ORGANIZACIÓN Y PROGRAMACIÓN DE COMPUTADORAS

OPC

A – D 2023

HLL Structured instructions. via

- Algorithm ---- program
- Structured thinking
- Algorithm structures
    - SELECTIVES: **If-then**, **if-then-else**, **switch-case**
    - REPETITIVES: **while**, **for**, **do-while**, **repeat**
- There are not such instructions in machine language.
- The programmer *must do* the implementation.

Flags Affected by Arithmetic

- The CPU has a number of *Status Flags* that reflect the outcome of arithmetic and bitwise operations (machine instructions)
  - based on the contents of the destination operand
  - Instructions ADD, CLC, CMP, DEC, INC, IMUL, NEG, TEST, and many more . . . except MOV (*)

- Essential flags:
  - ZeroF ZF, SignF SF, CarryF CF, OverflowF OF, AxiliaryF AF, ParityF PF

- (*)The MOV instruction never affects the flags.

Status Flags - Review iVc

- The Zero flag ZF is set when the destination (result) equals zero.
- The Carry flag CF is set when a result that is too large (or too small) for the destination; *unsigned* out-of-range result value.
- The Sign flag SF is set if the destination is negative, otherwise it is clear. A *Signed* result value.
- The Overflow flag OF is set when a destination is a *signed* out-of-range result value.
- The Auxiliary Carry flag AF is set when an operation produces a carry out from bit 3 to bit 4.
- The Parity flag PF is set when an instruction generates an even number of 1 bits in the low byte of the destination operand.

CONDITIONAL JUMPS WILL USE THESE FLAGS

Conditional Jumps

- Conditional Jump instructions *jump* if
  - a flag is *on* (*1*),
  - a flag is *off* (*0*),
  - some flags are *on*; or
  - some flags are *off*.

CPU flags affected

- Assembly Language for x86 Processors Book
  - Appendix B. The x86 Instruction Set

16-bit FLAGS register

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	nt	iop	iop	**OF**	df	if	tf	**SF**	**ZF**	0	**AF**	0	**PF**	1	**CF**

FLAGS categories

- STATUS flags: OF, SF, ZF, AF, PF, CF
- CONTROL flags: tf, if, df
- SYSTEM flags: iop, nt

- Reserved bits: 0s and 1.

32-bit EFLAGS register

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0	0	0	0	0	0	0	0	0	0	id	vip	vif	ac	vm	rf

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	nt	iop	iop	**OF**	df	if	tf	**SF**	**ZF**	0	**AF**	0	**PF**	1	**CF**

FLAGS categories

- STATUS flags: OF, SF, ZF, AF, PF, CF
- CONTROL flags: tf, if, df
- SYSTEM flags: iop, nt, rf, vm, ac, vif, vip, id

- Reserved bits: 0s and 1.

64-bit RFLAGS register

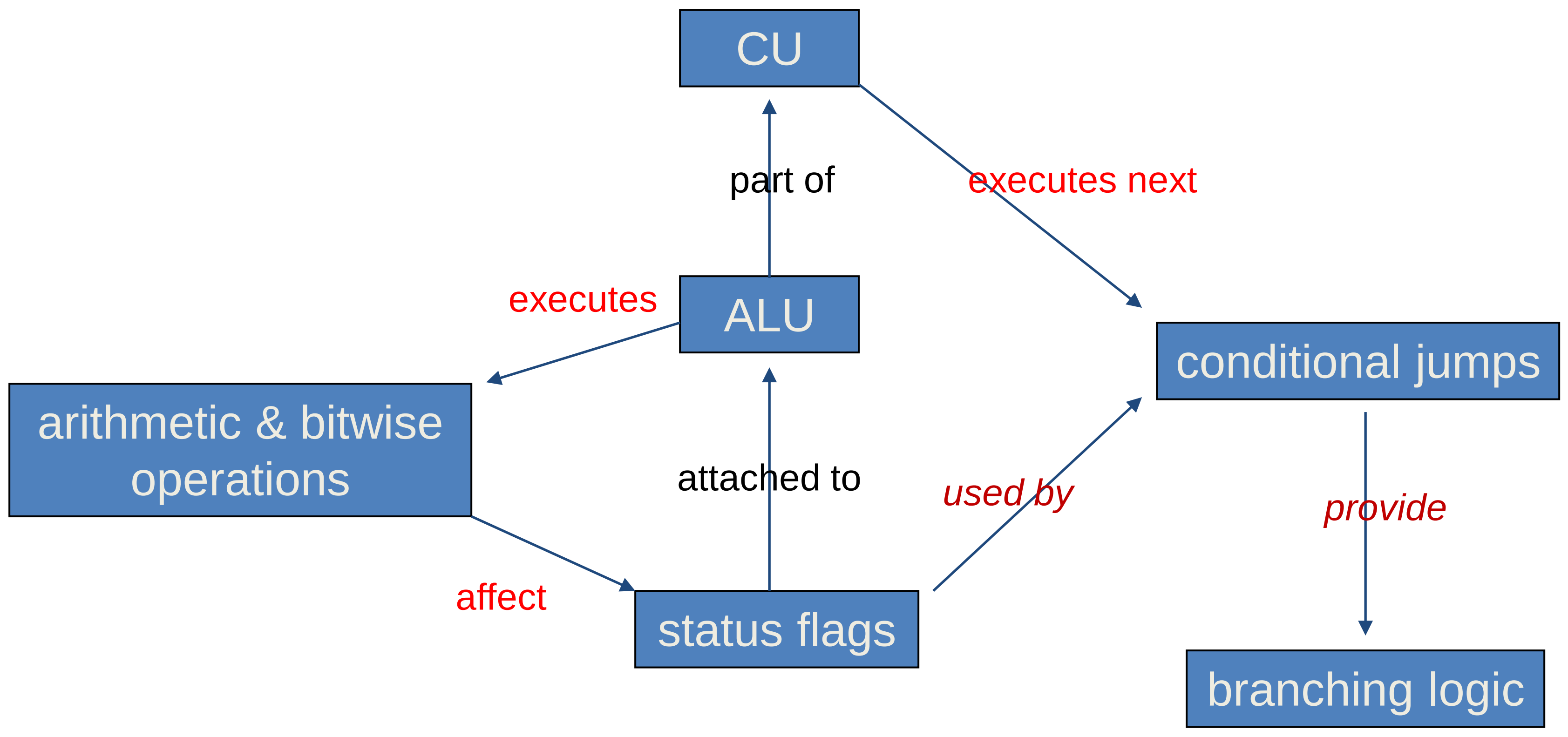
63	62	61	60	59	58	57	56	54	54	53	52	51	50	49	48
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

47	46	45	44	43	42	41	40	39	38	37	36	35	34	33	32
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
0	0	0	0	0	0	0	0	0	0	id	vip	vif	ac	vm	rf

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	nt	iop	iop	**OF**	df	if	tf	**SF**	**ZF**	0	**AF**	0	**PF**	1	**CF**

CPU Conceptual Map



Signed and ***Unsigned*** Integers

A CPU Hardware Viewpoint

- All CPU instructions operate exactly the same on signed and unsigned integers
- The CPU cannot distinguish between signed and unsigned integers
- YOU, the PROGRAMMER, are solely responsible for using the correct *data type* with each instruction

ZERO Flag (ZF)

The ZERO flag is set when the result of an operation produces zero in the destination operand.

```
MOV CX,1            ; none flag is changed
SUB CX,1            ; CX= 0, ZF= 1, CF= __
MOV AX,0FFFFh
INC AX              ; AX= 0, ZF= 1 , CF= __
INC AX              ; AX= 1, ZF= 0 , CF= __
```

Remember...

- A flag is set when it equals 1.
- A flag is clear when it equals 0.

CARRY Flag (CF)

The CARRY flag is set when the result of an operation generates an unsigned value that is out of range (too big or too small for the destination operand).

```
MOV AL,0FFh
ADD AL,1                          ; CF= 1, AL= 00, ZF= __

; Try to go below zero:

MOV AL,0
SUB AL,1                          ; CF= 1, AL= FF, ZF= __
```

Auxiliary Carry Flag. vlaiiVb

- The Auxiliary Carry (AC) Flag indicates a *carry* or a *borrow out* of bit 3 to bit 4 in the destination operand.
- Example:

```
MOV AL,0Fh                      MOV AL,22h
ADD AL, 1   ; ACF=1              SUB AL, 8h     ;ACF=1
```

- The develope

```
   0 0 0 0  1 1 1 1          0 0 1 0  0 0 1 0
 + 0 0 0 0  0 0 0 1        - 0 0 0 0  1 0 0 0
 ------------------        ------------------
   0 0 0 1  0 0 0 0           0 0 0 1  1 0 1 0
```

- Related operations: BCD (Binary-Coded Decimal) arithmetic operations in four bits. The digits 0..9 are represented en four bits.

SIGN Flag (SF)

The SIGN flag is set when the destination signed operand is negative. The flag is clear when the destination is positive.

```
 MOV CX,0
 SUB CX,1                ; CX= -1, SF= 1, ZF=   , CF=
 ADD CX,2                ; CX= 1, SF= 0, ZF=   , CF=
```

The SIGN flag is a copy of the destination's highest bit:

```
MOV AL,0
SUB AL,1                ; AL= 11111111b, SF= 1, ZF=   , CF=
ADD AL,2                ; AL= 00000001b, SF= 0, ZF=   , CF=
```

Your turn . . .

For each of the following marked entries, show the values of the destination operand and the Sign, Zero, and Carry flags:

```
MOV AX,00FFh
ADD AX,1                ; AX=        SF=    ZF=    CF=
SUB AX,1                ; AX=        SF=    ZF=    CF=
ADD AL,1                ; AL=        SF=    ZF=    CF=
MOV BH,6Ch
ADD BH,95h              ; BH=        SF=    ZF=    CF=

MOV AL,2
SUB AL,3                ; AL=        SF=    ZF=    CF=
```

OVERFLOW Flag (OF)

The OVERFLOW flag is set when the signed result of an operation is *invalid* or *out of range*.

```
; Example 1
MOV AL,+127
ADD AL,1                     ; OF=__, AL=__, CF=__, SF=__, ZF=__

; Example 2
MOV AL,7Fh                   ; OF=__, AL= __
ADD AL,1
```

A Rule of Thumb / signed

- When ADDing two integers, remember that the *Overflow flag* is only set when . . .
  - two positive operands are added and their sum is negative
  - two negative operands are added and their sum is positive

```
What will be the values of the Overflow flag?
    mov al,80h
    add al,92h                    ; OF=   , CF=   , SF=   , ZF=

     mov al,-2
     add al,+127                  ; OF= , CF=   , SF=   , ZF=

    mov al,+2
    add al,+127                   ; OF= , CF=   , SF=   , ZF=
```

NEG Instruction.

The processor implements NEG using the following internal operation:

```
        operand = 0 – operand          ; or
        operand = 0 + (-operand)     ; adding viewpoint
```

```
.DATA
valB BYTE 1,0
valC SBYTE -128
.CODE
   NEG valB               ; CF= _, OF= _, SF= _, ZF= _
   NEG [valB + 1]         ; CF= _, OF= _, SF= _, ZF= _
   NEG valC               ; CF= _, OF= _, SF= _, ZF= _
```

Your turn . . .

What will be the values of the given flags after each operation?

```
MOV AL,-128
NEG AL                   ; CF=   , OF=   , ZF=   , SF=

MOV AX,8000h
ADD AX,2                 ; CF=   , OF=   , ZF=   , SF=

MOV AX,0
SUB AX,2                 ; CF=   , OF=   , ZF=   , SF=

MOV AL,-5
SUB AL,+125              ; CF=   , OF=   , ZF=   , SF=
```

Parity Flag. vlaiiVb

- The Parity Flag (PF) is set when the least significant byte of the destination operand has an even number of 1 bits.
- Example:

```
MOV AL, 10001100b
ADD  AL, 00000010b          ; AL = 10001110,    PF = 1
SUB  AL, 10000000b          ; AL =  00001110,    PF = 0
```

Keeping track of Flags

Related instructions

- LAHF – Load status flags into AH
  - AH ← Low byte of EFLAGS (S, Z, 0, A, 0, P, 1, C)

- SAHF – Store AH into status flags
  - Low byte of EFLAGS (S, Z, 0, A, 0, P, 1, C) ← AH

```
.DATA
    one   BYTE ?

.CODE
    LAHF
    MOV one, AH
```

```
; following CODE
    MOV BL, one
    - - -
    MOV AH, one
    SAHF
```

Instructions around the Carry flag

- Set carry flag
  - STC ;CF=1
- Clear carry flag
  - CLC ;CF=0
- Complement carry flag
  - CMC ;CF= 1s complement of CF
- Add carry flag
  - ADC opd1,opd2 ; opd1 $\leftarrow$ opd1 + opd2 + CF

Referencias

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

ORGANIZACIÓN Y PROGRAMACIÓN DE COMPUTADORAS

OPC

A – D 2023

Multiplication and Division

- Multiplication and Division Instructions

- MUL – unsigned multiply
- IMUL – signed multiply

- DIV – unsigned division
- IDIV – signed división

MUL Instruction (Unsigned Multiply) iiVa

- The MUL (unsigned multiply) instruction multiplies an 8-, 16-, 32- or 64- bit operand by either AL, AX, EAX, or RAX.
- The one-operand instruction formats are:

```
MUL reg/mem8          ;16-bit product
MUL reg/mem16         ;32-bit product
MUL reg/mem32         ;64-bit product
```

- Implicit multiplicand: AL, AX or EAX.

Multiplicand	Multiplier	Product
AL	reg/mem8	AH:AL = AL * reg/mem8
AX	reg/mem16	DX:AX = AX * reg/mem16
EAX	reg/mem32	EDX:EAX = EAX * reg/mem32

- Check CARRY flag after MUL for significant bits in the upper half of the product.
- The product is twice the size of the multiplicand or the multiplier.

MUL Examples 1

2000h * 100h, using 16-bit operands:

```
.DATA
val1 WORD 2000h
val2 WORD 100h
.CODE
MOV AX,val1
MUL val2       ; DX:AX = 00200000h, CF=1
```

The CARRY flag indicates whether or not the upper half of the product contains significant digits.

12345h * 1000h, using 32-bit operands:

```
MOV EAX,12345h
MOV EBX,1000h
MUL EBX        ; EDX:EAX = 0000000012345000h, CF=0
```

MUL Examples 2

5h * 10h, using 8-bit operands:

```
 .CODE
MOV AL,05h
MOV BL,10h
MUL BL             ; AH:AL = 0050h, CF=0
```

Your turn . . . 1

What will be the hexadecimal values of DX, AX, and the Carry flag after the following instructions execute?

```
MOV AX,1234h
MOV BX,100h
MUL BX
```

Your turn . . . 2

What will be the hexadecimal values of EDX, EAX, and the Carry flag after the following instructions execute?

```
MOV EAX,00128765h
MOV ECX,10000h
MUL ECX
```

IMUL Instruction (Signed Multiply)

- IMUL (signed integer multiply) multiplies an 8-, 16-, 32- or 64- bit operand by either AL, AX, EAX, or RAX.
- Preserves the sign of the product by sign-extending it into the upper half of the destination register
- The one-operand instruction formats are:

```
IMUL reg/mem8            IMUL reg/mem32
IMUL reg/mem16           IMUL reg/mem64
```

Multiplicand	Multiplier	Product
AL	reg/mem8	AH:AL = AL * reg/mem8
AX	reg/mem16	DX:AX = AX * reg/mem16
EAX	reg/mem32	EDX:EAX = EAX * reg/mem32

- CF and OF are set if the upper half of the product is not a sign extension of the lower half.

