Our First NASM Programs

ICS312

Machine-Level and Systems Programming

Henri Casanova (henric@hawaii.edu)

Comments

- Before we learn assembly, let’s learn how to insert comments into a source file
  - Uncommented code is a bad idea
  - Uncommented assembly code is a really, really bad idea
  - In fact, commenting assembly is necessary
- With NASM, comments are added after a semicolon (‘;’)
  - You may have noted them in the previous module
- Example:

```
add eax, ebx     ; y = y + b
```

Assembly directives

- Most assemblers allow for “directives”, to do things that are not part of the machine code but are convenient
- Defining immediate constants
  - Say your code always uses the number 100 for a specific thing, e.g., the “size” of an array
  - You can just put this in the NASM code:

```
%define SIZE 100
```

  - Later on in your code you can 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   or   #include “some_file”
- Use **`%define`** whenever possible to avoid “code duplication”
  - Because code duplication is evil

NASM Program Structure

```
  ; include directives

segment .data
  ; DX directives

segment .bss
  ; RESX directives

segment .text
  ; instructions
```

NASM Program Structure

```
    ; include directives

segment .data
   ; DX directives

segment .bss
   ; RESX directives

segment .text
   ; instructions
```

- I am using red for pieces of the program you have to write “creatively”
- Other colors are for pieces of the program that you have to write but that are always there and always the same

C Driver for Assembly code

- Creating a *whole* program in assembly requires a lot of work
- You will never 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/C++/whatever
- In this class we “call” assembly code from C
  - We use a main C function as a driver

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

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

So what's in the text segment?

- The text segment defines the asm_main symbol:

```
global    asm_main   ; makes the symbol visible
asm_main:            ; marks the beginning of asm_main
    ; all instructions go here
```

- On Windows, you need a '_' before asm_main, even though 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 all the .asm files on the course's Web site)
  - If you want to do everything on Windows and then retro-fit it on Linux, that's great, but you'll get no help from us
- We can now augment our program a bit…

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”
  - This is so that the C can “call” the assembly correctly
- We’ll understand this later, but for now, let’s just accept the fact that your text segment will always look 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 is written to the .bss segment
- Let's live-code this "from scratch" right now before looking at the next slide…
  - There is a "NASM how to" reading in this module on the course's Web site, which describes all the steps we're about to do in class

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]         ; eax = int1
      add      eax, [integer2]         ; eax = int1 + int2
      mov      [result], eax           ; result = int1 + int2
      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 small 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
- 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`** function provided in asm_io.inc/asm
- This function prints the content of the eax register in base 10, interpreted as a **signed** decimal integer
  - yes, it's very limited: it only prints eax!
- We invoke **`print_int`** as:

```
call      print_int
```

