Computer Organization and Assembly Language

Lecture 3 – Assembly Language Fundamentals

Basic Elements of Assembly Language

An assembly language program is composed of :

- Constants
- Expressions
- Literals
- Reserved Words
- Mnemonics
- Identifiers
- Directives
- Instructions
- Comments

Integer Constants

- Integer constants can be written in decimal, hexadecimal, octal or binary, by adding a radix (or number base) suffix to the end .
- Radix Suffices:
  - **d** decimal (*the default*)
  - **h** hexadecimal
  - **q** or **o** octal
  - **b** binary

Examples of Integer Constants

- 26 Decimal
- 1Ah Hexadecimal
- 1101b Binary
- 36q Octal
- 2Bh Hexadecimal
- 42Q Octal
- 36D Decimal
- 47d Decimal

Integer Expressions

- An integer expressions is a mathematical expressions involving integer values and integer operators.
- The expressions must be one that can be stored in 32 bits (or less).
- The precedence:
  - ( ) Expressions in Parentheses
  - +, - Unary Plus and minus
  - *, /, Mod Multiply, Divide, Modulus
  - +, - Add, Subtract

Examples of Integer Expressions

Expression	Value
`16 / 5`	3
`- (3 + 4) * (6 – 1)`	-35
`-3 + 4 * 6 – 1`	20
`4 + 5 * 2`	1
`-5 + 2`	
`12 – 1 MOD 5`	
`(4 + 2) * 6`	

Real Number Constants

- There are two types of real number constants:
  - ***Decimal reals***, which contain a sign followed by a number with decimal fraction and an exponent:

    `[sign] integer.[integer][exponent]`

    Examples:

    `2.    +3.0 -44.2E+05 26.E5`
  - ***Encoded reals***, which are represented exactly as they are stored:
    `3F80000r`

Characters Constants

- A character constant is a single character enclosed in single or double quotation marks.
- The assembler converts it to the equivalent value in the binary code ***ASCII***:

  `'A'`

  `"d"`

String Constants

- A string constant is a string of characters enclosed in single or double quotation marks:

```
‘ABC’
“x”
“Goodnight, Gracie”
‘4096’
“This isn’t a test”
‘Say “Goodnight, ” Gracie’
```

Reserved Words

- Reserved words have a special meaning to the assembler and cannot be used for anything other than their specified purpose.
- They include:
  - Instruction mnemonics
  - Directives
  - Operators in constant expressions
  - Predefined symbols such as **`@data`** which return constant values at assembly time.

Identifiers

- ***Identifiers*** are ***names*** that the programmer chooses to represent variables, constants, procedures or labels.
- Identifiers:
  - can have 1 to 247 characters
  - are not case-sensitive
  - begin with a letter , underscore, **`@`** or **`$`** and can also contain digits after the first character.
  - cannot be reserved words

Examples of Identifiers

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

Directives

- Directives are commands for the ***assembler***, telling it how to assemble the program.
- Directives have a syntax similar to assembly language but do not correspond to Intel processor instructions.
- Directives are also case-insensitive:
- Examples

```
.data
.code
name     PROC
```

Instructions

- An instruction in Assembly language consists of a name (or label), an instruction mnemonic, operands and a comment
- The general form is:

  [*name*] [*mnemonic*] [*operands*] [; *comment*]
- Statements are free-form; i.e, they can be written in any column with any number of spaces between in each operand as long as they are on one line and do not pass column 128.

Labels

- Labels are identifiers that serve as place markers within the program for either code or data.
- These are replaces in the machine-language version of the program with numeric addresses.
- We use them because they are more readable:

```
mov    ax, [9020]
```

vs.

```
mov ax, MyVariable
```

Code Labels

- Code labels mark a particular point within the program's code.
- Code labels appear at the beginning and are immediately followed by a colon:

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

Data Labels

- Labels that appear in the operand field of an instruction:

```
mov   first, ax
```

- Data labels must first be declared in the data section of the program:

```
first BYTE 10
```

Instruction Mnemonics

- Instruction mnemonics are abbreviations that identify the operation carried out by the instruction:

  mov - move a value to another location
  add - add two values
  sub - subtract a value from another
  jmp - jump to a new location in the program
  mul - multiply two values
  call - call a procedure

Operands

- Operands in an assembly language instruction can be:
  - constants **`96`**
  - constant expressions **`2 + 4`**
  - registers **`eax`**
  - memory locations **`count`**

Operands and Instructions

- All instructions have a predetermined number of operands.
- Some instructions use no operands:

```
stc     ; set the Carry Flag
```

- Some instructions use one operand:

```
inc     ax   ; add 1 to AX
```

- Some instructions use two operands:

```
mov     count, bx ; add BX to count
```

