Introduction to NASM

ICS312 - Spring 2009
Machine-Level and Systems Programming

Henri Casanova (henric@hawaii.edu)

Machine code

- Each type of CPU understands its own machine language
- Instructions are numbers that are stored in bytes in memory
- Each instruction has its unique numeric code, called the opcode
- Instruction of x86 processors vary in size
  - Some may be 1 byte, some may be 2 bytes, etc.
- Many instructions include operands as well

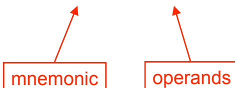
opcode	operands

- Example:
  - On x86 there is an instruction to add the content of EAX to the content of EBX and to store the result back into EAX
  - This instruction is encoded (in hex) as: 03C3
- Clearly, this is not easy to read/remember

Assembly code

- An assembly language program is stored as text
- Each assembly instruction corresponds to exactly one machine instruction
  - Not true of high-level programming languages
  - E.g.: a function call in C corresponds to many, many machine instructions
- The instruction on the previous slides (EAX = EAX + EBX) is written simply as:

```
add    eax, ebx
```



Assembler

- An assembler translates assembly code into machine code
- Assembly code is NOT portable across architectures
  - Different ISAs, different assembly language
- In this course we use the **N**etwide **A**ssembler (NASM) assembler to write 32-bit Assembler
  - You can install it on your own machine
  - You can use my server (give hostname and accounts)
- Note that different assemblers for the same processor may use slightly different syntaxes for the assembly code
  - The processor designers specify machine code, which must be adhered to 100%, but not assembly code syntax

Comments

- Before we learn any assembly, it's important to know how to insert comments into a source file
  - Uncommented assembly is a really, really, really bad idea
  - Comments are important in any language, but for a language as low-level as assembly they are completely necessary
- With NASM comments are added after a '**;**'
- Example:

```
add eax, ebx     ; this is a comment
```

Operands

- Since assembly instructions can have operands, it's important to know what kind of operands are possible
- Register: specifies one of the registers
  - **add eax, ebx**
  - eax = eax + ebx
- Memory: specifies an address in memory.
  - **add eax, [ebx]**
  - eax = eax + content of memory at address ebx
- Immediate: specifies a fixed value (i.e., a number)
  - **add eax, 2**
  - eax = eax + 2
- Implied: not actually encoded in the instruction
  - **inc eax**
  - eax = eax + 1

The move instruction

- This instruction moves data from one location to another
  - mov dest, src
- Note that destination goes first, and the source goes second
- At most one of the operands can be a memory operand
  - mov eax [ebx] ✓
  - mov [eax] ebx ✓
  - mov [eax] [ebx] ✗
- Both operands must be exactly the same size
  - For instance, AX cannot be stored into BL
- This type of "exceptions to the common case" make programming languages difficult to learn and assembly may be the worst offender here
- Examples:
  - mov eax, 3
  - mov bx, ax

Additions, subtractions

- Additions
  - add eax, 4 ; eax = eax + 4
  - add al, ah ; al = al + ah
- Subtractions
  - sub bx, 10 ; bx = bx - 10
  - sub ebx, edi ; ebx = ebx - edi
- Increment, Decrement
  - inc ecx ; ecx++ (a 4-byte operation)
  - dec dl ; dl-- (a 1-byte operation)

Assembly directives

- Most assembler provides "directives", to do some things that are not part of the machine code per se
- Defining immediate constants
  - Say your code always uses the number 100 for a specific thing, say the "size" of an array
  - You can just put this in the NASM code:
    - %define SIZE 100
  - Later on in your code you can just do things like:
    - mov eax, SIZE
- Including files
  - %include "some_file"
- If you know the C preprocessor, these are the same ideas as
  - #define SIZE 100 #include "stdio.h"
- Good idea to use %define whenever possible to avoid "code duplication"

