ASSEMBLY LANGUAGE (WEEK 2 – 3)

CHAPTER 3 AND 4(PARTIAL) FROM KIP IRVINE

Constants

- Integer Constants
  - Examples: –10, 42d, 10001101b, 0FF3Ah, 777o
  - Radix: b = binary, d = decimal, h = hexadecimal, and o = octal
  - If no radix is given, the integer constant is decimal
  - A hexadecimal beginning with a letter must have a leading 0
- Character and String Constants
  - Enclose character or string in single or double quotes
  - Examples: 'A', "d", 'ABC', "ABC", '4096'
  - Embedded quotes: "single quote ' inside", 'double quote " inside'
  - Each ASCII character occupies a single byte

Assembly Language Statements

- Three types of statements in assembly language
  - Typically, one statement should appear on a line

1. Executable Instructions
   - Generate machine code for the processor to execute at runtime
   - Instructions tell the processor what to do
2. Assembler Directives
   - Provide information to the assembler while translating a program
   - Used to define data, select memory model, etc.
   - Non-executable: directives are not part of instruction set
3. Macros
   - Shorthand notation for a group of statements
   - Sequence of instructions, directives, or other macros

Instructions

- Assembly language instructions have the format:

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

- Instruction Label (optional)
  - Marks the address of an instruction, must have a colon **:**
  - Used to transfer program execution to a labeled instruction
- Mnemonic
  - Identifies the operation (e.g. MOV, ADD, SUB, JMP, CALL)
- Operands
  - Specify the data required by the operation
  - Executable instructions can have zero to three operands
  - Operands can be registers, memory variables, or constants

Instruction Examples

- No operands

```
stc  ; set carry flag
```

- One operand

```
inc  eax ; increment register eax
call Clrscr  ; call procedure Clrscr
jmp  L1  ; jump to instruction with label L1
```

- Two operands

```
add  ebx, ecx  ; register ebx = ebx + ecx
sub  var1, 25  ; memory variable var1 = var1 -
  25
```

- Three operands

```
imul eax,ebx,5 ; register eax = ebx * 5
```

Identifiers

- Identifier is a programmer chosen name
- Identifies variable, constant, procedure, code label
- May contain between 1 and 247 characters
- Not case sensitive
- First character must be a letter (A..Z, a..z), underscore(_), @, ?, or $.
- Subsequent characters may also be digits.
- Cannot be same as assembler reserved word.

Comments

- Comments are very important!
  - Explain the program's purpose
  - When it was written, revised, and by whom
  - Explain data used in the program
  - Explain instruction sequences and algorithms used
  - Application-specific explanations
- Single-line comments
  - Begin with a semicolon **;** and terminate at end of line
- Multi-line comments
  - Begin with **COMMENT** directive and a chosen character
  - End with the same chosen character

