INTRODUCTION TO 80X86 ASSEMBLY LANGUAGE

Module 4
COSC 2329
Sam Houston State University
Dr. Tim McGuire

©1999-2016, Timothy J. McGuire, Ph.D. 1

STRUCTURE OF AN ASSEMBLY LANGUAGE PROGRAM

- Assembly language programs divide roughly into five sections
  - header
  - equates
  - data
  - body
  - closing

©1999-2016, Timothy J. McGuire, Ph.D. 2

THE HEADER

- The header contains various directives which do not produce machine code
- NASM headers are relatively simple (or even empty!)
- Sample header:

```
CPU 386
```

©1999-2016, Timothy J. McGuire, Ph.D. 3

NAMED CONSTANTS

- Symbolic names associated with storage locations represent addresses
- Named constants are symbols created to represent specific values determined by an expression
- Named constants can be numeric or string
- Some named constants can be redefined
- No storage is allocated for these values

©1999-2016, Timothy J. McGuire, Ph.D.

EQUATES

- Constant values are known as *equates*
- Sample equates section:

```
count    equ      10
element  equ      5
%assign   size count * element
%define   mystring "Maze of twisty passages"
%assign  size  0
```

- Cannot change value of **equ** symbol
- You may redefine **%assign** and **%define** symbols
- **%assign** is used for numeric values only
- **equ** expressions are evaluated where used; the others are evaluated where defined

©1999-2016, Timothy J. McGuire, Ph.D.

THE DATA SECTION

- Two types of data, *initialized* and *uninitialized*.
- Initialized portion begins with the directive **section .data**
- Uninitialized portion begins with the directive **section .bss**

©1999-2016, Timothy J. McGuire, Ph.D.

RESERVING SPACE FOR VARIABLES

- Sample data section

```
     section .data
numRows:     db 25
videoBase:   dw 0x0800

     section .bss
numColumns:  resb 1
```

- **db**, **dw**, and **resb** are common directives (**d**efine **b**yte, **d**efine **w**ord, and **res**erve **b**ytes)
- The symbols associated with variables are called *labels*
- Strings may be declared using the **db** directive:

```
aTOm    db "ABCDEFGHIJKLM"
```


RESERVING SPACE FOR VARIABLES

- Similarly, for uninitialized data

```
buffer:     resb  64   ; reserve 64 bytes
wordvar:    resw  1    ; reserve a word
doublevar:  resd  1    ; reserve a doubleword
realarray:  resq  10   ; array of ten quadwords
```


PROGRAM DATA AND STORAGE

- Pseudo-ops to define data or reserve storage
  - **db** - byte(s)
  - **dw** - word(s)
  - **dd** - doubleword(s)
  - **dq** - quadword(s)
  - **dt** - tenbyte(s)
- These directives require one or more operands
  - define memory contents
  - specify amount of storage to reserve for run-time data

DEFINING DATA

- Numeric data values
  - 100 - decimal
  - 100b - binary
  - 100h - hexadecimal
  - '100' - ASCII
  - "100" - ASCII
- Use the appropriate DEFINE directive (byte, word, etc.)
- A list of values may be used - the following creates 4 consecutive words

```
dw 40Ch,10b,-13,0
```

©1999-2016, Timothy J. McGuire, Ph.D. 10

DEFINING DATA

- Some examples (having exactly the same value):

```
200          ; decimal
0200         ; still decimal
0200d        ; explicitly decimal
0c8h         ; hex
$0c8         ; hex again: the 0 is required
0xc8         ; hex yet again
310q         ; octal
11001000b    ; binary
1100_1000b   ; same binary constant
```

©1999-2016, Timothy J. McGuire, Ph.D. 11

NAMING STORAGE LOCATIONS

- Names can be associated with storage locations

```
ANum DB   -4
     DW   17
ONE
UNO  DW   1
X    RESD 1
```

- These names are called variables
- ANum refers to a byte storage location, initialized to **FCh**
- The next word has no associated name
- ONE and UNO refer to the same word
- X is an uninitialized doubleword

