Introduction to Assembly Language

Presentation Outline

- Basic Elements of Assembly Language
- Flat Memory Program Template
- Example: Adding and Subtracting Integers
- Assembling, Linking, and Debugging Programs
- Defining Data
- Defining Symbolic Constants
- Data-Related Operators and Directives

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
```

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

TITLE and .MODEL Directives

- **TITLE** line (optional)
  - Contains a brief heading of the program and the disk file name
- **.MODEL** directive
  - Specifies the memory configuration
  - For our purposes, the **FLAT** memory model will be used
    - Linear 32-bit address space (no segmentation)
  - **STDCALL** directive tells the assembler to use …
    - Standard conventions for names and procedure calls
- **.686** processor directive
  - Used **before** the **.MODEL** directive
  - Program can use instructions of Pentium P6 architecture
  - At least the **.386** directive should be used with the **FLAT** model

.STACK, .DATA, & .CODE Directives

- **.STACK** directive
  - Tells the assembler to define a runtime stack for the program
  - The size of the stack can be optionally specified by this directive
  - The runtime stack is required for procedure calls
- **.DATA** directive
  - Defines an area in memory for the program data
  - The program's variables should be defined under this directive
  - Assembler will allocate and initialize the storage of variables
- **.CODE** directive
  - Defines the code section of a program containing instructions
  - Assembler will place the instructions in the code area in memory

INCLUDE, PROC, ENDP, and END

- **INCLUDE** directive
  - Causes the assembler to include code from another file
  - We will include **Irvine32.inc** provided by the author Kip Irvine
    - Declares procedures implemented in the **Irvine32.lib** library
    - To use this library, you should link **Irvine32.lib** to your programs
- **PROC** and **ENDP** directives
  - Used to define procedures
  - As a convention, we will define ***main*** as the first procedure
  - Additional procedures can be defined after ***main***
- **END** directive
  - Marks the end of a program
  - Identifies the name (***main***) of the program's startup procedure

Suggested Coding Standards

- Some approaches to capitalization
  - Capitalize nothing
  - Capitalize everything
  - Capitalize all reserved words, mnemonics and register names
  - Capitalize only directives and operators
  - MASM is NOT case sensitive: does not matter what case is used
- Other suggestions
  - Use meaningful identifier names
  - Use blank lines between procedures
  - Use indentation and spacing to align instructions and comments
    - Use tabs to indent instructions, but do not indent labels
    - Align the comments that appear after the instructions

Understanding Program Termination

- The **exit** at the end of main procedure is a **macro**
  - Defined in **Irvine32.inc**
  - Expanded into a call to **ExitProcess** that terminates the program
  - **ExitProcess** function is defined in the **kernel32** library
  - We can replace **exit** with the following:

```
push 0       ; push parameter 0 on stack
call ExitProcess    ; to terminate program
```

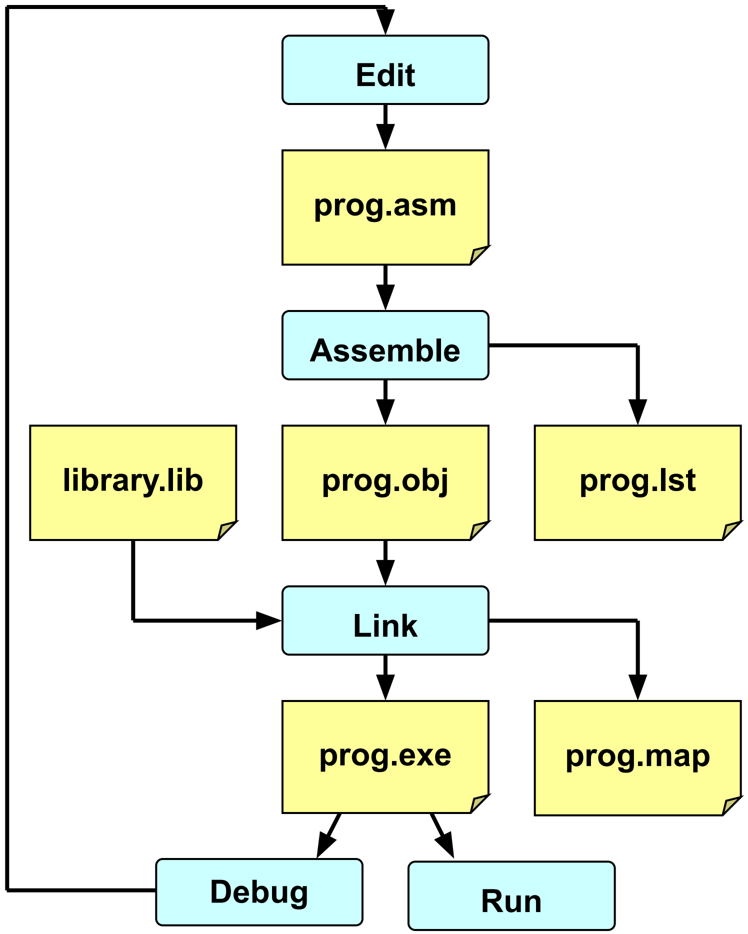
  - You can also replace **exit** with: `INVOKE ExitProcess, 0`
- **PROTO** directive (Prototypes)
  - Declares a procedure used by a program and defined elsewhere

