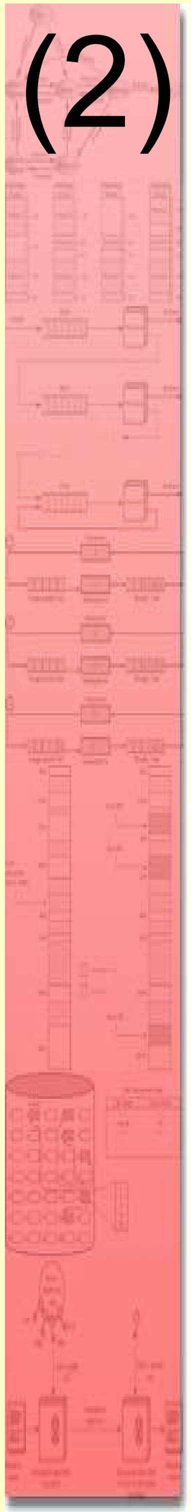
(2)



Escuela Politécnica Superior

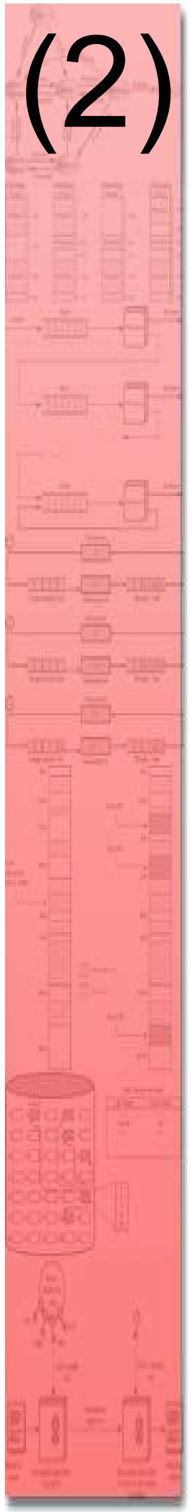
UAM Universidad Autónoma de Madrid

Unit 2 **Programming Model of the Intel 80x86**

MICROPROCESSOR-BASED SYSTEMS

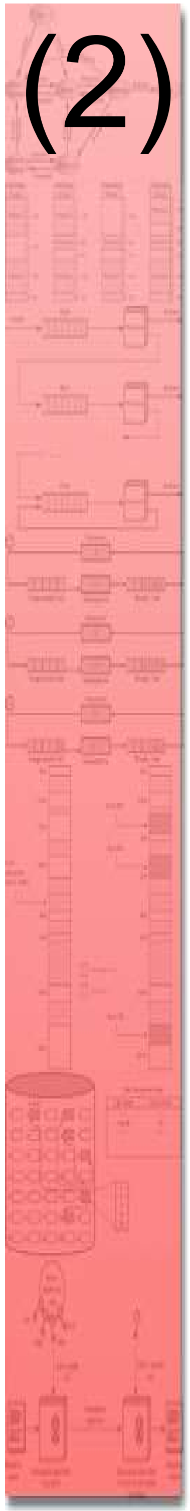
Degree in Computer Science Engineering
Double Degree in Computer Engineering and Mathematics

EPS - UAM



(2)

Index

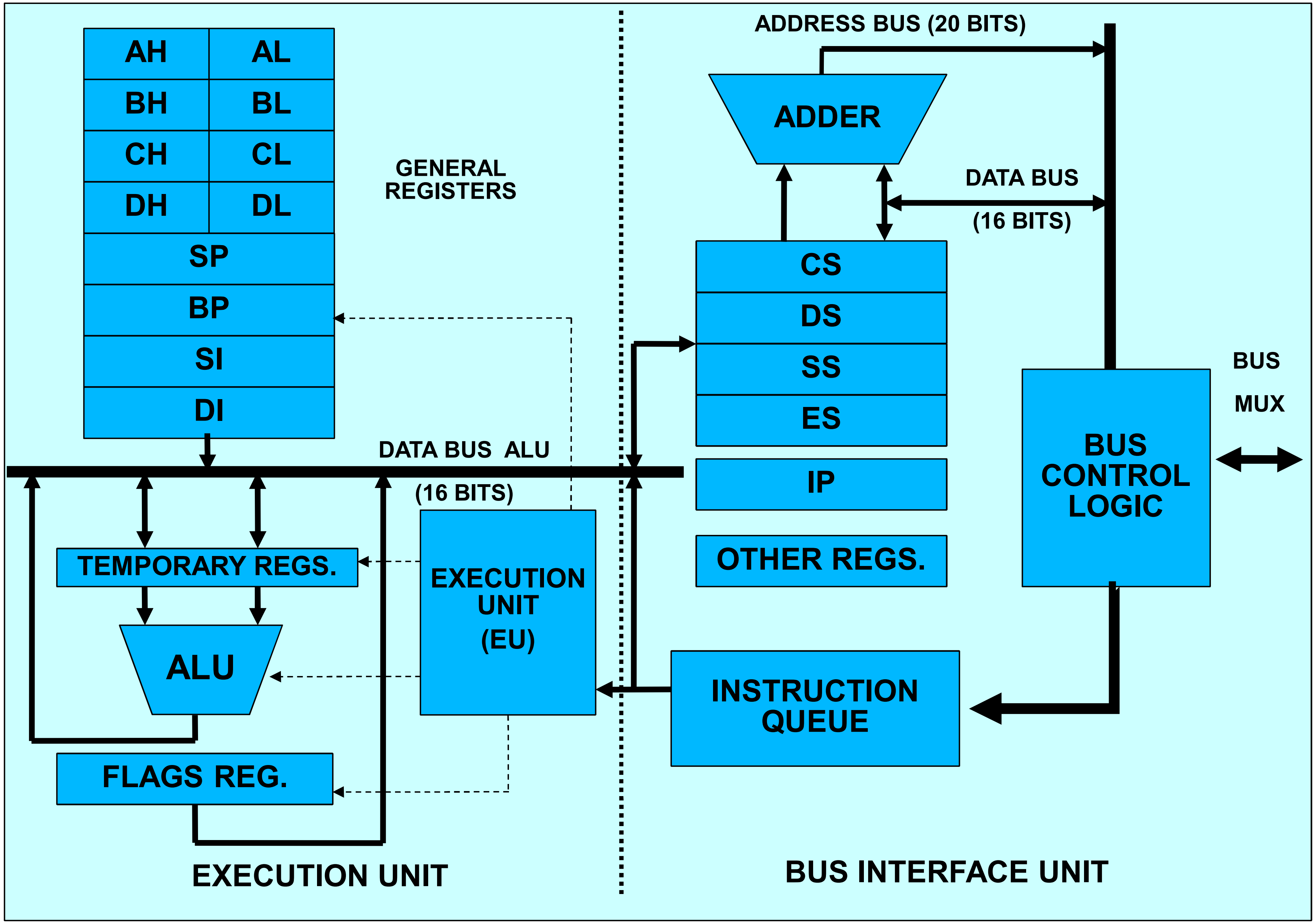


2.1. The 80x86 family as a particular case

- Microprocessors appear in the 70s (1971-…) with 4 bits and later 8 bits (**8085** with 64KB memory).
- Invented by **Intel** as integrated and programmable digital circuits in order to substitute discrete digital circuits.
- The 80x86 family was born in 1978 with the **8086** (16 bits and 1 MB memory). It proceeds with: **80186**, **80286**, **80386**, …
- In parallel, the **8088** was delivered (IBM's personal computer or PC): **8086** of 8 bits.
- Initial competitor: **Motorola 6800** (8 bits) and **68000** (16 bits).
- Intel guarantees compatibility of its microprocessors since the beginning and introduces memory segmentation (64 KB segments)
- Technologies: **CISC** vs. **RISC** (more modern)

(2)

2.2. Internal registers and architecture of the 80x86 (I)

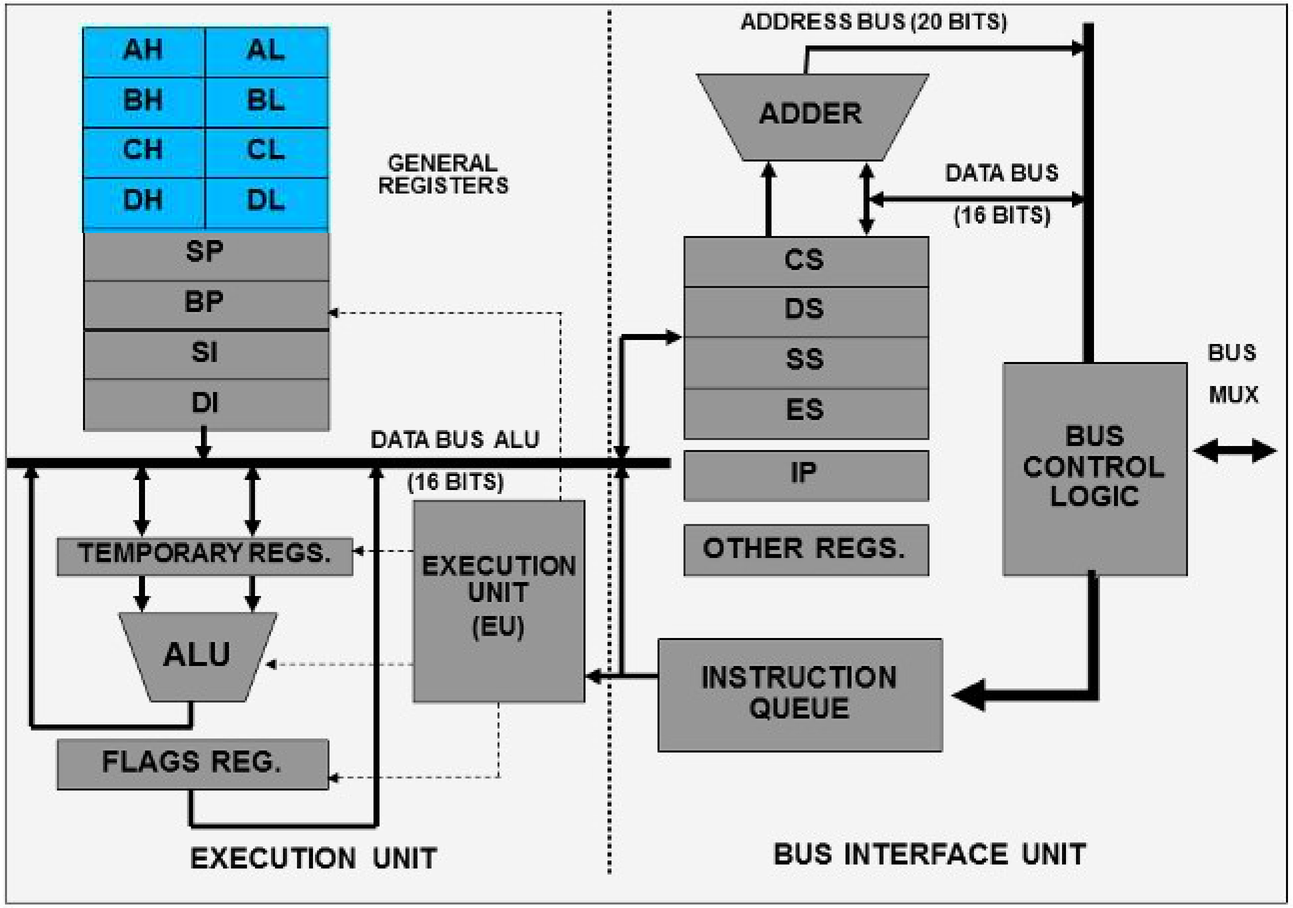


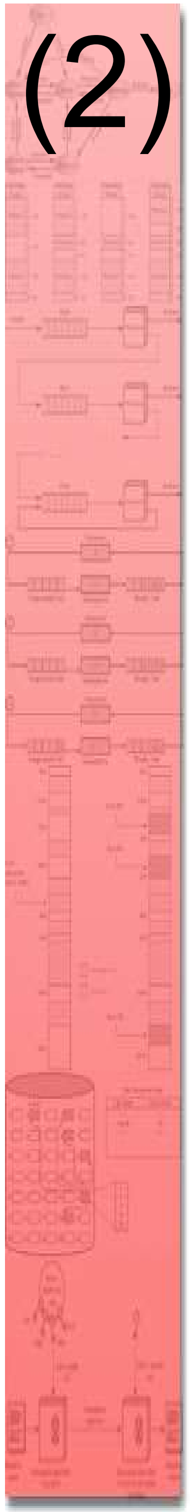
(2)

2.2. Internal registers and architecture of the 80x86 (II)

- Data registers

  A**X** (A**H**-A**L**), B**X** (B**H**-B**L**), C**X** (C**H**-C**L**), D**X** (D**H**-D**L**)





