X86 INSTRUCTION SET

cdio
4.0

KHOA CÔNG NGHỆ THÔNG TIN
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN

fit@hcmus

REMIND

☐ CISC

☐ MIPS-32 bits operations

PREREQUITES

- ☐ Take a view on tutorial video
- ☐ Install NASM already

What will you learn?

☐ Inside a CPU Intel 8080/8086

☐ Memory organization

☐ Registers

☐ Instruction Format

☐ Data addressing modes

☐ Operations

☐ Procedure

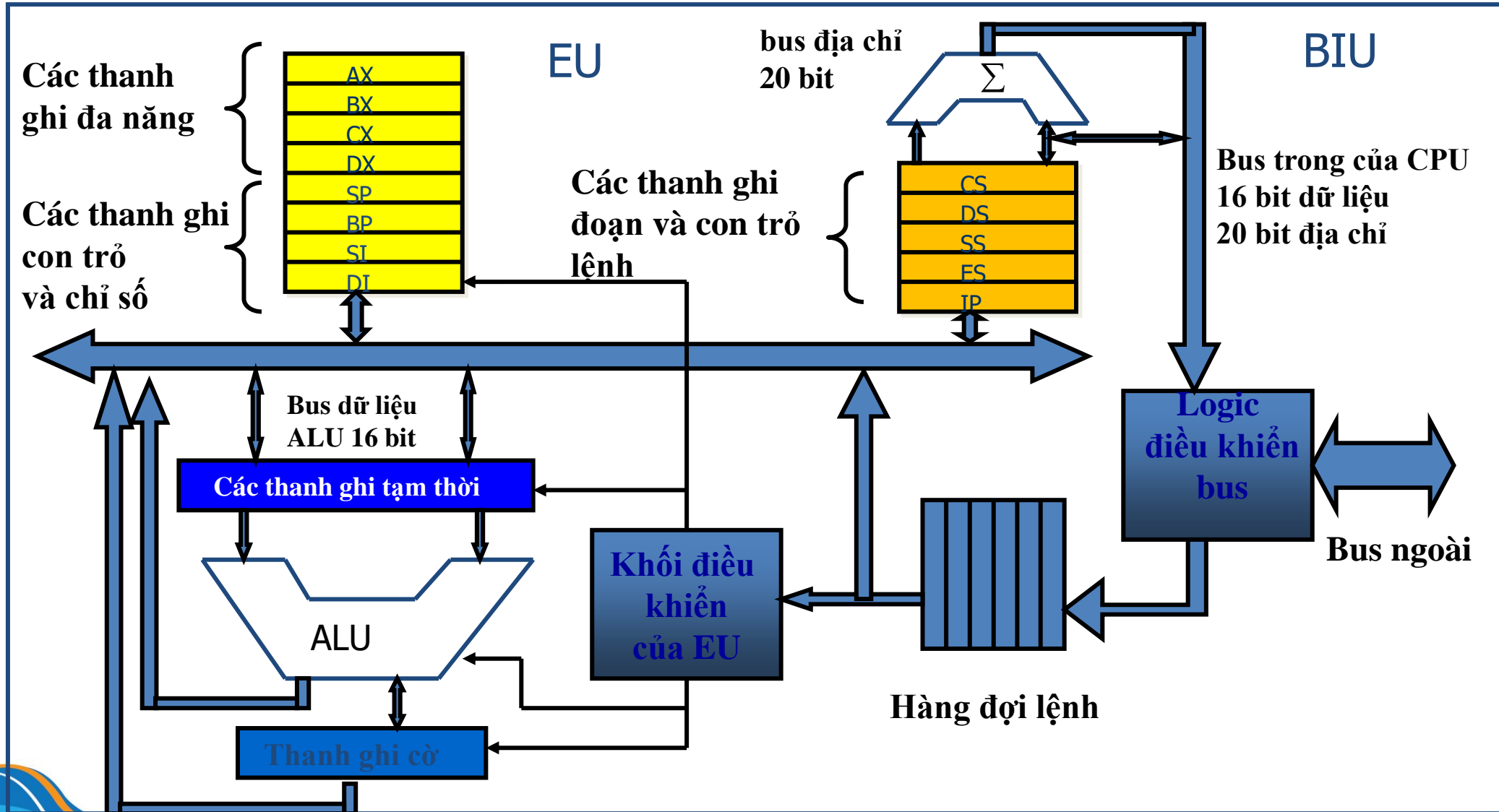☐ Input / Output

☐ X86 & MIPS comparison

X86 Architecture

- Complexity
  - instructions from 1 to 15 bytes long
  - one operand *must* act as both a source and destination
  - one operand *may* come from memory
  - several complex addressing modes

- Saving grace:
  - the most frequently used instructions are not too difficult to build
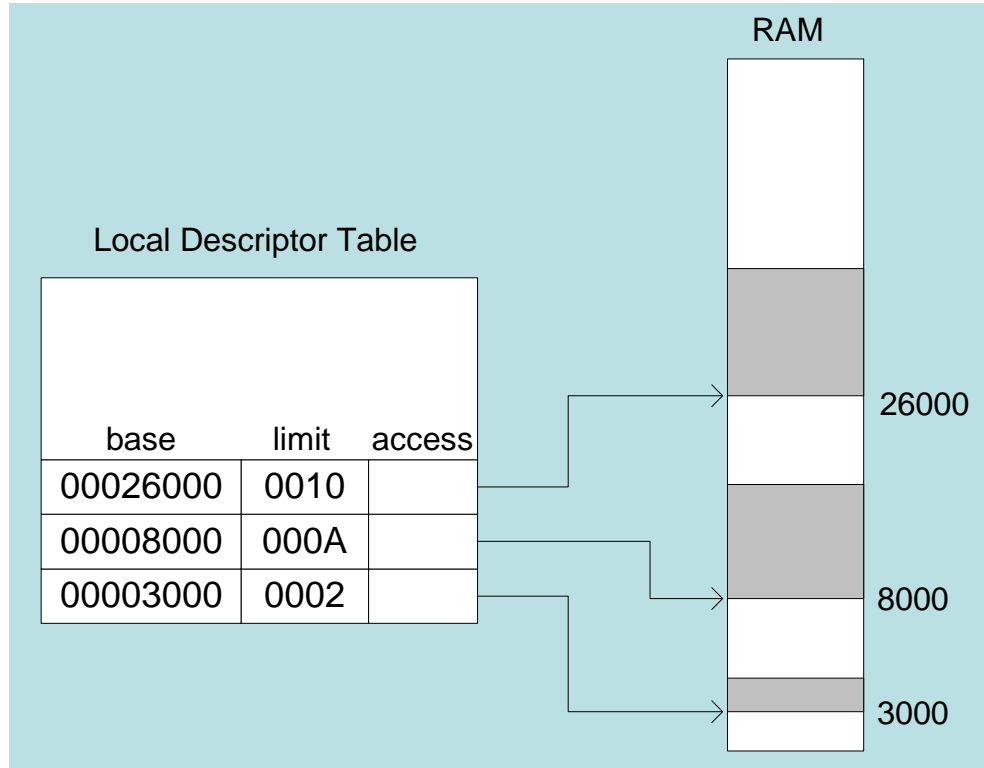 - compilers avoid the portions of the architecture that are slow