```
ExitProcess PROTO, ExitCode:DWORD
```

  - Specifies the parameters and types of a given procedure

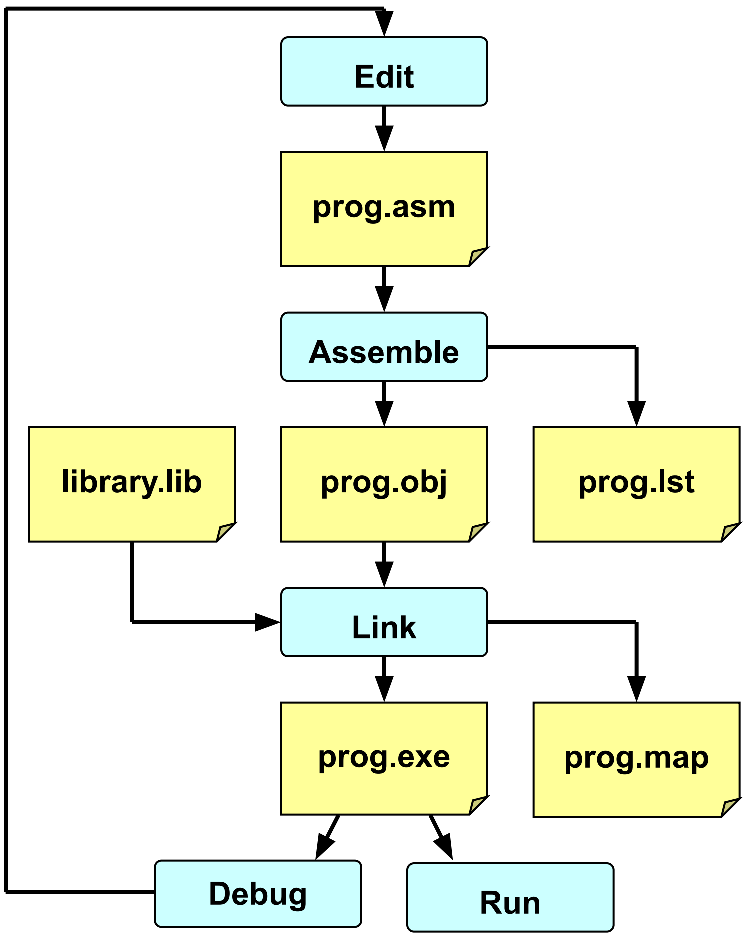
Assemble-Link-Debug Cycle

- 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
  - Can produce optional (**.map**) file



Assemble-Link-Debug Cycle – cont'd

- **MAKE32.bat**
  - Batch command file
  - Assemble and link in one step
- Debugger: **WINDBG.exe**
  - Trace program execution
    - Either step-by-step, or
    - Use breakpoints
  - View
    - Source (**.asm**) code
    - Registers
    - Memory by name & by address
    - Modify register & memory content
  - Discover errors and go back to the editor to fix the program bugs



Processor Registers

There are ten 32-bit and six 16-bit processor registers in IA-32 architecture. The registers are grouped into three categories −

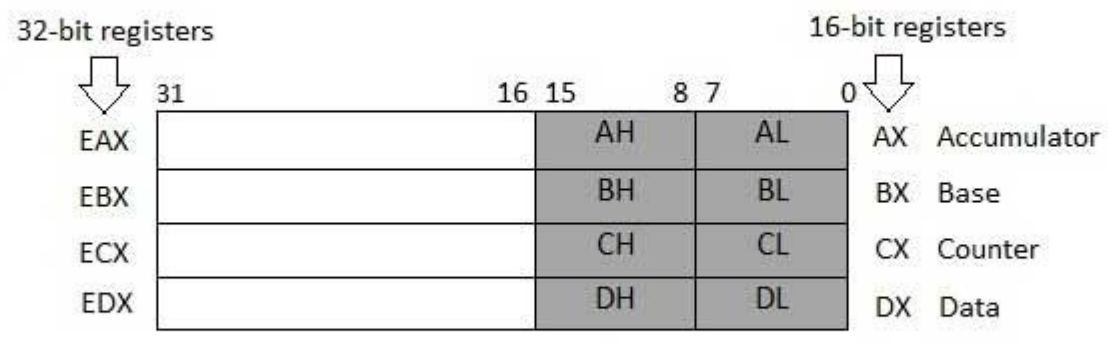
- General registers,
- Control registers, and
- Segment registers.

The general registers are further divided into the following groups −

- Data registers,
- Pointer registers, and
- Index registers.

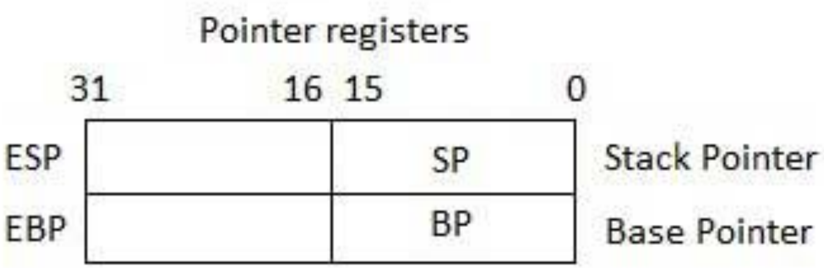
Data Registers

- Four 32-bit data registers are used for arithmetic, logical, and other operations. These 32-bit registers can be used in three ways −
- As complete 32-bit data registers: EAX, EBX, ECX, EDX.
- Lower halves of the 32-bit registers can be used as four 16-bit data registers: AX, BX, CX and DX.
- AX is the primary accumulator
- BX is known as the base register
- CX is known as the count register, as the ECX
- DX is known as the data register. It is also used in input/output operations



Pointer Registers

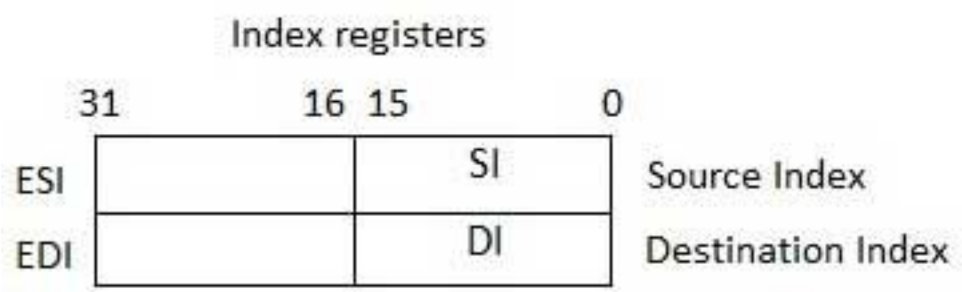
- The pointer registers are 32-bit EIP, ESP, and EBP registers and corresponding 16-bit right portions IP, SP, and BP. There are three categories of pointer registers −
- Instruction Pointer (IP) − The 16-bit IP register stores the offset address of the next instruction to be executed. IP in association with the CS register (as CS:IP) gives the complete address of the current instruction in the code segment.