(2)

2.2. Internal registers and architecture of the 80x86 (III)

- Data registers

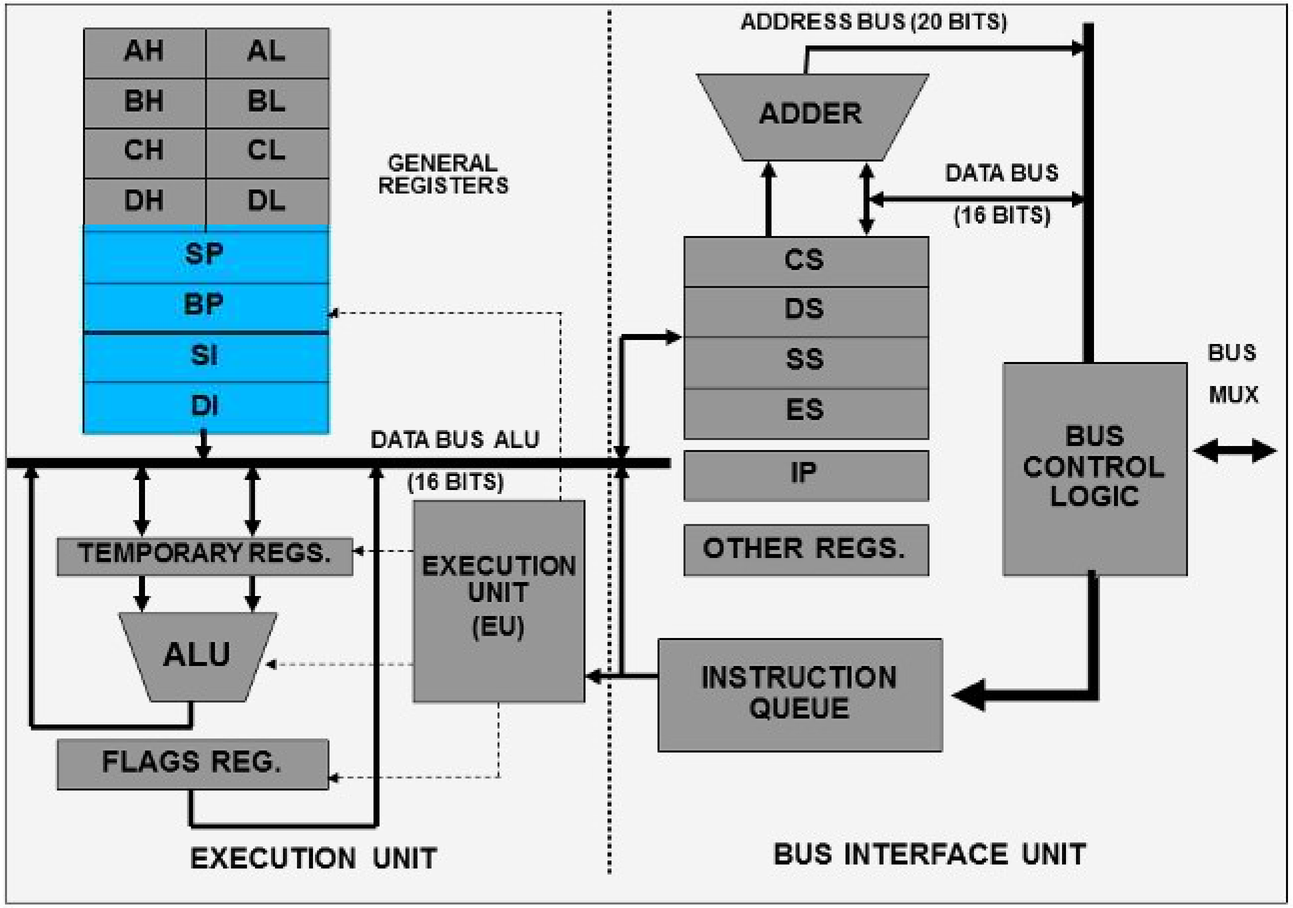
  A**X** (A**H**-A**L**), B**X** (B**H**-B**L**), C**X** (C**H**-C**L**), D**X** (D**H**-D**L**)

  - They behave as accumulators in transfer, logic and arithmetic instructions.
  - Each of 16 bits, divisible into 2 registers of 8 bits.
  - Specific tasks in some cases (for any use if they are free):
    - **AX**: Multiply, divide and I/O operations.
    - **BX**: Base register for indirect addressing (pointer to the base of a table)
    - **CX**: Loop counter.
    - **DX**: Multiply, divide and I/O operations.

(2)

2.2. Internal registers and architecture of the 80x86 (IV)

- Pointer registers: **SP , BP , SI , DI**



(2)

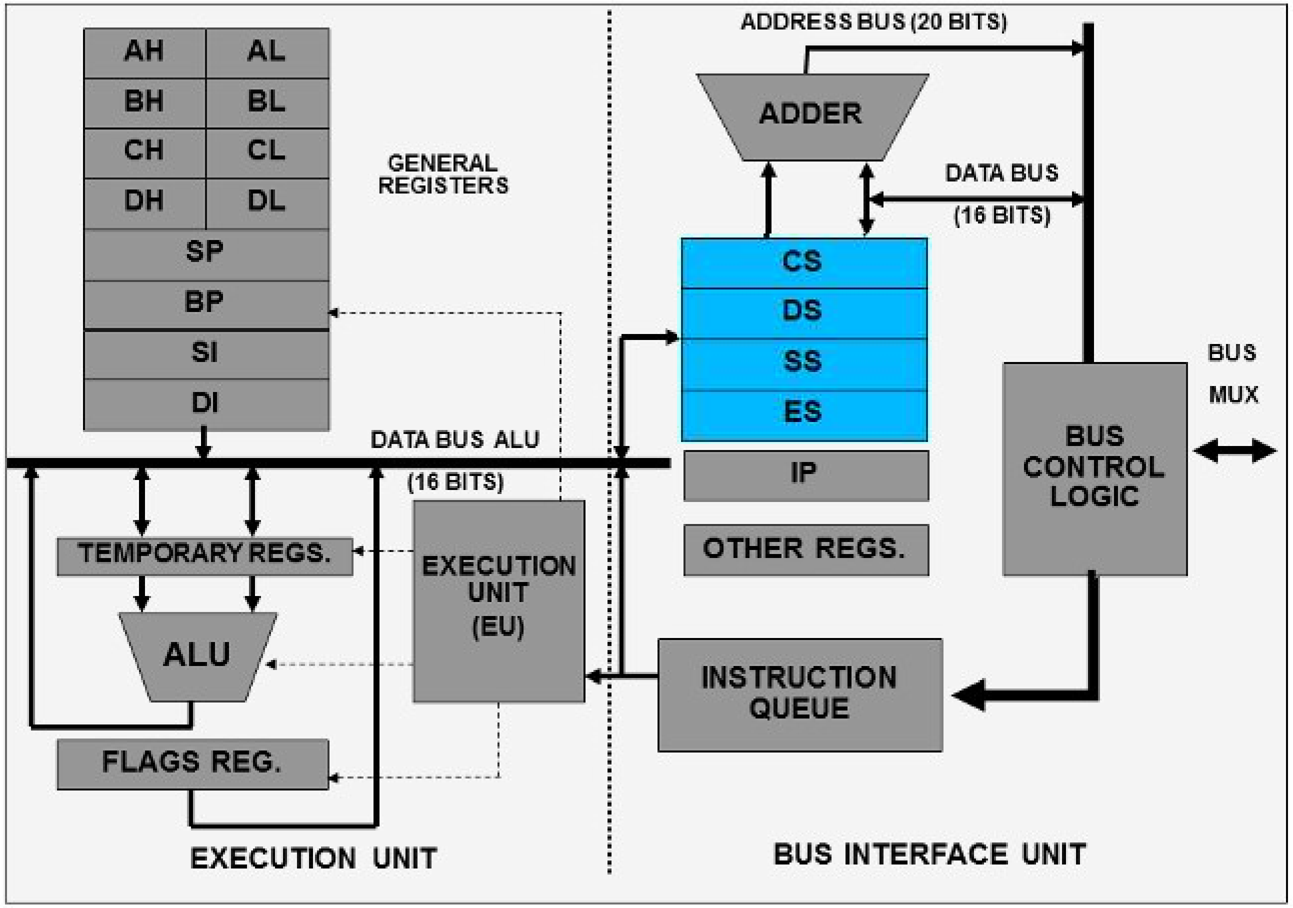
2.2. Internal registers and architecture of the 80x86 (V)

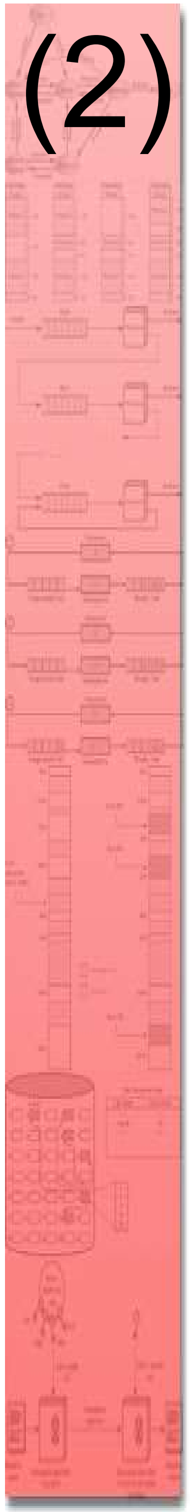
- Pointer registers: **SP , BP , SI , DI**
  - Involved in the memory addressing as displacements (*offsets*) with respect to the memory zones indicated in segment registers.
  - **SP** (*Stack Pointer*): Used in conjuntction with the stack segment register **SS**. Involved in:
    - Subroutine calls
    - Interrupts
    - Stack management instructions
  - **BP** (*Base Pointer*): Used in conjunction with the stack segment register **SS**. Useful for accessing the parameters of subroutines passed through the stack.
  - **SI** (*Source Index*): Used for indexing memory tables (reading). For any use if it is free.
  - **DI** (*Destination Index*): Used for indexing memory tables (writing). For any use if it is free.

(2)

2.2. Internal registers and architecture of the 80x86 (VI)

- Segment registers: **CS , SS , DS , ES**




(2)

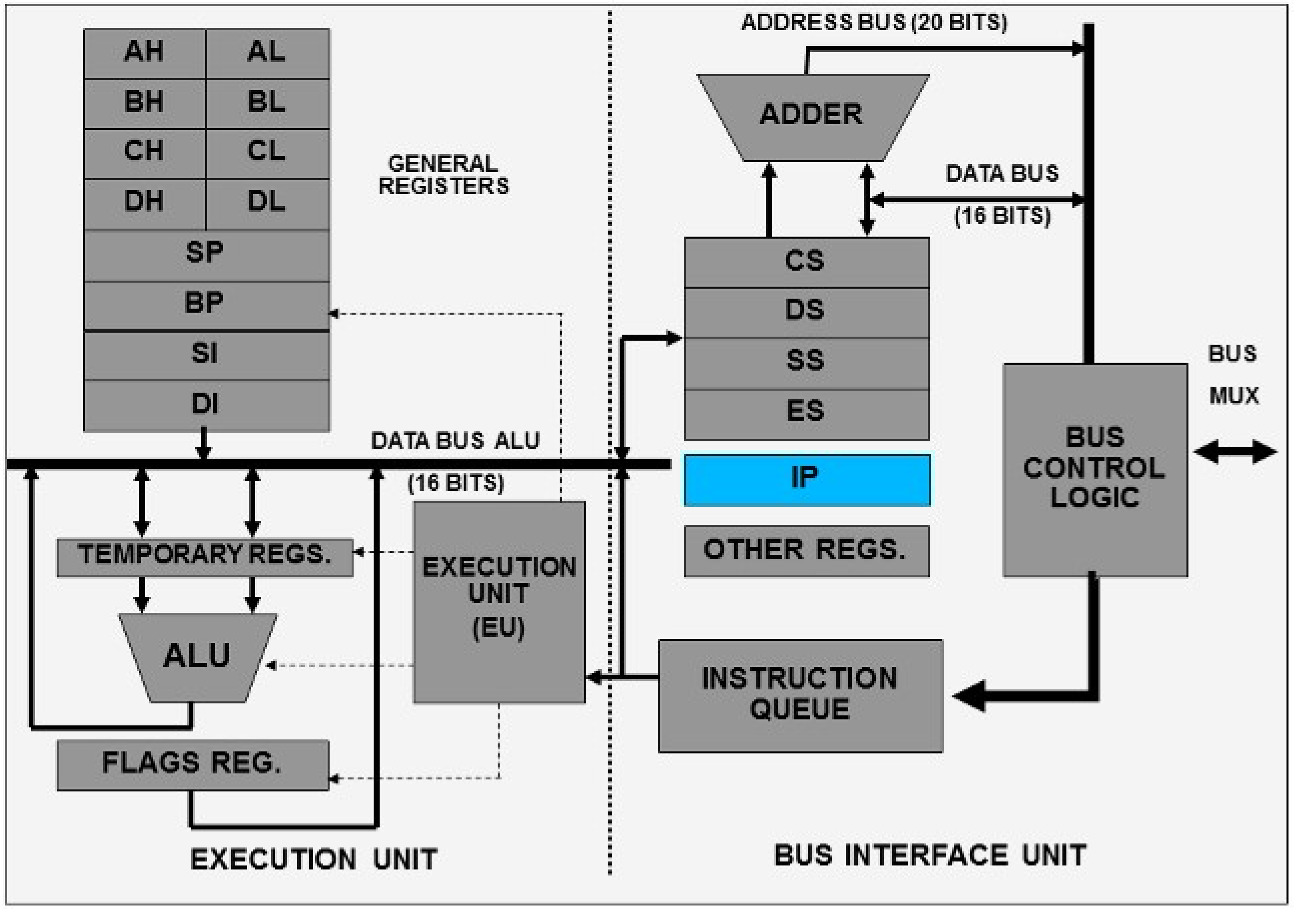
2.2. Internal registers and architecture of the 80x86 (VII)