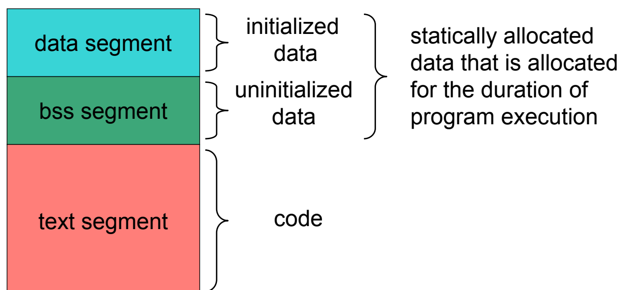
C Driver for Assembly code

- Creating a *whole* program in assembly requires a lot of work
  - e.g., set up all the segment registers correctly
- You will rarely write something in assembly from scratch, but rather only pieces of programs, with the rest of the programs written in higher-level languages like C
- So, in this class we will "call" our assembly code from C
  - The main C function is called a driver
  - Downloadable from the course's Web site

```
int main()     // C driver
{
  int ret_status;
  ret_status = asm_main();
  return ret_status;
}
```

```
...
add eax, ebx
mov ebx, [edi]
...
```

NASM Program Structure



The data and bss segments

- Both segments contains data directives that declare pre-allocated zones of memory
- There are two kinds of data directives
  - **DX directives**: initialized data (D = "defined")
  - **RESX directives**: uninitialized data (RES = "reserved")
- The "X" above refers to the data size:

Unit	Letter(X)	Size in bytes
byte	B	1
word	W	2
double word	D	4
quad word	Q	8
ten bytes	T	10

The DX data directives

- One declares a zone of initialized memory using three elements:
  - Label: the name used in the program to refer to that zone of memory
    - A pointer to the zone of memory, i.e., an address
  - DX, where X is the appropriate letter for the size of the data being declared
  - Initial value, with encoding information
    - default: decimal
    - b: binary
    - h: hexadecimal
    - o: octal

DX Examples

- L1 db 0
  - 1 byte, named L1, initialized to 0
- L2 dw 1000
  - 2-byte word, named L2, initialized to 1000
- L3 db 110101b
  - 1 byte, named L3, initialized to 110101 in binary
- L4 db **0**12h
  - 1 byte, named L4, initialized to 12 in hex (note the '**0**')
- L5 db 17o
  - 1 byte, named L5, initialized to 17 in octal (1*8+7=15 in decimal)
- L6 dd **0**FFFF1A92h (note the '**0**')
  - 4-byte double word, named L6, initialized to FFFF1A92 in hex
- L7 db "A"
  - 1 byte, named L7, initialized to the ASCII code for "A" (65)

ASCII Code

- Associates 1-byte numerical codes to characters
  - Unicode, proposed much later, uses 2 bytes and thus can encode 2^8 more characters (room for all languages, Chinese, Japanese, accents, etc.)
- A few values to know:
  - 'A' is 65d, 'B' is 66d, etc.
  - 'a' is 97d, 'b' is 98d, etc.
  - ' ' is 32d

DX for multiple elements

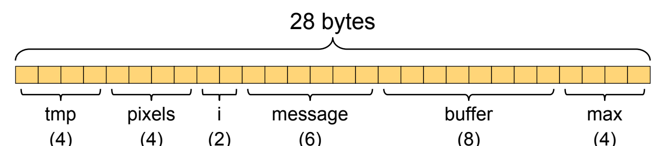
- L8 db 0, 1, 2, 3
  - Defines 4 bytes, initialized to 0, 1, 2 and 3
  - L8 is a pointer to the first byte
- L9 db "w", "o", 'r', 'd', **0**
  - Defines a **null-terminated** string, initialized to "word\0"
  - L9 is a pointer to the beginning of the string
- L10 db "word", 0
  - Equivalent to the above, more convenience

DX with the times qualifier

- Say you want to declare 100 bytes all initialized to 0
- NASM provides a nice shortcut to do this, the "times" qualifier
- L11 times 100 db 0
  - Equivalent to L11 db 0,0,0,....,0 (100 times)

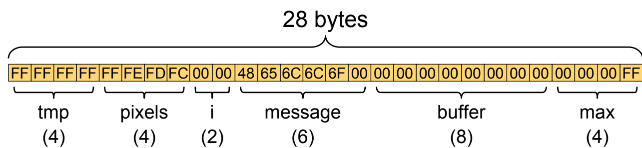
Data segment example

```
tmp       dd     -1
pixels    db     0FFh, 0FEh, 0FDh, 0FCh
i         dw     0
message   db     "H", "e", "l", "l", "o", 0
buffer    times  8    db   0
max       dd     255
```

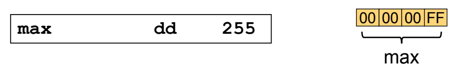


Data segment example

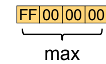
```
tmp       dd     -1
pixels    db     0FFh, 0FEh, 0FDh, 0FCh
i         dw     0
message   db     "H", "e", "l", "l", "o", 0
buffer    times  8    db  0
max       dd     255
```



Endianness?

