Assembly and Bomb Lab

15-213: Introduction to Computer Systems
Recitation 4: Monday, June 5, 2014

Aditya Gupta (adityag1@andrew)
Vinay Venkatesh (vinayven@Andrew)
Dues:

Bomb Lab Due: 06/10/2014

Buf Lab out: 06/10/2014
Agenda

- Assembly Refresher
- Overview of Bomb Lab
- Intro to GDB
- Bomb Lab Demo
What are Registers?

- Some Place in hardware that stores bits
  - It is NOT on the stack or in the main memory
- Remember when moving data between registers and memory, only the DATA moves, not the register.
More on Registers

- Quad = 64 bits
- Doubleword = 32 bits
- Word = 16 bits
- Byte = 8 bits

All of these are part of the same register

%rax = 64 bits
%eax = 32 bits
%ax = 16 bits
%al = 8 bits
Special Registers IA32

- **%eip- Instruction Pointer**
  - Points to the next instruction to be executed

- **%esp- Stack Pointer**
  - Points to the top of the Stack

- **%eax- Holds the return value**
  - Can also be used as a general purpose register
### x86-64 Integer Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>return</td>
</tr>
<tr>
<td>%rbx</td>
<td></td>
</tr>
<tr>
<td>%rcx</td>
<td>arg 4</td>
</tr>
<tr>
<td>%rdx</td>
<td>arg 3</td>
</tr>
<tr>
<td>%rsi</td>
<td>arg 2</td>
</tr>
<tr>
<td>%rdi</td>
<td>arg 1</td>
</tr>
<tr>
<td>%esp</td>
<td></td>
</tr>
<tr>
<td>%ebp</td>
<td></td>
</tr>
<tr>
<td>%rax</td>
<td>arg 5</td>
</tr>
<tr>
<td>%rbx</td>
<td></td>
</tr>
<tr>
<td>%rcx</td>
<td>arg 6</td>
</tr>
<tr>
<td>%rdx</td>
<td></td>
</tr>
<tr>
<td>%rsi</td>
<td></td>
</tr>
<tr>
<td>%rdi</td>
<td>%r10d</td>
</tr>
<tr>
<td>%esp</td>
<td>%r11d</td>
</tr>
<tr>
<td>%ebp</td>
<td>%r12d</td>
</tr>
<tr>
<td>%rax</td>
<td>%r13d</td>
</tr>
<tr>
<td>%rbx</td>
<td>%r14d</td>
</tr>
<tr>
<td>%rcx</td>
<td>%r15d</td>
</tr>
<tr>
<td>%rdx</td>
<td></td>
</tr>
<tr>
<td>%rsi</td>
<td></td>
</tr>
<tr>
<td>%rdi</td>
<td></td>
</tr>
<tr>
<td>%esp</td>
<td></td>
</tr>
<tr>
<td>%ebp</td>
<td></td>
</tr>
</tbody>
</table>
# Assembly: Operands

<table>
<thead>
<tr>
<th>Data type</th>
<th>Syntax</th>
<th>Examples</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Immediate values (constant integers)</strong></td>
<td><strong>Start with $</strong></td>
<td>$0x0</td>
<td>Don’t forget $0x$ means hex!</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$-15213$</td>
<td></td>
</tr>
<tr>
<td><strong>Registers</strong></td>
<td><strong>Start with %</strong></td>
<td>%esi</td>
<td>Can represent a value or an address</td>
</tr>
<tr>
<td></td>
<td></td>
<td>%rax</td>
<td></td>
</tr>
<tr>
<td><strong>Memory locations</strong></td>
<td><strong>Parentheses around a register, or addressing mode –</strong></td>
<td>(%esi)</td>
<td>Parentheses dereference. If %esi stores an address, (%esi) is the value at that address.</td>
</tr>
<tr>
<td></td>
<td><strong>D(Rb,Ri,S)</strong></td>
<td>0x8(%rax)</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>(%rax, %rsi, 4)</td>
<td></td>
</tr>
</tbody>
</table>

**Note:** IA32 imposes the restriction that a move instruction cannot have both operands refer to memory locations.
Representing Addresses