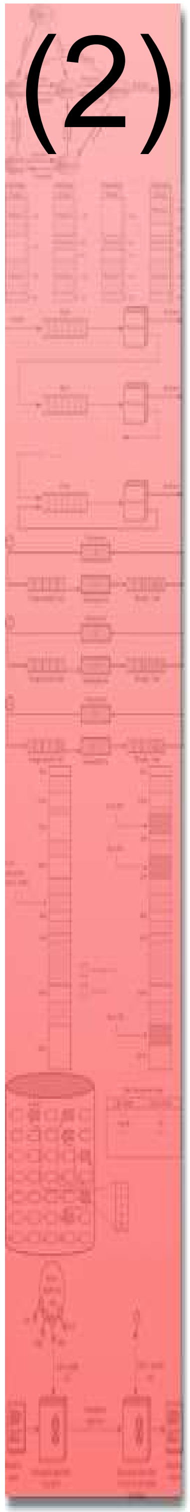
- Segment registers: **CS , SS , DS , ES**
  - Involved in the memory addressing pointing to 64KB areas of memory (segments).
  - **CS** (*Code Segment*): Pointer to the machine code segment (program). Along with the instruction pointer **IP**, it constitutes the *program counter*.
  - **SS** (*Stack Segment*): Pointer to the stack segment. Along with **SP** or **BP**, it points to an absolute memory position in the stack.
  - **DS** (*Data Segment*): Pointer to the main data segment (global variables).
  - **ES** (*Extra Segment*): Pointer to the additional data segment (global variables).

(2)

2.2. Internal registers and architecture of the 80x86 (VIII)

- Instruction pointer register: **IP**



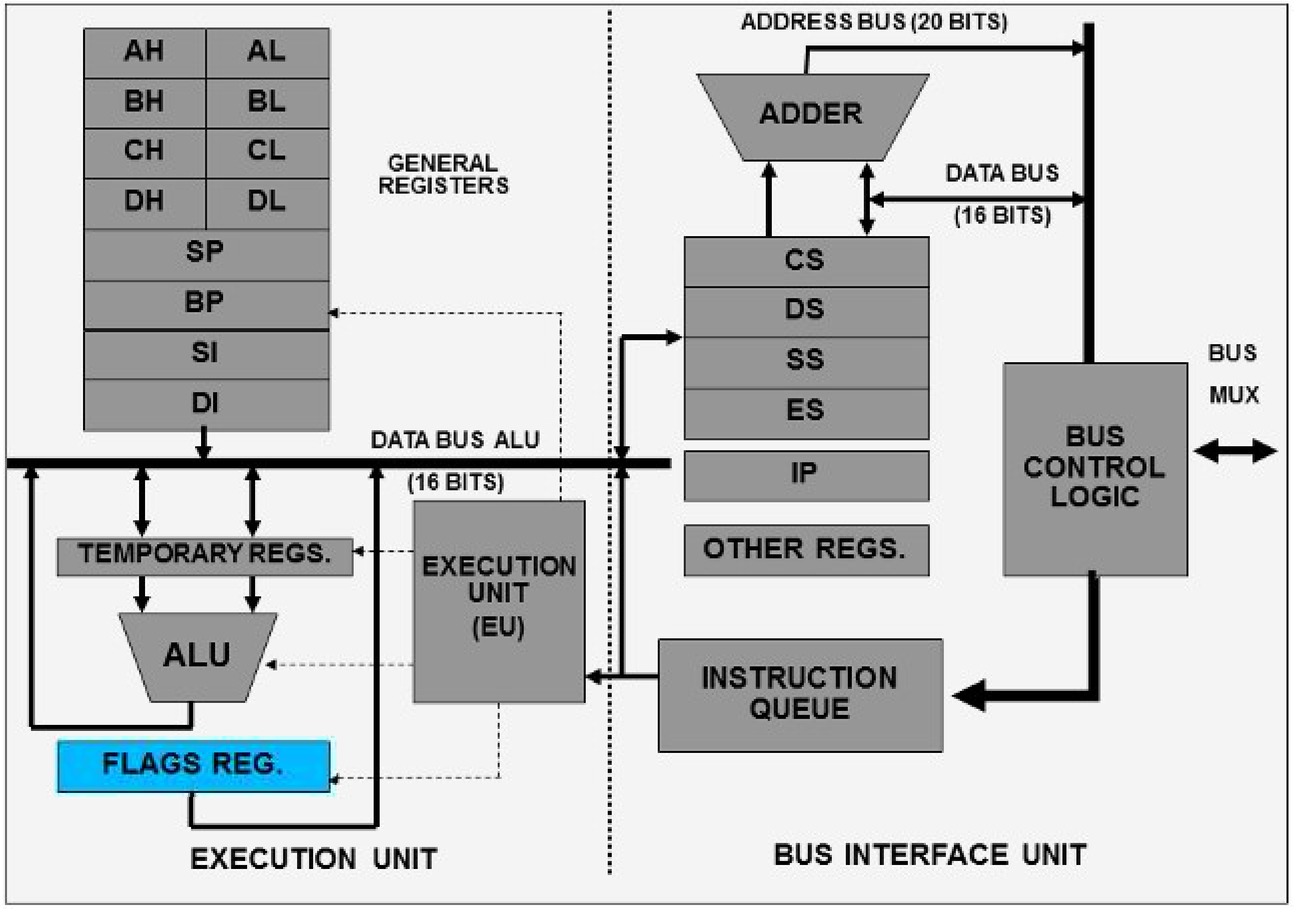


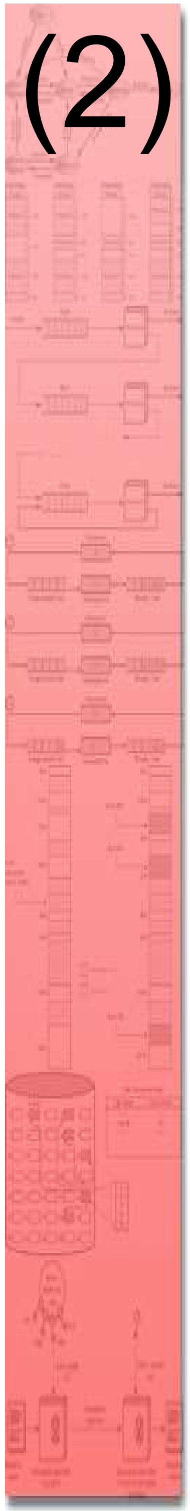
2.2. Internal registers and architecture of the 80x86 (IX)

- Instruction pointer register: **IP**
  - It keeps the displacement (*offset*) within the segment indicated by **CS** where the next machine code instruction to be executed is stored (program counter).

2.2. Internal registers and architecture of the 80x86 (X)

- State register (*FLAGS*)





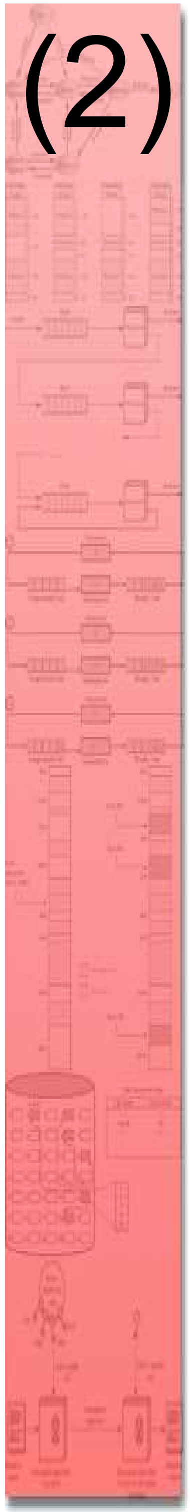
2.2. Internal registers and architecture of the 80x86 (XI)

- State register (*FLAGS*)
  - Some of its 16 bits indicate information about the current state of the processor and the last ALU operation.

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
				OF	DF	IF	TF	S	Z		AC		P		C

IF: Interrupt bit	**Z**: Zero bit	**C**: Carry bit
DF: Direction bit	**S**: Sign bit	**P**: Parity bit
OF: Overflow bit	TF: Trap bit	AC: Auxiliary carry bit

  - Flags **C**, **AC**, **S**, **P**, **Z** and **OF** depend on the last operation executed by the ALU.
  - Flag **IF** enables or disables the hardware interrupts.
  - Flag **TF** enables or disables the “step by step” execution.
  - Flag **DF** increments or decrements the index pointers in string instructions.
  - All bits can be set to **0** or **1** with specific instructions.



2.3. Memory access and organization (I)

- Physical memory on the 8086 organized as 2^{20} positions of 1 byte (1 MB).
- 1MB physical memory divided into logical "segments" of 64 KB.
- Segments start at multiples of 16.
- Two consecutive segments are 16 bytes apart.
- In a program, the instructions are usually in a segment, the data in one or several segments and the stack in another segment (this does not happen sometimes).
- The CPU can access up to four different segments simultaneously (registers **CS**, **DS**, **ES** and **SS** with different values).
- Segments can overlap totally or partially (extreme case: **CS**, **DS**, **ES** and **SS** with the same value).
- The program can change the values of the segment registers at any time.

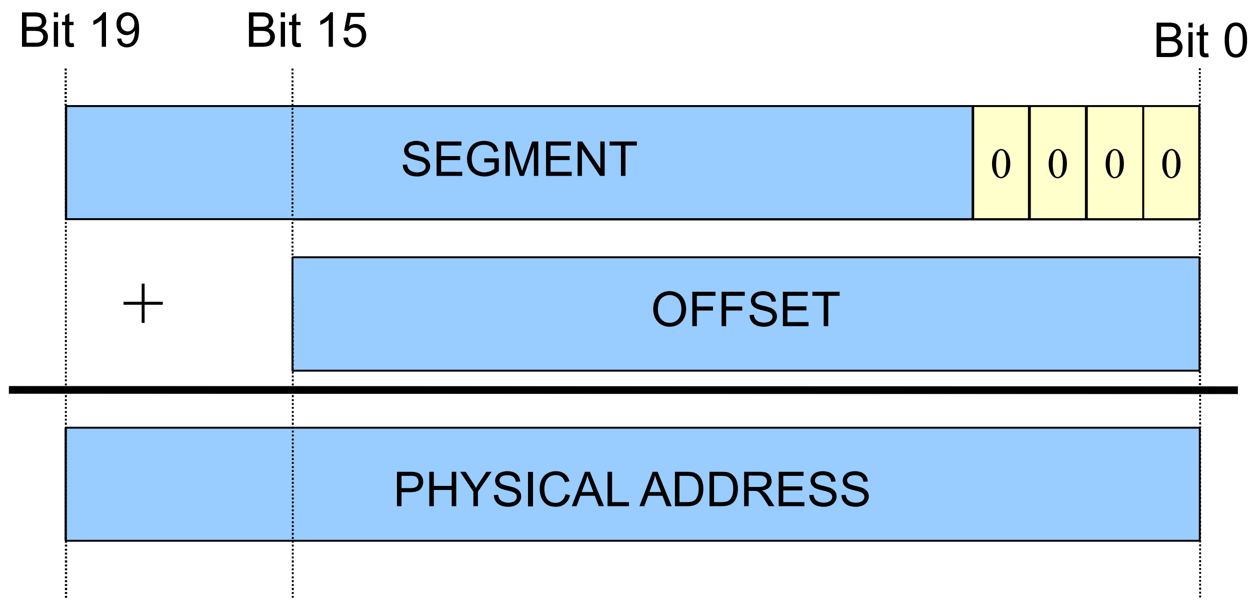
(2)

