CHAPTER

5

X86 INSTRUCTION SET

TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
TP. HỒ CHÍ MINH

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

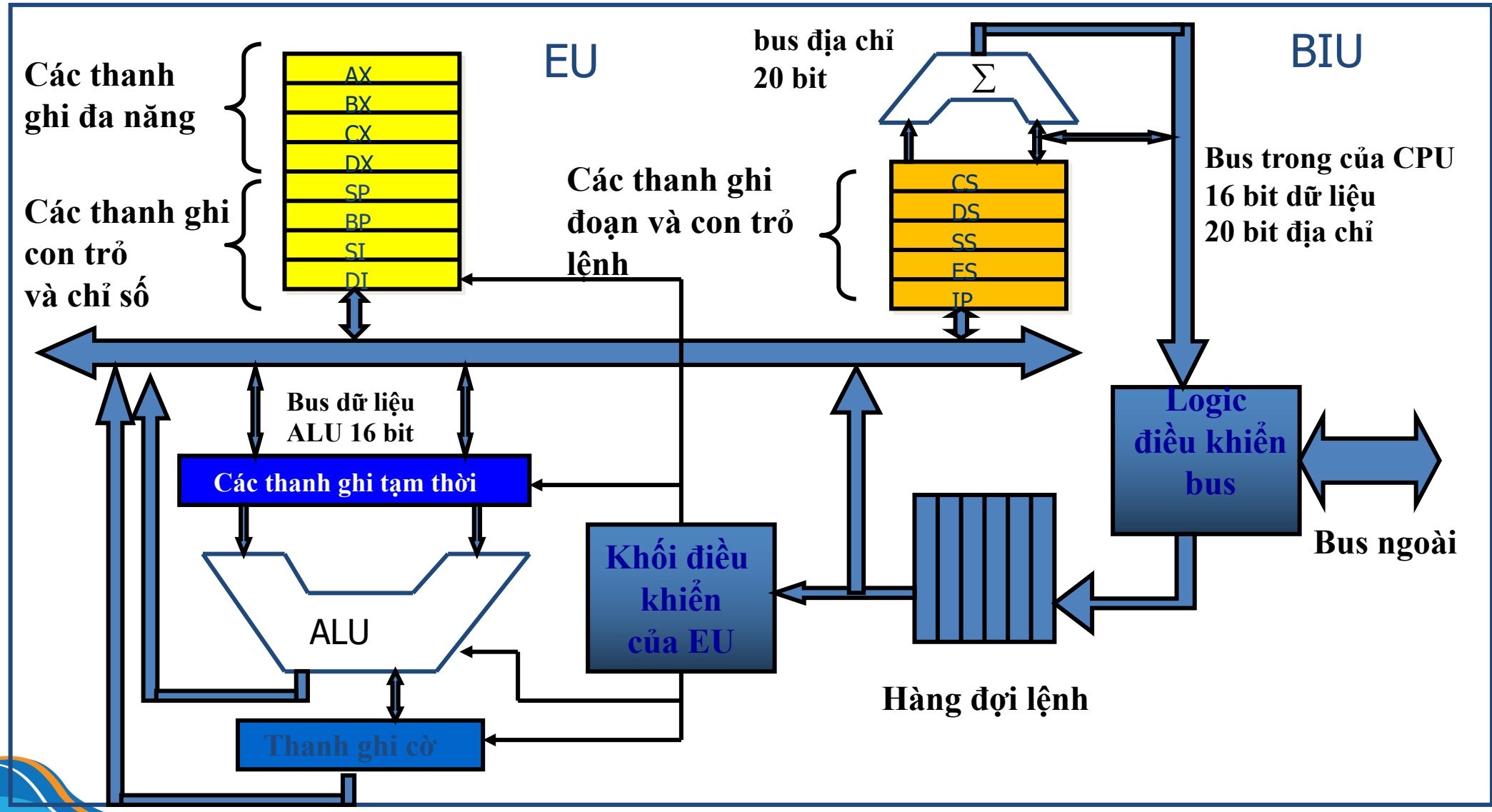
- Complexity
  - instructions from 1 to 15 bytes long
  - one operand *must* act as both a source and destination
  - one operand *may* come from memory
  - several complex addressing modes