©1999-2016, Timothy J. McGuire, Ph.D. 12

ARRAYS

- Any consecutive storage locations of the same size can be called an array

```
X  DW  040Ch,10b,-13,0
Y  DB  'This is an array'
Z  DD  -109236, FFFFFFFFh, -1, 100b
```

- Components of X are at X, X+2, X+4, X+6
- Components of Y are at Y, Y+1, …, Y+15
- Components of Z are at Z, Z+4, Z+8, Z+12

©1999-2016, Timothy J. McGuire, Ph.D. 13

TIMES

- Allows instructions or data to be repeated

```
TIMES 40  RESB 1
TIMES 10h DW  0
TIMES 3   DB  "ABC"
```

(but **RESB 40** is better)

©1999-2016, Timothy J. McGuire, Ph.D. 14

WORD STORAGE

- Word, doubleword, and quadword data are stored in reverse byte order (in memory)
- This is also called "***little endian***" format

```
Directive     Bytes in Storage
DW 256        00 01
DD 1234567h   67 45 23 01
DQ 10         0A 00 00 00 00 00 00 00
X DW 35DAh    DA 35
```

Low byte of X is at X, high byte of X is at X+1

©1999-2016, Timothy J. McGuire, Ph.D. 15

THE PROGRAM BODY

- Also known as the *code segment,* or *text section*
- Divided into four columns: labels, mnemonics, operands, and comments
- Labels refer to the positions of variables and instructions, represented by the *mnemonics*
- Operands are required by most assembly language instructions
- Comments aid in remembering the purpose of various instructions

©1999-2016, Timothy J. McGuire, Ph.D.

AN EXAMPLE

```
Label  Mnemonic Operand    Comment
------------------------------------------------
       org      100h
       SECTION .data
exCode: DW      0          ;A word variable
       SECTION .bss
myByte: RESB    1          ;Uninitialized byte var.
       SECTION .text
       jmp      Exit       ;Jump to Exit label
       mov      cx, 10     ;This line skipped!
Exit:  mov      ah, 4Ch    ;System call: Exit prog
       mov      al,[exCode];Return exit code value
       int      21h        ; call DOS
```

©1999-2016, Timothy J. McGuire, Ph.D.

THE LABEL FIELD

- Labels mark places in a program which other instructions and directives reference
- Label definitions in the code section always end with a colon
- Labels can be from 1 to 4095 characters long and may consist of letters, digits, and the special characters _, $, #, @, ~, and ?
- The only characters which may be used as the first character of an identifier are letters, _ and ?
- A label which begins with a period (.) is a local label (discussed later)
- The assembler is *case sensitive*, that is it makes a difference whether you call your label foo, Foo or FOO

©1999-2016, Timothy J. McGuire, Ph.D.

LEGAL AND ILLEGAL LABELS

- Examples of legal names
  - **COUNTER1**
  - **SUM_OF_DIGITS**
  - **DONE?**
- Examples of illegal names
  - **TWO WORDS** contains a blank
  - **2abc** begins with a digit
  - **YOU&ME** contains an illegal character

©1999-2016, Timothy J. McGuire, Ph.D.

THE MNEMONIC FIELD

- For an instruction, the operation field contains a symbolic operation code (*opcode*)
- The assembler translates a symbolic opcode into a machine language opcode
- Examples are: ADD, MOV, SUB
- In an assembler directive, the operation field contains a directive (*pseudo-op*)
- Pseudo-ops are not translated into machine code; they tell the assembler to do something

©1999-2016, Timothy J. McGuire, Ph.D.

THE OPERAND FIELD

- For an instruction, the operand field specifies the data that are to be acted on by the instruction. May have zero, one, or two operands

```
nop          ;no operands -- does nothing
inc  eax      ;one operand -- adds 1 to the contents of EAX
add  word [WORD1],2 ;two operands -- adds 2 to the contents
                  ; of memory word WORD1
```

- In a two-operand instruction, the first operand is the *destination operand*. The second operand is the *source operand*.
- For an assembler directive, the operand field usually contains more information about the directive.