2.3. Memory access and organization (II)

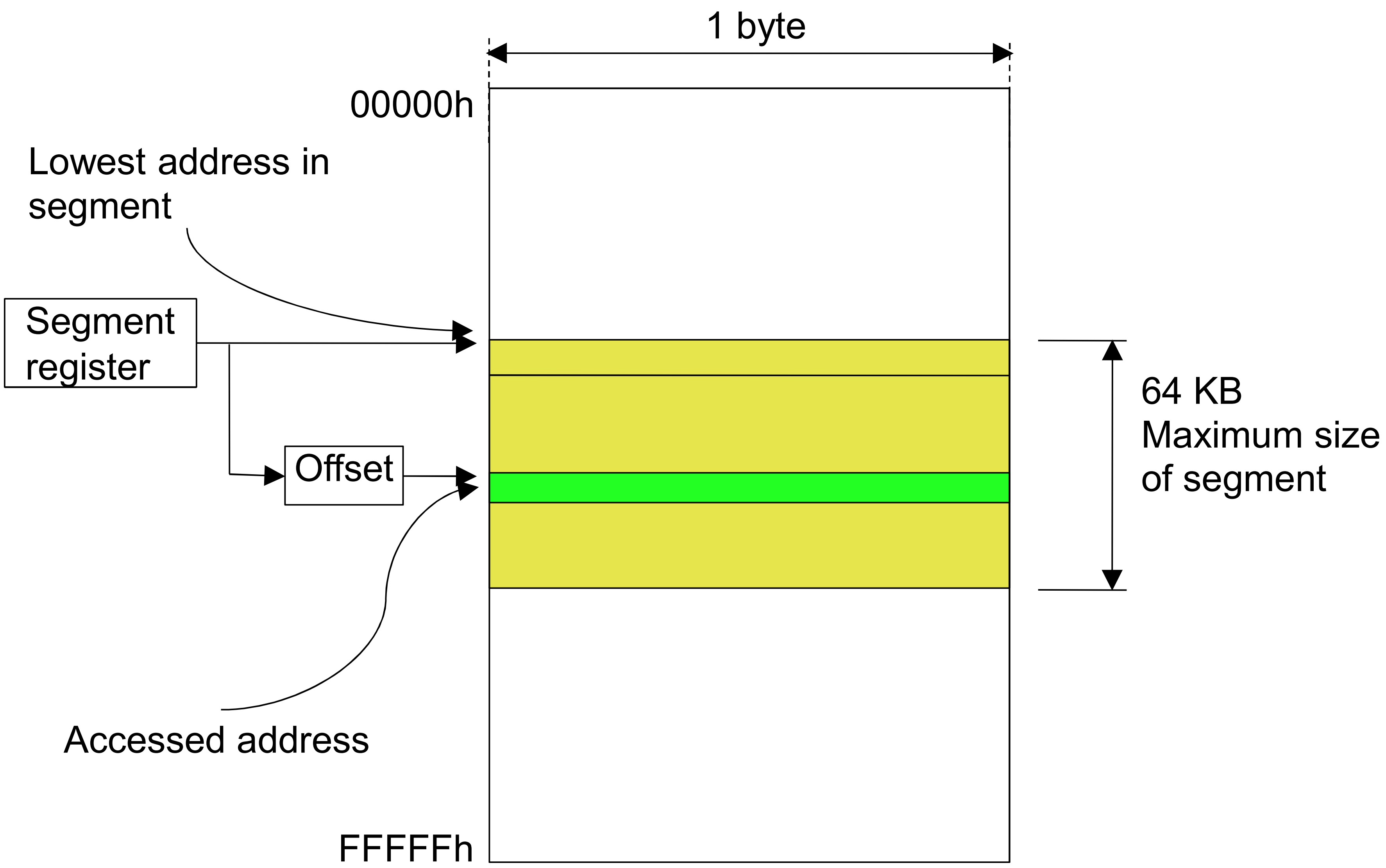
- Memory access (real mode)

  Hardware: 20 address bits (A19-A0)
  Software: 32 bits (16 bits for Segment + 16 bits for Offset)

PHYSICAL ADDRESS = **Segment** x **16** + **Offset**



2.3. Memory access and organization (III)



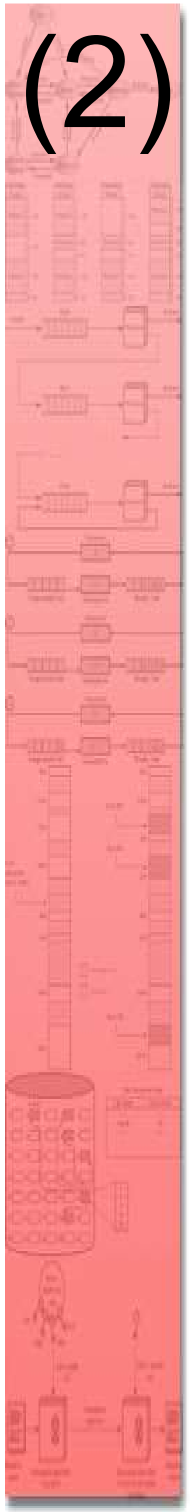
2.3. Memory access and organization (IV)

- Examples of memory access (real mode)
  - ***CS* = A783h** ***(segment)***
    IP* = 403Eh** ***(offset)

    Physical address = **A783h** x **16** + **403Eh** =
    A783h x **10h** + **403Eh =**
    A7830h + **403Eh** = **AB86Eh**
  - ***ES* = 54A3h** ***(segment)***
    DI* = 1F2Bh** ***(offset)

    Physical address = **54A30h** + **1F2Bh** = **5695Bh**
  - ***SS* = 4675h**
    ***SP* = A001h**

    Physical address = **46750h** + **A001h** = **50751h**



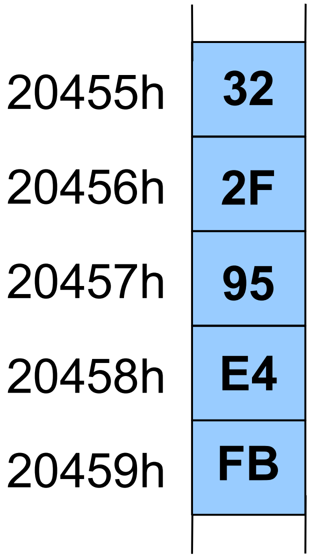
2.3. Memory access and organization (V)

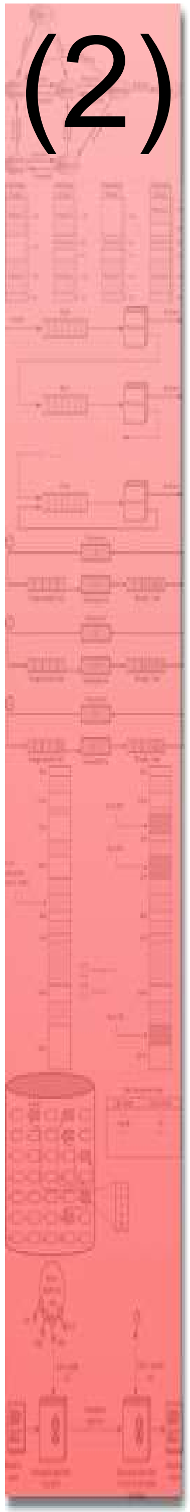
- Memory access from programs
  - CPU can access one byte or two consecutive bytes (one word) according to the register referred in the instruction.
  - ***Example***: If these instructions have previously been executed:

```
mov AX, 2000h
mov DS, AX
```

the result of the following operations is:

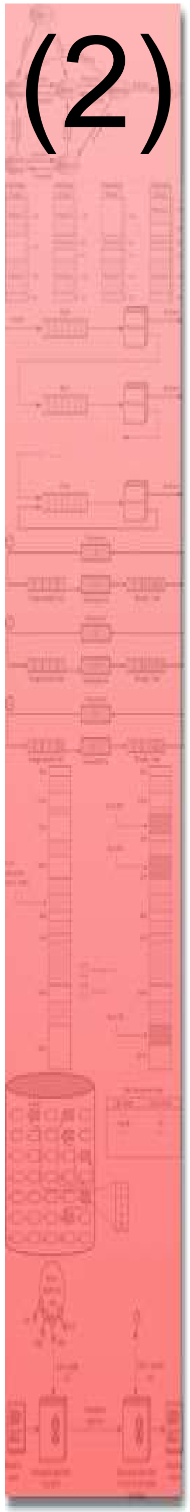
```
mov AX, [455h]        ;  AX = 2F32h
mov AX, [456h]        ;  AX = 952Fh
mov AH, [457h]        ;  AH = 95h
mov AL, [458h]        ;  AL = E4h
```





2.4. Addressing modes (I)

- Seven addressing modes:
  - Immediate
  - Register
  - Direct
  - Indirect
  - Relative
  - Indexed
  - Implicit
- The direct and indirect modes behave as “pointers” to memory.



2.4. Addressing modes (II)

- Immediate addressing

  The source operand is always a value and the destination a register.

  Examples:

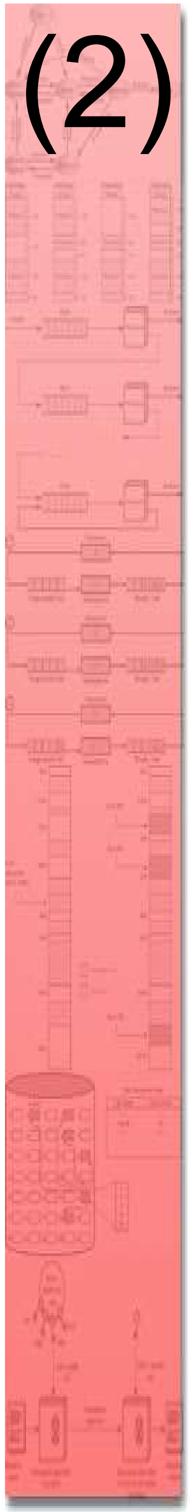
```
mov CL, 3Fh           ; 3Fh ⇒ CL
mov SI, 4567h         ; 4567h ⇒ SI
```

- Register addressing

  Both operands are always registers.

  Examples:

```
mov DX, CX            ; CX ⇒ DX
mov BH, CL            ; CL ⇒ BH
```



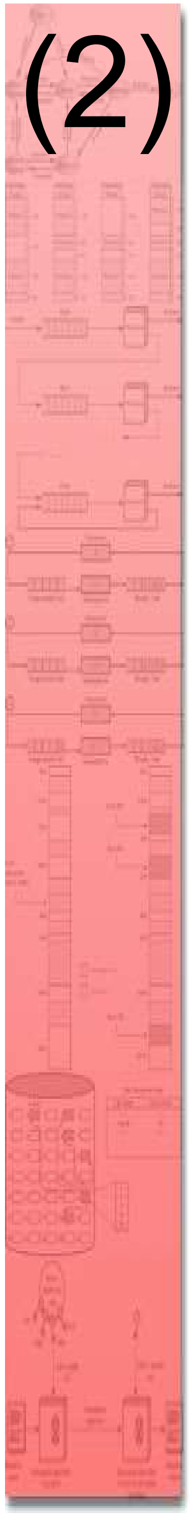
2.4. Addressing modes (III)

- Direct addressing

  The *offset* of the memory position to be accessed is specified in the instruction. By default, the segment indicates **DS**.

  Examples if DS = **3000h**:

```
mov DX, [678Ah]   ;  Load DL with the content of memory
                  ;  position 3678Ah and DH with the content
                  ;  of memory position 3678Bh.
mov AL, [32h]     ;  Load AL with the content of
                  ;  memory position 30032h
mov [800h], BL    ;  Write the content of BL into the memory
                  ;   position 30800h
```



2.4. Addressing modes (IV)

- Register indirect addressing

  The effective address of the operand is contained in registers **BX**, **BP**, **SI** or **DI**.

  Example:

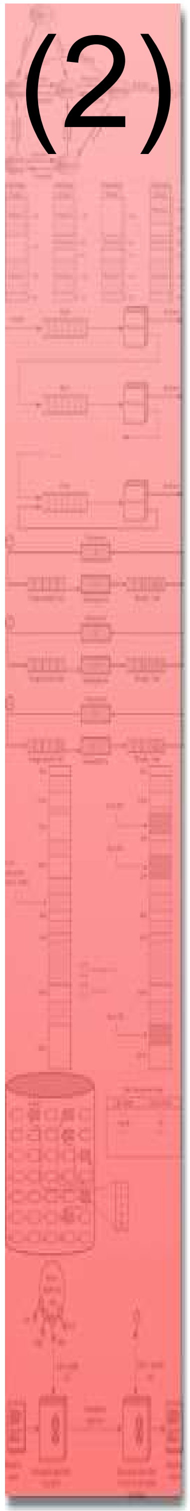
  ```
  mov AX, [BX]
  ```

- Based addressing

  The effective address is obtained by adding a displacement to register **BX** or **BP**.

