CPSC 232
Intro to Assembly Language Programming

Carter: Section 1.4, Chapter 2

The First NASM Program

- We're going to "cheat" a little bit

- Setting up a program's activation record is a little difficult
  - Will use a compiled C program that calls the assembly that we write
  - The C code for the driver program is:

```
int main() {
  int ret_status;
  ret status = asm_main();
  return ret_status;
}
```

- This code:
  - Creates activation record (stack frame, etc.), memory segments, etc.
  - Enables us to use C's libraries

NASM Programs

- Typically several sections (segments) within the program:
  - Recall: Protected Mode in Intel chipset
  - .data – Initialized data goes here
  - .bss – Uninitialized data goes here
  - .text – Program code goes here

- Within each section, the program can define labels for things:
  - Variables
  - Strings
  - Named sections of code

NASM Programs: Template

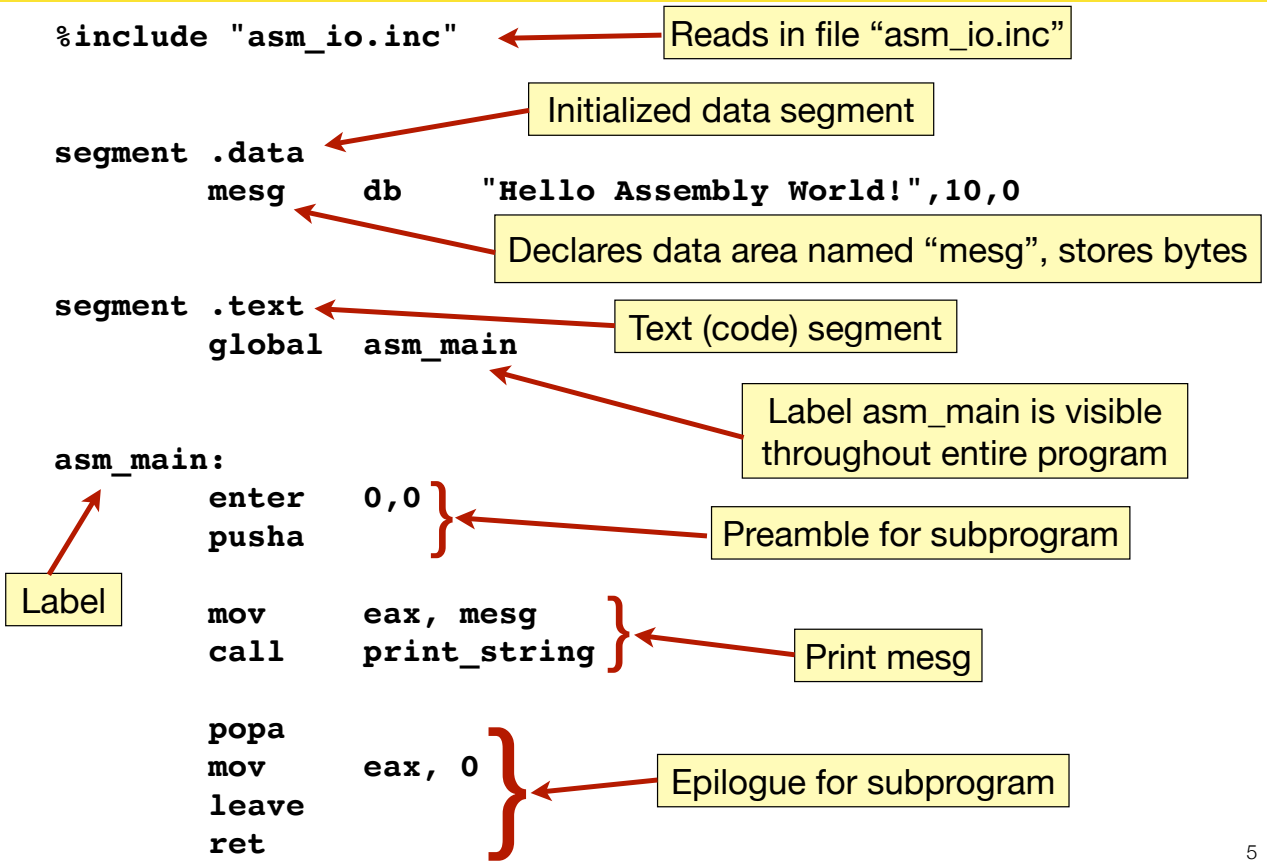
```
; file: skel.asm
; This file is a skeleton that can be used to start assembly programs.

%include "asm_io.inc"
segment .data
; initialized data is put in the data segment here

segment .bss
; uninitialized data is put in the bss segment

segment .text
        global  asm_main
asm_main:
        enter   0,0               ; setup routine
        pusha
; code is put in the text segment. Do not modify the code before
; or after this comment.
        popa
        mov     eax, 0            ; return back to C
        leave
        ret
```

