Lecture 3: Binary Arithmetic, Logic and Addressing

Table of contents

0.1 Objectives:

- In this lecture, we focus on ***binary*** arithmetic and logic instructions that operate on 8-, 16-, and 32-bit numeric data encoded as signed or unsigned binary integers. The instructions include
  - increment and decrement
  - addition and subtraction

  - Multiplication and division
  - Bitwise Boolean AND, OR, XOR and NOT

- Indirect addressing modes.

1 Addition and Subtraction

1.1 INC and DEC Instructions

Syntax

```
INC     <operand>
DEC     <operand>
```

- `INC` (increment) and `DEC` (decrement) instructions add 1 and subtract 1 from an operand, respectively.
- The operand must be either a memory or register. The format is

```
inc   reg/mem
dec   reg/mem
```

1.2 ADD and SUB Instructions

Syntax

```
ADD   <destination>, <source>
SUB   <destination>, <source>
```

- The `ADD` instruction adds a source operand to a destination operand. The sum is stored in the destination operand, and the source operand's value is unchanged.

- The SUB instruction subtracts a source operand from a destination operand. The result is stored in the destination operand. The source operand's value is unchanged

Rules

1. Both operands must be the same size
2. Both operands cannot be memory operands
3. The instruction pointer register (EIP, RIP) cannot be a destination operand

The following operand types are permitted for these instructions (both for ADD and SUB):

```
add   reg, reg
add   reg, mem
add   reg, imm
add   mem, reg
add   mem, imm
```

1.3 NEG Instruction

Syntax

```
NEG   <operand>
```

- The NEG (negate) instruction reverses the sign of an operand by converting the number to its two's complement. The following operands are permitted:

```
neg   reg/mem
```

- Intel implements NEG instruction by subtracting the operand from zero (i.e., `<operand> = 0 - <operand>`)

1.4 Example

```
        section .data
var1    dd      1000h           ; 4096
var2    dd      2000h           ; 8192

        section .text

        mov     eax, [var1]     ; EAX = 1000h
        inc     eax             ; EAX = 1001h
        dec     DWORD [var2]    ; VAR2 = 1FFFh
        add     eax, [var2]     ; EAX = 1001h + 1FFFh = 3001h
        sub     [var2], eax     ; [VAR2] = 1FFFh - 1001h = 0FFEh
        neg     [var1]          ; [VAR1] = FFFF F000h (-4096)
```

1.5 Implementing Arithmetic Expressions

In the following examples, we are going to learn how to translate high-level expressions into assembly instructions:

Example 1.1.

```
        b = a;
```

Solution:

```
        mov     eax, [a]          ; EAX = a
        mov     [b], eax          ; b = EAX
```

Example 1.2.

```
    b = a + 1;
```

Solution:

```
    mov     eax, [a]        ; EAX = a
    inc     eax             ; EAX ++
    mov     [b], eax        ; b = EAX
```

Another solution:

```
    mov     eax, [a]
    add     eax, 1
    mov     [b], eax
```

The following solution is not acceptable because it changes the value of variable a:

```
    add     [a], 1          ; ERROR: don't change the variable a
    mov     eax, [a]
    mov     [b], eax
```

The following solution has syntax error: memory to memory operands

```
    mov     [b], [a]        ; ERROR: memory to memory operands
    add     [b], 1
```

Example 1.3.

```
  c = b - a;
```

Solution:

```
        mov     eax, [a]            ; EAX = a
        mov     ebx, [b]            ; EBX = b
        sub     ebx, eax            ; EBX -= EAX
        mov     [c], ebx            ; c = EBX
```

Another solution:

```
        mov     eax, [b]            ; EAX = b
        sub     eax, [a]            ; EAX = EAX - a
        mov     [c], eax            ; c = EAX
```

Example 1.4.

```
    y = 5 - y;
```

Solution:

```
        mov     eax, 5              ; EAX = 5
        sub     eax, [y]            ; EAX = EAX - y
        mov     [y], eax            ; y = EAX
```

Another solution:

```
        neg     [y]                 ; y = -y
        add     [y], 5              ; y += 5
```