  Equivalent examples if *offset* of TABLE is 4:

  ```
  mov AX, [BX]+4
  mov AX, 4[BX]
  mov AX, TABLE[BX]
  mov AX, [BX+4]
  ```



(2)

2.4. Addressing modes (V)

- Indexed addressing

  The effective address is calculated by adding a displacement to the content of **SI** or **DI**.

  Equivalent examples if *offset* of TABLE is 4:

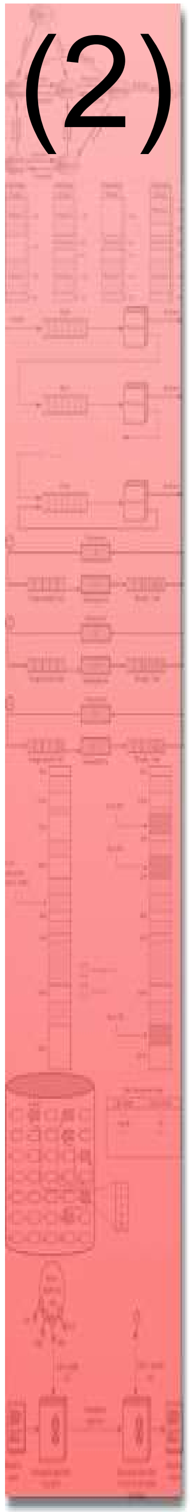
  ```
  mov AX, [SI]+4
  mov AX, 4[SI]
  mov AX, TABLE[SI]
  mov AX, [SI+4]
  ```

- Based-indexed addressing

  The effective address is obtained by adding **BX** or **BP** plus **SI** or **DI** and/or a direct offset.

  Examples:

  ```
  mov AX, TABLE[BX][SI]
  mov AX, TABLE+[BX]+[SI]
  ```



(2)

2.4. Addressing modes (VI)

- Relative addressing

  Used in conditional jumps: The operand is a displacement of 8 bits with sign (-128 to 127) that is added to the instruction pointer **IP**.

  Example:

```
jnc 26
jz label        ;  Whenever the label is at a distance larger
                ;  than or equal to -128 and lower than 128
```

- Implicit addressing

  It is not necessary to specify the operand (it is implicit).

  Examples:

```
cli     ;  Set the interrupt flag to 0
stc     ;  Set the carry flag to 1
```

(2)

2.4. Addressing modes (VII)

- Default segment registers for indirect, relative and indexed addressing:

$$\begin{tikzcd}
\boxed{\text{SI DI BX}} \arrow[r] & \boxed{\text{DS}} \\
\boxed{\text{BP}} \arrow[r] & \boxed{\text{SS}}
\end{tikzcd}$$

- Forced use of another segment register:

  The address is prefixed with the desired register.

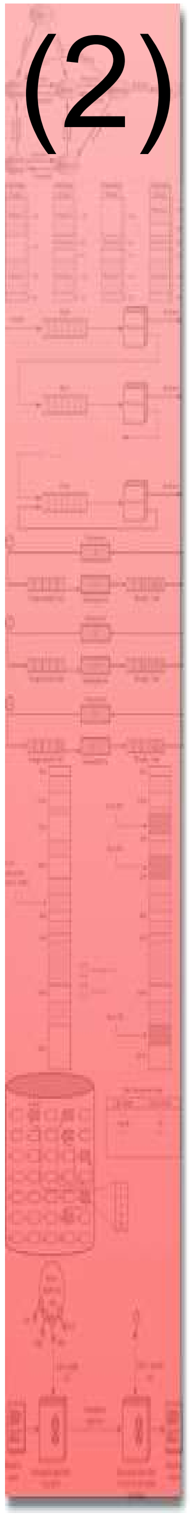
- Examples with **DS = 3000h**, **CS = 2000h**, **ES = A000h**, **SS = E000h**, **SI = 100h** and **BP = 500h**

```
mov DX, [678Ah]              ; [3678Ah] & [3678Bh] ⇒ DX
mov DX, CS:[678Ah]           ; [2678Ah] & [2678Bh] ⇒ DX
mov ES:[SI], AL              ; AL ⇒ [A0100h]
mov SS:[1000h+SI], CX        ; CL ⇒ [E1100h] & CH ⇒ [E1101h]
mov SI, [BP]                 ; [E0500h] & [E0501h] ⇒ SI
mov DS:[BP], DI              ; DI ⇒ [30500h] & [30501h]
```

2.4. Addressing modes (VIII)

- If data is an 8-bit constant, the number of bytes of the data transfer must be defined explicitly:

```
mov BYTE PTR [ 3Ah ], 4Fh        ;  4Fh ⇒ [3003Ah]
mov WORD PTR [ 3Ah ], 4Fh        ;  4Fh ⇒ [3003Ah] ,  0 ⇒ [3003Bh]
mov [ 3Ah ], 4F00h               ; 0 ⇒ [3003Ah] ,  4Fh ⇒ [3003Bh]
mov BYTE PTR [ 3Ah+SI ], 4       ; 4 ⇒ [3013Ah]
```



(2)

2.5. Directives and operators of the 80x86 assembler (I)

- Directives are instructions to the assembler.
- They are not translated to machine code instructions.
- Three main types of directives:
  - Definition of symbols
  - Definition of data
  - Definition of segments and procedures

2.5. Directives and operators of the 80x86 assembler (II)

- Symbol definition directives

  They assign symbolic names to expressions. After the assignment, the name can be utilized all over the program.

  - EQU can be used to assign text or numerical expressions.
  - = only allows numerical assignments and can be redefined.

Examples:

```
K               EQU         1024
TABLE           EQU         TABLE[BX+SI]
K2              EQU         K
COUNTER         EQU         CX
DOUBLEK         EQU         2*K
MIN_DAYS        EQU         60*24

CONSTANT =      20h
CONSTANT =      CONSTANT + 1
```

(2)

2.5. Directives and operators of the 80x86 assembler (III)

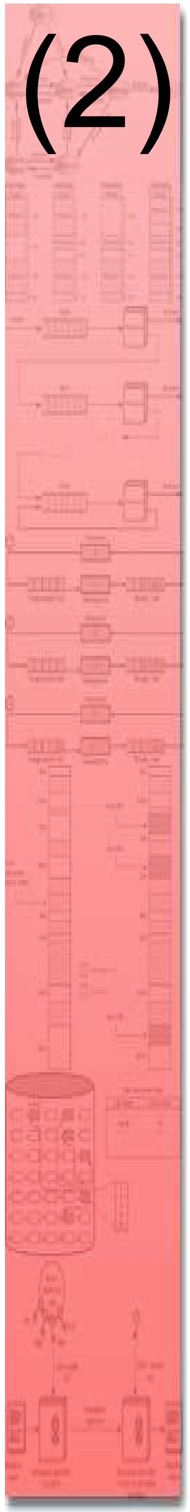
- Data definition directives

  They allocate memory space, assign a value and define a name for the variable.

  - **DB** allocates 1 byte
  - **DW** allocates 2 bytes (1 word)
  - **DD** allocates 4 bytes (2 words)
  - **DQ** allocates 8 bytes (4 words)

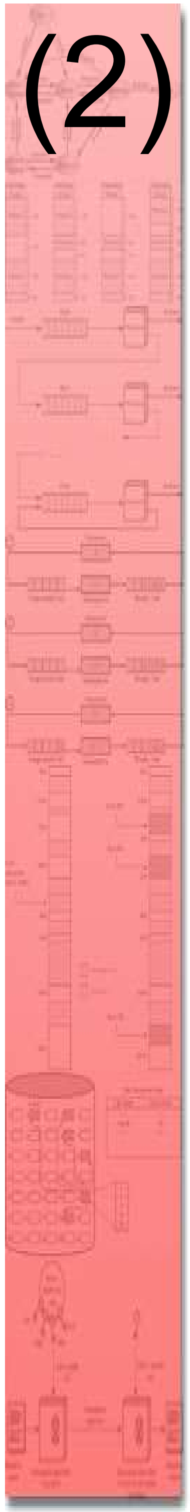
- Examples:

```
NUMBERS    DB   4, 5*9, 10h+4, 23h, 'A'     ; 1 byte per element
TEXT       DB   "Final", 13, 0Ah
NUM        DW   1000, -200, 400/60, 80h     ; 2 bytes per element
NUMMMM     DD   200000h                     ; 4 bytes per element
           DB   6 dup (10h)                 ; 10h six times
           DB   10h dup("Stack")            ; StackStackStack .....
LETTER     DB   ?                 ; allocate 1 byte without assigning a value
LETTERS    DB   8 dup (?)
NEAR       DW   LETTER            ; store offset of LETTER
FAR        DD   LETTER            ; store offset and segment of LETTER
```


(2)

2.5. Directives and operators of the 80x86 assembler (IV)

- Segment and procedure definition directives
  - **SEGMENT** and **ENDS**: They delimit the beginning and end of a logical segment (set of variables or assembly instructions) and give them a name. The stack segment is named **STACK**.
  - **ASSUME reg_seg : name_segment[, ...]**: They indicate the default segment register for addressing the variables specified within the indicated logical segment.
  - **PROC** and **ENDP**: They delimit the beginning and end of a procedure (routine, subroutine, …).
    - A procedure is a part of a program that can be called from different places of a program.
    - The procedure can be **NEAR** or **FAR**.
    - **Near**: It can only be called from the same segment.
    - **Far**: It can be called from any segment.



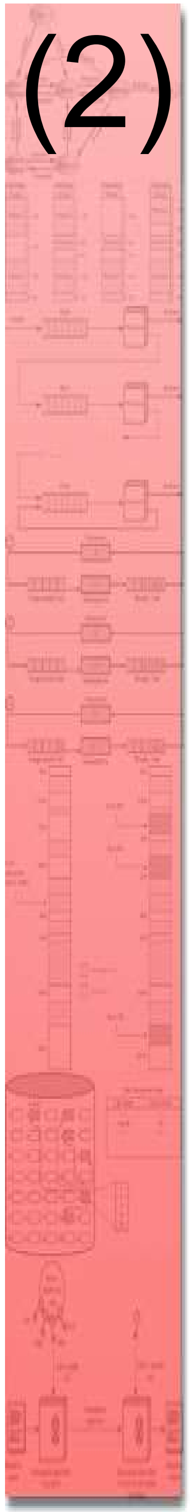
2.5. Directives and operators of the 80x86 assembler (V)

- Segment and procedure definition directives
  - **PUBLIC**: It tells the linker that a label (variable or procedure) declared within the file can be referenced from other files (it is public).
  - **EXTRN**: It tells the linker that a label is declared in another file (it is external).
  - **ORG *offset***: It forces the next variable or machine code instruction to start at the given displacement (*offset*).
  - **END:** It indicates the end of the program. Followed by a label, it tells the assembler the first instruction to be executed.