IMUL Examples

Example: multiply 48 * 4, using 8-bit operands:

```
MOV   AL,48          ; AL = 48 = 30h
MOV   BL,4
IMUL BL             ; AX(AH:AL) = 00C0h, OF=1
```

OF=1 because AH is not a sign extension of AL.

Example: multiply 48 * 4, using 16-bit operands:

```
MOV AX,48
MOV BX,4
IMUL BX             ; DX:AX = 000000C0h, OF=0
```

OF=0 because DX is a sign extension of AX.

IMUL Instruction (Signed Multiply)

- Two-operand 16-bit formats, 16-bit product:

```
IMUL reg16,reg/mem16
IMUL reg16,imm8
IMUL reg16,imm16
```

- Two-operand 32-bit formats, 32-bit product:

```
IMUL reg32,reg/mem32
IMUL reg32,imm8
IMUL reg32,imm32
```

Two-operand IMUL formats truncate the product. When significant digits are lost OF and CF are set.

- Two-operand 64-bit formats.

Multiplicand	Multiplier	Product
reg16	reg/mem16	reg16 = reg16 * reg/mem16
reg16	imm8/imm16	reg16 = reg16 * imm8/imm16
reg32	reg/mem32	reg32 = reg32 * reg/mem32
reg32	imm8/imm32	reg32 = reg32 * imm8/imm32

IMUL Instruction (Signed Multiply)

- Three-operand 16-bit formats:

```
IMUL reg16,reg/mem16,imm8
IMUL reg16,reg/mem16,imm16
```

- Three-operand 32-bit formats:

```
IMUL reg32,reg/mem32,imm8
IMUL reg32,reg/mem32,imm32
```

- Three-operand 64-bit formats.

Three-operand IMUL formats truncate the product. When significant digits are lost OF and CF are set.

Multiplicand	Multiplier	Product
reg16	reg/mem16	reg16 = reg/mem16 * imm8
reg16	imm8/imm16	reg16 = reg/mem16 * imm16
reg32	reg/mem32	reg32 = reg/mem32 * imm8
reg32	imm8/imm32	reg32 = reg/mem32 * imm32

IMUL Examples

Example: multiply 4,823,424 * –423:

```
MOV EAX,4823424
MOV EBX,-423
IMUL EBX            ; EDX:EAX = FFFFFFFF86635D80h, OF=0
```

OF=0, CF=0 because EDX is a sign extension of EAX.

Your turn . . . 3

What will be the hexadecimal values of DX, AX, and the OVERFLOW flag after the following instructions execute?

```
MOV AX,8760h
MOV BX,100h
IMUL BX
```

DIV Instruction (Unsigned Divide)

- The DIV (unsigned divide) instruction performs 8-bit, 16-bit, 32-bit and 64-bit division on unsigned integers
- A single operand is supplied (register or memory operand), which is assumed to be the divisor
- Instruction formats:

```
DIV reg/mem8        ;16-bit dividend
DIV reg/mem16       ;32-bit dividend
DIV reg/mem32       ;64-bit dividend
DIV reg/mem64
```

Dividend	Divisor	Quotient	Reminder
AH:AL	reg/mem8	AL	AH
DX:AX	reg/mem16	AX	DX
EDX:EAX	reg/mem32	EAX	EDX

DIV Examples

Divide 8003h by 100h, using 16-bit operands, 32-bit dividend

```
MOV DX,0                ; clear dividend, high
MOV AX,8003h            ; dividend, low
MOV CX,100h             ; divisor
DIV CX                  ; AX = 0080h, DX = 3
```

Same division, using 32-bit operands, 64-bit dividend:

```
MOV EDX,0               ; clear dividend, high
MOV EAX,8003h           ; dividend, low
MOV ECX,100h            ; divisor
DIV ECX                 ; EAX = 00000080h, EDX = 3
```

DIV conditions

- All arithmetic status flag values are undefined (?) after executing DIV instructions.

- What happens if the Quotient doesn’t fit the destination operand (register)?
  - A divide overflow condition results, causing a CPU interrupt and the current program halts.
- What happens if the divisor is zero?
  - A division by zero condition results, causing a CPU interrupt and the current program halts.
  - To prevent division by zero, test de divisor before dividing.

Your turn . . . 4

What will be the hexadecimal values of DX and AX after the following instructions execute? Or, if divide overflow occurs, you can indicate that as your answer:

```
MOV DX,0087h
MOV AX,6000h
MOV BX,100h
DIV BX
```

Your turn . . . 5

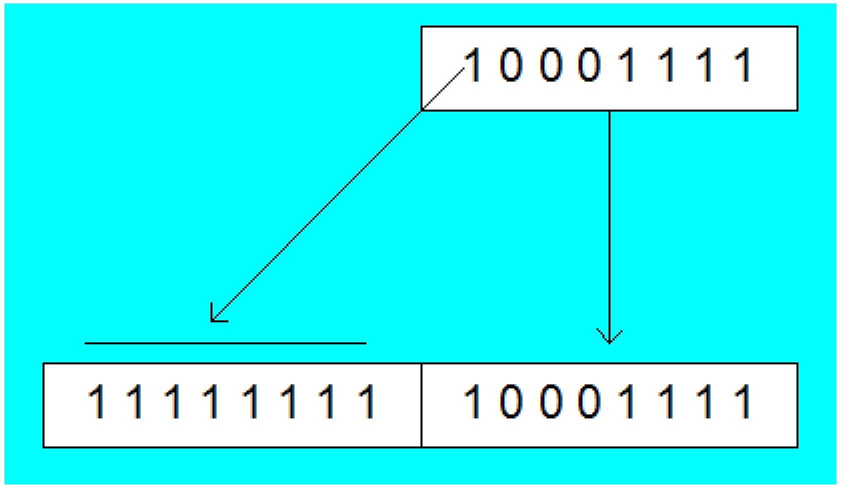
What will be the hexadecimal values of DX and AX after the following instructions execute? Or, if divide overflow occurs, you can indicate that as your answer:

```
MOV DX,0087h
MOV AX,6002h
MOV BX,10h
DIV BX
```

Signed Integer Division (IDIV)

- Signed integers must be sign-extended before division takes place
  - fill high byte/word/doubleword with a copy of the low byte/word/doubleword's sign bit

- For example, the high byte contains a copy of the sign bit from the low byte:



CBW, CWD, CDQ Instructions

- The CBW, CWD, and CDQ instructions provide important sign-extension operations:
  - CBW (convert byte to word) extends AL into AH
  - CWD (convert word to dword) extends AX into DX
  - CDQ (convert dword to qword) extends EAX into EDX
  - CQO (convert qword to oword) extends RAX into RDX
- Example:

```
.DATA
dwordVal SDWORD -101  ; FFFFFF9Bh
.CODE
MOV EAX,dwordVal
CDQ      ; EDX:EAX = FFFFFFFFFFFFFF9Bh
```

IDIV Instruction

- IDIV (signed divide) performs signed integer division
- Same syntax and operands as DIV instruction

Example: 8-bit division of –48 by 5

```
MOV  AL,-48         ; AL = D0h (-30h)
CBW                 ; extend AL into AH=FFh
MOV  BL,5
IDIV BL             ; AL = -9,  AH = -3
```

IDIV Examples

Example: 16-bit division of –48 by 5

```
MOV  AX,-48        ; AX = FFD0h
CWD                ; extend AX into DX
MOV  BX,5
IDIV BX            ; AX = -9,  DX = -3
```

Example: 32-bit division of –48 by 5

```
MOV  EAX,-48       ; EAX = FFFFFFD0h
CDQ                ; extend EAX into EDX
MOV  EBX,5
IDIV EBX           ; EAX = -9,  EDX = -3
```

Your turn . . . 6

What will be the hexadecimal values of DX and AX after the following instructions execute? Or, if divide overflow occurs, you can indicate that as your answer:

```
MOV  AX,0FDFFh            ;AX= -513 = -0201h
CWD
MOV  BX,100h              ;BX= 256
IDIV BX
```

IDIV conditions

- All arithmetic status flag values are undefined (?) after executing IDIV instructions.

- What happens if the Quotient doesn’t fit the destination operand (register)?
  - A divide overflow condition results, causing a CPU interrupt and the current program halts.
- What happens if the divisor is zero?
  - A division by zero condition results, causing a CPU interrupt and the current program halts.
  - To prevent division by zero, test de divisor before dividing.

Referencias

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

ORGANIZACIÓN Y PROGRAMACIÓN DE COMPUTADORAS

OPC

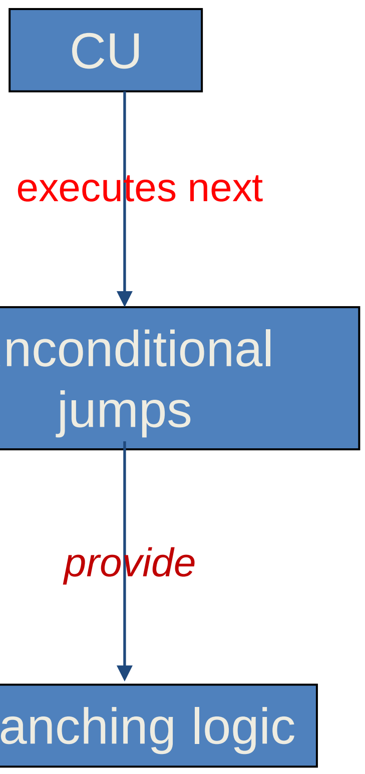
A – D 2023

Transfer of Control Instructions

Transfer of Control Unit (*jumps*) Instructions

- *Unconditional* or *Imperative*
    - jump instruction: **JMP**
    - **goto**, in the beginning of the HLL
- *Conditional*
    - *conditional* jump instructions: **J*cond*** family

CPU Conceptual Map / Jump



Jump instruction

JMP instruction

- Causes an *unconditional transfer of control* (jump) to a destination (*label:*), inside *.CODE*.
- This destination (*label:*) is translated like an offset.
- **Syntax**: JMP *label*
- **Logic**: EIP ← OFFSET *label*

Examples:

```
.CODE
lbl2:
     instrucA
     instrucB
     JMP lbl2
```

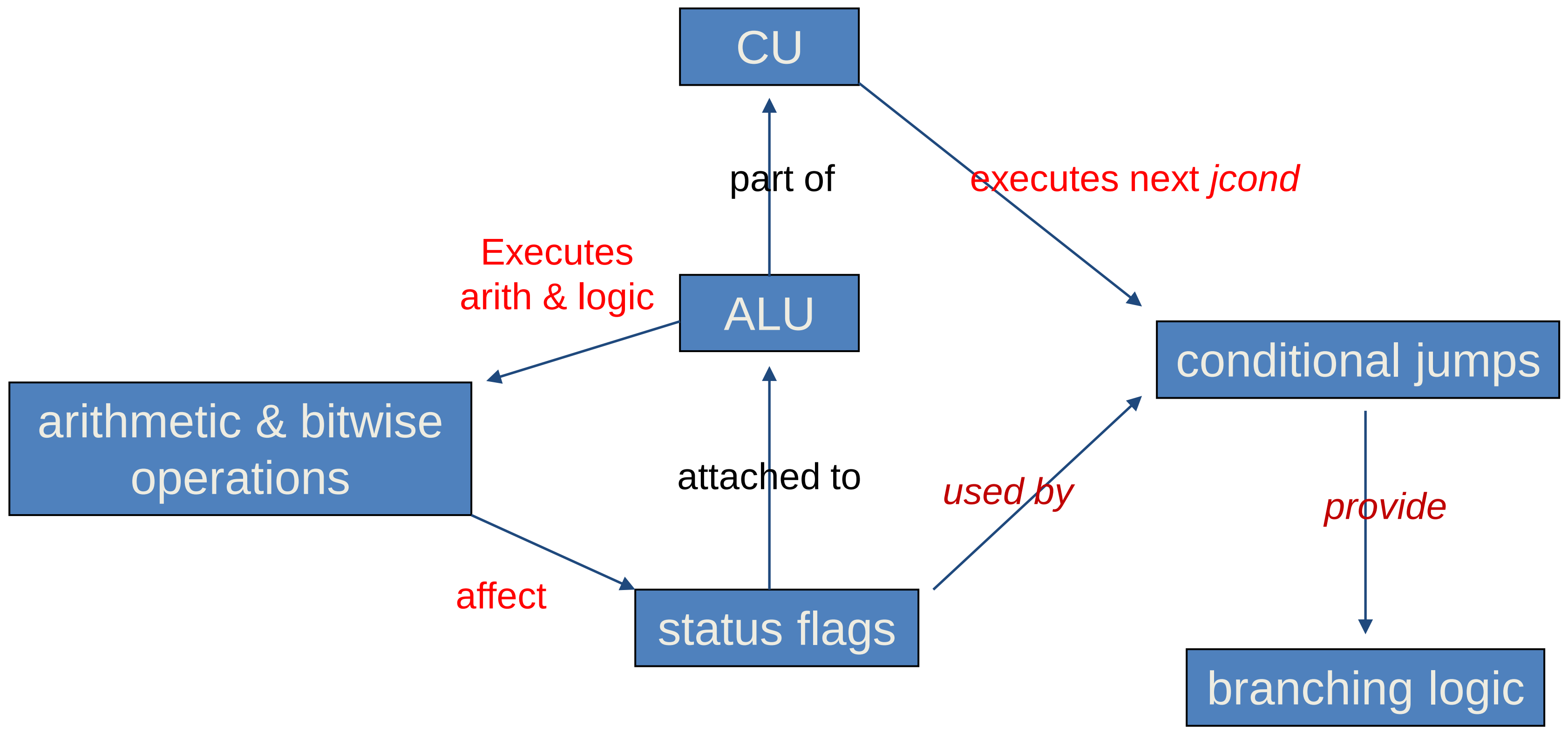
```
.CODE
     JMP lbl3
     instrucR
     instrucT
lbl3:
```

Conditional-Jump instructions

J*cond* instructions

- The *conditional transfer* (J*cond*) to a destination (*label:*), inside *.CODE*, jumps, only if the *condition of a flag*, or *several flags* are met.
- Remember, only ALU instructrions (arithmetic, or logic) can modify one, several or all *flags*.

CPU Conceptual Map / Jcond



HLL Structured instructions. via

- Algorithm ---- program
- Structured thinking
- Algorithm structures or Structured instructions
    - *Selectives*: If-then, if-then-else, switch-case
    - *Repetitives*: while, for, do-while, repeat
- There are not such instructions in machine language.
- The programmer must implement those algorithm structures.

