Microprocessors (0630371)
Fall 2010/2011 – Lecture Notes # 6

Assembly Language Programming -Introduction

Outline of the Lecture

- **Example: adds and subtracts integers.**
- **Assembly language programs structure.**
- **Defining Data (Data Types).**

Example: adds and subtracts integers

```
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
```

- The **TITLE** directive marks the entire line as a comment. You can put anything you want on this line.
- The **INCLUDE** directive copies necessary definitions and setup information from a text file named Irvine32.inc, located in the assembler's INCLUDE directory
- The **.code** directive marks the beginning of the code segment, where all executable statements in a program are located.
- The **PROC** directive identifies the beginning of a procedure.
- The **MOV** instruction moves (copies) the integer 10000h to the EAX register.
- The **ADD** instruction adds 40000h to the EAX register.
- The **SUB** instruction subtracts 20000h from the EAX register.
- The **CALL** statement calls a procedure that displays the current values of the CPU registers.
- The **exit** statement (indirectly) calls a predefined MS-Windows function that halts the program.
- The **ENDP** directive marks the end of the main procedure
- The **END** main directive marks the last line of the program to be assembled.

Program Output

```
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
```

Assembly language programs structure

```
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
```

Defining Segments One important function of assembler directives is to define program sections, or segments.

- The **.DATA** directive identifies the area of a program containing variables:
  ```
  .data
  ```
- The **.CODE** directive identifies the area of a program containing instructions:
  ```
  .code
  ```
- The **.STACK** directive identifies the area of a program holding the runtime stack, setting its size:
  ```
  .stack 100h
  ```

Defining Data (Data Types)

Data Types Essential Characteristics:

- Size in bits: 8, 16, 32, 48, 64, and 80.
- Signed and Unsigned.
- Pointer.
- Integral or Floating-point.

Data Definition Statement

A data definition has the following syntax:

```
[name] directive initializer [,initializer]...
```

- **Name** The optional name assigned to a variable must conform to the rules for identifiers, When you declare an integer variable by assigning a label to a data allocation directive, the assembler allocates memory space for the integer. The variable's name becomes a label for the memory space.
- **Directive** The directive in a data definition statement can be BYTE, WORD, DWORD, SBYTE, SWORD, or any of the types listed in the following table

Type	Usage
BYTE	8-bit unsigned integer
SBYTE	8-bit signed integer
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)
SDWORD	32-bit signed integer
FWORD	48-bit integer (Far pointer in protected mode)
QWORD	64-bit integer
TBYTE	80-bit (10-byte) integer
REAL4	32-bit (4-byte) IEEE short real
REAL8	64-bit (8-byte) IEEE long real
REAL10	80-bit (10-byte) IEEE extended real

- In addition, it can be any of the **legacy data definition directives** shown in the following table

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 tenbyte

- **Initializer** At least one initializer is required in a data definition, even if it is zero. Additional initializers, if any, are separated by commas.
- For integer data types, **initializer** is an integer constant or expression matching the size of the variable's type, such as BYTE or WORD.

- The Initializers and their descriptions are in the following table:

Directive	Description of Initializers
BYTE, DB (byte)	Allocates unsigned numbers from 0 to 255.
SBYTE (signed byte)	Allocates signed numbers from –128 to +127.
WORD, DW (word = 2 bytes)	Allocates unsigned numbers from 0 to 65,535 (64K).
SWORD (signed word)	Allocates signed numbers from –32,768 to +32,767.
DWORD, DD (doubleword = 4 bytes),	Allocates unsigned numbers from 0 to 4,294,967,295 (4 megabytes).
SDWORD (signed doubleword)	Allocates signed numbers from –2,147,483,648 to +2,147,483,647.
FWORD, DF (farword = 6 bytes)	Allocates 6-byte (48-bit) integers. These values are normally used only as pointer variables on the 80386/486 processors.
QWORD, DQ (quadword = 8 bytes)	Allocates 8-byte integers used with 8087-family coprocessor instructions.
TBYTE, DT (10 bytes),	Allocates 10-byte (80-bit) integers if the initializer has a radix specifying the base of the number.

Data Initialization

- You can initialize variables when you declare them with constants or expressions that evaluate to constants. The assembler generates an error if you specify an initial value too large for the variable type.
- A **?** in place of an initializer indicates you do not require the assembler to initialize the variable. The assembler allocates the space but does not write in it.
  - Use ? for **buffer** areas or variables your program will initialize at run time.
- You can declare and initialize variables in one step with the data directives, as these examples show.

```
integer         BYTE    16          ; Initialize byte to 16
negint          SBYTE   -16         ; Initialize signed byte to -16
expression      WORD    4*3         ; Initialize word to 12
signedexp       SWORD   4*3         ; Initialize signed word to 12
empty           QWORD   ?           ; Allocate uninitialized long int
                BYTE    1,2,3,4,5,6 ; Initialize six unnamed bytes
long            DWORD   4294967295  ; Initialize doubleword to
                                    ;   4,294,967,295
longnum         SDWORD  -2147433648 ; Initialize signed doubleword
                                    ;   to -2,147,433,648
tb              TBYTE   2345t       ; Initialize 10-byte binary number
```