Lecture 2: Assembly Language Fundamentals

Table of contents

1 An Assembly Program

Let us consider the following program:

```
;
;   Adding two numbers
;
      global _main

      section .data
sum   dd      0

      section .code
_main:
      mov     eax, 5      ; let eax = 5
      add     eax, 6      ; add 6 to eax
      mov     [sum], eax  ; store result into sum

      ret
```

1.1 Assembly Statements

- An assembly program consists of statements. There are three types of statements: instructions, directives, and macros.
- An **instruction** is translated by the assembler into one or more bytes of object code (machine code), which will be executed at run time.
- A **directive** tells the assembler to perform some special tasks, such as allocating memory space for a variable or creating a procedure.
- A **macro** is "shorthand" for a sequence of other statements.
- By default, assembly is not case-sensitive.
- Any statement has up to four fields, separated by at least one white space:

```
[label:]    instruction    [operand(s)]    [;comment]
```

The four fields are displayed in the following diagram:

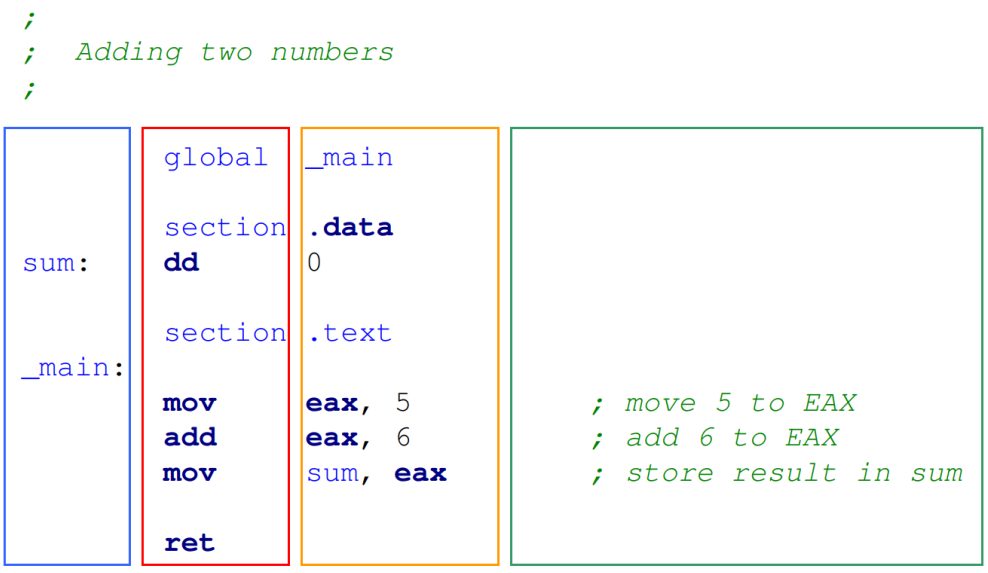


Figure 1: The layout of source lines. Blue rectangle for label; Red rectangle for either instruction or directive; Orange rectangle for operands or arguments; Green rectangle for comment

1.2 Comments

- A comment always starts with a semicolon character ; and ends with newline character, as shown as red rectangles in the following diagram:

```
;
;  Adding two numbers
;

        global  _main

        section .data
sum:    dd      0

        section .text
_main:
        mov     eax, 5          ; move 5 to EAX
        add     eax, 6          ; add 6 to EAX
        mov     sum, eax        ; store result in sum

        ret
```

- At the top of every program, you should add a comment block that contains:
  - The name of the source code file,
  - The date you created the file,
  - The date you last modified the file,
  - The name of the person who wrote it,
  - The name and version of the assembler used,
  - An overview description of the what the program or library does.
  - A copy of the commands used to build the file
  - Here is an example:

```
; ------------------------------------------------------------
; Source name   : addtwo.asm
; Version       : 1.0
; Created date  : 10/4/2020
; Last modified : 22/8/2024
; Author        : Khalid Elbadawi
; Compiling     : nasm -fwin32 addtwo.asm (NASM version 2.15)
; Description   : Store the result of 5 + 6
; ------------------------------------------------------------
```

