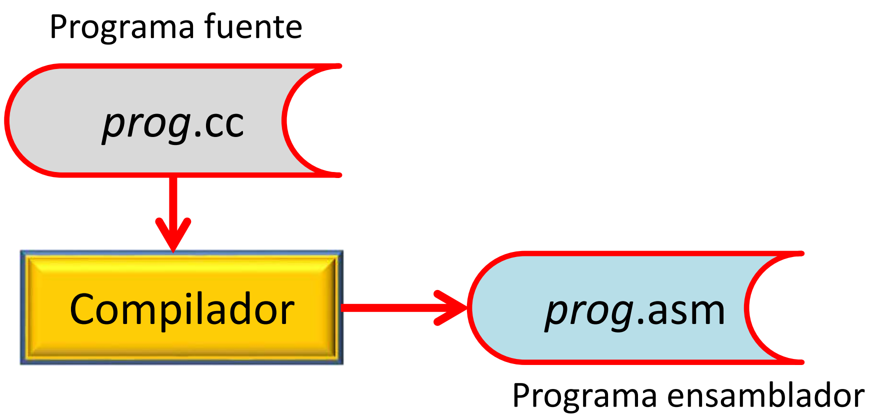
mov EAX, A
mul B
add EAX, C
mov F, EAX
call WriteInt

Intel Machine Language:

A1 00000000
F7 25 00000004
03 05 00000008
- - - -
E8 00500000

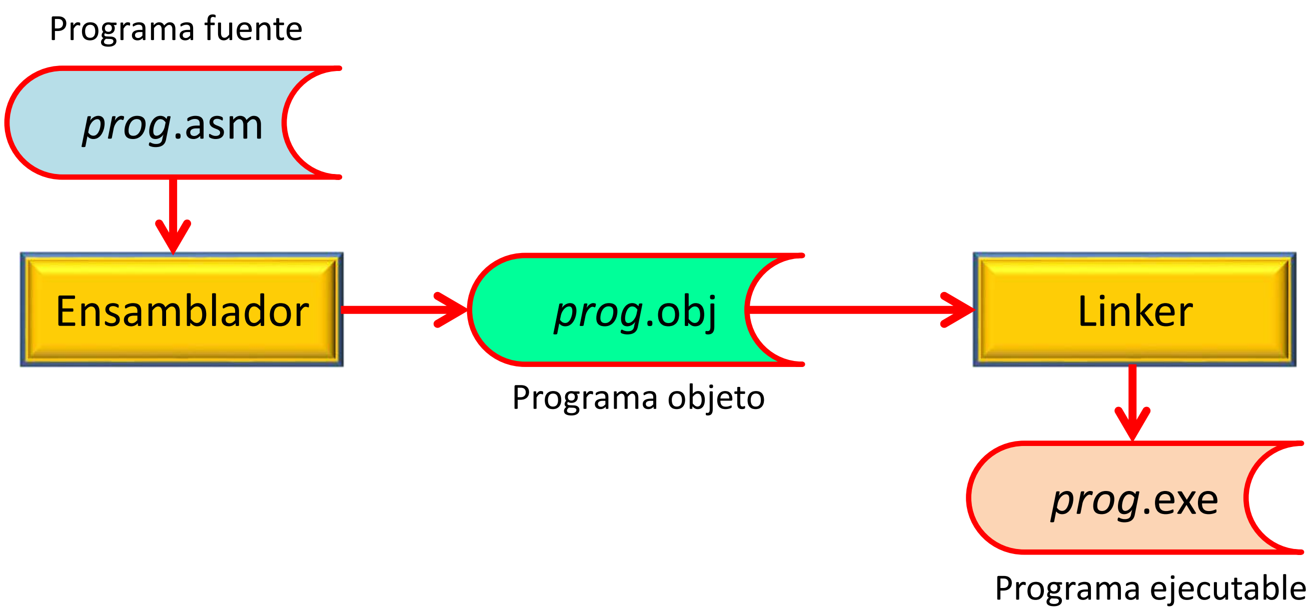
Compilación de Alto Nivel a Ensamblador

- Flujo para producir programas Ensamblador desde C / C++:
  - *Compilar* con el **Compilador**, con opción ensamblador,
  - de *prog.cc* a *prog.asm*



Ejecución de Programas en Leng. Ensamblador

- Flujo para ejecutar programas en Ensamblador:
  - *Ensamble* con el **Ensamblador**, *prog.asm* a *prog.obj*
  - *Ligado o vinculado* (linking) con el **Linker**, *prog.obj* a *prog.exe*
  - Ejecución de programa *prog.exe*



Assembly Language (AL)

- What is an Assembler (Ensamblador)?
- What background should I have?
- How does Assembly Language (AL) relate to machine language?
- How do C++ and Java relate to AL?
- Is AL portable?

Referencias

- Chapters: Tanenbaum, A. S.; Operating Systems.
- Chapters: Irvine, Kip R.; Assembly Language for x86 Processors.
- Notas: Ramón Ríos.
- Agosto – diciembre 2023

OPC - Basic Concepts: Overview

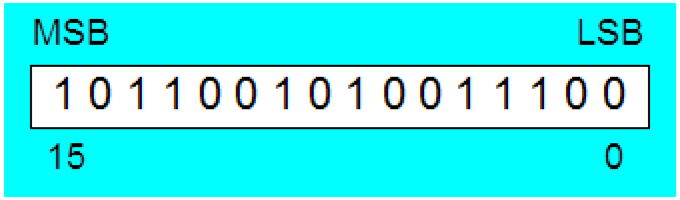
- Data Representation

Assembly Data Representation

- Binary Numbers
- Binary Addition
- Integers
- Storage Sizes
- Unsigned Integers
- Hexadecimal Integers

Binary Numbers

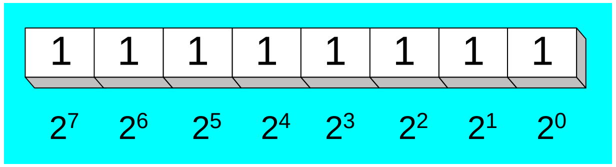
- Binary Digits are 1 and 0
  - 1 = true
  - 0 = false
- MSB – most significant bit
- LSB – least significant bit
- Bit numbering:



Unsigned Binary Numbers

Each digit (bit) is either 1 or 0

Each bit represents a power of 2:



Every binary number is a sum of powers of 2

Table 1-3 Binary Bit Position Values.

2^n	Decimal Value	2^n	Decimal Value
2^0	1	2^8	256
2^1	2	2^9	512
2^2	4	2^{10}	1024
2^3	8	2^{11}	2048
2^4	16	2^{12}	4096
2^5	32	2^{13}	8192
2^6	64	2^{14}	16384
2^7	128	2^{15}	32768

Translating Binary to Decimal

Weighted positional notation shows how to calculate the *decimal value* of each binary bit:

$$\mathbf{dec} = (B_{n-1} \times 2^{n-1}) + (B_{n-2} \times 2^{n-2}) + \ldots + (B_1 \times 2^1) + (B_0 \times 2^0)$$

B = binary digit

binary 00001001 = decimal ??:

$$(0_2 \times 2^7) + (0_2 \times 2^6) + (0_2 \times 2^5) + (0_2 \times 2^4) +$$

$$(1_2 \times 2^3) + (0_2 \times 2^2) + (0_2 \times 2^1) + (1_2 \times 2^0) = 9_{10}$$

Translating *Unsigned Decimal* to *Binary*

Repeatedly divide the decimal *integer by 2*. Each remainder is a binary digit in the translated value:

Division	Quotient	Remainder	
37 / 2	18	1	B_0
18 / 2	9	0	B_1
9 / 2	4	1	B_2
4 / 2	2	0	B_3
2 / 2	1	0	B_4
1 / 2	0	1	B_5

$$37_{10} = 100101_2$$

Binary Addition

Starting with the LSB, add each pair of digits, include the *carry* if present.

carry:					1				
	0	0	0	0	0	1	0	0	(4)
+	0	0	0	0	0	1	1	1	(7)
	0	0	0	0	1	0	1	1	(11)
bit position:	7	6	5	4	3	2	1	0	

Unsigned Integers & Storage Sizes

Standard sizes:

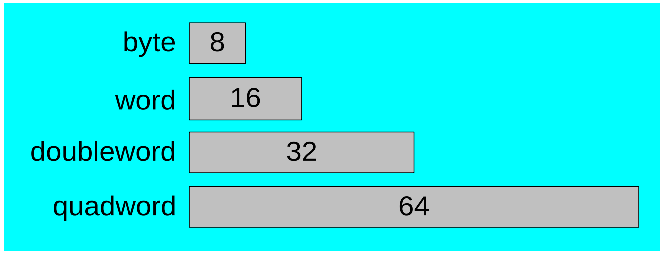


Table 1-4 Ranges of Unsigned Integers.

Storage Type	Range (low–high)	Powers of 2
Unsigned byte	0 to 255	0 to $(2^8 - 1)$
Unsigned word	0 to 65,535	0 to $(2^{16} - 1)$
Unsigned doubleword	0 to 4,294,967,295	0 to $(2^{32} - 1)$
Unsigned quadword	0 to 18,446,744,073,709,551,615	0 to $(2^{64} - 1)$

What is the largest unsigned integer that may be stored in 20 bits?

Hexadecimal Integers

Binary values are represented in hexadecimal.

Table 1-5 Binary, Decimal, and Hexadecimal Equivalents.

Binary	Decimal	Hexadecimal	Binary	Decimal	Hexadecimal
0000	0	0	1000	8	8
0001	1	1	1001	9	9
0010	2	2	1010	10	A
0011	3	3	1011	11	B
0100	4	4	1100	12	C
0101	5	5	1101	13	D
0110	6	6	1110	14	E
0111	7	7	1111	15	F

Translating Binary to Hexadecimal

- Each hexadecimal digit corresponds to 4 binary bits.
- Example: Translate the binary integer
  - $0001\ 0110\ 1010\ 0111\ 1001\ 0100_2$ to hexadecimal:

1	6	A	7	9	4
0001	0110	1010	0111	1001	0100

Powers of 16

Used when calculating hexadecimal values up to *n* digits long:

16^n	Decimal Value	16^n	Decimal Value
16^0	1	16^4	65,536
16^1	16	16^5	1,048,576
16^2	256	16^6	16,777,216
16^3	4096	16^7	268,435,456

Converting Hexadecimal to Decimal

- Multiply each digit by its corresponding power of 16:

$$dec = (H_3 \times 16^3) + (H_2 \times 16^2) + (H_1 \times 16^1) + (H_0 \times 16^0)$$

- Hex $\mathbf{1234_{16}}$ equals $(1 \times 16^3) + (2 \times 16^2) + (3 \times 16^1) + (4 \times 16^0)$, or decimal $4{,}660_{10}$.

- Hex $\mathbf{3BA4_{16}}$ equals $(3 \times 16^3) + (11 * 16^2) + (10 \times 16^1) + (4 \times 16^0)$, or decimal $15{,}268_{10}$.

Converting Decimal to Hexadecimal

Division	Quotient	Remainder	
422 / 16	26	6	H_0
26 / 16	1	A	H_1
1 / 16	0	1	H_2

decimal $422_{10} = 1A6_{16}$ hexadecimal

Hexadecimal Addition

- Divide the sum of two digits by the number base (16). The quotient becomes the carry value, and the remainder is the sum digit.

		1	1
36	28	28	6A
42	45	58	4B
78	6D	80	B5

21 / 16 = 1, rem 5

Important skill: Programmers frequently *add* and *subtract* the *addresses* of variables and instructions.

Binary Subtraction

Subtract A – B

$$\begin{array}{r} 0\,0\,0\,0\,1\,1\,0\,0 \\ -\ \underline{0\,0\,0\,0\,0\,0\,1\,1} \end{array}$$

Practice: Subtract 0101 from 1001.

Hexadecimal Subtraction

- When a borrow is required from the digit to the left, add 16 (decimal) to the current digit's value:

16 + 5 = 21

	–1
C6	75
A2	47
24	??

Practice: The address of **var1** is 00400020. The address of the next variable after var1 is 0040006A. How many bytes are used by var1?

Referencias

- Chapters: Tanenbaum, A. S.; Operating Systems.
- Chapters: Irvine, Kip R.; Assembly Language for x86 Processors.
- Notas: Ramón Ríos.
- Agosto – diciembre 2023

Ranges of Unsigned Integers

Standard sizes:

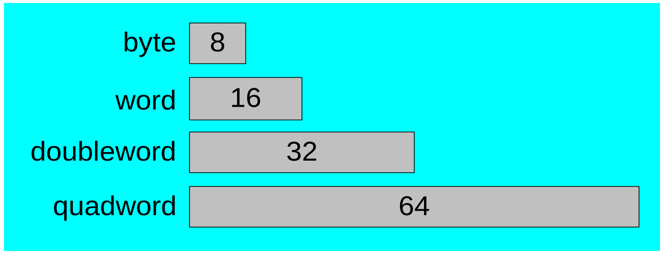
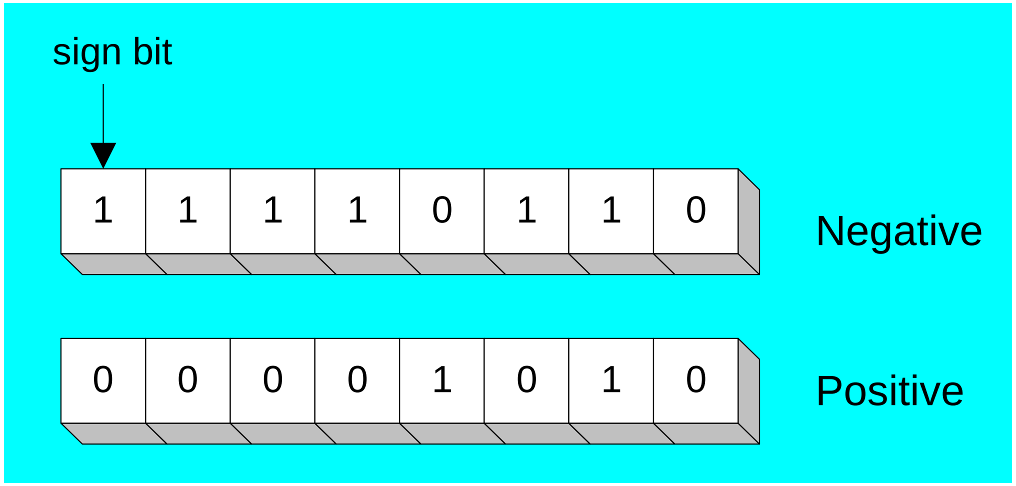


Table 1-4 Ranges of Unsigned Integers.

Storage Type	Range (low–high)	Powers of 2
Unsigned byte	0 to 255	0 to $(2^8 - 1)$
Unsigned word	0 to 65,535	0 to $(2^{16} - 1)$
Unsigned doubleword	0 to 4,294,967,295	0 to $(2^{32} - 1)$
Unsigned quadword	0 to 18,446,744,073,709,551,615	0 to $(2^{64} - 1)$

Signed Integers

The highest bit (MSB) indicates the sign. 1 = negative, 0 = positive



If the highest digit of a hexadecimal integer is > 7, the value is negative. Examples: F5, 8A, C5, A2, 9D

Ranges of Signed Integers

The highest bit is reserved for the sign. This limits the range:

Storage Type	Range (low–high)	Powers of 2
Signed byte	−128 to +127	-2^7 to $(2^7 - 1)$
Signed word	−32,768 to +32,767	-2^{15} to $(2^{15} - 1)$
Signed doubleword	−2,147,483,648 to 2,147,483,647	-2^{31} to $(2^{31} - 1)$
Signed quadword	−9,223,372,036,854,775,808 to +9,223,372,036,854,775,807	-2^{63} to $(2^{63} - 1)$

Practice: What is the largest positive value that may be stored in 20 bits?

Forming the Two's Complement

- Negative numbers are stored in Two’s Complement notation
  - 1st : do One’s Complement with the number
  - 2nd: add 1 to the One’s Complemented number
- Represents the additive Inverse

Starting value	00000001
Step 1: reverse the bits	11111110
Step 2: add 1 to the value from Step 1	11111110 +00000001
Sum: two’s complement representation	11111111

Note that 0000 0001 + 1111 1111 = 0000 0000

Binary Subtraction

When subtracting A – B, an alternative, convert B to its *Two's Complement.*

Add A to (–B)

$$
\begin{array}{r}
0\ 0\ 0\ 0\ 1\ 1\ 0\ 0 \\
-\ 0\ 0\ 0\ 0\ 0\ 0\ 1\ 1 \\
\hline
\end{array}
\longrightarrow
\begin{array}{r}
0\ 0\ 0\ 0\ 1\ 1\ 0\ 0 \\
+\ 1\ 1\ 1\ 1\ 1\ 1\ 0\ 1 \\
\hline
0\ 0\ 0\ 0\ 1\ 0\ 0\ 1
\end{array}
$$

Learn How To Do the Following:

- Converting UNSIGNED and SIGNED Integers
- Form the two's complement of a hexadecimal integer
- Convert signed binary to decimal
- Convert signed decimal to binary
- Convert signed decimal to hexadecimal
- Convert signed hexadecimal to decimal

Hex One's Complement

Hex		Bin		Bin C1s		Hex C1s
0	>	0000	C1s	1111	>	*F*
1	>	0001	C1s	1110	>	*E*
2	>	0010	C1s	1101	>	*D*
3	>	0011	C1s	1100	>	*C*
4	>	0100	C1s	1011	>	*B*
5	>	0101	C1s	1010	>	*A*
6	>	0110	C1s	1001	>	*9*
7	>	0111	C1s	1000	>	*8*
8	>	1000	C1s	0111	>	*7*
9	>	1001	C1s	0110	>	*6*
A	>	1010	C1s	0101	>	*5*
B	>	1011	C1s	0100	>	*4*
C	>	1100	C1s	0011	>	*3*
D	>	1101	C1s	0010	>	*2*
E	>	1110	C1s	0001	>	*1*
F	>	1111	C1s	0000	>	*0*

Referencias

- Chapters: Irvine, Kip R. Assembly Language for x86 Processors.
- Notas de Ramón Ríos.
- Agosto – diciembre 2023

Arithmetic Operator Precedence

- HLL operator precedence in evaluation of Arithmetic Expressions
- 1- **( )**
- 2- **-** unary
- 3- *, /
- 4- +, -
- Examples:
  - X = 6 - Y * 5 + 2;
  - R = -8 - T * (7 + M);

Character Interpretation & Storage

- Character sets
  - Standard ASCII (0 – 127), 8 bits
    - ‘u’, ‘U’, ‘A’, ‘8’, ‘2’
  - Extended ASCII (0 – 255), 8 bits
  - ANSI (0 – 255), 8 bits
  - Unicode (0 – 65,535), 16 bits in Java

Characters application

- To type down a text command in a computer
- To write down a computer program in a text file
- To send messages
- To send e-mails
- . . .

ASCII Code (7-bit)

American Standard Code for Information Interchange

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

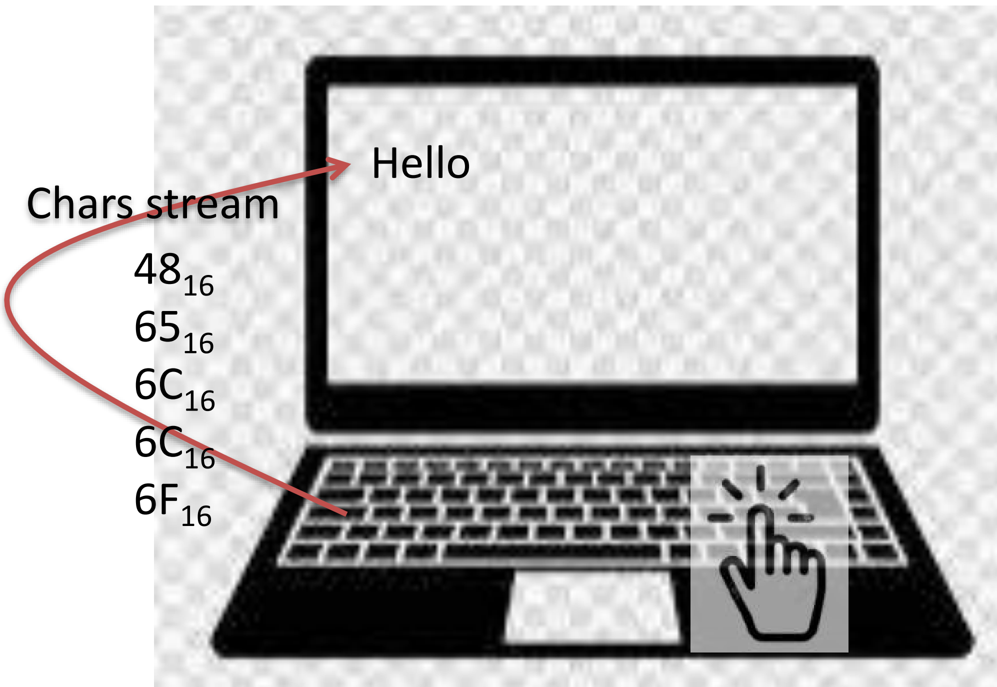
Source: www.LookupTables.com

Extended ASCII Code (8-bit)

128	Ç	144	É	160	á	176	░	192	└	208	╨	224	α	240	≡
129	ü	145	æ	161	í	177	▒	193	┴	209	╤	225	ß	241	±
130	é	146	Æ	162	ó	178	▓	194	┬	210	╥	226	Γ	242	≥
131	â	147	ô	163	ú	179	│	195	├	211	╙	227	π	243	≤
132	ä	148	ö	164	ñ	180	┤	196	─	212	╘	228	Σ	244	⌠
133	à	149	ò	165	Ñ	181	╡	197	┼	213	╒	229	σ	245	⌡
134	å	150	û	166	ª	182	╢	198	╞	214	╓	230	µ	246	÷
135	ç	151	ù	167	º	183	╖	199	╟	215	╫	231	τ	247	≈
136	ê	152	ÿ	168	¿	184	╕	200	╚	216	╪	232	Φ	248	°
137	ë	153	Ö	169	⌐	185	╣	201	╔	217	┘	233	Θ	249	∙
138	è	154	Ü	170	¬	186	║	202	╩	218	┌	234	Ω	250	·
139	ï	155	¢	171	½	187	╗	203	╦	219	█	235	δ	251	√
140	î	156	£	172	¼	188	╝	204	╠	220	▄	236	∞	252	ⁿ
141	ì	157	¥	173	¡	189	╜	205	═	221	▌	237	φ	253	²
142	Ä	158	₧	174	«	190	╛	206	╬	222	▐	238	ε	254	■
143	Å	159	ƒ	175	»	191	┐	207	╧	223	▀	239	∩	255	

Source: www.LookupTables.com

Characters use



Typing down “Hello”.