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

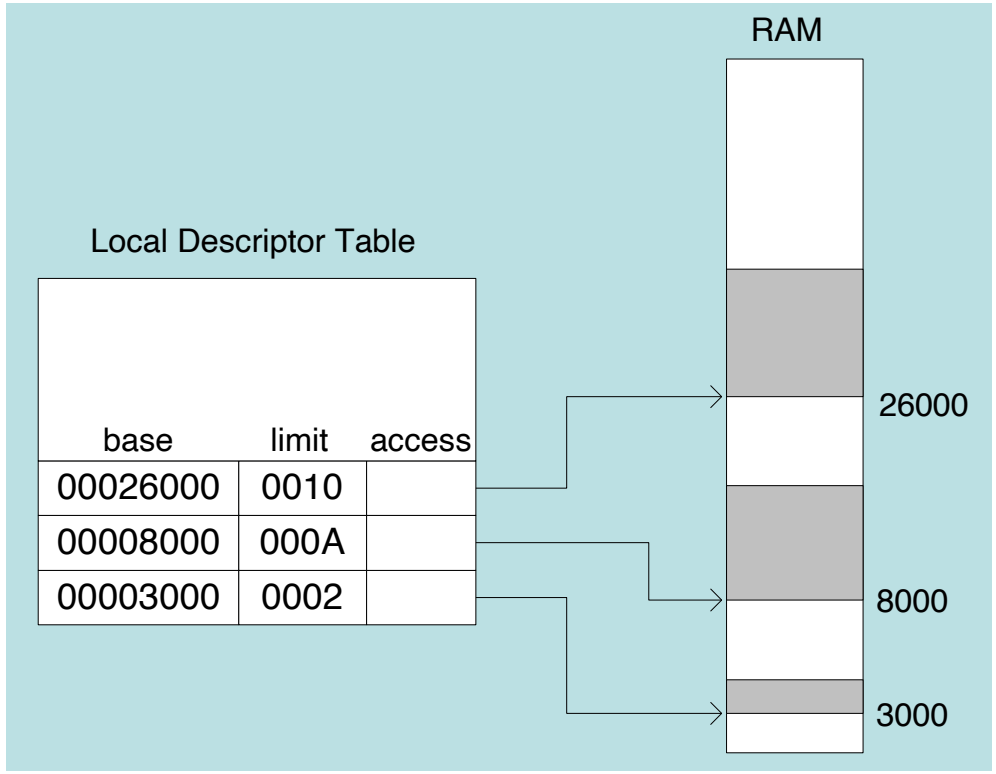
The Intel x86 ISA

- 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

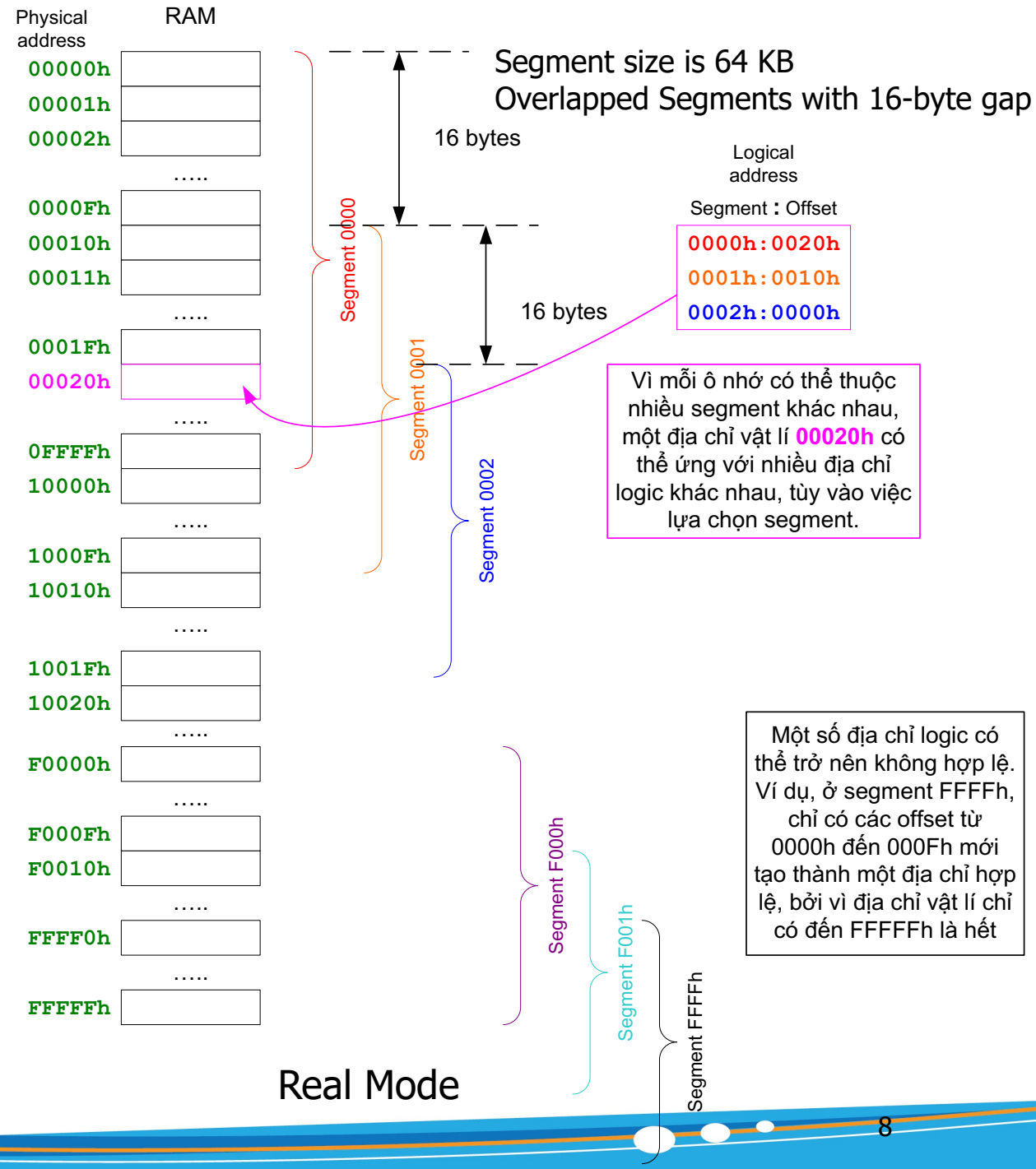
Inside an 8086-CPU



Memory Access

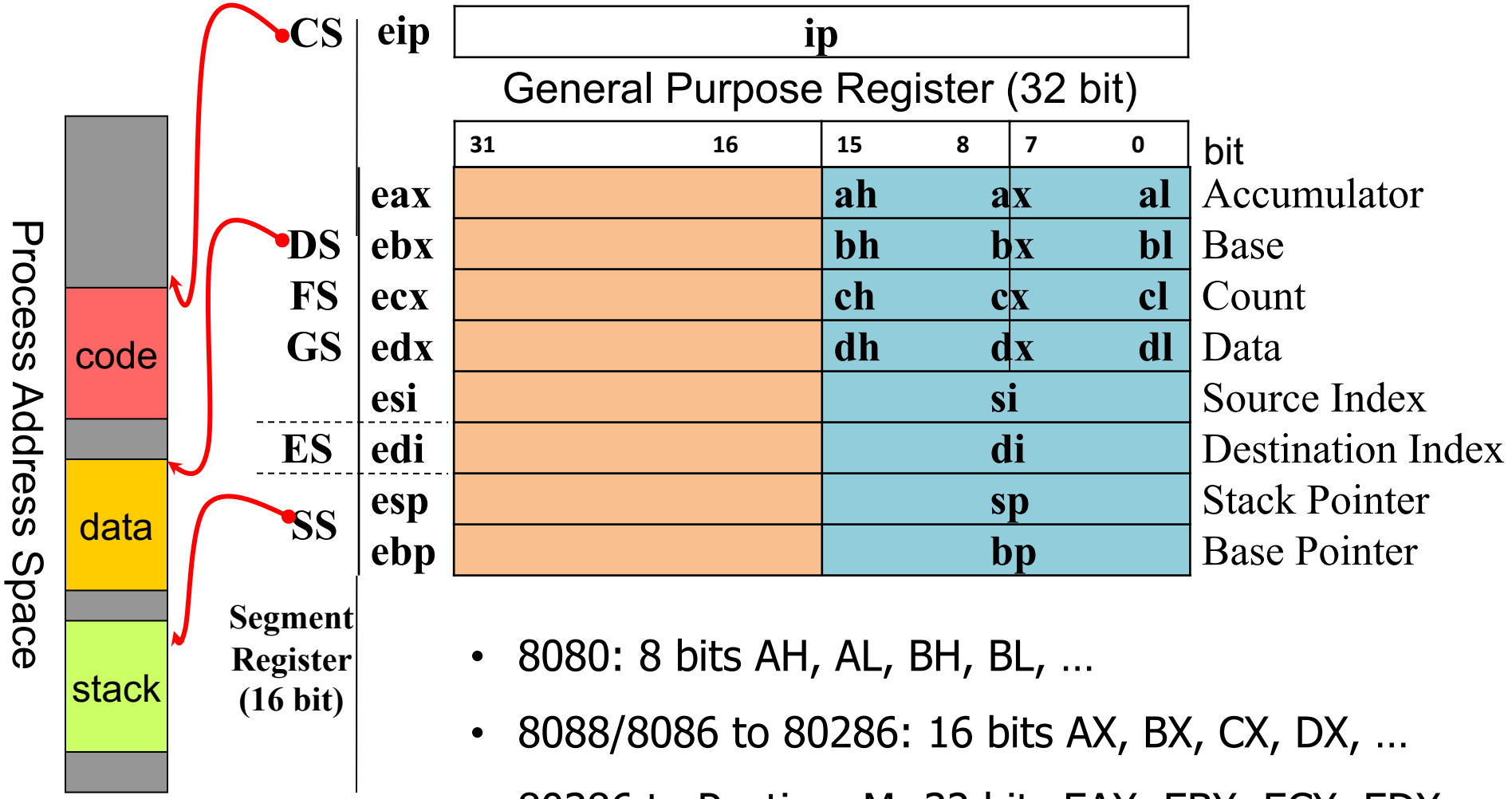


Protected Mode



Real Mode

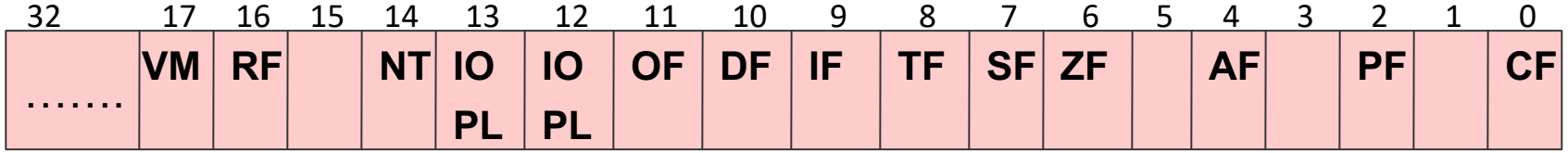
Register File



- 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

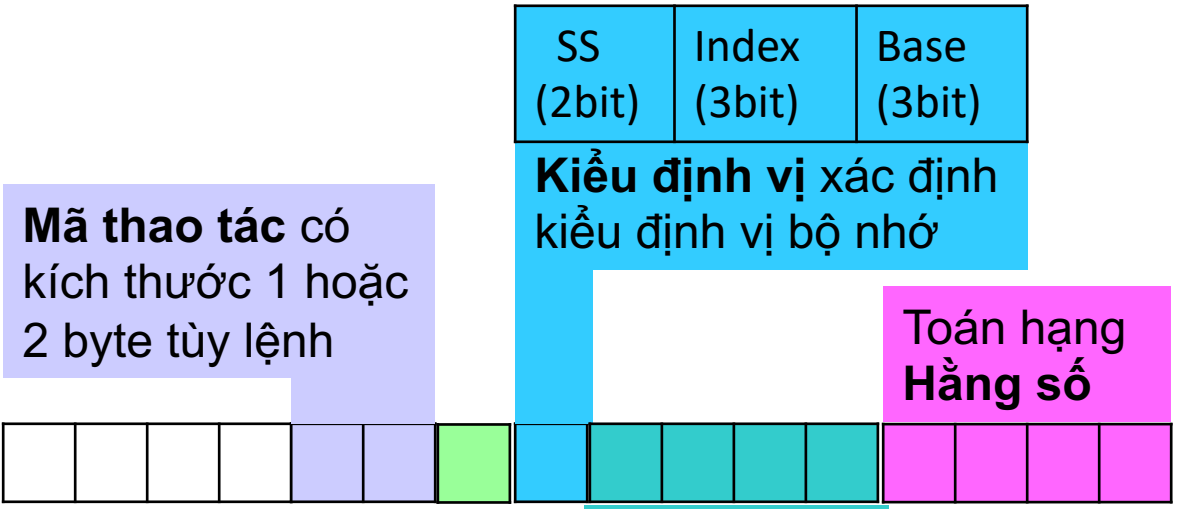
- Flag Register (EFLAGS – 32 bit)



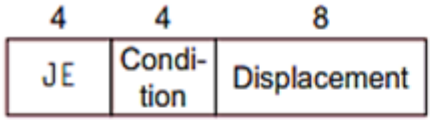
  - 6 bits are used to be status flags:
    - C/CF (carry flag)): CF=1
