# **Computer Architecture** (Practical Class)
# *Assembly*: Unconditional and Conditional Jumps; More Arithmetic Operations

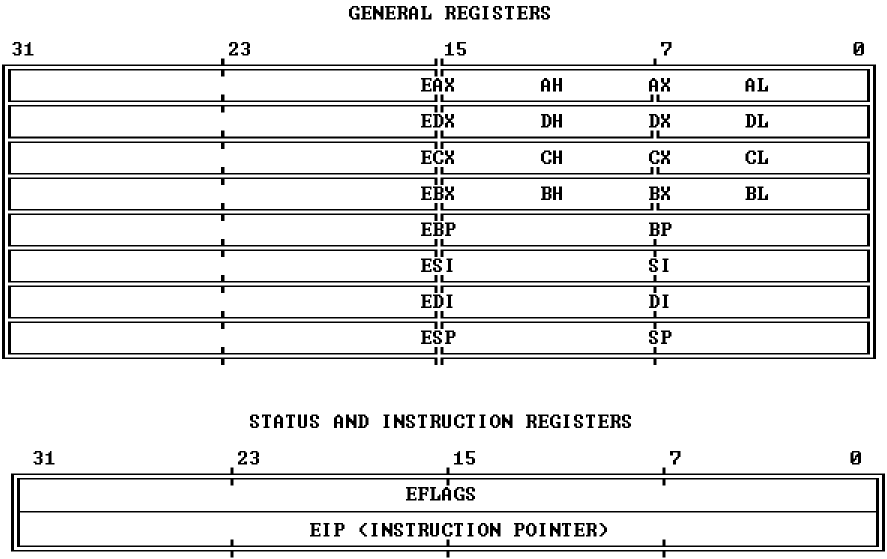
Luís Nogueira Raquel Faria

Departamento de Engenharia Informática
Instituto Superior de Engenharia do Porto

{lmn,arf}@isep.ipp.pt

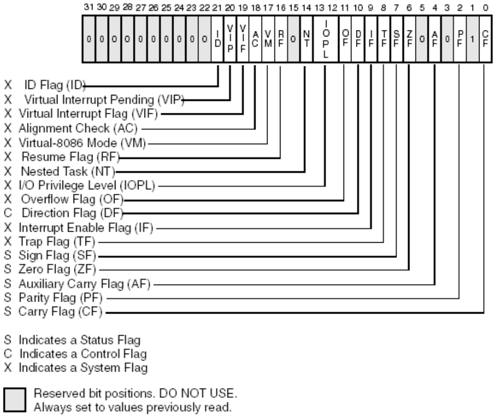
2020/2021

# IA32 Registers



# The EFLAGS Register

- 32-bit register used as a collection of bits representing Boolean values to store the results of operations and the state of the processor
- Each bit is a Boolean flag (1 - active/true, 0 - inactive/false)
- As instructions execute, they may change some of these flags

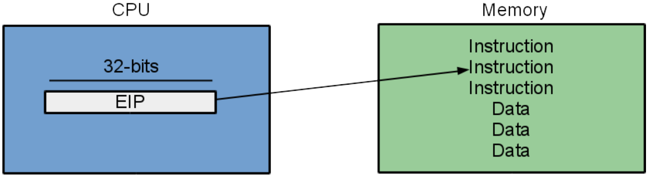


# The EFLAGS Register - Important Flags

- CF - carry flag (bit 0)
  - Set on most significant bit carry or borrow; cleared otherwise
- PF - parity flag (bit 2)
  - Set if least significant eight bits of result contain an even number of "1"bits; cleared otherwise
- ZF - zero flag (bit 6)
  - Set if result is zero; cleared otherwise
- SF - sign flag (bit 7)
  - Set equal to the most significant bit of result (0 if positive, 1 if negative)
- OF - overflow flag (bit 11)
  - Set if result is too large (a positive number) or too small (a negative number, excluding its sign bit) to fit in destination operand; cleared otherwise

# The EIP Register

- The Instruction Pointer - EIP - register contains the memory address of the next instruction to be executed.
- After the execution of the instruction, the EIP register is automatically increased to the address of next instruction, *unless* ...