YOUR TASK

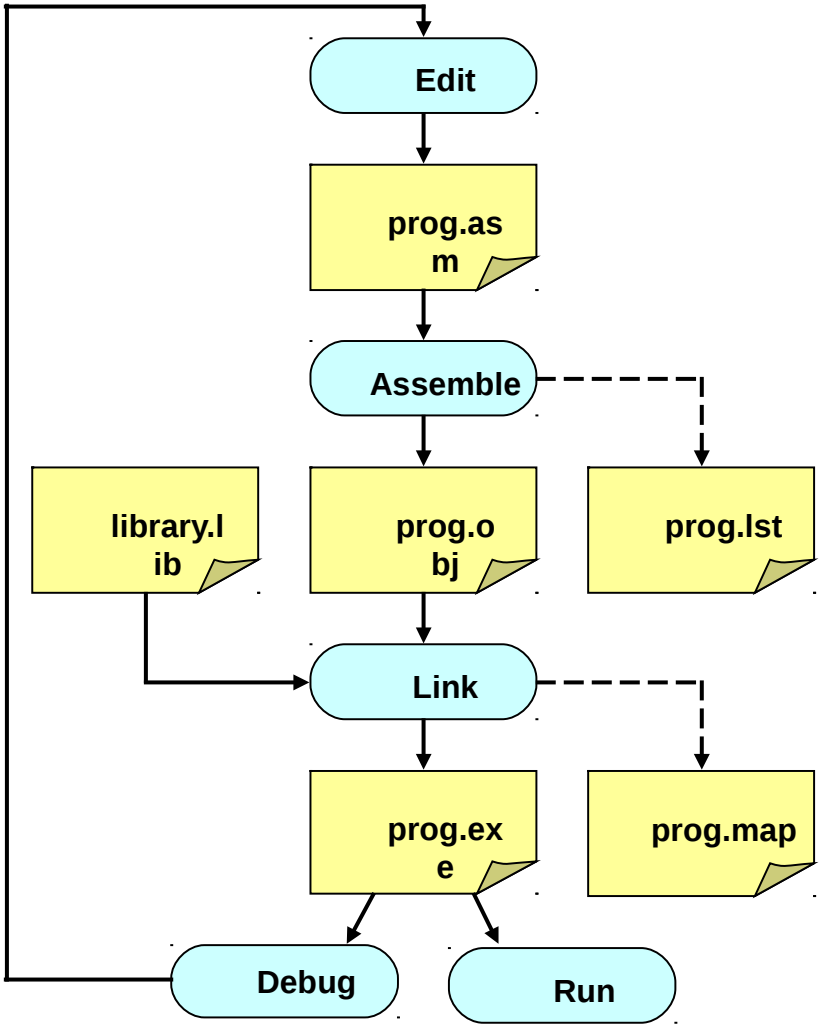
- DEFINE THREE 32 BIT VARIABLES VAR1, VAR2 ,VAR3 AND RESULT . ADD VAR1 AND VAR2 AND SUBTRACT THEIR RESULT WITH VAR3. FINALLY STORE THE RESULT IN A VARIABLE RESULT.

Adding Variables to AddSub

```
INCLUDE Irvine32.inc
.DATA
val1   DWORD 10000h
val2   DWORD 40000h
val3   DWORD 20000h
result DWORD ?
.CODE
main PROC
    mov  eax,val1  ; start with 10000h
  add  eax,val2  ; add 40000h
  sub  eax,val3  ; subtract 20000h
  mov  result,eax    ; store the result (30000h)
  call DumpRegs  ; display the registers
  exit
main ENDP
END main
```

Assemble-Link-Debug Cycle

- Editor
  - Write new (**.asm**) programs
  - Make changes to existing ones
- Assembler: **ML.exe** program
  - Translate (**.asm**) file into object (**.obj**) file in machine language
  - Can produce a listing (**.lst**) file that shows the work of assembler
- Linker: **LINK32.exe** program
  - Combine object (**.obj**) files with link library (**.lib**) files
  - Produce executable (**.exe**) file



Intrinsic Data Types

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

- REAL4
  - IEEE single-precision float
  - Occupies 4 bytes
- REAL8
  - IEEE double-precision
  - Occupies 8 bytes
- REAL10
  - IEEE extended-precision
  - Occupies 10 bytes

Data Definition Statement

- Sets aside storage in memory for a variable
- Syntax:

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

```
val1   BYTE    10
```

- 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
```

Defining Byte Arrays

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

- A string is implemented as an array of characters
  - For convenience, it is usually enclosed in quotation marks
  - It is often terminated with a NULL char (byte value = 0)
- Examples:

```
str1 BYTE "Enter your name", 0
str2 BYTE 'Error: halting program', 0
str3 BYTE 'A','E','I','O','U'
greeting  BYTE "Welcome to the Encryption "
          BYTE "Demo Program", 0
```

Defining Strings – cont'd

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

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

- End-of-line character sequence:
  - 0Dh = 13 = carriage return
  - 0Ah = 10 = line feed

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

Using the DUP Operator

- Use DUP to allocate space for an array or string
  - Advantage: more compact than using a list of initializers
- Syntax

  counter **DUP (*argument*)**

  Counter and *argument* must be constants expressions
- The DUP operator may also be nested

