ANITA’S SUPER AWESOME RECITATION SLIDES
15/18-213: Introduction to Computer Systems Assembly and GDB, 4 Jun 2013
Anita Zhang, Section M
Management and Stuff

- Bomb Lab due Tues, 11 Jun 2013, 11:59 pm EST
  - Apparently for distance students it’s 2 days after
  - This is my favorite lab!
- Buf Lab out Tues, 11 Jun 2013, 11:59 pm EST
  - Due the week after
- FAQ on the main site
  - Has some stuff
  - Answers to “Permission denied” errors, etc
What’s on the Menu Today?

- Help (again)
- Books (again)
- Motivation
- Registers
- Assembly
- Bomb Lab Overview
- GDB
- Walkthrough
- More Bomb Lab
Helping Us, Helping You?

- Email us: 15-213-staff@cs.cmu.edu
  - Please attach C files if you have a specific question
  - Responses within 2 minutes (record!)
- IRC: irc.freenode.net, ##213
  - Anita polls it every 3 hours
- Videos on Blackboard
- Everything else, Autolab: autolab.cs.cmu.edu
- Office hours: Sun-Thurs, 6pm – 9pm, Gates 5205
  - Both Michael and Anita will be there (mostly)
  - We leave at 7:30pm if no one shows up
WHAT HAVE YOU READ?


Why Are We Doing This Again?
DEFINITIONS AND CONVENTIONS

- **Register**
  - Some place in hardware that stores bits

- **Caller save**
  - Saved by the caller of a function
  - Before a function call, the caller must save any caller save register values it wants preserved

- **Callee save**
  - Saved by the callee of a function
  - The callee is required to save and restore the values in these registers if it is using them in the function
INSIGHT FOR THE INQUISITIVE

- Why are we not learning about the stack yet?
  - Because x86_64

- “Technology note”
  - x86(_64) only
REGISTERS AND ALL THEM BITS

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

These are all parts of the same register
WHAT WE’RE WORKING WITH

○ General Purpose (x86)
  - Caller Save: %eax, %ecx, %edx
  - Callee Save: %ebx, %esi, %edi, %ebp, %esp
  - x86_64 conventions on the next slide

○ Specials
  - %eip – instruction pointer
  - %ebp – frame pointer
  - %esp – stack pointer

○ Conditional Flags
  - Sit in a special register of its own
  - You only need to worry about the ones mentioned later
# x86_64, LOTS OF REGISTERS!

<table>
<thead>
<tr>
<th>64 bits wide</th>
<th>32 bits wide</th>
<th>16 bits wide</th>
<th>8 bits wide</th>
<th>8 bits wide</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>%eax</td>
<td>%ax</td>
<td>%ah</td>
<td>%al</td>
<td>Return Value</td>
</tr>
<tr>
<td>%rbx</td>
<td>%ebx</td>
<td>%bx</td>
<td>%bh</td>
<td>%bl</td>
<td>Callee Save</td>
</tr>
<tr>
<td>%rcx</td>
<td>%ecx</td>
<td>%cx</td>
<td>%ch</td>
<td>%cl</td>
<td>4th Argument</td>
</tr>
<tr>
<td>%rdx</td>
<td>%edx</td>
<td>%dx</td>
<td>%dh</td>
<td>%dl</td>
<td>3rd Argument</td>
</tr>
<tr>
<td>%rsi</td>
<td>%esi</td>
<td>%si</td>
<td></td>
<td>%sil</td>
<td>2nd Argument</td>
</tr>
<tr>
<td>%rdi</td>
<td>%edi</td>
<td>%di</td>
<td></td>
<td>%dil</td>
<td>1st Argument</td>
</tr>
<tr>
<td>%rbp</td>
<td>%ebp</td>
<td>%bp</td>
<td></td>
<td>%bpl</td>
<td>Callee Save</td>
</tr>
<tr>
<td>%rsp</td>
<td>%esp</td>
<td>%sp</td>
<td></td>
<td>%spl</td>
<td>Stack Pointer</td>
</tr>
<tr>
<td>%r8</td>
<td>%r8d</td>
<td>%r8w</td>
<td></td>
<td>%r8b</td>
<td>5th Argument</td>
</tr>
<tr>
<td>%r9</td>
<td>%r9d</td>
<td>%r9w</td>
<td></td>
<td>%r9b</td>
<td>6th Argument</td>
</tr>
<tr>
<td>%r10</td>
<td>%r10d</td>
<td>%r10w</td>
<td></td>
<td>%r10b</td>
<td>Caller Save</td>
</tr>
<tr>
<td>%r11</td>
<td>%r11d</td>
<td>%r11w</td>
<td></td>
<td>%r11b</td>
<td>Caller Save</td>
</tr>
<tr>
<td>%r12</td>
<td>%r12d</td>
<td>%r12w</td>
<td></td>
<td>%r12b</td>
<td>Callee Save</td>
</tr>
<tr>
<td>%r13</td>
<td>%r13d</td>
<td>%r13w</td>
<td></td>
<td>%r12b</td>
<td>Callee Save</td>
</tr>
<tr>
<td>%r14</td>
<td>%r14d</td>
<td>%rw</td>
<td></td>
<td>%r14b</td>
<td>Callee Save</td>
</tr>
<tr>
<td>%r15</td>
<td>%r15d</td>
<td>%r15w</td>
<td></td>
<td>%r15b</td>
<td>Callee Save</td>
</tr>
</tbody>
</table>
SOME MORE DEFINITIONS