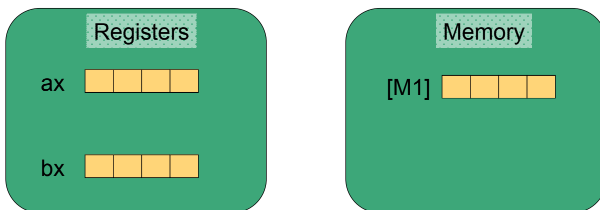


- In the previous slide we showed the above 4-byte memory content for a double-word that contains 255 = 000000FFh
- While this seems to make sense, it turns out that Intel processors do not do this!
  - Yes, the last 4 bytes shown in the previous slide are wrong
- The scheme shown above (i.e., bytes in memory follow the "natural" order): Big Endian
- Instead, Intel processors use Little Endian:



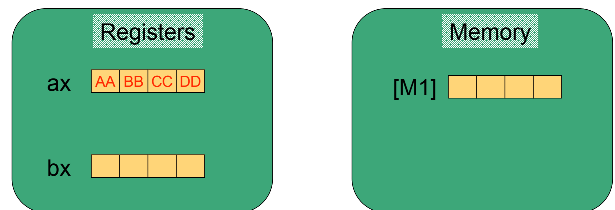
Little Endian

mov ax, 0AABBCCDDh
move [M1], ax
move bx, [M1]



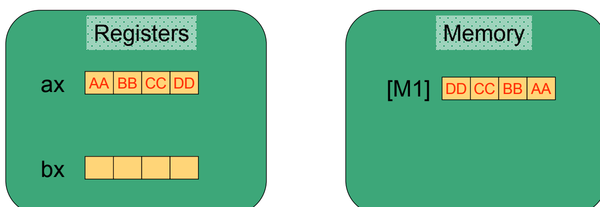
Little Endian

mov ax, 0AABBCCDDh
move [M1], ax
move bx, [M1]



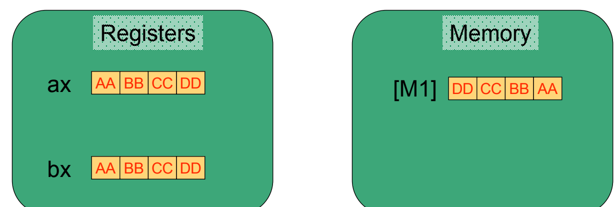
Little Endian

mov ax, 0AABBCCDDh
move [M1], ax
move bx, [M1]



Little Endian

mov ax, 0AABBCCDDh
move [M1], ax
move bx, [M1]



Little/Big Endian

- Motorola and IBM computers use Big Endian
- Intel uses Little Endian (we are using Intel in this class)
- When writing code in a high-level language one rarely cares
  - Although in C one can definitely expose the Endianness of the computer
  - And thus one can write C code that's not portable between an IBM and an Intel!!!
- This only matters when writing multi-byte quantities to memory and reading them differently (e.g., byte per byte)
- When writing assembly code one often does not care, but we'll see several examples when it matters, so it's important to know this *inside out*
- Some processors are configurable (either in hardware or in software) to use either type of endianness (e.g., MIPS processor)

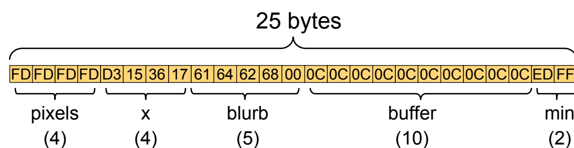
In-class Exercise

```
pixels      times 4     db    0FDh
x           dd    00010111001101100001010111010011b
blurb       db    "a", "d", "b", "h", 0
buffer      times  10   db  14o
min         dw    -19
```

- What is the layout and the content of the data memory segment?
  - Byte per byte, in hex

In-class Exercise

```
pixels      times 4     db    0FDh
x           dd    00010111001101100001010111010011b
blurb       db    "a", "d", "b", "h", 0
buffer      times  10   db  14o
min         dw    -19
```