- Stack Pointer (SP) − The 16-bit SP register provides the offset value within the program stack. SP in association with the SS register (SS:SP) refers to be current position of data or address within the program stack.
- Base Pointer (BP) − The 16-bit BP register mainly helps in referencing the parameter variables passed to a subroutine. The address in SS register is combined with the offset in BP to get the location of the parameter. BP can also be combined with DI and SI as base register for special addressing.



Index Registers

- The 32-bit index registers, ESI and EDI, and their 16-bit rightmost portions. SI and DI, are used for indexed addressing and sometimes used in addition and subtraction. There are two sets of index pointers −

- Source Index (SI) − It is used as source index for string operations.
- Destination Index (DI) − It is used as destination index for string operations.



Control Registers

- The 32-bit instruction pointer register and the 32-bit flags register combined are considered as the control registers.

- Many instructions involve comparisons and mathematical calculations and change the status of the flags and some other conditional instructions test the value of these status flags to take the control flow to other location.

The following table indicates the position of flag bits in the 16-bit Flags register:

Flag:					O	D	I	T	S	Z		A		P		C
Bit no:	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0

Control Registers: Flag Bits

Overflow Flag (OF) − It indicates the overflow of a high-order bit (leftmost bit) of data after a signed arithmetic operation.

Direction Flag (DF) − It determines left or right direction for moving or comparing string data. When the DF value is 0, the string operation takes left-to-right direction and when the value is set to 1, the string operation takes right-to-left direction.

Interrupt Flag (IF) − It determines whether the external interrupts like keyboard entry, etc., are to be ignored or processed. It disables the external interrupt when the value is 0 and enables interrupts when set to 1.

Trap Flag (TF) − It allows setting the operation of the processor in single-step mode. The DEBUG program we used sets the trap flag, so we could step through the execution one instruction at a time.