Memory Addressing
- How assemblers denote memory locations
  - Direct
  - Indirect
  - Relative
  - Absolute
  - ...
- Syntax differs, addresses do not
Reasons Why Intel is Ridiculous and Awesome

- Operations can take several forms:
  - Register-to-Register
  - Register-to-Memory / Memory-to-Register
  - Immediate-to-Register / Immediate-to-Memory
  - One address operations (push, pop)
  - Did I miss any?
Reasons Why Intel is Ridiculous and Awesome

- x86(_64) Addressing (some kind of indirect)
  - Offset(Base, Index, Scale)
  - $D(Rb, Ri, S) \rightarrow Mem[Rb + Ri*S + D]$
    - $D$ can be any signed integer
    - Scale is 1, 2, 4, 8 (assume 1 if omitted)
    - Assume 0 for base if omitted
Reasons Why Intel is Ridiculous and Awesome

- Using parenthesis
  - Most of the time parenthesis means dereference
    - This is still only x86(_64)

- 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
REASONS WHY INTEL IS RIDICULOUS AND AWESOME

- Using parenthesis
  - *Sometimes* parenthesis are used just for addressing
    - This is still only x86(_64)

- Example
  - `leal (%ebx, %ecx, 8), destination`
    - Take the address, `%ebx + 8*%ecx`
    - *Does not dereference*, uses the calculated value directly

- Examples of not using parenthesis
  - `%eax`
    - Use the value in `%eax`!
  - `$0x213`
    - A constant value
Review of Conditionals/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
  - Will be explained
FLAGS WE CARE ABOUT

- 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
**Prep for All the Cheat Sheets**

- Warning: The following slides contain lots of assembly instructions.
  - All from CS:APP (our textbook BTW)
  - We’re not going over every single one...
    - Use it as a reference for Bomb Lab