1.3 Program Sections

- Generally, programs written in assembly language are divided into **sections**, which represent **memory segments**.
- Sections are defined by the assembler directive.
- In NASM, sections are created by using `section` keyword followed by the section name, such as `.data`, `.bss`, or `.text`
  - The `.data` section: contains data definitions of initialized data items
  - The `.bss` section: contains data definitions of uninitialized data items
  - The `.text` section: contains the actual machines instructions (i.e., the code) that make up your program
- In the following diagram, the data section is enclosed by the GREEN rectangle and code (or text) section is enclosed by the RED rectangle:

```
;
;  Adding two numbers
;

        global  _main

        section .data
sum:    dd      0

        section .text
_main:
        mov     eax, 5          ; move 5 to EAX
        add     eax, 6          ; add 6 to EAX
        mov     sum, eax        ; store result in sum

        ret
```

1.4 Label field - User Identifier

- A label field is a programmer-chosen **identifier**. It might identify a variable, a constant, a procedure or an instruction label.
- Rules for choosing an identifier are (in NASM):
  - Must not exceed more than 247 characters
  - Must start with a letter, underscore or ?.
  - Subsequent characters may also be digits, @, #, $, or dot.
  - Must not be a reserved word
- In general, it's good idea to use descriptive names for identifiers.
- Labels, in NASM, are case sensitive.

1.5 Reserved Words

- There are different types of reserved words:
    1. Instruction mnemonics (the Instruction Set), such as `ADD`, `SUB`, `MOV`.
    2. Register names, such as `EAX`, `AX`, `AL`, `ch`
    3. Directives, such as `DB`, `DWORD`.
    4. Operand attributes, such `FAR`, `NEAR`, `BYTE`.
    5. Operators used in constant expressions.
- Reserved words are NOT case sensitives

1.6 Literals

1.6.1 Integer Literals

Syntax

```
[{+|-}] digits [radix]
```

where *radix* can be

1. `h` (hexadecimal),
2. `o` or `q` (octal),
3. `d` (decimal), and
4. `b` (binary)

Example

Integer constants	Base
`198`	decimal
`0200`	decimal
`-2d`	decimal
`+101b`	binary
`1100_0001b`	binary

`107o`	octal
`107q`	still octal (more readable)
`23bh`	hexadecimal
`23Bh`	hexadecimal
`0ah`	hexadecimal
`0xC8`	hexadecimal (C syntax)

1.6.2 Character or String Literals

- Characters and character string must be enclosed in single quotes, double quotes or back quotes.
  - `"A"` or `"Hello"`
  - `'A'` or `'Hello'`
  - `` `A` `` or `` `Hello` ``
- Characters are stored in contiguous memory and translated into their ASCII codes by the assemblers, so there is no difference between using `"A"` and `41h` in an assembly program.

Character/String Literal	Value in Hex
`'m'`	6Dh
`"School"`	53h, 63h, 68h, 6Fh, 6Fh, 5Ch
`"Joe"`	4Ah, 6Fh, 65h
`"A Bank."`	41h, 30h, 42h, 61h, 6Eh, 6Bh, 2Eh
`"Joe's"`	4Ah, 6Fh, 65h, 27h, 73h

1.7 Instructions

- An instruction consists of an *instruction mnemonic* followed by zero, one, two, or three *operands*.

- An **instruction mnemonic** is a short word that identifies an instruction, such as `mov`, `add`, and `sub`.
- An **instruction operand** can be used for input or output for the instruction. For example, consider the following instruction:

```
add   eax, [x]     ; x is a label for memory location (variable)
```

  - `add` is the instruction mnemonic
  - `eax` is the first operand. This operand acts as an input and output
  - `[x]` is the second operand. This operand acts as an input.
  - The semantic of this instruction is as follows. Add the content of `x` to `eax` and store the result into `eax` (i.e., `eax = eax + [x]`)
  - There is a natural ordering of operands. When instructions have multiple operands, the first one is typically called the **destination operand**. The second operand is usually called the **source operand**.

- Instruction operand can be:
  - Explicit, in which the operand appears in the instruction statement (see the example above)
  - Implicit, in which the operand does not appear in the statement. For example, consider the following integer multiplication instruction:

```
      imul    ebx
```

    The operand `ebx` represents the multiplier. The multiplicand is the register `EAX` and the product is stored in two registers: `EDX` and `EAX`. We are going to explain this instruction in much detail in future lectures.

