Assembly Language Basics

TEXT 2 (CHAPTER 3)

Visit: http://kipirvine.com/asm/gettingStartedVS2012/index.htm
for instructions on how to set up the programming environment.

- Assembly language is not that difficult to learn.
- Writing programs that are useful however requires a comparatively large amount of code.

CODE: OUR FIRST PROGRAM BLOCK

```
main PROC
    mov eax,5          ; move 5 to the EAX register
    add eax,6          ; add 6 to the EAX register
    call WriteInt      ; display value in EAX
    Exit               ; quit
main ENDP
```

Integers

- An *integer constant* (or integer literal) is made up of an optional leading sign, one or more digits, and an optional suffix character.
- The Syntax: [{+|−}] *digits* [*radix*]
- The radix may be one of the following:

h Hexadecimal	r Encoded real
q/o Octal	t Decimal *(alternate)*
d Decimal	y Binary (*alternate*)
b Binary	

- If no radix is given, the integer constant is assumed to be decimal. *Examples*?

- An *integer expression* is a mathematical expression involving integer values and arithmetic operators.
- The expression must evaluate to an integer, which can be stored in 32 bits (0 through FFFFFFFFh).

Operator	Name	Precedence Level
( )	Parentheses	1
+, −	Unary plus, minus	2
*, /	Multiply, divide	3
MOD	Modulus	3
+, −	Add, subtract	4

- What does the following evaluate to: $18/2 * 2/3/2 * 2 * 2 - 6 + 2 + 4/2 * 2$?

Identifiers

- An *identifier* is a programmer-chosen name. It might identify a variable, a constant, a procedure, or a code label.
- RULES:
  - They may contain between 1 and 247 characters.
    They are not case sensitive.
  - The first character must be a letter (A..Z, a..z), underscore (_), @ , ?, or $.
    Subsequent characters may also be digits.
  - An identifier cannot be the same as an assembler reserved word.

EXAMPLE: IDENTIFIERS

?hello	how	34543	?myCRopee
$money	_9999	open_file	var9

Directives

- A *directive* is a command embedded in the source code that is recognized and acted upon by the assembler.
- Directives do not execute at runtime.
- We will use directives for two main purposes
  - To define variables and procedures.

    Example: myVar DWORD 26   ; DWORD directive

  - To define program sections or segments

    i.e. .data,   .code,   .stack *size*

Instructions

- An *instruction* is a statement that becomes executable when a program is assembled.
- Instructions are translated by the assembler into machine language bytes, which are loaded and executed by the CPU at runtime.
- An instruction contains four basic parts:
  - Label (optional)
  - Instruction mnemonic (required)
  - Operand(s) (usually required)
  - Comment (optional)

CODE: SYNTAX

[*label*:] *mnemonic* [*operands*] [;*comment*]

Labels

- A *label* is an identifier that acts as a place marker for instructions and data.
- A label placed just before an instruction/variable implies its address.
- A *data label* identifies the location of a variable.
- It is possible to define multiple data items following a label.

EXAMPLE: LABELS

```
count   DWORD 100

array   DWORD 1024,  2048
        DWORD 4096,  8192
```

Code Labels

- Code labels are used as targets of jumping and looping instructions.
- Each code label must end with a semicolon.

CODE: ILLUSTRATING CODE LABELS

```
target:
        mov ax,bx
        …
        jmp target
```

- A code label can share the same line with an instruction, or it can be on a line by itself:

Instruction Mnemonics

- An *instruction mnemonic* is a short word that identifies an instruction.
- Assembly language instruction mnemonics such as mov, add, and sub provide hints about the type of operation they perform.

EXAMPLE: COMMON MNEMONICS

mov	Move (assign) one value to another
Add	Add two values
sub	Subtract one value from another
mul	Multiply two values
jmp	Jump to a new location
call	Call a procedure

- These mnemonics are sometimes referred to as the opcode.

Operands

- Assembly language instructions can have between zero and three operands.
- An operand can be a register, memory operand, constant expression, or input-output port.
- A *memory operand* is specified by the name of a variable or by one or more registers containing the address of a variable.

Example	Operand Type
96	Constant (*immediate value*)
2 + 4	Constant expression
eax	Register
count , [EBX]	Memory

- The STC instruction has no operand:

    stc ; set carry flat

- The INC instruction has one operand:

    inc eax ; add 1 to eax

- The MOV instruction has two operands:

    mov count, ebx ; move EBX to count

- The IMUL instruction has 3 operands:
    - imul eax,ebx,5 ; multiply ebx by 5 and store the result in eax

- In a two-operand instruction, the first operand is called the *destination*. The second operand is the *source*.