Sign Flag (SF) − It shows the sign of the result of an arithmetic operation. This flag is set according to the sign of a data item following the arithmetic operation. The sign is indicated by the high-order of leftmost bit. A positive result clears the value of SF to 0 and negative result sets it to 1.

Control Registers: Flag Bits contd..

Zero Flag (ZF) − It indicates the result of an arithmetic or comparison operation. A nonzero result clears the zero flag to 0, and a zero result sets it to 1.

Auxiliary Carry Flag (AF) − It contains the carry from bit 3 to bit 4 following an arithmetic operation; used for specialized arithmetic. The AF is set when a 1-byte arithmetic operation causes a carry from bit 3 into bit 4.

Parity Flag (PF) − It indicates the total number of 1-bits in the result obtained from an arithmetic operation. An even number of 1-bits clears the parity flag to 0 and an odd number of 1-bits sets the parity flag to 1.

Carry Flag (CF) − It contains the carry of 0 or 1 from a high-order bit (leftmost) after an arithmetic operation. It also stores the contents of last bit of a shift or rotate operation.

Segment Registers

Segments are specific areas defined in a program for containing data, code and stack. There are three main segments −

- **Code Segment −** It contains all the instructions to be executed. A 16-bit Code Segment register or CS register stores the starting address of the code segment.
- **Data Segment −** It contains data, constants and work areas. A 16-bit Data Segment register or DS register stores the starting address of the data segment.
- **Stack Segment −** It contains data and return addresses of procedures or subroutines. It is implemented as a 'stack' data structure. The Stack Segment register or SS register stores the starting address of the stack.

Apart from the DS, CS and SS registers, there are other extra segment registers - ES (extra segment), FS and GS, which provide additional segments for storing data.

A segment begins in an address evenly divisible by 16 or hexadecimal 10. So, the rightmost hex digit in all such memory addresses is 0, which is not generally stored in the segment registers.

Example