HLL Strings in central memory

- Two components
  - ➢ String
    - – Array of consecutive characters
    - – “Laura”
  - ➢ Null-terminated String
    - – A *null* (zero) character ending String
- Interpretation and Storage
  - – Central memory ????

Numeric Data Representation

- pure binary
  - can be calculated directly
- ASCII char binary
  - string of digits: "01010101"
- ASCII char decimal
  - string of digits: "65"
- ASCII char hexadecimal
  - string of digits: "9C"

Boolean Operations

- Easy to operate for ALU
- Bit-to-bit
- True:1,   False:0
- NOT
- AND
- OR
- Operator Precedence
- Truth Tables

NOT

Inverts (reverses) a boolean value

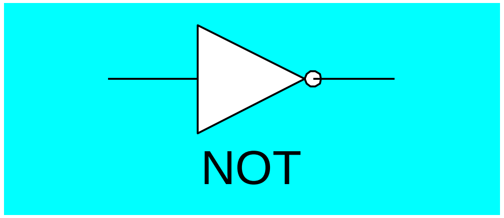
One operand

If operator is T, then F; if F, then T

Truth table for Boolean NOT operator:

X	¬X
F	T
T	F

Digital gate diagram for NOT:



AND

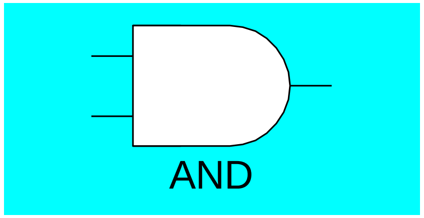
Two operands

Both must be T for T; otherwise F

Truth table for Boolean AND operator:

X	Y	$X \wedge Y$
F	F	F
F	T	F
T	F	F
T	T	T

Digital gate diagram for AND:



OR

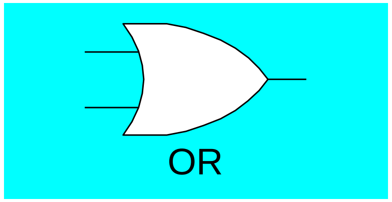
Two operands

Both must be F for F; otherwise T

Truth table for Boolean OR operator:

X	Y	X ∨ Y
F	F	F
F	T	T
T	F	T
T	T	T

Digital gate diagram for OR:



Boolean Operator Precedence

- **The HLL order of evaluation is:**

1. Parentheses ( )
2. NOT ¬
3. AND ^
4. OR v

- **Example 1:** F = A ^ (B v C) ^ (C v¬D)
- **Example 2:** F = A ^ B v C ^ C v¬D

NAND

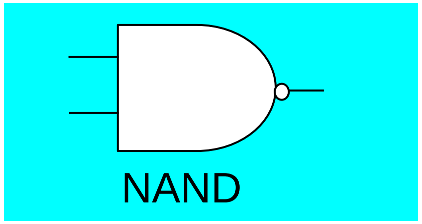
Two operands

Both T, then F; otherwise T

Truth table for Boolean NAND operator:

X	Y	X NAND Y
F	F	T
F	T	T
T	F	T
T	T	F

Digital gate diagram for NAND:



NOR

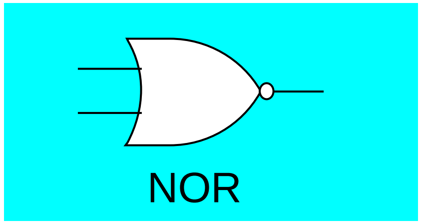
Two operands

Any T, then F; otherwise T

Truth table for Boolean NOR operator:

X	Y	X NOR Y
F	F	T
F	T	F
T	F	F
T	T	F

Digital gate diagram for NAND:



Truth Tables

- A Boolean function has one or more Boolean inputs, returns a single Boolean output.
- A truth table shows all the inputs and outputs of a Boolean function

Example: $\neg X \vee Y$

X	$\neg$X	Y	$\neg X \vee Y$
F	T	F	T
F	T	T	T
T	F	F	F
T	F	T	T

Referencias

- Chapters: Irvine, Kip R. Assembly Language for x86 Processors.
- Notas de Ramón Ríos.
- Agosto – diciembre 2023

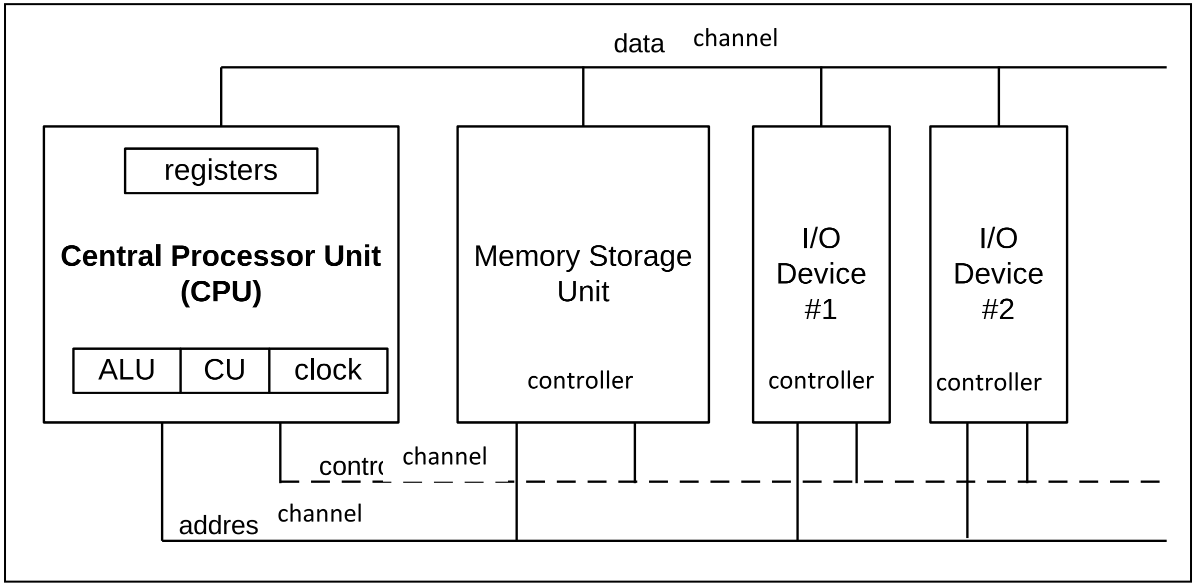
General Concepts

- The Microprocessor
- Cache Memory
- Instruction execution cycle
- Reading from Memory

The Microprocessor - 1

Also called Central Processor Unit (CPU), the controlling element in a computer system

- Control unit (CU) coordinates sequence of execution steps
  - Controls memory and I/O through connections called buses
- Arithmetic and Logic Unit (ALU) performs arithmetic and bitwise processing
- Clock synchronizes CPU operations (oscillator)



Block diagram of a microcomputer

The Microprocessor - 2

- CU performs:
  - data transfer between itself and the *memory* or *I/O systems*
  - program flow via simple decisions, PC

- ALU does:
  - simple arithmetic and logic operations
  - The resulting states of the ALU operations are placed in FLAGS (*psw*)

Registers inside the CPU

- *Registers* are hard-wired inside the CPU.

- They can storage, 8-bit, 16-bit, 32-bit, 64-bit, or 128-bit long information (data or address).

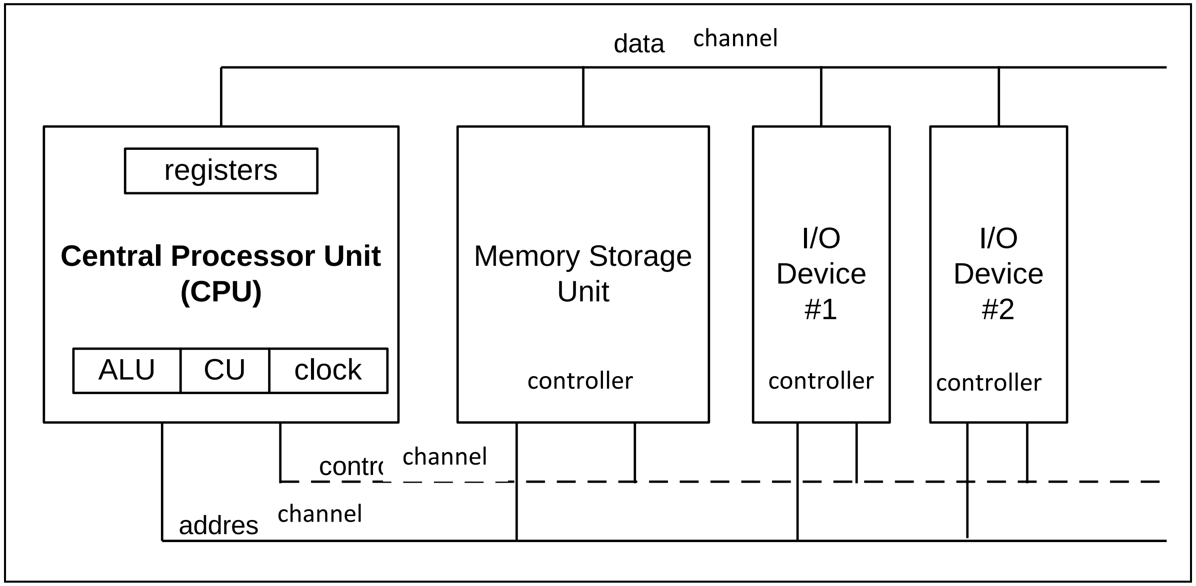
- Their speed access is very fast, and is the fastest in the computer hardware.

Memory Storage Unit

- Called Central or Main Memory
- Mostly, made out of RAM chips
- Storage, Addresses, Instructions and / or Data
- Stored programs make the microprocessor and computer system very powerful devices
- Conventional *memory* is outside the CPU, and it responds more slowly to access requests than *registers*

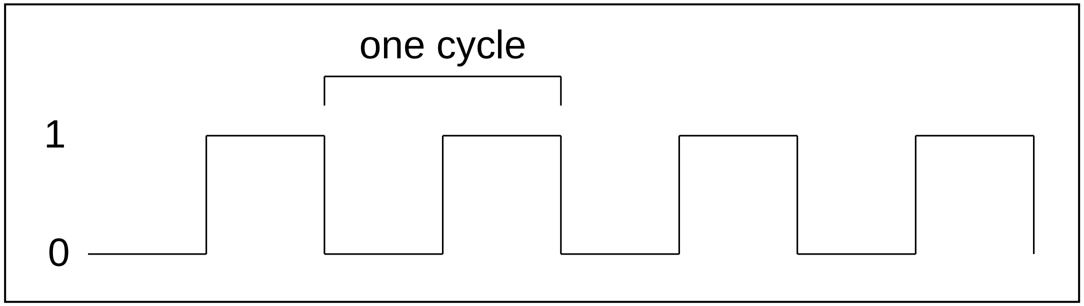
Bus

- A common group of wires that interconnect CPU, memory and I/O devices
- Transfer control information, addresses, and data between components
- **Bus**, containing three channels or sub-buses
  - **Control channel**: determines where data comes from and goes, and ALU activities
  - **Address channel**: selects where data comes from or goes to
  - **Data channel**: moves data between memory bytes, I/O and CPU registers



Clock (Oscillator)

- Synchronizes all CPU and BUS operations
- A *tick* every cycle
- Machine (clock) cycle measures time of a single operation / instruction
- Clock is used to trigger events



Computer BUS

- BUS has became a content element between all the components attached up

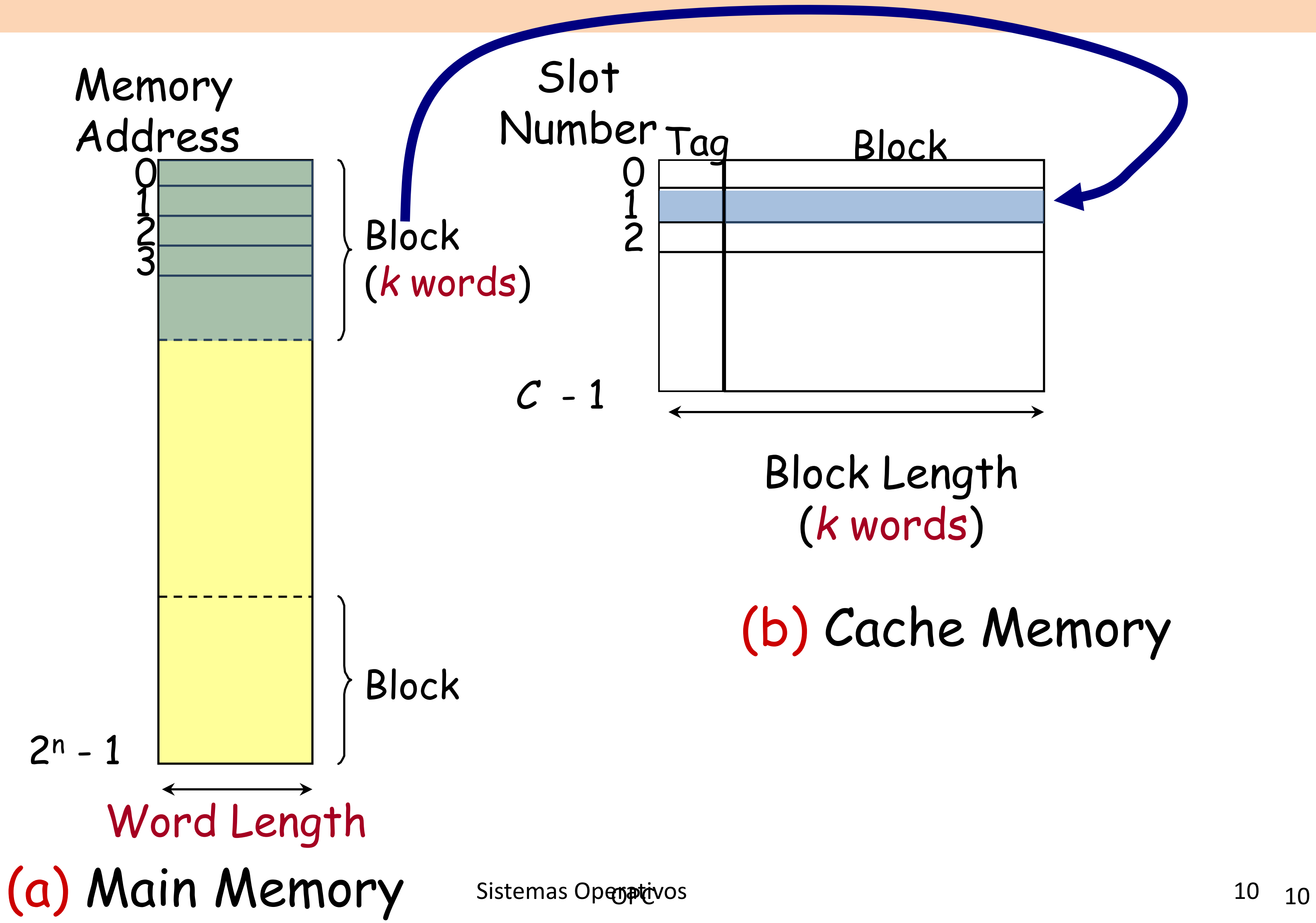
- CPU has to compete, against other devices, to seize the memory unit (main, central, or RAM memory)

- CPU needs to access the main memory to *fetch* next instruction and *execute* it up

Cache Memory

- *Closer* to the CPU
- *Access time*: faster than Main Memory (RAM)
- *Storage size*: smaller than Main Memory
- *Time to time*: a block of Main Memory locations is copied, by CPU, to Cache Memory

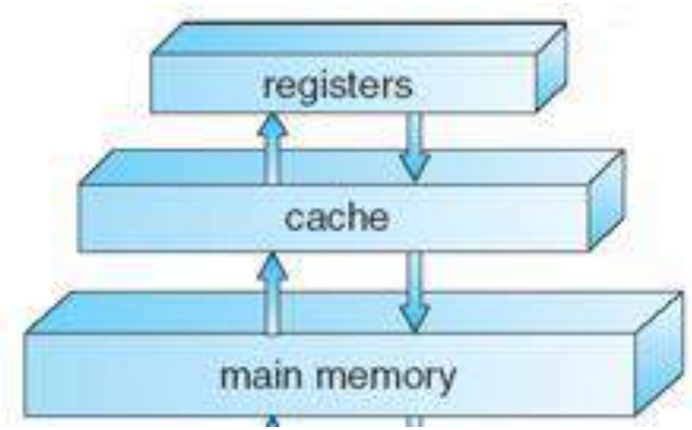
Main Memory <> Cache Memory



Issue

- What issue arises from having a *Main Memory* and a *Cache Memory*? _____

Storage Hierarchy



- Storage systems organized, in hierarchy, by
  - Access Speed ____
  - Cost per bit ____
  - Storage Size _____

Instruction Execution Cycle

Loop
- Instruc Fetch
- Decode
- Fetch operands
- Execute
- Store output

Repeat Loop until
EXIT / HALT

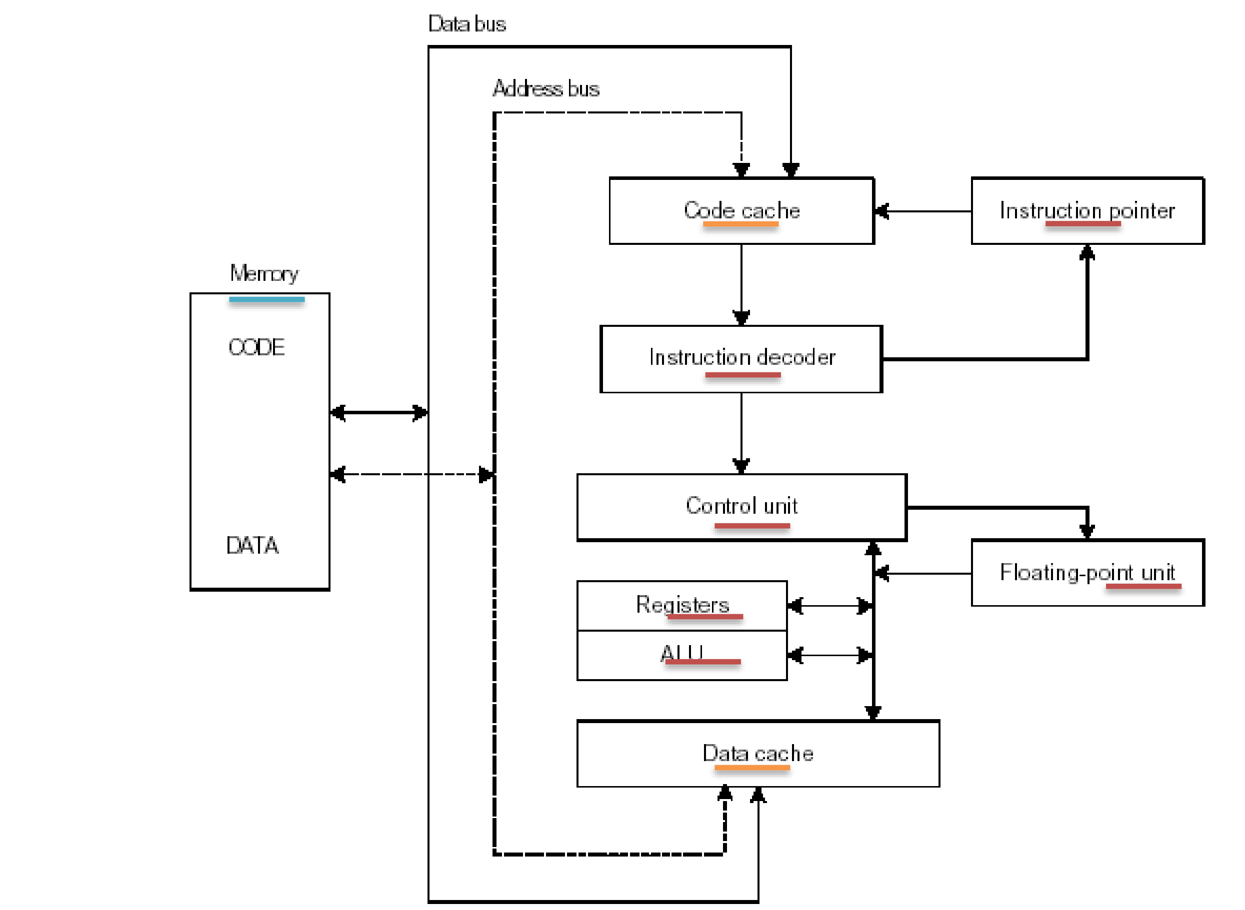
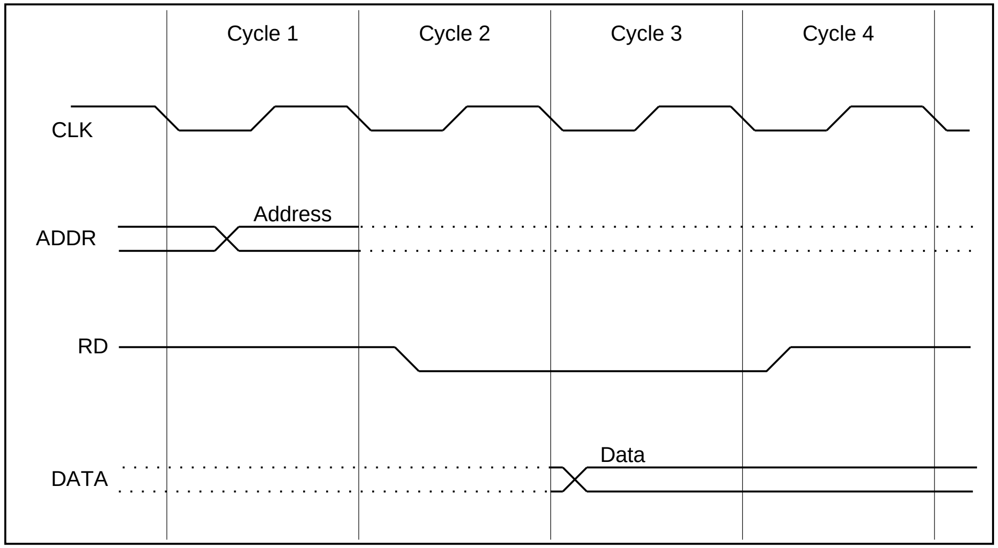


Figure 2–2 Simplified Pentium CPU Block Diagram.

