Topic 3
Lecture 3
Assembly Language Fundamentals

CSCI 150

Assembly Language / Machine Architecture

Prof. Dominick Atanasio

Chapter Overview

- Basic Elements of Assembly Language
- Example: Adding and Subtracting Integers
- Assembling, Linking, and Running Programs
- Defining Data
- Symbolic Constants

Basic Elements of Assembly Language

- Integer constants
- Integer expressions
- Character and string constants
- Reserved words and identifiers
- Directives and instructions
- Labels
- Mnemonics and Operands
- Comments
- Examples

Integer Constants

- Optional leading + or – sign
- binary, decimal, hexadecimal, or octal digits
- Common radix characters:
  - h – hexadecimal
  - d – decimal
  - b – binary
  - r – encoded real

Examples: 30d, 6Ah, 42, 1101b

Hexadecimal beginning with letter: 0A5h

Integer Constants

- We will be using the Netwide Assembler (NASM) which allows you to specify numbers in a variety of number bases, in a variety of ways:
- You can suffix with:
  - H or X for hex
  - D or T for decimal
  - Q or O for octal
  - B or Y for binary

Integer Constants

Some examples (all producing exactly the same code):

```
mov     ax,200          ; decimal
mov     ax,0200         ; still decimal
mov     ax,0200d        ; explicitly decimal
mov     ax,0d200        ; also decimal
mov     ax,0c8h         ; hex
mov     ax,$0c8         ; hex again: the 0 is required
mov     ax,0xc8         ; hex yet again
mov     ax,0hc8         ; still hex
mov     ax,310q         ; octal
mov     ax,310o         ; octal again
mov     ax,0o310        ; octal yet again
mov     ax,0q310        ; octal yet again
mov     ax,11001000b    ; binary
mov     ax,1100_1000b   ; same binary constant
mov     ax,1100_1000y   ; same binary constant once more
mov     ax,0b1100_1000  ; same binary constant yet again
mov     ax,0y1100_1000  ; same binary constant yet again
```

Constant Integer Expressions

- Operators and precedence levels:

Operator	Name	Precedence Level
( )	parentheses	1
+,-	unary plus, minus	2
*,/	multiply, divide	3
MOD	modulus	3
+,-	add, subtract	4

- Examples:

Expression	Value
16 / 5	3
-(3 + 4) * (6 - 1)	-35
-3 + 4 * 6 - 1	20
25 mod 3	1

Character and String Constants

- Enclose character in single or double quotes
  - 'A', "x"
  - ASCII character = 1 byte
- Enclose strings in single or double quotes
  - "ABC"
  - 'xyz'
  - Each character occupies a single byte
- Embedded quotes:
  - 'Say "Goodnight," Gracie'

Character and String Constants

A string constant looks like a character constant, only longer. It is treated as a concatenation of maximum-size character constants for the conditions. So the following are equivalent:

```
      db    'hello'              ; string constant
      db    'h','e','l','l','o'  ; equivalent character constants
```

And the following are also equivalent:

```
      dd    'ninechars'          ; doubleword string constant
      dd    'nine','char','s'    ; becomes three doublewords
      db    'ninechars',0,0,0    ; and really looks like this
```

Reserved Words and Identifiers

- Reserved words cannot be used as identifiers
  - Instruction mnemonics, directives, type attributes, operators, predefined symbols
  - See NASM reference
- Identifiers
  - Valid characters in labels are letters, numbers, _, $, #, @, ~, ., and ?
  - Case sensitive
  - First character must be a letter [a-z], _, ., or ?
  - An identifier may also be prefixed with a $ to indicate that it is intended to be read as an identifier and not a reserved word

Reserved Words and Identifiers

- NASM gives special treatment to symbols beginning with a period. A label beginning with a single period is treated as a local label, which means that it is associated with the previous non-local label.
  - So, for example:

```
label1  ; some code

.loop
        ; some more code

        jne     .loop
        ret

label2  ; some code

.loop
        ; some more code

        jne     .loop
        ret
```

Directives

- Commands that are recognized and acted upon by the assembler
  - Not part of the Intel instruction set
  - Used to declare code areas, data areas, select memory model, declare procedures, etc.
  - not case sensitive
- Different assemblers have different directives
  - NASM is not the same as MASM, for example

Instructions

- Assembled into machine code by assembler
- Executed at runtime by the CPU
- We use the Intel IA-32 instruction set
- An instruction contains:
  - Label (optional)
  - Mnemonic (required)
  - Operand(s) (depends on the instruction)
  - Comment (optional)