    - P/PF (parity flag): PF=1 (0) when the number of 1's bit in the result is even (odd)
    - A/AF (auxilary carry flag): extended carry flag
    - Z/ZF (zero flag): ZF=1 when the result is 0
    - S/SF (Sign flag): SF=1 when the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

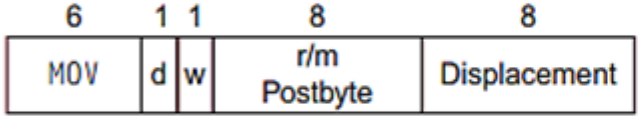


a. JE EIP + displacement

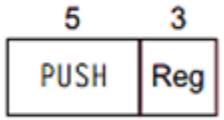


b. CALL

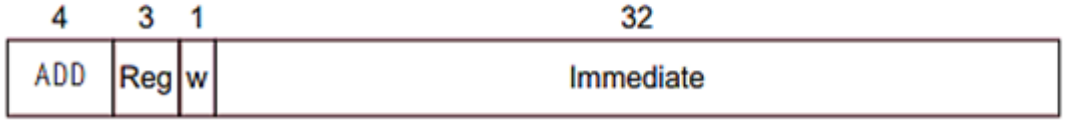
c. MOV EBX, [ESI + 45]



d. PUSH ESI



e. ADD EAX, #6765



f. TEST EDX, #42

x86 Assembly Language

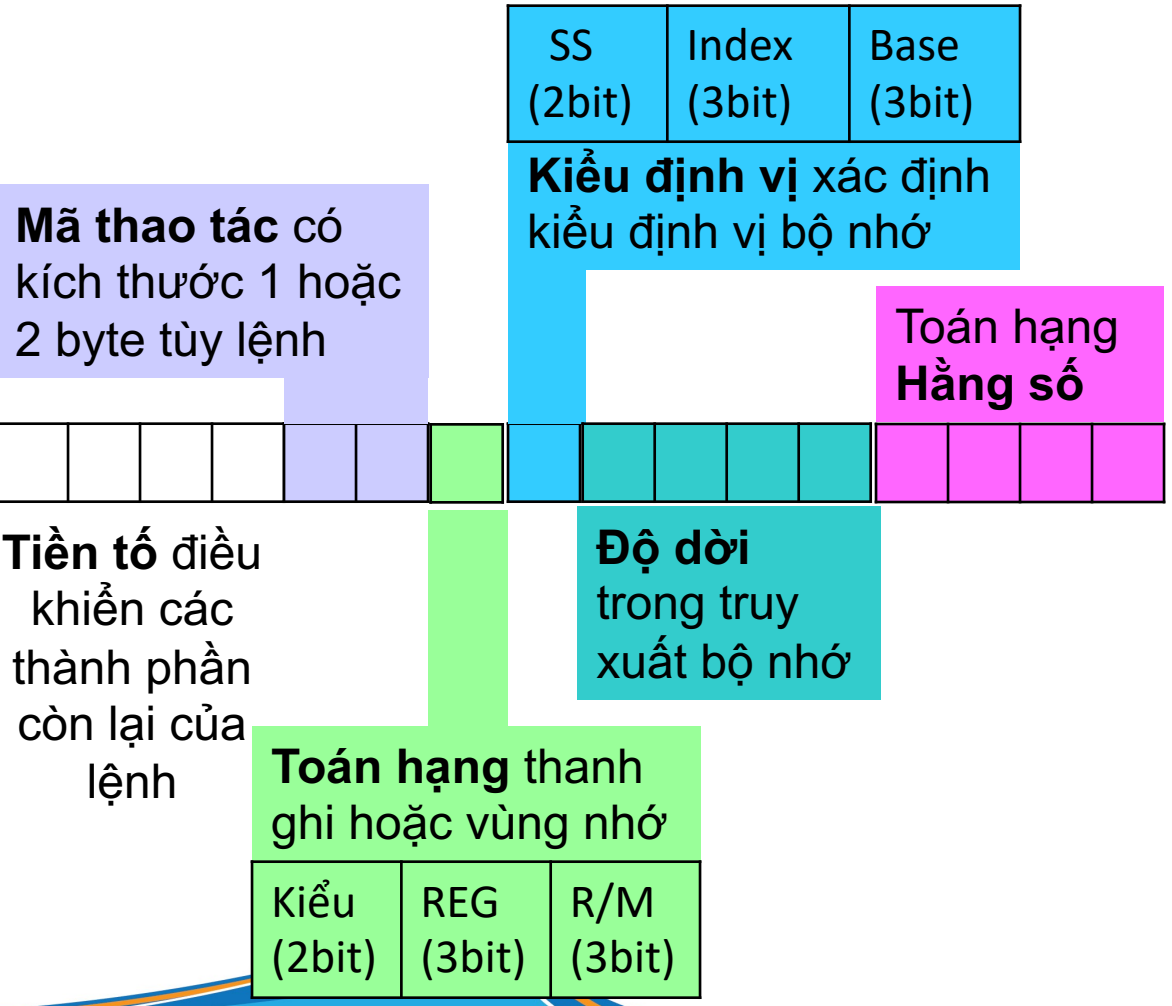
- x86 assembly has two alternative syntaxes available for it
  - Intel
  - AT&T