(2)

2.5. Directives and operators of the 80x86 assembler (VI)

- Operators are modifiers that appear within a directive or assembly instruction.
- They cannot contain neither variables nor registers since their value is calculated in assembly time.
- Four types of operators:
  - Arithmetic operators
  - Logic operators
  - Operators that return values
  - Attribute operators

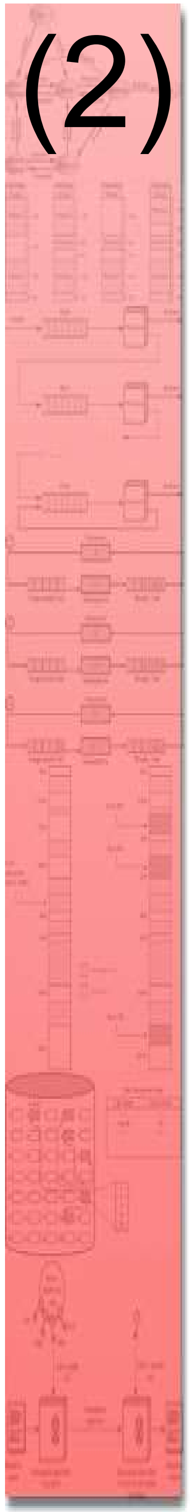


(2)

2.5. Directives and operators of the 80x86 assembler (VII)

- Arithmetic operators: **+ , - , * , / , MOD , SHL , SHR**
  - They combine numerical operands to yield a result.
  - Examples:

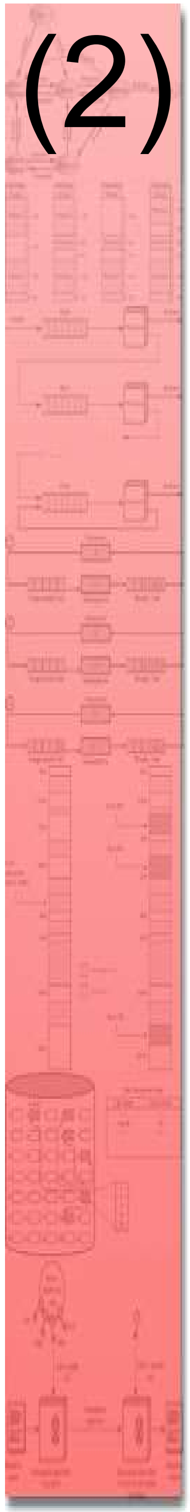
```
PI EQU  31415 / 10000            ;  Quotient of integer division
MOV AX, 2 * PI
MOV CX, 31415 MOD 10000         ;  Remainder of integer
                                 ; division
VAL        EQU  10011101b
VAL2       EQU  VAL SHL 2      ;   VAL2 is 1001110100b
VAL3       EQU  VAL SHR 2      ;   VAL3 is 100111b
```


(2)

2.5. Directives and operators of the 80x86 assembler (VIII)

- Logic operators: **OR , AND , XOR , NOT**
  - They combine numerical operands to yield a result.
  - Examples:

```
MASK        DB    4 AND 80
NUM         EQU  20
NUMNEG      EQU  (NOT NUM) +1
```



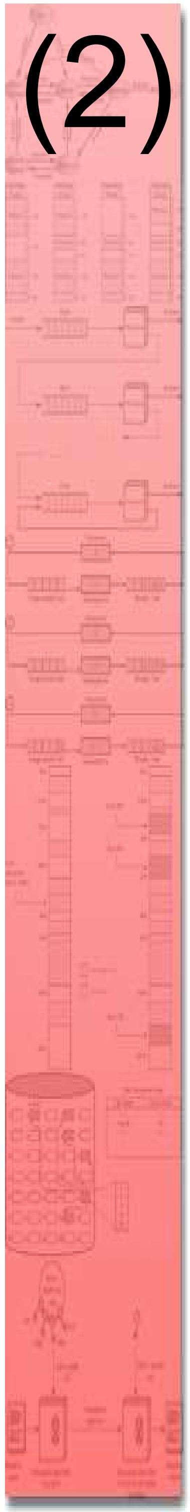
(2)

2.5. Directives and operators of the 80x86 assembler (IX)

- Operators that return values
  - **$**: It returns the displacement (offset) of the instruction or directive where it lies. Useful for calculating the size of strings.
  - **OFFSET** y **SEG**: They return the displacement and segment number of a variable.
  - Examples:

```
TEXT              DB   "Hey, what's up"
SIZE_TEXT         DB   $-TEXT

mov AX, SEG TEXT
mov DX, OFFSET TEXT
```



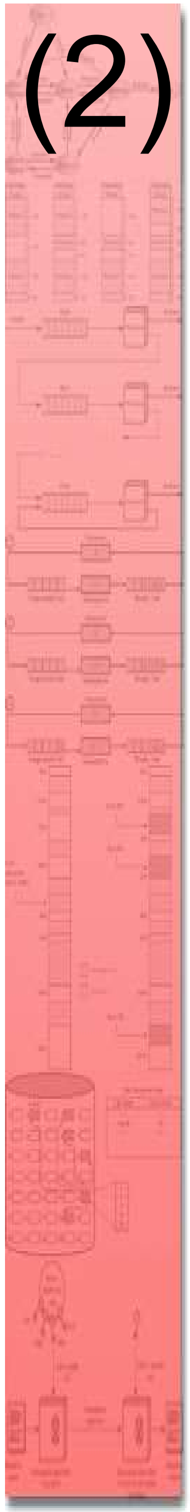
2.5. Directives and operators of the 80x86 assembler (X)

- Attribute operators
  - **PTR**: It modifies the data type (BYTE, WORD, DWORD) of an operand.
  - Examples:

```
TABLE        DB   100  dup (0)
TABLE2       DW   100  dup (0)

mov     AL, TABLE [0]
mov     AX, WORD PTR TABLE [0]
mov     AX, TABLE2 [0]
mov     AL, BYTE PTR TABLE2 [0]

add     WORD PTR TABLE [0], 0FFFFh
add     BYTE PTR TABLE2 [0], 0FFh
inc     WORD PTR [BX]
inc     BYTE PTR [BX]
```



2.6. Structure of an assembly program (I)

```
;  Segment with global variables
data segment

       ;  Variable declaration

data ends
```

```
;  There can be several segments
;   with global variables
```

```
;  Stack segment
stack segment stack "stack"

 ;  Declaration of array of bytes

stack ends
```

```
;  Instructions segment
code segment
           assume cs:code, ds:data
           assume ss:stack

    ;  main procedure
    start proc far

         ;  Assembly instructions

    start endp
....
         ;  other procedures
....

code ends
end start
```

(2)

2.6. Structure of an assembly program (II)

Example 1

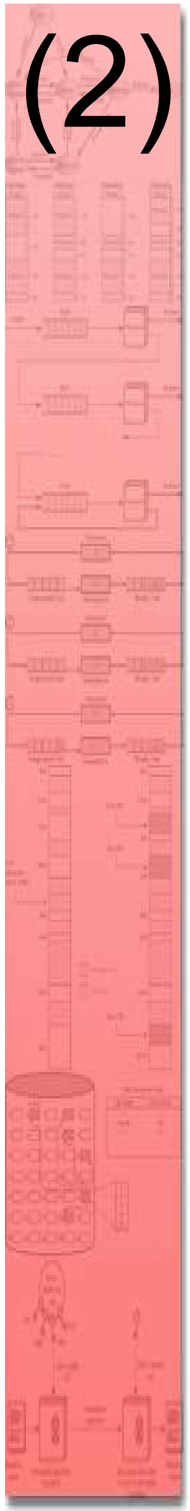
Unnecessary in this example

```
data segment
   size        dw 5
   table       db "abcde"
   table2      db "abcdX"
   result      db 0
data ends
```

```
stack segment stack "stack"
            db 64 dup (?)
stack ends
```

```
code segment
            assume cs:code, ds:data, ss:stack
start proc far
            mov ax, data
            mov ds, ax
            mov ax, stack
            mov ss, ax
            mov sp, 64
            mov si, 0
            mov result, 0
next:       mov al, table[ si ]
            cmp al, table2[ si ]
            jnz different
            inc si
            cmp si, size
            jnz next
            mov result, 1
different: mov ax, 4C00h
            int 21h
start endp

code ends
            end start
```


(2)

2.6. Structure of an assembly program (III)

Example 2

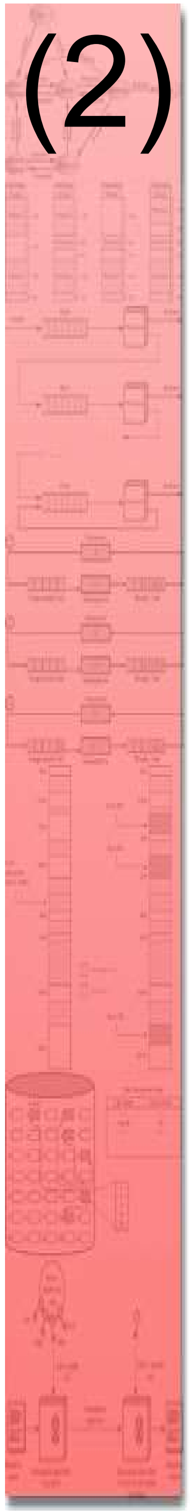
```
data segment
   size      dw 5
   table     db "abcde"
data ends
```

```
data2 segment
   table2    db "abcdX"
data2 ends
```

```
stack segment stack "stack"
            db 64 dup (?)
stack ends
```

```
code segment
            assume cs:code, ds:data, es: data2, ss:stack
start proc far
            jmp lab1
result      db 0
lab1:       mov ax, data
            mov ds, ax
            mov ax, data2
            mov es, ax
            mov ax, stack
            mov ss, ax
            mov sp, 64
            mov result, 0
            mov si, 0
next:       mov al, table[ si ]
            cmp al, table2[ si ]
            jnz different
            inc si
            cmp si, size
            jnz next
            mov result, 1
different: mov ax, 4C00h
            int 21h
start endp
end start
```

Unnecessary in this example


(2)

2.6. Structure of an assembly program (IV)

Example 3

```
data segment
        org 100
table   db "abcdef"
        org 200
table2  db "abcdeX"
data ends
```

```
data2 segment
table3  dw 3 dup (0)
data2 ends
```

```
stack segment stack "stack"
        db 64 dup (?)
stack ends
```

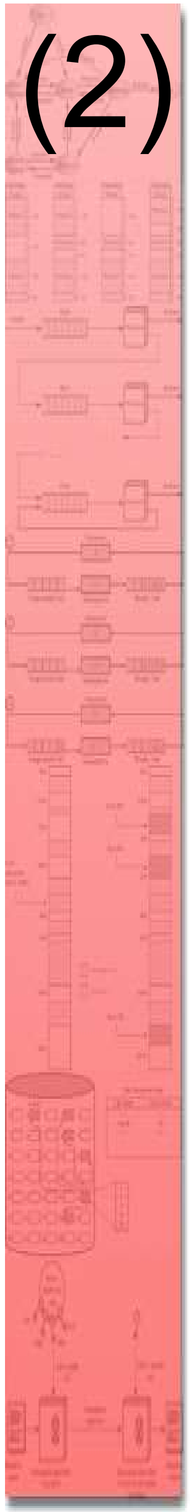
```
code segment
        assume cs:code, ds:data, es: data2, ss:stack
start proc far
        mov ax, data
        mov ds, ax
        mov ax, data2
        mov es, ax
        mov ax, stack
        mov ss, ax
        mov sp, 64
        mov si, offset table
        mov di, offset table2
        mov cx, 6
next:   mov al, [ si ]
        cmp al, [ di ]
        jnz different
        inc si
        inc di
        dec cx
        jnz next
        call store
different: mov ax, 4C00h
        int 21h
start endp
```

```
store proc
        mov si, 0
repeat: mov ax, word ptr table[ si ]
        mov table3[ si ], ax
        inc si
        inc si
        cmp si, 6
        jnz repeat
        ret
store endp


code ends
end start
```

2.7. Assembly instructions (I)

- Types of basic instructions
  - Data transfer
  - Arithmetic operations
  - Logic operations
  - Control transfer
  - Interrupts
  - Activation of flags

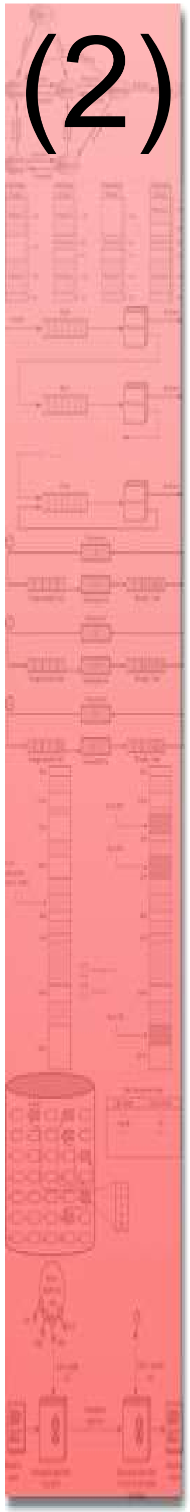


2.7. Assembly instructions (II)

Data transfer

- **MOV**: Transfer data between registers or registers and memory locations.
  MOV destination, source
- **XCHG**: Exchange the contents of two registers or a register and a memory position.
  XCHG destination, source
- **PUSH**: Store onto the stack.
  PUSH source
- **POP**: Extract from the stack.
  POP source

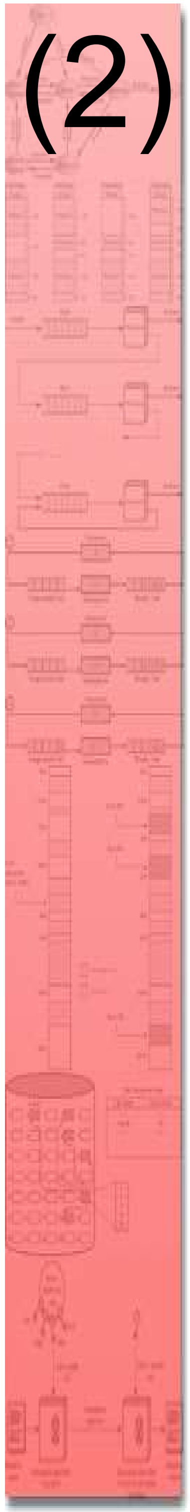
(2)



2.7. Assembly instructions (III)

Input / Output

- **IN**: Read a single data from a port.
  IN accumulator, port
- **OUT**: Send a single data to a port.
  OUT port, accumulator


(2)

2.7. Assembly instructions (IV)

Address transfer

- **LEA**: Load the effective address. It transfers the offset of a memory position to a 16 bit register.