HLL structures need jumps? via

- IF-THEN
- IF-THEN-ELSE

- WHILE / FOR
- DO-WHILE

Conditional Jump instructions

- A conditional jump instruction (**J*cond***) branches to a *label:*, when a specific flag or several flag conditions are met.
- The **J*cond*** analyze the *CPU status flags* affected by the *last executed ALU instruction*.
- Some conditional jumps:

  JC - jump to a label if the Carry flag is set

  JZ - jump to a label if the Zero flag is set

  JS - jump to a label if the Sign flag is set

Jconds usage

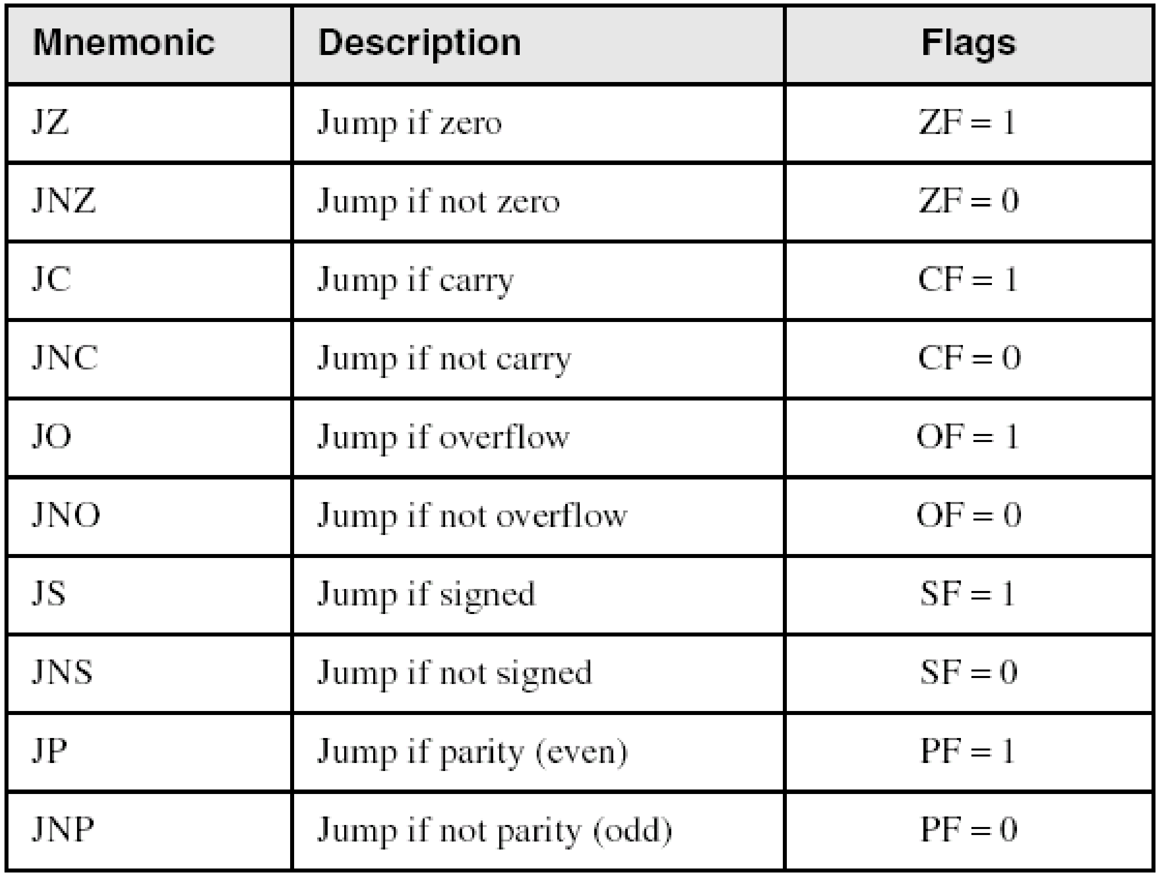
- Example 1

```
     instrucChgFlags
     Jcond la1
     . . .
     . . .
la1: . . .
```

- Example2

```
la2:
     . . .
     . . .
     instrucChgFlags
     Jcond la2
     . . .
```

J*cond*s Based on *one* Flag (after instruction)



Mnemonic	Description	Flags
JZ	Jump if zero	ZF = 1
JNZ	Jump if not zero	ZF = 0
JC	Jump if carry	CF = 1
JNC	Jump if not carry	CF = 0
JO	Jump if overflow	OF = 1
JNO	Jump if not overflow	OF = 0
JS	Jump if signed	SF = 1
JNS	Jump if not signed	SF = 0
JP	Jump if parity (even)	PF = 1
JNP	Jump if not parity (odd)	PF = 0

Jconds examples

- Example 1

```
     MOV EAX,1
     SUB EAX,1
     JZ la1
     . . .
la1:. . .
```

- Example2

```
la2:
     . . .
     MOV EAX,1
     ADD EAX,1
     JNC la2
     . . .
```

Referencias

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

ORGANIZACIÓN Y PROGRAMACIÓN DE COMPUTADORAS

OPC

A – D 2023

J*cond*s

- The predecesor arithmetical or logical instructions could affect one flag or several flags at the same time.

- Which one flag or flags must test?

Flags to Test for a Jcond?

- How many flags have to be tested for a Jcond, when implementing a *Boolean Expression* of a HLL's Algorithm Structure (*if*, *while*, …)?

- For *unsigned* expressions?
  - Chiefly, CF and ZF
- For *signed* expressions?
  - Mainly, SF and OF

CMP in Structured Instructions

- **CMP (compare) instruction** is used to create *conditional logic comparisons*
- When follow *CMP* with a *conditional jump Jcond* instruction, the result is the assembly language equivalent of an IF or a WHILE statement, or any other structured High Level Language instruction.

CMP Instruction

- Syntax: **CMP *leftOp, rightOp***
- Compares the *leftOp* operand to the *rightOp* operand
  - Nondestructive implied subtraction of *rightOp* from *leftOp* (neither operand is changed)
- Logic: ***leftOp - rightOp***
- The comparison affect all the flags.

- The following operand combinations are permitted
  - CMP reg, reg
  - CMP reg, mem
  - CMP reg, imm
  - CMP mem, reg
  - CMP mem, imm

CMP + *Jconds* usage

- Example 1

```
     cmp EAX,uno
     Jcond la1
     . . .
     . . .
la1:. . .
```

looks like an IF

- Example2

```
la2:
     . . .
     . . .
     cmp dos,EBX
     Jcond la2
     . . .
```

looks like repetitive

CMP of two Unsigned operands

- Example 1: it works when leftOp == rightOp

```
MOV AL,5
CMP AL,5                        ; ZF=1, CF=0
```

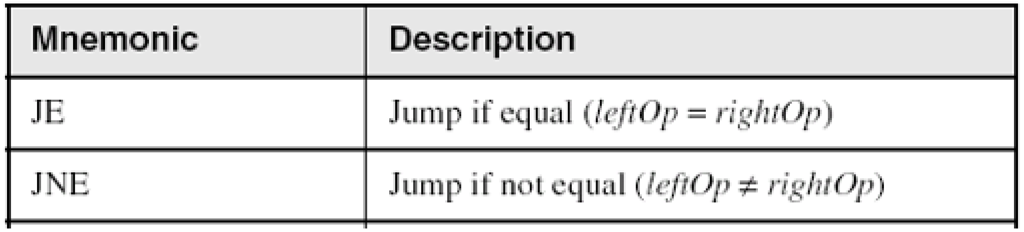
- Example 2: it works when leftOp < rightOp

```
MOV AL,4
CMP AL,5                        ; ZF=0, CF=1
```

- Example 3: it works when leftOp > rightOp

```
MOV AL,6
CMP AL,5                        ; ZF = 0, CF = 0
```

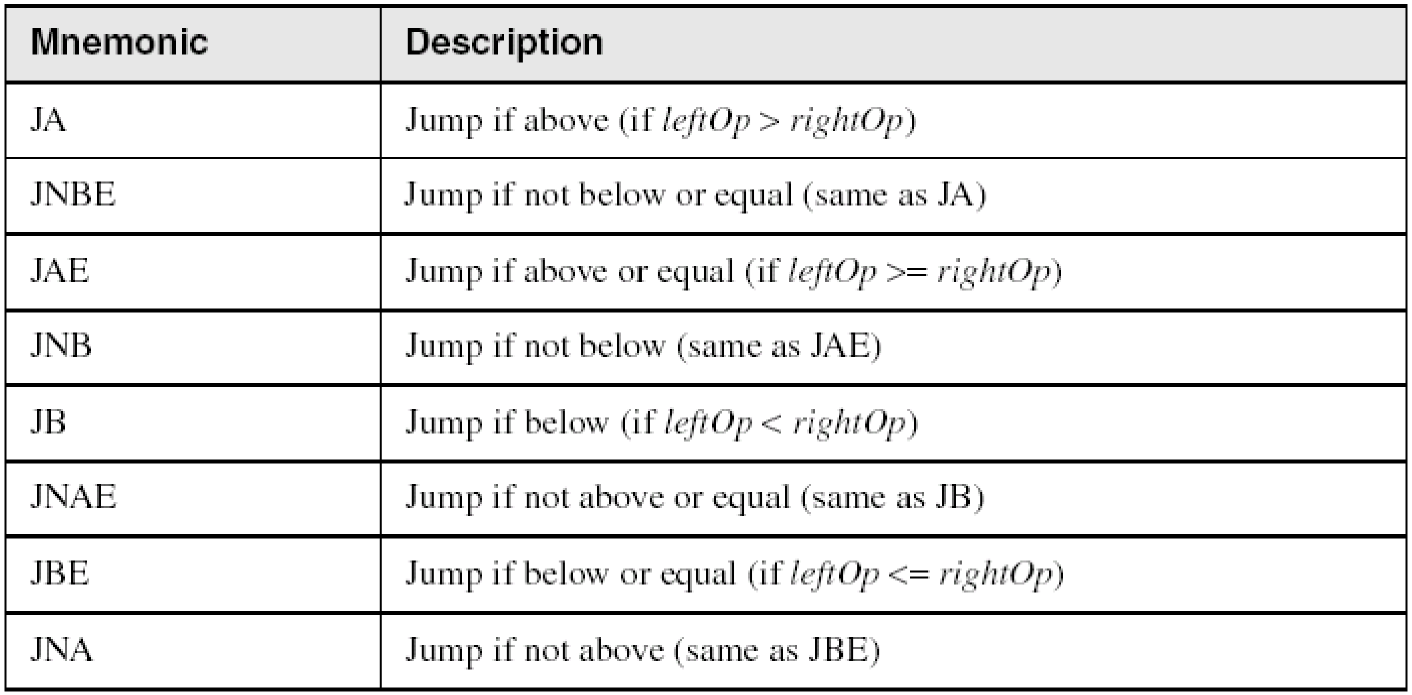
Jconds Based on Equality



Mnemonic	Description
JE	Jump if equal (*leftOp* = *rightOp*)
JNE	Jump if not equal (*leftOp* ≠ *rightOp*)

JE and JNE, for UNSIGNED or SIGNED use.

Jconds Based on Unsigned operands



Mnemonic	Description
JA	Jump if above (if *leftOp* > *rightOp*)
JNBE	Jump if not below or equal (same as JA)
JAE	Jump if above or equal (if *leftOp* >= *rightOp*)
JNB	Jump if not below (same as JAE)
JB	Jump if below (if *leftOp* < *rightOp*)
JNAE	Jump if not above or equal (same as JB)
JBE	Jump if below or equal (if *leftOp* <= *rightOp*)
JNA	Jump if not above (same as JBE)

Better use JA, JAE, JB and JBE

A-Above, B- Below

Examples, Unsigned- 1

- Task: Jump to a label *Larger* if unsigned EAX is greater than EBX
- Solution: Use CMP, followed by JA

```
CMP EAX,EBX
JA  Larger      ; above
```

- Jump to label *L1* if unsigned EAX is less than or equal to Val1

```
CMP EAX,Val1
JBE L1              ; below or equal
```

CMP of two Signed operands

- Example 4: it works when leftOp > rightOp

```
mov al,5
cmp al,-2                   ; Sign flag == Overflow flag
```

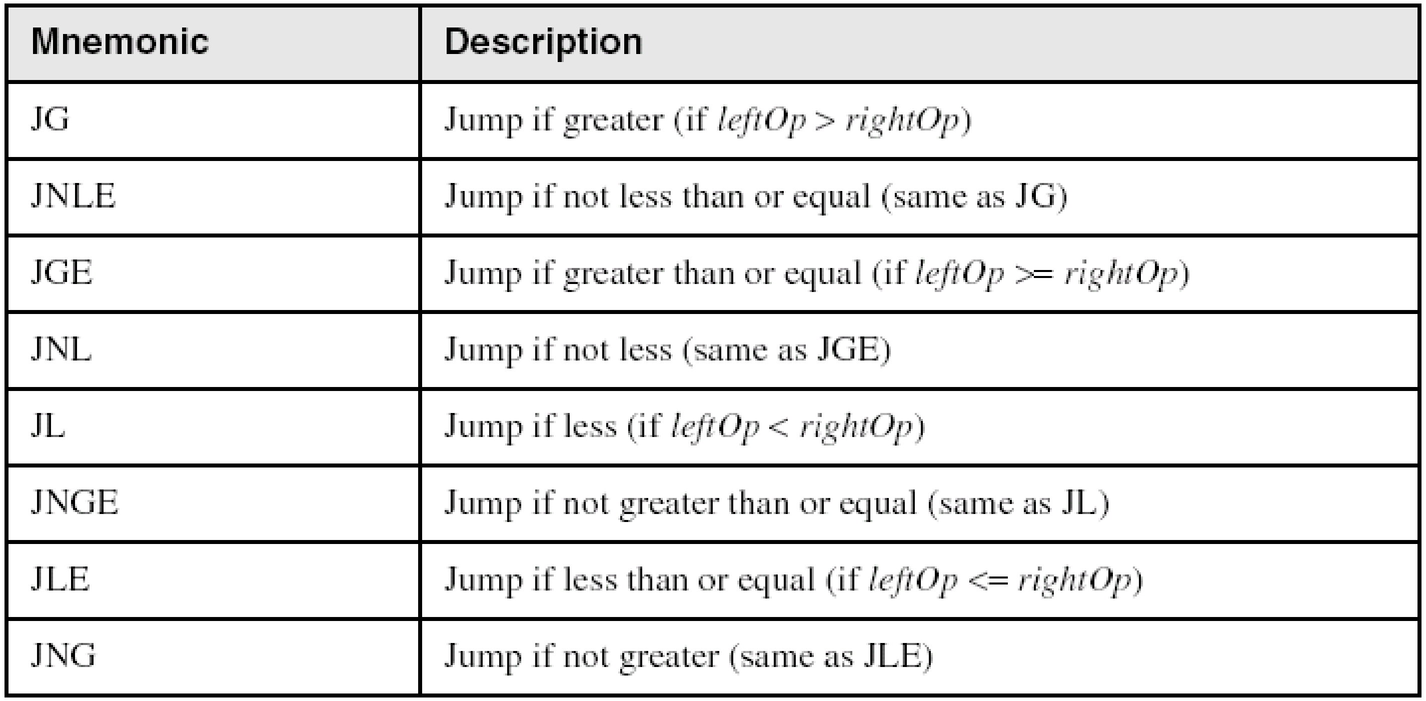
- Example 5: it works when leftOp < rightOp

```
mov al,-1
cmp al,5                    ; Sign flag != Overflow flag
```

- Example 6: it works when leftOp == rightOp

```
mov al,-1
cmp al,-1                   ; ZF=1, CF=don´t care
```

Jconds Based on Signed operands



Mnemonic	Description
JG	Jump if greater (if *leftOp* > *rightOp*)
JNLE	Jump if not less than or equal (same as JG)
JGE	Jump if greater than or equal (if *leftOp* >= *rightOp*)
JNL	Jump if not less (same as JGE)
JL	Jump if less (if *leftOp* < *rightOp*)
JNGE	Jump if not greater than or equal (same as JL)
JLE	Jump if less than or equal (if *leftOp* <= *rightOp*)
JNG	Jump if not greater (same as JLE)

Better use JG, JGE, JL and JLE

G-Greater, L-Lower

Examples, Signed- 2

- Task: Jump to a label *Greater* if signed EAX is greater than EBX
- Solution: Use CMP, followed by JG

```
CMP  EAX,EBX
JG   Greater
```

- Jump to label *L1* if signed EAX is less than or equal to Val1

```
CMP EAX,Val1
JLE L1
```

Referencias

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

ORGANIZACIÓN Y PROGRAMACIÓN DE COMPUTADORAS

OPC

A – D 2023

IF-then, IF-then-else, While, DO-while, *user implementation*, in Assembly

IF-then implementation

- The IF Assembly implementation involves a *conditional jump* for the *negative of the comparison* (*JnoCMP*)

High Level L

```
IF(LfOp cmpOpr RhOp)
{
  block. .;
}
```

Assembly

```
        CMP LfOp, RhOp
        JnoCMP outIF
              block. .
outIF:                ;next instruc
```

IF-then examples

- Compare unsigned AX to BX, and copy the larger of the two into a variable named Large

High Level L

```
Large = BX;
IF(AX > BX)
{
  Large = AX;
 }
```

Assembly

```
MOV Large,BX
CMP AX,BX
JBE outIF  ;Jump if AX<=BX
    MOV Large,AX  ; AX>BX
outIF:
```

- Compare signed AX to BX, and copy the smaller of the two into a variable named Small

High Level L

```
Small = AX;
IF(BX < AX)
{
    Small = BX;
 }
```

Assembly

```
MOV Small,AX
CMP BX,AX
JGE outIF ;Jump if BX>=AX
    MOV Small,BX  ;BX<AX
outIF:
```

IF-then-else implementation

- The IF Assembly implementation involves a conditional jump for the *negative of the comparison* (*JnoCMP*). A *JMP* is needed at the end of block1 to avoid enter into the ELSE area.