	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



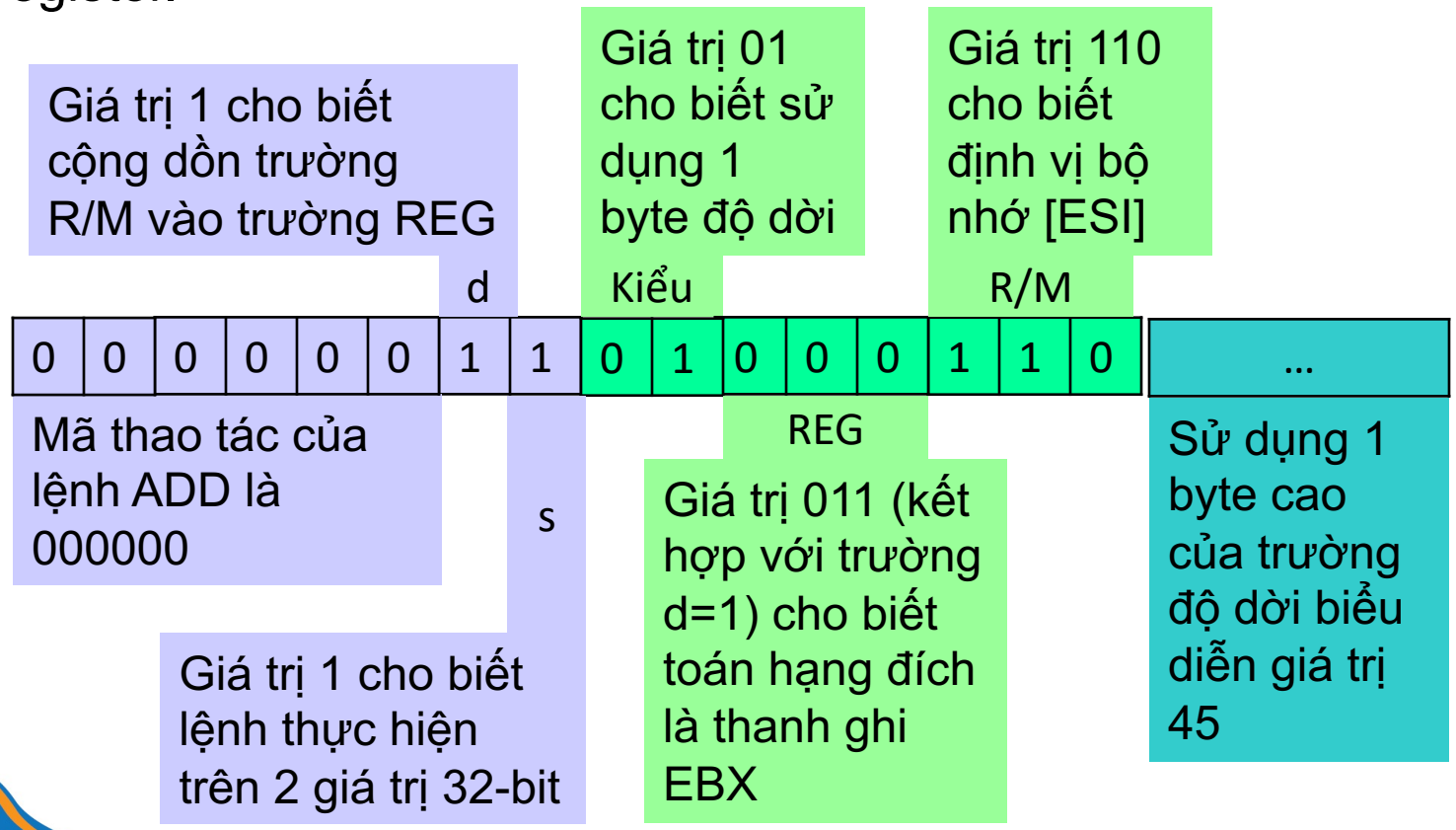
Type	Form	Operand value	Name
Immediate	$\$Imm$	Imm	Immediate
Register	r_a	$R[r_a]$	Register
Memory	Imm	$M[Imm]$	Absolute
Memory	(r_a)	$M[R[r_a]]$	Indirect
Memory	$Imm(r_b)$	$M[Imm + R[r_b]]$	Base + displacement
Memory	(r_b, r_i)	$M[R[r_b] + R[r_i]]$	Indexed
Memory	$Imm(r_b, r_i)$	$M[Imm + R[r_b] + R[r_i]]$	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