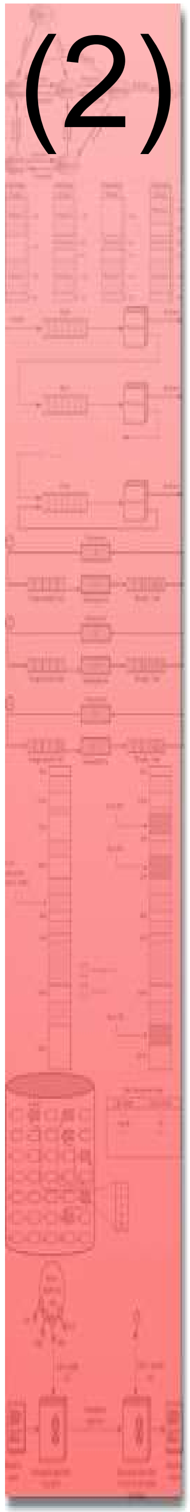
  LEA reg16, mem16

- **LDS**: Load pointer using DS. It transfers the content of the specified memory word into the given register and the content of the next word to DS.

  LDS reg16, mem16

- **LES**: Load pointer using ES. It transfers the content of the specified memory word into the given register and the content of the next word to ES.

  LES reg16, mem16



(2)

2.7. Assembly instructions (V)

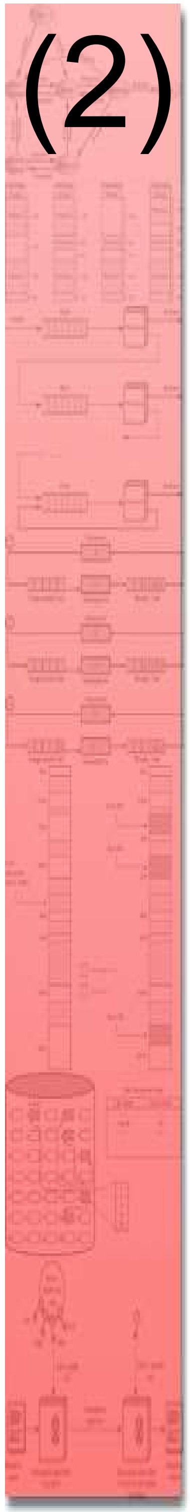
Transfer of flags register

- **PUSHF**: Store the flags register onto the stack.

  PUSHF

- **POPF**: Load the flags register from the stack.

  POPF



(2)

2.7. Assembly instructions (VI)

Arithmetic operations

- They operate integers of 8 or 16 bits.
- Unsigned integers:
  - **0** to **255** (8 bits), **0** to **65535** (16 bits)
- Signed integers:
  - **-128** to **127** (8 bits), **-32768** to **32767** (16 bits)
  - Most significant bit (sign): **0** (positive), **1** (negative)

2.7. Assembly instructions (VII)

Arithmetic operations

- **ADD**: Sum the source and destination operands, leaving the result in the destination.

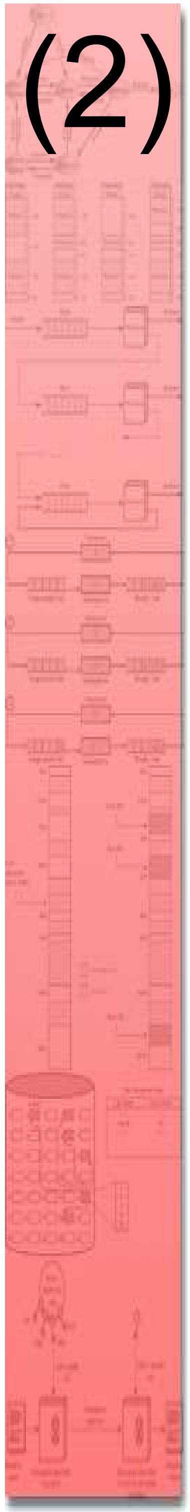
  ```
  ADD destination, source
  ```

- **ADC**: Sum the source and destination operands plus the value of the carry flag.

  ```
  ADC destination, source
  ```

- **INC**: Increment by one the register or memory position.

  ```
  INC operand
  ```



2.7. Assembly instructions (VIII)

Arithmetic operations

- **SUB**: Subtract the source operand from the destination operand, leaving the result in the destination.

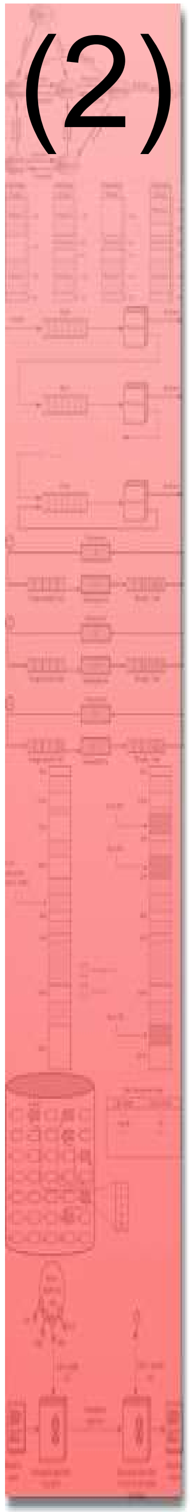
  SUB destination, source

- **SBB**: Subtract both the source operand and the value of the carry flag from the destination operand.

  SBB destination, source

- **DEC**: Decrement by one the register or memory position.

  DEC operand



(2)

2.7. Assembly instructions (IX)

Arithmetic operations

- **MUL**: Multiply operand by **AL** (8-bit operand) or by **AX** (16-bit operand). Result in **AX** (8-bit operand) or **DX:AX** (16-bit operand).

  MUL operand

- **IMUL**: Multiply with sign.

  IMUL operand

- **DIV**: Divide **AX** (8-bit operand) or **DX**:**AX** (16-bit operand) by the unsigned operand.
  Quotient in **AL** and remainder in **AH** (8-bit operand).
  Quotient in **AX** and remainder in **DX** (16-bit operand).

  DIV operand

- **IDIV**: Divide with sign.

  IDIV operand

2.7. Assembly instructions (X)

Arithmetic operations

- **NEG**: Perform the 2’s complement of a register or memory position.

  NEG operand

- **CMP**: Subtract source operand from destination operand without modifying destination (flags updated).

  CMP destination, source

(2)

2.7. Assembly instructions (XI)

Logic operations

- **AND**: Bitwise AND between registers or between register and memory position, leaving the result in the destination.

  AND destination, source

- **OR**: Bitwise OR between registers or between register and memory position, leaving the result in the destination.

  OR destination, source

- **XOR**: Bitwise Exclusive OR between registers or between register and memory position, leaving the result in the destination.

  XOR destination, source

- **NOT**: 1’s complement of a register or memory position.

  NOT operand

2.7. Assembly instructions (XII)

Logic operations

- **TEST**: Bitwise AND between source and destination operands without modifying destination (flags updated).

  ```
  TEST destination, source
  ```

- **SAL / SHL**: Left arithmetic or logic shift.

  ```
  SAL operand, 1
  SAL operand, CL
  ```

- **SAR**: Right arithmetic shift.

  ```
  SAR operand, 1
  SAR operand, CL
  ```

- **SHR**: Right logic shift.

  ```
  SHR operand, 1
  SHR operand, CL
  ```

2.7. Assembly instructions (XIII)

Logic operations

- **ROL**: Left rotation.

  ROL operand, 1
  ROL operand, CL

- **ROR**: Right rotation.

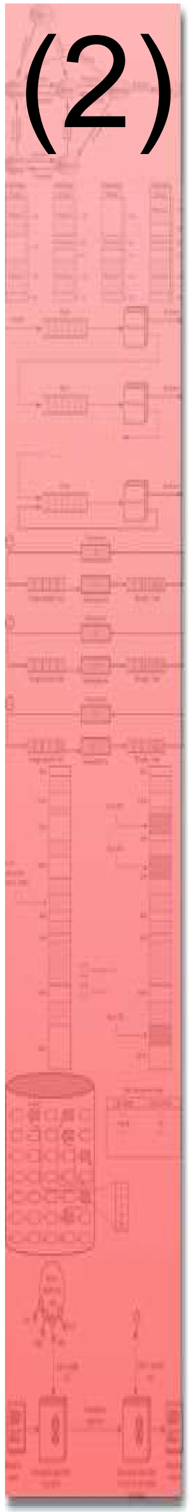
  ROR operand, 1
  ROR operand, CL

- **RCL**: Left rotation with carry flag.

  RCL operand, 1
  RCL operand, CL

- **RCR**: Right rotation with carry flag.

  RCR operand, 1
  RCR operand, CL



(2)

2.7. Assembly instructions (XIV)

Control transfer

- **CALL**: Start execution of a procedure or subroutine. It can be in the same or different segments.

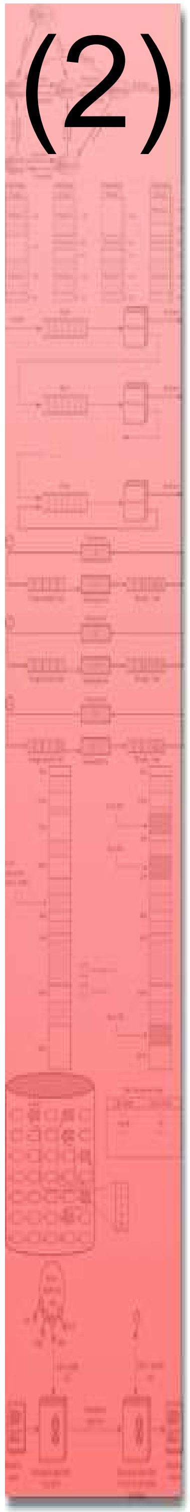
  CALL operand

- **RET**: Return from a procedure or subroutine.

  RET
  RET offset (return and add offset to **SP** for discarding input parameters from the stack)

- **JMP**: Jump to the instruction indicated by the operand.

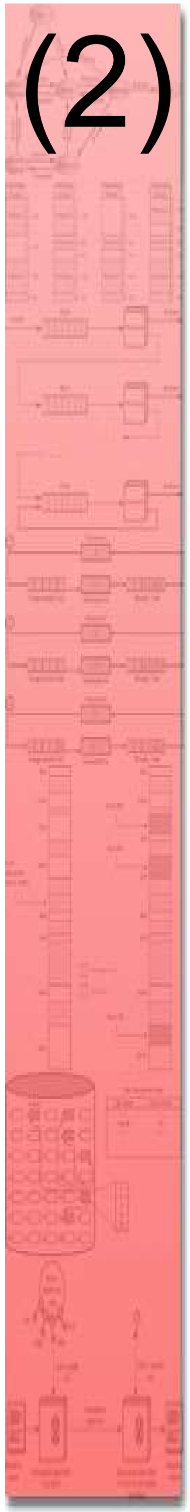
  JMP operand


(2)

2.7. Assembly instructions (XV)

Control transfer

- **Conditional jumps**:
  - They jump to the instruction indicated by the operand provided a flag condition is satisfied. They proceed with the next instruction whenever the condition is not satisfied.
  - They are usually executed after an arithmetic or logic operation (often after a comparison with **CMP** or **TEST**).
  - The jump cannot be larger than **127** bytes forward or **128** bytes backwards.