High Level L	**Assembly**
`IF(LfOp cmpOp RhOp)`	`CMP LfOp,RhOp`
`{`	`JnoCMP inELSE`
`    block1. .;`	`    block1. .`
`}`	`JMP outIF`
`else`	`inELSE:`
`{`	
`    black2. .;`	`    block2. .`
`}`	`outIF:        ;next instruc`

WHILE implementation

- The WHILE Assembly implementation involves a conditional jump for the *negative of the comparison* (J*noCMP*). After the block it requieres *to jump back* (JMP) to do compare.

High Level L	**Assembly**

`WHILE(LfOp cmpOp RhOp)`
`{`
`  block. .;`
`}`
	`inWHILE: CMP LfOp,RhOp`
	`         JnoCMP outWHILE`
	`              block. .`
	`         JMP inWhile`
	`outWHILE:        ;nextinstruc`

WHILE example

- The WHILE Assembly implementation involves a conditional jump for the *negative of the comparison* (*JnoCMP*). After the block it requires to jump back to do compare. Example signed.

High Level L

```
EAX=45;
EBX=1;
WHILE(EBX < 6)
{
  EAX=EAX-2;
  EBX=EBX+1;
}
```

Assembly

```
           MOV EAX,45
           MOV EBX,1
inWHILE:   CMP EBX,6
           JGE outWHILE
               SUB EAX,2
               INC EBX
           JMP inWhile
outWHILE:              ;nextinstruc
```

DO-WHILE implementation

- The DO-WHILE Assembly implementation involves a conditional jump for the *same comparison* (*JsaCMP*). After the block it requires to jump back to do compare.

```
High Level L
DO
     block. .;
WHILE(LfOp cmpOp RhOp)
```

```
      Assembly
inDO:
          block. .
      CMP LfOp,RhOp
      JsaCMP inDO
          ;nextinstruc
```

DO-WHILE example

- The DO-WHILE Assembly implementation involves a conditional jump for the positive of the comparison (J*saCMP*). After the block it requires to jump back to do compare. Example signed.

High Level L

```
EAX=50;
EBX=0;
DO
    EAX=EAX+1;
    EBX=EBX+1;
WHILE(EBX < 4)
```

Assembly

```
          MOV EAX,50
          MOV EBX,0
inDO:
              INC EAX
              INC EBX
          CMP EBX,4
          JL inDO
                ;nextinstruc
```

Referencias

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

ORGANIZACIÓN Y PROGRAMACIÓN DE COMPUTADORAS

OPC

A – D 2023

HLL Boolean Operators

- Logical Operators, for Boolean operations:
  - Boolean values: true (1d= 0…01b) / false (0d= 0…0b)
  - NOT: !a (!true>false, !false>true)
  - AND: a && b
  - OR: a || b
- Bitwise Operators, for bit-to-bit operations:
  - Integer-bit values: n bits (8, 16, 32, 64, … bits)
  - NOT: ~c (complement 1s)
  - AND: d & e
  - OR: d | e
  - XOR: d ^ e

Bitwise Boolean Instructions

- **Issue**: there are not logical Boolean ASSEMBLY Instructions,

- *Only Bitwise* (bit to bit computations): AND, OR, XOR, NOT, TEST

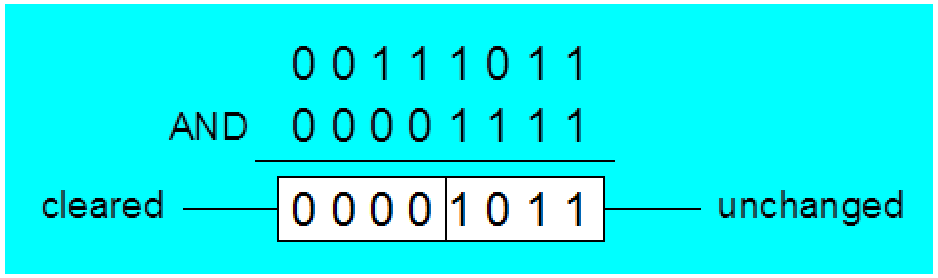
AND Instruction

- AND, bitwise, bit to bit
- Syntax: AND *destination*, *source*
- The following operand combinations are permitted
  - AND reg, reg
  - AND reg, mem
  - AND reg, imm
  - AND mem, reg
  - AND mem, imm
- Operands can be 8, 16, 32, or 64 bits, must be same size
- Bit masking with the *source* operand.
- Set INTERSECTION of bits

AND Instruction

- Performs a Boolean AND operation between each pair of matching bits in two operands
- Flags
  - Clears Overflow, Carry
  - Modifies Sign, Zero, and Parity
- Syntax:
    - AND *destination, source*

  (same operand types as MOV)



AND

x	y	x ∧ y
0	0	0
0	1	0
1	0	0
1	1	1

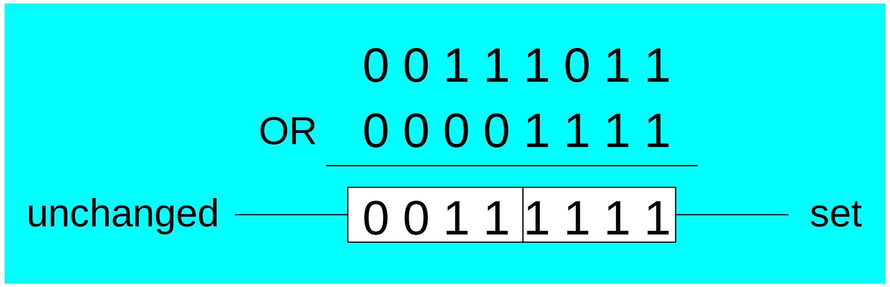
Bit masking, source: 0 in source clears a bit, 1 leaves it unchanged.

OR Instruction

- OR, bitwise, bit to bit
- Syntax: OR *destination*, *source*
- The following operand combinations are permitted
  - OR reg, reg
  - OR reg, mem
  - OR reg, imm
  - OR mem, reg
  - OR mem, imm
- Operands can be 8, 16, 32, or 64 bits, must be same size
- Useful when you want to set one or more bits without affecting the other bits
- Set UNION of bits

OR Instruction

- Performs a Boolean OR operation between each pair of matching bits in two operands
- Flags
  - Clears Overflow, Carry
  - Modifies Sign, Zero, and Parity
- Syntax:
  - OR *destination, source*



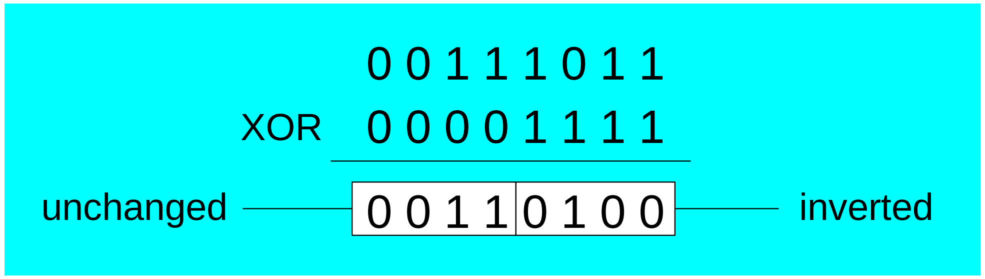
OR

x	y	x ∨ y
0	0	0
0	1	1
1	0	1
1	1	1

1 in source set a bit, 0 leaves it unchanged.

XOR Instruction

- Performs a Boolean eXclusive-OR bitwise operation between each pair of matching bits in two operands
- XOR with 0 retains its value, with 1 reverses value
- Flags
  - Clears Overflow, Carry
  - Modifies Sign, Zero, and Parity
- Syntax:
    - XOR *destination, source*



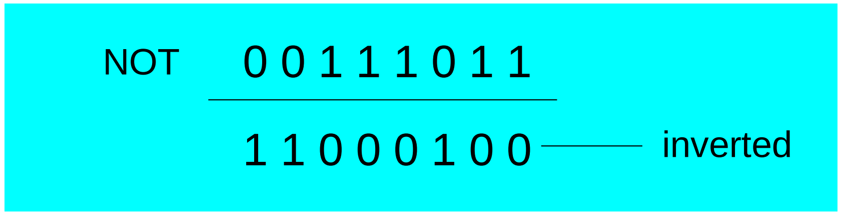
XOR

x	y	$x \oplus y$
0	0	0
0	1	1
1	0	1
1	1	0

XOR is a useful way to toggle (invert) the bits in an operand.

NOT Instruction

- Performs a Boolean NOT bitwise operation on a single destination operand
- The following operand combinations are permitted
  - NOT reg
  - NOT mem
- Flags
  - No flags are affected
- Syntax:
  - NOT *destination*



NOT

X	¬X
F	T
T	F

Results called *one's complement*

Set COMPLEMENT of bits

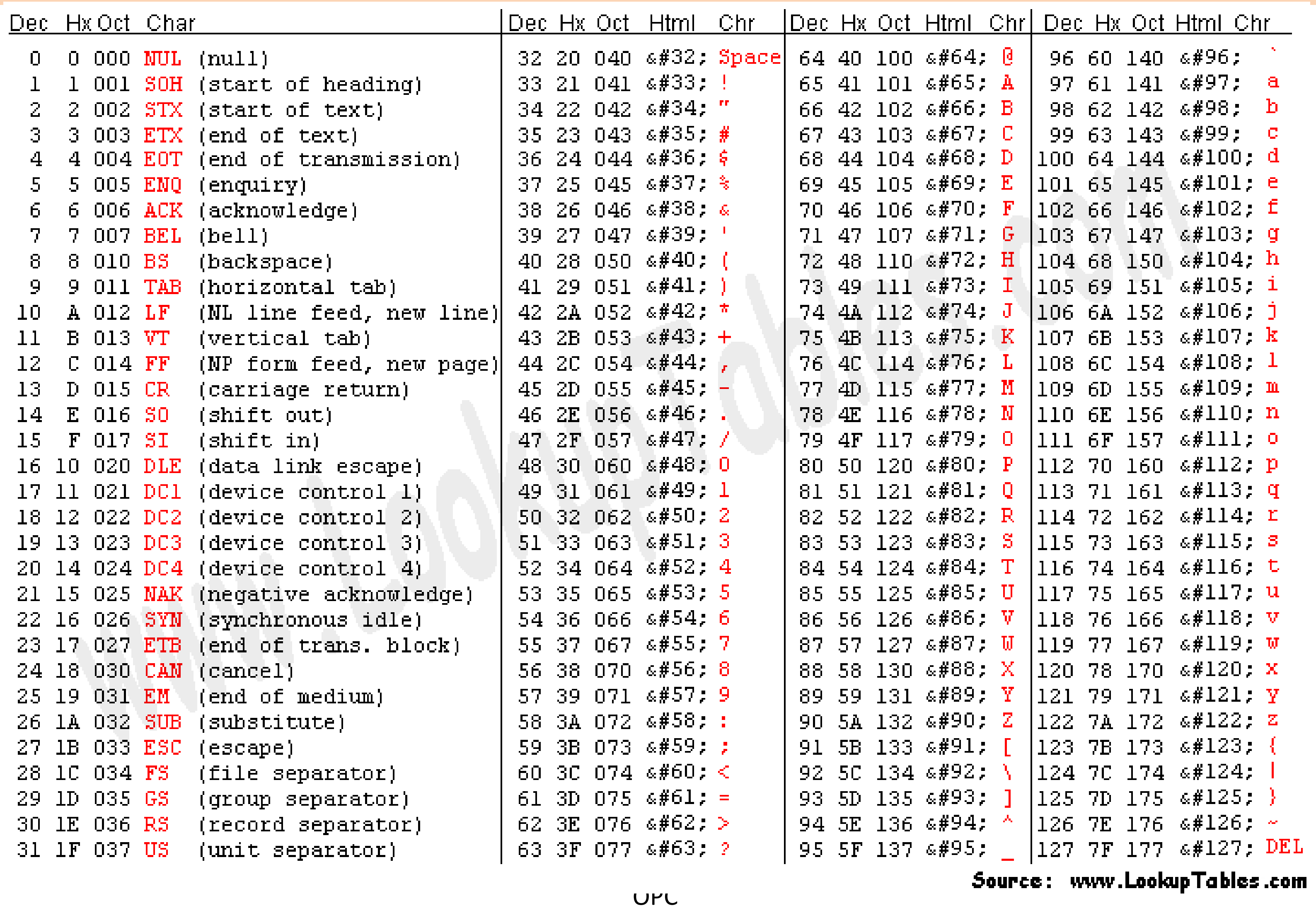
Bit-Mapped Set Operations

- Set Complement
  - MOV AL, SetB ; AL= SetB= 0000 0111b
  - NOT AL ; AL= 1111 1000b

- Set Intersection
  - MOV AL, SetB ; AL= SetB= 0000 0111b
  - AND AL, SetS ; SetS= 0110 0011b AL= 0000 0011b

- Set Union
  - MOV AL, SetB ; AL= SetB= 0000 0111b
  - OR EAX, SetS ; SetS= 0110 0011b AL= 0110 0111b

ASCII Code



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

Source: www.LookupTables.com

Applications (1 of 4)

- Task: Convert the character in AL to UPPER CASE.
- Solution: Use the AND instruction to clear bit 5.

```
mov al, 'a'                     ; AL = 01100001b,   ‘a’
and al, 11011111b               ; AL = 01000001b,   ‘A’
```

Applications (2 of 4)

- Task: Convert a binary decimal byte into its equivalent ASCII Decimal Digit.
- Solution: Use the OR instruction to set bits 4 and 5.

```
MOV AL,6                      ; AL = 00000110b,   6
OR  AL,00110000b              ; AL = 00110110b,  ‘6’
```

The ASCII digit '6' = 00110110b

Applications (3 of 4)

- Task: Jump to a *label* if an integer is *even*.
- Solution: AND the lowest bit with a 1. If the result is Zero, the number was *even*.

```
MOV AX, wordVal
AND AX, 1                 ; low bit set?
JZ  EvenValue             ; jump if Zero flag set
```

Your turn: Write code that jumps to a *label* if an integer is negative.

Applications (4 of 4)

- Task: Jump to a *label* if the value in AL is *not zero*.
- Solution: OR the byte with itself, then use the JNZ (*jump if not zero*) instruction.

```
OR  AL,AL
JNZ IsNotZero             ; jump if not zero
```

ORing any number with itself does not change its value.