Example 1.5.

```
    c = 2 * (a + b) + 10;
```

Solution:

```
; evaluate (a + b)
mov     eax, [a]
add     eax, [b]

; 2 * (a + b)
add     eax, eax

; 2 * (a + b) + 10
add     eax, 10

; c = <expr>
mov     [c], eax
```

1.6 Flags Affected by Arithmetic Instructions

- When a CPU executes arithmetic instructions, it consequently updates the status flags. (see lecture 1)
- Status flags provide information whether the output is
  - Positive or negative
  - zero or not zero
  - too large or too small to fit into the destination operand
  - the parity of least significant byte
- The status flags can be used to activate conditional branching instructions (will be discussed in next lecture).

Status Flags	Is set when
CF	unsigned integer overflow
OF	signed integer overflow
ZF	the output is zero
SF	the output is negative

Status Flags	Is set when
PF	The least significant byte of the output have even number of 1 bits
AF	there a 1 bit carried out of position 3 in the least significant byte of the output

1.6.1 Carry Flag

Carry flag is set when:

- There is a carry *out* of the most significant bit (MSB) when adding two numbers:

$$\begin{array}{rr} & 01000001 \\ & +11000010 \\ \hline CF = 1 & 00000011 \end{array}$$

- There is a borrow into the MSB when subtracting two numbers

$$\begin{array}{rr} & 00000001 \\ & -00000010 \\ \hline CF = 1 & 11111111 \end{array}$$

Remember

- In unsigned arithmetic, watch the carry flag to detect errors
- In signed arithmetic, the carry flag tells you nothing interesting

1.6.2 Overflow Flag

Overflow flag is when:

- the sum of two numbers with the sign bits off yields a result number with sign bit on.

$$
\begin{array}{rr}
 & 0\,0\,1\,0\,0\,0\,0\,1 \\
 & +\,0\,1\,1\,0\,0\,0\,1\,0 \\
\hline
OF = 1 & 1\,0\,0\,0\,0\,0\,1\,1
\end{array}
$$

- the sum of two numbers with the sign bits on yields a result number with sign bit off.

$$
\begin{array}{rr}
 & 1\,0\,1\,0\,0\,0\,0\,1 \\
 & +\,1\,0\,1\,0\,0\,0\,1\,0 \\
\hline
OF = 1 & 0\,1\,0\,0\,0\,0\,1\,1
\end{array}
$$

1.6.3 How the ALU calculates the Overflow flag

This material is optional reading.

- The value that carries out of the MSB is exclusive ORed with the carry into the MSB of the result.
- The value that borrows out of the MSB is exclusive ORed with the borrow into the MSB of the result.
- The resulting value of XOR is placed in the Overflow flag.
- Example

$$
\begin{array}{rr}
 & 1\,0\,0\,0\,0\,0\,0\,0 \\
 & +\,1\,1\,1\,1\,1\,1\,1\,1 \\
\hline
OF = 1 & 0\,1\,1\,1\,1\,1\,1\,1
\end{array}
$$

The carry into the MSB is 0 and the carry out of the MSB is 1, OF = 0 XOR 1 = 1

$$\begin{array}{rr} & 00000001 \\ & -00000010 \\ \hline OF = 0 & 11111111 \end{array}$$

The borrow into the MSB is 1 and the borrow out of the MSB is 1, OF = 1 XOR 1 = 0

Remember

- In signed arithmetic, watch the overflow flag to detect errors
- In unsigned arithmetic, the overflow flag tells you nothing interesting

1.7 Which Instructions Affect Flags

- In general, each time the processor executes an instruction, the flags are altered to reflect the result. However, some instructions don't affect any of the flags, affect only some of them, or may leave them undefined.
- Here is a summary of previous instructions:

Instructions	Flags
Transfer instructions (such as MOV, XCHG)	None
ADD and SUB	All flags
INC and DEC	All flags except CF
NEG	All flags (remember NEG = subtracting operand from zero)

2 Multiplication and Division

2.1 Terms

- When multiplying two numbers, the first number is **multiplicand**, the second number is **multiplier** and the result is **product**.
- If the size of multiplicand is n-bit and the size of multiplier is m-bit, then the size of product is at most (n+m)-bit.
- When dividing two numbers, the first number is **dividend**, the second number is **divisor**, and the result is **quotient**. So, the dividend is the number that is divided by the divisor. The number left behind after the division is called the **remainder**.