`jmp` *address*

- The `jmp` instruction changes the EIP register to address, a location within the program to jump to, usually denoted by a label

**Unconditional Jump Example**

```
.global jmptest
jmptest:
    ...
    movl %eax, %ebx
    addl %ebx, %ecx
    jmp end

    # this line is never executed!
    movl $1, %eax

end:
    movl $10, %ebx

    ...
    ret
```

# Controlling Execution Flow: Conditional Jumps

- Conditional jumps are taken or not depending on the state of the EFLAGS register at the time the branch is executed
- Each conditional jump instruction examines specific flag bits to determine whether the condition is proper for the jump to occur. Some examples:
  - `JE` – Jump if equal (ZF=1)
  - `JL` – Jump if less (SF<>OF)
  - `JG` – Jump if greater (ZF=0 e SF=OF)
- Similarly to the `jmp` instruction, they only take one argument indicating the address within the program to jump to

**Important note**

- Before a conditional jump **the EFLAGS must be set appropriately by some operation...**

`cmp` *operand1*, *operand2*

- The compare instruction is the most common way to evaluate two values for a conditional jump
- Compares the second operand with the first operand by executing a subtraction (operand2 – operand1)
- Does not change the operands, but changes the `EFLAGS` register
- Examples:
  - if operand2 == operand1 then ZF (zero flag) = 1
  - if operand2 > operand1 then SF (sign flag) = 0
  - if operand2 < operand1 then SF (sign flag) = 1
- The `CMP` instruction can be applied to 8 bits (b), 16 bits (w) or 32 bits (l)

**Controlling Execution Flow Example**

```
    ...
    #  compare esi with ebx (32 bits)
    cmpl %ebx, %esi
    jg jmp_is_greater
    je jmp_is_equal
    jl jmp_is_less
jmp_is_greater:
    movl $1, %eax
    jmp end
jmp_is_equal:
    movl $0, %eax
    jmp end
jmp_is_less:
    movl $-1, %eax
end:
    ...
```

JC – Jump if carry (CF=1)

- The *carry* flag is set if a mathematical operation on an **unsigned** integer value generates a carry or a borrow for the most significant bit
- The jump is taken if the *carry* flag is active (1)

**Test Carry Example**

```
.global addtest_carry

addtest_carry:
    ...
    addl %eax, %ebx

    # jump if carry
    jc output_com_carry
    movl $0, %eax
    jmp fim

output_com_carry:
    movl $1, %eax

fim:

        ..
        ret
```

# Arithmetic Operations: Detect *Carry* and *Overflow* (II)

JO – Jump if overflow (OF=1)

- The *overflow* flag is used in **signed** integer arithmetic when a positive value is too large, or a negative value is too small, to be properly represented in the register
- The jump is taken if the *overflow* flag is active (1)

**Test Overflow Example**

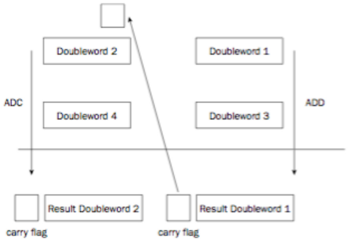
```
.global addtest_overflow

addtest_overflow:
    ...
    movb $-127, %bl
    addb $-10, %bl

    # jump if overflow
    jo with_overflow
    movl $0, %eax
    jmp fim

with_overflow:
    movl $1, %eax

fim:

    ...
    ret
```

# Arithmetic Operations: Add with Carry (ADC)

`adc` *origin*, *destination*

- The `ADC` instruction can be used to add two integer values, along with the value contained in the *carry* flag (set by a previous addition)
- Performs the operation: $destination = destination + origin + CF$

- *origin* can be a memory address, a constant value or a register
- *destination* can be a memory address or a register
- A memory address for *origin* and *destination* cannot be used simultaneously
- The `ADC` instruction can add numbers of 8(b), 16(w) or 32(l) bits.