Labels

- Act as place markers
  - marks the address (offset) of code and data
  - Are case-sensitive
- Follow identifer rules
- Data label
  - must be unique
  - example: **myArray:** (optionally, but recommended to be followed by colon)
- Code label
  - target of jump and loop instructions
  - example: **L1:** (optionally, but recommended to be followed by colon)

Mnemonics and Operands

- Instruction Mnemonics
  - A mnemonic is a memory aid
  - examples: MOV, ADD, SUB, MUL, INC, DEC
  - These are not case-sensitive
- Operands
  - constant
  - constant expression
  - register
  - memory (data label)

Constants and constant expressions are called immediate values

Comments

- Comments are good!
  - explain the program's purpose
  - when it was written, and by whom
  - revision information
  - tricky coding techniques
  - application-specific explanations
- Single-line comments
  - begin with semicolon (;)

Instruction Format Examples

- No operands
  - stc ; set Carry flag
- One operand
  - inc eax ; register
  - inc BYTE [myByte] ; memory
- Two operands
  - add ebx, ecx ; register, register
  - sub BYTE [myByte], 25 ; memory, constant
  - add eax, 36 % 25 ; register, constant-expression

What's Next (1 of 5)

- Basic Elements of Assembly Language
- **Example: Adding and Subtracting Integers**
- Assembling, Linking, and Running Programs
- Defining Data
- Symbolic Constants
- 64-Bit Programming

Example: Adding and Subtracting Integers

```
;;; add two immediate values

global _start

section  .text
_start:
      mov eax, 25 ; move the integer value of 20 into eax register
      add eax, 5  ; add to the eax register the integer value 5
                  ; (eax += 5)
_exit:
      mov eax, 1  ; syscall code for exit
      mov ebx, 0  ; exit code (0 means no error)
      int 0x80    ; perform syscall
```

Suggested Coding Standards (1 of 2)

- Some approaches to capitalization
  - capitalize nothing
  - capitalize everything
  - capitalize all reserved words, including instruction mnemonics and register names
  - capitalize only directives and operators
- Other suggestions
  - descriptive identifier names
  - spaces surrounding arithmetic operators
  - blank lines between procedures

Suggested Coding Standards (2 of 2)

- Indentation and spacing
  - code and data labels – no indentation
  - executable instructions – indent 4-5 spaces
  - comments: right side of page, aligned vertically
  - 1-3 spaces between instruction and its operands
    - ex: mov ax, bx
  - 1-2 blank lines between procedures

Program Template

```
; who: <your name and Mt SAC username goes here>
; what: <the function of this program>
; why: <the name of the lab>
; when: < the due date of this lab.

global      _start

section     .text

_start:



exit:
    mov     ebx, 0       ; return 0 status on exit - 'No Errors'
    mov     eax, 1       ; invoke SYS_EXIT (kernel opcode 1)
    int     80h          ; perform syscall


section     .bss


section     .data
```

What's Next (2 of 5)

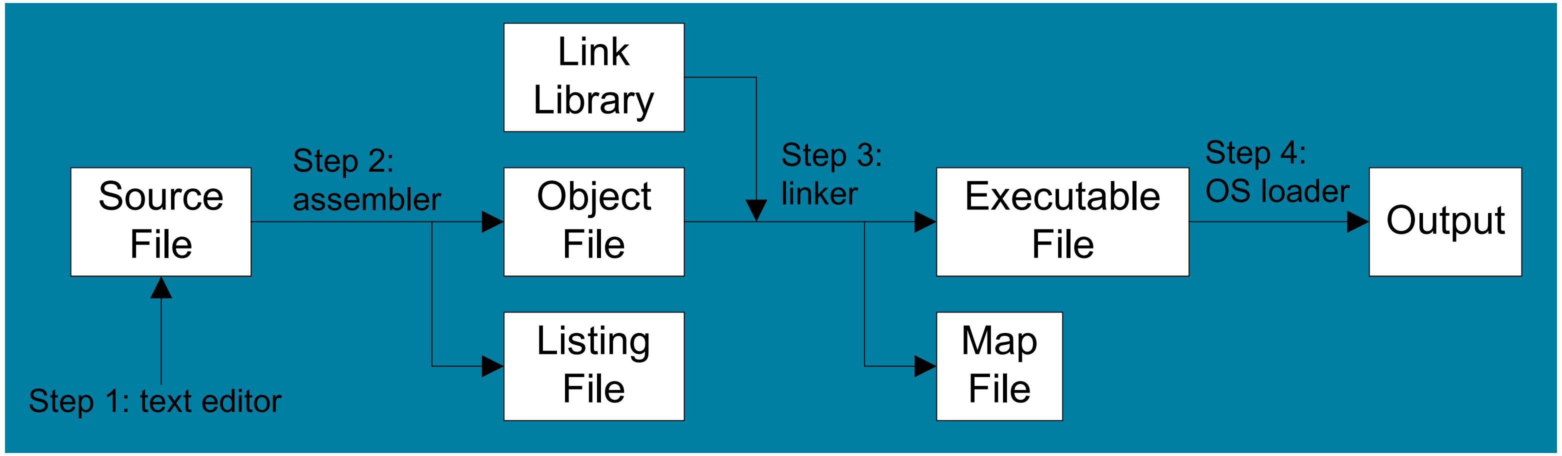
- Basic Elements of Assembly Language
- Example: Adding and Subtracting Integers
- **Assembling, Linking, and Running Programs**
- Defining Data
- Symbolic Constants
- 64-Bit Programming