Reading from Memory

- Multiple machine cycles are required when reading from memory, because it responds much more slowly than the CPU. The steps are:
  - Execution step
  - Cycle 1: address placed on address bus
  - Cycle 2: Read Line (RD) set low (0), changing the value of processor's RD
  - Cycle 3: CPU waits one cycle for memory chips to respond
  - Cycle 4: Read Line (RD) goes to 1, indicating that the data is on the data bus and can be copied



Memory Read Cycle

Referencias

- Chapters: Irvine, Kip R., Assembly Language for x86 Processors.
- Chapters: Brey, Barry B., The Intel Microprocessors
- Notas de Ramón Ríos.

- Agosto – diciembre 2023

ORGANIZACIÓN Y PROGRAMACIÓN DE COMPUTADORAS

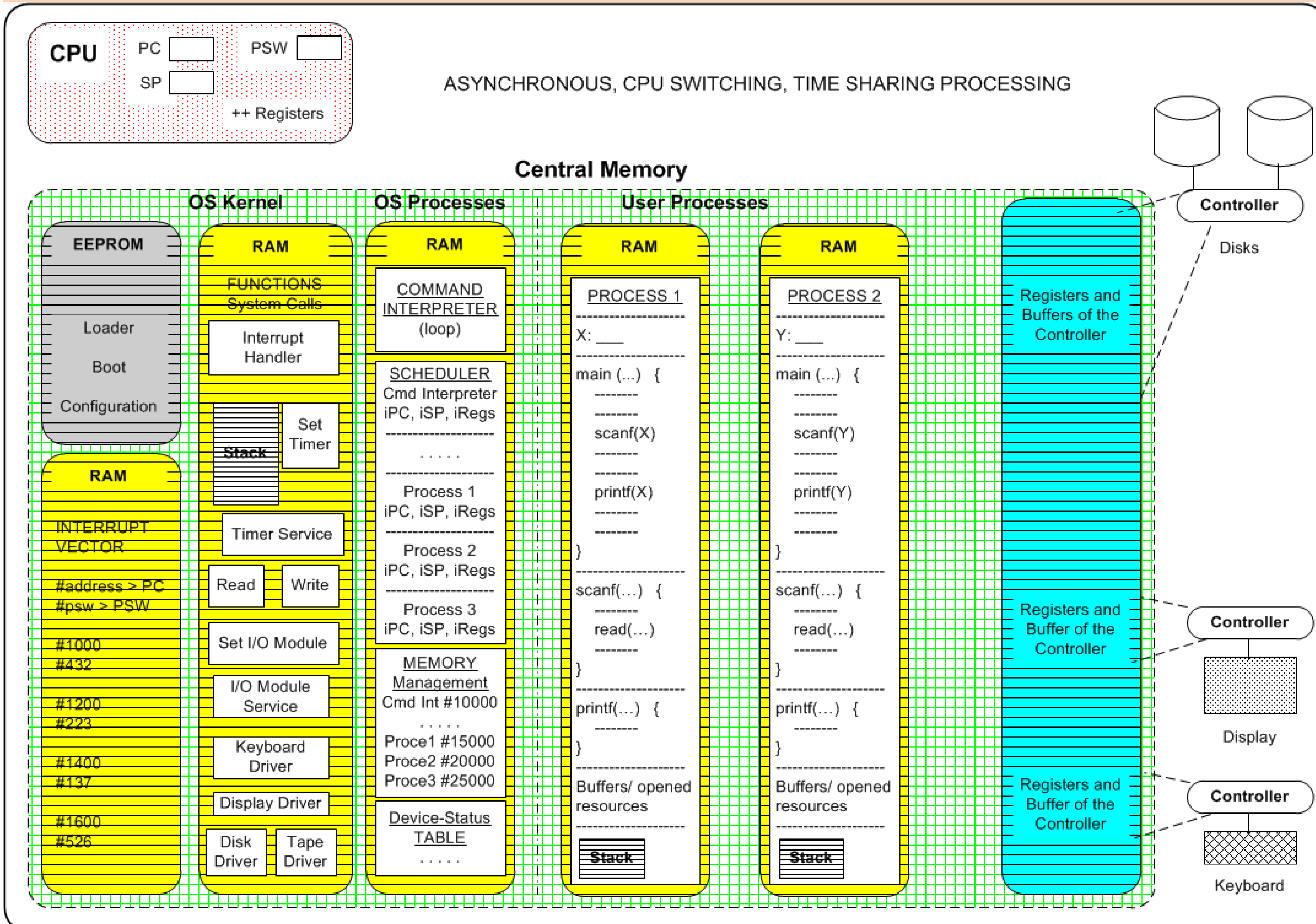
OPC

A – D 2023

Main, or Central, Memory

- Main Memory structure
  - A Linear Vector of Memory Bytes (*content*), with the *indexes* being the *memory addresses* to Access the *content* of the Memory Bytes
- Hardware components
  - EEPROMs
  - xRAMs (e.g. DRAM)
  - Device Controllers: each one with *Control Registers* and a *Data Buffer*

Memory Layout for Multiprogrammed System



Numerical Values: meanings

- Data content: unsigned and signed values
  - Registers
  - Memory Byte locations
- Memory Addresses
  - To Access Memory Byte locations
  - Represented as numerical unsigned values

Multiple Memory Byte Values

Byte Values used as Data

- One-byte content
- Two-byte content
- Four-byte content
- Eight-byte content
- Always Powers of 2, content

Main Memory Addressing

Memory	Central Memory	in	in	in
address bits	Bytes	KBytes	Mbytes	GBytes
4 bits	16			
6 bits	64			
8 bits	256			
16 bits	65,536	64		
20 bits	1,048,576	1,024	1	
24 bits	16,777,216	16,384	16	
32 bits	4,294,967,296	4,194,304	4,096	4
36 bits	68,719,476,736	67,108,864	65,536	64
48 bits	281,474,976,710,656	274,877,906,944	268,435,456	262,144
64 bits	18,446,744,073,709,600,000	18,014,398,509,482,000	17,592,186,044,416	17,179,869,184

Referencias

- Chapters: Irvine, Kip R. Assembly Language for x86 Processors.
- Notas de Ramón Ríos.
- Agosto – diciembre 2023

ORGANIZACIÓN Y PROGRAMACIÓN DE COMPUTADORAS

OPC

A – D 2023

Intel Microprocessor Evolution

- Intel 8080, 8088
- Intel 8086
- Intel 80286
- IA-32 processor family

Data and Addresses in BUS´s channels

Microprocessors / Computer Systems

Early Intel Microprocessors

- Intel 8080
  - 8-bit registers, 32-bit integers
  - 64KB addressable RAM (____-bit addressing)
  - S-100 BUS architecture, 8-bit data bus
  - MS-DOS Operating System
  - 8-inch floppy disks!
- Intel 8086
  - IBM-PC Used 8086
  - 16-bit registers
  - 1 MB addressable RAM (___-bit addressing)
  - 16-bit data bus
  - Separate floating-point unit (8087)
  - Backward-Compatibility: *this approach allows older software programs (binary) to run on newer computers*.

The IBM-PC/AT

- Intel 80286
  - 16-bit microprocessor registers
  - 16 MB addressable RAM (___-bit addressing)
  - 24-bit address bus
  - Protected memory mode
  - Several times faster than 8086
  - Introduced IDE (Integrated Drive Electronics) BUS architecture
  - Separate 80287 floating point unit (FPU)

Intel IA-32 (32-Bit x86) Family

- Intel386 (80386)
  - 32-bit data registers
  - 32-bit address, paging (virtual memory)
  - __ GB addressable RAM
  - *Windows NT* and *Linux* Operating System
- Intel486
  - Instruction pipelining
  - FPU, inside the main chip
- Pentium, +Pro, +II, +III, +4, +5
  - 32-bit address bus, 64-bit data
  - Instruction superescalar

IA-32 (32-Bit x86) Processor Architecture

- Modes of operation
- Basic execution environment
- Floating-Point Unit FPU

IA-32 Modes of Operation -1

- Protected Mode
  - native state for 80x86 CPUs (Windows NT, Linux)
  - every resource can be used
    - all the instructions, and devices
  - 4GB of central memory, 32-bit addresses
    - the whole main memory
  - programs (processes) given separate memory (processes) areas
  - security: every process can not address other process area

IA-32 Modes of Operation -2

- Real-Address Mode
  - implements programming environment of Intel 8086 processor
  - 1 MB of central memory, 20-bit addresses
    - only 1 MB out of the 4GB
  - native MS-DOS
  - backward compatibility
    - it runs on the 80x86 processor
  - the MS-DOS works like a *virtual machine*
  - program can cause MS-DOS crash
    - None, the other Windows NT processes

IA-32 Modes of Operation - 3

- System Management Mode
  - Provides OS with:
    - boot configuration
    - power management (power-on, battery level, shutdown)
    - system security
    - diagnostics
      - main memory, disks unit, graphical cards, nic cards, mouse, keyboard, usb unit, monitors, …
    - disks defragmenter
  - This functions are implemented by computer (hardware) manufacturers

IA-32 Modes of Operation - 4

- Virtual-8086 sub-mode
  - hybrid of Protected Mode
  - allow several sessions of Real-Address Mode
  - each session is a MS-DOS *virtual machine*
  - each program has its own 8086 computer
  - if a MS-DOS virtual machine crashes, it does not affect the other process
  - every MS-DOS *virtual machine* uses 1 MB of the central memory

Referencias

- Chapters: Irvine, Kip R. Assembly Language for x86 Processors.
- Notas de Ramón Ríos.
- Agosto – diciembre 2023

Basic Execution Environment

IA-32

- Addressable memory
- General-purpose registers
- Index and base registers
- Specialized register uses
- Status flags

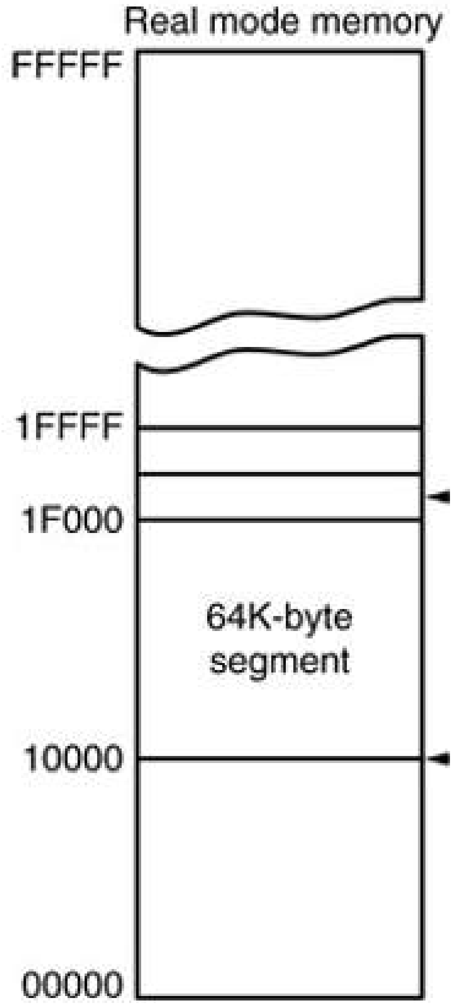
IA-32 Addressable Memory

- **Protected mode**
  - 4 GB
  - 32-bit address (0 - 4,294,967,295), 4-Byte address
- **Real-address mode** and **Virtual-8086 sub-mode**
  - 1 MB space
  - 20-bit address (0 - 1,048,575), 2.5-Byte address
  - In Protected Mode running multiple (Virtual-8086 sub mode) programs, each program has its own 1 MB memory area

Protected Mode Memory

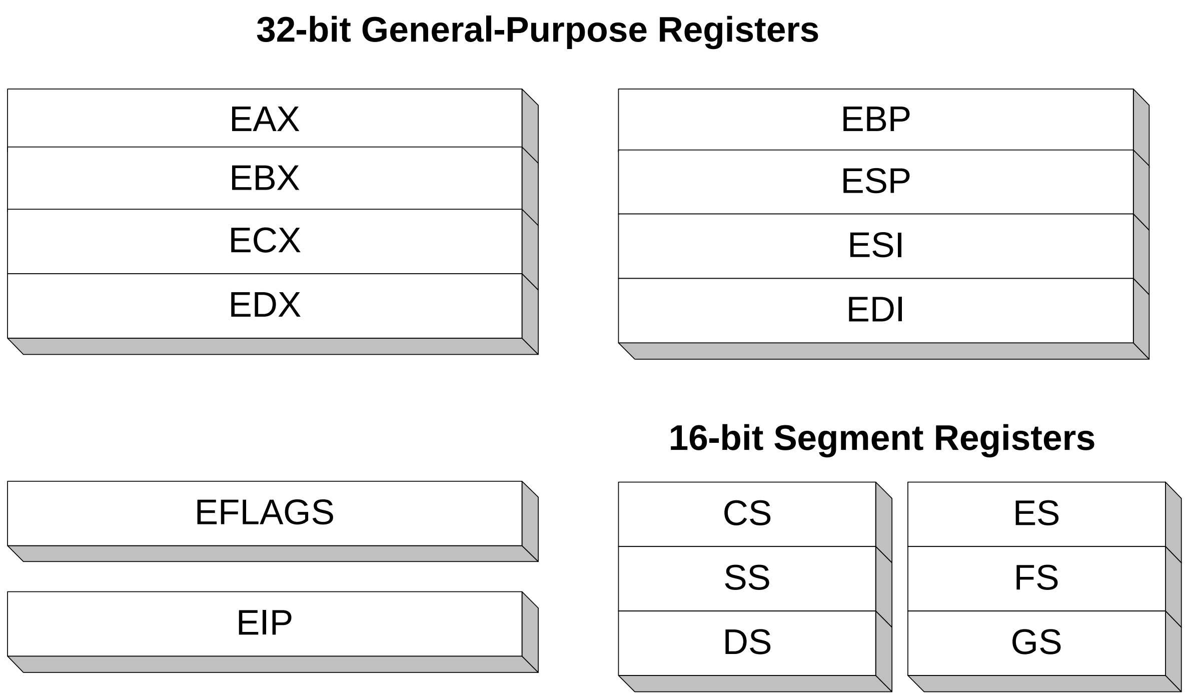
	Memory
FFFFFFFF	1-Byte
FFFFFFFE	1-Byte
FFFFFFFD	1-Byte
	1-Byte
	1-Byte
	1-Byte
	1-Byte
	1-Byte
	1-Byte
	1-Byte
	1-Byte
	1-Byte
00000010	1-Byte
00000001	1-Byte
00000000	1-Byte

Real-address Mode Memory



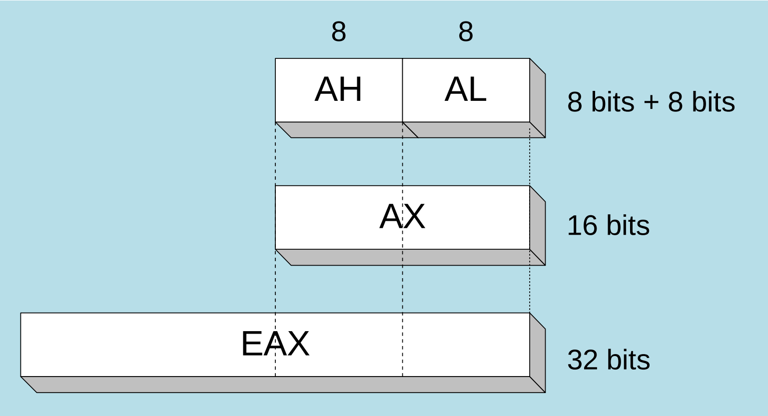
Program Execution Registers

Named storage locations inside the CPU, optimized for high-speed.



General-Purpose Registers (1/2)

- **32 bit-Registers**: EAX, EBX, ECX, and EDX
- Primarily used for *arithmetic* and *data movement*
- **Lower half** of these registers can be broken down as:
  - two 8-bit values, or / and
  - one 16-bit value (both for backward-compatibility)



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

General-Purpose Registers (2/2)

- **32 bit-Registers**: ESI, EDI, EBP, and ESP
- These registers are used for *addressing*
- **Lower half** of these registers can be broken down as:
  – one 16-bit value

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

Some Specialized Register Uses (1 / 3)

- 32-bit General-Purpose Registers
  - EAX – extended accumulator (*mult, divi*)
  - ECX – CPU loop counter
  - ESP – CPU extended stack pointer or SP
  - ESI, EDI – index registers
  - EBP – extended frame pointer (*stack*, for high-level languages parameters)
    - Should not be used for arithmetic or data transfer

Some Specialized Register Uses (2 / 3)

- EIP – *32-bit* Instruction Pointer (traditional **PC**, *Program Counter*)
  - Contains address of next instruction to be executed

- EFLAGS – *32-bit* Processor Status Flags
  - *status* and *control* flags
  - each flag is a single binary bit
  - A flag is set when it equals 1; it is clear (or reset) when it equals 0
  - The flags are affected after *exec instructions*

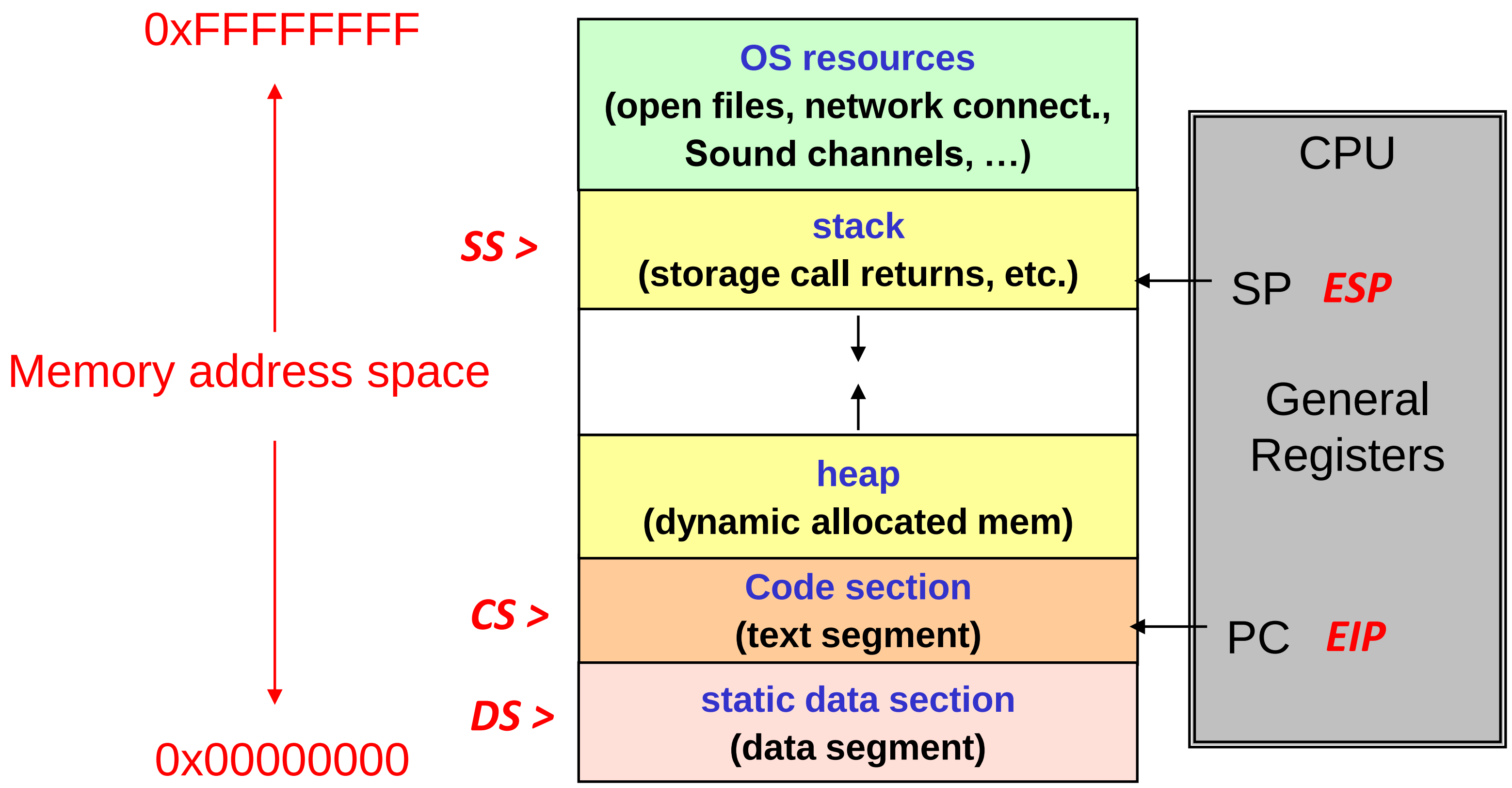
Status 1-bit Flags / EFLAGS

- Carry
  - unsigned arithmetic out of range
- Overflow
  - signed arithmetic out of range
- Sign
  - result is negative
- Zero
  - result is zero
- Auxiliary Carry
  - carry from bit 3 to bit 4, in 8-bit operand
- Parity
  - sum of 1 bits, is an even number (in LSB)

Some Specialized Register Uses (3 / 3)

- 16-bit Segment Registers (for processes)
  - CS – code segment, holds code; programs and procedures
  - DS – data segment, contains most data used by a program
  - SS – stack segment, defines the area of memory used for the stack
  - ES, FS, GS - additional segments for extra data

Process memory schema



Memory Addressing Schemes

Higher address

EXEC
FLOW

Lower address

	Memory
FFFFFFFF	1-Byte
FFFFFFFE	1-Byte
FFFFFFFD	1-Byte
	1-Byte
	1-Byte
	1-Byte
	1-Byte
	1-Byte
	1-Byte
	1-Byte
	1-Byte
	1-Byte
00000010	1-Byte
00000001	1-Byte
00000000	1-Byte

Higher address

EXEC
FLOW

Lower address

Data allocation

- 32-bit Registers
  - For 32-bit data values, signed or not.
  - MSB: bit 31; LSB bit 0.