Comments

Comments can be specified in two ways:

- Single-line comments begin with a semicolon (;). All characters following the semicolon on the same line are ignored.

```
; this is a comment in ASL
```

- Block comments begin with the COMMENT directive and a user-specified symbol.
- All subsequent lines of text are ignored by the assembler until the same user-specified symbol appears.

```
Comment !
        Everything here is a comment
        I am also a comment
      !
```

Example: Adding and Substracting Integers

CODE: ILLUSTRATING A FULL ASSEMBLY LANGUAGE PROGRAM

```
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
```

- *Program Output* The following is a snapshot of the program's output, generated by the call to DumpRegs:

```
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  AF=0  PF=1
```

QUICK NOTE: THE NOP INSTRUCTION

The safest (and the most useless) instruction you can write is called NOP (no operation).
It takes up 1 byte of program storage and doesn't do any work.

```
00000000 66 8B C3 mov ax,bx
00000003 90 nop          ; align next instruction
00000004 8B D1 mov edx,ecx
```

ASL Programming Style

- As mentioned, programs are organized around segments, which are usually named code, data, and stack.

- The *code* segment contains all of a program's executable instructions. Ordinarily, the code segment contains one or more procedures, with one designated as the *startup* procedure.
  - In the **AddSub** program, the startup procedure is **main**.

- Another segment, the *stack* segment, holds procedure parameters and local variables.

- The *data* segment holds variables.

A simple template you can use

```
TITLE Program Template        (Template.asm)

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

INCLUDE Irvine32.inc
.data
            ; (insert variables here)

.code
main PROC
        ; (insert executable instructions here)

        exit
main ENDP

        ; (insert additional procedures here)
END main
```

Assembling, Linking, and Running Programs

1. A programmer uses a **text editor** to create a *source file*.

2. The **assembler** reads the source file and produces an *object file*, a machine-language translation of the program. Optionally, it produces a *listing file*.

3. The **linker** reads the object file, combines it with required procedures and produces the *executable file*.

4. The operating system **loader** utility reads the executable file into memory and branches the CPU to the program's starting address, and the program begins to execute.

The listing file

- A *listing file* contains a copy of the program's source code, suitable for printing, with line numbers, offset addresses, translated machine code, and a symbol table.

A small part of the listing file generated by the AddSub.asm program

```
00000000                    .code
00000000                    main PROC

00000000  B8 00010000         mov    eax,10000h        ; EAX = 10000h
00000005  05 00040000         add    eax,40000h        ; EAX = 50000h
0000000A  2D 00020000         sub    eax,20000h        ; EAX = 30000h
0000000F  E8 00000000 E       call   DumpRegs
```

Data Definition

- Intrinsic data types in MASM are only differentiated by their size in bits: 8,16,32,48,64 and 80.

SYNTAX: DEFINING DATA

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

  - Example: myvar WORD 1234H or myvar DW 1234h
- Initializers are compulsory. If you do not want to set an initial value, use a question mark.
  - Example myvar DWORD ?

MASM Data Types

Type	Usage
BYTE	8-bit unsigned integer. B stands for byte
SBYTE	8-bit signed integer. S stands for signed
WORD	16-bit unsigned integer (can also be a Near pointer in real-address mode)
SWORD	16-bit signed integer
DWORD	32-bit unsigned integer (can also be a Near pointer in protected mode). D stands for double
SDWORD	32-bit signed integer. SD stands for signed double
FWORD	48-bit integer (Far pointer in protected mode)
QWORD	64-bit integer. Q stands for quad
TBYTE	80-bit (10-byte) integer. T stands for Ten-byte

ASL Legacy Data Types

Directive	Usage
DB	8-bit integer
DW	16-bit integer
DD	32-bit integer or real
DQ	64-bit integer or real
DT	define 80-bit (10-byte) integer

- Multiple initializers can be used in the same data definition as in:

    mylist BYTE 10,20,30,40,50

- In this case, the label (mylist) refers only to the first item. Remaining items are stored in subsequent memory locations.
- If mylist has an offset of 0000, then the above declaration has the following layout.