The Intel x86 ISA

- 1971: Intel 4004 (4-bit)
- 1972: Intel 8080 (8-bit)
- 1978: The Intel 8086 is announced (16-bit architecture)
- 1980: The 8087 floating point coprocessor is added
- 1982: The 80286 increases address space to 24 bits, +instructions
- 1985: The 80386 extends to 32 bits, new addressing modes
- 1989-1995: The 80486 (pipelined, on chip cache), Pentium, Pentium Pro add a few instructions (mostly designed for higher performance)
- 1997: MMX is added
- 2006-2008: Core 2 (64-bit), Core i3, i5, i7, Atom
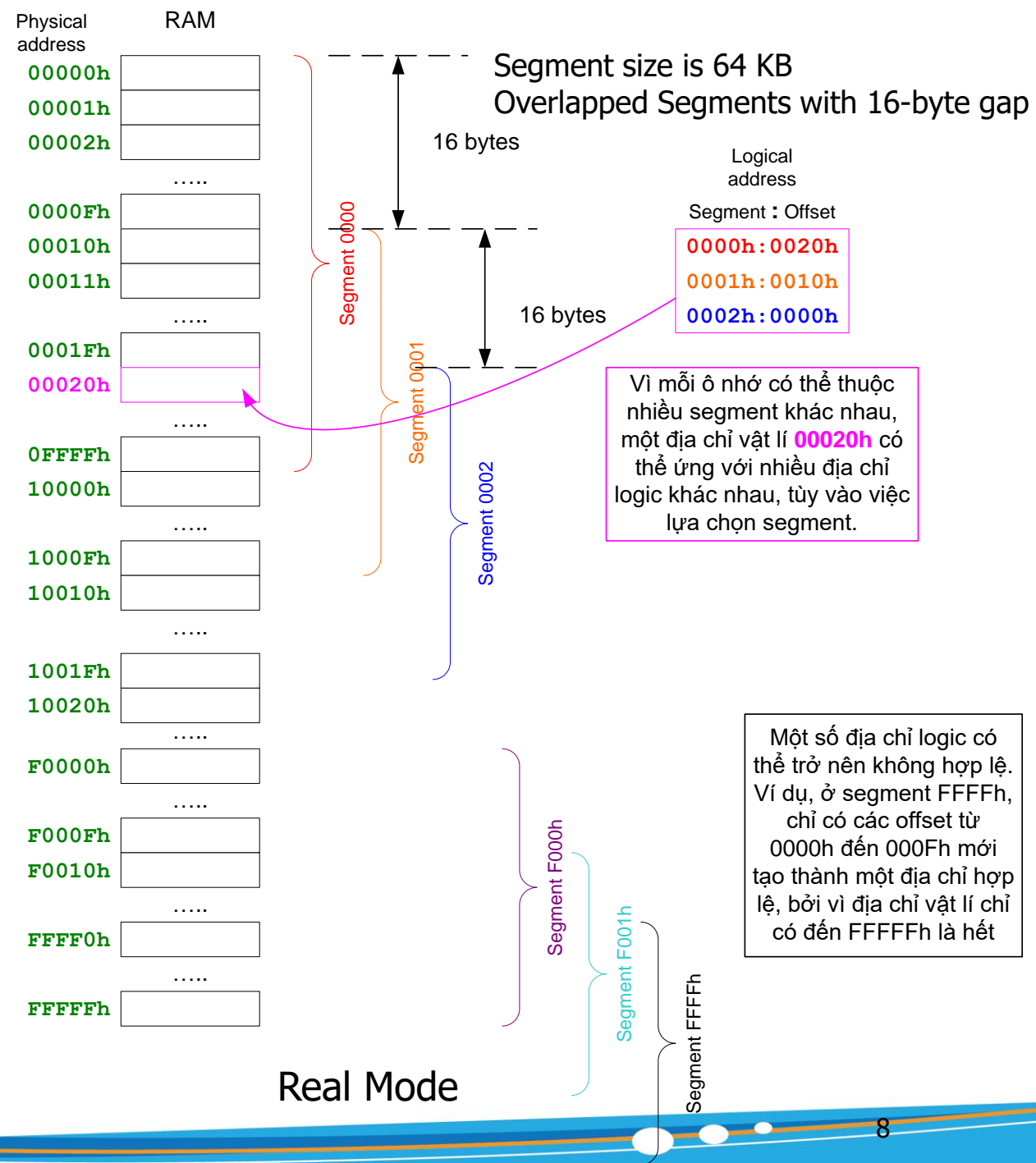- 2017: Core i9, instruction set extensions SSE4.1, SSE4.2, AVX2
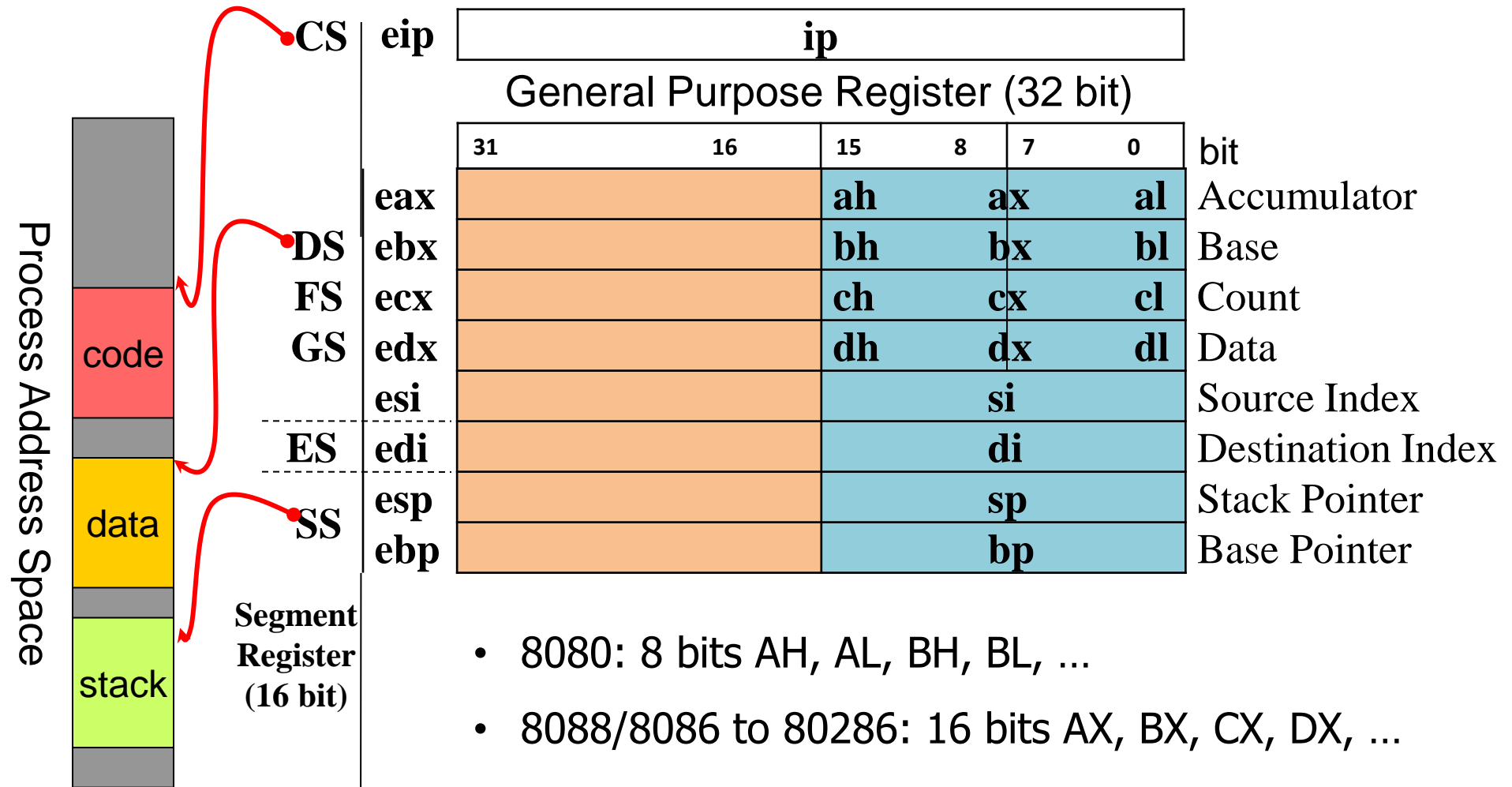
Inside an 8086-CPU

Memory Access

Local Descriptor Table

base	limit	access
00026000	0010	
00008000	000A	
00003000	0002	

RAM

26000

8000

3000

Protected Mode

Physical address — RAM

00000h
00001h
00002h
.....
0000Fh
00010h
00011h
.....
0001Fh
00020h
.....
0FFFFh
10000h
.....
1000Fh
10010h
.....
1001Fh
10020h
.....
F0000h
.....
F000Fh
F0010h
.....
FFFF0h
.....
FFFFFh

Segment size is 64 KB
Overlapped Segments with 16-byte gap

16 bytes

16 bytes

Logical address

Segment : Offset

0000h:0020h
0001h:0010h
0002h:0000h

Segment 0000
Segment 0001
Segment 0002

Vì mỗi ô nhớ có thể thuộc nhiều segment khác nhau, một địa chỉ vật lí **00020h** có thể ứng với nhiều địa chỉ logic khác nhau, tùy vào việc lựa chọn segment.

Một số địa chỉ logic có thể trở nên không hợp lệ. Ví dụ, ở segment FFFFh, chỉ có các offset từ 0000h đến 000Fh mới tạo thành một địa chỉ hợp lệ, bởi vì địa chỉ vật lí chỉ có đến FFFFFh là hết

Segment F000h
Segment F001h
Segment FFFFh

Real Mode

Register File

Process Address Space

code
data
stack

CS | eip | ip

Segment Register (16 bit)

DS
FS
GS

ES

SS

General Purpose Register (32 bit)

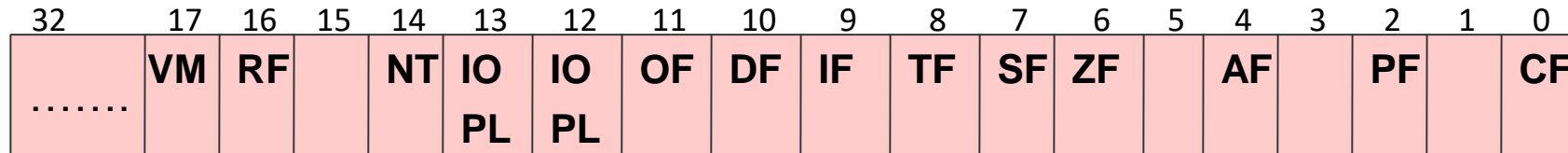|	31	16	15	8	7	0	bit
eax			ah	ax		al	Accumulator
ebx			bh	bx		bl	Base
ecx			ch	cx		cl	Count
edx			dh	dx		dl	Data
esi				si			Source Index
edi				di			Destination Index
esp				sp			Stack Pointer
ebp				bp			Base Pointer

- 8080: 8 bits AH, AL, BH, BL, ...

- 8088/8086 to 80286: 16 bits AX, BX, CX, DX, ...

- 80386 to Pentium M: 32 bits EAX, EBX, ECX, EDX, ...

- Core 2: 64 bits RAX, RBX, RCX, RDX, R8-15, ...