**Add With Carry (ADC) Example**

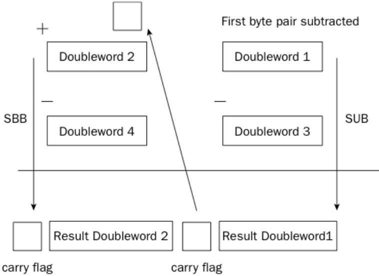
```
.global adctest
adctest:
    ...
    movb $0xFF, %al
    movb $0x1, %ah
    movb $0x1, %bl
    movb $0x0, %bh

    # bl = bl + al (8 bits)
    addb %al, %bl

    # bh = bh + ah + CF (8 bits)
    adcb %ah, %bh

    ...
    ret
```

# Arithmetic Operations: Subtract with Borrow (SBB)

`sbb` *origin*, *destination*

- The `SBB` instruction can be used to subtract two integer values, along with the value contained in the *carry* flag (set by a previous subtraction)
- Performs the operation: $destination = destination - (origin + CF)$

- *origin* can be a memory address, a constant value or a register;
- *destination* can be a memory address or a register.
- A memory address for *origin* and *destination* cannot be used simultaneously;
- The `SBB` instruction can subtract numbers of 8(b), 16(w) or 32(l) bits.



- Sometimes we need to convert a value to a larger data type

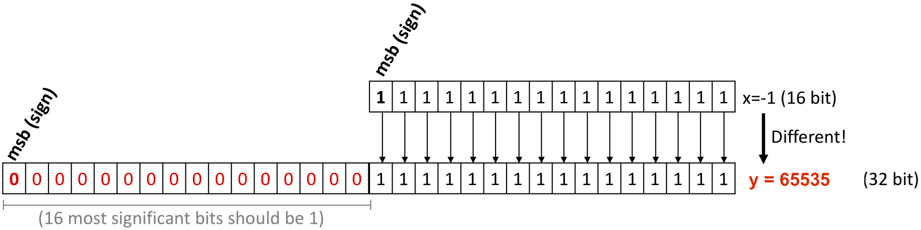
**C Code to move a 16 bit value to 32 bit variable**

```
short int x=-1; // 16 bits (signed)
int y=0;         // 32 bits (signed)
y=(int)x;        // move 16 to 32 bit
```

**Incorrect way to move a value from 16 to 32 bits**

```
movl $0, %eax   # init register
movw $-1, %ax   # 16 bits to register
movl %eax, y    # 32 bits to variable
```

- **The assembly code on the right does not work always**. Why ?
  - What if x contains a negative value ?



## Sign Extension (2/2)

**When converting values to a larger data type...**

- We need to copy the sign value throughout the upper bits. This is called **sign extension** !
- We could manually copy the sign value... Fortunately there is an assembly instruction to do this: `movsX` (move with sign extend)

`movsX` origin, *destination*

- *Origin* can be a memory address or a register (8 or 16 bit)
- *Destination* can be a register of 16(w) or 32(l) bits

**Correct sign extesion**

```
movb $-1, %al
movsbw %al, %ax
movsbl %al, %eax
movswl %ax, %eax
```

`mul` *origin*

- The `MUL` instruction multiplies two unsigned integers
- Performs the operation: $destination = origin \times operand2$
- *origin* can be a memory address, a constant value or a register
- *operand2* is the EAX, AX or AL register, depending on the size of *origin*
- *destination* is the EDX:EAX, DX:AX or AX registers, depending on the size of *origin*;

| Size of *origin* | *operand2* | *destination* |
| --- | --- | --- |
| 8 bits | AL | AX |
| 16 bits | AX | DX:AX |
| 32 bits | EAX | EDX:EAX |

- The `MUL` instruction can multiply numbers of 8(b), 16(w) or 32(l) bits.