2.2 `MUL` Instruction

Syntax

```
        MUL     <multiplier>
```

- This is the *unsigned* version of multiplication
- The multiplier can be either `mem` or `reg`.
- There are three formats depending on the size of multiplier (in 32-bit mode):

Multiplier	Multiplicand	Product
`reg/mem8`	`AL`	`AX`
`reg/mem16`	`AX`	`DX:AX`
`reg/mem32`	`EAX`	`EDX:EAX`
`imm`	NOT ALLOWED	

2.2.1 Flags Affected

The OF and CF flags are set to 0 if the upper half of the result is 0; otherwise, they are set to 1. The SF, ZF, AF, and PF flags are undefined.

2.2.2 Example

Before		*Instruction Executed*	*After*	
AX:	00 05		DX:	00 00
BX:	00 02	`mul bx`	AX:	00 0A
DX:	?? ??		OF,CF	0

Before		*Instruction Executed*	*After*	
EAX:	00 00 00 0A		EDX:	00 00 00 00
EDX:	?? ?? ?? ??	`mul eax`	EAX:	00 00 00 64
			OF,CF	0

Before		*Instruction Executed*	*After*	
AX:	22 05		AX:	04 FB
len:	FF	`mul BYTE [len]`	OF,CF:	1

2.3 `IMUL` Instruction

Syntax

```
IMUL    <source>
IMUL    <destination>, <source>
IMUL    <destination>, <source1>, <source2>
```

2.3.1 Single-Operand Form

- This form is identical to that used by the `MUL` instruction.

2.3.2 Two-Operand Form

- The destination operand is multiplied by the source operand. The product is truncated and stored in the destination operand.
- The destination operand must be register (either `reg16` or `reg32` only)
- The valid formats are

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

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

2.3.3 Three-Operand Form

- The first source operand is multiplied by the second operand. The product is truncated and stored in the destination operand
- The destination operand must be register (either `reg16` or `reg32`). The first operand must be a register or memory with same size as the destination operand. The second operand must be immediate operand
- The valid formats are:

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

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

2.3.4 Flags Affected

- For the one operand form of the instruction, the CF and OF flags are set when significant bits are carried into the upper half of the result and cleared when the result fits exactly in the lower half of the result.
- For the two- and three-operand forms of the instruction, the CF and OF flags are set when the result must be truncated to fit in the destination operand size and cleared when the result fits exactly in the destination operand size.
- The SF, ZF, AF, and PF flags are undefined.

> **Remember**
>
> With the two- and three- operand forms, the result is truncated to the length of the destination before it is stored in the destination register. Because of this truncation, the CF or OF flag should be tested to ensure that no significant bits are lost.

2.3.5 Examples

Example 2.1.

```
        section .data
varW    dw     4
varDW   dd     4

        section .text
        mov      ax, -16          ; AX = -16
        mov      bx, 2            ; BX = 2

        imul     bx, ax           ; BX = -32
        imul     bx, 2            ; BX = -64
        imul     bx, [varW]       ; BX = -256

        mov      eax, -16
```

```
        mov     ebx, 2

        imul    ebx, eax        ; EBX = -32
        imul    ebx, 2          ; EBX = -64
        imul    ebx, [varDW]    ; EBX = -256
```

Example 2.2. Translate `y = x * 7` to Assembly, assuming that `x` and `y` are declared as singed integer.

```
        mov    eax, [x]
        imul   eax, 7               ; eax = eax * 7
        mov    [y], eax
```

Or

```
        mov    eax, [x]
        mov    ebx, 7
        imul    ebx                 ; eax = eax * ebx
        mov    [y], eax
```

Or

```
        imul   eax, [x], 7
        mov    [y], eax
```

2.4 `DIV` Instruction

Syntax

```
        DIV     <divisor>
```

- This the unsigned version of division

- There are three formats depending on the size of divisor (in 32-bit mode):

Divisor	Dividend	Quotient	Remainder
`reg/mem8`	`AX`	`AL`	`AH`
`reg/mem16`	`DX:AX`	`AX`	`DX`
`reg/mem32`	`EDX:EAX`	`EAX`	`EDX`
immediate	NOT ALLOWED		

Remember

If you did not set the upper half of the dividend, don't forget to clear it using one of the following instructions:

```
xor   ax, ax
mov   ah, 0
sub   edx, edx
```

2.4.1 Flags Affected

The CF, OF, SF, ZF, AF, and PF flags are undefined.