©1999-2016, Timothy J. McGuire, Ph.D.

THE COMMENT FIELD

- A semicolon marks the beginning of a comment field
- The assembler ignores anything typed after the semicolon on that line
- It is almost impossible to understand an assembly language program without good comments
- Good programming practice dictates a comment on almost every line

©1999-2016, Timothy J. McGuire, Ph.D. 22

GOOD AND BAD COMMENTS

- Don't say something obvious, like

```
MOV   CX,0  ;move 0 to CX
```

- Instead, put the instruction into the context of the program

```
MOV   CX,0  ;CX counts terms, initially 0
```

- An entire line can be a comment, or be used to create visual space in a program

```
;
; Initialize registers
;
MOV   AX,0
MOV   BX,0
```

©1999-2016, Timothy J. McGuire, Ph.D. 23

THE CLOSING

- At the end of the program, control must be passed back to the operating system
- Under DOS, this is done with a system call, as follows:

```
mov ah,4Ch ; The system call for exit (sys_exit)
mov al,0   ; Exit with return code of 0 (no error)
int 21h    ; Call the kernel
```

©1999-2016, Timothy J. McGuire, Ph.D. 24

ASSEMBLING A PROGRAM

- The source file of an assembly language program is usually named with an extension of `.asm`

  `edit myprog.asm`
- The source file is processed (assembled) by the assembler (**NASM**) to produce an object file

  `nasm myprog –o myprog.com`

  produces `myprog.com`
- The machine code can then be run at the command prompt:

  `.\myprog.com`

©1999-2016, Timothy J. McGuire, Ph.D.

DEALING WITH ERRORS

- **NASM** will report the line number and give an error message for each error it finds
- Sometimes it is helpful to have a listing file (`.lst`), created by using **NASM** with the `-l` option

  `nasm myfile.asm -l myfile.lst`
- The `.lst` file contains a complete listing of the program, along with line numbers, object code bytes, and the symbol table

©1999-2016, Timothy J. McGuire, Ph.D.

ENDING A PROGRAM

- All programs, upon termination, must return control back to another program -- the operating system
- This is done by doing a system call

©1999-2016, Timothy J. McGuire, Ph.D.

DATA TRANSFER INSTRUCTIONS

- MOV *destination,source*
  - reg, reg
  - mem, reg
  - reg, mem
  - mem, immed
  - reg, immed
- Sizes of both operands must be the same
- **reg** can be any non-segment register except IP cannot be the target register
- MOV's between a segment register and memory or a 16-bit register are possible

©1999-2016, Timothy J. McGuire, Ph.D.

EXAMPLES

- **`mov ax, [word1]`**
  - "Move **`word1`** to **`ax`**"
  - Contents of register **`ax`** are replaced by the contents of the memory location **`word1`**
  - The brackets specify that the contents of **`word1`** are stored -- **`word1`**==address, **`[word1]`**==contents
- **`mov ah, bl`**
  - copies the contents of **`bl`** to **`ah`**
- **`Illegal: mov  [word1], [word2]`**
  - can't have both operands be memory locations

©1999-2016, Timothy J. McGuire, Ph.D.

SAMPLE MOV INSTRUCTIONS

```
b  db  4Fh
w  dw  2048

mov bl,dh
mov ax,[w]
mov ch,[b]
mov al,255
mov word [w],-100
mov byte [b],0
```

- The **word** and **byte** modifiers are necessary in the last two examples since **b** and **w** only represent addresses and not types

©1999-2016, Timothy J. McGuire, Ph.D.

ADDRESSES WITH DISPLACEMENTS

```
b db 4Fh, 20h, 3Ch
w dw 2048, -100, 0

mov bx, [w+2]
mov [b+1], ah
mov ah, [b+5]
mov dx, [w-3]
```

- Type checking is still in effect
- The assembler computes an address based on the expression
- *NOTE: These are address computations done at assembly time*

  `mov ax, [b-1]`

  *will **not** subtract 1 from the value stored at b*

©1999-2016, Timothy J. McGuire, Ph.D.