Hello World in Assembly



First NASM Program: first.asm

```
; file: first.asm
; First assembly program. This program asks for two integers as
; input and prints out their sum.
;
; To create executable:
; Using djgpp:
; nasm -f coff first.asm
; gcc -o first first.o driver.c asm_io.o
;
; Using Linux and gcc:
; nasm -f elf first.asm
; gcc -o first first.o driver.c asm_io.o
;
; Using Borland C/C++
; nasm -f obj first.asm
; bcc32 first.obj driver.c asm_io.obj
;
; Using MS C/C++
; nasm -f win32 first.asm
; cl first.obj driver.c asm_io.obj
;
; Using Open Watcom
; nasm -f obj first.asm
; wcl386 first.obj driver.c asm_io.obj
```

First NASM Program:

```
%include "asm_io.inc"
;
; initialized data is put in the .data segment
;
segment .data
;
; These labels refer to strings used for output
;
prompt1 db    "Enter a number: ", 0       ; don't forget nul terminator
prompt2 db    "Enter another number: ", 0
outmsg1 db    "You entered ", 0
outmsg2 db    " and ", 0
outmsg3 db    ", the sum of these is ", 0

;
; uninitialized data is put in the .bss segment
;
segment .bss
;
; These labels refer to double words used to store the inputs
;
input1  resd 1
input2  resd 1
```


First NASM Program:

```
;
; code is put in the .text segment
;
segment .text
        global  asm_main
asm_main:
        enter   0,0               ; setup routine
        pusha

        mov     eax, prompt1      ; print out prompt
        call    print_string

        call    read_int          ; read integer
        mov     [input1], eax     ; store into input1

        mov     eax, prompt2      ; print out prompt
        call    print_string

        call    read_int          ; read integer
        mov     [input2], eax     ; store into input2
```

First NASM Program:

```
        mov     eax, [input1]      ; eax = dword at input1
        add     eax, [input2]      ; eax += dword at input2
        mov     ebx, eax           ; ebx = eax
        dump_regs 1                ; dump out register values
        dump_mem 2, outmsg1, 1     ; dump out memory
;
; next print out result message as series of steps
;
        mov     eax, outmsg1
        call    print_string       ; print out first message
        mov     eax, [input1]
        call    print_int          ; print out input1
        mov     eax, outmsg2
        call    print_string       ; print out second message
        mov     eax, [input2]
        call    print_int          ; print out input2
        mov     eax, outmsg3
        call    print_string       ; print out third message
        mov     eax, ebx
        call    print_int          ; print out sum (ebx)
        call    print_nl           ; print new-line
```


First NASM Program:

```
        popa
        mov     eax, 0             ; return back to C
        leave
        ret
```