2.7. Assembly instructions (XVI)

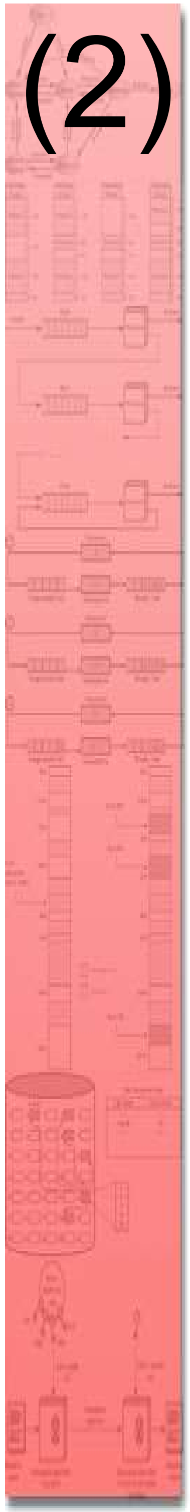
Control transfer (conditional jumps)

Operation between signed integers		
=	JE/JZ	Z=1
<>	JNE/JNZ	Z=0
>	JG	Z=0 & S=0
>=	JGE	S=0
<	JL	S<>0
<=	JLE	Z=1 or S<>0

Operation between unsigned integers		
=	JE/JZ	Z=1
<>	JNE/JNZ	Z=0
>	JA	Z=0 & C=0
>=	JAE	C=0
<	JB	C<>0
<=	JBE	Z=1 or C<>0

JCXZ	CX=0	If CX=0
JO	O=1	If overflow
JNO	O=0	If not overflow
JS	S=1	If sign -
JNS	S=0	If sign +
JC	C=1	If carry
JNC	C=0	If not carry
JP/JPE	P=1	If even parity
JNP/JPO	P=0	If odd parity

G: greater
L: less
E: equal
A: above
B: below



2.7. Assembly instructions (XVII)

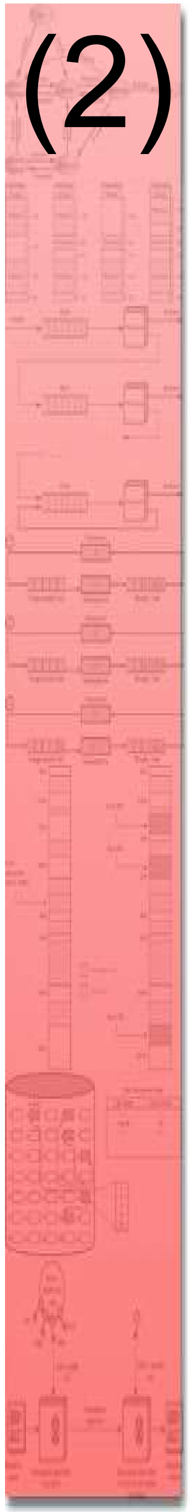
Interrupts

- **INT**: Execute the interrupt service routine indicated by the number.

  ```
  INT number
  ```

- **IRET**: Return from an interrupt service routine.

  ```
  IRET
  ```


(2)

2.7. Assembly instructions (XVIII)

Activation of flags

- STC: Carry **C** := 1.

  STC
- CLC: Carry **C** := 0.

  CLC
- CMC: Complement carry **C**.

  CMC
- STI: interrupts **IF** := 1 (enable interrupts)

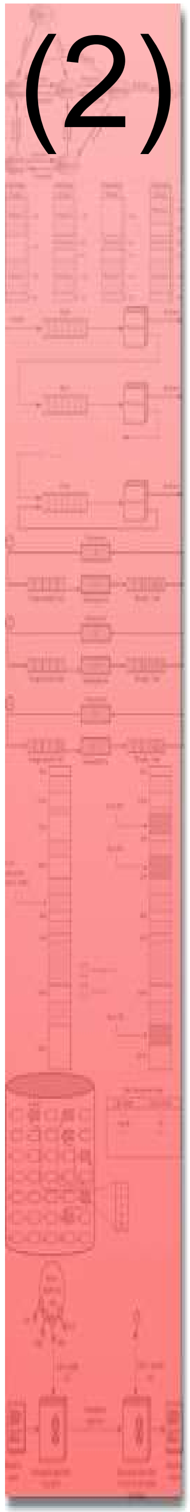
  STI
- CLI: interrupts **IF** := 0 (disable interrupts)

  CLI

(2)

2.8. Memory map of the PC system

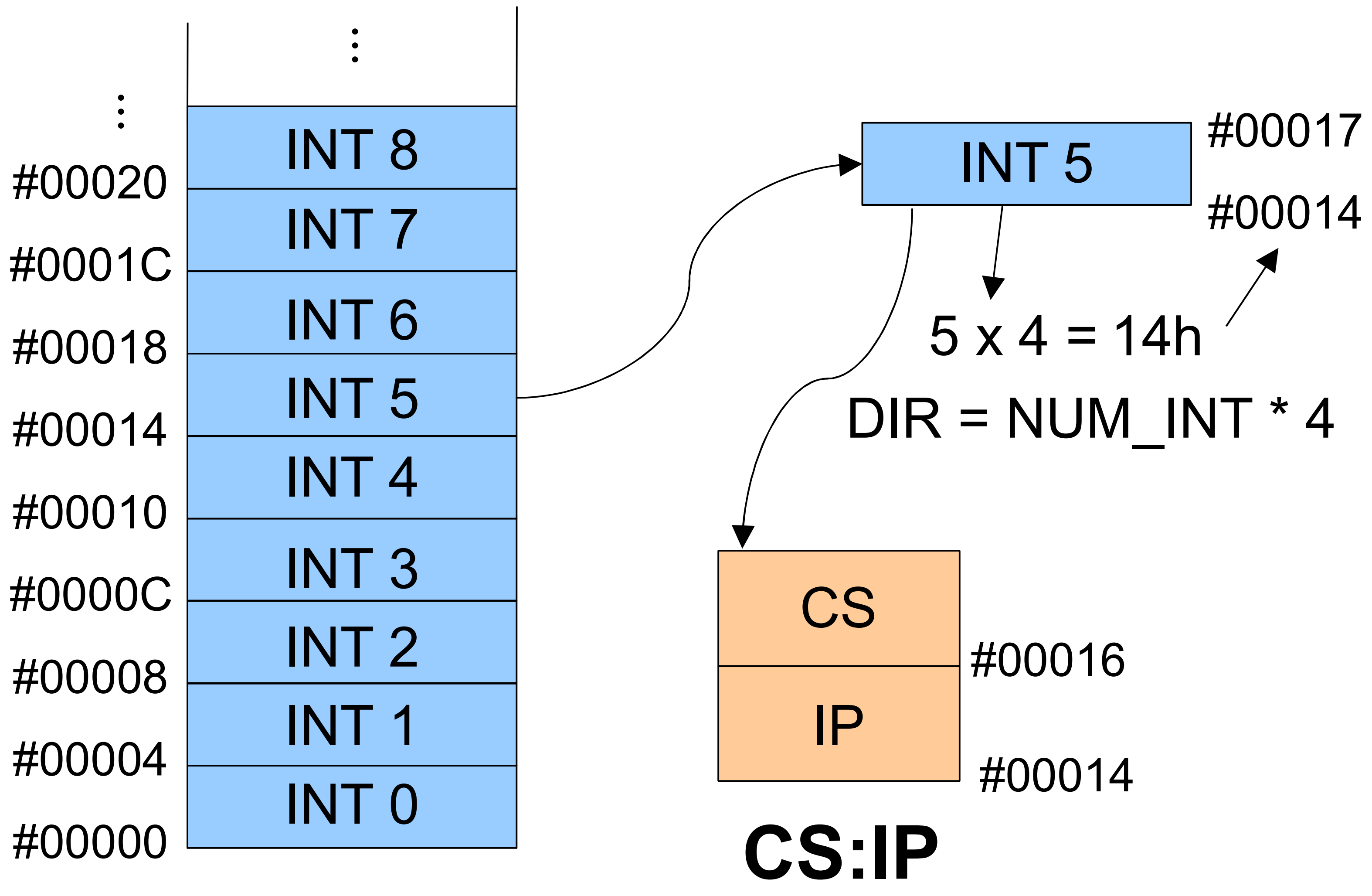
#00000-#0007F	BIOS interrupt vectors
#00080-#000FF	DOS interrupt vectors
#00100-#001FF	User interrupt vectors
#00200-#003FF	BASIC interrupt vectors
#00400-#004FF	BIOS data
#00500-#005FF	DOS and BASIC data
#00600-#9FFFF	User memory for programs
#A0000-#AFFFF	Screen expanded memory area
#B0000-#BFFFF	Screen memory area
#C0000-#EFFFF	BIOS extensions, not fully occupied
#F0000-#FFFFF	ROM BIOS

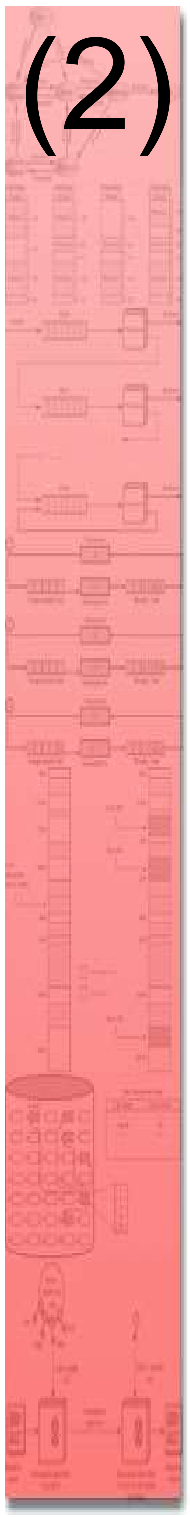


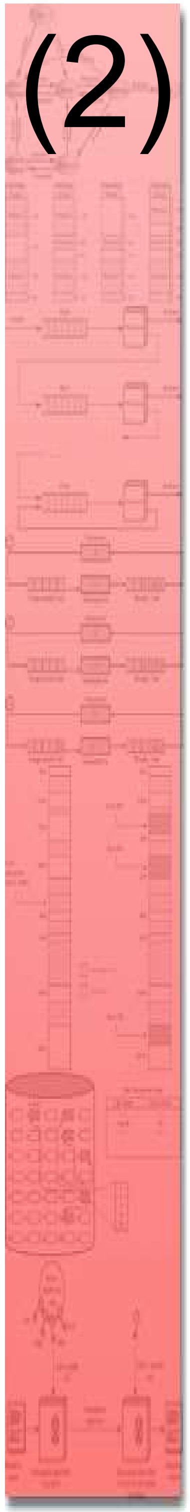
2.9. Interrupts: mechanism and interrupt vectors (I)

- Interrupts are calls to system routines (usually BIOS and OS services).
- These routines are “resident” in memory.
- The memory positions where the routines start are stored in a memory table.
- This table is located at the beginning of memory in DOS: from address 0 to 3FFh.
- Every 4 bytes in this table constitute an interrupt vector (offset and segment where the associated interrupt service routine starts).

2.9. Interrupts: mechanism and interrupt vectors (II)







(2)

2.9. Interrupts: mechanism and interrupt vectors (III)

- Installation of an interrupt service routine:

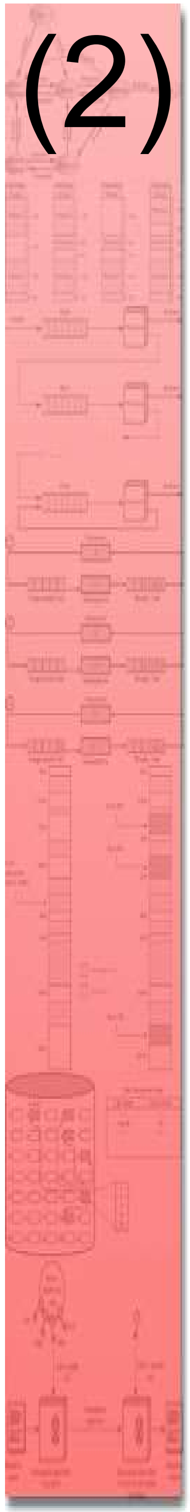
```
DIR     equu    4 * NUM_INT

mov ax, 0
mov es, ax
cli
mov es:[ DIR ], OFFSET service_routine
mov es:[ DIR + 2 ], SEG service_routine
sti
```

(2)

2.9. Interrupts: mechanism and interrupt vectors (IV)

- **Software interrupts:** Activated from a program with the instruction **INT** *n* (*n* between 0 and 255). They cannot be disabled (masked).
- **Hardware interrupts:** Activated through two microprocessor pins: *INTR*, *NMI*.
  - *Maskable*:
    - Activated by hardware by setting pin INTR to 1.
    - Masked with flag **IF** = 0.
    - Devices indicate the interrupt number through the data bus.
  - *Non-maskable*:
    - Activated by hardware by a rising edge in pin NMI.
    - Equivalent to software **INT** 2.



(2)

2.9. Interrupts: mechanism and interrupt vectors (V)

- Execution stages of an interrupt by the CPU:
  1. Flags and return address are stacked:
     - State register (2 bytes)
     - Code segment of return address.
     - Offset of return address.
  2. Interrupt flag **IF** and trace flag **TF** are set to 0 (masking hardware interrupts and deactivating step-by-step execution).
  3. Interrupt vector (**CS**:**IP**) with address of the first instruction of service routine is read.
  4. Service routine is executed.
  5. Service routines ends with instruction **IRET**.
  6. Return address and flags are unstacked:
     - **IP** := Offset return @
     - **CS** := Segment return @
     - State register := flags