Other Registers

☐ Flag Register (EFLAGS – 32 bit)

32		17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
…….		VM	RF		NT	IO PL	IO PL	OF	DF	IF	TF	SF	ZF		AF		PF		CF

☐ 6 bits are used to be status flags:
- C/CF (carry flag)): CF=1
- P/PF (parity flag): PF=1 (0) when the number of 1's bit in the result is even (odd)
- A/AF (auxilary carry flag): extended carry flag
- Z/ZF (zero flag): ZF=1 when the result is 0
- S/SF (Sign flag): SF=1 when the result is less than 0
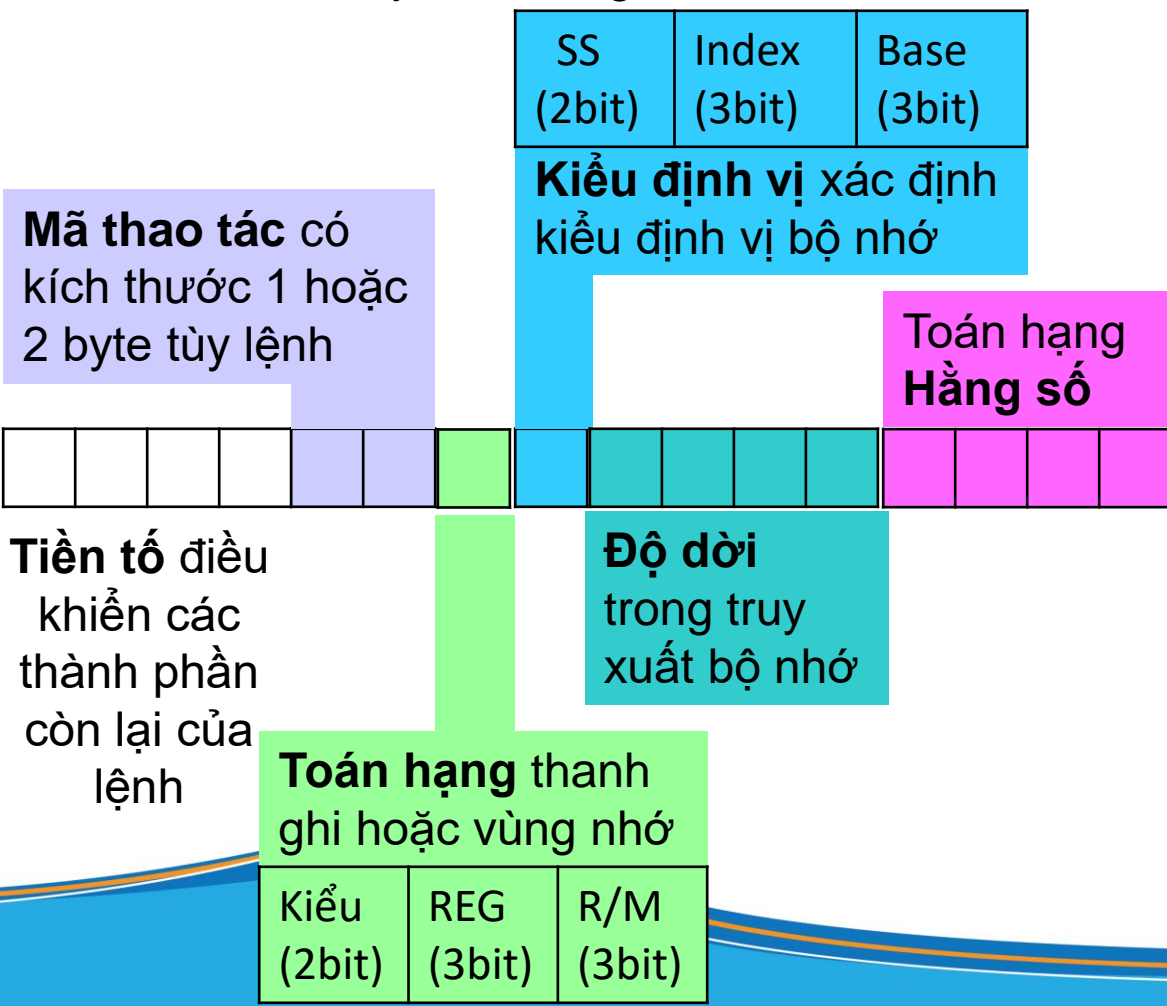- O/OF (Overflow flag): overflow detected in signed number computation

☐ 3 bits are used to be control flags:
- T/TF (trap flag)):): used for on chip debugging, TF=1 CPU will work in a single step mode. Generate an interrupt after each instruction
- I/IF (Interrupt enable flag): I = 1, CPU will recognize the interrupts from peripherals. For I = 0, the interrupts will be ignored
- D/DF (direction flag: D=1 the string will be accessed from higher memory address to lower memory address, and if D = 0, it will do the reverse
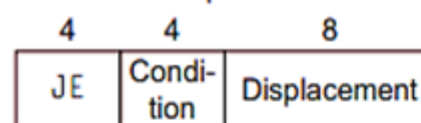
☐ Some others: IDTR (16bit), GDTR (48bit), LDTR (48bit), TR (16bit), ...

Instruction Format

☐ Although the instruction structure has a total of 16 bytes, only instructions are allowed up to 15 bytes in length.
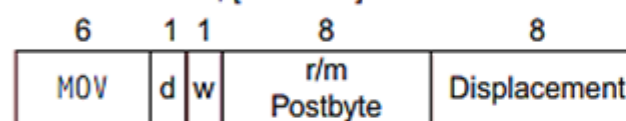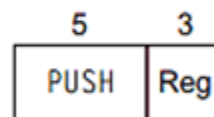
SS (2bit)	Index (3bit)	Base (3bit)

Kiểu định vị xác định kiểu định vị bộ nhớ

Mã thao tác có kích thước 1 hoặc 2 byte tùy lệnh

Toán hạng **Hằng số**

Tiền tố điều khiển các thành phần còn lại của lệnh

Độ dời trong truy xuất bộ nhớ

Toán hạng thanh ghi hoặc vùng nhớ

Kiểu (2bit)	REG (3bit)	R/M (3bit)

a. JE EIP + displacement

4	4	8
JE	Condi-tion	Displacement

b. CALL

8	32
CALL	Offset

c. MOV EBX, [ESI + 45]

6	1	1	8	8
MOV	d	w	r/m Postbyte	Displacement

d. PUSH ESI

5	3
PUSH	Reg

e. ADD EAX, #6765

4	3	1	32
ADD	Reg	w	Immediate

f. TEST EDX, #42

7	1	8	32
TEST	w	Postbyte	Immediate

11
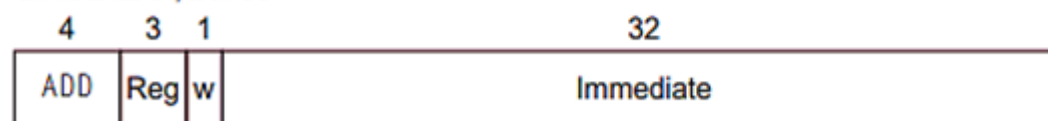
x86 Assembly Language

☐ x86 assembly has two alternative syntaxes available for it

  ☐ Intel

  ☐ AT&T

	Intel	AT&T
Comments	;	//
Instructions	Untagged *add*	Tagged with operand sizes: *addq*
Registers	eax, ebx, …	%eax,%ebx, …
Immediate	0x100	$0x100
Operand Order	mnemonic destination, source	mnemonic source, destination
Indirect	[eax]	(%eax)
General indirect	[base + reg * scale + displacement]	displacement(reg, reg, scale)

Data Addressing Mode

- Immediate
- Direct
- Indirect
- Register Direct
- Register Indirect
- Relative
- Indexed

Data Addressing Mode

SS (2bit)	Index (3bit)	Base (3bit)