```
var1 BYTE 20 DUP(0)          ; 20 bytes, all equal to zero
var2 BYTE 20 DUP(?)          ; 20 bytes, all uninitialized
```

YOUR TASK

- HOW MANY BYTES WILL GET STORED BY USING FOLLOWING DATA DEFINITIONS?

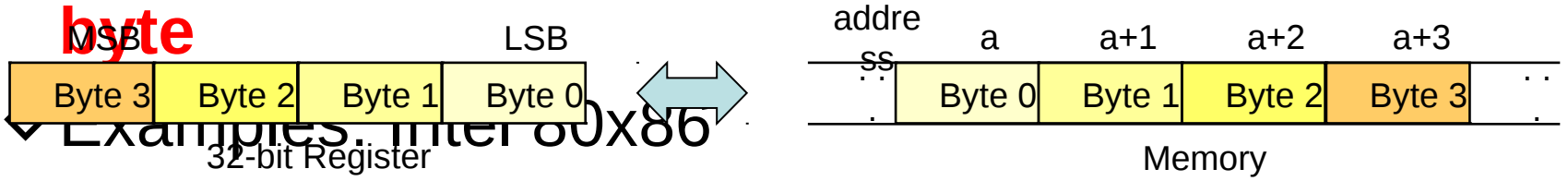
```
VAR5 BYTE 20 DUP(30 DUP("STACK"))
```

HOME TASK:

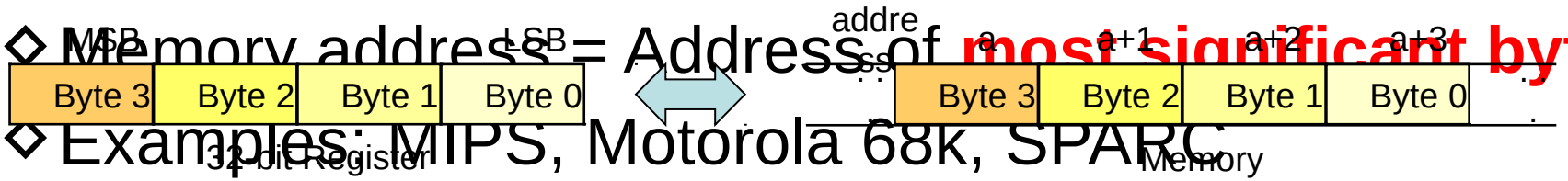
```
var3 BYTE 4 DUP("STACK")
var5 BYTE 2 DUP(5 DUP('*'), 5 DUP('!')) ;
var4 BYTE 10,3 DUP(0),20
```

Byte Ordering and Endianness

- Processors can order bytes within a word in two ways
- Little Endian Byte Ordering
  - Memory address = Address of **least significant byte**
  - Examples: Intel 80x86



- Big Endian Byte Ordering
  - Memory address = Address of **most significant byte**
  - Examples: MIPS, Motorola 68k, SPARC



QUESTION

- HOW DOES FOLLOWING DATA DEFINTION WILL GET STORED IN MEMORY IN LITTLE ENDIAN AND BIG ENDIAN FORMAT??
  - VAR3 DWORD 45678902H

HOME TASK

SHOW THE MEMORY CONTENTS AFTER STORING THE FOLLOWING BYTES IN BIG ENDIAN AND LITTLE ENDIAN FORMAT:

- VAR1 WORD 78H
- VAR2 DWORD 8976H
- VAR3 DWORD 5678452H

Defining Symbolic Constants

- Symbolic Constant
  - Just a name used in the assembly language program
  - Processed by the assembler ⇒ pure text substitution
  - Assembler does NOT allocate memory for symbolic constants
- Assembler provides three directives:
  - = directive
  - EQU directive
  - TEXTEQU directive
- Defining constants has two advantages:
  - Improves program readability
  - Helps in software maintenance: changes are done in one place

Equal-Sign Directive

- *Name* = *Expression*
  - *Name* is called a symbolic constant
  - *Expression* is an integer constant expression
- Good programming style to use symbols