- 8-bit Memory Locations
  - How come a 32-bit value is allocated in 8-bit memory locations?

1- Byte Data allocation

- 1-Byte Example, *AL* register and byte variable *alfa*:

AL: **containing a numerical value of** 12_h

alfa, **has a memory address location of** 00000100_2

AL	12					Memory
				00000100		12
				00000101		
				00000102		
				00000103		

Endianness memory storage

- Is the sequential order in which Bytes (more than one), representing a numerical value, are stored in main memory.
  - Numerical value storage > 1 Byte

- Two formats for Endianness
  - Big-endian, MSB stored first, … LSB stored last
  - Little-endian, LSB stored first, … MSB stored last

Little Endian Order

- 2-Byte Example, *BX* register and short variable *beta*:

***BX*: containing a numerical value of 1234$_h$**

***beta*, has a memory address location of 00000100$_2$**

BX	12	34				Memory
				00000100		34
				00000101		12
				00000102		
				00000103		

Little Endian Order

- 4-Byte Example, *ECX* register and integer variable *delta*:

ECX: containing a numerical value of 12345678$_h$

delta, has a memory address location of 00000100$_2$

ECX		12	34	56	78				Memory
							00000100		
							00000101		
							00000102		
							00000103		

64-bit x86-64 Processors - 1

- Intel64 (x86-64 specification)
  - P6 series, extended to 64 GB (___-bit addressing)
  - 64-bit linear address space
  - 64-bit Mode, Windows 64 uses this
  - Protected, Real-address and System Management Modes.

- IA-32e Mode (backward-compatibility) has two sub-modes
  - *Compatibility Mode* for legacy 16- and 32-bit applications, Windows supports only 32-bit apps in this mode
  - *64-bit Mode* uses 64-bit addresses and 64-bit (and 32-bit) operands

64-bit x86-64 Processors - 2

- 64-bit Operating Systems over x86-64 Processors
- E.g. Windows v7, v10, servers v2012, …
  - What exec programs are installed inside the next folders and Why Are these folders Split Up?:
    - "C:\Program Files"? ________
    - "C:\Program Files (x86)" ? ________
  - Task Manager display it up > (32 bits)

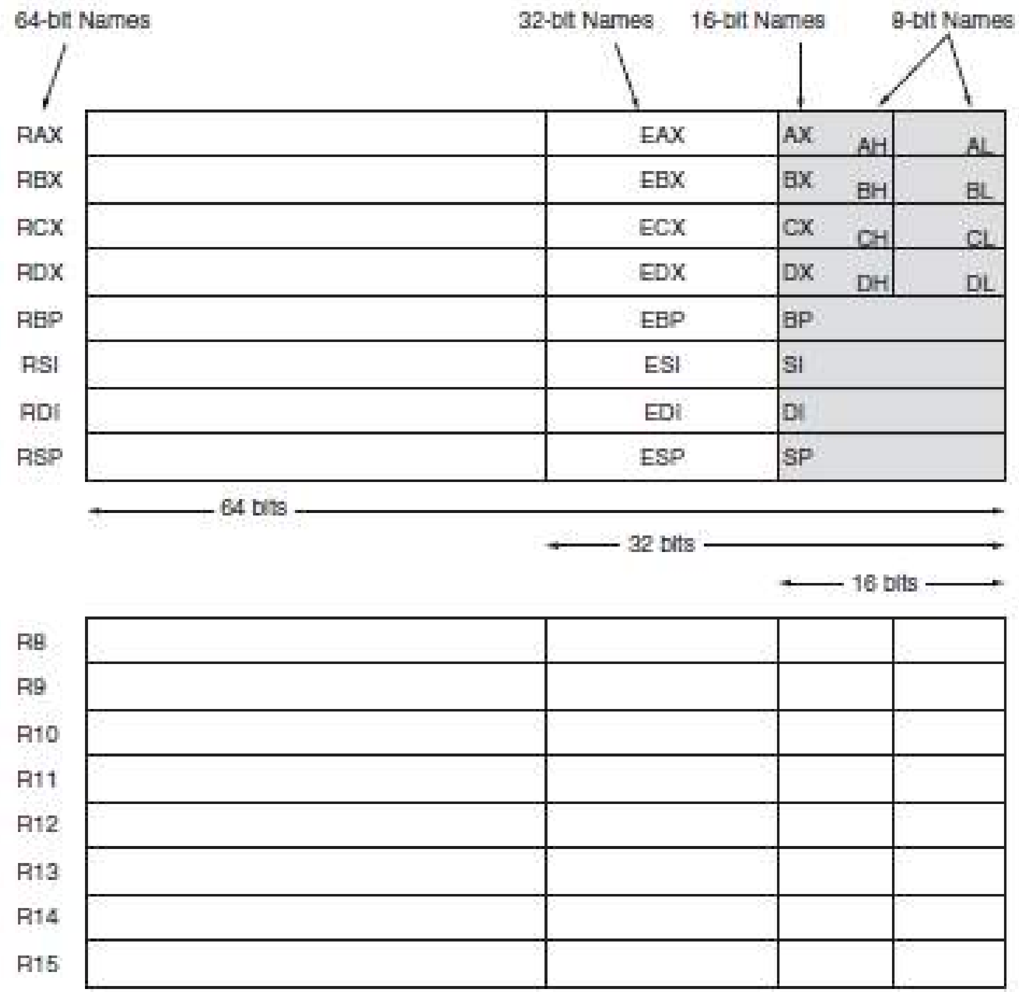
64-bit x86-64 Processors

- Basic Execution Environment
  - addresses can be 64 bits
  - 16 64-bit general purpose registers RAX-R15
  - A 64-bit status flags named RFLAGS
  - 64-bit instruction pointer (Program Counter) named RIP

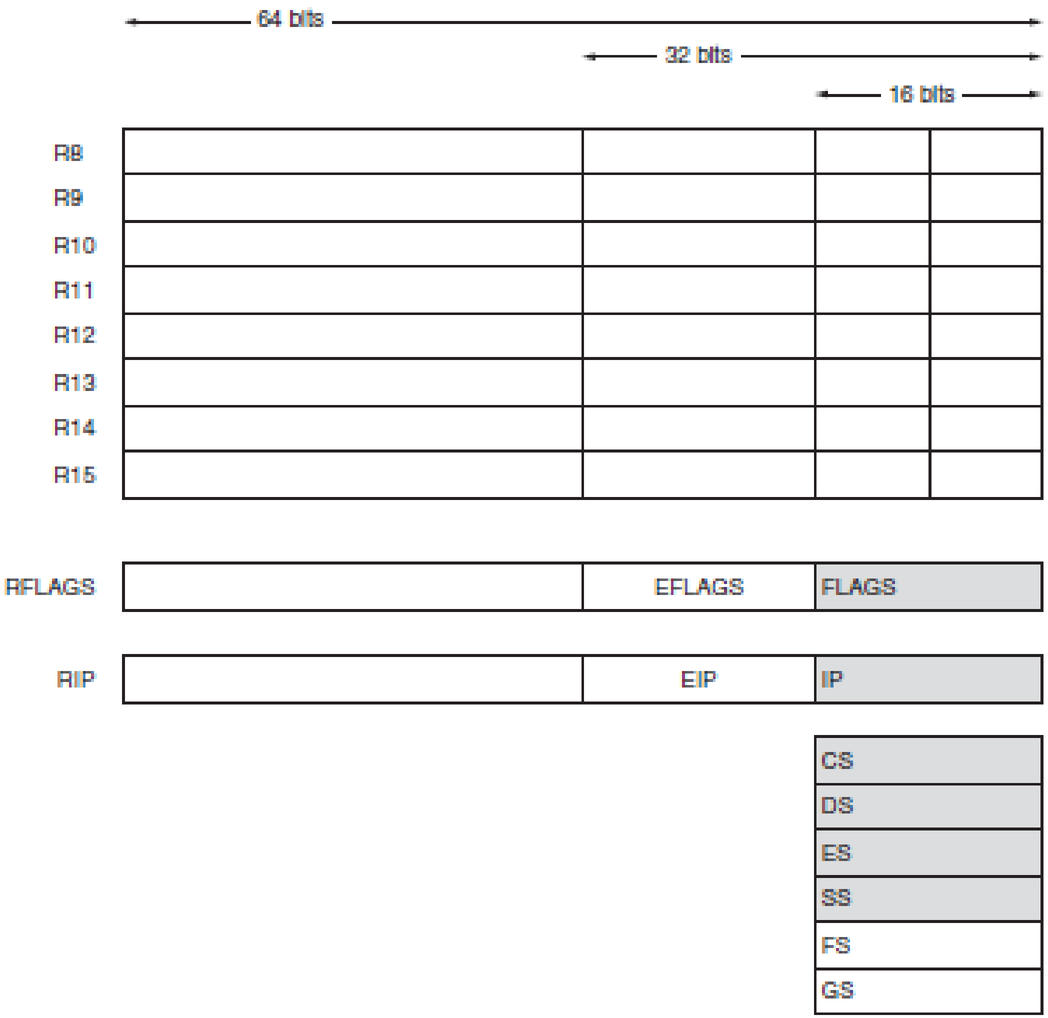
64-Bit General Purpose Registers

- 64-bit general purpose registers:
  - *RAX, RBX, RCX, RDX, RDI, RSI, RBP, RSP, R8-R15*
- 32-bit general purpose registers:
  - EAX, EBX, ECX, EDX, EDI, ESI, EBP, ESP, *R8D-R15D*
- 16-bit general purpose registers:
  - AX, BX, CX, DX, DI, SI, BP, SP, *R8W-R15W*
- 8-bit general purpose registers:
  - AL, BL, CL, DL, AH, BH, CH, DH, *R8B-R15B*

64-Bit Registers RAX-R15



64-Bit Registers RFLAGS, RIP



Multipurpose Registers - 1

- RAX - a 64-bit register (RAX), a 32-bit register (accumulator) (EAX), a 16-bit register (AX), or as either of two 8-bit registers (AH and AL).
- The accumulator is used for instructions such as multiplication, division, and some of the adjustment instructions.
- RBX, addressable as RBX, EBX, BX, BH, BL.
  – BX register (base index) sometimes holds offset address of a location in the memory system in all versions of the microprocessor

Multipurpose Registers - 2

- RCX, as RCX, ECX, CX, CH, or CL.
  – a (count) general-purpose register that also holds the count for various instructions
- RDX, as RDX, EDX, DX, DH, or DL.
  – a (data) general-purpose register
  – holds a part of the result from a multiplication or part of dividend before a division
- RBP, as RBP, EBP, or BP.
  – points to a memory (base pointer) location for memory data transfers

Multipurpose Registers - 3

- R8 - R15.
  - data are addressed as 64-, 32-, 16-, or 8-bit sizes and are of general purpose
  - bits 8 to 15 are not directly addressable as a byte

Multipurpose Registers - 4

- RDI addressable as RDI, EDI, or DI.
  - often addresses (destination index) string destination data for the string instructions
- RSI used as RSI, ESI, or SI.
  - the (source index) register addresses source string data for the string instructions
  - like RDI, RSI also functions as a general-purpose register

Special-Purpose Registers

- RIP addresses the next instruction in a section of memory.
  - defined as (instruction pointer) a code segment
- RSP addresses an area of memory called the stack.
  - the (stack pointer) stores data through this pointer
- RFLAGS indicate the condition of the microprocessor and control its operation.

Little Endian Order

- 8-Byte Example, *RAX* register and integer*8 variable *omega*:

RAX: containing a numerical value of D9B3563412 78A2AF$_h$

omega, has a memory address location of 00000100$_2$

RAX		D9	B3	56	34	12	78	A2	AF				Memory	
											00000100			
											00000101			
											00000102			
											00000103			
											00000104			
											00000105			

Referencias

- Chapters: Irvine, Kip R. Assembly Language for x86 Processors.
- Chapters: Brey, Barry B., The Intel Microprocessors
- Notas de Ramón Ríos. 05.
- Agosto – diciembre 2023

ORGANIZACIÓN Y PROGRAMACIÓN DE COMPUTADORAS

OPC

A – D 2023

Assembly Language vs High-Level L

Assembly

- One phrase per line
- One instruction per line
- Operations (+,-,*,/,…) are represented by mnemonics (representing instructions).
- Is not structured
- Very short object programs
- Runs faster

Example Program

;Add two numbers and displays the result

```
.CODE
main PROC
    mov  eax, 24          ;  move 24 to the EAX register
    add  eax, 17          ;  add 17 to the EAX register
    call  WriteInt        ;  display EAX content value
    exit                  ;  to end execution
main END
```

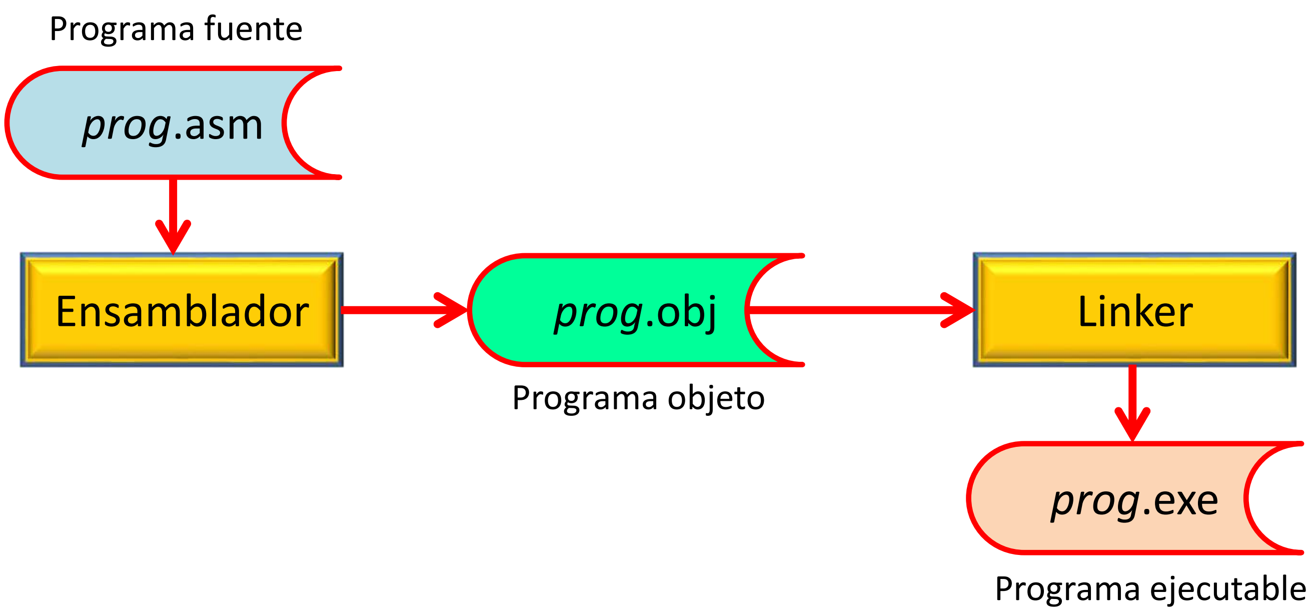
;".CODE", "main": directives of MASM

;"*WriteInt*":   function of the Library Subroutine of MASM (Assembler)

;"*exit*":   function of Windows; translated to "call exit"

Ejecución de Programas en Leng. Ensamblador

- Flujo tradicional para desarrollar y correr programas en **Leng. Ensamblador (Assembly)**:
  - *Ensamble* con el **Ensamblador (Assembler)**, *prog.asm* a *prog.obj*
  - *Ligado o vinculado* (linking) con el **Linker**, *prog.obj* a *prog.exe*
  - Ejecución de programa *prog.exe*



Basic Elements of Assembly Language

- Integer constants
- Integer expressions
- Character and string constants
- Identifiers and Reserved words
- Directives
- Mnemonics and Operands for instructions
- Labels
- Comments

Integer Constants

- [{+ | -}] digits [*radix*]
- Optional leading *+* or – sign, default *+*
- Radix: Binary, decimal, hexadecimal, or octal digits
- Common *radix* characters:
  - h – hexadecimal
  - q | o – octal
  - d – decimal (default)
  - b – binary
  - r – encoded real
  - t – decimal (alternate)
  - y – binary (alternate)
- If no radix given, assumed to be *decimal*

Examples: 30d, 30, 6Ah, -42, 1101b, 53o

Hexadecimal beginning with letter, prefix 0 (zero): 0B4h, 0A5h ;Why?

Real Number Constants -1

- Represented as decimal reals or encoded (hexadecimal) reals
- Decimal real contains optional sign followed by integer, decimal point, and optional integer that expresses a fractional and an optional exponent (FP formats)
  - [sign] integer.[integer] [exponent]
  - Sign {+, -}
  - Exponent E[{+, -}] integer
- Examples
  - 2.
  - +3.0
  - -44.2E+05
  - 26.E5

Real Number Constants -2

- Represented as decimal encoded (hexadecimal) reals
- Example
  - Binary representation of +1.0
  - 0011 1111 1000 0000 0000 0000 0000 0000b

- Example
  - Decimal encoded hexadecimal real of +1.0
  - 3F800000r

Character and String Constants

- Enclose character in single or double quotes
  - 'A', "x"
    - ASCII character = 1 byte
- Enclose strings in single or double quotes
  - "ABC"
  - 'xyz'
    - Allocation of characters is byte after byte, same single order
- Embedded quotes:
  - 'Say "Goodnight," Gracias'

Character and String Constants

```
.DATA
Alfa        SDWORD 7          ; allocate a signed 4-Byte memory
Beta        SDWORD 11h        ; 11h > 17
R           SDWORD 0
msgr        BYTE  "El Resultado R= ",  0     ; bytes allocated?

.CODE
main PROC
            mov EAX, Alfa         ;     EAX:7
            neg EAX              ;  -?    EAX: -7
            add EAX, 9            ;  -?  EAX: 2
            sub EAX, Beta        ;  -?   EAX: -15
            inc EAX              ;  -?  EAX: -14
            mov R, EAX      ; R = resultado          -?

            mov  EDX,  OFFSET msgr
            call  WriteString       ; imprime el String que comienza en  msgr
            call  WriteInt          ;imprime el contenido de EAX, o sea de R
            exit
main ENDP
END main
```

Identifiers

- Identifiers
  - Programmer-chosen name to identify a *variable*, *constant*, *procedure* (*function*), or *label*
  - 1-247 characters, including digits
  - not case sensitive
  - first character must be a *letter*, _, @, ?, or $
    - Subsequent characters may also be digits
  - Cannot be the same as a *reserved word*
  - @ is used by assembler as a prefix for predefined symbols, so avoid it identifiers
- Examples
  - var1, Count, $first, _hello, MAX,
  - open_file, myFile, xVal, _12345

Reserved Words

- Reserved words cannot be used as identifiers
  - Instruction mnemonics
    - MOV, ADD, MUL,, …
  - Register names
    - EAX, EBX, …
  - Directives –
    - .data, .code, .stack
    - PROC, END
    - BYTE, WORD, SDWORD
- See MASM reference in Appendix A; Irvine

Directives 1

- *Commands* embedded in the source code that are recognized and acted upon by the **Assembler**,
  - Not part of the Intel instruction set
  - tells MASM how to assemble programs
  - Work at assembly time
  - Do not execute (.exe) at runtime

Directives 2

- *Commands* . . .
  - Used to declare code, data areas, select memory model, declare procedures, etc.
    - .code, .data, .DATA, .Data, .stack
    - flat
    - PROC, END
    - Type attributes – provides size and usage information
      - BYTE, WORD, DWORD, SDWORD

```
one    DWORD 34                  ; DWORD directive, set aside
                                 ; enough space for double word
mov      eax, one                ; MOV instruction
```

Instructions

- An instruction is a statement that becomes executable when after a program is assembled.
- Assembled into machine code (0s and 1s) by assembler
- Loaded and executed at runtime by the CPU
- We use the Intel IA-32/64 instruction set
- An instruction contains four basic parts:
  - Label (optional)
  - Mnemonic (required)
  - Operand (depends on the instruction)
  - Comment (optional)
- Basic syntax
  - [label[:]] [mnemonic [operands]] [; comment] ?

Labels

- Act as place markers or addresses
  - marks the address (offset) of *code* and *data*
- Follow identifier rules
- **Data** label
  - must be unique
  - example: **temp** (not followed by colon)
  - `temp DWORD 100`
- **Code** label
  - target of some type of jump
  - example: **here:** (followed by colon)

```
here:
        mov   ebx, temp
        …
        jmp here
```

Mnemonics and Operands

- Instruction Mnemonics
  - short word that identifies an instruction
  - examples: MOV, ADD, SUB, MUL, INC, DEC, NOP, JUMP, CALL, IMUL
  - describe the type of operation
- Operands
  - each instruction can have between **0** and **tree** operands
  - constant 96
  - constant expression 4+5*2 ; in the Assembly time 14
  - register EAX
  - memory (data label) temp ; variable alike temp+2

Constants and constant expressions are often called immediate values (at assembly time)

Mnemonics and Operands Examples

- STC instruction
- stc ; set Carry flag

- INC instruction
- inc eax ; add 1 to EAX

- MOV instruction
- mov temp, ebx ; move EBX to temp
- ; first operation is destination
- ; second is the source
- IMUL instruction (three operands)
- imul eax, ebx, 5 ; EBX multiplied by 5, product in EAX

Comments 1

- Comments are good!
  - explain the program's purpose
  - when it was written, and by whom
  - revision information
  - Technical notes about coding (programming) techniques
  - application-specific explanations
- Single-line comments
  - begin with semicolon (;)
- Multi-line comments
  - begin with COMMENT directive and a programmer-chosen character
  - end with the same programmer-chosen character

Comments 2

- Single line comment
  - inc eax ; single line at end of instruction
  - ; single line at beginning of line
- Multiline comment

```
COMMENT !
        This line is a comment
        This line is also a comment
!
COMMENT &
        This is a comment
        This is also a comment
&
```

NOP instruction

- Doesn't do anything, just waste a clock cycle (tick)
- Takes up one byte of memory
- Sometimes used by compilers and assemblers to align code to even-address boundaries. (as file blocks)
- The following MOV generates three machine code bytes. The NOP aligns the address of the third instruction to a doubleword boundary (even multiple of 4)

```
00000000    66    8B    C3    mov ax, bx
00000003    90                nop    ;  align next instruction
00000004
```

Instruction Format Examples

- No operands
  - stc, nop ; set Carry flag, no operation
- One operand
  - mul 7 ; constant
  - inc temp ; memory
- Two operands
  - add ebx, ecx ; register, register
  - sub temp, 25 ; memory, constant
  - add eax, 36*25 ; register, constant-expression

Referencias

- Chapters: Irvine, Kip R. Assembly Language for x86 Processors.
- Notas de Ramón Ríos. 05.
- Agosto – diciembre 2023

ORGANIZACIÓN Y PROGRAMACIÓN DE COMPUTADORAS

OPC

A – D 2023

IA Directives: Defining Data

DATA SECTION (.data, .DATA) in 32-bit programs

- Intrinsic Data Types
- Data Definition Statement
- Defining BYTE and SBYTE Data
- Following the *Little-Endian Order*
- Defining WORD and SWORD Data
- Defining DWORD and SDWORD Data
- Defining QWORD Data
- Defining TBYTE Data
- Defining Real Number Data
- Declaring Uninitialized Data

IA Directives: Defining Data

- These Assembly Directives in DATA SECTION:
  - Allocate memory space (bytes), and
  - Can define an initial value in the memory space.