Kiểu định vị xác định kiểu định vị bộ nhớ

Mã thao tác có kích thước 1 hoặc 2 byte tùy lệnh

Toán hạng Hằng số

Tiền tố điều khiển các thành phần còn lại của lệnh

Độ dời trong truy xuất bộ nhớ

Toán hạng thanh ghi hoặc vùng nhớ

Kiểu (2bit)	REG (3bit)	R/M (3bit)
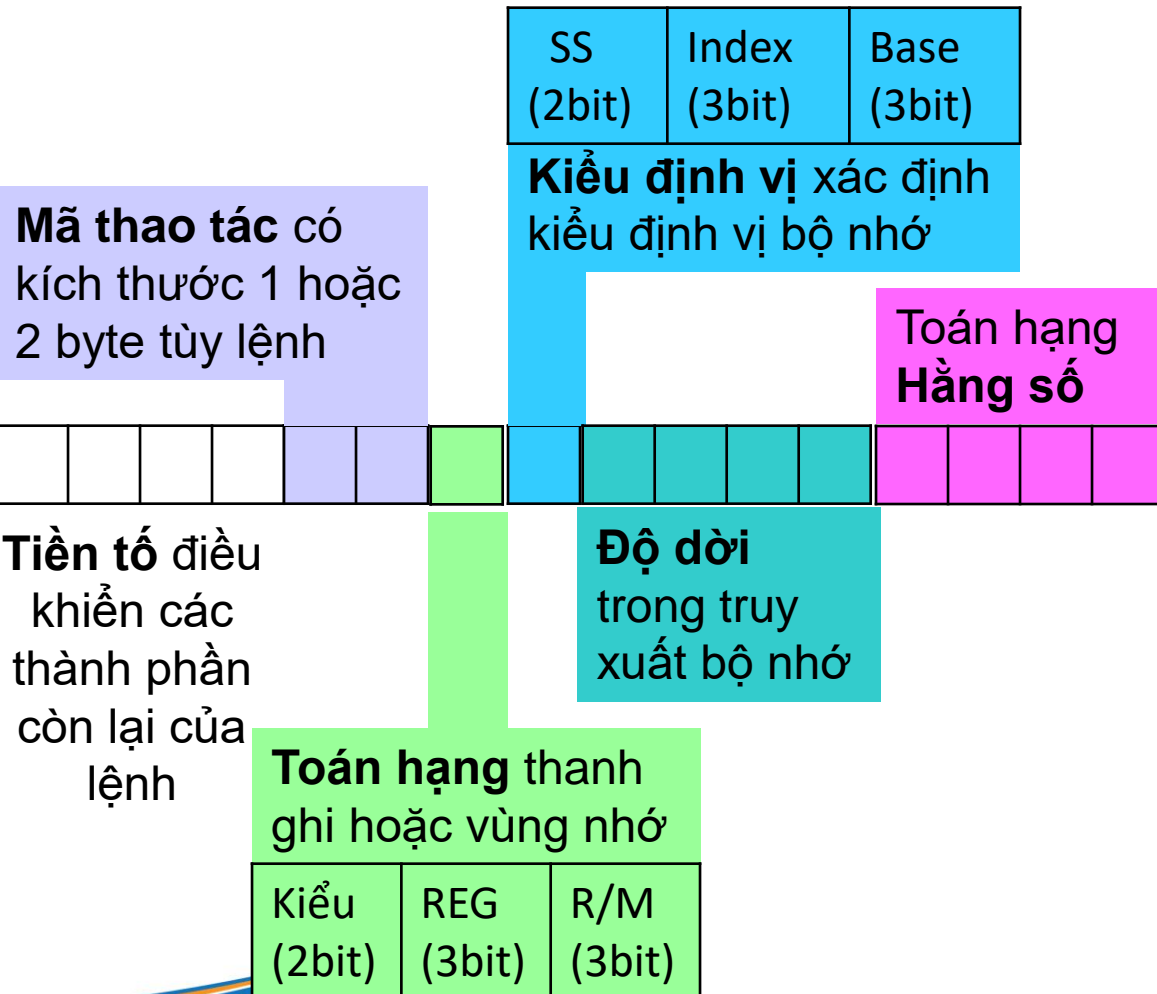
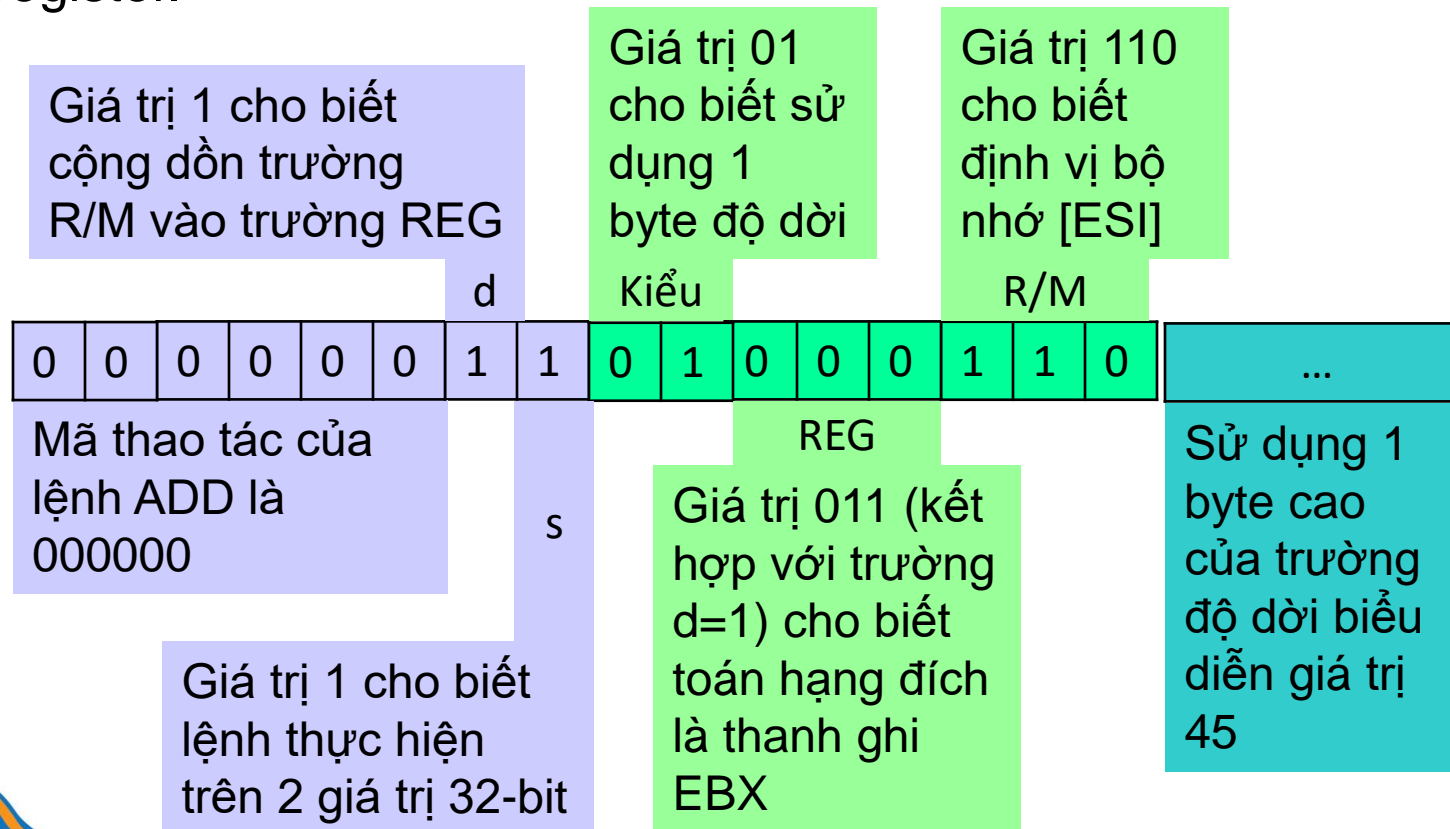| Type	Form	Operand value	Name
Immediate	Imm	Imm	Immediate
Register	r_a	$R[r_a]$	Register
Memory	Imm	$M[Imm]$	Absolute
Memory	(r_a)	$M[R[r_a]]$	Indirect
Memory	$Imm(r_b)$	$M[Imm + R[r_b]]$	Base + displacement
Memory	(r_b, r_i)	$M[R[r_b] + R[r_i]]$	Indexed
Memory	$Imm(r_b, r_i)$	$M[Imm + R[r_b] + R[r_i]]$	Indexed
Memory	$(, r_i, s)$	$M[R[r_i] \cdot s]$	Scaled indexed
Memory	$Imm(, r_i, s)$	$M[Imm + R[r_i] \cdot s]$	Scaled indexed
Memory	(r_b, r_i, s)	$M[R[r_b] + R[r_i] \cdot s]$	Scaled indexed
Memory	$Imm(r_b, r_i, s)$	$M[Imm + R[r_b] + R[r_i] \cdot s]$	Scaled indexed

Example of Register Addressing

□ ADD ECX, EAX

  □ This instruction adds the value in the EAX register to the ECX register

Giá trị 001 (kết hợp với trường d=0) cho biết toán hạng đích là thanh ghi ECX

Giá trị 11 cho biết trường R/M là thanh ghi

Giá trị 0 cho biết cộng dồn trường REG vào trường R/M

							d	Kiểu						R/M	
0	0	0	0	0	0	0	1	1	1	0	0	0	0	0	1

Mã thao tác của lệnh ADD là 000000

s

REG

Giá trị 000 (kết hợp với trường d=0) cho biết toán hạng nguồn là thanh ghi EAX

Giá trị 1 cho biết lệnh thực hiện trên 2 giá trị 32-bit

Example of Base + Displacement Addressing
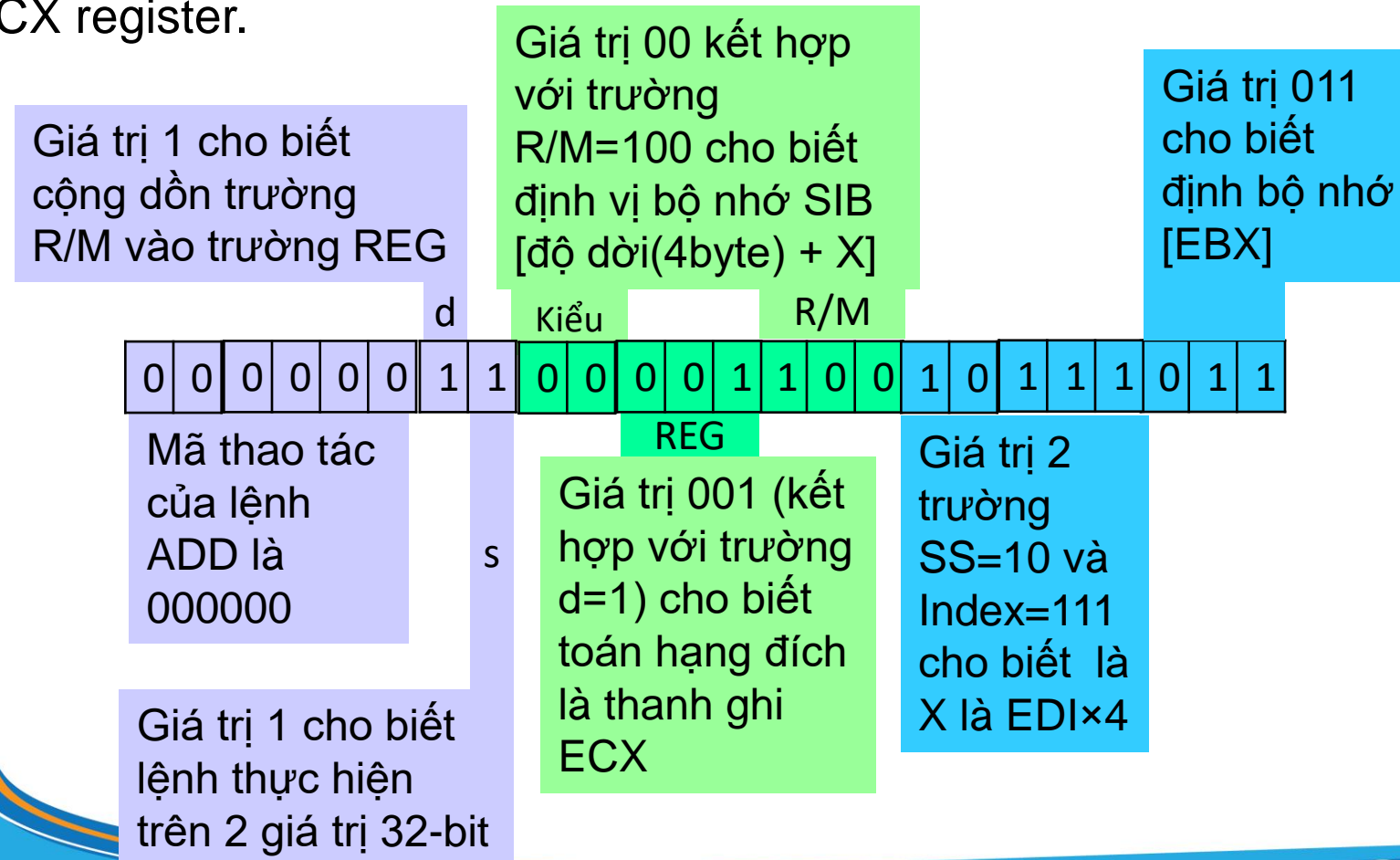
☐ ADD EBX, [ESI + 45]

  ☐ This instruction adds the value of a 4-byte memory word starting with DS:(ESI+45) into the EBX register.