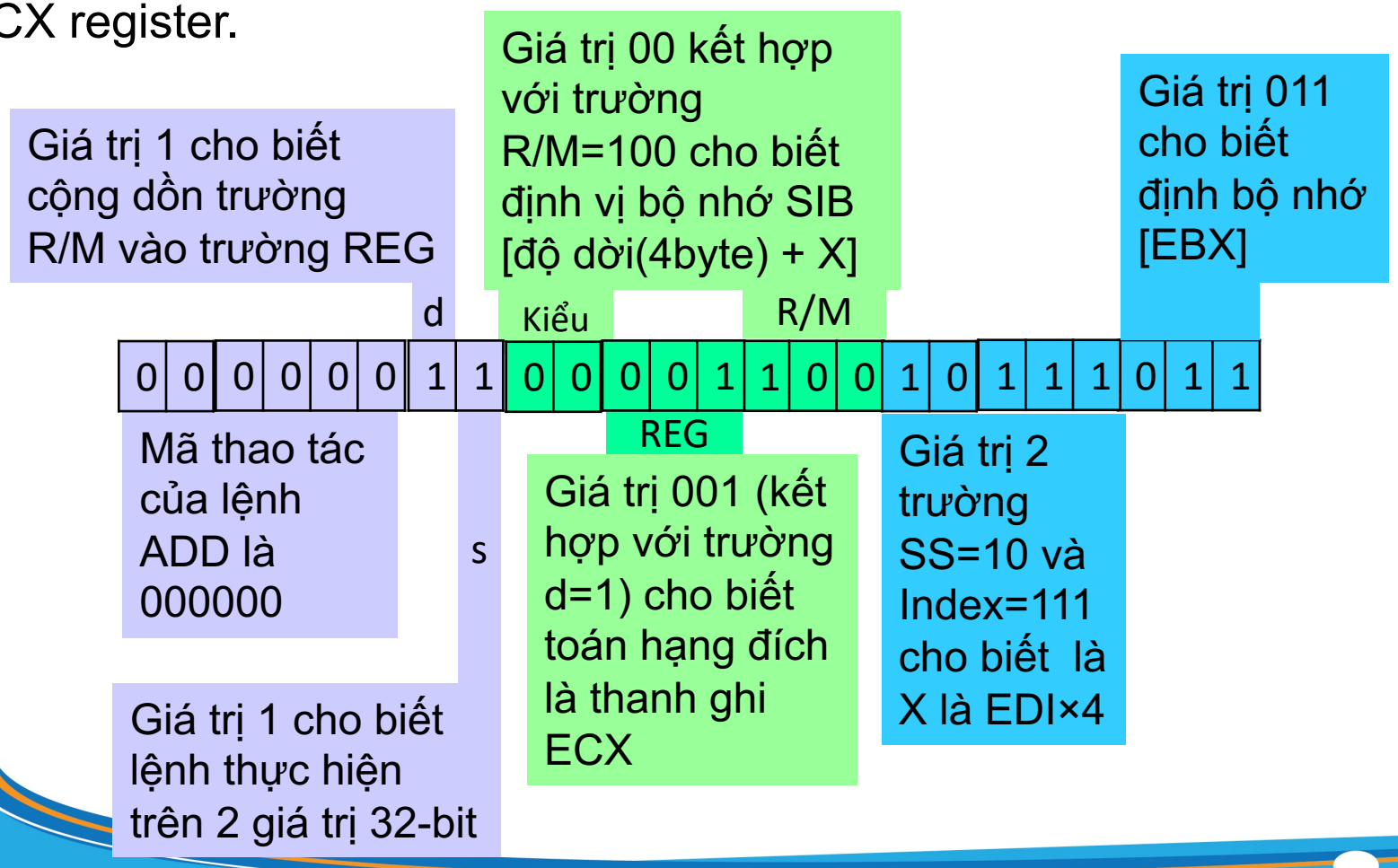
Example of Base + Displacement Addressing

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



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.



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`

OPERATIONS

- 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
```

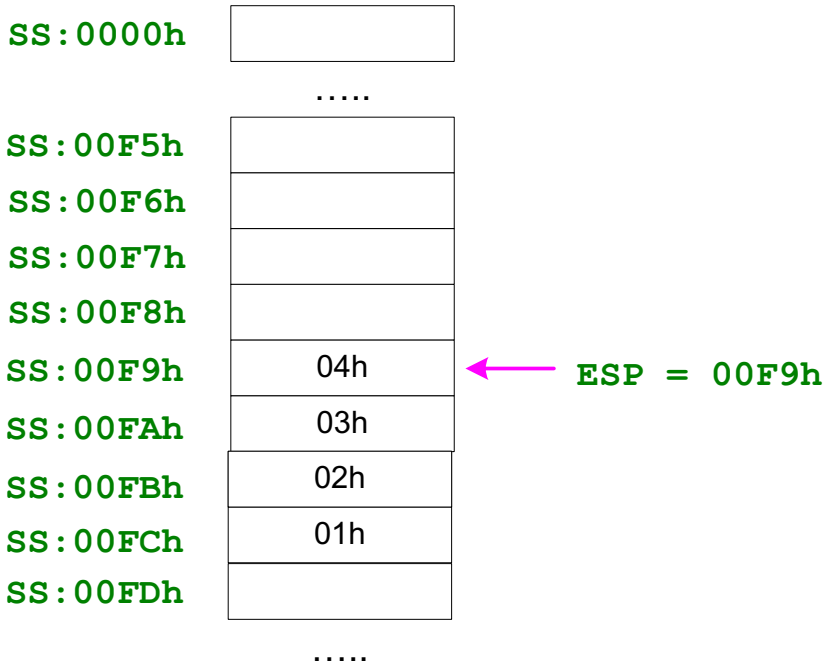
Data movement instructions

- 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>
```

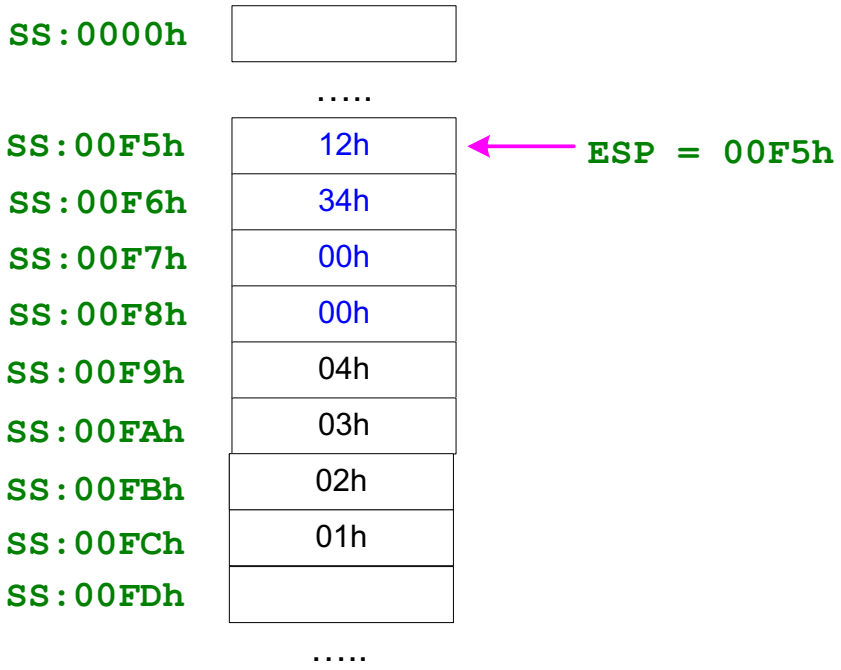
```
Example:
;Push eax on the stack
push EAX
;push the 4 bytes at
address var onto the stack
push [var]
```

