Assembly Fundamentals

Computer Organization and Assembly Languages

Yung-Yu Chuang

2005/10/13

with slides by Kip Irvine

Announcements

- Homework#1 assigned, due on 10/27
- Next week's class (10/20) will be taught by TAs
- Midterm examination will be held on the week of 11/10

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

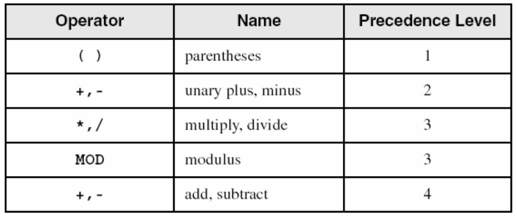
- [{+|-}] *digits* [*radix*]
- Optional leading + or – sign
- binary, decimal, hexadecimal, or octal digits
- Common radix characters:
  - **h** – hexadecimal
  - **d** – decimal (default)
  - **b** – binary
  - **r** – encoded real
  - **o** – octal

Examples: **`30d, 6Ah, 42, 42o, 1101b`**

Hexadecimal beginning with letter: **`0A5h`**

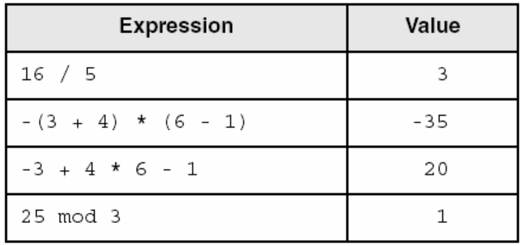
Integer expressions

- 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

Real number constants (encoded reals)

- Fixed point v.s. floating point

1	8	23
S	E	M

$\pm 1.bbbb \times 2^{(E-127)}$

- Example **`3F800000r=+1.0,37.75=42170000r`**

- double

1	11	52
S	E	M

Real number constants (decimal reals)

- [*sign*]*integer*.[*integer*][*exponent*]

  sign → {+|-}

  exponent → E[{+|-}]integer
- Examples:

`2.`

`+3.0`

`-44.2E+05`

`26.E5`

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’`**
  - **`"This isn't a test"`**

Reserved words and identifiers

- Reserved words (Appendix D) cannot be used as identifiers
  - Instruction mnemonics, directives, type attributes, operators, predefined symbols
- Identifiers
  - 1-247 characters, including digits
  - case insensitive (by default)
  - first character must be a letter, **`_`**, **`@`**, or **`$`**
  - examples:

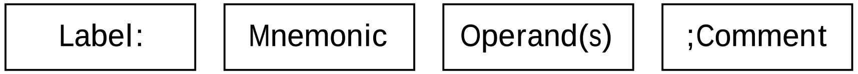
```
var1      Count     $first
_main     MAX       open_file
@@myfile  xVal      _12345
```

Directives

- Commands that are recognized and acted upon by the assembler
  - Part of assembler’s syntax but not part of the Intel instruction set
  - Used to declare code, data areas, select memory model, declare procedures, etc.
  - case insensitive
- Different assemblers have different directives
  - NASM != MASM, for example
- Examples: **`.data`** **`.code`** **`PROC`**

Instructions

- Assembled into machine code by assembler
- Executed at runtime by the CPU
- Member of the Intel IA-32 instruction set
- Four parts
  - Label (optional)
  - Mnemonic (required)
  - Operand (usually required)
  - Comment (optional)



Labels

- Act as place markers
  - marks the address (offset) of code and data
- Easier to memorize and more flexible

  `mov ax, [0020]` → **`mov ax, val`**
- Follow identifier rules
- Data label
  - must be unique
  - example: **`myArray BYTE 10`**
- Code label
  - target of jump and loop instructions
  - example:

```
L1: mov ax, bx
    ...
    jmp L1
```