Comments

- Comments serve as a way for programmers to document their programs,
- Comments can be specified:
  - on a single line, beginning with a semicolon until the end of the line:

```
stc   ; set the Carry Flag
```

  - in a block beginning with the directive COMMENT and a user-specified symbol wchih also ends the comment:

```
COMMENT  !
… …
!
```

Example: Adding Three Numbers

```
TITLE Add And Subtract  (AddSub.asm)
; This program adds and subtracts 32-bit
  integers.
INCLUDE     Irvine32.inc
.code
main  PROC
  mov eax, 10000h       ;Copies 10000h into EAX
  add eax, 40000h       ;Adds 40000h to EAX
  sub eax, 20000h ; Subtracts 20000h from EAX
  call DumpRegs   ; Call the procedure DumpRegs
  exit            ; Call Windows procedure Exit
                  ; to halt the program
main  ENDP        ; marks the end of main
  end main        ; last line to be assembled
```

marks the program's title Treated like a comment

Copies the file's contents into the program

Program output

```
EAX=00030000  EBX=00530000  ECX=0063FF68  EDX=BFFC94C0
ESI=817715DC  EDI=00000000  EBP=0063FF78  ESP=0063FE3C
EIP=00401024  EFL=00000206  CF=0  SF=0  ZF=0  OF=0
```

An Alternative **AddSub**

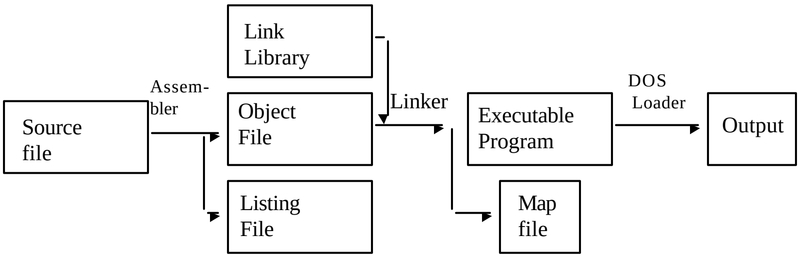
```
TITLE Add And Subtract  (AddSubAlt.asm)
; This program adds and subtracts 32-bit
  integers.

.386    ; Minimum CPU to run this is an Intel 386
.MODEL  flat, stdcall  ; Protected mode program
                       ; using call Windows calls
.STACK 4096     ; The stack is 4096 bytes in size
ExitProcess PROTO, dwExitCode:DWORD
DumpRegs PROTO     ; ExitProcess is an MS-Windows
                   ; procedure
                   ; DumpRegs is a procedure in
                   ; Irvine32.inc
                   ; dwExitCode is a 32-bit value
```

```
.code
main  PROC

   mov eax, 10000h
   add eax, 40000h
   sub eax, 20000h
   call DumpRegs

   INVOKE ExitProcess, 0  ; INVOKE is a directive
                          ; that calls procedures.
                          ; Call the ExitProcess
                          ; procedure
                          ; Pass back a return
                          ; code of zero.
main  ENDP
   end main
```

A 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
```

Assembling, Linking and Running Programs



Assembling and Linking the Program

- A 32-bit assembly language program can be assembled and linked in one step by typing:
  `make32` *filename*
- A ***16***-bit assembly language program can be assembled and linked in one step by typing:
  `make16` *filename*
- Example:

```
make32 addsub
```

Other Files

- In addition to the .asm file (assembler source code), .obj file (object code) and .exe file (executable file), there are other files created by the assembler and linker:
- **.LST (listing) file** – contains the source code and object code of the program
  - **.MAP file** – contains information about the segments being linked
  - **.PDB (Program database) file** – contains supplemental information about the program

Intrinsic Data Types

Type	**Usage**
BYTE	8-bit unsigned integer
SBYTE	8-bit signed integer
WORD	16-bit unsigned integer; also Near Pointer in Real Mode
SWORD	16-bit signed integer
DWORD	32-bit unsigned integer; also Near pointer in Protected Mode
SDWORD	32-bit signed integer

Intrinsic Data Types (continued)

Type	Usage
FWORD	48-bit integer ; Far Pointer in Protected mode
QWORD	64-bit integer
TBYTE	80-bit (ten-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

Defining Data

- A data definition statement allocates storage in memory for variables.
- We write:

  [*name*] *directive* *initializer* [**,** *initializer*]
- There must be at least one initializer.
- If there is no specific intial value, we use the expression ?, which indicate no special value.
- All initializer are converted to binary data by the assembler.

Defining 8-bit Data

- BYTE and SBYTE are used to allcoate storage for an unsigned or signed 8-bit value:

```
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  ?     ; no initial value
.data
value7   BYTE  10h   ; offset is zero
value8   BYTE  20h   ; offset is 1
```

db Directive

- **db** is the older directive for allocating storage for 8-bit data.
- It does not distinguish between signed and unsigned data:

```
val1   db   255  ; unsigned byte
val2   db   -128 ; signed byte
```

Multiple Initializers

- If a definition has multiple initializers, the label is the offset for the first data item:

```
.data
list   BYTE     10, 20, 30, 40
```

Offset	0000	0001	0002	0003
Value:	10	20	30	40

Multiple Initializers (continued)

- Not all definitions need labels:

```
.data
list    BYTE      10, 20, 30, 40
        BYTE      50, 60, 70, 80
        BYTE      81, 82, 83, 84
```

Offset	0000	0001	0002	0003	0004	0005
Value:	10	20	30	40	50	60

Multiple Initializers (continued)

- The different initializers can use different radixes:

```
.data
list1   BYTE      10, 32, 41h, 00100010b
list2   BYTE      0aH, 20H, 'A', 22h
```

- list1 and list2 will have the identical contents, albeit at different offsets.

Defining Strings

- To create a string data definition, enclose a sequence of characters in quotation marks.
- The most common way to end a string is a null byte (0):

```
greeting1   BYTE "Good afternoon", 0
```

is the same as

```
greeting1   BYTE 'G', 'o', 'o',  … 0
```

Defining Strings (continued)

- Strings can be spread over several lines:

```
greeting2 BYTE "Welcome to the Encryption"
          BYTE " Demo program"
          BYTE "created by Kip Irvine",\
               0dh, 0aH
   BYTE   " If you wish to modify this"
          " program, please"
   BYTE   "send me a copy", 0dh, 0ah
```

Concatenates two lines

Using **dup**

- **DUP** repeats a storage allocation however many times is specified:

```
BYTE     20 DUP(0)    ; 20 bytes of zero
BYTE     20 DUP(?)    ; 20 bytes uninitialized
BYTE     2 DUP("STACK")
                      ; 20 bytes "STACKSTACK"
```

Defining 16-bit Data

- The **WORD** and **SWORD** directives allocate storage of one or more 16-bit integers:

```
word1    WORD  65535 ; largest unsigned value
word2    SWORD -32768; smallest signed value
word3    WORD  ?     ; uninitialized value
```

- The **dw** directive can be used to allocated storage for either signed or unsigned integers:

```
val1     dw    65535       ; unsigned
val2     dw    -32768      ; signed
```

Arrays of Words

- You can create an array of word values by listing them or using the **DUP** operator:

```
myList    WORD  1, 2, 3, 4, 5
```

Offset	0000	0002	0004	0006	0008
Value:	1	2	3	4	5

```
array WORD  5     DUP(?)
                        ; 5 values, uninitialized
```

Defining 32-bit Data

- The **DWORD** and **SDWORD** directives allocate storage of one or more 32-bit integers:

```
val1     DWORD       12345678h   ; unsigned
val2     SDWORD      -21474836648; signed
val3     DWORD       20 DUP(?)
                           ; unsigned array
```

- The **dd** directive can be used to allocated storage for either signed or unsigned integers:

```
val1     dd    12345678h        ; unsigned
val2     dw    -21474836648     ; signed
```

Arrays of Doublewords

- You can create an array of word values by listing them or using the **DUP** operator:

```
myList    DWORD 1, 2, 3, 4, 5
```

Offset	0000	0004	0008	000C	0010
Value:	1	2	3	4	5

Defining 64-bit Data

- The **QWORD** directive allocate storage of one or more 64-bit (8-byte) values:

```
quad1    QWORD       1234567812345678h
```

- The **dq** directive can be used to allocated storage:

```
quad1    dq          1234567812345678h
```

Defining 80-bit Data

- The **TBYTE** directive allocate storage of one or more 80-bit integers, used mainly for binary-coded decimal numbers:

```
val1     TBYTE       1000000000123456789h
```

- The **dq** directive can be used to allocated storage:

```
val1     dt          1000000000123456789h
```

Defining Real Number Data

- There are three different ways to define real values:
  - REAL4 defines a 4-byte single-precision real value.
  - REAL8 defines a 8-byte double-precision real value.
  - REAL10 defines a 10-byte extended double-precision real value.
- Each requires one or more real constant initializers.

Examples of Real Data Definitions

```
rVal1      REAL4    -2.1
rVal2      REAL8    3.2E-260
rVal3      REAL10   4.6E+4096
ShortArray REAL4    20 DUP(?)