Uninitialized Data

- The RESX directive is very similar to the DX directive, but *always specifies* the number of memory elements
- L20 resw 100
  - 100 uninitialized 2-byte words
  - L20 is a pointer to the first word
- L21 resb 1
  - 1 uninitialized byte named L21

Use of Labels

- It is important to constantly be aware that when using a label in a program, the label is a pointer, not a value
- Therefore, a common use of the label in the code is as a memory operand, in between square brackets **'[' ']'**
- mov AL, [L1]
  - Move the data at address L1 into register AL
- Question: how does the assembler know how many bits to move?
- Answer: it's up to the programmer to do the right thing, that is load into appropriately sized registers
- Labels do not have a type!
- So although it's tempting to think of them as variables, they are much more limited: just pointers to a byte somewhere in memory

Moving to/from a register

- Say we have the following data segment
  L db 0F0h, 0F1h, 0F2h, 0F3h
- Example: mov AL, [L]
  - AL: Lowest bits of AX, i.e., 1 byte
  - Therefore, value F0 is moved into AL
- Example: mov [L], AX
  - Moves 2 bytes into L, overwriting the first two bytes
- Example: mov [L], EAX
  - Moves 4 bytes into L, overwriting all four bytes
- Example: mov AX, [L]
  - AX: 2 bytes
  - Therefore value F1F0 is moved into AL
  - Note that this is reversed because of Little Endian!!

More About Little Endian

- Consider the following data segment

L1 db 0AAh, 0BBh, 0CCh, 0DDh
L2 dd 0AABBCCDDh

- The instruction: mov eax, [L1] puts DDCCBBAA into eax
  - Note that we're loading 4x1 bytes as a 4-byte quantity
- The instruction: mov eax, [L2] puts AABBCCDD into eax!!!
- When declaring a value in the data segment, that value is declared as it would be appearing in registers when loaded "whole"
  - It would be _really_ confusing to write numbers in little endian mode in the program

Moving immediate values

- Consider the instruction: mov [L], 1
- The assembler will give us an error: "operation size not specified"!
- This is because the assembler has no idea whether we mean for "1" to be 01h, 0001h, 00000001h, etc.
  - Again, labels have no type
- Therefore the assembler provides us with a way to specify the size of immediate operands
- mov dword [L], 1
  - 4-byte double-word
- 5 size specifiers: byte, word, dword, qword, tword

Size Specifier Examples

- mov [L1], 1 ; Error
- mov byte [L1], 1 ; 1 byte
- mov word [L1], 1 ; 2 bytes
- mov dword [L1], 1 ; 4 bytes
- mov [L1], eax ; 4 bytes
- mov [L1], ax ; 2 bytes
- mov [L1], al ; 1 byte
- mov eax, [L1] ; 4 bytes
- mov ax, [L1] ; 2 bytes
- mov ax, 12 ; 2 bytes

Brackets or no Brackets

- mov eax, [L]
  - Puts the content at address L into eax
  - Puts 32 bits of content, because eax is a 32-bit register
- mov eax, L
  - Puts the address L into eax
  - Puts the 32-bit address L into eax
- mov ebx, [eax]
  - Puts the content at address eax (= L) into ebx
- inc eax
  - Increase eax by one
- mov ebx, [eax]
  - Puts the content at address eax (= L + 1) into ebx

Example

```
first     db    00h, 04Fh, 012h, 0A4h
second    dw    165
third     db    "adf"
```

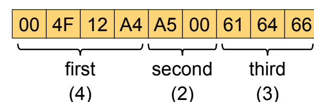
```
mov   eax, first
inc   eax
mov   ebx, [eax]
mov   [second], ebx
mov   byte [third], 11o
```

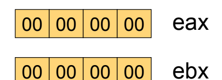
What is the content of "data" memory after the code executes?