```
section .text
   global _start         ;must be declared for linker (gcc)

_start:             ;tell linker entry point
   mov  edx,len   ;message length
   mov  ecx,msg   ;message to write
   mov  ebx,1     ;file descriptor (stdout)
   mov  eax,4     ;system call number (sys_write)
   int  0x80      ;call kernel

   mov  edx,9     ;message length
   mov  ecx,s2    ;message to write
   mov  ebx,1     ;file descriptor (stdout)
   mov  eax,4     ;system call number (sys_write)
   int  0x80      ;call kernel

   mov  eax,1     ;system call number (sys_exit)
   int  0x80      ;call kernel

section .data
msg db 'Displaying 9 stars',0xa ;a message
len equ $ - msg  ;length of message
s2 times 9 db '*'
```

Listing File

- Use it to see how your program is assembled
- Contains
  - Source code
  - Object code
  - Relative addresses
  - Segment names
  - Symbols
    - Variables
    - Procedures
    - Constants

Object & source code in a listing file

```
00000000 .code
00000000 main PROC
00000000  B8 00060000     mov eax, 60000h
00000005  05 00080000     add eax, 80000h
0000000A  2D 00020000     sub eax, 20000h

0000000F  6A 00          push 0
00000011  E8 00000000 E        call
ExitProcess
00000016 main ENDP
         END main
```

Relative Addresses **object code (hexadecimal)** **source code**

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
- May optionally assign a name (label) to the data
- 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
```

- MASM does not prevent you from initializing a BYTE with a negative value, but it's considered poor style.
- If you declare a SBYTE variable, the Microsoft debugger will automatically display its value in decimal with a leading sign.

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

Defining 16-bit and 32-bit Data

- Define storage for 16-bit and 32-bit integers
  - Signed and Unsigned
  - Single or multiple initial values

```
word1  WORD   65535       ; largest unsigned 16-bit value
word2  SWORD  -32768      ; smallest signed 16-bit value
word3  WORD   "AB"     ; two characters fit in a WORD
array1 WORD   1,2,3,4,5   ; array of 5 unsigned words
array2 SWORD  5 DUP(?)    ; array of 5 signed words
dword1 DWORD  0ffffffffh   ; largest unsigned 32-bit
value
dword2 SDWORD -2147483648  ; smallest signed 32-bit value
array3 DWORD  20 DUP(?)    ; 20 unsigned double words
array4 SDWORD -3,-2,-1,0,1 ; 5 signed double words
```

QWORD, TBYTE, and REAL Data

- QWORD and TBYTE
  - Define storage for 64-bit and 80-bit integers
  - Signed and Unsigned
- REAL4, REAL8, and REAL10
  - Defining storage for 32-bit, 64-bit, and 80-bit floating-point data

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

Symbol Table

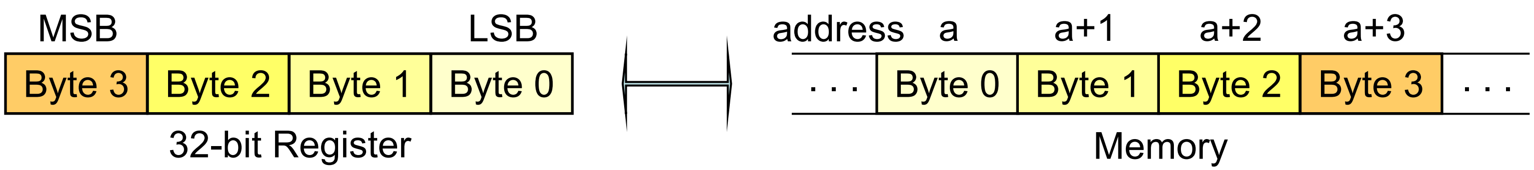
- Assembler builds a symbol table
  - So we can refer to the allocated storage space by name
  - Assembler keeps track of each name and its offset
  - Offset of a variable is relative to the address of the first variable
- Example Symbol Table

```
.DATA
value   WORD    0
sum     DWORD   0
marks   WORD   10 DUP (?)
msg     BYTE   'The grade is:',0
char1   BYTE   ?
```