2.5 `IDIV` Instruction

- This is the signed version of division
- It has the same syntax as `DIV`
- If $q = a/b$ where q is the quotient, a is the dividend, and b is divisor, then $a = q * b + r$
- The sign of the remainder is always the same as the sign of the dividend
- If the divisor is 0, an interrupt called `#DE` is generated to terminate the program.

Remember

The dividend must be sign-extended before the division takes place. As such, x86 architecture provides `CBW`, `CWD`, and `CDQ` to sign extend the dividend.

2.5.1 CBW Instruction

- **C**onvert **B**yte to **W**ord. This instruction requires no operand.
- Impact: If MSB of `AL` is 1, then `AH = 0xFF`, otherwise `AH = 0`.

2.5.2 CWD Instruction

- **C**onvert **W**ord to **D**oubleword. This instruction requires no operand.
- Impact: If MSB of `AX` is 1, then `DX = 0xFFFF`, otherwise `DX = 0`.

2.5.3 CDQ Instruction

- **C**onvert **D**oubleword to **Q**uadword. This instruction requires no operand.
- Impact: If MSB of `EAX` is 1, then `EDX = 0xFFFF−FFFF`, otherwise `EDX = 0`.

Example 2.3.

```
mov     al, 125       ; dividend = AL = 125
cbw                   ; AH = 0
mov     bl, 2         ; BL = 2
div     bl            ; AL =125/2=62=3Dh, AH = 01h

mov     al, -125      ; AL = -125 = 83h
cbw                   ; AX = FF83h
```

```
        mov     bl, 2
        idiv    bl              ; AL = -125/2 = -62 = 0C2h, AH = -1 = 0FFh

        mov     dx, 0           ; clear dividend, high
        mov     ax, 8003h       ; dividend, low
        mov     cx, 100h        ; divisor
        div     cx              ; AX = 0080h, DX = 0003h
```

In the remaining examples, we are going to translate from high-level language to assembly language. All variables are 32-bit signed integers

Example 2.4.

```
      f = 9/5*c + 32;
```

```
        mov     eax, [c]        ; EAX = c
        imul    eax, 9          ; EAX = 9 * c
        mov     ebx, 5          ; EBX = 5 (divisor)
        cdq                     ; Sign-extend EAX to EDX
        idiv    ebx             ; EAX = EAX / EBX
        mov     [f], eax        ; f = EAX
```

Example 2.5.

```
      v4 = (v1 * 5) / (v2 - 3);
```

```
        mov   eax, [v1]         ; EAX = v1
        imul  eax, 5            ; EAX = EAX * 5
        mov   ebx, [v2]         ; EBX = v2
        sub   ebx, 3            ; EBX = v2 - 3
        cdq                     ; Sign-extend EAX to EDX
        idiv  ebx               ; EAX = EAX / EDX
        mov   [v4], eax         ; v4 = EAX
```

Example 2.6.

```
v4 = -v1 * 5/(-v2 % v3)
```

```
mov   eax, [v1]      ; EAX = v1
neg   eax            ; EAX = -v1
imul  ecx, eax, 5    ; ECX = -v1 * 5

mov   eax, [v2]      ; EAX = v2
neg   eax            ; EAX = -v2
mov   ebx, [v3]      ; EBX = v3
cdq                  ; sign-extend EAX to EDX
idiv  ebx            ; EAX = EAX / EBX, Rem = EDX

mov   eax, ecx       ; Set EAX = ECX (enumerator)
mov   ebx, edx       ; Set EBX = EDX (denominator)
cdq
idiv  ebx            ; EAX = EAX / EBX

mov   [v4], eax      ; v4 = EAX
```

3 Logic Instructions

- The Intel instruction set contains the AND, OR, XOR, NOT, and many more instructions , which directly implement boolean operations on binary bits.