Example

```
first     db    00h, 04Fh, 012h, 0A4h
second    dw    165
third     db    "adf"
```

```
mov   eax, first
inc   eax
mov   ebx, [eax]
mov   [second], ebx
mov   byte [third], 11o
```



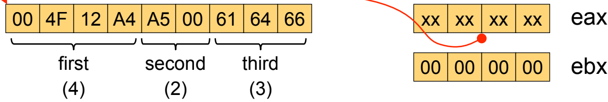


Example

```
first     db    00h, 04Fh, 012h, 0A4h
second    dw    165
third     db    "adf"
```

```
mov   eax, first
inc   eax
mov   ebx, [eax]
mov   [second], ebx
mov   byte [third], 11o
```

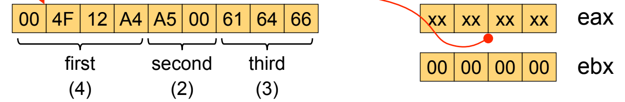
Put an **address** into eax (addresses are 32-bit)



Example

```
first     db    00h, 04Fh, 012h, 0A4h
second    dw    165
third     db    "adf"
```

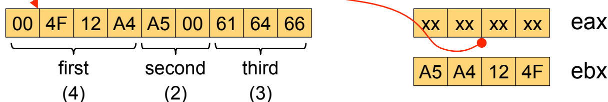
```
mov   eax, first
inc   eax
mov   ebx, [eax]
mov   [second], ebx
mov   byte [third], 11o
```



Example

```
first     db    00h, 04Fh, 012h, 0A4h
second    dw    165
third     db    "adf"
```

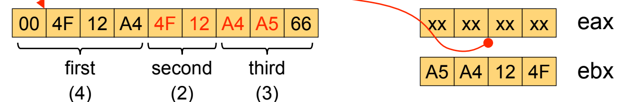
```
mov   eax, first
inc   eax
mov   ebx, [eax]
mov   [second], ebx
mov   byte [third], 11o
```



Example

```
first     db    00h, 04Fh, 012h, 0A4h
second    dw    165
third     db    "adf"
```

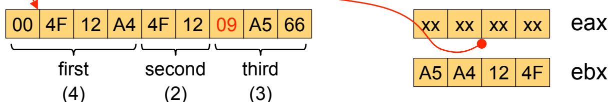
```
mov   eax, first
inc   eax
mov   ebx, [eax]
mov   [second], ebx
mov   byte [third], 11o
```



Example

```
first     db    00h, 04Fh, 012h, 0A4h
second    dw    165
third     db    "adf"
```

```
mov   eax, first
inc   eax
mov   ebx, [eax]
mov   [second], ebx
mov   byte [third], 11o
```

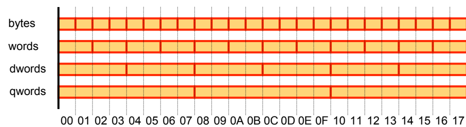


Assembly is Dangerous

- Although the previous example is really a terrible program, it's a good demonstration of how the assembly programmer must be really careful
- For instance, we were able to store 4 bytes into a 2-byte label, thus overwriting the first 2 characters of a string that merely happened to be stored in memory next to that 2-byte label
- Playing such tricks can lead to very clever programs that do things that would be impossible (or very cumbersome) to do with a high-level programming language (e.g., in Java)
- But you really must know what you're doing

x86 Assembly is Dangerous

- Another dangerous thing we did in our assembly program was the use of unaligned memory accesses
  - We stored a 4-byte quantity at some address
  - We incremented the address by 1
  - We read a 4-byte quantity from the incremented address!
  - This really removes all notion of a structured memory
- Some architectures only allow aligned accesses
  - Accessing an X-byte quantity can only be done for an address that's a multiple of X!



In-Class Exercise

- Consider the following program

```
var1        dd     179
var2        db     0A3h, 017h, 012h
var3        db     "bca"
```

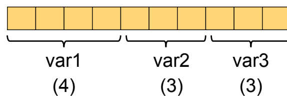
```
mov    eax, var1
add    eax, 3
mov    ebx, [eax]
add    ebx, 5
mov    [var1], ebx
```

- What is the layout of memory starting at address var1?