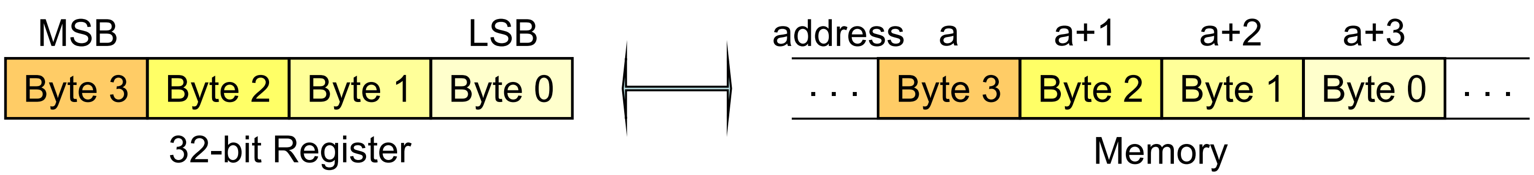
Name	Offset
value	0
sum	2
marks	6
msg	26
char1	40

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



Adding Variables to AddSub

```
TITLE Add and Subtract, Version 2            (AddSub2.asm)
.686
.MODEL FLAT, STDCALL
.STACK
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
```

Defining Symbolic Constants

- Symbolic Constant
  - Just a name used in the assembly language program
  - Processed by the assembler $\Rightarrow$ 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
MOVAL   TEXTEQU  <mov al,COUNT>
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
  - In FLAT memory, one address space is used for code and data
  - OFFSET = linear address of a variable (32-bit number)

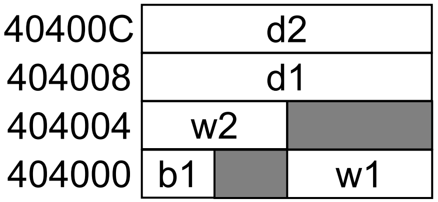
```
.DATA
bVal   BYTE  ? ; Assume bVal is at 00404000h
wVal   WORD  ?
dVal   DWORD ?
dVal2  DWORD ?

.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, or 16
- 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

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,
       30,40,
       50,60

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

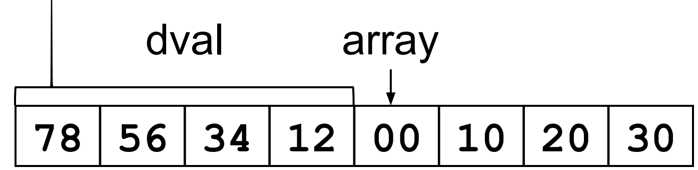
```
.DATA
array  WORD 10,20
    WORD 30,40
    WORD 50,60

.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
```

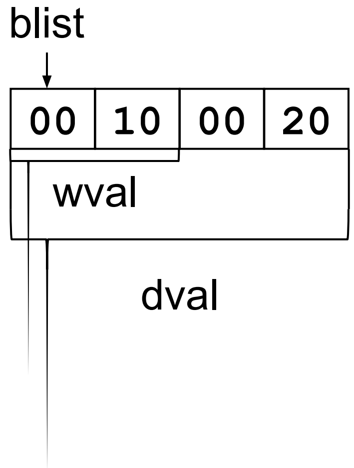



```
.CODE
mov al,   dval               ; error – why?
mov al,   BYTE PTR dval      ; al = 78h
mov ax,   dval               ; error – why?
mov ax,   WORD PTR dval      ; ax = 5678h
mov eax,  array              ; error – why?
mov eax,  DWORD PTR array    ; eax = 30201000h
```

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 = 20001000h
mov cx,  wval       ; cx  = 1000h
mov dl,  blist      ; dl  = 00h
```



Assembly - System Calls

- System calls are APIs for the interface between the user space and the kernel space.

You can make use of Linux system calls in your assembly programs. You need to take the following steps for using Linux system calls in your program −

- Put the system call number in the EAX register.
- Store the arguments to the system call in the registers EBX, ECX, etc.
- Call the relevant interrupt (80h).
- The result is usually returned in the EAX register.