- Generally, the instructions we are going to study in this course fall under general–purpose Instructions, which perform basic ***data movement***,

arithmetic, ***logic***, ***program flow***, and ***string operations*** that programmer commonly use to write application and system software. In this course we are going to focus on:

1. Data movement/transfer instructions,
2. Arithmetic and Logic instructions,
3. Control transfer instructions, and
4. Flag control instructions

2 Defining Data

2.1 Intrinsic Data Types

In assembly language, data types are expressed in terms of their *sizes* (byte, word, doubleword, etc).

Size	Type	Note
Byte	**B**YTE or HWORD	8 bits
Word (two-byte)	**W**ORD	16 bits
Double word (2-word)	**D**WORD	32 bits
Quad word (4-word)	**Q**WORD	64 bits
Ten-Byte (5-word)	**T**BYTE or TWORD	80 bits
Double-quad word (8-word)	**O**WORD or DQWORD	128 bits
Quad-quad word (16-word)	**Y**WORD	256 bits
32-Word	**Z**WORD	512 bits

2.2 Defining Initialized Data (using D*x*)

- A ***data definition with initialization*** sets aside storage in memory (in data segment) for a variable, with an optional label. The syntax:

Syntax

```
[name:]      directive       initializer [,
initializer]
```

- **name** is an identifier to label the allocated memory. The label is always attached to the first memory location.
- **directive** is a pseudo-instruction (not real x86 instruction and are used in the instruction field) that can be:

Table 4: Data Definition in the form of **D***x*

Directive	Description
DB	Define (i.e., Allocate) BYTE
DW	Define WORD
DD	Define DWORD
DQ	Define QWORD
DT	Define TBYTE
DO	Define OWORD
DY	Define YWORD
DZ	Define ZWORD

- **Initializer** is used to initialize the allocated memory. Additional initializers, if any, are separated by commas.
- Example

Examples of Data Definition

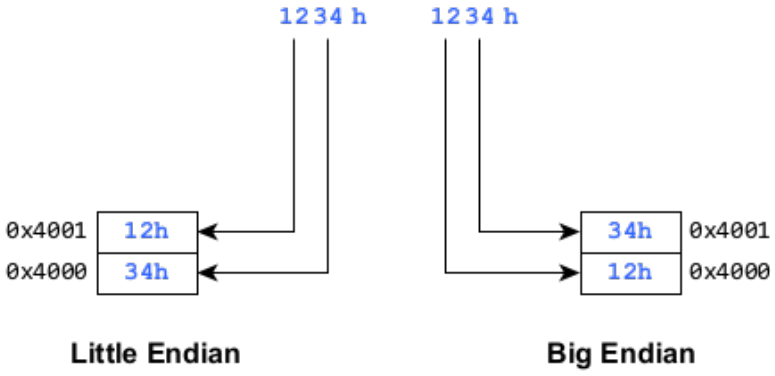
```
letter:    db     'A'    ; value is 41h
resp       db     'Y'    ; default user response
num1:      db     -91    ; value is A5h
num2       dw     -1     ; value is FFFFh
sum:       dd     0      ; value is 0h
```

2.2.1 Little Endian vs. Big Endian

Consider the following definition

```
num:      dw      1234h
```

- All x86 microprocessors are byte addressable. This means that each memory address can store only a single byte.
- Therefore, in the above definition, the assembler will allocate two bytes for `num`. But, the question that may arise: how a CPU can store the initial value into 2-byte memory?
- There are two ways to store `1234h` in memory:
  - **Little Endian**, in which the least significant byte is stored at the smallest memory address
  - **Big Endian**, in which the most significant byte is stored at the smallest memory address.
  - The following diagram illustrates the difference:



  - x86 microprocessors follow Little Endianness.