PUSH Example



EAX = 3412h

Before PUSH EAX



After PUSH EAX

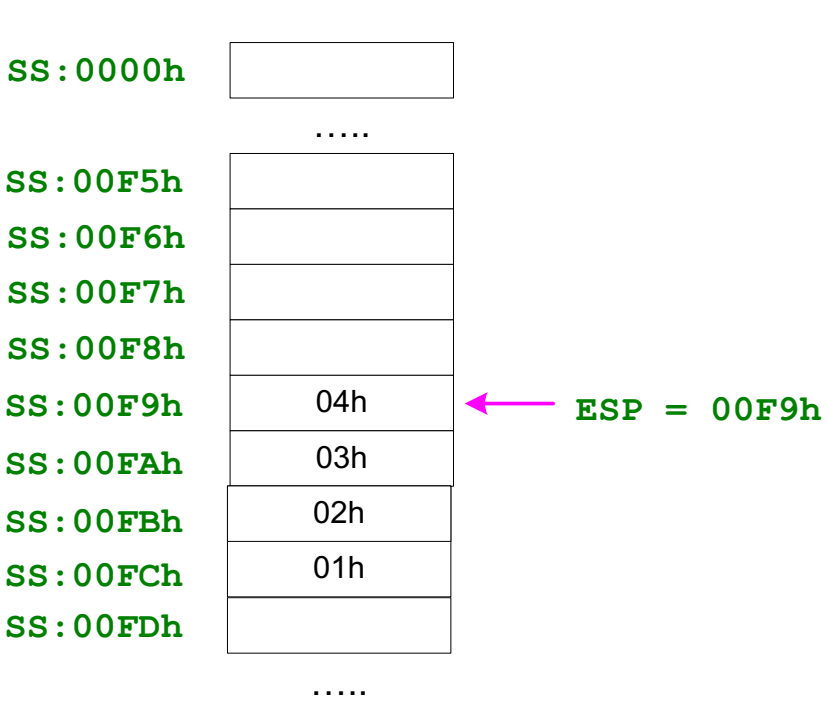
Data movement instructions

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

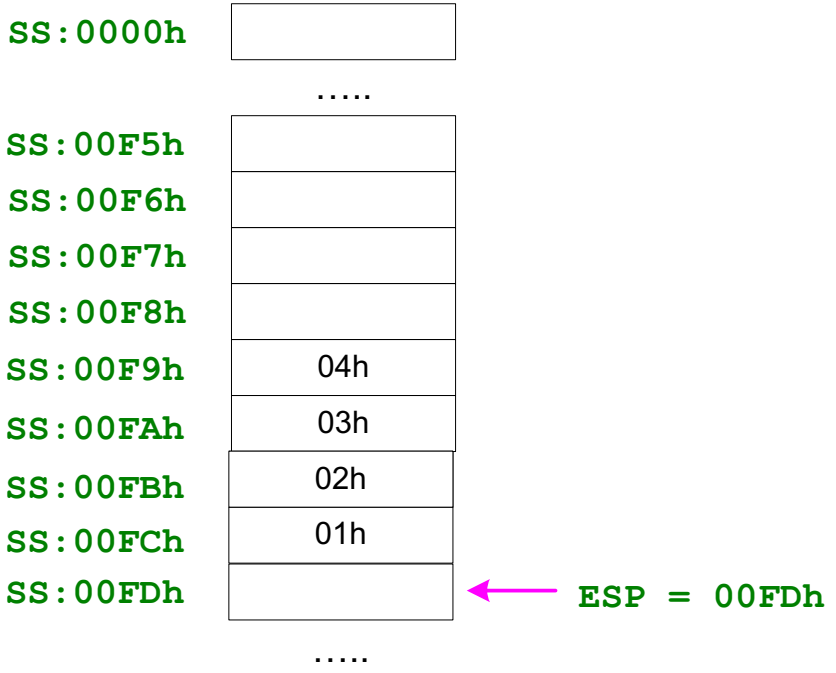
```
Syntax
pop <reg32>
pop <mem>
```

```
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



Before POP EBX



After POP EBX
EBX = 01020304h

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
```

Arithmetic and Logic instructions

- 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>
```

Arithmetic and Logic instructions

- 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
```

Arithmetic and Logic instructions

- 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]
```

Arithmetic and Logic instructions

- 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 2^n 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
```

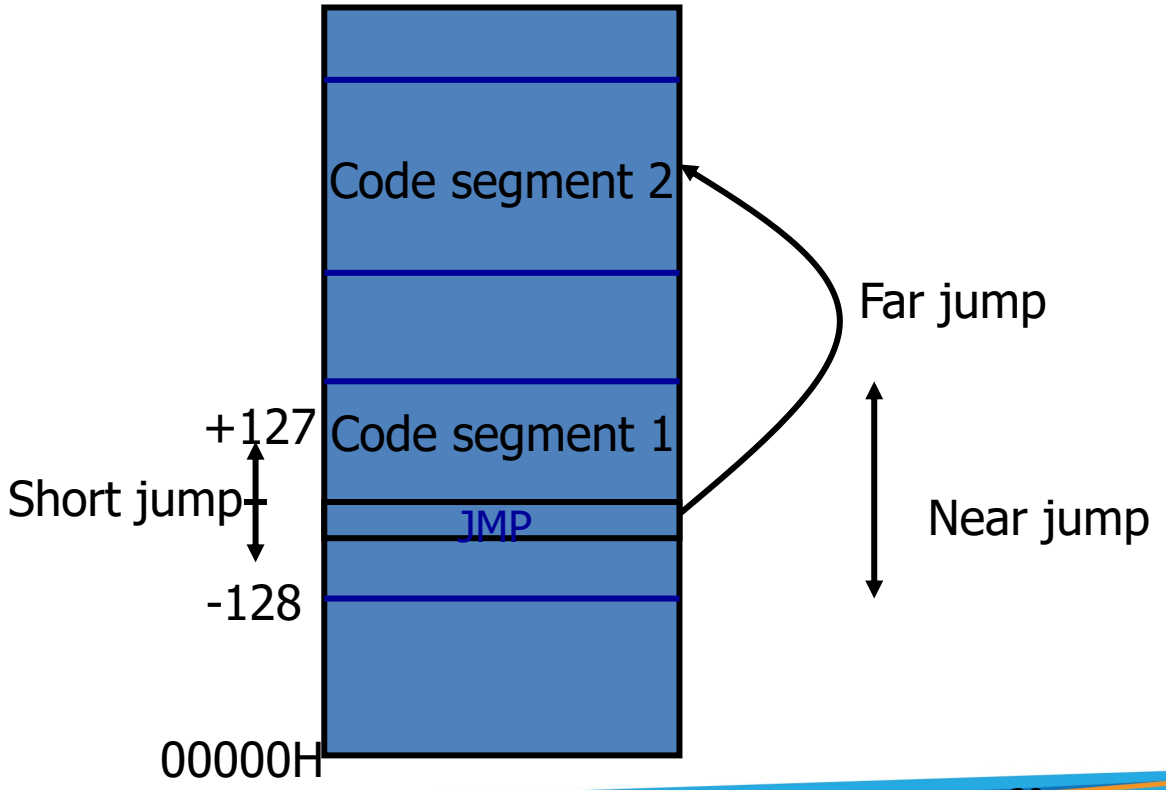
```
    MOV     AL, 36h
    AND     AL, 0Fh ; AL = 06h
                    ; AL = 00000110b
    AND     AL, 00000010b   ; AL = 02h
    OR      AL, 30h         ; AL = 32h
    XOR     AL, AL          ; AL = 0
    NOT     AL              ; AL = FFh
    MOV     AX, 1234h
    MOV     CL, 4
    SHR     AX, 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
```

Control flow instructions

- 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)



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
opcode <Label>
```

```
Example:
;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

```
If (AX==0)
    AX = AX + 1;
BX = AX;
```

ASM (2)

```
    CMP AX, 0
    JNE TIEP
    INC AX
TIEP:
    MOV BX, AX
```

ASM (1)

```
    CMP AX, 0
    JE CONG
    JMP TIEP
CONG:
    INC AX
TIEP:
    MOV BX, AX
```

C Language

```
If (AX<0)
    AX = AX + 1;
Else
    AX = AX – 1;
BX = AX;
```

ASM (2)

```
    CMP AX, 0
    JNL LONHON
    INC AX
    JMP TIEP
LONHON:
    DEC AX
TIEP:
    MOV BX, AX
```

ASM (1)

```
    CMP AX, 0
    JL NHOHON
    DEC AX
    JMP TIEP
NHOHON:
    INC AX
TIEP:
    MOV BX, AX
```

Example

C Language

```
If (AL==‘S’)
        printf (“Chao buoi sang”);
else if (AL==‘T’)
        printf (“Chao buoi trua”);
else if (AL==‘C’)
        printf (“Chao buoi chiều”);
```