In-Class Exercise

```
var1        dd     179
var2        db     0A3h, 017h, 012h
var3        db     "bca"
```

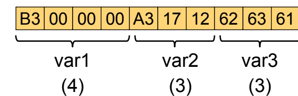
```
mov    eax, var1
add    eax, 3
mov    ebx, [eax]
add    ebx, 5
mov    [var1], ebx
```



In-Class Exercise

```
var1        dd     179
var2        db     0A3h, 017h, 012h
var3        db     "bca"
```

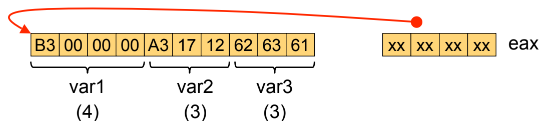
```
mov    eax, var1
add    eax, 3
mov    ebx, [eax]
add    ebx, 5
mov    [var1], ebx
```



In-Class Exercise

```
var1        dd     179
var2        db     0A3h, 017h, 012h
var3        db     "bca"
```

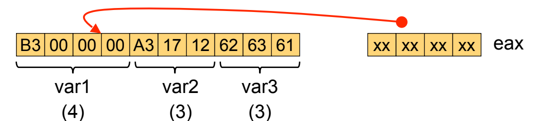
```
mov    eax, var1
add    eax, 3
mov    ebx, [eax]
add    ebx, 5
mov    [var1], ebx
```



In-Class Exercise

```
var1        dd     179
var2        db     0A3h, 017h, 012h
var3        db     "bca"
```

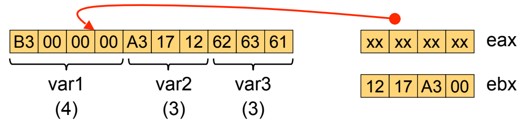
```
mov    eax, var1
add    eax, 3
mov    ebx, [eax]
add    ebx, 5
mov    [var1], ebx
```



In-Class Exercise

```
var1        dd      179
var2        db      0A3h, 017h, 012h
var3        db      "bca"
```

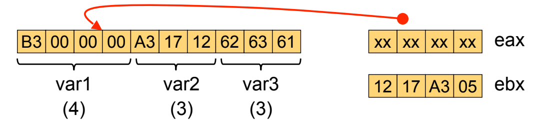
```
mov     eax, var1
add     eax, 3
mov     ebx, [eax]
add     ebx, 5
mov     [var1], ebx
```



In-Class Exercise

```
var1        dd      179
var2        db      0A3h, 017h, 012h
var3        db      "bca"
```

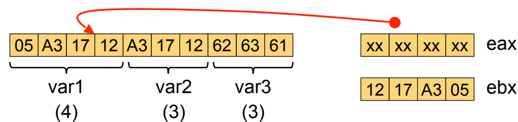
```
mov     eax, var1
add     eax, 3
mov     ebx, [eax]
add     ebx, 5
mov     [var1], ebx
```



In-Class Exercise

```
var1        dd      179
var2        db      0A3h, 017h, 012h
var3        db      "bca"
```

```
mov     eax, var1
add     eax, 3
mov     ebx, [eax]
add     ebx, 5
mov     [var1], ebx
```



Homework #2

- Homework #2 to be posted shortly

NASM Program Structure

```
; include directives

segment .data
  ; DX directives

segment .bss
  ; RESX directives

segment .text
  ; instructions
```

What is in the text segment?

- Remember the C driver:

```
int main()    // C driver
{
  int ret_status;
  ret_status = asm_main();
  return ret_status;
}
```

- The text segment defines the asm_main symbol:

```
    global _asm_main   ; makes the symbol visible
    _asm_main:         ; marks the beginning of the routine
            ; instructions
```

- On Windows, you need the '_' before asm_main although in C the call is simply to "asm_main" not to "_asm_main"
- On Linux you do not need the '_'
- I'll assume Linux from now on (e.g., in the .asm files on the course's Web site)

NASM Program Structure

```
; include directives

segment .data
    ; DX directives

segment .bss
    ; RESX directives

segment .text
        global asm_main
    asm_main:
        ; instructions
```

More on the text segment

- Before and after running the instructions of your program there is a need for some "setup" and "cleanup"
- We'll understand this later, but for now, let's just accept the fact that your text segment will always looks like this:

```
enter       0,0
pusha
;
; Your program here
;
popa
mov     eax, 0
leave
ret
```

NASM Skeleton File

```
    ; include directives

segment .data
    ; DX directives

segment .bss
    ; RESX directives

segment .text
        global asm_main
    asm_main:
        enter   0,0
        pusha
        ; Your program here
        popa
        mov     eax, 0
        leave
        ret
```

Our First Program

- Let's just write a program that adds two 4-byte integers and writes the result to memory
  - Yes, this is boring, but we have to start somewhere
- The two integers are initially in the .data segment, and the result will be written in the .bss segment

Our First Program

```
segment .data
    integer1    dd      15      ; first int
    integer2    dd      6       ; second int
segment .bss
    result resd 1       ; result
segment .text
        global asm_main
    asm_main:
        enter   0,0
        pusha
        mov     eax, [integer1]
        add     eax, [integer2]
        mov     [result], eax
        popa
        mov     eax, 0
        leave
        ret
```

File ics312_first_v0.asm on the Web site

I/O?