- **.DATA** / **.data** section:
  - Fixes the initial address of the user program (*.asm* file).

Directives of Intrinsic Data Types 1

INTEGERs: size and content (unsigned, signed)

- BYTE, SBYTE
  - 8-bit unsigned integer; 8-bit signed integer (1 byte)
- WORD, SWORD
  - 16-bit unsigned & signed integer (2 bytes)
- DWORD, SDWORD
  - 32-bit unsigned & signed integer (Double word, 4 bytes)
- QWORD, SQWORD
  - 64-bit unsigned integer (Quad word, 8 bytes)
  - 64-bit signed integer (not valid in IA-32)
- TBYTE
  - 80-bit integer (Ten bytes)

Directives of Intrinsic Data Types 2

REALs, with fractions

- REAL4
  - 4-byte IEEE short real
- REAL8
  - 8-byte IEEE long real
- REAL10
  - 10-byte IEEE extended real

Data Definition Statement

- A *data definition statement* (a directive) sets aside storage in memory for a variable.
- May optionally assign a *name* (data label) to the data
- Syntax:

  [*name*] *directive* *initializer* [,*initializer*] . . .

```
alfa BYTE 10
```

- All initializers become *binary data* in memory

Defining BYTE and SBYTE Data

Each of the following defines a single byte of storage:

```
beta    BYTE 'A'            ; character constant
gamma    BYTE 0             ; smallest unsigned byte
omega   BYTE 255            ; largest unsigned byte
delta   SBYTE -128          ; smallest signed byte
sigma   SBYTE +127          ; largest signed byte
eagle BYTE ?                ; uninitialized byte
falcon BYTE 10h
```

Defining Multiple Initializers

Examples that use multiple initializers:

```
list1 BYTE 10,20h,30,40h
list2 BYTE 10h,20,30h,40h
      BYTE 50h,60h,70h,80h
      BYTE 81,82h,83h,84h
list3 BYTE ?,32,41h,00100010b
list4 BYTE 0Ah,20h,'A',22h
```

Offset = relative address

	Offset	Value (h)
list1	0000	0A
	0001	20
	0002	1E
	0003	40
list2	0004	10
	0005	14
	0006	30
	0007	40
	0008	50
	0009	60
	000A	70
	000B	80
	000C	51
	000D	82
	000E	83
	000F	84
list3	0010	?

Defining Strings 1

- A string is implemented as an array of characters
  - For convenience, it is usually enclosed in quotation marks
  - For HLL It will be null-terminated (ending with ,0)
- Examples:

```
str1 BYTE "Enter your name",0
str2 BYTE 'Error: halting program',0
str3 BYTE 'A','E','I','O','U’
AEQuote   BYTE "Imgination is more important than "
          BYTE “knowledge, by Albert Einstein.",0
```

Defining Strings 2

- *New-Line* characters sequence (HLL: “\n”):
  - 0Dh = carriage return (*End-of-Line*)
  - 0Ah = line feed (*Next-Line*)

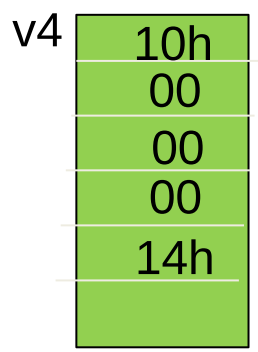
```
; HLL “\n” example:
;    String str4= “Name*\n” +
;         “Enter: “;

str4 BYTE “Name*",0Dh,0Ah
     BYTE “Enter: ",0
```

Idea: Define all strings used by your program in the same area of the data segment.

Using the DUP Operator

- Use DUP to allocate (create space for) an array or string. Syntax: *counter* DUP (*argument*)
- *Counter* and *argument* must be constants or constant expressions

```
v1    BYTE 20 DUP(0)           ; 20 bytes, all with zero
v2    BYTE 20 DUP(?)           ; 20 bytes, uninitialized
v3    BYTE 4 DUP("STACK")      ; 20 bytes: "STACKSTACKSTACKSTACK"
v4    BYTE 10h,3 DUP(0),20     ; 5 bytes
```



Little Endian Order

- All data types larger than one *byte*, store their individual bytes in reverse order. The *least significant byte* occurs at the *first (lowest)* memory address.

- 2-Byte Example:

```
val1 WORD 1234h
```

0000 0000:	34
0000 0001:	12

Defining WORD and SWORD Data

- Define storage for 16-bit integers
  - or double characters
  - single value or multiple values

```
word1   WORD   65535          ; largest unsigned value
word2   SWORD  -32768         ; smallest signed value
word3   WORD   ?              ; uninitialized, unsigned
word4   WORD   "AB"           ; double characters
myList  WORD   1,2,5          ; array of words
array   WORD   5 DUP(25h)     ; uninitialized array
```

Little Endian Order

- 4-Byte Example:

```
val2 DWORD 12345678h
```

0000 0000:	78
0000 0001:	56
0000 0002:	34
0000 0003:	12

Defining DWORD and SDWORD Data

Storage definitions for signed and unsigned 32-bit integers (4 bytes):

```
val2 DWORD  12345678h          ; unsigned
val3 SDWORD -2147483648        ; signed
val4 SDWORD -3, 2              ; signed array
val5 DWORD "ABCD"              ; quad characters

pVal DWORD val4    ;What´s going on here?

val6 DWORD  20 DUP(?)          ; unsigned array
```

Defining QWORD, TBYTE, Real Data

Storage definitions for quadwords, tenbyte values, and real numbers:

```
quad1 QWORD  123456789ABCDEF0h       ;____ Bytes?
val1  TBYTE  1020304050607080 90A0h

rVal1 REAL4  -2.1
rVal2 REAL8  3.2E-260
rVal3 REAL10 4.6E+4096
ShortArray REAL4 20 DUP(0.0)
```

Adding Variables to AddSub

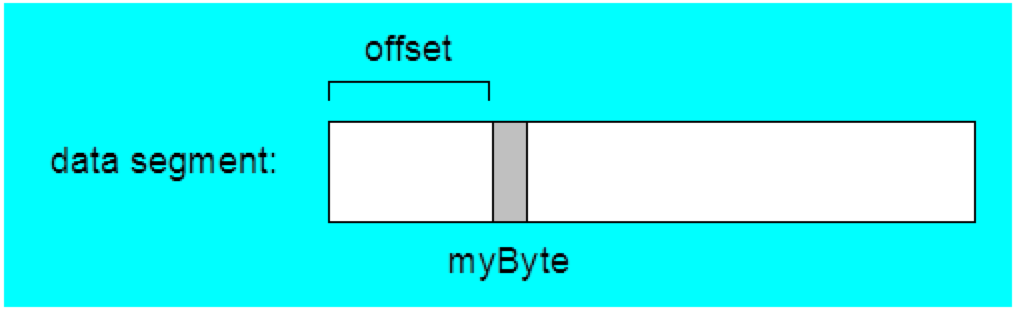
```
TITLE Add and Subtract, Version 2              (AddSub2.asm)
; This program adds and subtracts 32-bit unsigned
; integers and stores the sum in a variable.
INCLUDE Irvine32.inc
.DATA
sal1 DWORD 10000h
sal2 DWORD 40000h
sal3 DWORD 20000h
finalVal DWORD ?
.CODE
main PROC
    MOV EAX,sal1                ; EAX ?
    ADD EAX,sal2                ; EAX ?
    SUB EAX,sal3                ; EAX ?
    MOV finalVal,EAX            ; store the result (_____?)
    CALL DumpRegs               ; display the registers
    EXIT
main ENDP
END main
```

Two-pass Assembler

- When Assembling, the Assembler parses twice the source code of Assembly program (.asm).
  - PASS-1: Define identifiers (e.g. labels), allocate each one with a memory address, and remember them in a Symbol Table.
  - PASS-2: Generate object code (.obj), by converting instructions (nmonics and operands), into respective machine language (binary).

OFFSET Operator Directive

- OFFSET works like part of one operand in an instruction (CODE segment)
- OFFSET returns the distance, in bytes, between the address of a DATA LABEL and the beginning of its enclosing DATA segment
  - Protected mode: 32, 64 bits
  - Real mode: 16 bits



OFFSET Examples

```
.DATA
     BYTE 404000h DUP(?)
bVal BYTE 3
wVal WORD 7
dVal DWORD 4
dVal2 DWORD 5


.CODE
MOV ESI,OFFSET bVal         ; ESI = 00______h
MOV ESI,OFFSET wVal         ; ESI = 00______
MOV ESI,OFFSET dVal         ; ESI = 00______
MOV ESI,OFFSET dVal2        ; ESI = 00______
```

Relating to C/C++

The value returned by OFFSET is a pointer. Compare the following code written for both C++ and assembly language:

```
// C++ version:


char array[1000];



char * p = array;
```

```
; Assembly language:

.DATA
array BYTE 1000 DUP(?)
    p DWORD ?

.CODE
MOV  ESI,OFFSET array
MOV p, ESI
```

Referencias

- Chapters: Irvine, Kip R. Assembly Language for x86 Processors.
- Notas de Ramón Ríos. 06.
- Agosto – diciembre 2023.

ORGANIZACIÓN Y PROGRAMACIÓN DE COMPUTADORAS

OPC

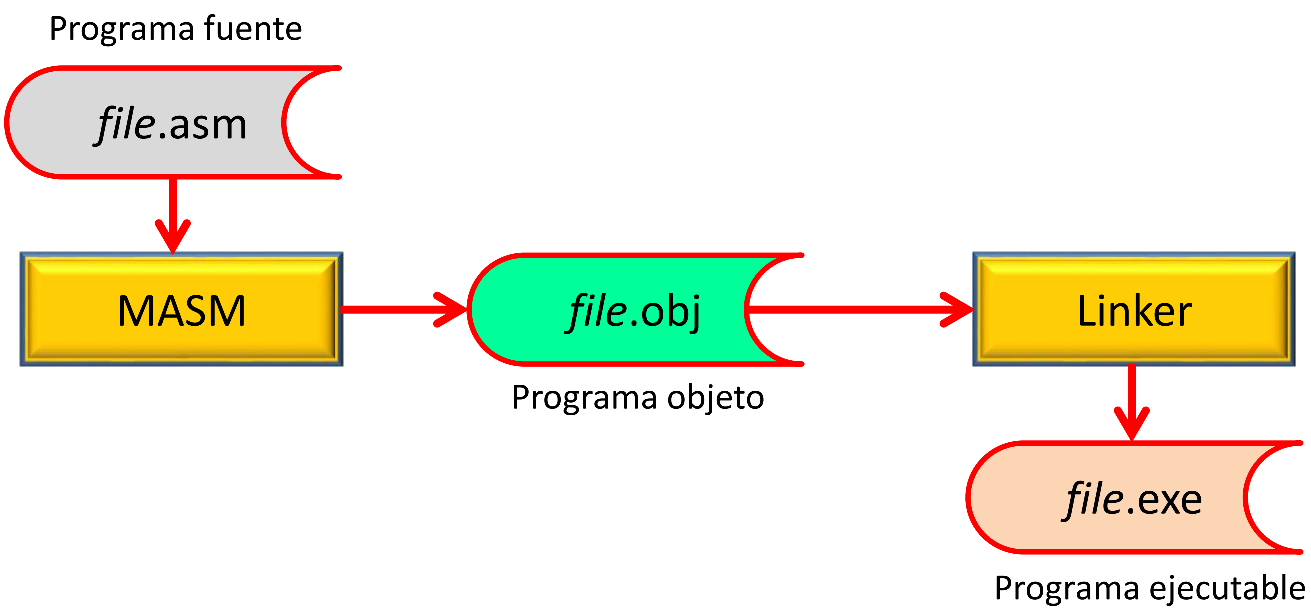
A – D 2023

MS-MASM

- MASM is the Microsoft *M*acro *AS*se*M*bler for x86 Intel processors.
  - MASM is an x86 assembler that uses the Intel syntax for MS-DOS and MS-WINDOWS.
  - There are two versions of the MASM:
    - One (ML) for 16-bit and 32-bit assembly sources, and
    - another (ML64) for 64-bit assembly sources only.

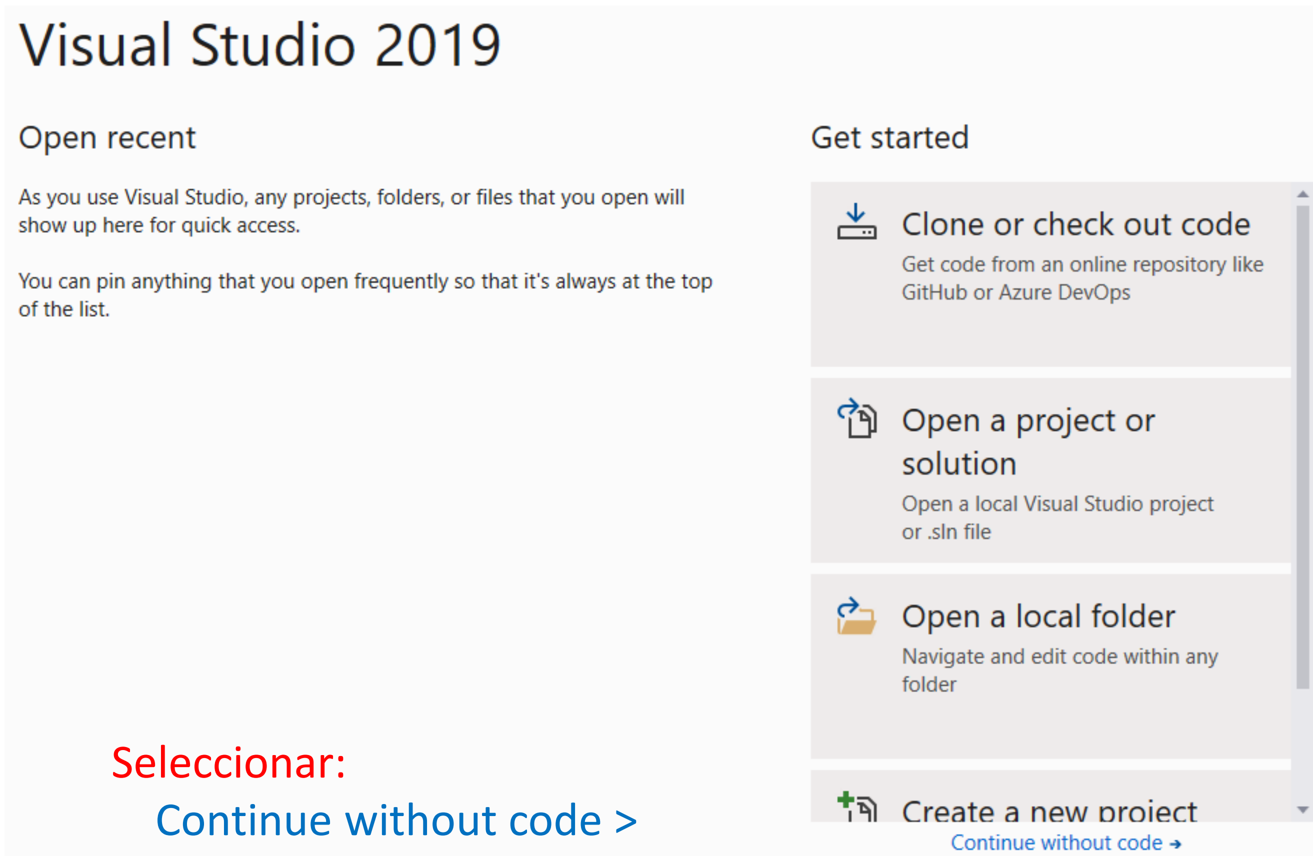
Ejecución de Programas en Leng. Ensamblador

- Flujo para ejecutar programas en Ensamblador:
  - *Ensamble* con el Ensamblador **MASM** (**M**acro **AS**se**M**bler)
  - *Ligado o vinculado* (linking) con el **Linker**
  - Ejecución de programas en Lenguaje Ensamblador (Assembly)



Arrancando Visual Studio 2019

- En Windows:

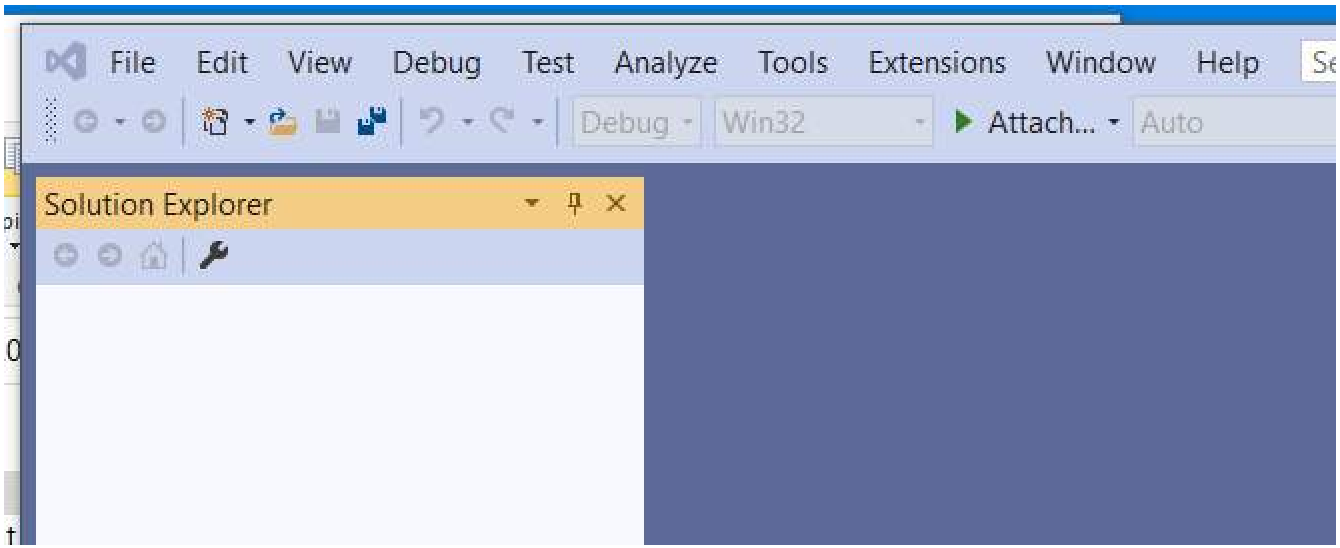


Seleccionar:

Continue without code >

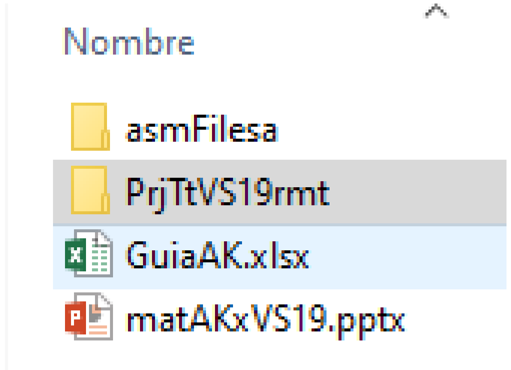
Ventana de Visual Studio 2019

- En Windows:



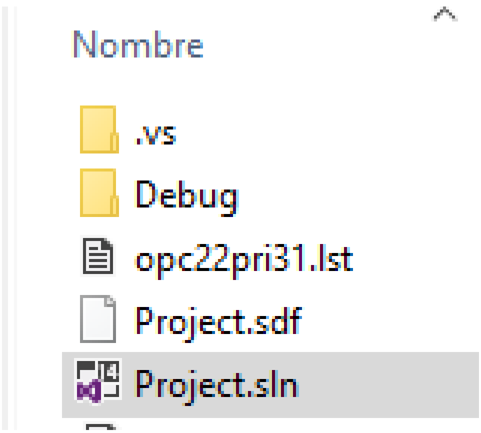
Apertura del proyecto MASM 1

- Proyecto *PrjTtVS19rmt*, previamente preparado para ensamblador, seleccionar:
  - *File > Open > Project / Solution*
- Seleccione la trayectoria de *PrjTtVS19rmt* de su computadora