2.2.2 Multiple Initializers

- The pseudo-instruction **D***x* can take multiple initializers.
- If multiple initializers are used, the memory space is allocated large enough to hold these values.
- Remember, the label is always attached to the first memory location (i.e., first byte).
- Examples: In the comments, the first byte is located at the smallest memory address.

```
    db    55h, 56h, 57h    ; 0x55 0x56 0x57
    db    'a', 55h         ; 0x61 0x55
    db    'ab', 13, 10, 0  ; 0x61 0x62 0x0D 0x0A 0x00
    dw    1234h, 5678h     ; 0x34 0x12 0x78 0x56
    dw    12h, 34h         ; 0x12 0x00 0x34 0x00
    dw    'ab'             ; 0x61 0x62
    dw    'abc'            ; 0x61 0x62 0x63 0x00
    dd    0x12345678       ; 0x78 0x56 0x34 0x12
   ;db    300              ; invalid
```

- In data definitions, the labels are optional and pseudo-instructions are executed by the assembler in sequential. Therefore, the following definition:

```
list    db    10, 20, 30
```

  can be written as

```
list    db    10
        db    20
        db    30
```

2.2.3 DUP Operator

- The syntax of DUP operator:

Syntax

```
[name:]    Dx    <counter> DUP (initializer-list)
```

- The DUP operator duplicates the initializer-list specified number of times
- counter is an constant 32-bit integer expression.
- DUP operator is useful when allocating space for an array.
- Examples:

```
    db  3 dup(0)       ; db  0, 0, 0
    dw  8 dup(1)       ; dw  1, 1, 1, 1, 1, 1, 1, 1
    db  4 dup('ABC')   ; db  'ABC', 'ABC', 'ABC', 'ABC'
```

2.3 Defining Uninitialized Data (using RESx)

- The RESx directive is quite similar to the Dx directive, but always specifies the number of elements.
- The RESx directive defines uninitialized storage space inside the BSS segment.
- The syntax:

Syntax

```
[name:]      RESx       <counter>
```

  - counter is the number of x to be reserved. It must be a constant expression.

- Examples:

```
buffer:    resb    64     ; reserve 64 bytes
list:      resd    100    ; reserve 100 32-bit integers
```

2.4 Defining Constants

- NASM provides a directive, named `EQU`, to define a symbolic constant.
- The syntax is

Syntax

```
name    EQU   <expression>
```

- Examples:

```
ESC_KEY   equ   27
COUNT     equ   100
```

In the code section:

```
          mov   eax, COUNT     ; good style
          mov   eax, 100       ; bad style

          mov   al, ESC_KEY    ; good style
          mov   al, 27         ; bad style
```

2.5 Constant Integer Expression

- NASM assembler (and also MASM) supports what is called **constant integer expression** that *must* evaluate to a 32-bit integer.

Table 5: Assembler Operators

Operator	Name	Precedence
()	Parentheses	1
+, −, ~, !	Unary plus, negative, negation, logical not	2
*, /, //, %, %%	multiplication, unsigned division, signed division, unsigned modulus, signed modulus	3
+, −	Addition and subtraction	4
<<, >>	Shift to left or right (always unsigned)	5

Operator	Name	Precedence
&	Bitwise AND	6
^	Bitwise XOR	7
|	Bitwise OR	8

- Constant Integer Expression will always be evaluated by the assembler.

2.5.1 The Current Location ($) and Section Location ($$)

- One of the most important symbols of all, shown as $, is called the ***current location***.
- The assembler handles the current location and section location expressions as follows
  - When the assembler first encounters a section/segment statement, the current location counter ($) and the section location ($$) are set to the beginning of the current section.
  - When encountering instructions or pseudo-instructions (such as `Dx` or `RESx`), the assembler increments the location counter for each byte written to the object file (i.e, after generating the object code). So, you can tell how far into the section you are by using ($ - $$).
- In the following code, the current location expression is used to calculate the string length. The calculation is performed by the assembler.

```
myStr:    db    "This is a long string, containing "
          db    "any number of characters."
STRLEN    equ   ($ - myStr)
```

3 Addressing Modes

- We mentioned that a CPU instruction can have zero, one, two, or three operands.

  For exmaple:

```
        add   eax, 5
```

  The instruction `add` takes two operands. The first operand is a CPU register, and the second operand is a constant integer.

- In assembly language, the way an operand is specified is known as **addressing mode**.

- Generally, the addressing modes used by Intel architectures are:

  1. **Immediate addressing mode**: when an operand is a constant expression
  2. **Register addressing mode**: when an operand is a CPU register
  3. **Memory addressing modes**, in which we have
     - **Direct memory addressing mode**: when an operand is constant reference to memory location.
     - **Indirect memory addressing modes**: when an operand is reference to memory location and this reference is determined during the running-time.