TEST Instruction

- Performs a nondestructive AND bitwise operation between each pair of matching bits in two operands
- No operands are modified.
- Always clears the Overflow and Carry flags.
- Modifies the Sign, Zero, and Parity flags.
- Example: jump to a label if either bit 0 or bit 1 in AL is set.

```
TEST AL,00000011b          TEST AL,03h
JNZ  ValueFound            JNZ  ValueFound
```

- Example: jump to a label if neither bit 0 nor bit 1 in AL is set.

```
TEST AL,00000011b
JZ   ValueNotFound
```

Example

- The value 0000 0011 in this example is called a bit mask.

```
TEST AL,00000011b
0 0 1 0 0 1 0 1  <- AL input value
0 0 0 0 0 0 1 1  <- test value
0 0 0 0 0 0 0 1  <- result:  ZF = 0
```

- The value 0000 1001 in this example is called a bit mask.

```
TEST AL,00001001b
0 0 1 0 0 1 0 0   <- AL input value
0 0 0 0 1 0 0 1  <- test value
0 0 0 0 0 0 0 0   <- result:  ZF = 1
```

Referencias

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

ORGANIZACIÓN Y PROGRAMACIÓN DE COMPUTADORAS

OPC

A – D 2023

HLL’s COMPOUND EXPRESSIONS in CONDITIONAL STRUCTURES

Bitwise Boolean Instructions

- To build up compound conditional expressions for IF-THEN, IF-THEN-ELSE, WHILE, DO-WHILE, REPEAT-UNTIL and FOR, logical Boolean instructions are required.
  - E.g.
    - (A > B) && (C <= 5)
    - && is the Logical Boolean operator *and*, in a HLL

Compound Expression with AND - 1

- When implementing the logical AND operator, consider that HLLs use *short-circuit evaluation*
- In the following example, *if the First expression is false*, the *Second expression is skipped*:

```
if (al > bl) AND (bl > cl)
  X = 1;
```

Compound Expression with AND - 2

```
    if (al > bl) AND (bl > cl)
       X = 1;
```

This is one possible implementation . . . no sign

```
    cmp al,bl                   ; First expression...
    JA  L1                       true
    JMP next                    ;false :: a1<=b1
L1:
    cmp bl,cl                   ; Second expression...
    JA  L2                       ; true
    JMP next                      false
L2:                             ; both are true
    mov X,1                     ; set X to 1
next:
```

Compound Expression with AND - 3

```
if (al > bl) AND (bl > cl)
  X = 1;
```

But the following implementation uses 29% less code by reversing the first relational operator. We allow the program to "fall through" to the second expression:

```
    cmp al,bl                  ; First expression...
    JBE next                   ; quit if false
    cmp bl,cl                  ; Second expression...
    JBE next                   ; quit if false
    mov X,1                    ; both are true
next:
```

Compound Expression with AND - 4

Implement the following pseudocode in assembly language. All values are unsigned:

```
if( ebx <= ecx
    && ecx > edx )
{
  eax = 5;
  edx = 6;
}
```

```
    cmp ebx,ecx
    JA  next
    cmp ecx,edx
    JBE next
      mov eax,5
      mov edx,6
next:
```

(There are multiple correct solutions to this problem.)

Compound Expression with OR - 1

- When implementing the logical OR operator, consider that HLLs use short-circuit evaluation
- In the following example, if the First expression is true, *the Second expression is skipped*:

```
if (al > bl) OR (bl > cl)
   X = 1;
```

$\longrightarrow$

Compound Expression with OR - 2

```
if (al > bl) OR (bl > cl)
  X = 1;
```

We can use "fall-through" logic to keep the code as short as possible:

```
     cmp al,bl                       ; is AL > BL?
     JA  L1                          ; yes
     cmp bl,cl                       ; no: is BL > CL?
     JBE next                        ; no: skip next statement
  L1:
        mov X,1                      ; set X to 1
  next:
```

Referencias

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

ORGANIZACIÓN Y PROGRAMACIÓN DE COMPUTADORAS

OPC

A – D 2023

MACRO Procedure

MACRO (DIRECTIVE)

- A *MACRO procedure* is a *named block* of Assembly Language instructions.
- Every time that this *named block* is invoked , Assembler during *preprocessing step*, *expands* a copy of the block, of instructions, into the programe code.
- The *expanded code* is passed to the assembly step, where it is checked for correctness.

Macro-Directives, in Section 10.2, Irvine.

Defining a Macro

- A macro must be defined before it can be used.
- Parameters are optional.
- Each parameter follows the rules for identifiers. It is a string that is assigned a value when the macro is invoked.
- Syntax:

```
macroname MACRO [parameter-1, parameter-2,...]
      statement-list
ENDM
```

myWriteStr Macro - an example

```
; Macro definition with an argument
myWriteStr  MACRO  buffer
            PUSH  EDX
            MOV  EDX, OFFSET buffer
            CALL  WriteString
            POP  EDX
ENDM

.DATA
   str1  BYTE "Welcome!", 0

.CODE
  myWriteStr  str1
```

```
; Assembler program, after initial
; preprocessing Assembly (step 0)
; myWriteStr  str1  is expanded out

.DATA
   str1  BYTE  "Welcome!", 0

.CODE
   PUSH  EDX
   MOV  EDX, OFFSET str1
   CALL  WriteString
   POP  EDX
```

Invoking Macros

- When you invoke a macro, each argument you pass matches a declared parameter.
- Each parameter is replaced by its corresponding argument when the macro is expanded.
- When a macro *expands*, it generates assembly language source code.
- Arguments are treated as simple text by the preprocessor.

myDumpMem Macro - an example

The myDumpMem macro streamlines calls to the link library's *DumpMem* procedure.

```
myDumpMem MACRO address, itemsCount, itemSize
    push ebx
    push ecx
    push esi
    mov  esi, address
    mov  ecx, itemsCount
    mov  ebx, itemSize
    call DumpMem
    pop  esi
    pop  ecx
    pop  ebx
ENDM
```

Referencias

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

ORGANIZACIÓN Y PROGRAMACIÓN DE COMPUTADORAS

OPC

A – D 2023

CONDITIONAL CONTROL FLOW MACRO DIRECTIVES

SELECTIVES

.IF *condition*, .ELSE, .ELSEIF *condition*, .ENDIF

REPETITIVES

.WHILE *condition*, .ENDW

.REPEAT, .UNTIL *condition*

.BREAK, .CONTINUE, used with .WHILE and .REPEAT

Macro-Directives, in Section 6.7, Irvine.

Conditional Control Flow Directives

Directive	Description
.BREAK	Generates code to terminate a .WHILE or .REPEAT block
.CONTINUE	Generates code to jump to the top of a .WHILE or .REPEAT block
.ELSE	Begins block of statements to execute when the .IF condition is false
.ELSEIF condition	Generates code that tests condition and executes statements that follow, until an .ENDIF directive or another .ELSEIF directive is found
.ENDIF	Terminates a block of statements following an .IF, .ELSE, or .ELSEIF directive
.ENDW	Terminates a block of statements following a .WHILE directive
.IF condition	Generates code that executes the block of statements if condition is true.
.REPEAT	Generates code that repeats execution of the block of statements until condition becomes true
.UNTIL condition	Generates code that repeats the block of statements between .REPEAT and .UNTIL until condition becomes true
.WHILE condition	Generates code that executes the block of statements between .WHILE and .ENDW as long as condition is true

.IF family macro-directives

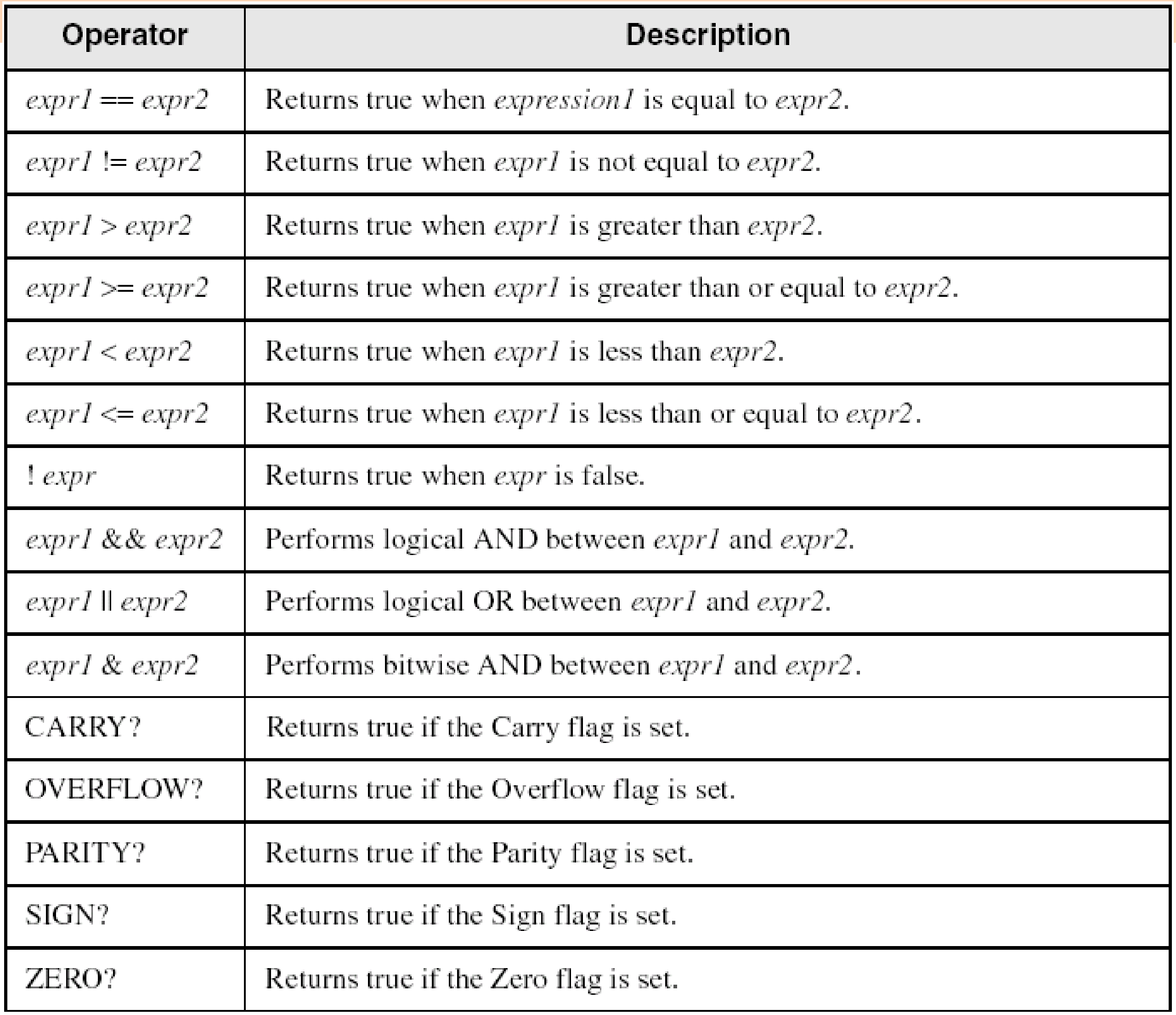
- .IF, .ELSE, .ELSEIF, and .ENDIF can be used to evaluate runtime expressions and create block-structured IF statements.
- Assembly examples:

```
.IF eax > ebx
    mov edx,1
.ELSE
    mov edx,2
.ENDIF
```

```
.IF eax > ebx && eax > ecx
    mov edx,1
.ELSE
    mov edx,2
.ENDIF
```

- MASM generates (macro-expands) "hidden" code for you, consisting of code labels, CMP and conditional jump instructions, and replacing *arguments*.

Relational and Logical Operators



Operator	Description
expr1 == *expr2*	Returns true when *expression1* is equal to *expr2*.
expr1 != *expr2*	Returns true when *expr1* is not equal to *expr2*.
expr1 > *expr2*	Returns true when *expr1* is greater than *expr2*.
expr1 >= *expr2*	Returns true when *expr1* is greater than or equal to *expr2*.
expr1 < *expr2*	Returns true when *expr1* is less than *expr2*.
expr1 <= *expr2*	Returns true when *expr1* is less than or equal to *expr2*.
! *expr*	Returns true when *expr* is false.
expr1 && *expr2*	Performs logical AND between *expr1* and *expr2*.
expr1 || *expr2*	Performs logical OR between *expr1* and *expr2*.
expr1 & *expr2*	Performs bitwise AND between *expr1* and *expr2*.
CARRY?	Returns true if the Carry flag is set.
OVERFLOW?	Returns true if the Overflow flag is set.
PARITY?	Returns true if the Parity flag is set.
SIGN?	Returns true if the Sign flag is set.
ZERO?	Returns true if the Zero flag is set.

AND and OR: Compound Expressions

- When using any directive, the || is the logical OR

```
.IF expression1  ||  expression2
    Statements . . .
.ENDIF
```

- When using any directive, the **&&** symbol is the logical AND

```
.IF expression1  &&  expression2
    Statements . . .
.ENDIF
```

Expression examples

```
eax > 10000h
val1 <= 100
val2 == eax
val3 != ebx
```

```
(eax > 0)  &&  (eax > 10000h)
(val1 <= 100)  ||  (val2 <= 100)
(val2 != ebx)  &&  !CARRY?
(dl < 0) || (dl > 79)
```

Signed and *Unsigned* Comparisons - 1

```
.DATA
val1   DWORD 5
result DWORD ?

.CODE
mov EAX,6
.IF EAX > val1
  mov result,1
.ENDIF
```

Expanded code:

```
    mov EAX,6
    cmp EAX,val1
    JBE @C0001
      mov result,1
@C0001:
```

MASM automatically generates an unsigned jump (JBE) because val1 is unsigned.

Signed and *Unsigned* Comparisons - 2

```
.DATA
val1   SDWORD 5
result SDWORD ?

.CODE
mov EAX,6
.IF EAX > val1
  mov result,1
.ENDIF
```

Expanded code:

```
    mov EAX,6
    cmp EAX,val1
    JLE @C0001
      mov result,1
@C0001:
```

MASM automatically generates a signed jump (JLE) because val1 is signed.

Signed and *Unsigned* Comparisons - 3

```
.DATA
result DWORD ?

.CODE
mov EBX,5
mov EAX,6
.IF EAX > EBX
  mov result,1
.ENDIF
```

Expanded code:

```
    mov EBX,5
    mov EAX,6
    cmp EAX,EBX
    JBE @C0001
      mov result,1
@C0001:
```

MASM automatically generates an unsigned jump (JBE) when both operands are registers . . .

Signed and *Unsigned* Comparisons - 4

```
.DATA
result SDWORD ?

.CODE
mov EBX,5
mov EAX,6
.IF SDWORD PTR EAX > EBX
  mov result,1
.ENDIF
```

Expanded code:

```
    mov EBX,5
    mov EAX,6
    cmp EAX,EBX
    JLE @C0001
      mov result,1
@C0001:
```

. . . unless you prefix one of the *register operands* with the *type* PTR operator. Then a signed jump is generated.