ASM (2)

```
        CMP AL, 'S'
        JNE KP_SANG
        ; xuất thông báo
        ; “Chao buoi sang”
        ;
        JMP THOAT
KP_SANG:
        CMP AL, 'T'
        JNE KP_TRUA
        ; xuất thông báo
        ; “Chao buoi trua”
        ;
        JMP THOAT
KP_TRUA:
        CMP AL, 'C'
        JNE THOAT
        ; xuất thông báo
        ; “Chao buoi chieu”
        ;
THOAT:
```

ASM (1)

```
        CMP AL, 'S'
        JE CHAO_BUOI_SANG
        CMP AL, 'T'
        JE CHAO_BUOI_TRUA
        CMP AL, 'C'
        JE CHAO_BUOI_CHIEU
        JMP THOAT
CHAO_BUOI_SANG:
        ; xuất thông báo
        ; “Chao buoi sang”
        ;
        JMP THOAT
CHAO_BUOI_TRUA:
        ; xuất thông báo
        ; “Chao buoi trua”
        ;
        JMP THOAT
CHAO_BUOI_CHIEU:
        ; xuất thông báo
        ; “Chao buoi chieu”
        ;
THOAT:
```

Example

C Language

```
If (AL>='a' and AL<='z')
        AX = AX + 1;
else
        AX = AX – 1;
BX = AX;
```

ASM (2)

```
        CMP AL, 'a'
        JB KPTHUONG
        CMP AL, 'z'
        JA KPTHUONG
        INC AX
        JMP TIEP
KPTHUONG:
        DEC AX
TIEP:
        MOV BX, AX
```

ASM (1)

```
        CMP AL, 'a'
        JAE CTTHUONG
        DEC AX
        JMP TIEP
CTTHUONG:
        CMP AL, 'z'
        JBE THUONG
        DEC AX
        JMP TIEP
THUONG:
        INC AX
TIEP:
        MOV BX, AX
```

ASM (3)

```
        CMP AL, 'a'
        JB KPTHUONG
        CMP AL, 'z'
        JBE THUONG
KPTHUONG:
        DEC AX
        JMP TIEP
THUONG:
        INC AX
TIEP:
        MOV BX, AX
```

