Introduction to 64 Bit Intel Assembly Language Programming

Edited from the work of Ray Seyfarth

July 23, 2015
Goals for Cos 284

- Learn internal data formats
- Learn basic 64 bit Intel/AMD instructions
- Write pure assembly programs
- Write mixed C and assembly programs
- Use the gdb debugger for ASM
- Floating point instructions
- Arrays
- Functions
- Structs
- Using system calls and C libraries
- Data structures and high performance ASM
Problems with assembly language

- Assembly is the poster child for non-portability
  - Different CPU = different assembly
  - Different OS = different function ABI (application binary interface)
  - Intel/AMD CPUs operate in 16, 32 and 64 bit modes
- Difficult to program
  - More time = more money
  - Less reliable
  - Difficult to maintain
- Syntax does not resemble mathematics
- No syntactic protection
  - No structured ifs, loops
- No typed variables
  - Can use a pointer as a floating point number
  - Can load a 4 byte integer from a double variable
- Variable access is roughly like using pointers

© Edited from work of Ray Seyfarth
What’s good about assembly language?

- **Assembly language is fast**
  - Optimizing C/C++ compilers will be faster than a novice most of the time.
  - You need to dissect an algorithm and rearrange it to use a special feature that the compiler can’t figure out
  - Generally you must use a special instructions
  - There are over 1000 instructions
  - Still it can be faster

- **Assembly programs are small**
  - But memory is cheap and plentiful
  - C/C++ compilers can optimize for size
  - Compilers can re-order code sections to reduce size

- **Assembly can do things not possible in C/C++**
  - I/O instructions
  - Manage memory mapping registers
  - Manipulate other internal control registers
What’s good about assembly for ordinary mortals?

- Teaches you how the programs really works
- Teaches you how storage and arithmetic is done in registers
- Teaches you C/C++ function register and stack usages
- Teaches you how stack frames are built and destroyed.
- Optimization techniques are explained.
- Computer bugs are more immediately related to machine instructions and limitations
- You will learn how the compiler implements
  - if/else statements
  - loops
  - functions
  - structures
  - arrays
  - recursion
- Your coding will improve.
Generation of languages

- First generation - machine language
- Second generation - assembly language
  - Names for instructions
  - Names for variables
  - Names for locations of instructions
  - Perhaps with macros - code replacement
- Third generation - not machine instructions
  - Modeled after mathematics - Fortran
  - Modeled after English - Cobol
  - List processing - Lisp
- Fourth generation - domain specific
  - SQL
- Fifth generation - describe problem, computer generates algorithm
  - Prolog
Assembly example

; Program: exit
;
; Executes the exit system call
;
; No input
;
; Output: only the exit status ($? in the shell)
;
segment .text
global _start

_start:
  mov   eax,1   ; 1 is the exit syscall number
  mov   ebx,5   ; the status value to return
  int    0x80   ; execute a system call

©Edited from work of Ray Seyfarth
Assembly syntax

- ; starts comments
- Labels are strings which are not instructions
  - Usually start in column 1
  - Can end with a colon to avoid confusion with instructions
- Instructions can be machine instructions or assembler instructions
  - mov and int are machine instructions or opcodes
  - segment and global are assembler instructions or pseudo-ops
- Instructions can have operands
  - here: mov eax, 1
  - Here is a label for the instruction
  - mov is an opcode
  - eax and 1 are operands
Some assembler instructions

- section or segment define a part of the program
  - .text is where instructions go for Linux
- global defines a label to be used by the linker
- global _start makes _start a global label
- _start or main is where a program starts
  - _start is more basic
  - main is called (perhaps indirectly) by _start
Assembling the exit program

- `yasm -f elf64 -g dwarf2 -l exit.lst exit.asm`
- `-f elf64` says we want a 64 bit object file (elf=extensible linking format)
- `-g dwarf2` says we want `dwarf2` debugging info (why `dwarf`?)
  - `dwarf2` works pretty well with the `gdb` debugger
- `-l exit.lst` asks for a listing in `exit.lst`
- `yasm` will produce `exit.o`, an object file
  - machine instructions not ready to execute
%line 1+1 exit.asm

[segment .text]
[global _start]

_start:

00000000 B801000000 mov eax,1
00000005 BB05000000 mov ebx,5
0000000A CD80 int 0x80
Linking means combining object files to make an executable file.

For programs with _start:
- `ld -o exit exit.o`
- Builds a file named exit
- Default is a.out

For programs with main:
- `gcc -o exit exit.o`
- Gets default _start function from the C library

./exit to run the program
Floating point numbers

Consider 1.75, in 32bit-IEEE 754 the number becomes:

0 01111111 1100000000000000000000000
Positive 127  (1).75

Grouping into 4 bit nibbles:

0011 1111 1110 0000 0000 0000 0000 0000
3 f e 0 0 0 0 0

But this is stored reversed and with each nibble pair swapped:

0 0 0 0 0 e 0 3 f
Consider the following asm file “fp.asm”.

1 segment .data
2 zero dd 0.0
3 one dd 1.0
4 neg1 dd -1.0
5 a dd 1.75
6 b dd 122.5
7 d dd 1.1
8 e dd 10000000000.0

The **dd** command specifies a double word data item. A word is 2 bytes. So a double word is 32 bits.