NASM Segment Locations

```
; file: skel.asm
; This file is a skeleton that can be used to start assembly programs.

%include "asm_io.inc"
segment .data
; initialized data is put in the data segment here

segment .bss
; uninitialized data is put in the bss segment

segment .text
        global  asm_main
asm_main:
        enter   0,0               ; setup routine
        pusha
; code is put in the text segment. Do not modify the code before
; or after this comment.
        popa
        mov     eax, 0            ; return back to C
        leave
        ret
```

11

NASM – Data Section

- Declaring data storage locations
  - Initialized data will start with d (define):

Name	Size	Example
db	byte	db 0xae
dw	word (2-byte)	dw ‘ab’
dd	double (4-byte)	dd 1.234567e20
dq	quad word (8-byte)	dq 0x123456789abcdef0
dt	ten-bytes	dt 1.234567e20
do	octo-word (16-byte)	do 0x112233445566778899aabbccddeeff0
ddq	doublequad (16-byte)	ddq 0x11223344556677889aabbccdd
dy	YMM ref (256 bytes)	dy ... (too big!)

12

NASM – Data Section

- Declaring data storage locations
  - Uninitialized data will start with res (reserve):
  - Reserving space, but not giving it a value

Name	Size	Example
resb	byte	`buffer resb 128`
resw	word (2-byte)	`wordVar resw 1`
resd	double (4-byte)	`dVar resd 1`
resq	quad word (8-byte)	`realArray resq 10`
rest	ten-bytes	`tArray rest 20`
resdq	doublequad (16-byte)	`dqVar resdq 1`
reso	octo-word (16-byte)	`oVar reso 1`
resy	YMM reg (256 byte)	`ymmVar resy 1`

Numbers, Numbers, Gimme a Break!

- It's not enough to know storage types & names

- Need to understand endian-ness!
  - How is the value stored in memory?
  - Big Endian:
    - Stored the "normal" way that we would think
    - Example: the double 0x12345678 is stored in memory as 12 34 56 78
    - Typically used on SPARC, Motorolla, & IBM processors (mostly RISC)
  - Little Endian
    - Stored kinda "backwards" (Least significant byte first)
    - Example: the double 0x12345678 is stored in memory as 78 56 34 12
    - Typically used on Intel processors

- May only need to deal with this when:
  - Transferring data between different endian arch's
  - Stored multi-byte value to memory and reading/working with single bytes at a time

Types of Operations

- **Data Operations:**
  - Data Movement (move data around): **`mov dest, source`**
    - Memory to register
    - Register to memory
    - Register to register
  - **Arithmetic Operations:**
    - Add/Subtract
    - Multiply/Divide
    - Compare
    - Shifts, Sets
  - **Logic Operations:**
    - AND, OR, NOT, etc
    - Shifts
  - Array/String (load, store, repetition)
- **Control Operations (Alter the flow of control): `jmp target`**
  - Jump
  - Loop

15

Moving Number/Data Around

- Moving data
  - May cause headaches
  - May be no problem

} Ok, it depends on what you are doing!

- Move from large to small (e.g., word to byte)
  - Simple – just truncate value!
  - Warning: May lose information if destination is too small
  - Example: move 0xFFFF to byte storage
    - Signed? Becomes 0xFF (-1) is the same as 0xFFFF (-1)
    - Unsigned? Becomes 0xFF (255) is NOT the same as 0xFFFF (16535)!
- Move from small to large
  - Need to know if original is signed or unsigned!
  - Example: move 0xFF to word storage
    - Signed: (must sign extend) becomes 0xFFFF
    - Unsigned: zero-out high bits, becomes 0x00FF
  - Select appropriate instruction based on signedness:
    - **mov** – equal size operands only
    - **movzx** – zero out high bits & “drop” smaller value to larger register (low bits)
    - **movsx** – sign extend smaller value to larger register

16

Data Movement

- Most often, will use the “move” instruction
  - Move data between/among registers & memory:

```
mov   eax, ecx    ; move contents of ecx to eax
mov   eax, 15     ; move immediate value 15 into eax
mov   edx, [ecx]  ; move memory contents of address
                  ; stored in ecx (pointer) into edx
```

- Load & Store
  - For array/string data
  - Specialized to repeatedly load/store array/string data (one character at a time)
  - We will examine this later. . .