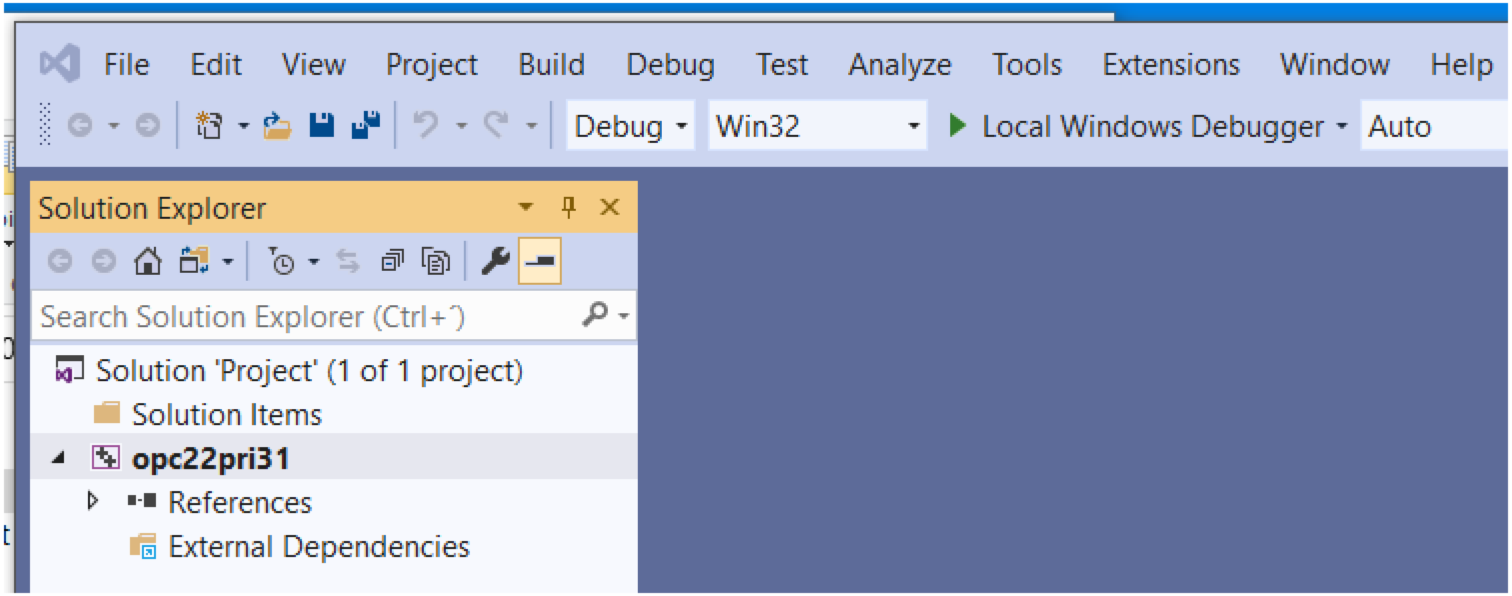
Apertura del proyecto MASM 2

- Apareciendo



- La solución del proyecto queda como *Project.sln*

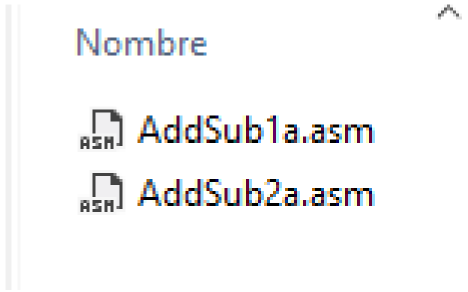
Apertura del proyecto MASM 3



- Note: el nombre del fólder es *PrjTtVS19rmt*, el nombre del Solution Project es *Project.sln* , pero el ejecutable que se generara se llamara ***opc22pri31.exe***

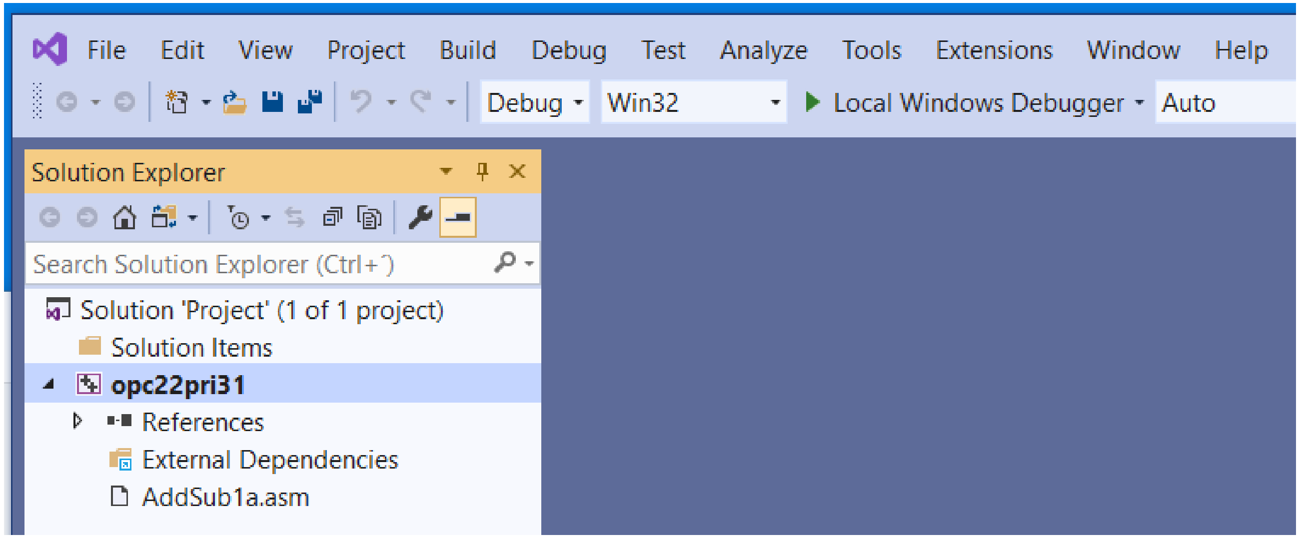
Característica del proyecto

- El fólder *PrjTtVS19rmt*, no lleva archivo fuente “.asm”:
- Hay que agregarlo al Proyecto de manera “virtual”, siendo que el archivo “.asm” se encuentra en otro folder, en este caso en el folder *asmFilesa*



Agregado de un archivo “.asm”

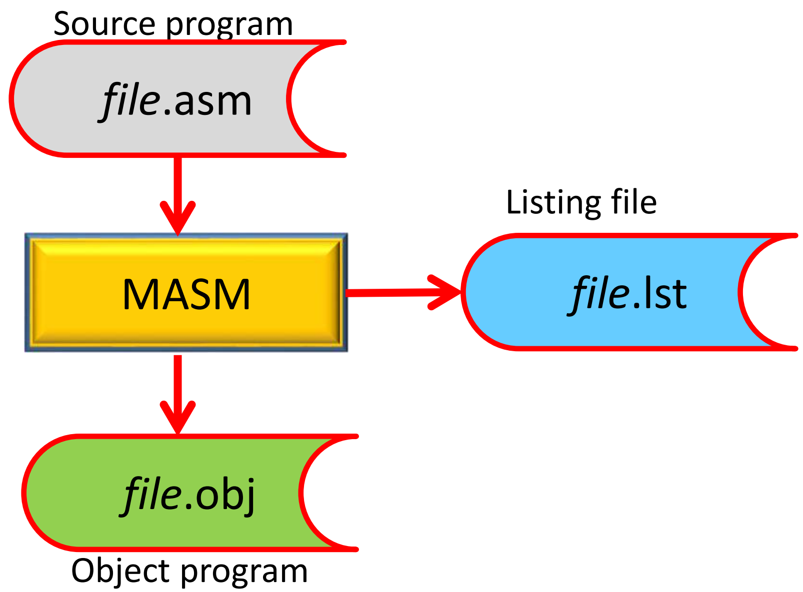
- Seleccione *AddSub1.asm*:
  - Arrastrelo, y depositélo sobre *opc22pri31* en VS, o
  - Haga copy, y paste sobre *opc22pri31* en VS
- Apareciendo dicho archive como sigue:



Ensamblado (Assembling)

Ensamblado es la actividad, llevada a cabo por el **Ensamblador (ml.exe)**, de traducir el archivo de entrada (.asm, programa fuente en Lenguaje Ensamblador) en un programa en Lenguaje Máquina, puesto en el archivo de salida (.*obj*, programa objeto).

El archivo de listado (.*lst*) es opcional.



Ensamble del programa

- Seleccione la línea de *opc22pri31*
- Seleccione
- Build > Build opc22pri31, o
- Build > Rebuild opc22pri31

- En la ventana de *Output* apareceran publicados lo errores en caso de existir.

Archivo *opc22pri31.lst*

- Ubicado en el folder clsAKasm\PrjTtVS19rmt
- Se puede abrir con Bloq de Notas o Notepad++
- *¿Que contiene?* _____

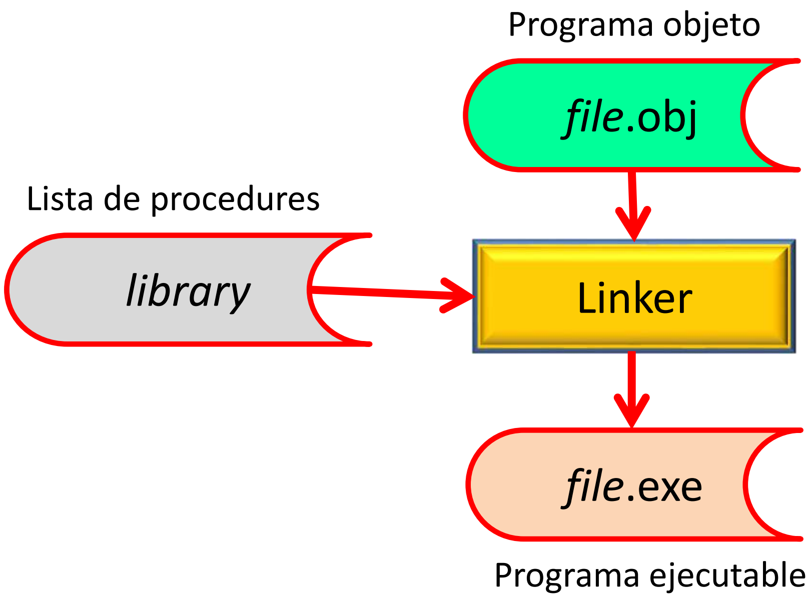
Archivo *.lst*

- Muestra la información que como el programa *.asm* fue ensamblado.
- Contiene
  - código fuente
  - direcciones, offsets
  - código objeto (languaje máquina)
  - nombres de los segmentos
  - símbolos (variables, procedimientos, y constantes)

Ligado (Linking), lanzado por el ensamble

Ligado o vinculado , es la actividad, desarrollada por el **Ligador (link.exe)**, de tomar el archivo objeto (.*obj*), revisando si el programa objeto contiene cualquier llamada a procedimientos o funciones de librería (del Sistema).

El ligador, agrega cualesquier procedimeinto o función de la librería, al programa ejecutable (.*exe*). El archivo ejecutable contiene el *program* a ser ejecutado por el Sistema Operativo.



Running

Running, or the act where the CPU executes the executable program, first, the operating system ***loader*** utility reads the executable file into memory and branches the CPU to the program's starting address, and the program begins to execute.

Command to run the executable program:

file.exe

Ejecución del programa 1

- Ubicarse en el folder clsAKasm\PrjTtVS19rmt\Debug
- Doble click sobre *Console.bat* abriéndose la ventana, desde la cual prodra ejecutar el programa *opc22pri31.exe*

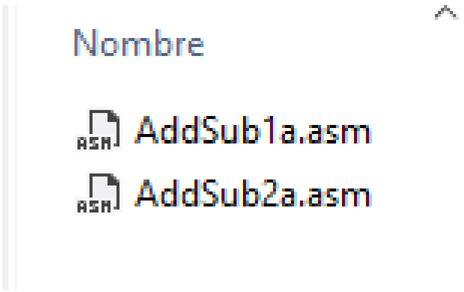
Ejecución del programa 2

"Console to exec"

```
113dac25b
Microsoft Windows [Versión 10.0.19044.2006]
(c) Microsoft Corporation. Todos los derechos reservados.

E:\jubn\OPC\AD2022\clsB0\PrjTtVS19rmt\Debug>opc22pri31.exe

  EAX=00010000  EBX=002A0000  ECX=0040100A  EDX=0040100A
  ESI=0040100A  EDI=0040100A  EBP=0019FF80  ESP=0019FF74
  EIP=00403666  EFL=00000246  CF=0  SF=0  ZF=1  OF=0  AF=0  PF=1


  EAX=00050000  EBX=002A0000  ECX=0040100A  EDX=0040100A
  ESI=0040100A  EDI=0040100A  EBP=0019FF80  ESP=0019FF74
  EIP=00403670  EFL=00000206  CF=0  SF=0  ZF=0  OF=0  AF=0  PF=1


  EAX=00030000  EBX=002A0000  ECX=0040100A  EDX=0040100A
  ESI=0040100A  EDI=0040100A  EBP=0019FF80  ESP=0019FF74
  EIP=0040367A  EFL=00000206  CF=0  SF=0  ZF=0  OF=0  AF=0  PF=1


E:\jubn\OPC\AD2022\clsB0\PrjTtVS19rmt\Debug>
```

Visualización y edición del “.asm”

- Desde el Bloq de notas o Notepad++, o
- Desde el mismo IDE del VS,
- Tenga mucho cuidado, al modificarlo, de verificar que se guarde adecuadamente.



Para finalizar

- Cerrar la ventana de *Console.bat*
- Remover el archive *AddSub#.asm* del Proyecto con
- seleccionar *AddSub#.asm*, con botón derecho del ratón, y seleccionar *Exclude from proyect*
- Volver a ensamblar *Build > Rebuild opc22pri31* para que el Proyecto vuelva al estado inicial.

Referencias

- Capítulos: Irvine, Kip R. Assembly Language for x86 Processors.
- Notas de Ramón Ríos, 07
- Agosto diciembre, 2023

ORGANIZACIÓN Y PROGRAMACIÓN DE COMPUTADORAS

OPC

A – D 2023

Instruction Format

- x86 instruction format
  - [label:] [**mnemonic [*operands***]] [;comments]
- mnemonic [operands]
  - mnemonic
  - mnemonic *source*
  - mnemonic *destination*
  - mnemonic *destination*, *source*
  - mnemonic *destination*, *source-1* , *source-2*

Operand Types

- ***Imm*-ediate** – a constant integer (8, 16, or 32 bits)
  – value is encoded within the instruction
- ***Reg*-ister** – the name of a register (8, 16, or 32 bits)
  – register name is converted to a number and encoded within the instruction
- ***Mem*-ory** – reference to a location in memory (8, 16, or 32 bits)
  – memory address is encoded within the instruction, or a register holds the address of a memory location

Instruction Operand Notation

Operand	Description
reg8	8-bit general-purpose register: AH, AL, BH, BL, CH, CL, DH, DL
reg16	16-bit general-purpose register: AX, BX, CX, DX, SI, DI, SP, BP
reg32	32-bit general-purpose register: EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP
reg	Any general-purpose register
sreg	16-bit segment register: CS, DS, SS, ES, FS, GS
imm	8-, 16-, or 32-bit immediate value
imm8	8-bit immediate byte value
imm16	16-bit immediate word value
imm32	32-bit immediate doubleword value
reg/mem8	8-bit operand, which can be an 8-bit general register or memory byte
reg/mem16	16-bit operand, which can be a 16-bit general register or memory word
reg/mem32	32-bit operand, which can be a 32-bit general register or memory doubleword
mem	An 8-, 16-, or 32-bit memory operand

Instruction Set

Data Transfer Instruction

MOV

(HLL =)

MOV Instruction

- Move from *source operand* to *destination operand*.
- Syntax:

MOV *destination, source*

; HLL, destination=source

General Operand-Variants of MOV

- MOV reg, reg
- MOV mem, reg
- MOV reg, mem
- MOV mem, imm
- MOV reg, imm

- No more than one *memory operand* permitted

Direct Memory Operands

- A direct memory operand is a named reference to storage in memory
- The named reference (*label*) is automatically dereferenced by the assembler

```
.DATA
var1 BYTE 10h

.CODE
MOV EAX,21h                    ; EAX = 00000021h
MOV ECX,EAX                    ; ECX = 00000021h
MOV AL,var1                    ; AL = 10h
MOV AL,[var1]                  ; AL = 10h
```

MOV Instruction 1

Explain why each of the following MOV statements are valid or invalid:

```
.DATA
count BYTE 100
wVal  WORD 2

.CODE
   MOV BL,count              ; ?
   MOV AX,wVal               ; ?
   MOV count,AL              ; ?
   MOV AL,wVal               ; ?
   MOV AX,count              ; ?
   MOV EAX,count             ; ?
```

MOV Instruction 2

- CS, EIP, and IP cannot be the destination
- No *immediate* to segment moves

Explain why each of the following MOV statements are valid or invalid:

```
.DATA
bVal   BYTE    100
bVal2  BYTE    ?
wVal   WORD    2
dVal   DWORD   5
.CODE
    MOV DS,45                     ?
    MOV ESI,wVal                  ?
    MOV EIP,dVal                  ?
    MOV 25,bVal                   ?
    MOV bVal2,bVal                ?
```

Direct-Offset Operands 1

A *constant offset* is added to a data label to produce an effective address (EA). The address is dereferenced to get the value inside its memory location.

```
.DATA
arrayB BYTE 10h,20h,30h,40h

.CODE
MOV AL,arrayB+1                 ; AL = 20h
MOV AL,[arrayB+1]               ; alternative notation
```

Q: Why doesn't arrayB+1 produce 11h?

Direct-Offset Operands 2

A constant offset is added to a data label to produce an effective address (EA). The address is dereferenced to get the value inside its memory location.

```
.DATA
arrayW  WORD 1000h,2000h,3000h
arrayD  DWORD 1,2,3,4
.CODE
MOV AX,[arrayW+2]               ; AX = 2000h
MOV AX,[arrayW+4]               ; AX = 3000h
MOV EAX,[arrayD+4]              ; EAX = 00000002h
```

```
; Will the following statements assemble?
MOV AX,[arrayW-2]           ; ??
MOV EAX,[arrayD+16]         ; ??
```

What will happen when they run?

ADD and SUB Instructions

- ADD, syntax:

  ADD *destination, source*

  ; HLL, destination= destination+source

- SUB, syntax:

  SUB *destination, source*

  ; HLL, destination= destination-source

Two operand instructions ADD, SUB

- ADD reg, reg SUB reg, reg
- ADD mem, reg SUB mem, reg
- ADD reg, mem SUB reg, mem
- ADD mem, imm SUB mem, imm
- ADD reg, imm SUB reg, imm

INC and DEC Instructions

- INC, syntax:

  INC *destination*

  ; HLL, destination= destination+1

- DEC, syntax:

  DEC *destination*

  ; HLL, destination= destination-1

One operand instructions INC, DEC

- INC reg DEC reg

- INC mem DEC mem

Addressing in Operands

- Direct Addressing
  - Register
  - Memory
  - Immediate

- And many more Addressing ...

Instruction Set

- Assembly Language of x86 Processors
- Appendix B
- **The x86 Instruction Set**.

Referencias

- Chapters: Irvine, Kip R. Assembly Language for x86 Processors.
- Notas de Ramón Ríos. 07
- Agosto - diciembre, 2023

ORGANIZACIÓN Y PROGRAMACIÓN DE COMPUTADORAS

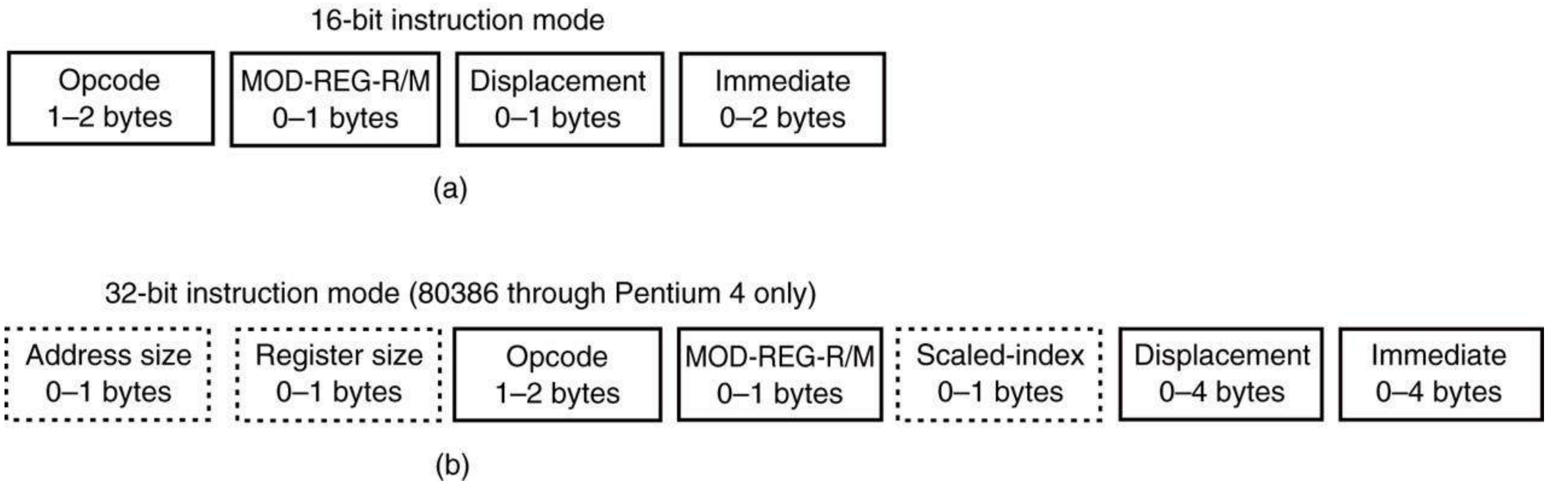
OPC

A – D 2023

Machine Language

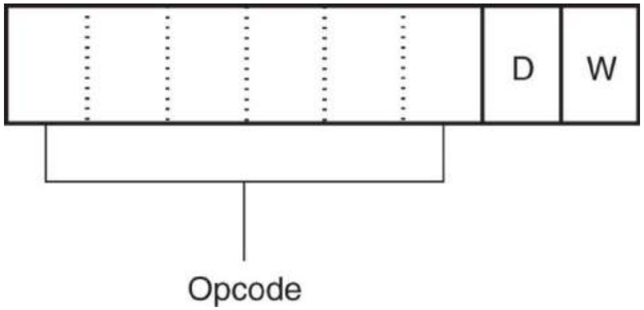
- Native *binary code* microprocessor instructions vary in length from 1 to 13 Bytes
- Over 100,000 variations of machine language instructions. There is no complete list of these variations
- Some bits in a machine language instruction are given (*opcode*); remaining bits are determined for each variation of the instruction: *byte*, *word* or *dword operands*.