Offset	Value
0000:	10
0001:	20
0002:	30
0003:	40

- Within a single data definition, its initializers can use different radixes. Character and string constants can be freely mixed.

    list BYTE 10, ‘W’, 41h, 00100010b;

Strings

- To define a string of characters, enclose them in single or double quotation marks, as in:

```
greeting1 BYTE "Good afternoon",0
greeting2 BYTE 'Good night',0
```

- The most common type of string ends with a null byte (containing 0).

- Each character will take up one byte of storage.

- A string can be divided between multiple lines without having to supply a label for each line:

```
Welcome_message1 BYTE “Welcome to COE 381”, 0dh, 0ah,
                 BYTE “This is a very easy course to understand.”, 0
```

- Hexadecimal codes 0d and 0a produce a carriage return.

The DUP Operator

- The DUP operator can be used to assign a single value to multiple memory locations.

- It can be used with both uninitialized and initialized data.

- Examples:

```
BYTE 20 DUP(0)
BYTE 20 DUP(?)
BYTE 4 DUP("STACK")
```

SELF TEST EXERCISE

Rewrite the ASL program addSub.asm of slide 14, by replacing all immediate values with variables.

Symbolic Constants

- It is possible to create constants in ASL.
- The values of constants cannot change at *runtime*.

SYNTAX: DEFINING SYMBOLS
name = initializer   eg. COUNT = 10; Esc_key = 27

- Constants can however be redefined during *assembly time*.

```
COUNT = 5
mov al,COUNT        ; AL = 5
COUNT = 10
mov al,COUNT        ; AL = 10
COUNT = 100
mov al,COUNT
```

Endianness

- Generically “endianness” refers to the way sub-elements are numbered within an element, for example the way that bytes are numbered in a word.
- In little-endian addressing, the lowest byte (or bit) is stored in the lowest address.
- In big-endian addressing, the most significant byte (or bit) is stored in the least address.
- Example: considering the instruction, how is the number 12345678 stored, if count begins at offset 0000?

  ```
  count DD 12345678h
  ```

- x86 processors store and retrieve data from memory using *little endian* order.

The $ operator

- In ASL, the $ operator also known as the *current location counter* returns the offset associated with the current program statement.
- The following statements set LISTSIZE to 0008, given that list is located at address 0000.

```
mylist WORD 1234h,2500h, 12, 12;
MYLISTSIZE = $;
```

- For this to be useful the $ statement has to follow the mylist statement immediately.
- Example: what is the value of ListSize?

```
list BYTE 10,20,30,40
var2 BYTE 20 DUP(?)
ListSize = ($ - list)
```

- The $ operator can be used to calculate the size of arrays and strings.

The EQU directive

- The *EQU directive* associates a symbolic name with an integer expression or some arbitrary text.

SYNTAX: DEFINING SYMBOLS WITH EQU

name EQU *expression*	*egs. PI EQU <3.142>*
name EQU *symbol*	*pressKey EQU <"Press any key to continue...", 0>*
name EQU <*text*>	

- Unlike the = directive, a symbol defined with EQU can never be redefined in the same source code file.

The TEXTEQU directive

- The *TEXTEQU directive*, similar to EQU, creates what is known as a *text macro*.

SYNTAX: DEFINING SYMBOLS WITH TEXTEQU

name TEXTEQU <*text*>
name TEXTEQU *textmacro*
name TEXTEQU *%constExpr*

- Unlike symbols created with EQU, text macros can be redefined at any time.
- The can also build on each other as in the following example.

SELF TEST EXERCISE

After executing the code below, what will the statement setupAL translate to?

```
rowSize = 5
count TEXTEQU %(rowSize * 2)
move TEXTEQU <mov>
setupAL TEXTEQU <move al,count>
```

SELF TEST EXERCISES

1. Declare a symbolic constant using the equal-sign directive that contains the ASCII code (08h) for the Backspace key.

2. Declare a symbolic constant named **SecondsInDay** using the equal-sign directive and assign it an arithmetic expression that calculates the number of seconds in a 24-hour period.

3. Write a statement that causes the assembler to calculate the number of bytes in the following array, and assign the value to a symbolic constant named **ArraySize**:
myArray WORD 20 DUP(?)

4. Show how to calculate the number of elements in the following array, and assign the value to a symbolic constant named **ArraySize**:
myArray DWORD 30 DUP(?)

5. Use a TEXTEQU expression to redefine “PROC” as “PROCEDURE.”