EXCHANGE

- XCHG *destination,source*
  - reg, reg
  - reg, mem
  - mem, reg
- MOV and XCHG cannot perform memory to memory moves
- This provides an efficient means to swap the operands
  - No temporary storage is needed
  - Sorting often requires this type of operation
  - This works only with the general registers

©1999-2016, Timothy J. McGuire, Ph.D.

EXAMPLES

- `xchg   ax, [word1]`
  - "Exchange `word1` with `ax`"
  - Contents of register `ax` are replaced by the old contents of memory location `word1` and *vice versa*
- `xchg   ah, bl`
  - Swaps the contents of `ah` and `bl`
- `Illegal: xchg [word1], [word2]`
  - can't have both operands be memory locations

©1999-2016, Timothy J. McGuire, Ph.D.

ARITHMETIC INSTRUCTIONS

ADD *dest, source*
SUB *dest, source*
INC *dest*
DEC *dest*
NEG *dest*

- *Operands must be of the same size*
- *source* can be a general register, memory location, or constant
- *dest* can be a register or memory location
  - except operands cannot both be memory

©1999-2016, Timothy J. McGuire, Ph.D.

ADD AND INC

- ADD is used to add the contents of
  - two registers
  - a register and a memory location
  - a register and a constant
- INC is used to add 1 to the contents of a register or memory location

©1999-2016, Timothy J. McGuire, Ph.D.

EXAMPLES

- **`add ax, [word1]`**
  - "Add `word1` to `ax`"
  - Contents of register `ax` and memory location `word1` are added, and the sum is stored in `ax`
- **`inc ah`**
  - Adds one to the contents of `ah`
- **`Illegal: add [word1], [word2]`**
  - can't have both operands be memory locations

©1999-2016, Timothy J. McGuire, Ph.D.

SUB, DEC, AND NEG

- SUB is used to subtract the contents of
  - one register from another register
  - a register from a memory location, or vice versa
  - a constant from a register
- DEC is used to subtract 1 from the contents of a register or memory location
- NEG is used to negate the contents of a register or memory location

©1999-2016, Timothy J. McGuire, Ph.D. 37

EXAMPLES

- **`sub   ax, [word1]`**
  - "Subtract **`word1`** from **`ax`**"
  - Contents of memory location **`word1`** is subtracted from the contents of register **`ax`**, and the sum is stored in **`ax`**
- **`dec   bx`**
  - Subtracts one from the contents of **`bx`**
- **`Illegal: sub  [byte1], [byte2]`**
  - can't have both operands be memory locations

©1999-2016, Timothy J. McGuire, Ph.D. 38

TYPE AGREEMENT OF OPERANDS

- The operands of two-operand instructions must be of the same type (byte or word)
  - **`mov ax, bh      ;illegal`**
  - **`mov ax, byte [byte1]  ;illegal`**
  - **`mov ax, [byte1] ;legal- moves two bytes into ax`**
  - **`mov ah,'A'      ;legal -- moves 41h into ah`**
  - **`mov ax,'A'      ;legal -- moves 0041h into ax`**

©1999-2016, Timothy J. McGuire, Ph.D. 39

TRANSLATION OF HLL INSTRUCTIONS

- B = A
```
mov  ax, [A]
mov  [B], ax
```
  - memory-to-memory moves are illegal
- A = B - 2*A
```
mov  ax, [B]
sub  ax, [A]
sub  ax, [A]
mov  [A], ax
```

©1999-2016, Timothy J. McGuire, Ph.D.