- Quick note on Intel vs. AT&T
  - This is AT&T syntax (also, Bomb Lab syntax)
    - Looks like: “src, dest”
  - Intel tends to follow “dest, src”
    - Check out their ISA sometime
### All the Cheat Sheets (Movement)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>movb</td>
<td>S, D Move byte</td>
</tr>
<tr>
<td>movw</td>
<td>S, D Move word</td>
</tr>
<tr>
<td>movl</td>
<td>S, D Move doubleword</td>
</tr>
<tr>
<td>movsbw</td>
<td>S, D Move byte to word (sign extended)</td>
</tr>
<tr>
<td>movsbl</td>
<td>S, D Move byte to doubleword (sign extended)</td>
</tr>
<tr>
<td>movswl</td>
<td>S, D Move word to doubleword (sign extended)</td>
</tr>
<tr>
<td>movzbw</td>
<td>S, D Move byte to word (zero extended)</td>
</tr>
<tr>
<td>movzbl</td>
<td>S, D Move byte to doubleword (zero extended)</td>
</tr>
<tr>
<td>movzwl</td>
<td>S, D Move word to doubleword (zero extended)</td>
</tr>
<tr>
<td>pushl</td>
<td>S Push double word</td>
</tr>
<tr>
<td>popl</td>
<td>D Pop double word</td>
</tr>
</tbody>
</table>
# All the Cheat Sheets (Bit Ops)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEAL</td>
<td>S, D</td>
</tr>
<tr>
<td>INC</td>
<td>D</td>
</tr>
<tr>
<td>DEC</td>
<td>D</td>
</tr>
<tr>
<td>NEG</td>
<td>D</td>
</tr>
<tr>
<td>NOT</td>
<td>D</td>
</tr>
<tr>
<td>ADD</td>
<td>S, D</td>
</tr>
<tr>
<td>SUB</td>
<td>S, D</td>
</tr>
<tr>
<td>IMUL</td>
<td>S, D</td>
</tr>
<tr>
<td>XOR</td>
<td>S, D</td>
</tr>
<tr>
<td>OR</td>
<td>S, D</td>
</tr>
<tr>
<td>AND</td>
<td>S, D</td>
</tr>
<tr>
<td>SAL</td>
<td>k, D</td>
</tr>
<tr>
<td>SHL</td>
<td>k, D</td>
</tr>
<tr>
<td>SAR</td>
<td>k, D</td>
</tr>
<tr>
<td>SHR</td>
<td>k, D</td>
</tr>
</tbody>
</table>
# All the Cheat Sheets (Specials)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
</tr>
</thead>
</table>
| imull       | \( R[\%edx]:R[\%eax] \leftarrow S \times R[\%eax] \)  
Signed full multiply of \%eax by S  
Result stored in \%edx:\%eax |
| mull        | \( R[\%edx]:R[\%eax] \leftarrow S \times R[\%eax] \)  
Unsigned full multiply of \%eax by S  
Result stored in \%edx:\%eax |
| cltd        | \( R[\%edx]:R[\%eax] \leftarrow \text{SignExtend}(R[\%eax]) \)  
Sign extend \%eax into \%edx |
| idivl       | \( R[\%edx] \leftarrow R[\%edx]:R[\%eax] \mod S; \)  
\( R[\%eax] \leftarrow R[\%edx]:R[\%eax] \div S \)  
Signed divide of \%eax by S  
Quotient stored in %eax  
Remainder stored in %edx |
| divl        | \( R[\%edx] \leftarrow R[\%edx]:R[\%eax] \mod S; \)  
\( R[\%eax] \leftarrow R[\%edx]:R[\%eax] \div S \)  
Unsigned divide of \%eax by S  
Quotient stored in %eax  
Remainder stored in %edx |
### All the Cheat Sheets (Comparisons)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmpb S2, S1</td>
<td>Compare byte S1 and S2, Sets conditional flags based on S1 – S2.</td>
</tr>
<tr>
<td>cmpw S2, S1</td>
<td>Compare word S1 and S2, Sets conditional flags based on S1 – S2.</td>
</tr>
<tr>
<td>cmpl S2, S1</td>
<td>Compare double word S1 and S2, Sets conditional flags based on S1 – S2.</td>
</tr>
<tr>
<td>testb S2, S1</td>
<td>Compare byte S1 and S2, Sets conditional flags based on S1 &amp; S2.</td>
</tr>
<tr>
<td>testw S2, S1</td>
<td>Compare word S1 and S2, Sets conditional flags based on S1 &amp; S2.</td>
</tr>
<tr>
<td>testl S2, S1</td>
<td>Compare double word S1 and S2, Sets conditional flags based on S1 &amp; S2.</td>
</tr>
</tbody>
</table>
# All the Cheat Sheets (Set)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>sete/ setz</td>
<td>D ( D \leftarrow ZF ) (“set if equal to 0”)</td>
</tr>
<tr>
<td>setne/ setnz</td>
<td>D ( D \leftarrow \sim ZF ) (set if not equal to 0)</td>
</tr>
<tr>
<td>sets</td>
<td>D ( D \leftarrow SF ) (set if negative)</td>
</tr>
<tr>
<td>setns</td>
<td>D ( D \leftarrow \sim SF ) (set if nonnegative)</td>
</tr>
<tr>
<td>setg/ setnle</td>
<td>D ( D \leftarrow \sim (SF \land OF) \land \sim ZF ) (set if greater (signed (&gt;)))</td>
</tr>
<tr>
<td>setge/ setnl</td>
<td>D ( D \leftarrow \sim (SF \land OF) ) (set if greater or equal (signed (\geq)))</td>
</tr>
<tr>
<td>setl/ setnge</td>
<td>D ( D \leftarrow SF \lor OF ) (set if less than (signed (&lt;)))</td>
</tr>
<tr>
<td>setle/ setng</td>
<td>D ( D \leftarrow (SF \lor OF) \lor ZF ) (set if less than or equal (signed (\leq)))</td>
</tr>
<tr>
<td>seta/ setnbe</td>
<td>D ( D \leftarrow \sim CF \land \sim ZF ) (set if above (unsigned (&gt;)))</td>
</tr>
<tr>
<td>setae/ setnb</td>
<td>D ( D \leftarrow \sim CF ) (set if above or equal (unsigned (\geq)))</td>
</tr>
<tr>
<td>setb/ setnae</td>
<td>D ( D \leftarrow CF ) (set if below (unsigned (&lt;)))</td>
</tr>
<tr>
<td>setbe/ setna</td>
<td>D ( D \leftarrow CF \lor ZF ) (set if below or equal (unsigned (\leq)))</td>
</tr>
</tbody>
</table>
# All the Cheat Sheets (Jump)