- This is all well and good, but it's not very interesting if we can't "see" anything
- We would like to:
  - Be able to provide input to the program
  - Be able to get output from the program
- Also, debugging will be difficult, so it would be nice if we could tell the program to print out all register values, or to print out the content of some zones of memory
- Doing all this requires quite a bit of assembly code and requires techniques that we will not see for a while
- The author of our textbook provides a nice I/O package that we can just use, without understanding how it works for now

asm_io.asm and asm_io.inc

- The "PC Assembly Language" book comes with many add-ons and examples
  - Downloadable from the course's Web site
- A very useful one is the I/O package, which comes as two files:
  - asm_io.asm (assembly code)
  - asm_io.inc (macro code)
- Simple to use:
  - Assemble asm_io.asm into asm_io.o
  - Put "%include asm_io.inc" at the top of your assembly code
  - Link everything together into an executable

Simple I/O

- Say we want to print the result integer in addition to having it stored in memory
- We can use the print_int "macro" provided in asm_io.inc/asm
- This macro prints the content of the eax register, interpreted as an integer
- We invoke print_int as:

  call print_int
- Let's modify our program

Our First Program

```
%include "asm_io.inc"

segment .data
    integer1        dd   15     ; first int
    integer2        dd   6      ; second int
segment .bss
    result  resd        1       ; result
segment .text
        global asm_main
    asm_main:
        enter       0,0
        pusha
        mov         eax, [integer1]
        add         eax, [integer2]
        mov         [result], eax
        call        print_int
        popa
        mov         eax, 0
        leave
        ret
```

File ics312_first_v1.asm on the Web site

How do we run the program?

- Now that we have written our program, say in file ics312_first_v1.asm using a text editor, we need to assemble it
- When we assemble a program we obtain an object file (a .o file)
- We use NASM to produce the .o file:

  % nasm **-f elf** ics312_first_v1.asm **-o** ics312_first_v1.o
- So now we have a .o file, that is a machine code translation of our assembly code
- We also need a .o file for the C driver:

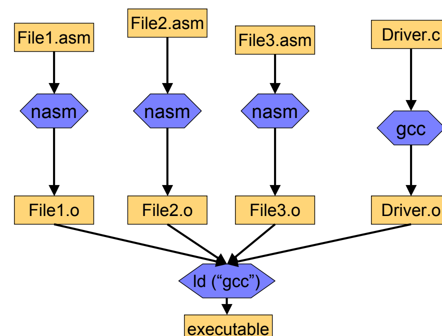
  % gcc -m32 -c driver.c -o driver.o
  - We generate a 32-bit object (my server is 64-bit)
- We also create asm_io.o by assembling asm_io.asm
- Now we have three .o files.
- We link them together to create an executable:

  % gcc driver.o ics312_first_v1.o asm_io.o -o ics312_first_v1
- And voila...

NASM HowTo

- I have create a "how to" page for NASM on the course Web site
- Let's look at it now and compile/run our sample program using a convenient Makefile

The Big Picture



More I/O

- print_char: prints out the character corresponding to the ASCII code stored in AL
- print_string: prints out the content of the string stored at the address stored in eax
  - The string must be null-terminated (last byte = 00)
- print_nl: prints a new line
- read_int: reads an integer from the keyboard and stores it into eax
- read_char: reads a character from the keyboard and stores it into AL
- Let us modify our code so that the two input integers are read from the keyboard, so that there are more convenient messages printed to the screen