Giá trị 1 cho biết cộng dồn trường R/M vào trường REG

Giá trị 01 cho biết sử dụng 1 byte độ dời

Giá trị 110 cho biết định vị bộ nhớ [ESI]

						d		Kiểu				R/M		
0	0	0	0	0	0	1	1	0	1	0	0	0	1	1	0	...

Mã thao tác của lệnh ADD là 000000

REG

Sử dụng 1 byte cao của trường độ dời biểu diễn giá trị 45

s

Giá trị 011 (kết hợp với trường d=1) cho biết toán hạng đích là thanh ghi EBX

Giá trị 1 cho biết lệnh thực hiện trên 2 giá trị 32-bit

Example of Scaled Indexed Addressing

☐ ADD ECX, [EBX + EDI × 4]

   ☐ This instruction adds the value of a 4-byte memory word starting with DS:(EDI × 4 + EBX) into the ECX register.

Giá trị 00 kết hợp với trường R/M=100 cho biết định vị bộ nhớ SIB [độ dời(4byte) + X]

Giá trị 011 cho biết định bộ nhớ [EBX]

Giá trị 1 cho biết cộng dồn trường R/M vào trường REG

| d | Kiểu | | R/M | |

| 0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 | 0 | 1 | 1 |

REG

Mã thao tác của lệnh ADD là 000000

s

Giá trị 001 (kết hợp với trường d=1) cho biết toán hạng đích là thanh ghi ECX

Giá trị 2 trường SS=10 và Index=111 cho biết là X là EDI×4

Giá trị 1 cho biết lệnh thực hiện trên 2 giá trị 32-bit

Data Addressing Mode

- Assume the following are stored as an indicated memory address and register

Address	Value	Register	Value
0x100	0xFF	%rax	0x100
0x104	0xAB	%rcx	0x1
0x108	0x13	%rdx	0x3
0x10C	0x11		

- Fill in the following table showing the value for indicated operands:

Operand	Value
%rax	_____
0x104	_____
$0x108	_____
(%rax)	_____
4(%rax)	_____
9(%rax,%rdx)	_____
260(%rcx,%rdx)	_____
0xFC(,%rcx,4)	_____
(%rax,%rdx,4)	_____

Data Addressing Mode

- Assume the following are stored as an indicated memory address and register

Address	Value		Register	Value
0x100	0xFF		%rax	0x100
0x104	0xAB		%rcx	0x1
0x108	0x13		%rdx	0x3
0x10C	0x11			

- Fill in the following table showing the value for indicated operands: (Solutions)

Operand	Value	Comment
%rax	0x100	Register
0x104	0xAB	Absolute address
$0x108	0x108	Immediate
(%rax)	0xFF	Address 0x100
4(%rax)	0xAB	Address 0x104
9(%rax,%rdx)	0x11	Address 0x10C
260(%rcx,%rdx)	0x13	Address 0x108
0xFC(,%rcx,4)	0xFF	Address 0x100
(%rax,%rdx,4)	0x11	Address 0x10C

Data Addressing Mode

Mode	Description	Register restrictions	MIPS equivalent
Register indirect	Address is in a register.	Not ESP or EBP	`lw $s0,0($s1)`
Based mode with 8- or 32-bit displacement	Address is contents of base register plus displacement.	Not ESP	`lw $s0,100($s1) # <= 16-bit` `                    # displacement`

Base plus scaled index	The address is
	Base + (2^{Scale} x Index)
	where Scale has the value 0, 1, 2, or 3.
		Base: any GPR
		Index: not ESP
			`mul    $t0,$s2,4`
			`add    $t0,$t0,$s1`
			`lw     $s0,0($t0)`

Base plus scaled index with 8- or 32-bit displacement	The address is
	Base + (2^{Scale} x Index) + displacement
	where Scale has the value 0, 1, 2, or 3.
		Base: any GPR
		Index: not ESP
			`mul    $t0,$s2,4`
			`add    $t0,$t0,$s1`
			`lw     $s0,100($t0) # <=16-bit`
			`                # displacement`

☐ Data movement instructions

☐ String instructions

☐ Arithmetic and Logic instructions

☐ Control flow

Data movement instructions

☐ MOV: The mov instruction copies the data item referred to by its second operand into the location referred to by its first operand

```
Syntax:
mov <reg>,<reg>
mov <reg>,<mem>
mov <mem>,<reg>
mov <reg>,<const>
mov <mem>,<const>
```

```
Example:
;copy the value in bx into ax
mov AX, BX
;store the value 5 into the byte
at location var
mov byte ptr[var], 5
```

Data movement instructions

☐ **The stack memory**

- ☐ Works according to LIFO (Last In First Out) mechanism
- ☐ Used in the decreasing direction of the address (different from the usual memory areas used in the increasing direction of the address)
- ☐ The SS:ESP register pair contains the segment:offset address of the top of the stack

☐ **PUSH: places its operand onto the top of the hardware supported stack in memory.**
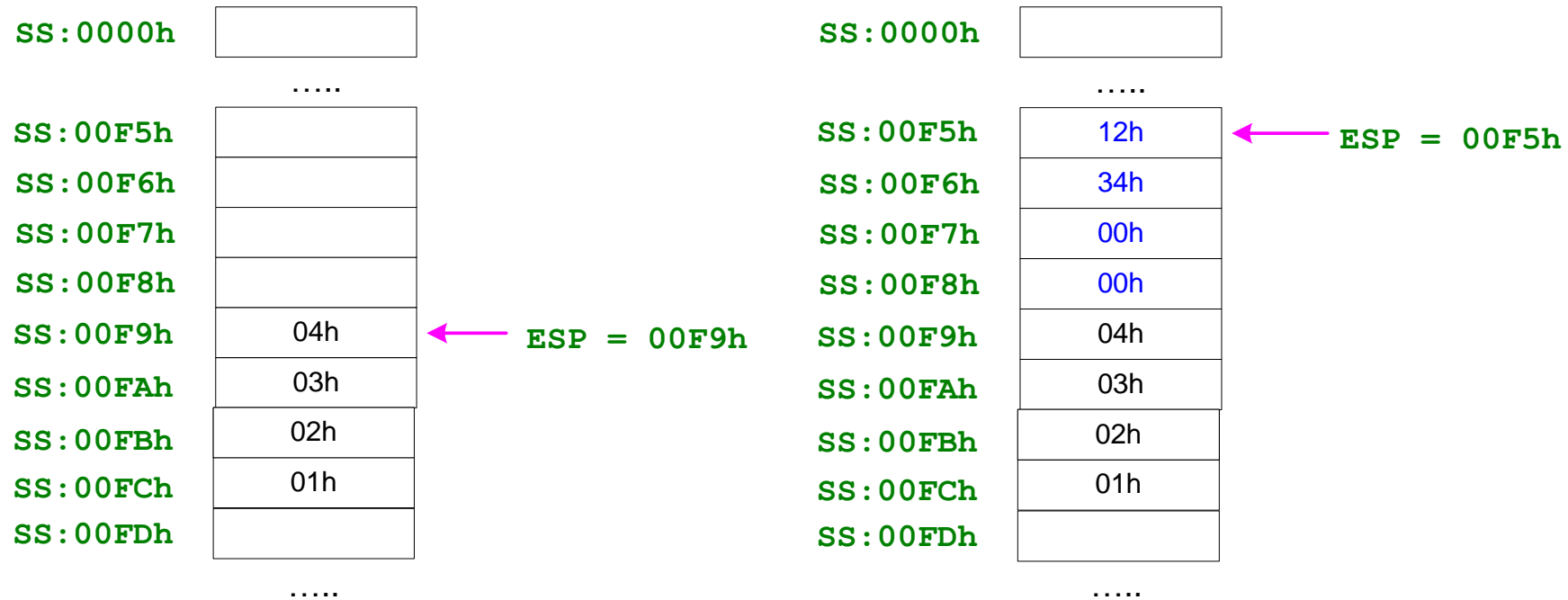
```
Syntax
push <reg32>
push <mem>
push <con32>
```

```
Example:
;Push eax on the stack
push EAX
;push the 4 bytes at
address var onto the stack
push [var]
```

PUSH Example

SS:0000h	

.....

SS:00F5h	
SS:00F6h	
SS:00F7h	
SS:00F8h	
SS:00F9h	04h
SS:00FAh	03h
SS:00FBh	02h
SS:00FCh	01h
SS:00FDh	

ESP = 00F9h

.....

SS:0000h	

.....

SS:00F5h	12h
SS:00F6h	34h
SS:00F7h	00h
SS:00F8h	00h
SS:00F9h	04h
SS:00FAh	03h
SS:00FBh	02h
SS:00FCh	01h
SS:00FDh	

ESP = 00F5h

.....

EAX = 3412h

Before PUSH EAX

After PUSH EAX

24

Data movement instructions

□ POP: removes the 4-byte data element from the top of the hardware-supported stack into the specified operand.

```
Syntax
pop <reg32>
pop <mem>
```

```
Example:
;pop the top element of the stack into EDI
pop EDI
;pop the top element of the stack into
memory at the four bytes starting at
location EBX
pop [EBX]
```
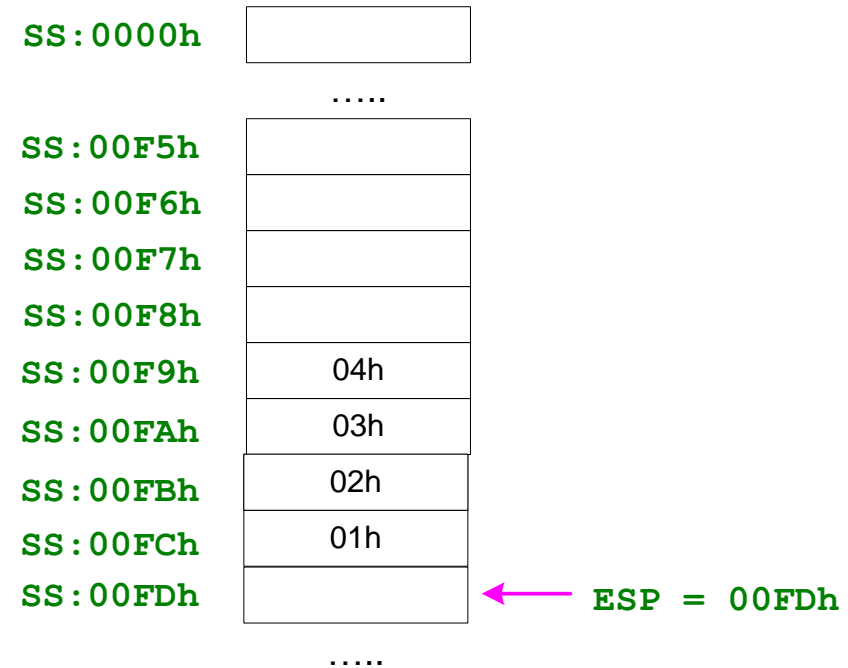
POP Example

SS:0000h [ ]

.....