  - The following table is taken from Intel manuals that describes the standard operand types:

Table 6: Instruction Operand Notation, 32-bit Mode.

Operand	Description
reg	Any general-purpose register
reg8	8-bit general-purpose register: AH, AL, BH, BL, CH, CL, DH, DL
reg16	16-bit general-purpose register: AX, BX, CX, DX, SI, DI, SP, BP
reg32	32-bit general-purpose register: EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP
sreg	16-bit segment register: CS, DS, SS, ES, FS, GS

Operand	Description
imm	8-, 16-, or 32-bit immediate value
imm8	8-bit immediate byte value
imm16	16-bit immediate word value
imm32	32-bit immediate doubleword value
mem	An 8-, 16-, or 32-bit memory operand
reg/mem8	8-bit operand, which can be an 8-bit general register or memory byte
reg/mem16	16-bit operand, which can be an 16-bit general register or memory word
reg/mem32	32-bit operand, which can be an 32-bit general register or memory doubleword

3.1 Direct Memory Addressing Mode

- A ***direct memory operand*** is an operand identifier that refers to a specific ***offset*** within the data segment.
- The offset address ***must*** be specified inside *square brackets* `[]`.

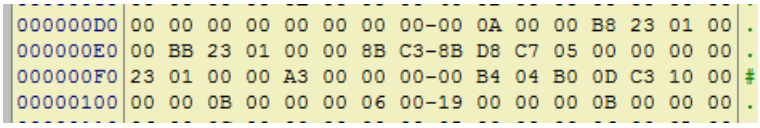
```
        section .bss
sum:    resd    1

        section .text
_main:
        mov     eax, 5
        add     eax, 6
        mov     [sum], eax
```

3.2 Addressing mode and Machine Instruction

Examine the following instruction statements. Assume the variable `x` is already defined.

```
mov eax, 123h          ; B8 23010000
mov   ebx, 123h        ; BB 23010000
mov   eax, ebx         ; 8BC3
mov   ebx, eax         ; 8BD8
mov   WORD [x], 123h   ; C705  00000000  23010000
mov   [x], eax         ; A3 00000000
mov   ah, 4            ; B4 04
mov   al, 13           ; B0 0D
```

If you read the object file using *Hexdump* tool, we will get:



4 Data Transfer Instructions

- Data transfer instructions are
  - `MOV`, `XCHG`, `MOVSX`, `MOVZX` : to move (as copy) data from source operand to destination operand.
  - `PUSH`, `POP`, `PUSHA`, `POPA`, `PUSHAD`, `POPAD`: to move data between an operand and run-time stack.
  - `CBW`, `CWD`, `CDQ`, `CWDE` : to sign-extend an operand
- In this lecture, we will cover `MOV`, `XCHG`, `MOVSX`, and `MOVZX` . The remaining instructions will be covered in future lectures.

4.1 MOV Instruction

Syntax

```
mov   <destination>, <source>
```

The destination operand's content changes, but the source operand is unchanged. Right to Left data transfer is similar to the assignment statement in C++ or Java:

```
destination = source;
```

Rules

- Both operands must be of the same size.
- Both operands cannot be memory operands.
- The instruction pointer (IP, EIP, RIP) cannot be a destination operand.

Accordingly, here is a list of valid MOV instruction formats:

```
mov   reg, reg
mov   reg, mem
mov   reg, imm
mov   mem, reg
mov   mem, imm    ; you must specify the size (see next warning)
```

⚠ NASM, by its design, does not store the type of variables you declared. For example:

```
var     dw    4      ; var is 16-bit variable
```

The following lines are all **valid**:

```
        mov   eax, [var]   ; transfer 4 bytes starting from offset var
        mov   ax, [var]    ; transfer 2 bytes starting from offset var
        mov   ah, [var]    ; transfer 1 byte from offset var
```