Instruction Formats



- In the *Real* mode (DOS over 8086), 80386 and above assume all instructions are 16-bit mode instructions.
- In *Protected* mode (Windows & Linux), the upper byte of the descriptor contains the bit that selects either the 16- or 32-bit instruction mode

Opcode field, Byte 1



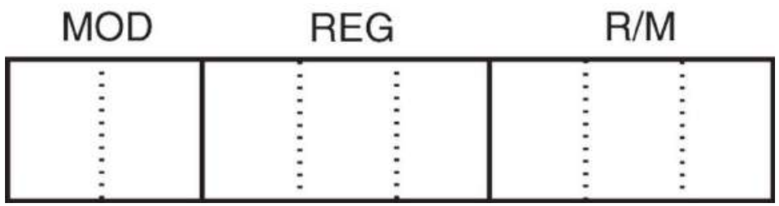
- Identifies the *operation* (mov, add, sub, …).
- Either 1 or 2 bytes long for instructions.
- *Binary* ***Opcode***: first 6 bits of the first byte.
- ***D*** bit, indicates the **direction** of the data flow:
  - Into or From, a Register
- ***W*** bit indicates whether the data are a byte or a word

32-bit instruction examples (1)

ASSEMBLY MEMORY ALLOCATION

- NOP ;1 Byte
  - 1st Byte: Opcode&D&W

[Opcode field, Byte 2]



- MOD field: *addressing mode code*
- REG field:
  - *register* code, destination/source operand
- R/M field:
  - *Register*, *memory* or *immediate* operand
  - If *register*: *register* code, source
  - If *memory*: code and address, destination/source
  - If *immediate*: code and data, source

32-bit instruction examples (1)

ASSEMBLY MEMORY ALLOCATION

- NOP ;1 Byte
  - 1st Byte: Opcode&D&W

- INC EAX ;2 Bytes
  - 1st Byte: Opcode&D&W
  - 2nd Byte: MOD&REG&R/M

Instruction: [Bytes 3, 4, …]

- These Bytes exist when R/M field is:
  - a Memory operand (Displacement, Offset), or
  - an Immediate operand

32-bit instruction examples (3)

ASSEMBLY MEMORY ALLOCATION

- MOV *alfa*, EBX ;6 Bytes
  - 1st Byte: Opcode&D&W
  - 2nd Byte: MOD&REG&R/M
  - 3rd, 4th, 5th, 6th Bytes: memory address of *alfa*

32-bit instruction examples (3)

ASSEMBLY MEMORY ALLOCATION

- MOV ECX, EDX ; Bytes? ___
  - 1st Byte: Opcode&D&W
  - 2nd Byte: MOD&REG&R/M
  - ?

- MOV alfa, 34h ; Bytes? ___
  - 1st Byte: Opcode&D&W
  - 2nd Byte: MOD&REG&R/M
  - ?

Referencias

- Chapters: Brey, Barry B., The Intel Microprocessors.
- Notas de Ramón Ríos. 08.
- Agosto – diciembre, 2023

ORGANIZACIÓN Y PROGRAMACIÓN DE COMPUTADORAS

OPC

A – D 2023

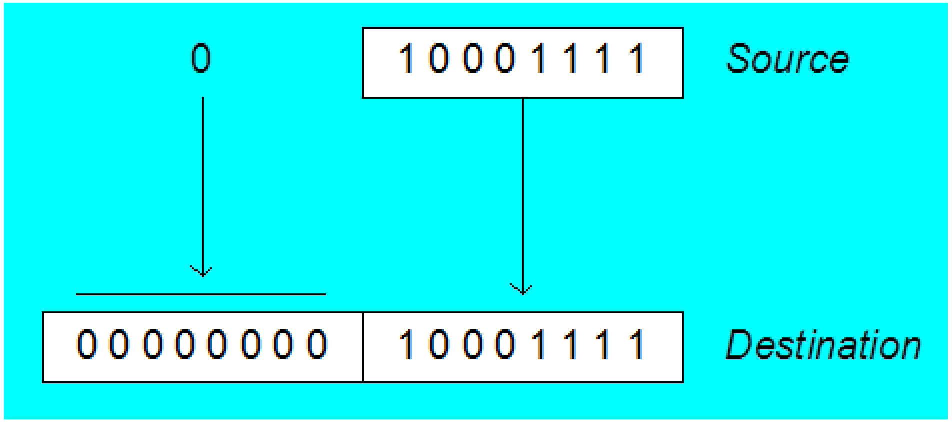
Instruction Set

Data Transfer Instructions (cont…)

Besides MOV

Zero Extension

When you copy a smaller value into a larger destination, the MOVZX instruction fills (extends) the upper half of the destination with zeros.



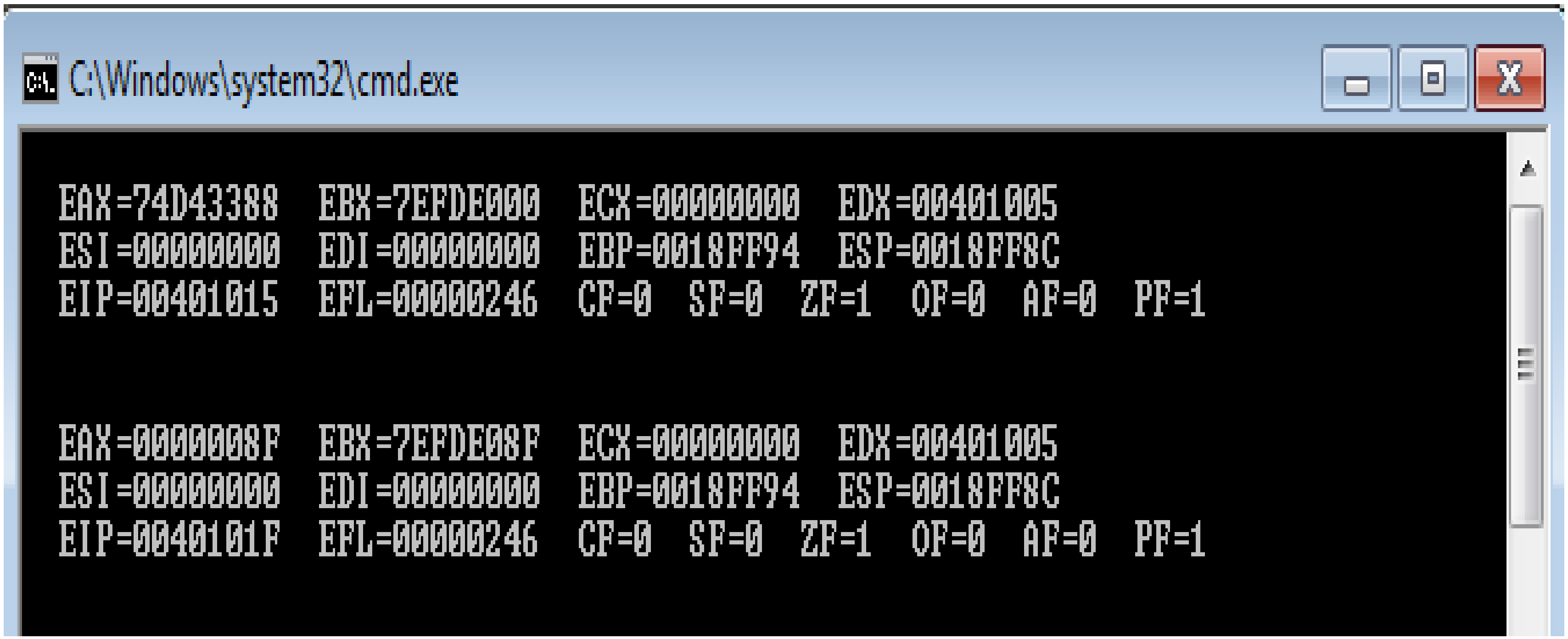

```
MOV BL,10001111b
MOVZX AX,BL                    ; zero-extension
```

The destination must be a register.

General Variants of MOVZX

- MOVZX reg32, reg/mem8
- MOVZX reg32, reg/mem16
- MOVZX reg16, reg/mem8

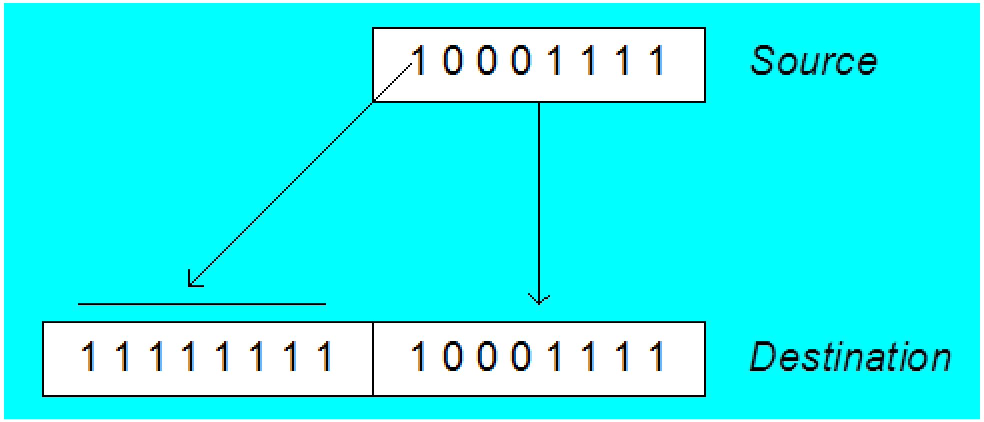
MOVZX with EAX register




```
CALL DumpRegs
MOV BL,10001111b        ; chart Zero Extension BL=__h
MOVZX EAX,BL            ; showing MOVZX with EAX register
CALL DumpRegs
```

Sign Extension

The MOVSX instruction fills the upper half of the destination with a copy of the source operand's sign bit.




```
MOV BL,10001111b
MOVSX AX,BL                    ; sign extension
```

The destination must be a register.

General Variants of MOVSX

- MOVSX reg32, reg/mem8
- MOVSX reg32, reg/mem16
- MOVSX reg16, reg/mem8

XCHG Instruction

XCHG exchanges the values of two operands. At least one operand must be a register. *No immediate operands are permitted*.

```
.DATA
var1 WORD 1000h
var2 WORD 2000h
.CODE
; EAX contains 12345678h, EBX has ABCDEF01h.
XCHG var1,BX        ; var1=____h, BX=____h, EBX=________h
XCHG EAX,EBX        ; EAX=________, EBX=________
XCHG AX,BX          ; AX=____, BX=____
XCHG AH,AL          ; AH=__, AL=____
XCHG var1,var2      ; var1=____h, var2=____h
```

General Variants of XCHG

- XCHG reg, reg
- XCHG reg, mem
- XCHG mem, reg

Your turn. . .

Write a program that rearranges the values of three *doubleword* values in the following array as: 3, 1, 2.

```
.data
arrayD DWORD 1,2,3
.code
```

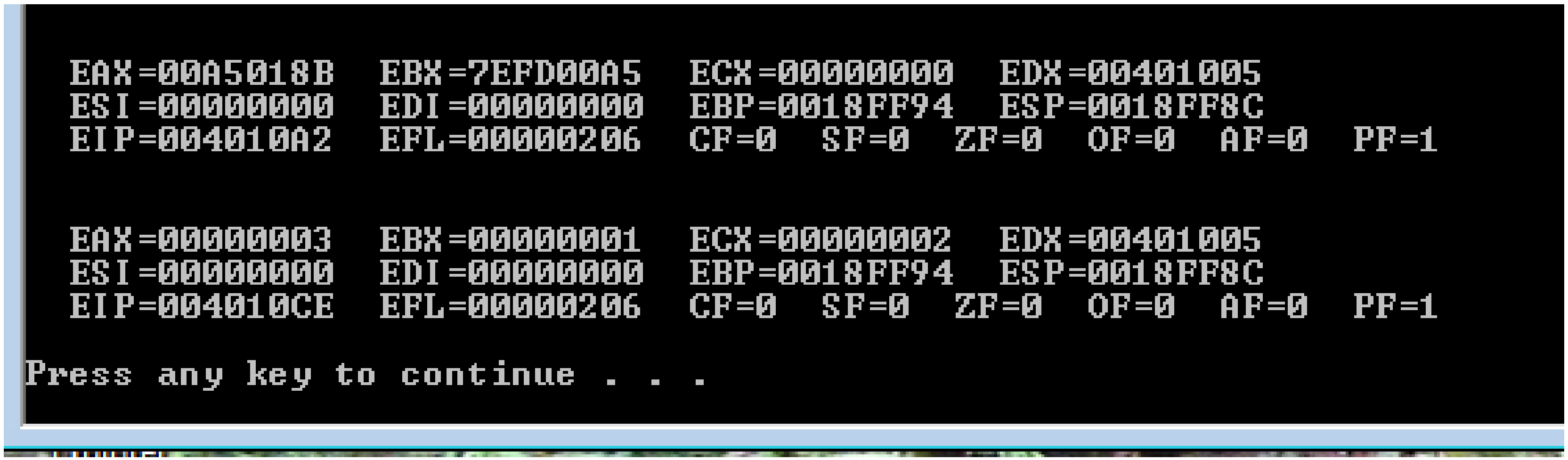
- Step1:

```
MOV EAX, arrayD
XCHG . . .
```

- Step 2:

```
XCHG . . .
MOV  . . .
```

XCHG Example




```
; xchg examples
CALL DumpRegs
MOV EAX, arrayD
XCHG EAX, [arrayD+4]
XCHG EAX, [arrayD+8]
MOV arrayD, EAX
. . .
```

```
MOV EAX, [arrayD]
MOV EBX, [arrayD+4]
MOV ECX, [arrayD+8]
CALL DumpRegs
```

Evaluate this . . .

```
.data
myBytes BYTE 80h,66h,0A5h
```

- How about the following code. Is anything missing?

```
; EAX has 00A5668Bh, EBX has 7EFDE08Fh
MOVZX AX, myBytes
MOV   BL,[myBytes+1]
ADD   AX, BX
MOV   BL,[myBytes+2]
ADD   AX, BX                ; AX = ?
```

Evaluate this . . . (cont)

```
EAX=00A5668B  EBX=7EFDE08F  ECX=00000000  EDX=00401005
ESI=00000000  EDI=00000000  EBP=0018FF94  ESP=0018FF8C
EIP=00401060  EFL=00000287  CF=1  SF=1  ZF=0  OF=0  AF=0  PF=1

EAX=00A5C18B  EBX=7EFDE0A5  ECX=00000000  EDX=00401005
ESI=00000000  EDI=00000000  EBP=0018FF94  ESP=0018FF8C
EIP=0040107F  EFL=00000287  CF=1  SF=1  ZF=0  OF=0  AF=0  PF=1
```

```
CALL DumpRegs
; MOVZX examples     EAX: 00A5668Bh, EBX: 7EFDE08Fh
MOVZX  AX, myBytes                   ; EAX:  00A50080h
MOV   BL, [myBytes+1]              ; EBX: 7EFDE066h
ADD   AX, BX                          ; EAX:  00A5E0E6h
MOV   BL, [myBytes+2]              ; EBX: 7EFDE0A5h
ADD   AX, BX     ; AX =                EAX:  00A5C18Bh
CALL DumpRegs
```

Evaluate this . . . (cont)

```
EAX=00A5C18B  EBX=7EFDE0A5  ECX=00000000  EDX=00401005
ESI=00000000  EDI=00000000  EBP=0018FF94  ESP=0018FF8C
EIP=0040107F  EFL=00000287  CF=1  SF=1  ZF=0  OF=0  AF=0  PF=1

EAX=00A5018B  EBX=7EFD00A5  ECX=00000000  EDX=00401005
ESI=00000000  EDI=00000000  EBP=0018FF94  ESP=0018FF8C
EIP=004010A2  EFL=00000206  CF=0  SF=0  ZF=0  OF=0  AF=0  PF=1
```

```
CALL DumpRegs
; MOVZX examples      EAX: 00A5668Bh, EBX: 7EFDE08Fh
MOVZX  AX,myBytes                  ; EAX:  00A50080h
MOV  BX,0                          ; EBX: 7EFD0000h
MOV   BL,[myBytes+1]               ; EBX: 7EFD0066h
ADD   AX,BX                        ; EAX:  00A500E6h
MOV   BL,[myBytes+2]               ; EBX: 7EFD00A5h
ADD   AX,BX      ; AX =              EAX:  00A5018Bh
CALL DumpRegs
```

Referencias

- Chapters: Irvine, Kip R. Assembly Language for x86 Processors.
- Notas de Ramón Ríos.
- Agosto – diciembre, 2023

ORGANIZACIÓN Y PROGRAMACIÓN DE COMPUTADORAS

OPC

A – D 2023

Program structure

```
TITLE primer
    ;Descripcion del programa, fecha, versión

INCLUDE Irvine32.inc    ;libreria de funciones; falta agregar más

.DATA
    ; directivas de almacenamiento y tipos de datos

.CODE
main PROC           ; Inicia el procedimiento main

    ; instrucciones, mnemónicos

main ENDP           ; Termina el procedimiento principal

END main            ; Termina el area de Ensamble
```

Irvine32 Library Procedures

- INCLUDE Irvine32.inc ; chapter 5
- These procedures are for I/Os

- DumpRegs
- DumpMem

- ReadInt
- ReadHex

- WriteInt
- WriteHex

- WriteString
- Crlf
- ReadString

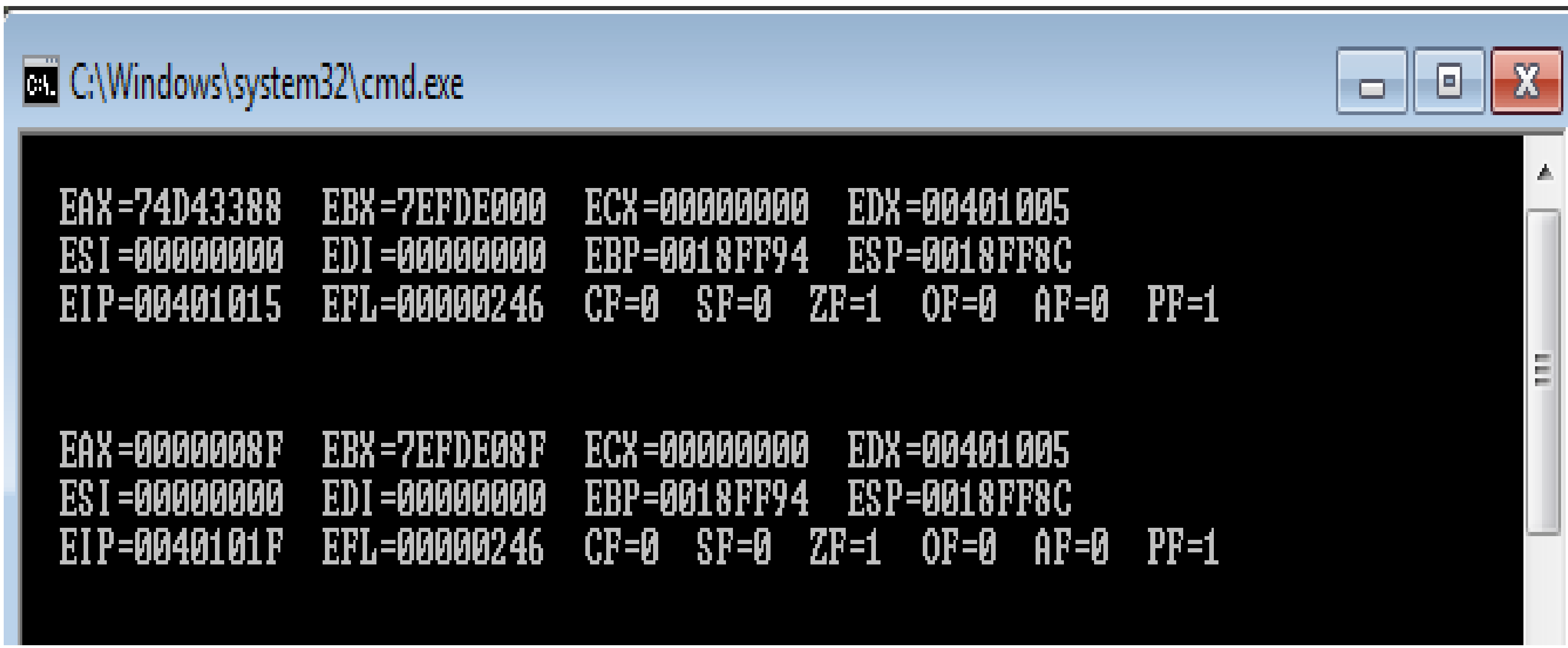
DumpRegs

- DumpRegs
  - It desplays all the general registers and the flags.