```
COUNT = 500     ; NOT a variable (NO memory allocation)
. . .
mov eax, COUNT ; mov eax, 500
. . .
COUNT = 600     ; Processed by the assembler
. . .
mov ebx, COUNT ; mov ebx, 600
```

- *Name* can be redefined in the program

EQU Directive

- Three Formats:
  - *Name* EQU *Expression* Integer constant expression
  - *Name* EQU *Symbol* Existing symbol name
  - *Name* EQU *<text>* Any text may appear within < …>

```
SIZE      EQU 10*10          ; Integer constant expression
PI        EQU <3.1416>       ; Real symbolic constant
PressKey EQU <"Press any key to continue...",0>
.DATA
prompt BYTE PressKey
```

- No Redefinition: *Name* cannot be redefined with EQU

TEXTEQU Directive

- TEXTEQU creates a text macro. Three Formats:

  Name TEXTEQU *<text>* assign any text to *name*

  Name TEXTEQU *textmacro* assign existing text macro

  Name TEXTEQU *%constExpr* constant integer expression

- *Name* can be redefined at any time (unlike EQU)

```
ROWSIZE = 5
COUNT   TEXTEQU  %(ROWSIZE * 2)    ; evaluates to 10
ContMsg TEXTEQU  <"Do you wish to continue (Y/N)?">
.DATA
prompt  BYTE     ContMsg
.CODE
MOVAL       ; generates: mov al,10
```

OFFSET Operator

- OFFSET = address of a variable within its segment

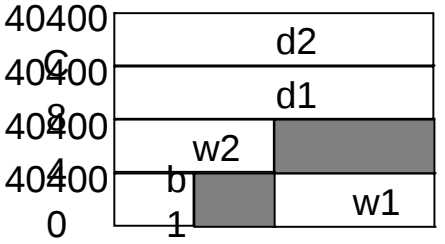
```
.DATA
bVal   BYTE  ?   ; Assume bVal is at 00404000h
wVal   WORD  ?
dVal   DWORD ?
dVal2  DWORD ?

.CODE
mov esi, OFFSET bVal  ; ESI = 00404000h
mov esi, OFFSET wVal  ; ESI = 00404001h
mov esi, OFFSET dVal  ; ESI = 00404003h
mov esi, OFFSET dVal2 ; ESI = 00404007h
```

ALIGN Directive

- ALIGN directive aligns a variable in memory
- Syntax: ALIGN *bound*
  - Where *bound* can be 1, 2, 4
- Address of a variable should be a multiple of *bound*
- Assembler inserts empty bytes to enforce alignment

```
.DATA          ; Assume that
b1 BYTE   ?  ; Address of b1 = 00404000h
ALIGN 2        ; Skip one byte
w1 WORD   ?  ; Address of w1 = 00404002h
w2 WORD   ?  ; Address of w2 = 00404004h
ALIGN 4        ; Skip two bytes
d1 DWORD  ?  ; Address of d1 = 00404008h
d2 DWORD  ?  ; Address of d2 = 0040400Ch
```



TYPE Operator

- TYPE operator
  - Size, in bytes, of a single element of a data declaration

```
.DATA
var1 BYTE ?
var2 WORD ?
var3 DWORD ?
var4 QWORD ?

.CODE
mov eax, TYPE var1 ; eax = 1
mov eax, TYPE var2 ; eax = 2
mov eax, TYPE var3 ; eax = 4
mov eax, TYPE var4 ; eax = 8
```

LENGTHOF Operator

- LENGTHOF operator
  - Counts the number of elements in a single data declaration

```
.DATA
array1     WORD     30 DUP(?),0,0
array2     WORD     5 DUP(3 DUP(?))
array3     DWORD    1,2,3,4
digitStr   BYTE    "12345678",0