# Arithmetic Operations: Unsigned Multiplication - Example

**Unsigned Multiplication Example**

```
.global multest
multest:

    ...

    movw $200, %ax
    movw $2, %cx

    # multiply %cx by %ax
    # result in %dx:%ax
    mulw %cx

    ...

    ret
```

# Arithmetic Operations: Signed Multiplication (IMUL)

`imul` *origin*
`imul` *origin*, *destination*
`imul` *multiplier*, *origin*, *destination*

- The `IMUL` instruction multiplies two signed integers
- `imul` *origin*
  - Similar to `MUL`
- `imul` *origin*, *destination*
  - Performs *destination* = *destination* × *origin*
  - *origin* can be a memory address, a constant value or a register
  - *destination* can be a 16 or 32-bit register
- `imul` *multiplier*, *origin*, *destination*
  - Performs *destination* = *origin* × *multiplier*
  - `multiplier` is an integer constant
  - `origin` is a memory address or register
  - `destination` is a 16 or 32-bit register

**Signed Multiplication Example**

```
.global imultest
imultest:
    ...

    movw $200, %ax
    movw $2, %cx

    # multiply %ax by %cx
    # result in %cx
    imulw %ax, %cx

    # multiply 4 by %cx
    # result in %dx
    imulw $4, %cx, %dx

    ...
    ret
```

# Arithmetic Operations: Unsigned and Signed Division (DIV and IDIV)

`div` *divisor*
`idiv` *divisor*

- The `DIV/IDIV` instruction is used for unsigned/signed division
- Performs the operations:
  - $quotient = dividend \div divisor$
  - $remainder = dividend \mod divisor$
- *divisor* can be a memory address or a register
- *dividend* is the EDX:EAX, DX:AX or AX register, depending on the size of *divisor*
- The *quotient* and *remainder* of the division are put in different sections of the EAX and EDX registers, depending on the size of *divisor*

| Size of *divisor* | *dividend* | *quotient* | *remainder* |
|---|---|---|---|
| 8 bits | AX | AL | AH |
| 16 bits | DX:AX | AX | DX |
| 32 bits | EDX:EAX | EAX | EDX |

# Arithmetic Operations: Unsigned and Signed Division - Example

**Unsigned Division Example**

```
.global divtest
divtest:
...

  # dividend: ax
  movw $100, %ax
  # divisor: cl
  movb $3, %cl

  # divides %ax by %cl
  # remainder in %ah
  # quotient in %al
  divb %cl

  ...
  ret
```

## Important note

- Always initialize **all** bits of the dividend!

**Very bad idea!**

```
.global bad_div

bad_div:
  ...
  # dividend: eax
  movl $100, %eax
  # divisor: ecx
  movl $3, %ecx

  # divides %edx:%eax by %ecx
  # Problem: the unkown content in %edx becomes part of the dividend

  # remainder in %edx
  # quotient in %eax
  divl %ecx

  ...
  ret
```

## CBW/CWD/CDQ instructions

- The `cbw` (convert byte to word), `cwd` (convert word to double word) and `cdq` (convert double word to quad word) instructions can be used to produce a correct dividend before a division instruction

- `cbw` - converts the signed byte in AL to a signed word in AX by copying the sign bit of AL
- `cwd` - converts the signed word in AX to a signed double word in EAX by copying the sign bit of AX
- `cdq` - converts the signed double word in EAX to a signed quad word in EDX:EAX by copying the sign bit of EAX to all bits of EDX

**Correct sign extension**

```
.global div_ok

div_ok:
  ...
  # dividend: eax
  movl $-100, %eax
  # extends signal to %edx
  cdq

  # divisor: ecx
  movl $3, %ecx

  # divides %edx:%eax by %ecx (remainder in %edx, quotient in %eax)
  idivl %ecx
  ...
  ret
```

# Practice

**Division Snippet ($x$ and $y$ are 32 bit unsigned integers)**

```
1 ...
2 movl x,%eax
3 cdq
4 divl y
5 ...
```

- Active learning activity: Given the above code, choose the correct and most complete option.
  - A. The code divides $x$ by $y$. The quotient will be in $EAX$ and the remainder in $EDX$.
  - B. The code divides $y$ by $x$. The quotient will be in $EAX$ and the remainder in $EDX$.
  - C. The code divides $x$ by $y$. The quotient will be in $EAX$ and the remainder in $EDX$, but line 3 is not needed.
  - D. None of the above.

- Write the following in Assembly:
  - A function that performs: $D = A \times B \div C$ (global integers defined in Assembly).
  - A function that returns the greater of two integer variables $A$ and $B$.