Assembly - System Calls

The following code snippet shows the use of the system call sys_exit –

```
mov       eax,1               ; system call number (sys_exit)
int       0x80                ; call kernel
```

The following code snippet shows the use of the system call sys_write –

```
mov       edx,4               ; message length
mov       ecx,msg             ; message to write
mov       ebx,1               ; file descriptor (stdout)
mov       eax,4               ; system call number (sys_write)
int       0x80                ; call kernel
```

Assembly - System Calls

All the syscalls are listed in /usr/include/asm/unistd.h, together with their numbers (the value to put in EAX before you call int 80h).

The following table shows some of the system calls used in this tutorial –

%eax	Name	%ebx	%ecx	%edx	%esx	%edi
1	sys_exit	int	-	-	-	-
2	sys_fork	struct pt_regs	-	-	-	-
3	sys_read	unsigned int	char *	size_t	-	-
4	sys_write	unsigned int	const char *	size_t	-	-
5	sys_open	const char *	int	int	-	-
6	sys_close	unsigned int	-	-	-	-

Allocating Storage Space for Initialized Data

The syntax for storage allocation statement for initialized data is −

[variable-name]   define-directive   initial-value   [,initial-value]...

Where, variable-name is the identifier for each storage space. The assembler associates an offset value for each variable name defined in the data segment. There are five basic forms of the define directive −

Directive	Purpose	Storage Space
DB	Define Byte	allocates 1 byte
DW	Define Word	allocates 2 bytes
DD	Define Doubleword	allocates 4 bytes
DQ	Define Quadword	allocates 8 bytes

Allocating Storage Space for Initialized Data

Example:

```
choice          DB    'y'
number          DW    12345
neg_number      DW    -12345
big_number      DQ    123456789
real_number1    DD    1.234
real_number2    DQ    123.456
```

Assembly - Instructions

Some common Assembly language instructions are:

1. MOV
2. INC
3. DEC
4. ADD and SUB
5. MUL/IMUL
6. DIV/ IDIV
7. Logical Instructions
8. Conditional

1. MOV

We have already used the MOV instruction that is used for moving data from one storage space to another. The MOV instruction takes two operands.

The syntax of the MOV instruction is −

```
MOV  destination, source
```

The MOV instruction may have one of the following five forms −

```
MOV  register, register
MOV  register, immediate
MOV  memory, immediate
MOV  register, memory
MOV  memory, register
```

2. INC

The INC instruction is used for incrementing an operand by one. It works on a single operand that can be either in a register or in memory.

The INC instruction has the following syntax −

```
INC destination
```

The operand destination could be an 8-bit, 16-bit or 32-bit operand.
Example

INC EBX ; Increments 32-bit register
INC DL ; Increments 8-bit register
INC [count] ; Increments the count variable

3. DEC

The DEC instruction is used for decrementing an operand by one. It works on a single operand that can be either in a register or in memory.

The DEC instruction has the following syntax −

```
DEC destination
```

The operand *destination* could be an 8-bit, 16-bit or 32-bit operand.

```
segment .data
   count dw  0
   value db  15

segment .text
   inc [count]
   dec [value]

   mov ebx, count
   inc word [ebx]

   mov esi, value
   dec byte [esi]
```

4. ADD and SUB

The ADD and SUB instructions are used for performing simple addition/subtraction of binary data in byte, word and doubleword size, i.e., for adding or subtracting 8-bit, 16-bit or 32-bit operands, respectively.

The ADD and SUB instructions have the following syntax −

```
ADD/SUB destination, source
```

The ADD/SUB instruction can take place between −

- Register to register
- Memory to register
- Register to memory
- Register to constant data
- Memory to constant data

5. MUL/ IMUL

There are two instructions for multiplying binary data. The MUL (Multiply) instruction handles unsigned data and the IMUL (Integer Multiply) handles signed data. Both instructions affect the Carry and Overflow flag.

```
MUL/IMUL multiplier
```

- Multiplicand in both cases will be in an accumulator, depending upon the size of the multiplicand and the multiplier and the generated product is also stored in two registers depending upon the size of the operands. Following section explains MUL instructions with three different cases −

MOV AL, 10

MOV DL, 25

MUL DL

MOV DL, 0FFH ; DL= -1

MOV AL, 0BEH ; AL = -66

IMUL DL

6. DIV/ IDIV

The division operation generates two elements - a quotient and a remainder. In case of multiplication, overflow does not occur because double-length registers are used to keep the product. However, in case of division, overflow may occur. The processor generates an interrupt if overflow occurs.