SS:00F5h [ ]

SS:00F6h [ ]

SS:00F7h [ ]

SS:00F8h [ ]

SS:00F9h [04h] ← **ESP = 00F9h**

SS:00FAh [03h]

SS:00FBh [02h]

SS:00FCh [01h]

SS:00FDh [ ]

.....

Before POP EBX

SS:0000h [ ]

.....

SS:00F5h [ ]

SS:00F6h [ ]

SS:00F7h [ ]

SS:00F8h [ ]

SS:00F9h [04h]

SS:00FAh [03h]

SS:00FBh [02h]

SS:00FCh [01h]

SS:00FDh [ ] ← **ESP = 00FDh**

.....
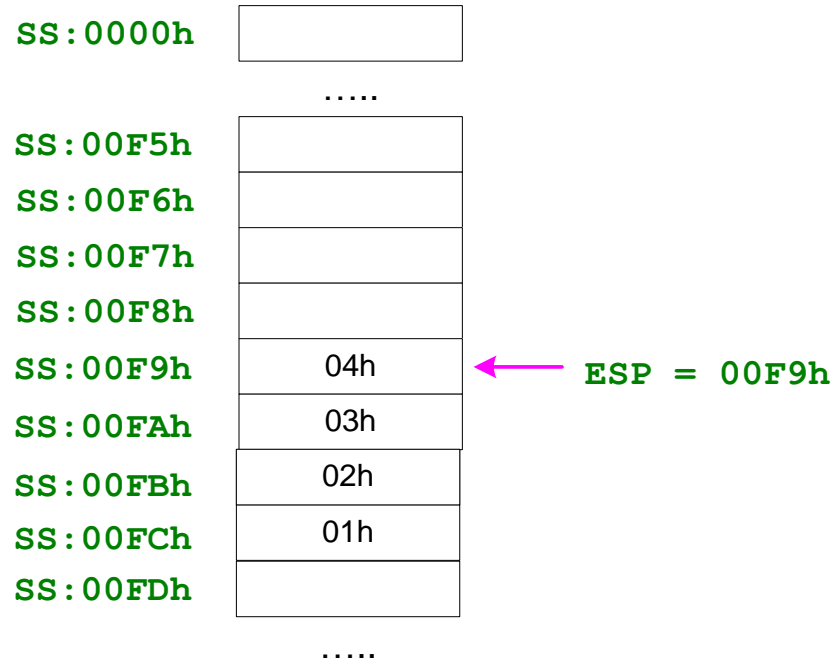
After POP EBX

EBX = 01020304h

26

Data movement instructions

☐ LEA: Load effective address.

```
Syntax
lea <reg32> <mem>
```

```
Example:
;the quantity EBX+4*ESI is placed in EDI
lea edi, [ebx+4*esi]
;the value in var is placed in EAX
lea eax, [var]
;the value val is placed in EAX
lea eax, [val]
;the address of variable x is placed in
EAX
lea eax, x
```

String instructions

☐ MOVS, MOVSB, MOVSW: copy from the string source (located in data segment) to destination (located in extra segment) by increment ESI and EDI; may be repeated

```
Example:
;move a string of length 4 bytes from source to destination
MOV SI, SRC
MOV DI, DST
MOV CX, 04H
CLD; Clear the direction flag
REP MOVSB
```

Arithmetic and Logic instructions

☐ ADD: adds together its two operands, storing the result in its first operand.

```
Syntax
add <reg>,<reg>
add <reg>,<mem>
add <mem>,<reg>
add <reg>,<con>
add <mem>,<con>
```

```
Example:
;EAX ← EAX + 5
add eax, 5
;add 5 to the single byte stored at
memory address var
add BYTE PTR[var], 5
```

☐ SUB: adds together its two operands, storing the result in its first operand.

```
Syntax                    Example:
sub <reg>,<reg>           ;AL ← AL - AH
sub <reg>,<mem>           sub AL, AH
sub <mem>,<reg>           ;subtract 5 from the value stored at EAX
sub <reg>,<con>           sub EAX, 5
sub <mem>,<con>
```

☐ **INC/DEC:** increments/ decrements the contents of its operand by one.

```
Syntax
inc <reg>
inc <mem>
dec <reg>
dec <mem>
```

```
Example:
;add one to the 32-bit integer stored at
location var
inc DWORD PTR [var]
;subtract 1 from the contents of EAX
dec EAX
```

Arithmetic and Logic instructions

☐ iMUL: three basic formats: one-operand, two-operand and three-operand

```
Syntax
imul <reg32>
imul <mem>
imul <reg32>,<reg32>
imul <reg32>,<mem>
imul <reg32>,<reg32>,<con>
imul <reg32>,<mem>,<con>
```

```
Example:
;multiply the contents of ECX by
EAX. Result stored in EDX:EAX
imul ECX
;multiply the contents of EAX by
the 32-bit contents of the
memory location var. Store the
result in EAX
imul EAX, [var]
;EDI ← ESI * 25
imul EDI, ESI, 25
```

☐ **iDIV**: divides the contents of the 64 bit integer EDX:EAX by the specified operand value. The quotient result of the division is stored into EAX, while the remainder is placed in EDX

```
Syntax
idiv <reg32>
idiv <mem>
```

```
Example:
;divide the contents of EDX:EAX by the
contents of EBX
idiv EBX
;divide the contents of EDX:EAX by the
32-bit value stored at memory
location var
idiv DWORD PTR [var]
```

☐ CMP: Compare the values of the two specified operands, setting the condition codes in the machine status word appropriately (based on flag register)

- Đích = nguồn : CF=0 ZF=1
- Đích > nguồn : CF=0 ZF=0
- Đích < nguồn : CF=1 ZF=0

```
Syntax
cmp <reg>,<reg>
cmp <reg>,<mem>
cmp <mem>,<reg>
cmp <reg>,<con>
```

```
Example:
;If the 4 bytes stored at location var are
equal to the 4-byte integer constant 3,
jump to the location labeled loop
cmp DWORD PTR [var], 3
jeq loop
```

Arithmetic and Logic instructions

☐ AND, OR, XOR: Bitwise logical and, or and exclusive or. Placing the result in the first operand location

```
Syntax
opcode <reg>,<reg>
opcode <reg>,<mem>
opcode <mem>,<reg>
opcode <reg>,<con>
opcode <mem>,<con>
```

```
Example:
;clear all but the last 4 bits of EAX
and EAX, 0fH
;set the contents of EDX to zero
xor EDX, EDX
```

Arithmetic and Logic instructions

☐ **SHL, SHR**: shift the bits in their first operand's contents left and right, padding the resulting empty bit positions with zeros

```
Syntax
opcode <reg>,<con8>
opcode <mem>,<con8>
opcode <reg>,<cl>
opcode <mem>,<cl>
```

```
Example:
;Multiply the value of EAX by 2 (if
the most significant bit is 0)
shl EAX, 1
;Store in EBX the floor of result of
dividing the value of EBX by 2ⁿ where
n is the value in CL
shr EBX, CL
```

Example

```
section    .data
    a      DW 4321h
           DW 8765h
    b      DW 0FFFFh
           DW 0

section    .code

    ; perform b = b + a
    MOV    AX, a
    MOV    BX, a+2
    ADD    b, AX      ; 4320h with CF=1
    ADC    b+2,BX     ; 8766h

    ; why not ?
    MOV    EAX, DWORD PTR a
    ADD    DWORD PTR b, EAX
```

```
    MOV    CX, 128

    ; perform DX:AX = AX * CX
    MOV    AX, 0F000h  ; 61440 dec
    MUL    CX          ; DX:AX = 0078:0000
                       ; (7864320=61440*128)

    MOV    AX, 0F000h  ; -4096 dec
    IMUL   CX          ; DX:AX = FFF8:0000
                       ; (-524288=-4096*128)

    ; perform AX = DX:AX / CX
    MOV    AX, 0F000h          ; 61440 dec
    DIV    CX                  ; AX = 01E0h

    MOV    AX, 0F000h          ; -4096 dec ???
    IDIV   CX                  ; AX = ?

    MOV    AX, 0F000h     ; -4096 dec
    CWD
    IDIV   CX             ; AX = FFE0h = -32

    NEG    AX             ; 0020h = 32
```

```
    MOV    AL, 36h
    AND    AL, 0Fh ; AL = 06h
                   ; AL = 00000110b

    AND    AL, 00000010b  ; AL = 02h
    OR     AL, 30h        ; AL = 32h
    XOR    AL, AL         ; AL = 0
    NOT    AL             ; AL = FFh
    MOV    AX, 1234h
    MOV    CL, 4
    SHR    AX, CL         ; 0123h
    SHL    AX, CL         ; 1230h

    MOV    AL, -4    ; -4 = FCh = 11111100
    SAR    AL, 1     ; -2 = FEh = 11111110

    MOV    AL, -4    ; -4 = FCh = 11111100
    SHR    AL, 1     ; 126 = 7Eh = 01111110

    MOV    AL, 10101010b    ; AAh
    ROL    AL, 1     ; 01010101 = 55h

    MOV    AL, 10101010b
    STC                     ; CF = 1
    RCR    AL, 1 ; 11010101 = D5h  CF = 0
```