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]   ; eax = int1
        add      eax, [integer2]   ; eax = int1 +   int2
        mov      [result], eax     ; result = int1 + int2
        call     print_int         ; print result
        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
- I used a Makefile a minute ago… but what does it do?
- Assembling a program means building 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
```

- We get a .o file: a machine code translation of our assembly code
- We also need a .o file for the C driver:

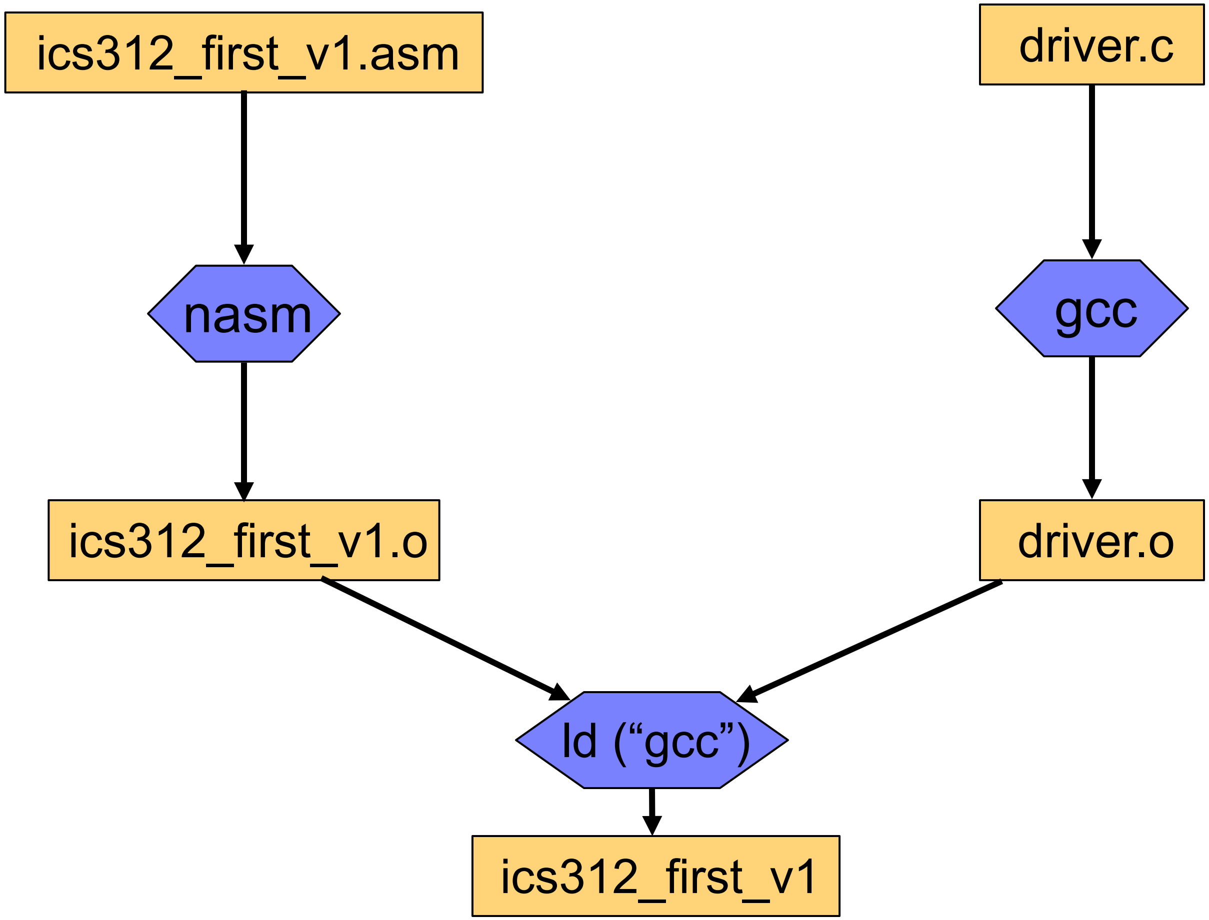
```
gcc -m32 -c driver.c -o driver.o
```

  - We generate a 32-bit object (our machines are all 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:

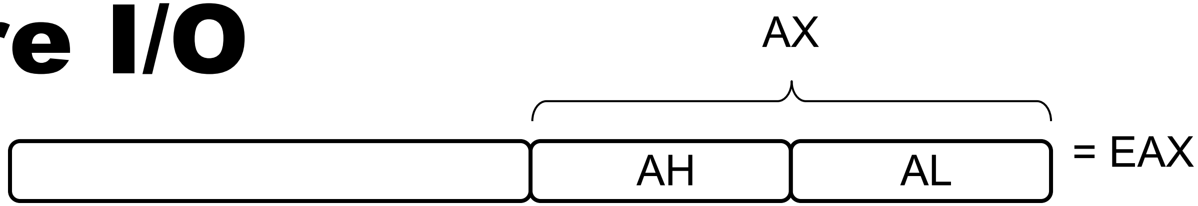
```
gcc driver.o ics312_first_v1.o asm_io.o -o ics312_first_v1
```

- And voila... let’s see it on a picture

The Full Picture



More I/O



- **`print_char`**: prints out the character corresponding to the ASCII code stored in AL
- **`print_string`**: prints out the characters in 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 a **signed** integer from the keyboard and stores it as a 4-byte value into EAX
- **`read_char`**: reads a character from the keyboard and stores its ASCII code into EAX as follows: 00 00 00 xx (AL is the 1-byte ASCII code)
- Let us modify our code so that the two input integers are read from the keyboard, and 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
```

In the examples accompanying our textbook there is a very similar example of a first program (called first.asm)

File ics312_first_v2.asm on the Web site...