- **x86(_64) Common Addressing**
  - Offset(Base, Index, Scale)
  - \[ D(Rb, Ri, S) \rightarrow \text{Mem}[Rb + Ri*S + D] \]
    - D can be any signed integer
    - Scale is 1, 2, 4, 8 (assume 1 if omitted)
    - Assume 0 for base if omitted
Representing Addresses

- Using parenthesis
  - Most of the time parenthesis means dereference

Examples of parenthesis usage:

- (%eax)
  - Contents of memory at address stored, %eax

- (%ebx, %ecx)
  » Contents of memory stored at address, %ebx + %ecx

- (%ebx, %ecx, 8)
  » Contents of memory stored at address, %ebx + 8*%ecx

- 4(%ebx, %ecx, 8)
  » Contents of memory stored at address, %ebx + 8*%ecx + 4
Representing Addresses

- Using parenthesis
  - Sometimes parenthesis are used just for addressing

**Example**

- leal(%ebx, %ecx, 8), destination
  - Take only the values %ebx + 8*%ecx
  - Does not dereference, uses the calculated value directly
- jmpq*0x402660(%rax,8)
  - The * does the dereference

**Examples of not using parenthesis**

- %eax
  - Use the value in %eax!
- $0x213
  - A constant value
Conditional Flags

- Most operations will set conditional flags
  - Bit operations
  - Arithmetic
  - Comparisons...

- Core idea: For conditionals, look one instruction before it to see whether it is true or false
Some Important Flags

- **Carry (CF)**
  - Arithmetic carry/ borrow

- **Parity (PF)**
  - Odd or even number of bits set

- **Zero (ZF)**
  - Result was zero

- **Sign (SF)**
  - Most significant bit was set

- **Overflow (OF)**
  - Result does not fit into the location
# Assembly: Some Common Operations

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>mov %rdi, %rax</code></td>
<td>( rax = rdi )</td>
</tr>
<tr>
<td><code>add %rdi, %rax</code></td>
<td>( rax = rax + rdi )</td>
</tr>
<tr>
<td><code>sub %rdi, %rax</code></td>
<td>( rax = rax - rdi )</td>
</tr>
<tr>
<td><code>lea (%rdi, %rsi, 2), %rax</code></td>
<td>( rax = rdi + (2 \times rsi) ) (doesn’t dereference)</td>
</tr>
<tr>
<td><code>call foo</code></td>
<td>Calls function “foo”</td>
</tr>
<tr>
<td><code>push %eax</code></td>
<td>Pushes eax onto the stack</td>
</tr>
<tr>
<td><code>pop %eax</code></td>
<td>Pops a value off the stack and into eax</td>
</tr>
<tr>
<td><code>ret</code></td>
<td>Returns to the return address (i.e., the next line in the calling function)</td>
</tr>
<tr>
<td><code>nop</code></td>
<td>Does nothing!</td>
</tr>
</tbody>
</table>

You may see suffixes on the end: \( b, w, l, q \)  
Specify operand is 1, 2, 4, 8 bytes
Assembly: Comparisons and Jumps

- Remember from class that Assembly uses comparisons and jumps (gotos) to execute various conditionals and loops.
- `cmp b, a` sets the same flags as computing `a – b`.
- `test b, a` sets the same flags as computing `a & b`.
- These are usually followed by a conditional jump instruction that relies on the results.
- Watch out for operand order:

  ```
cmpl %eax, %edx  # if %edx > %eax,
jg 401095  # jump to 401095
  ```
# Assembly: Comparisons and Jumps

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
<th>Instruction</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmp</td>
<td>Always jump</td>
<td>ja</td>
<td>Jump if above (unsigned &gt;)</td>
</tr>
<tr>
<td>je/jz</td>
<td>Jump if /=0</td>
<td>jae</td>
<td>Jump if above or equal</td>
</tr>
<tr>
<td>jne/jnz</td>
<td>Jump if ≠/0</td>
<td>jb</td>
<td>Jump if below (unsigned &lt;)</td>
</tr>
<tr>
<td>jg</td>
<td>Jump if &gt;</td>
<td>jbe</td>
<td>Jump if below or equal</td>
</tr>
<tr>
<td>jge</td>
<td>Jump if &gt;=</td>
<td>js</td>
<td>Jump if negative</td>
</tr>
<tr>
<td>jl</td>
<td>Jump if &lt;</td>
<td>jns</td>
<td>Jump if nonnegative</td>
</tr>
<tr>
<td>jle</td>
<td>Jump if &lt;=</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Assembly: Comparisons and Jumps