The DIV (Divide) instruction is used for unsigned data and the IDIV (Integer Divide) is used for signed data.

```
DIV/IDIV        divisor
```

The dividend is in an accumulator. Both the instructions can work with 8-bit, 16-bit or 32-bit operands.

MOV AL, 10

MOV DL, 25

MUL DL

MOV DL, 0FFH ; DL= -1

MOV AL, 0BEH ; AL = -66

IMUL DL

7. Logical Isntructions

The processor instruction set provides the instructions AND, OR, XOR, TEST, and NOT Boolean logic, which tests, sets, and clears the bits according to the need of the program.

The format for these instructions −

Sr.No.	Instruction	Format
1	AND	AND operand1, operand2
2	OR	OR operand1, operand2
3	XOR	XOR operand1, operand2
4	TEST	TEST operand1, operand2
5	NOT	NOT operand1

The first operand in all the cases could be either in register or in memory. The second operand could be either in register/memory or an immediate (constant) value.

8. Conditional

Conditional execution in assembly language is accomplished by several looping and branching instructions.

1. The CMP instruction compares two operands. It is generally used in conditional execution. This instruction basically subtracts one operand from the other for comparing whether the operands are equal or not. It does not disturb the destination or source operands. It is used along with the conditional jump instruction for decision making.

   CMP destination, source

2. Unconditional Jump using the JMP instruction provides a label name where the flow of control is transferred immediately.

   JMP label

3. Conditional Jump: If some specified condition is satisfied in conditional jump, the control flow is transferred to a target instruction.

8. Conditional contd.

Following are the conditional jump instructions used on signed data used for arithmetic operations –

Instruction	Description	Flags tested
JE/JZ	Jump Equal or Jump Zero	ZF
JNE/JNZ	Jump not Equal or Jump Not Zero	ZF
JG/JNLE	Jump Greater or Jump Not Less/Equal	OF, SF, ZF
JGE/JNL	Jump Greater/Equal or Jump Not Less	OF, SF
JL/JNGE	Jump Less or Jump Not Greater/Equal	OF, SF
JLE/JNG	Jump Less/Equal or Jump Not Greater	OF, SF, ZF

Following are the conditional jump instructions used on unsigned data used for logical operations –

Instruction	Description	Flags tested
JE/JZ	Jump Equal or Jump Zero	ZF
JNE/JNZ	Jump not Equal or Jump Not Zero	ZF
JA/JNBE	Jump Above or Jump Not Below/Equal	CF, ZF
JAE/JNB	Jump Above/Equal or Jump Not Below	CF
JB/JNAE	Jump Below or Jump Not Above/Equal	CF
JBE/JNA	Jump Below/Equal or Jump Not Above	AF, CF

Hello World Program in ASM:

```
section .text
global main                          ;must be declared for linker (ld)
main:                                ;tells linker entry point
     mov edx,len                     ;message length
     mov ecx,msg                     ;message to write
     mov ebx,1                       ;file descriptor (stdout)
     mov eax,4                       ;system call number (sys_write)
     int 0x80                        ;call kernel
     mov eax,1                       ;system call number (sys_exit)
     int 0x80                        ;call kernel
section .data
msg db 'Hello, world!', 0xa          ;our dear string
len equ $ - msg                      ;leng
```

Compiling and Linking an Assembly Program in NASM

- Make sure you have set the path of nasm and ld binaries in your PATH environment variable. Now take the following steps for compiling and linking the above program:

1. Type the above code using a text editor and save it as hello.asm.

   Make sure that you are in the same directory as where you saved hello.asm.

2. To assemble the program, type

   nasm -f elf hello.asm

If there is any error, you will be prompted about that at this stage. Otherwise an object file of your program named hello.o will be created.

3. To link the object file and create an executable file named hello, type

   ld -m elf_i386 -s -o hello hello.o

4. Execute the program by typing

   ./hello

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