Example 1

```
SetCursorPosition  PROC
;  Sets the cursor position.
;  Receives:  DL = X-coordinate,  DH = Y-coordinate.
;  Checks the ranges of DL and DH.
;  Returns:  nothing
; -----------------------------------------------
. DATA
BadXCoordMsg BYTE " X-Coordinate out of range! " , 0Dh, 0Ah, 0
BadYCoordMsg BYTE " Y-Coordinate out of range! " , 0Dh, 0Ah, 0
. CODE
.IF ( dl < 0 )  | |  ( dl > 79 )
        mov  edx, OFFSET BadXCoordMsg
        call WriteString
        jmp  quit
.ENDIF
.IF ( dh < 0 )  | |  ( dh > 24)
        mov  edx, OFFSET BadYCoordMsg
        call WriteString
        jmp  quit
.ENDIF
call Gotoxy
quit:
    ret
SetCursorPosition  ENDP
```

```
. CODE
   ;  .IF ( dl < 0 )  | |  ( dl > 79 )
cmp dl, 000h
jb @C0002
cmp dl,  04Fh
jbe @C0001
@C0002 :
   mov edx, OFFSET BadXCoordMsg
   call WriteString
   jmp  quit
   ;  . ENDIF
@C0001:
   ;  .IF ( dh < 0 )  | |  ( dh > 24)
cmp dh, 000h
jb @C0005
cmp dh, 018h
jbe @C0004
@C0005:
   mov  edx, OFFSET BadYCoordMsg
   call WriteString
   jmp  quit
   ;  .ENDIF
@C0004:
call Gotoxy
quit:
ret
```

Example 2

```
; Determine registration based on two criteria:
; Average Grade
; Credits the person wants to take

.DATA
TRUE = 1
FALSE = 0
gradeAverage  WORD 275              ;  test value
credits          WORD 12              ;  test value
OkToRegister  BYTE ?
.CODE
mov OkToRegister, FALSE
.IF gradeAverage > 350
  movOkToRegister, TRUE
.ELSEIF (gradeAverage > 250 )  && ( credits <= 16)
  mov OkToRegister, TRUE
.ELSEIF ( credits <= 12 )
  mov OkToRegister, TRUE
.ENDIF
```

```
       mov  byte ptr OkToRegister, FALSE
;.IF gradeAverage > 350
       cmp  word ptr gradeAverage, 350
       jbe  @C0006
          mov  byte ptr OkToRegister, TRUE
       jmp  @C0008
@C0006:
;.ELSEIF (gradeAverage > 250 )  && ( credits <= 16)
       cmp  word ptr gradeAverage, 250
       jbe  @C0009
       cmp  word ptr credits, 16
       ja   @C0009
          mov  byte ptr OkToRegister, TRUE
       jmp  @C0008
@C0009 :
;.ELSEIF ( credits <= 12 )
       cmp  word ptr credits, 12
       ja   @C0008
       mov  byte ptr OkToRegister, TRUE
@C0008:
;.ENDIF
```

.WHILE family directives

Tests the loop condition before executing the loop body The .ENDW directive marks the end of the loop.

```
.WHILE condition
        statements
.ENDW
```

Example:

```
; Display integers 1 – 10:

mov EAX,0
.WHILE EAX < 10
   inc EAX
   call WriteDec
   call Crlf
.ENDW
```

.REPEAT family directives

Executes the loop body before testing the loop condition associated with the .UNTIL directive.

.REPEAT

    statements

.UNTIL *condition*

Example:

```
; Display integers 1 – 10:

mov EAX,0
.REPEAT
   inc EAX
   call WriteDec
   call Crlf
.UNTIL EAX == 10
```

Example 3: .WHILE nesting an .IF

```
;HLL




while(  op1 < op2  )
{
   op1++;
   if(  op1 == op3  )
      X = 2 ;
   else
      X = 3 ;
}
```

```
;ASM
. DATA
X       DWORD 0
op1   DWORD 2                    ;  test data
op2  DWORD 4                     ;  test data
op3  DWORD 5                     ;  test data

. CODE
        mov eax, op1
        .WHILE eax < op2
                inc eax
                mov op1, eax
                .IF eax == op3
                      mov X, 2
                .ELSE
                      mov X, 3
                .ENDIF
        .ENDW
```

Referencias

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

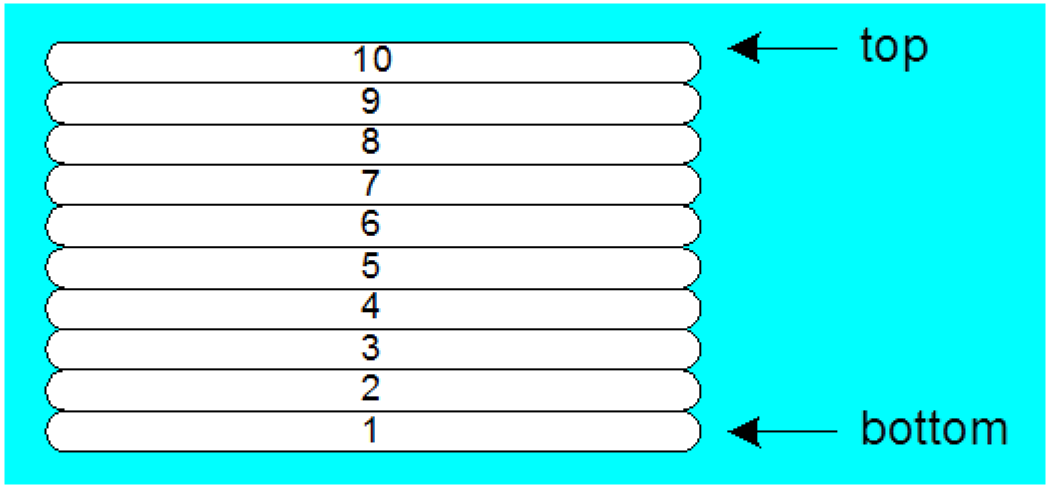
ORGANIZACIÓN Y PROGRAMACIÓN DE COMPUTADORAS

OPC

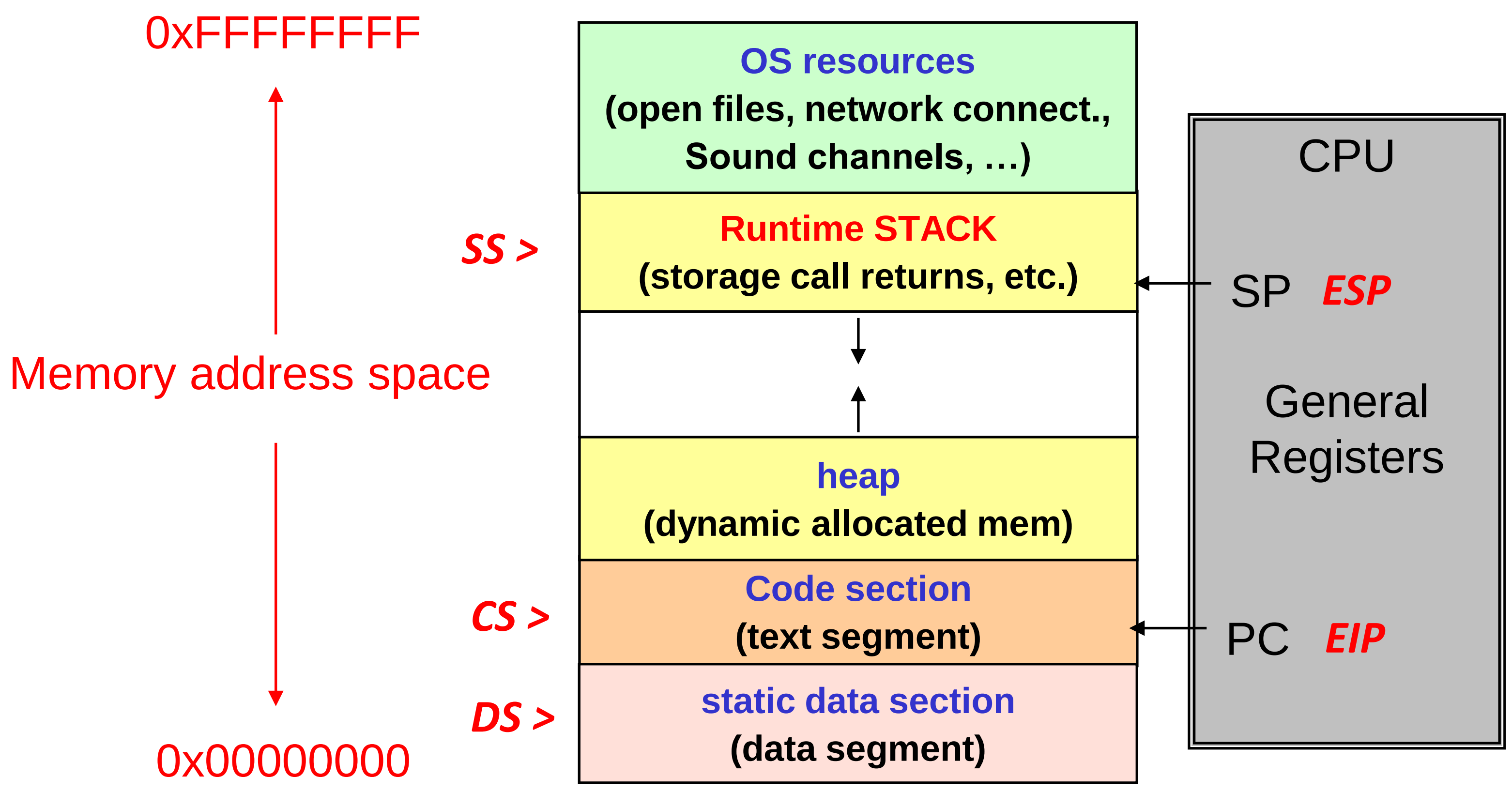
A – D 2023

STACK

- STACK data structure
- A STACK of values (data)
  - values are only added to the top (PUSH operation)
  - values are only removed from the top (POP operation)
  - LIFO (Last-in, First-out) data structure

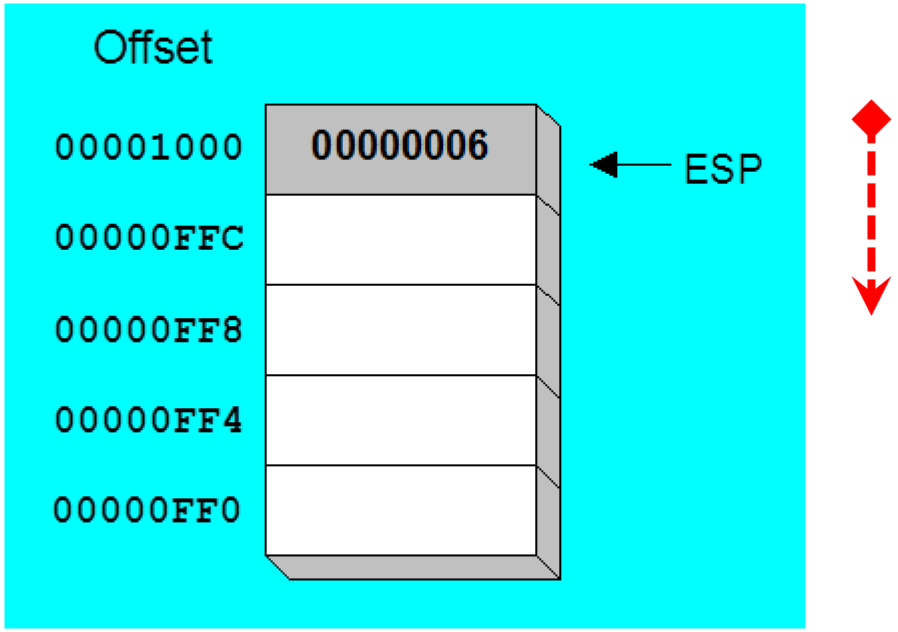


Process memory space and resources



Runtime STACK

- STACK inside every Process
- Managed by the CPU, using a stack pointer register
  - ESP (Extended Stack Pointer)
    - ✓ ESP always points to the last value added



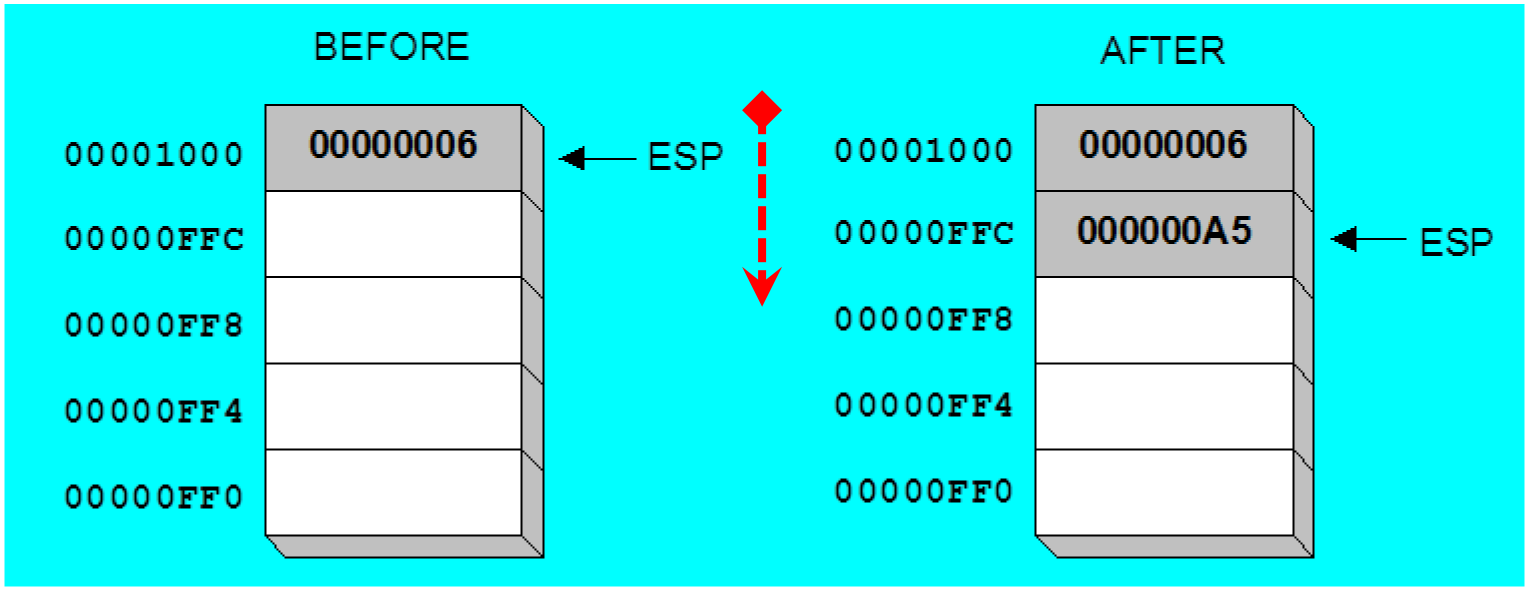
PUSH Instruction

- PUSH syntax, one operand
  - PUSH *reg/mem16* ; *for 2-Byte, 16-bit operand*
  - PUSH *reg/mem32* ; *for 4-Byte, 32-bit operand*
  - PUSH *reg/mem64* ; *for 8-Byte, 64-bit operand*

  - PUSH *imm32* ; *for 4-Byte, 32-bit operand*
  - PUSH *imm64* ; *for 8-Byte, 64-bit operand*