Assembling, Linking, and Running Programs

- Assemble-Link-Execute Cycle
- Listing File
- Map File

Assemble-Link Execute Cycle

- The following diagram describes the steps from creating a source program through executing the compiled program.
- If the source code is modified, Steps 2 through 4 must be repeated.
- Our assembler is called, NASM, which stands for, “Netwide Assembler”



Listing File

- Use it to see how your program is compiled
- Contains
  - source code
  - addresses
  - object code (machine language)
  - segment names
  - symbols (variables, procedures, and constants)

What's Next (3 of 5)

- Basic Elements of Assembly Language
- Example: Adding and Subtracting Integers
- Assembling, Linking, and Running Programs
- **Defining Data**
- Symbolic Constants
- 64-Bit Programming

Defining Data (1 of 2)

- Intrinsic Data Types (actually, data widths)
- Data Definition Statement (in data or bss sections)
- db - data byte (8 bits)
- dw - data word (2 bytes or 16 bits)
- dd - data double (2 words or 32 bits)
- dq - data quad (2 double words or 64 bits)
- dt - data ten (10 bytes or 80 bits)

Defining Data (2 of 2)

- Defining Real Number Data
- Little Endian Order
- Adding Variables to the AddSub Program
- Declaring Uninitialized Data

Data Definition Statement

- A data definition statement sets aside storage in memory for a variable.
- May optionally assign a name (label) to the data
- Syntax:

  [*label*]: *directive initializer* [,*initializer*] . . .

Example:

```
value1 db 10
```

- All initializers become binary data in memory

Defining db and db Data

Each of the following defines a single byte of storage:

value1: db 'A' ; character constant
value2: db 0 ; smallest unsigned byte
value3: db 255 ; largest unsigned byte
value4: db 128; smallest signed byte
value5: db 127 ; largest signed byte

- NASM does not prevent you from initializing a db with a negative value, but it's considered poor style.

Defining Byte Arrays

Examples that use multiple initializers:

```
list1: db 10,20,30,40
list2: db 10,20,30,40
       db 50, 60, 70, 80
       db 81,82,83,84
list3: db 44, 32, 41h, 00100010b
list4: db 0Ah, 20h, ‘A’, 22h
```

Defining Strings (1 of 3)

- A string is implemented as an array of characters
  - For convenience, it is usually enclosed in quotation marks
  - It often will be null-terminated
- Examples:

```
str1: db "Enter your name", 0
str2: db 'Error: halting program', 0
str3: db 'A','E','I','O','U'
greeting: db "Welcome to the Webly Game Demo program “
          db "created by Prof.A.",0
```

Defining Strings (2 of 3)

- To continue a single string across multiple lines, end each line with a comma:

```
menu: db  "Checking Account", 0ah, 0ah,
          "1. Create a new account",0ah,
          "2. Open an existing account", 0ah,
          "3. Credit the account", 0ah,
          "4. Debit the account", 0ah,
          "5. Exit",0ah,0ah,
          "Choice> ",0
```

Defining Strings (3 of 3)

- End-of-line character :
  - 0Ah = line feed (dec value 10)

```
str1: db "Enter your name: “, 0x0A
      db "Enter your address: ", 0
newline:  db 0x0A, 0
```

Idea: Define all strings used by your program in the same area of the data segment.