Debugging???

- What if we have a bug to track?
  - Initially, assembly code is very bug-prone
- One option: rely on print statements to print out all registers, etc.
  - **This can be a huge waste of time**
  - It's basically like debugging C with print statements, which is not great, BUT with the difference that:
    - Our print statements are very weak
    - Our bugs can be much weirder (or as weird as if you write the most terrible/insane C code)
  - Much easier to actually look at bytes in RAM and registers to figure out bugs
- So asm_io 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 a meaningless integer, that you can use to distinguish outputs from multiple calls to dump_regs
- The macro **`dump_mem`** prints out the bytes stored in memory (in hex). It takes three arguments:
  - A meaningless integer for distinguishing outputs
  - The address at which memory should be displayed
  - The number plus one of 16-byte segments that should be displayed
  - for instance

```
dump_mem           29, integer1, 3
```

  - prints out “29”, and then (3+1)*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
  - Declare in the data segment a 4-byte hex quantity whose bytes are the ASCII codes: "live"
    - "l" = 6Ch, "i" = 69h, "v" = 76h, "e" = 65h
  - Print that 4-byte quantity as if it where a string
  - Then load it into a register
  - Use **`dump_mem`** and **`dump_reg`** to check out byte values
- Let's do it live again…

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 Web site

Output of the program

The program prints "evil" and not "live"

The address of "bytes" is 0804C028"

"bytes" starts here

```
evil
Memory Dump # 0 Address = 0804C028
0804C020 00 00 00 00 00 00 00 00 65 76 69 6C 00 00 00 00 "????????evil????"
0804C030 25 69 00 25 73 00 52 65 67 69 73 74 65 72 20 44 "%i?%s?Register D"
Register Dump # 0
EAX = 6C697665 EBX = 0804C000 ECX = E92F7D6B EDX = 40800DC0
ESI = 40800E54 EDI = 3FFFEB80 EBP = 40800D78 ESP = 40800D58
EIP = 080491D4 FLAGS = 0216                AF PF
```

and yes, it's "evil"

The "dump" starts at address 0804A020 (a multiple of 16)

bytes in eax are in the "live" order

Word of Caution

- Each time I teach assembly I says "using print for debugging is a waste of time" and "debugging is done by looking at registers and RAM using **`dump_regs`** and **`dump_mem`**"
- Each time, a large fraction of students strongly resist this, desperately clinging to the delusion that we're using a high-level language with variables
  - But even for high-level languages using print statements can be pretty limited for debugging!
- You will save hours if you let go of that delusion (and you will learn a lot and feel empowered in the process)
  - And you will avoid the "I spent 4 hours tracking a bug, then at office hours the prof/TA added a **`dump_mem`** statement and found it in 10 seconds" psychologically damaging ordeal

Thoughts on the Previous Module

- Now that we have **`dump_mem`**, you have an automatic validity checker for all the practice problems and sample homework from the previous module…
- So if you're wondering how some tweaks of the declarations impact the byte order in RAM, just try it
- Every semester somebody asks at least one assembly programming question I don't know the answer to, and I'll just "try it" as well

Example program

- Say that we want to write a program that prompts the user for two characters, say X and Y, and then prints the string “F(X,X,YYY)”
  - Yes, it’s totally useless and arbitrary, but we have to start somewhere
- Let’s live-code it in class
  - And perhaps discover something about reading characters from the keyboard!!
  - And then try to do it without using print_char

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
  - We have seen some convenient macros/functions provided by the textbook author

- As you know, all homework assignments are optional
- BUT, as I said at the beginning of the semester, for most of you, not attempting to do the homework assignments at all will make it hard to absorb the content

- And so, let’s look at optional Homework #3
  - Which is very much like the live-coding we did a few minutes ago
- Remember that MIDTERM #1 is coming up