<table>
<thead>
<tr>
<th>Instructions</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmp</td>
<td>Label</td>
</tr>
<tr>
<td>jmp</td>
<td>*Operand</td>
</tr>
<tr>
<td>je/ jz</td>
<td>Label</td>
</tr>
<tr>
<td>jne/ jnz</td>
<td>Label</td>
</tr>
<tr>
<td>js</td>
<td>Label</td>
</tr>
<tr>
<td>jns</td>
<td>Label</td>
</tr>
<tr>
<td>jg/ jnle</td>
<td>Label</td>
</tr>
<tr>
<td>jge/ jnl</td>
<td>Label</td>
</tr>
<tr>
<td>jl/ jnge</td>
<td>Label</td>
</tr>
<tr>
<td>jle/ jng</td>
<td>Label</td>
</tr>
<tr>
<td>ja/ jnbe</td>
<td>Label</td>
</tr>
<tr>
<td>jae/ jnb</td>
<td>Label</td>
</tr>
<tr>
<td>jb/ jnae</td>
<td>Label</td>
</tr>
<tr>
<td>jbe/ jna</td>
<td>label</td>
</tr>
</tbody>
</table>
# All the Cheat Sheets (Cmove)

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>cmov/ cmovz</td>
<td>S, R</td>
</tr>
<tr>
<td>cmovne/ cmovnz</td>
<td>S, R</td>
</tr>
<tr>
<td>cmovs</td>
<td>S, R</td>
</tr>
<tr>
<td>cmovns</td>
<td>S, R</td>
</tr>
<tr>
<td>cmovg/ cmovnle</td>
<td>S, R</td>
</tr>
<tr>
<td>cmovge/ cmovnl</td>
<td>S, R</td>
</tr>
<tr>
<td>cmovl/ cmovnge</td>
<td>S, R</td>
</tr>
<tr>
<td>cmovle/ cmovg</td>
<td>S, R</td>
</tr>
<tr>
<td>cmova/ cmovnbe</td>
<td>S, R</td>
</tr>
<tr>
<td>cmovae/ cmovnb</td>
<td>S, R</td>
</tr>
<tr>
<td>cmovb/ cmovnae</td>
<td>S, R</td>
</tr>
<tr>
<td>cmovbe/ cmovna</td>
<td>S, R</td>
</tr>
</tbody>
</table>
**All the Cheat Sheets (Calling)**

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>call</td>
<td>Push return and jump to label</td>
</tr>
<tr>
<td>call</td>
<td>Push return and jump to specified location</td>
</tr>
</tbody>
</table>
| leave       | Prepare stack for return. Set stack pointer to %ebp and pop top stack into %ebp. In assembly (AT&T syntax of source, destination):

```
mov %ebp, %esp
pop %ebp
```

| ret          | Pop return address from stack and jump there |
Dr. Evil and Bomblab

- 6 stages, each asking for input
  - Wrong input → bomb explodes (lose 1/2 point)
  - Each stage may have multiple answers

- You get:
  - Bomb executable
  - Partial source of Dr. Evil mocking you

- Speed up next phase traversal with a text file
  - Place answers on each line
  - Run with bomb as ./bomb <solution file>
How it Works

“But how do I find the solutions if I don’t have C code to work from?”