Syntax

```
AND     <destination>, <source>
OR      <destination>, <source>
XOR     <destination>, <source>
NOT     <destination>
```

- The following table describes the operation of each instruction

Instruction	Description
AND	Bitwise logical AND operation between a source operand and a destination operand
OR	Bitwise logical OR operation between a source operand and a destination operand
XOR	Bitwise logical exclusive-OR operation between a source operand and a destination operand
NOT	Bitwise NOT operation on a destination operand

Rules

1. Both operands must be the same size
2. Both operands cannot be memory operands
3. The instruction pointer register (EIP, RIP) cannot be a destination operand

The following operand types are permitted for these instructions:

```
and   reg, reg
and   reg, mem
and   reg, imm
and   mem, reg
and   mem, imm
not   reg
not   mem
```

3.0.1 Flags Affected

- The OF and CF are cleared.
- The SF, ZF, and PF are set according to the result.
- The state of AF is undefined.

3.1 Implementations

3.1.1 Clearing/Masking Bits

- The AND instruction allows us to clear 1 or more bits in an operand without affecting other bits. This technique is called **bit masking**.
- To clear a bit (or more), use AND with a **mask** that has a 0 in the position which is to be cleared. All other positions in the mask contain 1.
- For example, suppose we have the bit string $1110\,0110$ and we want to clear bit 2 (change it to 0). We can use mask $1111\,1011_2$ as follows:

```nasm
    mov     AL, 1110-0110b       ; bit string
    and     AL, 1111-1011b       ; AL=1110-0010
```

3.1.2 Setting Bits

- The OR instruction allows us to set 1 or more bits in an operand without affecting any other bits.
- To set a bit (or more), use OR with a mask that has 1 in the position which is to be set. All other positions in the mask contain 0.
- The following code:

  ```
          or      AL, 0000-0100b     ; set bit 2
  ```

  will set bit at position 2 to 1 and leave other bits unchanged.

3.1.3 Toggling Bits

- We can toggle or switch the value of individual bits from 1 to 0 or vice versa.
- To toggle a bit, use XOR with a mask that has 1 in the position which is to be toggled. All other positions in the mask contain 0.
- Consider the following example

$$\begin{array}{r}
\text{Original Value} = 1\,0\,0\,1\,0\,1\,1\,0 \\
\text{XOR'ed Value} = 0\,0\,0\,0\,1\,1\,1\,1 \\
\hline
\text{Toggled Value} = 1\,0\,0\,1\,1\,0\,0\,1
\end{array}$$

3.1.4 Testing a Bit

- To test a bit, use AND with a mask that has 1 in the position which is to be tested. All other positions in the mask contain 0.
- Example, suppose we have the bit string $1110\,0110$ and we want to test bit 5 (check its value). We can use mask $0010\ 0000_2$ and the AND:

```
        mov     AL, 1110-0110b     ; our bit string
        and     AL, 0010-0000b     ; The result = 0010 0000
```

- Since bit 5 was set, the result is a nonzero value. However, suppose we want to test bit 4. We use mask $0001\ 0000$ and the AND operation:

```
        mov     AL, 1110-0110b     ; our bit string
        and     AL, 0001-0000b     ; The result = 0000 0000
```

- Since bit 4 was not set, the result is a zero value.
- So after the AND, we need to test the result; if the result is nonzero, the bit is set, otherwise it is not.

4 Indirect Addressing Modes

- Memory locations (where the objects are located) can be specified:
  - **Directly** as static value (also known as displacement)
  - **Indirectly** through an address computation made up of one or more of the following components:
    1. Displacement
    2. Base
    3. Index
    4. Scale-Factor
- The offset address which results from adding these components is called an **effective address**.