Data Movement

- Register/Stack movement
  - Push registers onto stack:

```
push   eax    ; push contents of eax onto the stack
pusha         ; push all registers onto the stack
              ; (EAX, EBX, ECX, EDX, ESI, EDI, EBP)
```

  - Pop stack into registers:

```
pop    eax    ; pop stack top into eax
popa          ; pop stack tops into all registers
              ; (EAX, EBX, ECX, EDX, ESI, EDI, EBP)
```

- XCHG
  - Exchange/swap values between registers/memory

```
xchg   bl, bh ; swap contents of bl and bh
              ; (will swap bytes of bx)

xchg   eax, ebx ; swap contents of eax and ebx
```

Arithmetic

- Add/Sub take source & destination
  - Can be register, memory, or immediate
  - At least one operand must be register (no memory to memory)

```
add   eax, ecx     ; EAX = EAX + ECX
add   ecx, 21      ; ECX = ECX + 21
add   eax, [ecx]   ; EAX = EAX + contents of memory
                   ; stored at address [ecx]
```

- Most forms of mul, div have implied argument: EAX

- CMP
  - Compare two values & set condition codes (examine those later)

- SETcc
  - Set byte of register/memory depending upon condition code

Common Opcodes: Multiplication

- Unsigned Multiplication:

```
mul  source
```

  - Source can be register or memory address
  - In all cases, dest is A(L/X)
    - If source is 8-bit: mult AL by source & store in AX
    - If source is 16-bit: mult AX by source & store in DX:AX
    - If source is 32-bit: mult EAX by source & store in EDX:EAX

- Signed Multiplication:

```
imul  source
imul  dest,  src1 [,  src2]
```

- Destination register: Any general purpose register (or implied EAX, also)
  - Source1 can be register, memory, or immediate
  - Source2 is an immediate value

imul

`imul source`

`imul  dest,  src1 [,  src2]`

dest	source1	source2	Action
	reg/mem8		AX = AL*source1
	reg/mem16		DX:AX = AX*source1
	reg/mem32		EDX:EAX = EAX*source1
reg16	reg/mem16		dest *= source1
reg32	reg/mem32		dest *= source1
reg16	immed8		dest *= immed8
reg32	immed8		dest *= immed8
reg16	immed16		dest *= immed16
reg32	immed32		dest *= immed32
reg16	reg/mem16	immed8	dest = source1*source2
reg32	reg/mem32	immed8	dest = source1*source2
reg16	reg/mem16	immed16	dest = source1*source2
reg32	reg/mem32	immed32	dest = source1*source2

Common Opcodes: Division

- Works similar to mul (in terms of operands, etc.)

  `div  source`

  - Unsigned division
  - 8-bit source:
    - Div AX by source
    - Store quotient in AL, remainder in AH
  - 16-bit source:
    - Div DX:AX by source
    - Store quotient in AX, remain in DX
  - If source is 32-bit:
    - Div EDX:EAX by source
    - Store quotient in EAX & remain in EDX

- idiv only has one form:

  `idiv  source`

  - Signed division
  - Source can be register or memory
  - Again, divide (e)ax by source

Comparison

- In assembler, use compare instruction:

```
cmp   src1,   src2
```

  - This sets a compare flag
    - Unsigned: zero flag (ZF), carry flag (CF)
      compute src1 - src2, if result is:
      - zero (values are equal) – ZF is set
      - positive (src1>src2) – no flags are set
      - negative (src1<src2) – CF is set
    - Signed: zero flag (ZF), overflow flag (OF), sign flag (SF)
      compute src1 - src2, if result is
      - zero (equal) – ZF is set
      - positive (src1>src2) – ZF is not set & OF == SF
      - negative (src1<src2) – ZF is not set & OF != SF