.code
mov ecx, LENGTHOF array1 ; ecx = 32
mov ecx, LENGTHOF array2 ; ecx = 15
mov ecx, LENGTHOF array3 ; ecx = 4
mov ecx, LENGTHOF digitStr  ; ecx = 9
```

SIZEOF Operator

- SIZEOF operator
  - Counts the number of bytes in a data declaration
  - Equivalent to multiplying LENGTHOF by TYPE

```
.DATA
array1      WORD     30 DUP(?),0,0
array2      WORD     5 DUP(3 DUP(?))
array3      DWORD    1,2,3,4
digitStr    BYTE    "12345678",0

.CODE
mov ecx, SIZEOF array1   ; ecx = 64
mov ecx, SIZEOF array2   ; ecx = 30
mov ecx, SIZEOF array3   ; ecx = 16
mov ecx, SIZEOF digitStr ; ecx = 9
```

Multiple Line Declarations

A data declaration spans multiple lines if each line (except the last) ends with a comma

The LENGTHOF and SIZEOF operators include all lines belonging to the declaration

```
.DATA
array WORD  10,20,
      30,40,
      50,60

.CODE
mov eax, LENGTHOF array ;
6
mov ebx, SIZEOF array   ;
12
```

In the following example, array identifies the first line WORD declaration only

Compare the values returned by LENGTHOF and SIZEOF here to those on the left

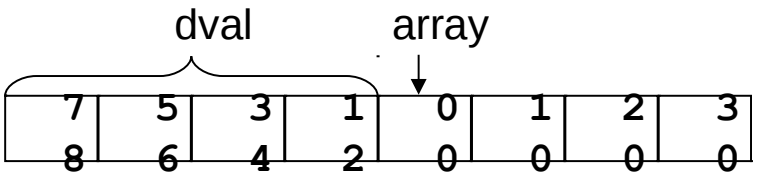
```
.DATA
array WORD 10,20
   WORD 30,40
   WORD 50,60

.CODE
mov eax, LENGTHOF array ;
2
mov ebx, SIZEOF array   ;
4
```

PTR Operator

- PTR Provides the flexibility to access part of a variable
- Can also be used to combine elements of a smaller type
- Syntax: *Type* PTR (Overrides default type of a variable)

```
.DATA
dval  DWORD 12345678h
array BYTE
00h,10h,20h,30h
```

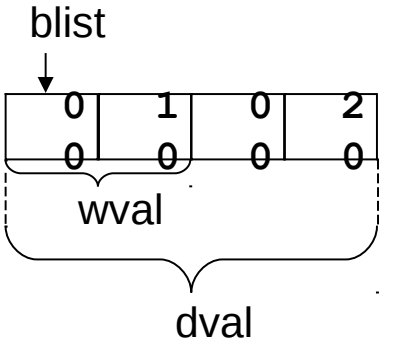



```
.CODE
mov al,  dval                    ; error – why?
mov al,  BYTE PTR dval           ; al = 78h
mov ax,  dval                    ; error – why?
mov ax,  WORD PTR dval           ; ax = 5678h
mov eax, array                   ; error – why?
mov eax, DWORD PTR array         ; eax =
                                 30201000h
```

LABEL Directive

- Assigns an alternate name and type to a memory location
- LABEL does not allocate any storage of its own
- Removes the need for the PTR operator
- Format: *Name* LABEL *Type*

```
.DATA
dval    LABEL DWORD
wval    LABEL WORD
blist   BYTE 00h,10h,00h,20h
.CODE
mov eax, dval       ; eax =
mov cx,  wval       20001000h
mov dl,  blist      ; cx  = 1000h
                    ; dl  = 00h
```



Summary

- Instruction ⇒ executed at runtime
- Directive ⇒ interpreted by the assembler
- .STACK, .DATA, and .CODE
  - Define the code, data, and stack sections of a program
- Edit-Assemble-Link-Debug Cycle
- Data Definition
  - BYTE, WORD, DWORD, QWORD, etc.
  - DUP operator
- Symbolic Constant
  - =, EQU, and TEXTEQU directives
- Data-Related Operators
  - OFFSET, ALIGN, TYPE, LENGTHOF, SIZEOF, PTR, and LABEL