☐ JMP: transfers program control flow to the instruction at the memory location indicated by the operand
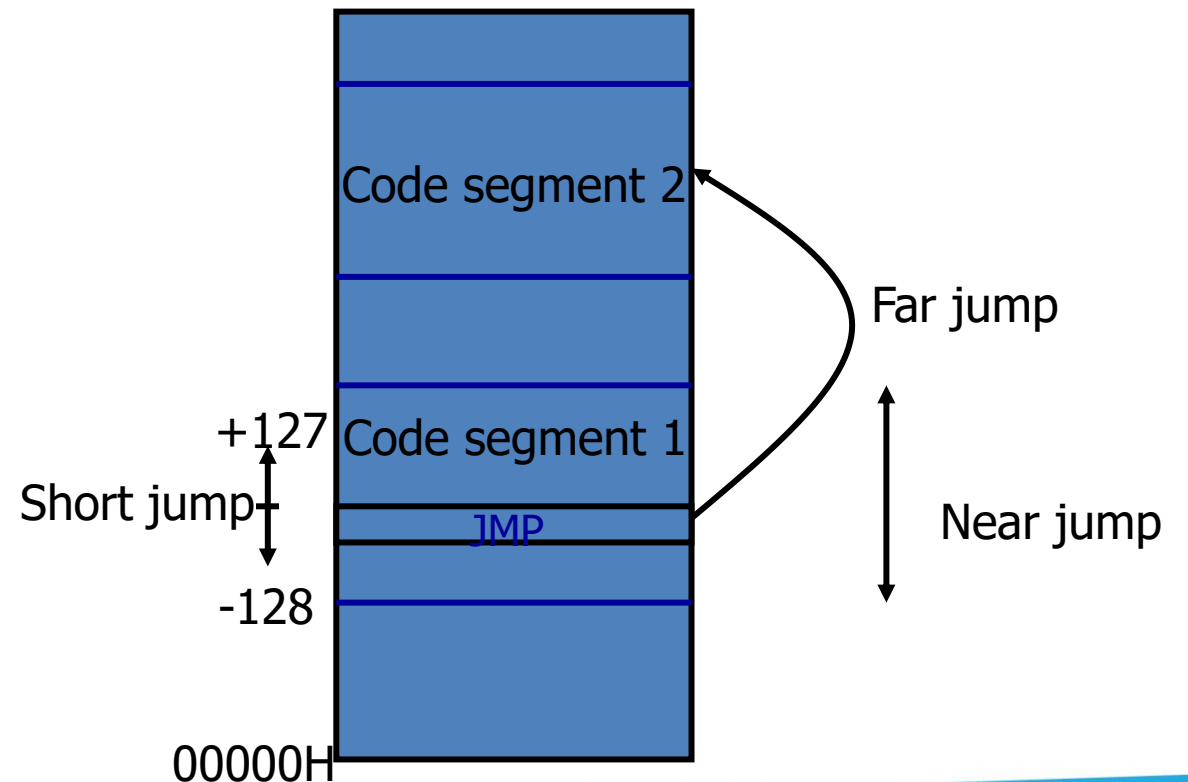
```
Syntax
jmp <Label>
```

Example:
```
;Jump to the label named "BEGIN"
Jmp BEGIN
```

☐ 3 types of JMP instruction:

- ☐ JMP SHORT(short jump)
- ☐ JMP NEAR (near jump)
- ☐ JMP FAR (far jump)

Code segment 2

Far jump

+127 Code segment 1

Short jump

JMP

Near jump

-128

00000H

Control flow instructions

☐ **Conditional jump**

  ☐ **Jump with flags for unsigned results:**

    ■ JA(JNBE), JB(JNAE), JE(JZ), JNA(JBE), JNB(JAE), JNE(JNZ)

  ☐ **Jump with flags for signed results:**

    ■ JG(JNLE), JL(JNGE), JE(JZ), JNG(JLE), JNL(JGE), JNE(JNZ)

  ☐ Jump with the value of a flag

    ■ JC, JZ(JE), JS, JO, JNC, JNZ(JNE), JNS, JNO

☐ **Based on the status of a set of condition codes that are stored in a special register called the machine status word**

```
Syntax                  Example:
opcode <Label>          ;Jump to the instruction named "DONE"
                        if the condition satisfies
                        cmp EAX, 0
                        jg done
```

Control flow instructions

☐ LOOP, LOOPE/LOOPZ, LOOPNE/LOOPNZ: is a combination instruction of DEC CX and JNZ

```
Syntax
<Label:>
      Task
Loop <Label>
```

```
Example:
;Repeat when CX != 0. Decrements CX
after each loop
XOR AL, AL

MOV CX, 16

myloop:

      INC AL

      LOOP myloop
```

Example

C Language	ASM (2)	ASM (1)
If (AX==0)	CMP AX, 0	CMP AX, 0
	JNE TIEP	JE CONG
AX = AX + 1;	INC AX	JMP TIEP
	TIEP:	CONG:
BX = AX;	MOV BX, AX	INC AX
		TIEP:
		MOV BX, AX
If (AX<0)	CMP AX, 0	CMP AX, 0
	JNL LONHON	JL NHOHON
AX = AX + 1;	INC AX	DEC AX
	JMP TIEP	JMP TIEP
Else	LONHON:	NHOHON:
AX = AX – 1;	DEC AX	INC AX
	TIEP:	TIEP:
BX = AX;	MOV BX, AX	MOV BX, AX

Example

C Language	ASM (2)	ASM (1)
If (AL=='S')	CMP AL, 'S'	CMP AL, 'S'
	JNE KP_SANG	JE CHAO_BUOI_SANG
printf ("Chao buoi sang");	; xuất thông báo	CMP AL, 'T'
	; "Chao buoi sang"	JE CHAO_BUOI_TRUA
	;	CMP AL, 'C'
	JMP THOAT	JE CHAO_BUOI_CHIEU
else if (AL=='T')	KP_SANG:	JMP THOAT
	CMP AL, 'T'	CHAO_BUOI_SANG:
	JNE KP_TRUA	; xuất thông báo
printf ("Chao buoi trua");	; xuất thông báo	; "Chao buoi sang"
	; "Chao buoi trua"	;
	;	JMP THOAT
	JMP THOAT	CHAO_BUOI_TRUA:
else if (AL=='C')	KP_TRUA:	; xuất thông báo
	CMP AL, 'C'	; "Chao buoi trua"
	JNE THOAT	;
printf ("Chao buoi chiều");	; xuất thông báo	JMP THOAT
	; "Chao buoi chieu"	CHAO_BUOI_CHIEU:
	;	; xuất thông báo
	THOAT:	; "Chao buoi chieu"
		;
		THOAT:

Example

C Language	ASM (2)	ASM (1)	ASM (3)
If (AL>='a' and AL<='z')	CMP AL, 'a'	CMP AL, 'a'	CMP AL, 'a'
	JB KPTHUONG	JAE CTTHUONG	JB KPTHUONG
	CMP AL, 'z'	DEC AX	CMP AL, 'z'
	JA KPTHUONG	JMP TIEP	JBE THUONG
AX = AX + 1;	INC AX	CTTHUONG:	KPTHUONG:
	JMP TIEP	CMP AL, 'z'	DEC AX
else	KPTHUONG:	JBE THUONG	JMP TIEP
AX = AX – 1;	DEC AX	DEC AX	THUONG:
	TIEP:	JMP TIEP	INC AX
BX = AX;	MOV BX, AX	THUONG:	TIEP:
		INC AX	MOV BX, AX
		TIEP:	
		MOV BX, AX	

fit@hcmus
cdio 4.0

Example

C Language

If (AL>='A' and AL<='Z')

    printf ("La ky tu hoa");

else if (AL>='0' and AL<='9')

    printf ("La ky tu so");

else
    printf ("La ky tu khac");

ASM (2)

```
CMP AL, 'A'
JB XETSO
CMP AL, 'Z'
JA KHAC
; xuất thông báo
; "La ky tu hoa"
;
JMP THOAT
XETSO:
CMP AL, '0'
JB KHAC
CMP AL, '9'
JA KHAC
; xuất thông báo
; "La ky tu so"
;
JMP THOAT
KHAC:
; xuất thông báo
; "La ky tu khac"
;
THOAT:
```

ASM (1)

```
CMP AL, '0'
JAE CTLASO
JMP KHAC
CTLASO:
CMP AL, '9'
JBE LASO
CMP AL, 'A'
JAE CTLAHOA
JMP KHAC
CTLAHOA:
CMP AL, 'Z'
JBE LAHOA
JMP KHAC
LASO:
; xuất thông báo
; "La ky tu so"
;
JMP THOAT
LAHOA:
; xuất thông báo
; "La ky tu hoa"
;
JMP THOAT
KHAC:
; xuất thông báo
; "La ky tu khac"
;
JMP THOAT
THOAT:
```

ASM (3)

```
CMP AL, '0'
JB KHAC
CMP AL, '9'
JBE LASO
CMP AL, 'A'
JB KHAC
CMP AL, 'Z'
JBE LAHOA
KHAC:
; xuất thông báo
; "La ky tu khac"
;
JMP THOAT
LASO:
; xuất thông báo
; "La ky tu so"
;
JMP THOAT
LAHOA:
; xuất thông báo
; "La ky tu hoa"
;
THOAT:
```

Procedure

- ☐ **CALL <Procedure Name>**
  - ☐ Use stack to store (PUSH) the address of the next instruction right after the CALL instruction (where to return)
  - ☐ Write to the EIP instruction pointer register the address of the first instruction of the procedure.

- ☐ **Procedure declaration**

```
<Procedure Name> PROC        sample PROC
    .                            .
    .                            .
    ret                          ret
<Procedure Name> ENDP        sample ENDP
```

- ☐ **RET**
  - ☐ Gets (POP) the value from the top of the stack and writes it to EIP register, so the next instruction to be executed as the instruction right after the CALL instruction.