- `cmp $0x42, %edi
  je 400d3b
  if ___edi == 66___, jump to 400d3b`

- `cmp %esi, %edx
  jle 400e71
  if ___edx <= esi___, jump to 400e71`

- `test %rdi, %rdi
  jne 400e87
  if ___%rdi != 0___, jump to 400e87`
Agenda

- Assembly Refresher
- Overview of Bomb Lab
- Intro to GDB
- Bomb Lab Demo
Bomb Lab

- Oh no! Dr. Evil has written an evil program that will “explode” the Shark machines!

- The program is in phases, each of which reads in input – something like a password – from standard input.

- If your input is correct, you go on to the next phase.

- If not, the bomb explodes. The program prints “BOOM!!!” and terminates, and you lose half a point. (Your score is updated automatically – you don’t have to upload anything to Autolab.)
Bomb Lab

- We give you:
  - Partial source code, in which Dr. Evil mocks you
  - The executable file itself

You can’t read the C source code. So how can you figure out what the program does?

From the binary executable!
Agenda

- Assembly Refresher
- Overview of Bomb Lab
- Intro to GDB
- Bomb Lab Demo
Your Defusing Toolkit

- objdump -t bomb prints the symbol table
- strings bomb prints all printable strings
- objdump -d bomb prints the Assembly
- gdb bomb shows you the executable file in Assembly and lets you step through it line by line, peeking into the registers and stack as you go

All the GDB commands you need are in [http://csapp.cs.cmu.edu/public/docs/gdbnotes-x86-64.pdf](http://csapp.cs.cmu.edu/public/docs/gdbnotes-x86-64.pdf)
GDB: Stepping Through Code

**break <location>**
- sets a breakpoint. Location can be a function name or an address.
- Pro tip: you have to reset your break points when you restart GDB!

**run / run <filename>**
- runs the program up till the next breakpoint.
- Pro tip: instead of typing in your inputs each time, you can put them in a text file, one per line, and run that.

**disassemble** (or disas – but not dis!!!)
- shows you the current function, with an arrow to the next line.

**step / steipi / nexti**
- step executes one C statement – it doesn’t work for us.
- steipi steps to the next line of Assembly.
- nexti does the same but doesn’t stop in function calls.
- steipi <n> or nexti <n> steps through n lines.
GDB: Examining Data

- **info registers**
  - prints the (hex) contents of every register.

- **print $<register>**
  - prints the contents of a register.
  - Note the $ – not a %.
  - Use /x or /d, to specify hex or decimal: `print /d $rax`. 

- **x $<register> / x 0x<address>**
  - prints what the register points to (or what’s at the given address).
  - By default, prints one word (a “word” here is 4 bytes).
  - However, in addition to specifying format (now including /s, string), you can specify how many objects of what size to print, in the format `x /[num][size][format]`, for example: `x /4wd $rsp`
One Last Hint: `sscanf`

- The bomb frequently calls `sscanf` to read in formatted arguments.
- If you’re not familiar with the formatting used by `printf`, now’s the time!
- Example: `%s %x %s` represents an input of a string, hex number, and string.
- This could be handy in figuring out what kinds of arguments a phase is expecting.
- `man scanf`!
Resources

- Assignment write-up
- GDB cheat sheet: [http://csapp.cs.cmu.edu/public/docs/gdbnotes-x86-64.pdf](http://csapp.cs.cmu.edu/public/docs/gdbnotes-x86-64.pdf)
- CS:APP Chapter 3
- If you’re stuck, check the course FAQ: [http://www.cs.cmu.edu/~213/faq.html](http://www.cs.cmu.edu/~213/faq.html)
- If that doesn’t help, email us: 15-213-staff@cs.cmu.edu
- Office hours: Sun-Thu, 5:30-8:30 pm, in Wean 5207
- Peer tutoring: Tue 8:30-11, Mudge Reading Room

- Note: if you Google Assembly instructions, make sure what you find is in AT&T syntax, not Intel. (The operands are reversed.)
Agenda

- Overview of Bomb Lab
- Assembly Refresher
- Intro to GDB
- Bomb Lab Demo