- **dw** is a data word
- **db** is a byte
- **dq** is a data quad-word
Now if we create the file listing using:

```
yasm -f elf64 -g dwarf2 -l fp.lst fp.asm
```

The result is:

```
1 %line 1+1 fp.asm
2 [section .data]
3 00000000 00000000 zero dd 0.0
4 00000004 0000803F one dd 1.0
5 00000008 000080BF neg1 dd -1.0
6 0000000C 0000E03F a dd 1.75
7 00000010 0000F542 b dd 122.5
8 00000014 CDCC8C3F d dd 1.1
9 00000018 F9021550 e dd 10000000000.0
```
Memory mapping

- Computer memory is an array of bytes from 0 to \( n - 1 \) where \( n \) is the memory size.
- Programs perceive “logical” addresses which are mapped to physical addresses.
- 2 people can run a program starting at logical address 0x4004c8 while using different physical memory.
- CPU translates logical addresses to physical during instruction execution.
- The CPU translation can be just as fast as if the software used physical addresses.
- The x86-64 CPUs can map pages of sizes 4096 bytes and 2 megabytes.
- Linux uses 2 MB pages for the kernel and 4 KB pages for programs.
- Some recent CPUs support 1 GB pages.
Suppose an instruction references address 0x43215628
With 4 KB pages, the rightmost 12 bits are an offset into a page
With 0x43215628 the page offset is 0x628
The page number is 0x43215
Let's assume that the computer is set up to translate page 0x43215 to physical addresses 0x7893000 - 0x7893fff
Then address 0x43215628 is mapped to 0x7893628
Benefits of memory mapping

- User processes are protected from each other
  - Your process can’t read my process’s data
  - Your process can’t write my data
- The operating system is protected from malicious or errant code
- It is easy for the operating system to give processes contiguous chunks of “logical” memory
Why study memory mapping?

- If you write programs, the mapping is automatic
- We will not discuss instructions for changing mapping tables
- So what difference does it make?
- It helps explain page faults
  - Suppose you allocate an array of 256 bytes at logical address 0x45678200
  - Then all addresses from 0x45678000 to 0x45678fff are valid
  - You can go well past the end of the array before you can get a segmentation violation
- Knowledge is power!
A Linux process has 4 logical segments

- **text**: machine instructions
- **data**: static data initialized when the program starts
- **heap**: data allocated by malloc or new
- **stack**: run-time stack
  - return addresses
  - some function parameters
  - local variables for functions
  - space for temporaries

In reality it is more complex

- 131TB is 47 bits of all 1’s
- CPU could use 48 bit logical addresses

Edited from work of Ray Seyfarth
The text segment is named `.text` in `yasm`
  - `_start` and `main` are not actually at 0
  - The text segment does not need to grow, so the data segment can be placed immediately after it

The data segment is in 2 parts
  - `.data` which contains initialized data
  - `.bss` which contains reserved data (initialized to 0)
  - “bss” stands for “Block Started by Symbol”

The heap and the stack both need to grow
  - The heap grows up
  - The stack grows down
  - They meet in the middle and explode
Stack segment limits

- The stack segment is limited by the Linux kernel
- The typical size is 16 MB for 64 bit Linux
- This can be inspected using “ulimit -a” or “ulimit -s’
- 16 MB seems fairly small, but it is fine until you start using large arrays as local variables in functions
- The stack address range is 0x7fffffff000000 to 0x7fffffffffffffff
- A fault to addresses in this range are recognized by the kernel to allow the stack to grow as needed
Memory example source code

    segment .data

  a  dd    4
  b  dd    4.4
  c  times 10 dd 0
  d  dw    1, 2
  e  db    0xfb
  f  db    "hello world", 0

    segment .bss

  g  resend 1
  h  resend 10
  i  resb    100
Memory example source code (2)

```assembly
segment .text

global main ; let the linker know about main

main:
push rbp ; set up a stack frame for main
mov rbp, rsp ; set rbp to point to the stack frame
sub rsp, 16 ; leave some room for local variables
            ; leave rsp on a 16 byte boundary
xor eax, eax ; set rax to 0 for return value
leave ; undo the stack frame manipulations
ret
```
Memory example listing file

1 %line 1+1 memory.asm
2 [section .data]
3 00000000 04000000 a dd 4
4 00000004 CDCC8C40 b dd 4.4
5 00000008 00000000<rept> c times 10 dd 0
6 00000030 01000200 d dw 1, 2
7 00000034 FB e db 0xfb
8 00000035 68656C6C6F20776F72- f db "hello world", 0
9 00000035 6C6400

- Addresses are relative to start of .data in this file
- Notice that the 4 byte of 4 is at address 0 (backwards)
- \( b = 0x408ccccd = 0 \ 10000001 \ 00011001100110011001101 \)
- Sign bit is 0, exponent field is 0x81 = 129, exponent = 2
- Fraction is 1.00011001100110011001101
Memory example listing file (2)

11 [section .bss]
12 00000000 <gap> g resd 1
13 00000004 <gap> h resd 10
14 0000002C <gap> i resb 100

- Notice that the addresses start again at 0
- The commands reserve space
- resd 1 reserves 1 double word or 4 bytes
- resd 10 reserves 10 double words or 40 bytes
- resb 100 reserves 100 bytes
[section .text]
[global main]
main:
  push rbp
  mov rbp, rsp
  sub rsp, 16
  xor eax, eax
  leave
  ret
Examining memory

Useful tools to examine memory are:

- gdb
- ebe