Set

- Sets register/memory byte to zero/one based on condition code:

```
setg    ah          ; AH = one if GREATER flag set, zero otherwise
setc    bl          ; BL = one if CARRY flag set, zero otherwise
```

- Condition Codes (Flags):
  - Carry/No Carry (C/NC)
  - Zero/Not Zero (Z/NZ)
  - Sign/No Sign (S/NS)
  - Overflow/No Overflow (O/NO)
  - Parity/No Parity (P/NP)
  - Equal/Not Equal (E/NE)
  - Greater/Not Greater (G/NG)
  - Less/Not Less (L/NL)
  - Greater than or Equal/Not Greater than or Equal (GE/NGE)
  - Less than or Equal/Not Less than or Equal (LE/NLE)

- Requires either:
  - Comparison operation before set
  - Operation that sets condition code (add, multiply, etc) before set

Logic

- Logical AND, OR, NOT, XOR

- AND, OR, XOR
  - Operands can be register, memory, immediate

```
and    ecx, eax     ; ECX = bitwise AND of EAX, ECX
XOR    bx, ax       ; BX = bitwise XOR of AX, BX
OR    bx, 0FFFFH    ; BX = bitwise OR of BX, 0xFFFF
```

- TEST
  - Similar to AND, but **does not store results**
  - Only sets condition flags

- NOT
  - Operands can be register or memory

```
not    eax          ; EAX = NOT EAX
```


Control Structures

- All those elegant, high-level **`if-then-else`**, **`do-while`**, **`for`**, etc.
  - Implemented at low level by branch-compare, jumps, & gotos
  - It is possible to design very poorly structured code or very clean assembly
    - Recall that we discourage use of “gotos” in structured programs
      - Why, because this leads to code that is difficult to understand & test
    - We prefer proper loops, and method/procedure calls

- Typical process
  - Compare two values
  - Depending upon comparison result, transfer control to some other location in code

- Sounds like high-level code?!
  - But we do with cmp & some flavor of jump!

Branch Instructions

- Two “flavors” of branch:
  - Conditional (may or may not be taken)
    - **jcc** (jump based on some condition code)
  - Unconditional (always taken) – think “goto”
    - **jmp**
  - Three address modes:
    - Short: 128 bytes away
      (example: **`jmp short <target>`**)
    - Near: 32K bytes away
      (example: **`jmp <target>`**)
      (or anywhere within segment: jmp word <target>
    - Far: Jump to other segment
      (example: **`jmp far <target>`**)

Conditional Branches

Equal, Not, Less-than, Greater

```
        cmp     eax, 5
        jge     thenblock
        mov     ebx, 2
        jmp     next
thenblock:
        mov     ebx, 1
next:
```

JZ	branches only if ZF is set
JNZ	branches only if ZF is unset
JO	branches only if OF is set
JNO	branches only if OF is unset
JS	branches only if SF is set
JNS	branches only if SF is unset
JC	branches only if CF is set
JNC	branches only if CF is unset
JP	branches only if PF is set
JNP	branches only if PF is unset

Equal, Not, Below, Above

Signed		Unsigned	
JE	branches if `vleft = vright`	JE	branches if `vleft = vright`
JNE	branches if `vleft` $\neq$ `vright`	JNE	branches if `vleft` $\neq$ `vright`
JL, JNGE	branches if `vleft < vright`	JB, JNAE	branches if `vleft < vright`
JLE, JNG	branches if `vleft` $\leq$ `vright`	JBE, JNA	branches if `vleft` $\leq$ `vright`
JG, JNLE	branches if `vleft > vright`	JA, JNBE	branches if `vleft > vright`
JGE, JNL	branches if `vleft` $\geq$ `vright`	JAE, JNB	branches if `vleft` $\geq$ `vright`