Using the DUP Operator (only supported in NASM >= ver. 2.15)

- Use DUP to allocate an array or string.
  Syntax: *counter* DUP (*argument*)
- *Counter* and *argument* must be constants or constant expressions

```
var1: TIMES 20 db 0         ; 20 bytes, all equal to zero
var2: TIMES 20 resb 1       ; 20 bytes, uninitialized
var3: TIMES 4 db "STACK“    ; 20 bytes:  "STACKSTACKSTACKSTACK“
```

- Applied to an instruction

```
TIMES 100 inc eax
```

Using the TIMES Operator

- Use TIMES to repeat instructions or data :
  - *TIMES counter* type default_value
- *Counter* and *argument* must be constants or constant expressions

var1: TIMES 20 db 0   ; 20 bytes, all equal to zero (in data section)

var2: TIMES 20 resb 1 ; 20 bytes, uninitialized (in bss section) (same as var2: resb 20)

var3: TIMES 4 db "STACK" ; 20 bytes: "STACKSTACKSTACKSTACK"

Defining a Word (dw)

- Define storage for 16-bit integers (words)
  - or double characters
  - single value or multiple values

```
word1:   dw    65535         ; largest unsigned value (in data section)
word2:   dw   –32768         ; smallest signed value (in data section)
word3:   resw   1            ; uninitialized, unsigned (in bss section)
word4:   dw    "AB"          ; double characters (in data section)
myList:  dw    1, 2, 3, 4, 5 ; array of words (in data section)
array:   resw  5             ; uninitialized array of 5 words (in bss section)
```

Defining Double-word Data

Storage definitions for signed and unsigned 32-bit integers:

In data section

```
val1:   dd   12345678h        ; unsigned
val2:   dd   –2147483648      ; signed
val3:   dd   –3,–2,–1, 0, 1   ; signed array
```

In bss section

```
val4:   resd   20             ; unsigned array
```

Defining Quad-word, Ten-byte

Storage definitions for quadwords, tenbyte values, and real numbers:

```
quad1:   dq   0x1234567812345678
bigval:  dt   0x1000000000123456789A
```

Little Endian Order

- All data types larger than a byte store their individual bytes in reverse order. The least significant byte is stored in the first (lowest) memory address.
- Example:

val1:   dd 0x12345678

Address	Byte
0003	0x12
0002	0x34
0001	0x56
0000	0x78

Adding Variables to AddSub

```
; who: Prof.A
; what: add and subtract values
; why:  topic 3 demo
; when: 2022-03-15 class

section     .text

global      _start

_start:
    mov     eax, [val1]
    add     eax, [val2]
    sub     eax, [val3]
    mov     [final], eax

exit:
    mov     ebx, 0       ; return 0 status on exit - 'No Errors'
    mov     eax, 1       ; invoke SYS_EXIT (kernel opcode 1)
    int     80h

section     .bss
    final   resd 1

section     .data
    val1    dd 1000d
    val2    dd 4000d
    val3    dd 2000d
```

Declaring Unitialized Data

- Use the .bss section to declare a unintialized data
- Within the segment, declare variables with res*x* where *x* is replaced with the type:

```
final:    resd 1   ; reserve 4 bytes (double-word)
```

Advantage: the program's executable file size is reduced.

Chapter Continuation

- Basic Elements of Assembly Language
- Example: Adding and Subtracting Integers
- Assembling, Linking, and Running Programs
- Defining Data
- **Symbolic Constants**

Symbolic Constants

- Calculating the Sizes of Arrays and Strings
- EQU Directive

Calculating the Size of a Byte Array

- current location counter: $
  - subtract address of list
  - difference is the number of bytes

```
list: db 10,20,30,40            ; list of bytes
listSize: equ ($ - list)        ; size in bytes
```

Calculating the Size of a Word Array

Divide total number of bytes by 2 (the size of a word)

```
list: dw 1000h, 2000h, 3000h, 4000h       ; list of words
ListSize: equ ($ - list) / 2              ; size in words
```

Calculating the Size of a Doubleword Array

Divide total number of bytes by 4 (the size of a double-word)

```
list: dd 1, 2, 3, 4                 ; List of double-words
ListSize:  equ  ($ - list) / 4      ; Size in double-words
```

EQU Directive

- Gives a symbolic name to a numeric **constant**
- A pseudo instruction
- Defines a symbol as an integer
- This is a constant that has no memory location
  - Analogous to #define preprocessor directive
- Cannot be redefined
- Must be an integer value or an expression that evaluates to an integer

```
qty:  equ  10
```

4C 61 46 69 6E