Our First Program

```
%include "asm_io.inc"

segment .data
      msg1    db          "Enter a number: ", 0
      msg2    db          "The sum of ", 0
      msg3    db          " and ", 0
      msg4    db          " is: ", 0
segment .bss
      integer1            resd    1     ; first integer
      integer2            resd    1     ; second integer
      result              resd    1     ; result
segment .text
      global asm_main
asm_main:
      enter   0,0
      pusha
      mov     eax, msg1         ; note that this is a pointer!
      call    print_string
      call    read_int          ; read the first integer
      mov     [integer1], eax   ; store it in memory
      mov     eax, msg1         ; note that this is a pointer!
      call    print_string
      call    read_int          ; read the second integer
      mov     [integer2], eax   ; store it in memory
```

```
      mov     eax, [integer1]   ; eax = first integer
      add     eax, [integer2]   ; eax += second integer
      mov     [result], eax     ; store the result
      mov     eax, msg2         ; note that this is a pointer
      call    print_string
      mov     eax, [integer1]   ; note that this is a value
      call    print_int
      mov     eax, msg3         ; note that this is a pointer
      call    print_string
      mov     eax, [integer2]   ; note that this is a value
      call    print_int
      mov     eax, msg4         ; note that this is a pointer
      call    print_string
      mov     eax, [result]     ; note that this is a value
      call    print_int
      call    print_nl
      popa
      mov     eax, 0
      leave
      ret
```

File ics312_first_v2.asm
on the Web site... let's compiler/run it

Our First Program

- In the examples accompanying our textbook there is a very similar example of a first program (called first.asm)
- So, this is great, but what if we had a bug to track?
  - We will see that writing assembly code is very bug-prone
- It would be _very_ cumbersome to rely on print statements to print out all registers, etc.
- So asm_io.inc/asm also provides two convenient macros for debugging!

dum_regs and dump_mem

- The macro dump_regs prints out the bytes stored in all the registers (in hex), as well as the bits in the FLAGS register (only if they are set to 1)

  dump_regs   13

  - '13' above is an arbitrary integer, that can be used to distinguish outputs from multiple calls to dump_regs
- The macro dump_memory prints out the bytes stored in memory (in hex). It takes three arguments:
  - An arbitrary integer for output identification purposes
  - The address at which memory should be displayed
  - The number of 16-byte segments that should be displayed
  - for instance

    dump_mem   29, integer1, 3

  - prints out "29", and then 3*16 bytes

Using dump_regs and dump_mem

- To demonstrate the usage of these two macros, let's just write a program that highlights the fact that the Intel x86 processors use Little Endian encoding
- We will do something ugly using 4 bytes
  - Store a 4-byte hex quantity that corresponds to the ASCII codes: "live"
    - "l" = 6Ch
    - "i" = 69h
    - "v" = 76h
    - "e" = 65h
  - Print that 4-byte quantity as a string

Little-Endian Exposed

```
%include "asm_io.inc"

segment .data
      bytes   dd          06C697665h  ; "live"
      end     db          0           ; null

segment .text
      global asm_main
asm_main:
      enter               0,0
      pusha
      mov                 eax, bytes    ; note that this is an address
      call                print_string  ; print the string at that address
      call                print_nl      ; print a new line
      mov                 eax, [bytes]  ; load the 4-byte value into eax
      dump_mem            0, bytes, 1   ; display the memory
      dump_regs           0             ; display the registers
      pusha
      popa
      mov                 eax, 0
      leave
      ret
```

File
ics312_littleendian.asm
on the site...let's run it

Output of the program

The program prints "evil" and not "live"

The address of "bytes" is 080499AC"

"bytes" starts here

```
evil
Memory Dump # 0 Address = 080499AC
080499A0 00 00 00 00 00 00 00 00 A8 98 04 08 65 76 69 6C "????????????evil"
080499B0 00 00 00 00 25 69 00 25 73 00 52 65 67 69 73 74 "????%i?%s?Regist"
Register Dump # 0
EAX = 6C697665 EBX = 4014EFF4 ECX = BFFFDD60 EDX = 00000001
ESI = 00000000 EDI = 40015CC0 EBP = BFFFDD28 ESP = BFFFDD08
EIP = 0804844D FLAGS = 0286       SF       PF
```

and yes, it's "evil"

The "dump" starts at address 080499A0 (a multiple of 16)

bytes in eax are in the "live" order

Conclusion

- It is paramount for the assembly language programmer to understand the memory layout precisely
- We have seen the basics for creating an assembly language program, assembling it with NASM, linking it with a C driver, and running it
- Time for you to start playing around with the sample programs
- We're now ready to learn how to write more real programs

In-Class Quiz

- Quiz #3 will be on this set of slides
  - Introduction to NASM
- The quiz will be on…