In the following section, we consider the following data definition

```
        section .data
v1      db      50, 62
arrX    dw      1, 2, 3, 4, 5
matr    dd      10, 11, 12, 13
        dd      20, 21, 22, 23
```

4.1 Displacement Addressing Mode

- A displacement is 8-, 16-, or 32-bit constant value or expression
- Displacement alone represents a direct offset to the operand.
- The syntax is:

  `[ <disp32> ]`

  where `disp32` is 32-bit displacement value

- Example

```
mov   al, [v1]           ; AL = 50
add   al, [v1 + 1]       ; AL = 50 + 62 = 112
```

4.2 Base Addressing Mode

- Base alone represents an indirect offset to the operand.
- In this mode, the offset address is contained in a *general purpose register*. We say that the register acts as **a pointer** to the memory location.
- The syntax for 32-bit architecture is:

```
[ <reg32> ]
```

- Example:

```
mov     esi, arrX      ; ESI points to the 1st element in arr
mov     ax, 0          ; set sum to zero
add     ax, [esi]      ; add arrX[0] to sum, AX = 1
add     esi, 2         ; move pointer to next element
add     ax, [esi]      ; add arrX[1] to sum, AX = 1+2 = 3
add     esi, 2         ; move pointer to next element
add     ax, [esi]      ; add arrX[2] to sum, AX = 3+3 = 6
```

4.3 Base + Displacement Addressing Mode

- A base register and a displacement can be used together to access a field of a record. The base register holds the address of the beginning of the record, while the displacement holds a static offset to the field
- This mode is specially useful when we need to access parameters in a procedure activation record. We will elaborate this mode when we discuss procedures.

- The syntax is

```
[ <reg32> + <disp> ]

[ <reg32> - <disp> ]
```

- Example:

```
mov   esi, v1         ; esi points to v1
mov   al, [esi + 0]   ; AL = 50
mov   bl, [esi + 1]   ; BL = 62
```

4.4 Index Addressing Mode

- In this mode, there are two forms:

```
[index + displacement]

[index*scale + displacement]
```

- The index is a general purpose register except ESP
- For 32-bit mode, the syntax:

```
[ <reg32> + <disp32> ]

[ <reg32>*<scale> + <disp32> ]
```

where <scale> can be either 1, 2, 4, or 8

- In index mode, the displacement locates the beginning of an array, the index register holds the subscript of the desired array element, and scale represents the data type.
- For example, `arrX[3]` can be translated to assembly as `[ecx*2 + arrX]` where the register `ECX` holds the value `3`.
- Example

```
        mov     ax, 0                 ; set sum to zero
        mov     esi, 0                ; set array index to 0
        add     ax, [esi*2 + arrX]    ; add arrX[0] to sum, AX = 1
        inc     esi                   ; increment the index by 1
        add     ax, [esi*2 + arrX]    ; add arrX[1] to sum, AX = 1+2 =
        inc     esi                   ; increment the index by 1
        add     ax, [esi*2 + arrX]    ; add arrX[2] to sum, AX = 3+3 =
```

4.5 Base + Index Addressing Mode

- In this mode, the base register holds the address of a dynamic array, while the index register holds offset address or subscript index.
  - In 2D arrays, the base register holds the address of the beginning of a row. The index register holds the column's subscription
- The syntax is:

```
[ <base> + <index> ]

[ <base> + <index>*<scale> ]
```

where `<base>` and `<index>` are reg32.

4.6 Base + Index + Displacement Addressing Mode

- This mode is a combination of previous modes
- This mode is suitable for traversing array of records.

4.7 Summary

The following diagram, adopted from Intel manual summarizes the memory addressing modes:

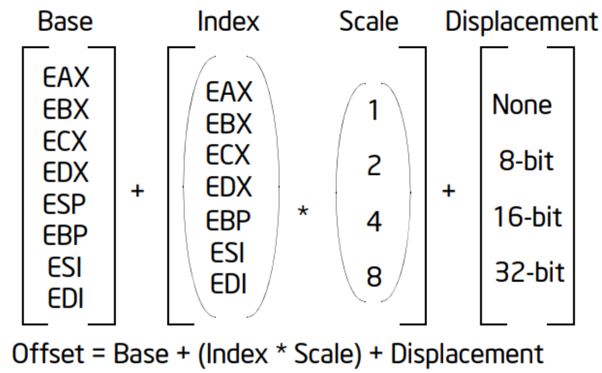

Figure 1: Offset (or Effective Address) Computation