Procedure Example

```
                section .code
                    ...
0005h               MOV  AX,'a'
0008h               CALL ToUpper
000Bh               MOV  BX,AX
000Dh               MOV  AX,'z'
0010h               CALL ToUpper
0013h               MOV  CX,AX

                    ...
                    ...
                    ...
                    ...

                ToUpper PROC
0045h               SUB  AX,20h
0048h               RET
                ToUpper ENDP
                    ...
```

Explanation of the first call to *ToUpper* procedure

Before CALL

Stack segment

....	
01FCh	
01FDh	
01FEh	
01FFh	
0200h	← ESP

EIP	0008h
ESP	0200h

After CALL

Stack segment

....		
01FCh	0Bh	← ESP
01FDh	00h	
01FEh	00h	
01FFh	00h	
0200h		

EIP	0045h
ESP	01FCh

After RET

Stack segment

....		
01FCh	0Bh	
01FDh	00h	
01FEh	00h	
01FFh	00h	
0200h		← ESP

EIP	000Bh
ESP	0200h
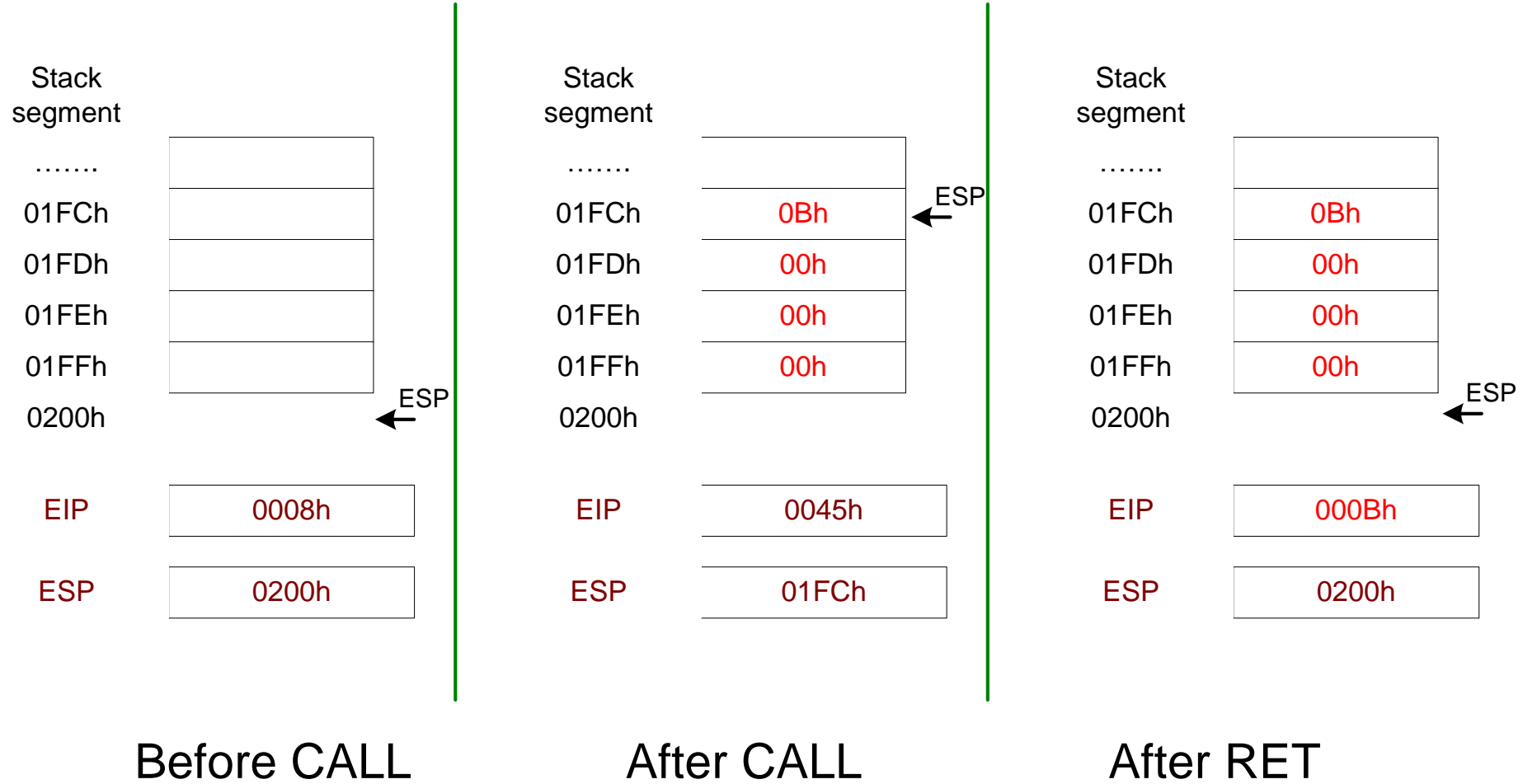
Explanation of the second call to *ToUpper* procedure

	Before CALL		After CALL		Before RET
	Stack segment		Stack segment		Stack segment

Before CALL

Stack segment

....
01FCh	0Bh
01FDh	00h
01FEh	00h
01FFh	00h
0200h	← ESP

| EIP | 0010h |
| ESP | 0200h |

After CALL

Stack segment

....
01FCh	13h	← ESP
01FDh	00h
01FEh	00h
01FFh	00h
0200h	

| EIP | 0045h |
| ESP | 01FCh |

Before RET

Stack segment

....
01FCh	13h
01FDh	00h
01FEh	00h
01FFh	00h
0200h	← ESP

| EIP | 0013h |
| ESP | 0200h |

Before CALL   After CALL   Before RET

Nested procedure call

```
section .code
    ...
    MOV    AX,'V'        0005h
    CALL   Upcase        0008h
    MOV    BX,AX         000Bh
    MOV    AX,'n'        000Dh
    CALL   Upcase        0010h
    MOV    CX,AX         0013h
    ...
    ...
ToUpper PROC
    SUB    AX,20h        0045h
    RET                  0048h
ToUpper ENDP
Upcase PROC
    CMP    AX,'a'        0049h
    JB     Notaz         004Ch
    CMP    AX,'z'        004Eh
    JA     Notaz         0051h
    CALL   ToUpper       0053h
Notaz:
    RET
Upcase ENDP
    ...                  0056h
```

Stack segment		Stack segment		Stack segment		Stack segment		Stack segment	
....		....		....		....		....	
01F8h		01F8h		01F8h	56h ← ESP	01F8h	56h	01F8h	56h
01F9h		01F9h		01F9h	00h	01F9h	00h	01F9h	00h
01FAh		01FAh		01FAh	00h	01FAh	00h	01FAh	00h
01FBh		01FBh		01FBh	00h	01FBh	00h	01FBh	00h
01FCh		01FCh	0Bh ← ESP	01FCh	0Bh	01FCh	0Bh ← ESP	01FCh	0Bh
01FDh		01FDh	00h	01FDh	00h	01FDh	00h	01FDh	00h
01FEh		01FEh	00h	01FEh	00h	01FEh	00h	01FEh	00h
01FFh		01FFh	00h	01FFh	00h	01FFh	00h	01FFh	00h
0200h ← ESP		0200h		0200h		0200h		0200h ← ESP	

EIP	0008h	EIP	0049h	EIP	0045h	EIP	0056h	EIP	000Bh
ESP	0200h	ESP	01FCh	ESP	01F8h	ESP	01FCh	ESP	0200h

Before CALL Upcase	After CALL Upcase	After CALL ToUpper	After RET ToUpper	After CALL Upcase

49

Input / Output

- ## Independent on system
  - ### Interrupt generated by the software
    - Commands to swap with out of external devices: IN, OUT, …
    - DOS and BIOS interrupt server subroutines: INT 21h, …
- ## Dependent on system
  - ### Linux
    - syscall
    - C Library: puts, …
  - ### Windows
    - API: call   _WriteConsoleA@20, ...
    - C Library: call   _printf, …

x86-32bit Assembly Program "Hello World !"

```
global _WinMain@16
extern _MessageBoxA@16


[section .data]
    title db "Message",0
    message db "Hello World!",0


[section .code]
_WinMain@16:
    push 0
    push title
    push message
    push 0
    call _MessageBoxA@16
    ret 16
```

Compare 32-bit MIPS and x86 instructions

☐ MIPS: "Three-Operand Architecture"

  ☐ 2 source operands and 1 destination operand

```
add $s0,$s1,$s2 # s0=s1+s2
```

  ☐ Advantages: Fewer instructions $\Rightarrow$ Faster processing

☐ x86: "Two-Operand Architecture"

  ☐ 1 source operand and 1 operand play the role of destination operand and source operand

```
add EBX,EAX ; EBX=EBX+EAX
```

  ☐ Advantages: Shorter commands $\Rightarrow$ Smaller source code

Compare 32-bit MIPS and x86 instructions

- MIPS: "Load-Store Architecture"
  - Only the Load/Store instruction accesses memory; the rest of the instructions operate on registers and constants
    ```
    lw $t0, 12($gp)
    add $s0, $s0,$t0 # s0=s0+Mem[12+gp]
    ```
  - Advantages: Simpler processing circuit $\Rightarrow$ Easy to increase speed by using parallel techniques

- x86: "Register-memory architecture"
  - All instructions can access memory
    ```
    ADD EAX,[ESI + 12] ;EAX=EAX+Mem[12+ESI]
    ```
  - Advantages: Fewer commands $\Rightarrow$ Smaller source code

Compare 32-bit MIPS and x86 instructions

☐ MIPS: "<u>Fixed Length Instructions</u>"

- ☐ All instructions are 4 bytes in size
- ☐ Simpler processing circuit $\Rightarrow$ Faster processing
- ☐ Jump instructions: multiple of 4 bytes

☐ x86: "<u>Variable length instructions</u>"

- ☐ Instruction size varies from 1 byte to 16 bytes
- $\Rightarrow$ Source code can be smaller (30% ?)
- ☐ Use cache more efficiently
- ☐ Instructions can have 8-bit or 32-bit constant/immediate