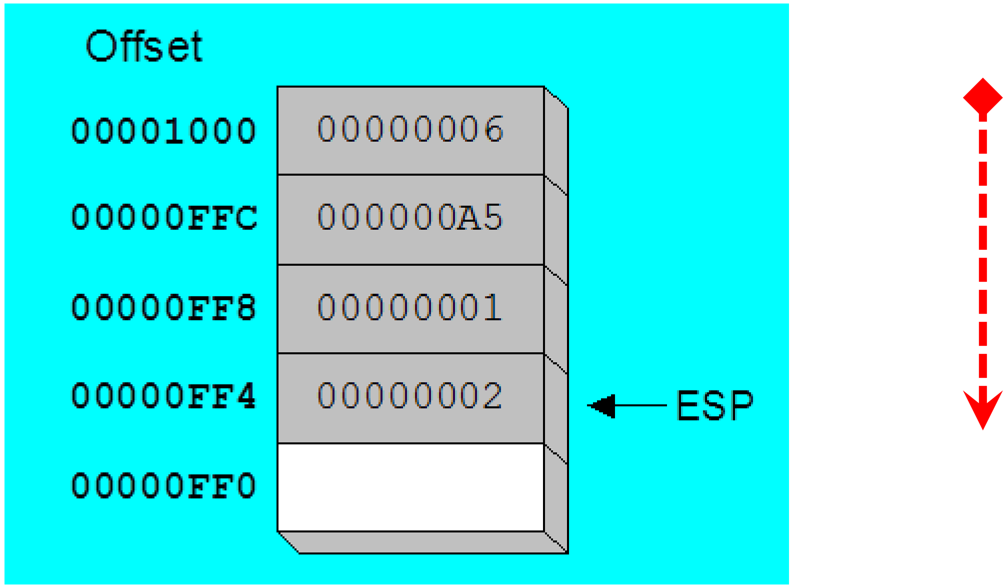
PUSH operation 1

- A PUSH operation decrements the stack pointer by *nByOp* (*2* [16 bits], *4* [32 bits] or *8* Bytes [64 bits]), and copies a *value* into the location pointed to by the stack pointer *ESP*.
    1. ESP ← ESP – *nByOp* *; nByOp=4 Bytes, a 32-bit operand*
    2. STACK [ESP] ← reg/var/value



PUSH operation 2

- Same stack after pushing two more integers:



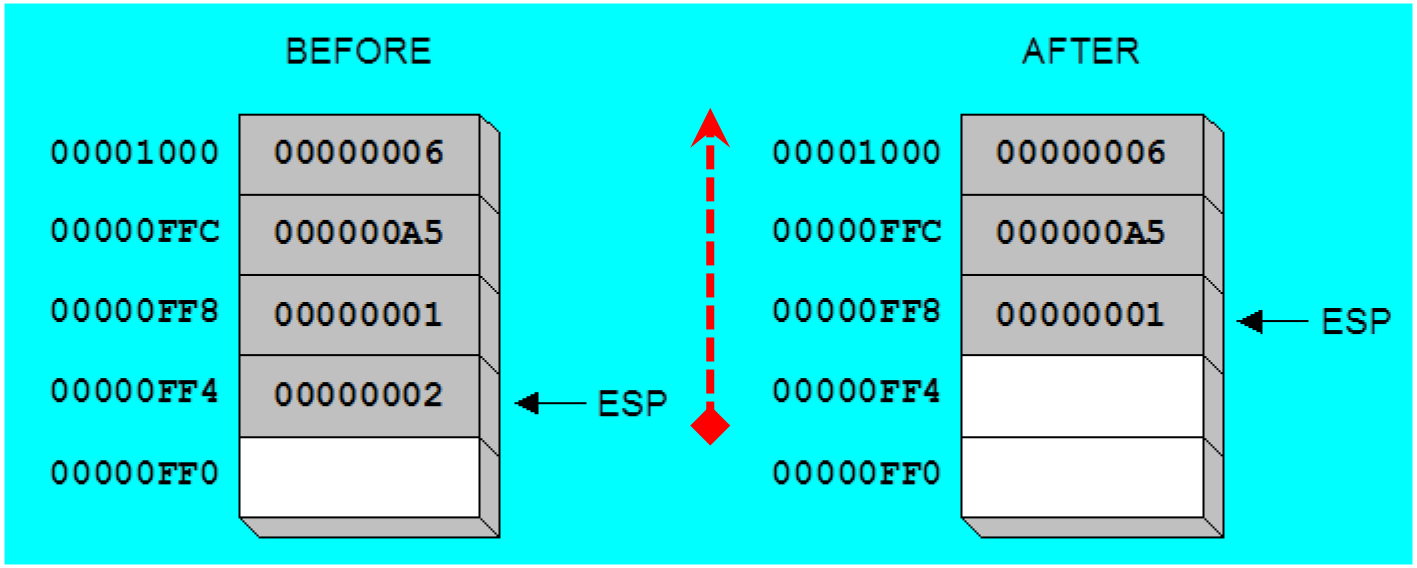
The stack grows downward. The area below ESP is always available (unless the stack has overflowed).

POP Instruction

- POP syntax, one operand
  - POP *reg/mem16* ; *for 2-Byte, 16-bit operand*
  - POP *reg/mem32* ; *for 4-Byte, 32-bit operand*
  - POP *reg/mem64* ; *for 8-Byte, 64-bit operand*

POP Operation

1. Copies value at STACK[ESP] into a register or variable.
   – Reg/Var ← STACK[ESP]
2. Adds *nByOp* to ESP, where *n* is either 2, 4 or 8.
   – value of *nByOp* depends on the attribute of the operand receiving the data
   – ESP ← ESP + *nByOp*



Using PUSH and POP

Save and restore registers when they contain important values. PUSH and POP instructions occur in the opposite order.

```
.DATA
dwordVal DWORD 675
.CODE
PUSH ESI                              ; push registers
PUSH ECX
PUSH EBX

MOV  ESI,OFFSET dwordVal              ; display some memory
MOV  ECX,LENGTHOF dwordVal
MOV  EBX,TYPE dwordVal
CALL DumpMem

POP  EBX                              ; restore registers
POP  ECX                              ; aware of LIFO
POP  ESI
```

Related Instructions

- PUSHFD and POPFD
  - push and pop the EFLAGS register (32-bit)
- PUSHAD pushes the 32-bit general-purpose registers on the stack
  - order: EAX, ECX, EDX, EBX, ESP, EBP, ESI, EDI
- POPAD pops the same registers off the stack in reverse order
- PUSHA and POPA, do the same for 16-bit registers
  - order: AX, CX, DX, BX, SP, BP, SI, DI

PUSHAD

- The value pushed for the ESP register is its value before prior to pushing the first register.

```
Temporary = ESP;
Push(EAX);
Push(ECX);
Push(EDX);
Push(EBX);
Push(Temporary);
Push(EBP);
Push(ESI);
Push(EDI);
```

POPAD

- The ESP register is incremented after each register is loaded.

```
EDI = Pop();          ESP=ESP+4
ESI = Pop();          ESP=ESP+4
EBP = Pop();          ESP=ESP+4
                      ESP = ESP + 4; //skip next 4 bytes of stack
EBX = Pop();          ESP=ESP+4
EDX = Pop();          ESP=ESP+4
ECX = Pop();          ESP=ESP+4
EAX = Pop();          ESP=ESP+4
```

Stack applications

- *Temporary save area* for registers, once the registers end a set of operations, their original values can be restored.
- The stack provides *temporary storage* for local variables.
- The *Call* for a procedure, saves *an address* that uses up the *Return* of the procedure called.
- HLL: The *formal* parameters and the *automatic* variables

Referencias

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

ORGANIZACIÓN Y PROGRAMACIÓN DE COMPUTADORAS

OPC

A – D 2023

Addressing Modes

- Addressing Mode.
  - refers to the way, in which, the CPU accesses the *effective values* from the operand(s) of an instruction

- Two general Addressing Modes:
  - Direct Addressing (seen up to now)
  - Indirect Addressing

Direct Addressing Mode

- With DIRECT addressing mode, in Operands
  - The operand has the *location* to place an *effective value*, or has the *content* to get an *effective value*,
  - Examples
    - mov EAX, alfa
    - mov beta, EBX
    - mov alfa, 354
    - Direct Addressing Operands
      - Register Direct O.: **EAX**, **EBX** (*location* or *content*)
      - Memory Direct O.: **alfa**, **beta** (*location* or *content*)
      - Immediate Direct O: **354** (*content*)

Indirect Addressing Mode -1

- With INDIRECT addressing mode, in operands
  - The operand has the *memory address* (offset) of the *location* to place a value, or has the *memory address* (offset) of the *content* to get a value
  - It means, the operand has a pointer (or a reference)
  - Example using pseudo-code - 1
    - mov ESI, OFFSET beta
    - mov EAX, *ind{ESI}*

Indirect Addressing Mode -2

– Example using pseudo-code - 2
  - add ESI, 4
  - mov EAX, *ind{ESI}*

Indirect Addressing Modes -3

Two approaches

✓ **Indirect** Operands

- – **[***reg***]**
    - ***reg*** contains an *address* (*offset*, *pointer*, or *reference*)

- **Indexed** Operands
    - – Coming next time!

Indirect Operands - 1

- ***reg***: could be any general-purpose register
  - Rmn, . . ., EAX, EBX, ECX, EDX, RAX … (64, 32, 16, and 8 bits)
  - Rxy, . . ., ESI, EDI, EBP, ESP, RSI … (64, 32 and 16 bits)
- ESI and EDI prefered registers
  - RSI, ESI: Source Index
  - RDI, EDI: Destination Index

- For each approach?
  - How come, you get the *effective address*? _____
  - How long (bits) must be the *effective address*? _____
  - What TYPE must be the *location* or the *content*? _____

Indirect Operands - 2

- An **indirect operand** holds the *initial address of a variable* (pointer, offset, or reference), usually a LIST, an ARRAY or STRING.
- It can be dereferenced (just like a pointer).

- In Protected Mode, any indirect operand can be any of the 32-bit or 64-bit general-purpose registers, surrounded by brackets.
- Also, a memory place (DWORD / QWORD) can hold an address value.

- General Protection Fault: in Protected Mode, if the effective address, points to an area outside your program's data segment, the CPU executes a *General Protection Fault (GP)*.

Indirect Operands - 3a

An indirect operand holds the *initial address of a variable*, usually a list, an array or string. It can be dereferenced (just like a pointer).

```
.DATA
val1 BYTE 10h,20h,30h

.CODE
MOV ESI,OFFSET val1
MOV AL,[ESI]                    ; ESI=0000 0000h; AL = 10h
                                ; Size of the location?___
                                ; Size of the content?___
INC ESI
MOV AL,[ESI]                    ; ESI=0000 ____h; AL = 20h

INC ESI
MOV AL,[ESI]                    ; ESI=0000 ____h; AL = 30h
```

Indirect Operands - 3b

Use PTR to clarify the size attribute of a memory operand.

```
.DATA
myCount WORD 5

.CODE
MOV ESI,OFFSET myCount
INC [ESI]                   ; error: ambiguous
    INC WORD PTR [ESI]      ; ok
```

Should PTR be used here?

```
ADD [ESI],20
```

yes, because [ESI] could point to a byte, word, or doubleword

Pointers - 3c

You can declare a pointer variable that contains the *offset* of another variable (it must be 32-bit or 64-bit long).

```
.DATA
       BYTE 16 DUP(?)
arrayW WORD 1000h,2000h,3000h
ptrW DWORD arrayW

.CODE
   MOV ESI, ptrW
   MOV AX,[ESI]               ; AX = 1000h
```

Alternate format:

```
ptrW DWORD OFFSET arrayW
```

Array Sum Example - 3d

Indirect operands are ideal for *traversing an array*. Note that the register in brackets must be incremented by a value that matches the *array type*.

```
.DATA
arrayW WORD 1000h,2000h,3000h

.CODE
   MOV ESI,OFFSET arrayW
   MOV AX,[ESI]
   ADD ESI,2             ;or better: ADD ESI,TYPE arrayW
   ADD AX,[ESI]
   ADD ESI,2
   ADD AX,[ESI]              ; AX = sum of the array
```

Direct and Indirect operands

- ***Direct operands*** in two-operand instructions (f.i. CMP)
  - CMP reg, reg
  - CMP reg, mem
  - CMP reg, imm
  - CMP mem, reg
  - CMP mem, imm

- CMP mem, mem ?_

- ***Indirect operands*** in two-operand instructions (f.i. CMP)
  - CMP reg, [reg] like direct operand CMP reg, ___
  - CMP [reg], reg like direct operand CMP ___, reg
  - CMP [reg], imm like direct operand CMP ___, imm

  - CMP [reg], [reg] ?_

Referencias

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

ORGANIZACIÓN Y PROGRAMACIÓN DE COMPUTADORAS

OPC

A – D 2023

Indirect Addressing Operands

- With Indirect operands
  - [*reg*]
    - ***reg*** contains an *address* (*offset*)

✓ With Indexed operands (also Indexed Addressing)
  - To get the *effective address* need to operate:
    - a *constant*, *address* and/or *imm*; and
    - a *register*
  - Effective address = initial offset + displacement

Indexed Operands - 4

- Types of indexed operands
  - Basic Indexed Operand (I.O.),
  - Adding Displacements in I. O.,
  - Scale Factors in I. O.

Basic Indexed Operand – 4a

An **indexed operand** is conformed with the addition of a *constant* plus the content of a *register*, to generate an effective address. The format of the operand:

- *constant* **[***reg***]** or **[***constant + reg***]**
  - ***constant*** must represent an *address/offset*
  - ***reg*** contains a *displacement value*

This addition is an *indexed* value with respect to an *address*.

Basic Indexed Operand – 4b

```
.DATA
; Starting address 500h
    arrayW WORD 1000h,2000h,3000h
;      position:    0,     1,     2
; displacement:    0,     2,     4  = position*(TYPE arrayW)

.CODE
   ; Basic Indexed or INDEXED
   MOV ESI,0                      ; index=0, displacement=0
   MOV AX, arrayW[ESI]            ; AX = 1000h
   MOV AX, [arrayW + ESI]         ; Alternative

   ADD ESI,2                      ; index=1, displacement=2
   ADD AX, [arrayW + ESI]            ; AX = ___
   ADD AX, arrayW[ESI]               ; Alternative
```

Adding Displacements I. O. – 4c

Format of the operand: **`[`*`reg + constant`*`]`**

reg: must hold an initial address

constant: *meaning fixed displacement*

Adding Displacements I. O. – 4d

```
.DATA
; Starting address 500h
    arrayW WORD 1000h,2000h,3000h
;      position:    0,     1,     2
; displacement:    0,     2,     4

.CODE
    ; ADDING DISPLACEMENTS
   MOV ESI, OFFSET arrayW   ; ESI holds an offset
   MOV AX, [ESI + 2]              ; AX = 2000h
   MOV AX, [ESI + 4]              ; AX = 3000h
```

Scale Factors in I. O. – 4e

Format of the operand: ***`offset [reg * constant]`***

offset: fixed initial address

reg: holds an HLL position

constant: type of element (byte, word, dword, …)

The *reg* works like an *index* of an HLL array element.

The *scale factor* is the size (TYPE) of the array component.

Displacement= *reg * constant*

Index Scaling – 4f

You can scale an indirect or indexed operand to the offset of an array element. This is done by multiplying the index by the array's TYPE.

```
.DATA
;         postion  0,  1,  2,  3,  4,  5
    arrayB BYTE  10, 11, 12, 13, 14, 15
; displacement:    0,  1,  2,  3,  4,  5

    arrayW WORD  10, 11, 12, 13, 14, 15
; displacement:    0,  2,  4,  6,  8, 10

    arrayD DWORD 10, 11, 12, 13, 14, 15
; displacement:    0,  4,  8, 12, 16, 20
```

Index Scaling – 4g

```
.CODE
MOV ESI,4           ; position 4
MOV AL, arrayB[ESI * TYPE arrayB]     ; 14

MOV BX, arrayW[ESI * TYPE arrayW]      ; 0014

MOV EDX, arrayD[ESI * TYPE arrayD]     ; 00000014
```

Referencias

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

ORGANIZACIÓN Y PROGRAMACIÓN DE COMPUTADORAS

OPC

A – D 2023

Creating Procedures

- Large problems can be divided into smaller tasks to make them more MANAGEABLE, MODULAR, and UNDERSTANDABLE

- Declared using **PROC** and **ENDP** directives
- A *named block* of statements
- Must be assigned a name (*valid identifier*)

- A procedure is the ASM equivalent of a *Java Method, C* or *C++ Procedure or Function*.
- Following is an assembly language procedure named sample:

```
nomPro PROC
    .
    .
    RET     ;return instruction
nomPro ENDP
```

Documenting Procedures

Suggested documentation for each procedure:

- A description of all tasks accomplished by the procedure.
- Receives: A list of input parameters; state their usage and requirements.
- Returns: A description of values returned by the procedure.
- Requires: Optional list of requirements called preconditions that must be satisfied before the procedure is called.