rVal1      DD   -1.2
rVal2      dq   3.2E-260
rVal3      dt   4.6E+4096
```

Ranges For Real Numbers

Data Type	**Significant Digits**	**Approximate Range**
Short Real	6	1.18×10^{-38} to 3.40×10^{38}
Long Real	15	2.23×10^{-308} to 1.79×10^{308}
Extended Real	19	3.37×10^{-4932} to 1.18×10^{4932}

Little Endian Order

- Consider the number 12345678h:

Little-endian

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

Big-endian

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

Adding Variables to AddSub

```
TITLE Add And Subtract   (AddSub2.asm)
; This program adds and subtracts 32-bit
  integers.
; and stores the sum in a variable
INCLUDE   Irvine32.inc
.data
val1      DWORD10000h
val2      DWORD40000h
val3      DWORD20000h
finalVal  DWORD?
```

```
.code
main PROC
  mov     eax, val1 ; Start with 10000h
  add     eax, val2 ; Add 40000h
  sub     eax, val3 ; Subtract 2000h
  mov     finalVal, eax  ; Save it
  call    DumpRegs  ; Display the
                    ; registers
  exit
main ENDP
  end     main
```

Symbolic Constants

- Equate directives allows constants and literals to be given symbolic names.
- The directives are:
  - Equal-Sign Directive
  - EQU Directive
  - TEXTEQU Directive

Equal-Sign Directive

- The equal-sign directive creates a symbol by assigning a numeric expression to a name.
- The syntax is:

  `name = expression`

- The equal sign directive assigns no storage; it just ensures that occurrences of the name are replaces by the expression.

Equal-Sign Directive (continued)

- Expression must be expressable as 32-bit integers (this requires a .386 or higher directive).
- Examples:

```
prod = 10 * 5    ; Evaluates an expression
maxInt = 7FFFh   ; Maximum 16-bit signed value
minInt = 8000h   ; Minimum 16-bit signed value
maxUInt = 0FFFh  ; Maximum 16-bit unsigned value
String = 'XY'    ; Up to two characters allowed
Count = 500
endvalue = count + 1    ;Can use a predefined symbol

.386
maxLong = 7FFFFFFFh
                 ; Maximum 32-bit signed value
minLong = 80000000h; Minimum 32-bit signed value

maxULong  = 0fffffffh; Maximum 32-bit unsigned value
```

Equal-Sign Directive (continued)

- A symbol defined with an equal-sign directive can be redefined with a different value later within the same program:
  - Statement: Assembled as:

```
count = 5
mov al, count       mov al, 5
mov dl, al          mov al, dl
count = 10
mov cx, count       mov cx, 10
mov dx, count       mov dx, 10
count = 2000
mov ax, count       mov ax, 2000
```

EQU Directive

- The EQU Directive assigns a symbolic name to a string or numeric constant
- Symbols defined using EQU cannot be redefined.
- Expressions are evaluated as integer values, but floating point values are evaluated as strings.
- Strings may be enclosed in the brackets < > to ensure their correct interpretation.
- Examples:

```
Example                        Type of value
maxint   equu   32767          Numeric
maxuint  equ   0FFFFh          Numeric
count    equ 10 * 20           Numeric
float1   equ   <2.345>         String
```

TEXTEQU Directive

- The TEXTEQU directive assigns a name to a sequence of characters.
- Syntax:

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

- Textmacro is a predefined text macro *(more about this later)* constExpr is a numeric expression which is evaluated and used as a string.
- Example:

```
continueMsg  textequ   <"Do you wish to continue?">
.data
prompt1         db       ContinueMsg
```

TEXTEQU Examples

```
;Symbol declarations:
move      texteq u   <mov>
address   textequ   <offset>
; Original code:
  move bx, address value
  move al, 20

; Assembled as:
  mov     bx, offset value
  mov     al, 20
```

TEXTEQU Examples (continued)

```
.data
myString  BYTE “A string”, 0
.code
p1   textequ   <offset MyString>
     mov  bx, p1
               ; bx = offset myString
p1   textequ   <0>
     mov  si, p1         ; si = 0
```

Real-Address Mode Programming

```
TITLE Add And Subtract   (AddSub3.asm)
; This program adds and subtracts 32-bit
; integers and stores the sum in a
; variable. Target : Real Mode
INCLUDE   Irvine16.inc
.data
val1      DWORD10000h
val2      DWORD40000h
val3      DWORD20000h
finalVal  DWORD?
```

```
.code
main  PROC
  mov       ax, @data
  mov       ds, ax       ; initialize the data
                         ; segment register
  mov       eax, val1    ; Start with 10000h
  add       eax, val2    ; Add 40000h
  sub       eax, val3    ; Subtract 2000h
  mov       finalVal, eax     ; Save it
  call      DumpRegs     ; Display the
                         ; registers
  exit
main  ENDP
  end       main
```