- Read a lot of bomb disassembly
  - All of the phases are just loops and patterns
  - Or just dead simple (see the demo)
- GDB

If you’re not working on a shark machine, your bomb won’t work.
  - Will get “illegal host”
WORKING THROUGH THIS THING

- Read the disassembly
  - phase_1, phase_2, phase_3...
  - explode_bomb
  - Understand what’s going on

- GNU Debugger
  - Step through each instruction, examine registers..
  - Set up breakpoints
  - Make sure to type “kill” when you hit the explode_bomb breakpoint
    - You’re screwed once you hit here, so why not exit?
But I Don’t Know How to GDB??

- Here have a cheat sheet
  - [http://csapp.cs.cmu.edu/public/docs/gdbnotes-x86-64.pdf](http://csapp.cs.cmu.edu/public/docs/gdbnotes-x86-64.pdf)
  - Everything you need to use GDB to solve bomblab
FANCY GDB

Welcome to my library! You have 6 phases with
which to blow yourself up. Have a nice day!

(gdb) r
Starting program: /afs/cmu.edu/usr8/anitasha/private/TA_15-213/bomb115/bomb
Fancy GDB Commands

- Layout commands split GDB into cool windows
  - May/ may not lag a lot.
  - Has a tendency to not work properly sometimes

- `layout asm`
  - Splits GDB into assembly and GDB command

- `layout src`
  - Splits GDB into C source and GDB command

- `layout regs`
  - Splits GDB into register window with either source or assembly, and GDB command

- Arrow, page up/down to traverse layout windows

- `ctrl+x a` to switch back to normal GDB
GETTING STARTED

- Download and untar ON A SHARK MACHINE

- shark> objdump –d bomb > disassembly filename
- shark> objdump –t bomb > symbol table filename
- shark> strings bomb > strings filename
- shark> gdb bomb
**Speed up the Wait**

- When you have solutions, put it into a text file
  - Separate each solution with a newline
  - Your bomb will auto-advance completed phases with pre-filled solutions
- Then when you run gdb next time:
  - `(gdb)> run solution_file`
DEMO TIME
BOMB LAB SPECIFICS

- int sscanf (const char *s, const char *format, ...);
  - s
    - Source string to retrieve data from
  - format
    - Formatting string used to get values from the source string
  - ...
    - Depending the format string, one location (address) per formatter used to hold values extracted from source string
SSCANF EXAMPLE

#include <stdio.h>

int main () {
    char sentence []="Rudolph is 12 years old";
    char str [20];
    int i;
    sscanf (sentence,"%s %*s %d",str,&i);
    printf ("%s -> %d\n",str,i);
    return 0;
}

- Outputs: Rudolph -> 12
Why do we care about `sscanf`?

- Mostly used to read in arguments
- Note of which locations read in values will be stored
  - Important for knowing where arguments will be stored
  - And how they will be used
MORE BOMB LAB SPECIFICS

- Jump tables
  - In memory is an “array” of locations
  - In assembly it is possible to index into this “array”
  - Each entry of the array will potentially hold addresses to the next instruction to go to
Jump Tables

- The tip-off is something like this:
  - `jmpq *0x400600(%rax,8)`
    - Empty base means implied 0
    - `%rax` is the “index”
    - 8 is the “scale”
      - In a jump table with addresses, 64-bit machines addresses are 8 bytes
    - `*` indicates a dereference (as in regular C)
      - Like `leal`; does not do a dereference just with parenthesis
  - Put it all together: “Jump to the address stored in the address 0x400600 + `%rax*8`”

- Using GDB (example output): `x/8g 0x400600`
  0x400600: 0x00000000004004d1 0x00000000004004c8
  0x400610: 0x00000000004004c8 0x00000000004004be
  0x400620: 0x00000000004004c1 0x00000000004004d7
  0x400630: 0x00000000004004c8 0x00000000004004be
CREDITS & QUESTIONS

- [Link](http://stackoverflow.com/questions/757398/what-are-some-ways-you-can-manage-large-scale-assembly-language-projects)
- P. 274 of CS:APP – x86_64 Registers
- P. 171 - 221 of CS:APP – Assembly Instructions
- [Link](http://www.cplusplus.com/reference/cstdio/scanf/)