Mnemonics and operands

- Instruction mnemonics
  - "reminder"
  - examples: **MOV, ADD, SUB, MUL, INC, DEC**
- Operands
  - constant (immediate value), **96**
  - constant expression, **2+4**
  - Register, **eax**
  - memory (data label), **count**
- Number of operands: 0 to 3

```
– stc              ; set Carry flag
– inc ax           ; add 1 to ax
– mov count, bx    ; move BX to count
```

Comments

- Comments are good!
  - explain the program's purpose
  - tricky coding techniques
  - application-specific explanations
- Single-line comments
  - begin with semicolon (;)
- block comments
  - begin with COMMENT directive and a programmer-chosen character and end with the same programmer-chosen character

```
COMMENT !
   This is a comment
   and this line is also a comment
!
```

Example: adding/subtracting integers

directive marks comment

```
TITLE Add and Subtract              (AddSub.asm)
; This program adds and subtracts 32-bit integers.

INCLUDE Irvine32.inc
.code
main PROC
    mov eax,10000h          ; EAX = 10000h
    add eax,40000h          ; EAX = 50000h
    sub eax,20000h          ; EAX = 30000h
    call DumpRegs           ; display registers
    exit
main ENDP
END main
```

comment

copy definitions from Irvine32.inc

code segment. 3 segments: code, data, stack

beginning of a procedure

source

destination

defined in Irvine32.inc to end a program

mark the last line and startup procedure

Example output

Program output, showing registers and flags:

```
EAX=00030000  EBX=7FFDF000  ECX=00000101  EDX=FFFFFFFF
ESI=00000000  EDI=00000000  EBP=0012FFF0  ESP=0012FFC4
EIP=00401024  EFL=00000206  CF=0  SF=0  ZF=0  OF=0
```

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 (used by the book)
- 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: begin at column 40-45, aligned vertically
  - 1-3 spaces between instruction and its operands
    - ex: **`mov  ax,bx`**
  - 1-2 blank lines between procedures

Alternative version of AddSub

```
TITLE Add and Subtract              (AddSubAlt.asm)

; This program adds and subtracts 32-bit integers.
.386
.MODEL flat,stdcall
.STACK 4096

ExitProcess PROTO, dwExitCode:DWORD
DumpRegs PROTO

.code
main PROC
   mov eax,10000h             ; EAX = 10000h
   add eax,40000h             ; EAX = 50000h
   sub eax,20000h             ; EAX = 30000h
   call DumpRegs
   INVOKE ExitProcess,0
main ENDP
END main
```

Program template

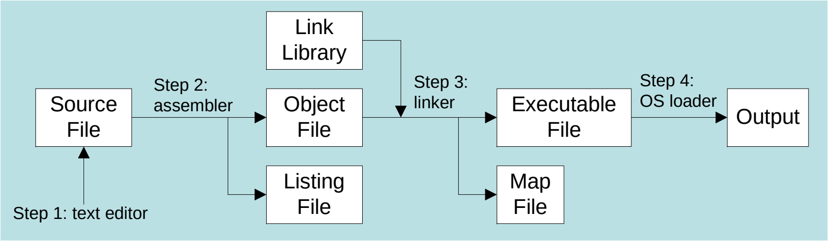
```
TITLE Program Template           (Template.asm)

; Program Description:
; Author:
; Creation Date:
; Revisions:
; Date:              Modified by:

INCLUDE Irvine32.inc
.data
    ; (insert variables here)
.code
main PROC
    ; (insert executable instructions here)
    exit
main ENDP
    ; (insert additional procedures here)
END main
```

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.



make32.bat

- Called a batch file
- Run it to assemble and link programs
- Contains a command that executes ML.EXE (the Microsoft Assembler)
- Contains a command that executes LINK32.EXE (the 32-bit Microsoft Linker)
- Command-Line syntax:

  make32 *progName*

  (*progName* includes the .asm extension)

(use make16.bat to assemble and link Real-mode programs)