If a procedure is called without its preconditions satisfied, it will probably not produce the expected output.

Example: Sum3 Procedure

```
main PROC
    . . .
    CALL Sum3
    . . .
    EXIT
main ENDP
;----------------------------------------
Sum3 PROC    ; Sum3(EBX, ECX, EDX)
; Calculates and returns the sum of three 32-bit integers.
; Receives: EBX, ECX, EDX, the three integers. May be
; signed or unsigned.    Like Irvine
; Returns: EBX = sum, and the status flags (Carry,
; Overflow, etc.) are changed.    Like Irvine
; Requires: nothing
;----------------------------------------
    ADD EBX,ECX
    ADD EBX,EDX
    RET
Sum3 ENDP
END main
```

Global Values with Registers

- REGISTERS. All CPU Registers are *global* to *every procedure* in the .CODE segment.

- Using CPU Registers inside a procedure, to pass values, is *not the best practice*; but, at least, the Registers allow some extent of flexibility, passing values at CALLing and RETurning time.

CALL and RET Instructions

- The **CALL** instruction *calls* a procedure
  - Pushes the *offset* of next instruction, after CALL, on the stack: PUSH EIP
  - Copies the *address* of the called procedure into EIP (Program Counter): MOV EIP, OFFSET nomProc
- The **RET** instruction returns from a procedure
  - pops top of stack, having the *offset* of the next-instruction *offset*, into EIP (Program Counter): POP EIP

CALL-RET Example 1

0000025 is the offset of the instruction immediately following the CALL instruction

00000040 is the offset of the first instruction inside `Sum3 and also its offset`

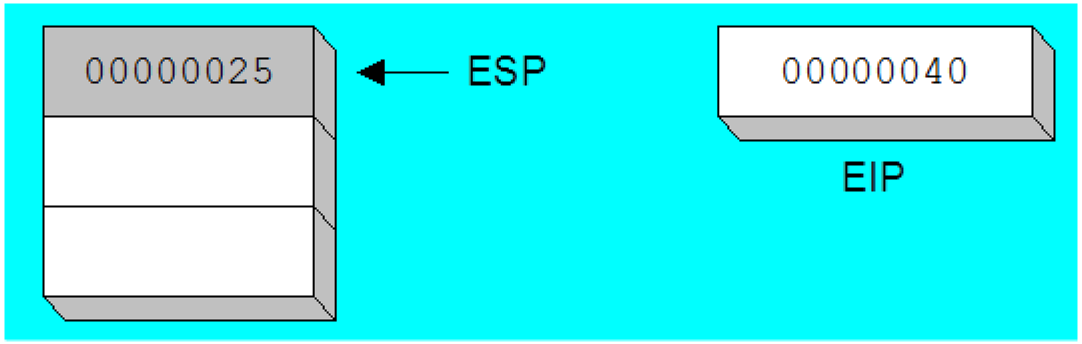
```
main PROC
                                        ;EIP=00000020h
    00000020 CALL Sum3                 ;EIP=00000025h
                                        ;EIP=00000040h
    00000025 MOV EAX,EBX
    .  .  .
main ENDP

Sum3 PROC
                                        ;EIP=00000040h
    00000040 ADD EBX,ECX
    .  .  .  ADD EBX,EDX
    .  .  .
    000000B0 RET                       ;EIP=00000025h
Sum3 ENDP

END main
```

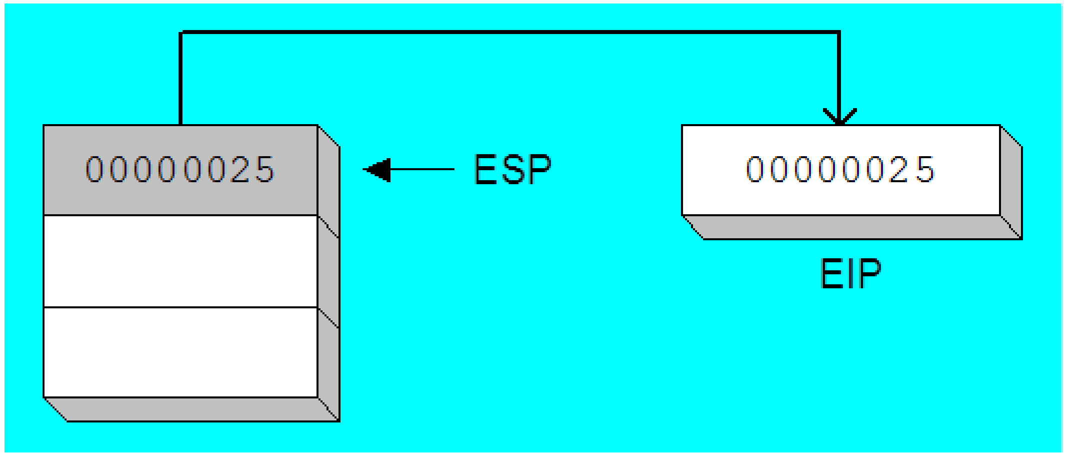
CALL & RET Example 2

The **CALL** instruction pushes 00000025 onto the stack, and loads 00000040 into EIP

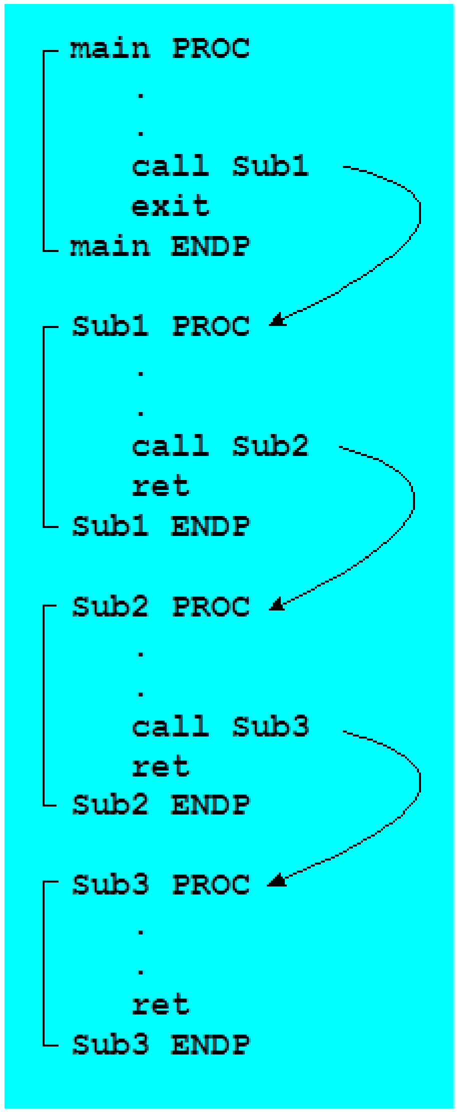


(stack shown before RET executes)

The **RET** instruction pops 00000025 from the stack into EIP



Nested Procedure Calls



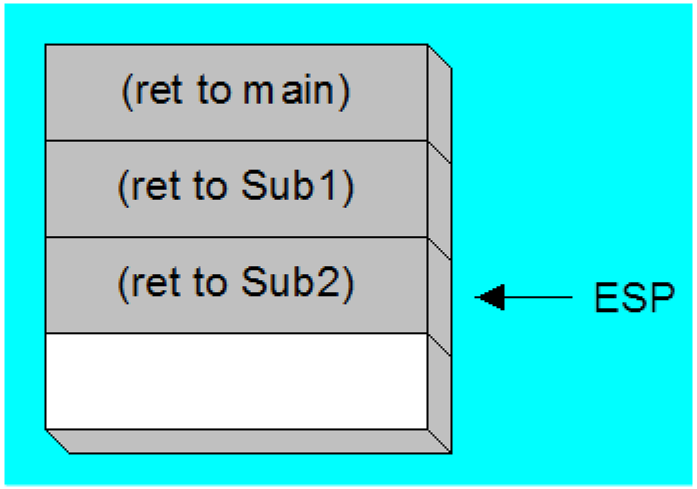
```
main PROC
    .
    .
    call Sub1
    exit
main ENDP

Sub1 PROC
    .
    .
    call Sub2
    ret
Sub1 ENDP

Sub2 PROC
    .
    .
    call Sub3
    ret
Sub2 ENDP

Sub3 PROC
    .
    .
    ret
Sub3 ENDP
```

By the time Sub3 is called, the stack contains all three return addresses:



Global Data Labels in Procedures

The **ArraySum** procedure calculates the sum of an array. It makes two references to specific variable names:

```
ArraySum PROC
     MOV ESI,0                           ; array index
     MOV EAX,0                           ; set the sum to zero
     MOV ECX,0                           ; set to zero, counter

     .WHILE ECX < LENGTHOF myArray
       ADD EAX, myArray[ESI]             ; add each integer to sum
       ADD ESI,TYPE myArray              ; point to next integer
       inc ECX
     .ENDW                               ; repeat for array size

     MOV theSum, EAX                     ; store the sum
     RET
ArraySum ENDP
```

What if you wanted to invoke to calculate the sum two or three with different arrays?

Global Data with Registers

This version of **ArraySum** returns the sum of any doubleword array whose address is in ESI. The sum is returned in EAX:

```
ArraySum PROC
; Receives: ESI points to an array of doublewords,
;      EBX = TYPE doubleword            Like Irvine
;      ECX = number of array elements.
; Returns: EAX = sum
;--------------------------------------------------
    MOV EAX,0                          ; set the sum to zero

    .WHILE ECX > 0
      ADD EAX,[ESI]                    ; add each integer to sum
      ADD ESI,EBX                      ; point to next integer
      dec ECX
    .ENDW                              ; repeat for array size

    RET
ArraySum ENDP
```

Global Scope of Data Labels

- All Data Labels (*variables*) defined in the .DATA segment are global to every procedure defined in the .CODE segment, of the same file.
- Using global variables (Data Labels) inside a procedure, to *pass values*, is a bad practice.
- A *better conceptual approach*, is to hold a *Local Area for each procedure*, having own local variables, and passing arguments using stack and CPU-Registers at CALLing and RETurning time.

Procedure Arguments / Parameters

- A good procedure might be *usable* in many different programs
  - but carefully when general-purpose registers and even worst when offset variable are used

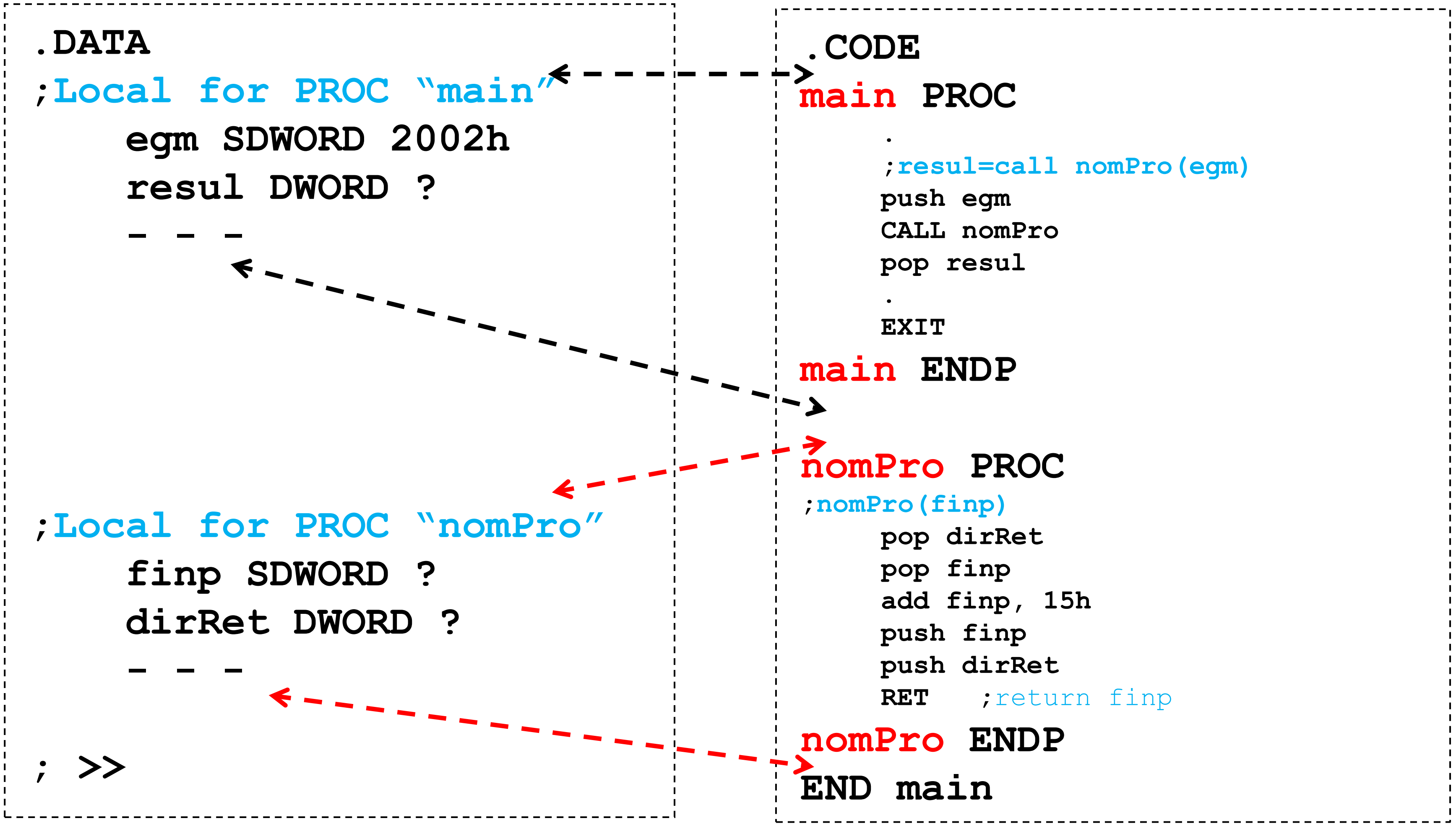
- *Arguments / Parameters* help to make procedures flexible because parameter values can change at runtime

Stack application – passing values

- The CPU, *pushes* an *offset*, with *CALL or INVOKE to* a PROCEDURE.
- The CPU, *pops* an offset, with *RETurn from* a PROCEDURE.

- When calling a PROCEDURE, you can pass arguments *PUSHing* them on the stack.
- Inside a PROCEDURE, you can recover arguments *POPing* them from the stack.

.DATA vision for Procedures



```
.DATA
;Local for PROC "main"
    egm SDWORD 2002h
    resul DWORD ?
    - - -



;Local for PROC "nomPro"
    finp SDWORD ?
    dirRet DWORD ?
    - - -

; >>
```

```
.CODE
main PROC
        .
        ;resul=call nomPro(egm)
        push egm
        CALL nomPro
        pop resul
        .
        EXIT
main ENDP

nomPro PROC
;nomPro(finp)
        pop dirRet
        pop finp
        add finp, 15h
        push finp
        push dirRet
        RET    ;return finp
nomPro ENDP
END main
```

Referencias

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