PROGRAM SEGMENT STRUCTURE

- Data Segments
  - Storage for variables
  - Variable addresses are computed as offsets from start of this segment
- Code Segment
  - contains executable instructions
- Stack Segment
  - used to set aside storage for the stack
  - Stack addresses are computed as offsets into this segment
- Segment directives
```
section .data
section .text
```

©1999-2016, Timothy J. McGuire, Ph.D.

PROGRAM SKELETON

```
section .data
  ;declarations
section .text
  ;main proc code
  ;return to DOS
  ;other procs
```

- Declare variables
- Write code
  - organize into procedures

©1999-2016, Timothy J. McGuire, Ph.D.

INPUT AND OUTPUT USING 8086 ASSEMBLY LANGUAGE

- Most input and output is not done directly via the I/O ports, because
  - port addresses vary among computer models
  - it's much easier to program I/O with the service routines provided by the manufacturer
- There are BIOS routines (which we'll look at later) and DOS routines for handling I/O (using interrupt number 21h)

©1999-2016, Timothy J. McGuire, Ph.D. 43

INTERRUPTS

- The interrupt instruction is used to cause a software interrupt (system call)
  - An interrupt interrupts the current program and executes a subroutine, eventually returning control to the original program
  - Interrupts may be caused by hardware or software
- int *interrupt_number* ;software interrupt

©1999-2016, Timothy J. McGuire, Ph.D. 44

OUTPUT TO MONITOR

- DOS Interrupts : interrupt 21h
  - This interrupt invokes one of many support routines provided by DOS
  - The DOS function is selected via AH
  - Other registers may serve as arguments
- AH = 2, DL = ASCII of character to output
  - Character is displayed at the current cursor position, the cursor is advanced, AL = DL

©1999-2016, Timothy J. McGuire, Ph.D. 45

OUTPUT A STRING

- Interrupt 21h, function 09h
  - DX = offset to the string (in data segment)
  - The string is terminated with the '$' character
- To place the address of a variable in DX, use one of the following
  - lea DX,[theString] ;load effective address
  - mov DX, theString ;immediate data

©1999-2016, Timothy J. McGuire, Ph.D. 46

PRINT STRING EXAMPLE

```
;TITLE "First Program -- HELLO.ASM"
      org   100h
 section   .data
msg    DB      "Hello, World!$"
 section .text
Start:
        mov dx,[msg]       ;get message
       mov ah,09h         ;display string function
       int 21h            ;display message
Exit:  mov ah,4Ch         ;DOS function: Exit program
       mov al,0           ;Return exit code value
       int 21h            ;Call DOS.  Terminate program
```

©1999-2016, Timothy J. McGuire, Ph.D. 47

INPUT A CHARACTER

- Interrupt 21h, function 01h
- Filtered input with echo
  - This function returns the next character in the keyboard buffer (waiting if necessary)
  - The character is echoed to the screen
  - AL will contain the ASCII code of the non-control character
    - AL=0 if a control character was entered

©1999-2016, Timothy J. McGuire, Ph.D. 48

AN EXAMPLE PROGRAM

```
;TITLE "Case Conversion"
 org 100h
 section   .data
MSG1    DB      'Enter a lower case letter:  $'
MSG2    DB      0Dh,0Ah,'In upper case it is: '
CHAR    DB      ' ','$'
exCode  DB      0
 section    .text
;print user prompt
 mov     ah,9            ; display string fcn
 mov     dx, MSG1        ; get first message
 int     21h             ; display it
```

©1999-2016, Timothy J. McGuire, Ph.D.


```
;input a character and convert to upper case
 mov     ah,1            ; read char fcn
 int     21h             ; input char into AL
 sub     al,20h          ; convert to upper case
 mov     [CHAR],al       ;  and store it
;display on the next line
 mov     dx, MSG2        ; get second message
 mov     ah,9            ; display string function
 int     21h             ; display message and upper case
;return to DOS
Exit:
 mov     ah,4Ch          ; DOS function: Exit program
 mov     al,[exCode]     ; Return exit code value
 int     21h             ; Call DOS.  Terminate program
```

©1999-2016, Timothy J. McGuire, Ph.D.