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)
- Example: addSub.lst

Defining data

- Intrinsic data types
- Data Definition Statement
- Defining BYTE and SBYTE Data
- Defining WORD and SWORD Data
- Defining DWORD and SDWORD Data
- Defining QWORD Data
- Defining TBYTE Data
- Defining Real Number Data
- Little Endian Order
- Adding Variables to the AddSub Program
- Declaring Uninitialized Data

Intrinsic data types (1 of 2)

- **BYTE**, **SBYTE**
  - 8-bit unsigned integer; 8-bit signed integer
- **WORD**, **SWORD**
  - 16-bit unsigned & signed integer
- **DWORD**, **SDWORD**
  - 32-bit unsigned & signed integer
- **QWORD**
  - 64-bit integer
- **TBYTE**
  - 80-bit integer

Intrinsic data types (2 of 2)

- **REAL4**
  - 4-byte IEEE short real
- **REAL8**
  - 8-byte IEEE long real
- **REAL10**
  - 10-byte IEEE extended real

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:

  [*name*] *directive initializer* [,*initializer*] . . .

  At least one initializer is required, can be **?**
- All initializers become binary data in memory

Defining BYTE and SBYTE Data

Each of the following defines a single byte of storage:

```
value1 BYTE 'A'            ; character constant
value2 BYTE 0              ; smallest unsigned byte
value3 BYTE 255            ; largest unsigned byte
value4 SBYTE -128          ; smallest signed byte
value5 SBYTE +127          ; largest signed byte
value6 BYTE ?              ; uninitialized byte
```

A variable name is a data label that implies an offset (an address).

Defining multiple bytes

Examples that use multiple initializers:

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

Defining strings (1 of 2)

- A string is implemented as an array of characters
  - For convenience, it is usually enclosed in quotation marks
  - It usually has a null byte at the end
- Examples:

```
str1 BYTE "Enter your name",0
str2 BYTE 'Error: halting program',0
str3 BYTE 'A','E','I','O','U'
greeting1 BYTE "Welcome to the Encryption Demo program "
          BYTE "created by Kip Irvine.",0
greeting2 \
   BYTE "Welcome to the Encryption Demo program "
   BYTE "created by Kip Irvine.",0
```

Defining strings (2 of 2)

- End-of-line character sequence:
  - 0Dh = carriage return
  - 0Ah = line feed

```
str1 BYTE "Enter your name:    ",0Dh,0Ah
     BYTE "Enter your address: ",0

newLine BYTE 0Dh,0Ah,0
```

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

Using the DUP operator

- Use **DUP** to allocate (create space for) an array or string.
- Counter and argument must be constants or constant expressions

```
var1 BYTE 20 DUP(0)           ; 20 bytes, all equal to zero
var2 BYTE 20 DUP(?)           ; 20 bytes, uninitialized
var3 BYTE 4 DUP("STACK")      ; 20 bytes: "STACKSTACKSTACKSTACK"
var4 BYTE 10,3 DUP(0),20
```

Defining WORD and SWORD data

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

```
word1 WORD   65535         ; largest unsigned value
word2 SWORD –32768         ; smallest signed value
word3 WORD   ?             ; uninitialized, unsigned
word4 WORD "AB"            ; double characters
myList WORD 1,2,3,4,5      ; array of words
array WORD 5 DUP(?)        ; uninitialized array
```

Defining DWORD and SDWORD data

Storage definitions for signed and unsigned 32-bit integers:

```
val1 DWORD 12345678h          ; unsigned
val2 SDWORD –2147483648       ; signed
val3 DWORD 20 DUP(?)          ; unsigned array
val4 SDWORD –3,–2,–1,0,1      ; signed array
```

Defining QWORD, TBYTE, Real Data

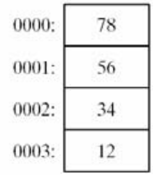
Storage definitions for quadwords, tenbyte values, and real numbers:

```
quad1 QWORD  1234567812345678h
val1  TBYTE  1000000000123456789Ah
rVal1 REAL4  -2.1
rVal2 REAL8  3.2E-260
rVal3 REAL10 4.6E+4096
ShortArray REAL4 20 DUP(0.0)
```