4.1.1 Example

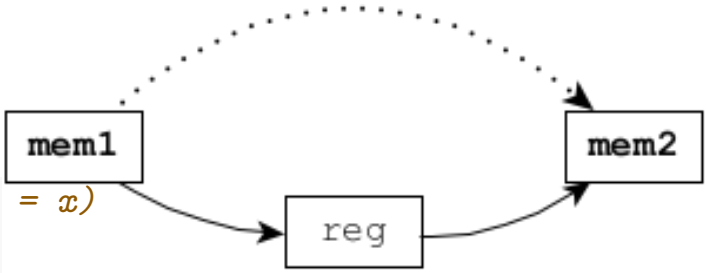
```
;
;  adding three numbers
;

        section .data
arr     dd      4, 28, 35

        section .text
_main:
        mov   eax, [arr]       ; move the value 4 to eax
        add   eax, [arr + 4]   ; add the value 28 to eax
        add   eax, [arr + 8]   ; add the value 35 to eax
```

ℹ Invalid MOV statements

```
        mov   4, eax       ; dest cannot be imm
        mov   [x], [y]     ; both operands cannot be memory
        mov   x, eax       ; dest cannot be imm
        mov   [x], 10      ; unknown data types (or size)
```

4.1.2 Memory to Memory

A single MOV instruction cannot be used to move data directly from one memory location to another.




```
; assign x to y (i.e., y = x)
        mov     eax, [x]
        mov     [y], eax
```

4.1.3 Immediate to Memory

In assembly language, the size of integer literals can be 8, 16, or 32 bits. Therefore, in the following instruction:

```
      mov   ax, 5
```

the size of the source operand is unknown. However, the assembler can infer the size of immediate operand by the size of other operand (`ax`). Thus, the assembler translates the source operand as `imm16`.

The problem arises when the size of the other operand is also unknown; typically when we move data from `imm` to `mem`. In this case, you must tell the assembler the amount of bytes to be transferred from the source to destination by specifying the data type of at least one operand (see Intrinsic Data Types).

Example

```
        mov   DWORD [x], 5
```

Here, the size of the first operand is doubleword. The size of the second operand will be inferred from the first one.

4.2 MOVSX Instruction

- Can we transfer data from a smaller size to a larger size?
- The answer is: Yes. But, you must be cautious. Consider the following example; let us assume that we need to transfer data from mem16 to reg32 (ECX).

```
        section .data
v1      dw      -16         ; FFF0h
v2      dw      16          ; 0010h

        section .text

        mov     ecx, [v1] ; WRONG. ecx=0010 FFF0h

        mov     ecx, 0    ; 0000 0000h
        mov     cx, [v1]  ; 0000 FFF0h

        mov     ecx, 0FFFFFFFFh
        mov     cx, [v2]; FFFF 0010h
```

- MOVSX instruction (move with sign-extension) copies the content of a source operand into a destination operand with sign-extends to 16 or 32 bits.
- This instruction is only used with signed integers and there are three variants:

```
        movsx     reg32, reg/mem8
        movsx     reg32, reg/mem16
        movsx     reg16, reg/mem8
```

- The following code corrects the above code:

```
        movsx     ecx, WORD [v1]  ; ecx = FFFF FFF0h = -16

        movsx     edx, WORD [v2]  ; edx = 0000 0010h = 16
```

4.3 MOVZX Instruction

- MOVZX is similar to MOVSX, but with zero-extend. It is only used with *unsigned* integers

4.4 XCHG Instruction

Syntax

```
XCHG    <operand1> , <operand2>
```

The XCHG (exchange data) instruction exchanges the contents of two operands.

Rules

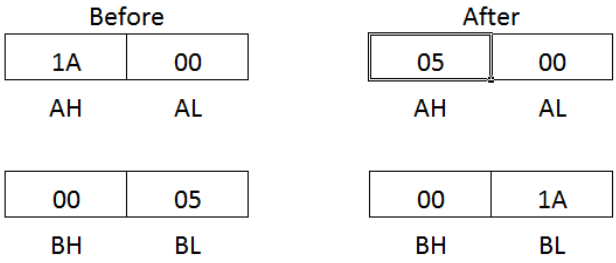
- Both operands must be of the same size.
- Both operands cannot be memory operands.
- Immediate operands are not allowed.
- The instruction pointer (IP, EIP, RIP) cannot be a destination operand.

Accordingly, this instruction has three formats:

```
XCHG    reg, reg
XCHG    reg, mem
XCHG    mem, reg
```

Example: What is the value of AX and BX after executing the following code:

```
mov     ax, 0A100h
mov     bx, 0005h
xchg    ah, bl
```


Before
1A
00
AH
AL
00
05
BH
BL
After
05
00
AH
AL
00
1A
BH
BL