Introduction to 64 Bit Intel Assembly Language Programming

Ray Seyfarth

July 30, 2011
Outline

1. Goals for this course

2. Why study assembly language?

3. What is a computer?

4. Machine language

5. Assembly language

6. Assembling and linking
Goals for this course

- 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
- Floating point instructions
- Arrays
- Functions
- Structs
- Data structures
- Using system calls and C libraries
- SSE and AVX instructions
Problems with assembly language

- Assembly is the poster child for non-portability
  - Different CPU = different assembly
  - Different OS = different function ABI
  - 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
- Language is not orthogonal
What’s good about assembly language?

- Assembly language is fast
  - Optimizing C/C++ compilers can be faster
  - 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?

- Explains how the computer works
- Numbers are stored in registers
- Arithmetic is done with registers
- C function register and stack usage defined
- Stack frames are used by debuggers
- 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 C/C++ coding will improve
What is a computer?

- A machine to process bits
  - We consider the bits to mean things
  - True or false
  - Integers
  - Floating point numbers
  - Characters and strings
  - User-created types
    - Physical objects, animals, plants, minerals
    - Lists of things
    - Stacks of things
    - Queues of things
    - Priority queues of things
    - Trees of various types
    - Hash tables

64 Bit Intel Assembly Language ©2011 Ray Seyfarth
Bytes

- Memory is organized as 8 bit bytes
- First byte of memory is at address 0
- Second byte is at address 1
- Memory is an array of bytes
- Consider the byte with bits 01010101
  - Considered as a decimal number it is 85
  - In the right context it can be a machine instruction
    - Push the rbp register onto the run-time stack
  - Considered as a character is it ‘U’
  - It could be part of the string “Undefined”
  - It could be part of a larger number, like 85*256 + 17 = 21777
  - It could be part if an address in the computer
Program execution

The 12 bytes to the right constitute a program which if placed in memory and executed, simply exits with status 5. The addresses are shown in hexadecimal to emphasize that the addresses are fairly close to the beginning of a page starting at 0x400000.

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>4000b0</td>
<td>184</td>
</tr>
<tr>
<td>4000b1</td>
<td>1</td>
</tr>
<tr>
<td>4000b2</td>
<td>0</td>
</tr>
<tr>
<td>4000b3</td>
<td>0</td>
</tr>
<tr>
<td>4000b4</td>
<td>0</td>
</tr>
<tr>
<td>4000b5</td>
<td>187</td>
</tr>
<tr>
<td>4000b6</td>
<td>5</td>
</tr>
<tr>
<td>4000b7</td>
<td>0</td>
</tr>
<tr>
<td>4000b8</td>
<td>0</td>
</tr>
<tr>
<td>4000b9</td>
<td>0</td>
</tr>
<tr>
<td>4000ba</td>
<td>205</td>
</tr>
<tr>
<td>4000bb</td>
<td>128</td>
</tr>
</tbody>
</table>
Machine language

- Machine language is a sequence of bytes
- The bytes specify instructions and data
- Many instructions include a data address
- Branching instructions include instruction addresses
- Adding a new instruction changes all subsequent instruction addresses
- Changes to data can alter data addresses
  - Increasing an array size changes all subsequent data addresses
  - Adding a data item can change subsequent data addresses
- Each changed address must be changed in all instructions using the address
- This is hard enough to stimulate creativity
- People figured out how to use symbolic names for data and instructions
Second generation language

- 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
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
  - `-g dwarf2` says we want `dwarf2` debugging info
    - `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:

mov eax,1
mov ebx,5
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