Little Endian order

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

```
val1 DWORD 12345678h
```



0000:	78
0001:	56
0002:	34
0003:	12

Adding variables to AddSub

```
TITLE Add and Subtract, Version 2            (AddSub2.asm)
; This program adds and subtracts 32-bit unsigned
; integers and stores the sum in a variable.
INCLUDE Irvine32.inc
.data
val1 DWORD 10000h
val2 DWORD 40000h
val3 DWORD 20000h
finalVal DWORD ?
.code
main PROC
    mov eax,val1                 ; start with 10000h
    add eax,val2                 ; add 40000h
    sub eax,val3                 ; subtract 20000h
    mov finalVal,eax             ; store the result (30000h)
    call DumpRegs                ; display the registers
    exit
main ENDP
END main
```

Declaring unitialized data

- Use the **.data?** directive to declare an unintialized data segment:

  ```
  .data?
  ```
- Within the segment, declare variables with "?" initializers:

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

```
.data
smallArray DWORD 10 DUP(0)
.data?
bigArray   DWORD 5000 DUP(?)
```

Mixing code and data

```
.code
mov eax, ebx
.data
temp DWORD ?
.code
mov temp, eax
```

Symbolic constants

- Equal-Sign Directive
- Calculating the Sizes of Arrays and Strings
- EQU Directive
- TEXTEQU Directive

Equal-sign directive

- *name* = *expression*
  - expression is a 32-bit integer (expression or constant)
  - may be redefined
  - *name* is called a symbolic constant
- good programming style to use symbols
  - Easier to modify
  - Easier to understand, **ESC_key**
  - **Array DWORD COUNT DUP(0)**
  - **COUNT=5**

```
Mov al, COUNT
COUNT=10
Mov al, COUNT
```

```
COUNT = 500
.
mov al,COUNT
```

Calculating the size of a byte array

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

```
list BYTE 10,20,30,40
ListSize = 4
```

```
list BYTE 10,20,30,40
ListSize = ($ - list)
```

```
list BYTE 10,20,30,40
Var2 BYTE 20 DUP(?)
ListSize = ($ - list)
```

```
myString BYTE “This is a long string.”
myString_len = ($ - myString)
```

Calculating the size of a word array

- current location counter: $
  - subtract address of list
  - difference is the number of bytes
  - divide by 2 (the size of a word)

```
list WORD 1000h,2000h,3000h,4000h
ListSize = ($ - list) / 2
```

```
list DWORD 1,2,3,4
ListSize = ($ - list) / 4
```

EQU directive

- name EQU expression

  name EQU symbol

  name EQU <text>
- Define a symbol as either an integer or text expression.
- Can be useful for non-integer constant
- Cannot be redefined

EQU directive

```
PI EQU <3.1416>
pressKey EQU <"Press any key to continue...",0>
.data
prompt BYTE pressKey
```

```
Matrix1 EQU 10*10
matrix1 EQU <10*10>
.data
M1 WORD matrix1            ; M1 WORD 100
M2 WORD matrix2            ; M2 WORD 10*10
```

TEXTEQU directive

- name TEXTEQU <text>

  name TEXTEQU textmacro

  name TEXTEQU %constExpr
- Define a symbol as either an integer or text expression.
- Called a text macro
- Can be redefined

```
continueMsg TEXTEQU <"Do you wish to continue (Y/N)?">
rowSize = 5
.data
prompt1 BYTE continueMsg
count TEXTEQU %(rowSize * 2)      ; evaluates the expression
move TEXTEQU <mov>
setupAL TEXTEQU <move al,count>
.code
setupAL                           ; generates: "mov al,10"
```

Chapter recap

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