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
    .
    .
    ret
<Procedure Name> ENDP
```

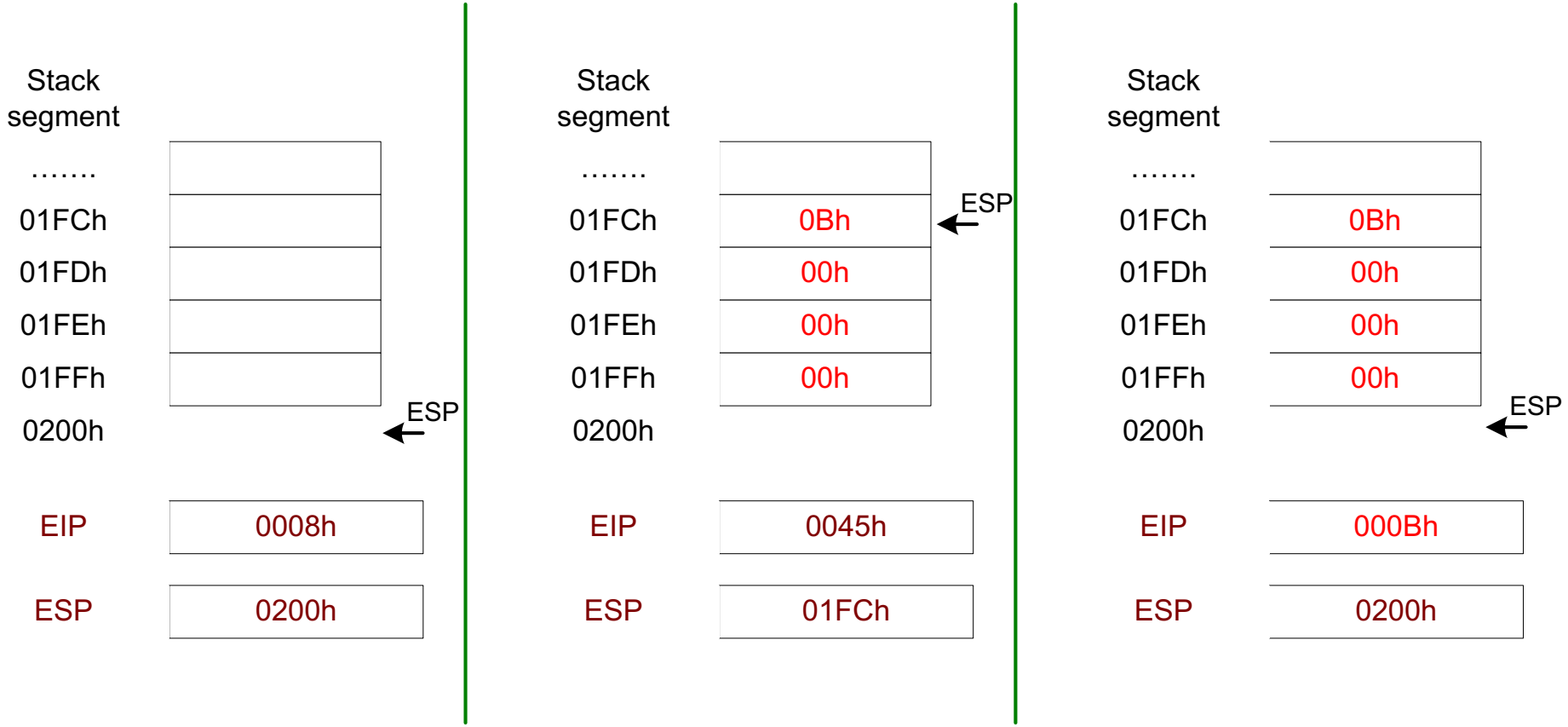
```
sample PROC
    .
    .
    ret
sample ENDP
```

- 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

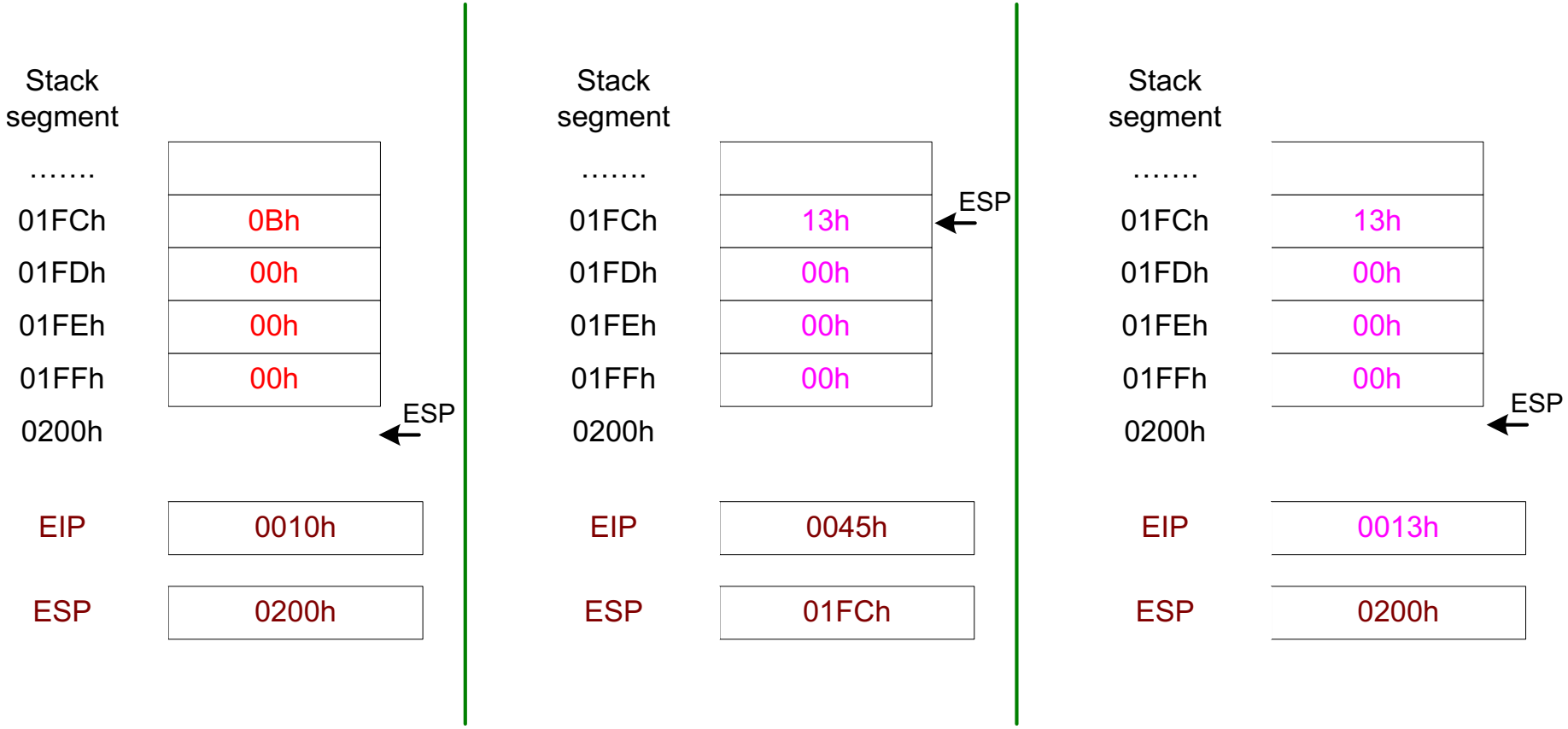


Before CALL

After CALL

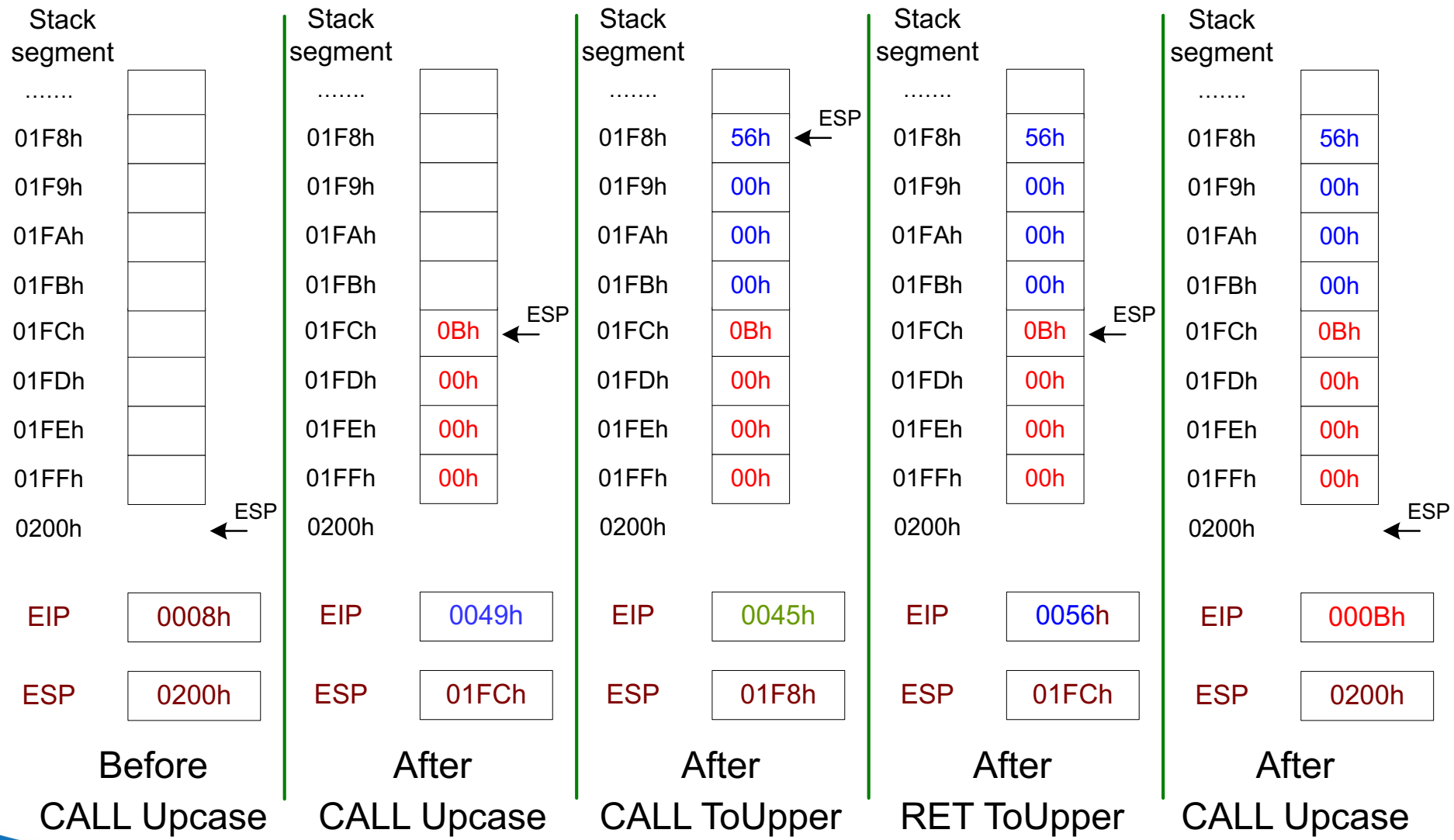
After RET

Explanation of the second call to *ToUpper* procedure



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                   0056h
Upcase ENDP
   ...
```



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 ⇒ 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 ⇒ 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 ⇒ 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 ⇒ Smaller source code

Compare 32-bit MIPS and x86 instructions

- MIPS: "Fixed Length Instructions"
  - All instructions are 4 bytes in size
  - Simpler processing circuit ⇒ Faster processing
  - Jump instructions: multiple of 4 bytes

- x86: "Variable length instructions"
  - Instruction size varies from 1 byte to 16 bytes

  ⇒ Source code can be smaller (30% ?)
  - Use cache more efficiently
  - Instructions can have 8-bit or 32-bit constant/immediate