- Sample call

```
.CODE
MOV ECX, 0
CALL  DumpRegs

MOV EAX, 8Fh
CALL DumpRegs
```

DumpRegs example



DumpMem

- DumpMem
  - It writes a range/block of memory to the console window in hexadecimal.
  - Pass, in ESI the starting address of the block, in ECX the number of units or elements, and in EBX the unit size (1 : byte, 2 : word, 4 : doubleword).
- Sample call

```
.DATA
    array DWORD 11, 12, 13

.CODE
    MOV  ESI, OFFSET array
    MOV ECX, 3              ; ? _________
    MOV EBX, 4              ; ? _________
   CALL  DumpMem
```

WriteInt

- WriteInt
  - Writes a 32-bit signed integer to the console window in decimal format with a leading sign and no leading zeros.
  - Pass the integer into EAX.
- Sample

```
.DATA
    valInt  SDWORD -317432

.CODE
   MOV  EAX, valInt
   CALL  WriteInt
   CALL Crlf

   MOV  EAX, 235896
   CALL  WriteInt
```

WriteHex

- WriteHex
  - Writes a 32-bit unsigned integer to the console window in 8-digit hexadecimal format.
  - Leading zeroes are inserted if necessary.
  - Pass the integer into EAX.
- Sample

```
.DATA
    valHex DWORD 6ABCh

.CODE
   MOV  EAX, valHex
   CALL  WriteHex
   CALL Crlf

   MOV  EAX, 8EF9h
   CALL  WriteHex
```

WriteString

- WriteString
  - It writes a null-terminated string to the console window.
  - Pass, in EDX register, the string’s offset.
- Sample call

```
.DATA
    line1  BYTE  “Enter the data: “, 0

.CODE
    MOV  EDX, OFFSET line1
    CALL  WriteString
```

Crlf

- Crlf
  - It advances the cursor, inwindow console, to the beginning of the next line.
  - It writes down a string containing the ASCII characters 0Dh and 0Ah.
  - 0Dh is the code of CR (Carriage Return, Enter) and 0Ah is the code of LF (Line Feed).
  - Check out the appendices 8-bit ASCII Code tables.
- Sample call

```
.DATA
    line1  BYTE  “Enter the data: “, 0

.CODE
    MOV  EDX, OFFSET line1
    CALL  WriteString
    CALL Crlf
```

ReadInt

- ReadInt
  - Reads a 32-bit signed integer from the keyboard and returns the value in EAX register.
  - Can be typed an optional leading plus or minus sign.
  - It sets the Overflow flag and display an error message if the value cannot be represented as a 32-bit signed integer (-2,147,483,648 to + 2,147,483,647)
- Sample

```
.DATA
    valInt  SDWORD ?

.CODE
   CALL  ReadInt
   MOV  valInt, EAX
```

ReadHex

- ReadHex
  - Reads a 32-bit hexadecimal integer from the keyboard and returns the value in EAX register.
  - No error checking is performed for invalid characteres.
  - Can use both uppercase letters and lowercase letters for the digits A through F.
- Sample

```
.DATA
    valHex  DWORD ?

.CODE
   CALL  ReadHex
   MOV  valHex, EAX
```

ReadString

- ReadString
  - It reads a string from the keyboard, stopping when the user types the ENTER key.
  - Pass the buffer's offset in EDX register.
  - Set ECX to the maximum number of characters that the user can type, plus 1 to save space for the terminting null byte.
  - It returns, in EAX, the count of the number of characters typed by the user.
- Sample

```
.DATA
    bufferR  BYTE 81 DUP(0)    ; 80 characters plus 0 (terminator)
    charCountR  DWORD ?
.CODE
    MOV  EDX, OFFSET bufferR
    MOV  ECX, 81
    CALL  ReadString
    MOV  charCountR, EAX
```

8-bit ASCII Codes: 0-127

Dec	Hx	Oct	Char	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr	Dec	Hx	Oct	Html	Chr
0	0	000	NUL (null)	32	20	040		Space	64	40	100	@	@	96	60	140	`	`
1	1	001	SOH (start of heading)	33	21	041	!	!	65	41	101	A	A	97	61	141	a	a
2	2	002	STX (start of text)	34	22	042	"	"	66	42	102	B	B	98	62	142	b	b
3	3	003	ETX (end of text)	35	23	043	#	#	67	43	103	C	C	99	63	143	c	c
4	4	004	EOT (end of transmission)	36	24	044	$	$	68	44	104	D	D	100	64	144	d	d
5	5	005	ENQ (enquiry)	37	25	045	%	%	69	45	105	E	E	101	65	145	e	e
6	6	006	ACK (acknowledge)	38	26	046	&	&	70	46	106	F	F	102	66	146	f	f
7	7	007	BEL (bell)	39	27	047	'	'	71	47	107	G	G	103	67	147	g	g
8	8	010	BS (backspace)	40	28	050	(	(	72	48	110	H	H	104	68	150	h	h
9	9	011	TAB (horizontal tab)	41	29	051	)	)	73	49	111	I	I	105	69	151	i	i
10	A	012	LF (NL line feed, new line)	42	2A	052	*	*	74	4A	112	J	J	106	6A	152	j	j
11	B	013	VT (vertical tab)	43	2B	053	+	+	75	4B	113	K	K	107	6B	153	k	k
12	C	014	FF (NP form feed, new page)	44	2C	054	,	,	76	4C	114	L	L	108	6C	154	l	l
13	D	015	CR (carriage return)	45	2D	055	-	-	77	4D	115	M	M	109	6D	155	m	m
14	E	016	SO (shift out)	46	2E	056	.	.	78	4E	116	N	N	110	6E	156	n	n
15	F	017	SI (shift in)	47	2F	057	/	/	79	4F	117	O	O	111	6F	157	o	o
16	10	020	DLE (data link escape)	48	30	060	0	0	80	50	120	P	P	112	70	160	p	p
17	11	021	DC1 (device control 1)	49	31	061	1	1	81	51	121	Q	Q	113	71	161	q	q
18	12	022	DC2 (device control 2)	50	32	062	2	2	82	52	122	R	R	114	72	162	r	r
19	13	023	DC3 (device control 3)	51	33	063	3	3	83	53	123	S	S	115	73	163	s	s
20	14	024	DC4 (device control 4)	52	34	064	4	4	84	54	124	T	T	116	74	164	t	t
21	15	025	NAK (negative acknowledge)	53	35	065	5	5	85	55	125	U	U	117	75	165	u	u
22	16	026	SYN (synchronous idle)	54	36	066	6	6	86	56	126	V	V	118	76	166	v	v
23	17	027	ETB (end of trans. block)	55	37	067	7	7	87	57	127	W	W	119	77	167	w	w
24	18	030	CAN (cancel)	56	38	070	8	8	88	58	130	X	X	120	78	170	x	x
25	19	031	EM (end of medium)	57	39	071	9	9	89	59	131	Y	Y	121	79	171	y	y
26	1A	032	SUB (substitute)	58	3A	072	:	:	90	5A	132	Z	Z	122	7A	172	z	z
27	1B	033	ESC (escape)	59	3B	073	;	;	91	5B	133	[	[	123	7B	173	{	{
28	1C	034	FS (file separator)	60	3C	074	<	<	92	5C	134	\	\	124	7C	174			|
29	1D	035	GS (group separator)	61	3D	075	=	=	93	5D	135	]	]	125	7D	175	}	}
30	1E	036	RS (record separator)	62	3E	076	>	>	94	5E	136	^	^	126	7E	176	~	~
31	1F	037	US (unit separator)	63	3F	077	?	?	95	5F	137	_	_	127	7F	177	&#127;	DEL

Source: www.LookupTables.com

8-bit ASCII Codes: 128-255

128	Ç	144	É	160	á	176	░	192	└	208	╨	224	α	240	≡
129	ü	145	æ	161	í	177	▒	193	┴	209	╤	225	ß	241	±
130	é	146	Æ	162	ó	178	▓	194	┬	210	╥	226	Γ	242	≥
131	â	147	ô	163	ú	179	│	195	├	211	╙	227	π	243	≤
132	ä	148	ö	164	ñ	180	┤	196	─	212	╘	228	Σ	244	⌠
133	à	149	ò	165	Ñ	181	╡	197	┼	213	╒	229	σ	245	⌡
134	å	150	û	166	ª	182	╢	198	╞	214	╓	230	µ	246	÷
135	ç	151	ù	167	º	183	╖	199	╟	215	╫	231	τ	247	≈
136	ê	152	ÿ	168	¿	184	╕	200	╚	216	╪	232	Φ	248	°
137	ë	153	Ö	169	⌐	185	╣	201	╔	217	┘	233	Θ	249	∙
138	è	154	Ü	170	¬	186	║	202	╩	218	┌	234	Ω	250	·
139	ï	155	¢	171	½	187	╗	203	╦	219	█	235	δ	251	√
140	î	156	£	172	¼	188	╝	204	╠	220	▄	236	∞	252	^{n}
141	ì	157	¥	173	¡	189	╜	205	═	221	▌	237	φ	253	^{2}
142	Ä	158	₧	174	«	190	╛	206	╬	222	▐	238	ε	254	■
143	Å	159	ƒ	175	»	191	┐	207	╧	223	▀	239	∩	255	

Source: www.LookupTables.com

Review these procedures

Chapter 5 (5.4.2, 5.4.3)

- Clrscr
- ReadChar, ReadDec, ReadKey
- WriteBin, WriteBinB
- WriteChar, WriteDec, WriteHexB

Referencias

- Capítulos: Irvine, Kip R. Assembly Language for x86 Processors.
- Notas de Ramón Ríos
- Agosto - diciembre 2023

ORGANIZACIÓN Y PROGRAMACIÓN DE COMPUTADORAS

OPC

A – D 2023

Instruction Set

Addition and Subtraction

add, sub, inc, dec, neg

Focus: Carry Flag

ADD and SUB Instructions

- ADD destination, source
  - *destination* $\leftarrow$ *destination* + source

- SUB destination, source
  - *destination* $\leftarrow$ *destination* – source

- Same operand rules as for the MOV instruction

Two operand instructions ADD, SUB

- ADD reg, reg SUB reg, reg
- ADD mem, reg SUB mem, reg
- ADD reg, mem SUB reg, mem
- ADD mem, imm SUB mem, imm
- ADD reg, imm SUB reg, imm

ADD and SUB Examples

```
.DATA
    var1 DWORD 10000h
    var2 DWORD 20000h
.CODE
        MOV EAX, var1       ; __________ h
        ADD EAX, var2       ; __________ h
        ADD AX, 0FFFFh      ; __________ h
        ADD EAX, 1          ; __________ h
        SUB AX, 1           ; __________ h
```

INC and DEC Instructions

- Add 1, Subtract 1 from destination operand
  - operand may be register or memory

- INC *destination*
    - *destination* $\leftarrow$ *destination* + 1

- DEC *destination*
    - *destination* $\leftarrow$ *destination* – 1

One operand instructions INC, DEC

- INC reg   INC mem
- DEC reg   DEC mem

INC and DEC Examples

```
.DATA
    myWord  WORD 1000h
    myDword DWORD 10000000h
.CODE
        INC myWord              ; _________ h
        DEC myWord              ; _________ h
        INC myDword             ; _________ h

        MOV  AX, 00FFh      ; AX= ____ h
        INC  AX             ; AX= ____ h
        MOV  AX, 00FFh      ; AX= ____ h
        INC  AL             ; AL= __h  AX= ____ h
```

Your turn...

Show the value of the destination operand after each of the following instructions executes:

```
.DATA
    myByte BYTE 0FFh, 0

.CODE
    MOV AL, myByte            ; AL= __h
    MOV AH,[myByte+1]         ; AH= __h, AX= ____h
    DEC AH                    ; AH= __h, AX= ____h
    INC AL                    ; AL= __h, AX= ____h
    DEC AX                    ; AX= ____h
```

One operand instruction NEG

- NEG reg NEG mem

NEG (negate) Instruction

The processor implements ***NEG operand*** using the following internal operation:

```
operand = 0 – operand
```

Two's complement operation.

```
.DATA
    valB SBYTE 1,0
    valC SBYTE -128

.CODE
    NEG valB            ; valB = ________
    NEG [valB + 1]      ; valB+1 = ________
    NEG valC            ; valC = ________
```

NEG (negate) Instruction

Reverses the sign of an operand. Operand can be a register or memory operand.

```
.DATA
   valB SBYTE -1
   valW SWORD +32767
.CODE
    MOV AL, valB                 ; AL = ____
    NEG AL                       ; AL = ____
    NEG valW                     ; valW = ________
```

Suppose AX contains –32,768 and we apply NEG to it. Will the result be valid?

Implementing Arithmetic Expressions

High Level Languages compilers translate mathematical expressions into assembly language. Recall precedence order.

For example:

```
Rval = -Xval + (Yval – Zval)
```

```
Do not modify Xval, Yval and Zval contents.
```

```
.DATA
   Rval DWORD ?
   Xval DWORD 26
   Yval DWORD 30
   Zval DWORD 40
.CODE
   MOV . . .
```

Your turn...

Translate the following expression into assembly language.
Do not permit Xval, Yval, or Zval to be modified:

```
Rval = Xval - (-Yval + Zval)
```

Assume that all values are signed doublewords.

```
MOV ...
```

Referencias

- Capítulos: Irvine, Kip R. Assembly Language for x86 Processors.
- Notas de Ramón Ríos
- Ago-Dic 2023

ORGANIZACIÓN Y PROGRAMACIÓN DE COMPUTADORAS

OPC

A – D 2023

Operators in operands

Data-Related Operators in operands

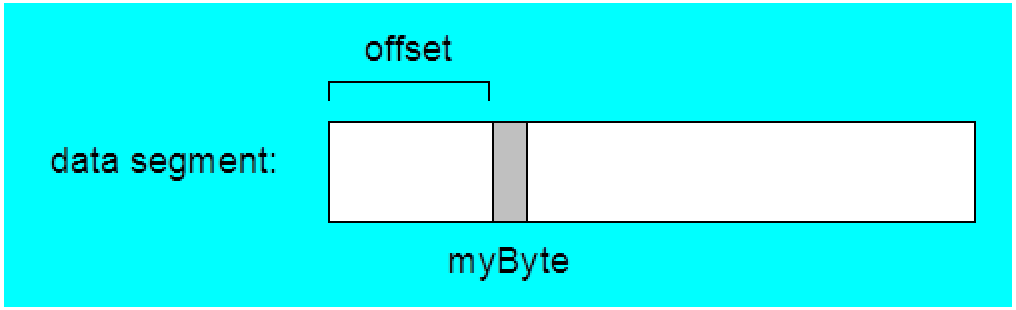
Data-Related Operators in **Operands**

Data-Related Operators are a kind of Directives, that are not executable instructions, instead they are only assembled by the assemblers (.CODE segment).

- OFFSET Operator
- TYPE Operator
- LENGTHOF Operator
- SIZEOF Operator
- and more …

OFFSET Operator

- OFFSET returns the *distance* in bytes, of a *label* from the beginning of its enclosing DATA segment
  - Protected mode: 32, 64 bits
  - Real mode: 16 bits



OFFSET Examples

Let's assume that the DATA segment begins at 00000000h:

```
.DATA
    BYTE 404000h DUP(?)
bVal BYTE ?
wVal WORD ?
dVal DWORD ?
dVal2 DWORD ?

.CODE
MOV ESI,OFFSET bVal        ; ESI = 0040____
MOV ESI,OFFSET wVal        ; ESI = 0040____
MOV ESI,OFFSET dVal        ; ESI = 0040____
MOV ESI,OFFSET dVal2       ; ESI = 0040____
```

Relating to C/C++

The value returned by OFFSET is a pointer. Compare the following code written for both C++ and assembly language:

```
// C++ version:


char array[1000];



char * p = array;
```

```
; Assembly language:

.DATA
array BYTE 1000 DUP(?)
    p DWORD ?

.CODE
MOV  ESI,OFFSET array
MOV p, ESI
```

TYPE Operator

The TYPE operator returns the *size*, in bytes, of a single element of a data declaration.

```
.DATA
var1 BYTE ?
var2 WORD ?
var3 DWORD ?
var4 QWORD ?

.CODE
MOV EAX, TYPE var1          ; 1
MOV EAX, TYPE var2          ; 2
MOV EAX, TYPE var3          ; _
MOV EAX, TYPE var4          ; _
```

LENGTHOF Operator

The LENGTHOF operator *counts* the number of elements (magnitude) in a single data declaration.

```
.DATA                                             LENGTHOF  label
byte1   BYTE 10,20,30                             ; 3
array1 WORD 30 DUP(?),0,0                         ; 32
array2 WORD 5 DUP(3 DUP(?))                       ; 15
array3 DWORD 1,2,3,4                              ; 4
digitStr BYTE "12345678",0                        ; 9

.CODE
MOV ECX, LENGTHOF array1                          ; 32
MOV EBX, LENGTHOF digitStr                        ; 9
ADD EBX, TYPE digitStr                            ; __
MOV EAX, TYPE array3                              ; __
```

SIZEOF Operator

The SIZEOF operator returns a *value* (Bytes) that is equivalent to multiplying *LENGTHOF by TYPE*.

SIZEOF label

```
.DATA
byte1  BYTE 10,20,30                    ; 3
array1 WORD 30 DUP(?),0,0               ; 64
array2 WORD 5 DUP(3 DUP(?))             ; 30
array3 DWORD 1,2,3,4                    ; 16
digitStr BYTE "12345678",0              ; 9

.CODE
MOV EAX,TYPE array1                     ; __
MOV EBX,LENGTHOF array1                 ; __
MOV ECX,SIZEOF array1                   ; __
```

Spanning Multiple Lines (1 of 2)

In the following example, *array1* identifies only the first WORD directive. Compare the values returned by LENGTHOF and SIZEOF here to those in the next slide:

```
.DATA
array1 WORD 10,20
       WORD 30,40
       WORD 50,60
Banderita . . .

.CODE
mov eax, LENGTHOF array1          ; 2
mov ebx, SIZEOF array1            ; 4
mov ecx, OFFSET Banderita         ; _
mov edx, OFFSET Banderita-array1  ; _
```

Spanning Multiple Lines (2 of 2)

A data Directive spans multiple lines if each line (except the last) ends with a comma. The LENGTHOF and SIZEOF operators include all lines belonging to the same Directive:

```
.DATA
array1 WORD 10,20,
   30,40,
   50,60

.CODE
MOV EAX,LENGTHOF array1          ; 6
MOV EBX,SIZEOF array1            ; __
```

Symbols, Symbolic Constant

- A symbol is an identifier.
    - ValorFinal, Fecha23

- Symbolic constant (*is not a label*)
    - Is an identifier associated with a 32/64-bit integer expression (or constant)
    - Syntax identifier = expression (“=“ *equal-sign directive*)

      ValorFinal = 452

      MOV EAX, ValorFinal (really MOV EAX, 452)

Symbols, Symbolic Constant - 2

- Symbols do not reserve storage.
- Symbols are not labels.
- Symbols only exist during the *time* while the Assembler is Assembling a program in Assembly Language.

Symbolic Constants

- May be redifined

  ValorFinal = 483

  MOV EAX, ValorFinal ; really MOV EAX, 483

  ValorFinal = 627

  MOV EAX, ValorFinal ; really MOV EAX, 627

- Why use Symbols? Clarifies the program
  - Better when you see *ValorFinal* instead of *483*

  EscKey = 27

  MOV AL, EscKey ; really MOV EAX, 27

Assembly Program Practice

EjerAP.doc

- Code it up ¡¡

Referencias

- Capítulos: Irvine, Kip R. Assembly Language for x86 Processors.
- Notas de Ramón Ríos
- Agosto - diciembre 2023

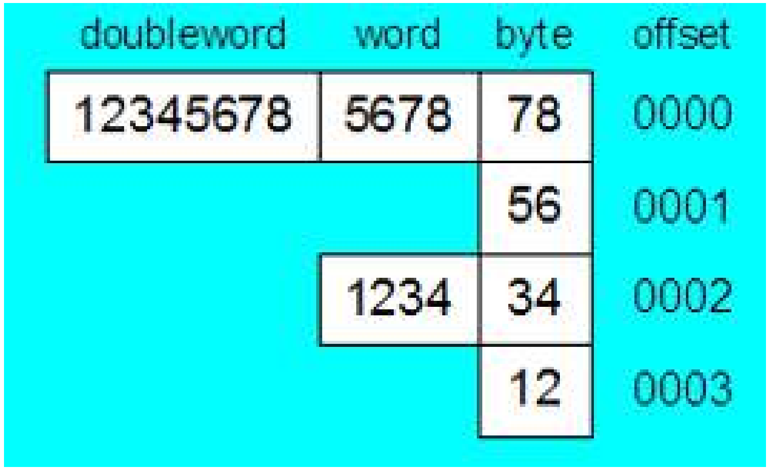
ORGANIZACIÓN Y PROGRAMACIÓN DE COMPUTADORAS

OPC

A – D 2023

Memory storage of values

```
.DATA
Alfa DWORD 12345678h
     ; How is the DWORD stored in memory?

; Other ways to storage the same values
Beta WORD __..._
Delta BYTE __..._
```



Mapping data values

- Mapping smaller data Registers inside longer data registers: e.g. AL, AH <>AX <> EAX, or, DL, DH <>DX <> EDX

- What if now we start to do data mapping with memory storage locations?

- Two ways for mapping:
- Label Directive / Data Segment
- “PTR” operator / Code Segment

LABEL Directive

- assigns a *label*, with *type* and *data value(s)*, the one to be mapped (e.g. *zeta*)
  – assigns an alternate *label-name* and *type* to an existing storage location, the mapping one
  – does *not allocate* any *storage* of its own. Symbol table?

```
.DATA
gamma   LABEL DWORD     ; no storage
epsilon LABEL WORD      ; no storage
zeta  BYTE 00h,10h,00h,20h    ; storage

.CODE
mov EAX, gamma              ; 20001000h
mov CX, epsilon             ; 1000h
mov DL, zeta                ; 00h
```

“type” PTR – Operand Operator

- ***“type”* PTR operator:**
  - **overrides the declared size (*“type”*) of an operand.**
  - **allows the *selection of some part* of a defined variable (*label, variable*).**
- It works in the section .CODE
- Operand operator (*“type”* PTR) that works at assembly time (like directives TYPE, LENGTHOF, etc.).
- Similar concept: HLL *casting*

“type” PTR - Operator Examples 1

```
.DATA
myDouble DWORD 12345678h

.CODE
MOV EAX, myDouble                       ; EAX =
MOV AX, myDouble                        ; error – why?

MOV AX, WORD PTR myDouble               ; loads 5678h
MOV WORD PTR myDouble, 4A9Bh            ; saves 4A9Bh

MOV AL, BYTE PTR  myDouble              ; AL =
MOV AL, BYTE PTR [myDouble+1]           ; AL =
MOV AL, BYTE PTR [myDouble+2]           ; AL =
MOV AL, BYTE PTR [myDouble+3]           ; AL =

MOV AX, WORD PTR  myDouble              ; AX =
MOV AX, WORD PTR [myDouble+2]           ; AX =
```

“type” PTR - Operator Examples 2

PTR can also be used to *combine elements of a smaller data* type and move them into a larger operand. The CPU will automatically consider the bytes in little-endian format.

```
.DATA
myBytes BYTE 12h,34h,56h,78h

.CODE
MOV AX, WORD PTR [myBytes]        ; AX =
MOV AX, WORD PTR [myBytes+2]      ; AX =
MOV EAX, DWORD PTR myBytes        ; EAX =
```

“type” PTR - Operator Examples 3

```
.DATA
varB BYTE 65h,31h,02h,05h
varW WORD 6543h,1202h
varD DWORD 12345678h

.CODE
MOV AX, WORD PTR [varB+2]          ; a. AX=
MOV BL, BYTE PTR varD              ; b. BL=
MOV BL, BYTE PTR [varW+2]          ; c. BL=
MOV AX, WORD PTR [varD+2]          ; d. AX=
MOV EAX, DWORD PTR varW            ; e. EAX=
```

Referencias

- Capítulos: Irvine, Kip R. Assembly Language for x86 Processors.
- Notas de referencia, Ramón Ríos, bxf
- Agosto - diciembre 2023