Assembly & BombLab Introduction

15-213 (18-213): Introduction to Computer Systems
Recitation 4, Feb 4 2013

Alexey Tumanov (atumanov)
Section C, Monday 11:30 – 12:20, WeH 5302
Outline

- Introduction to Assembly
  - register overview
  - assembly instructions
  - assembly instruction operands
  - managing control flow

- Introduction to Bomblab
  - lab overview
  - auxiliary tools
    - GNU debugger (gdb)
Administrativia

- **FAQ:** [http://www.cs.cmu.edu/~213/faq.html](http://www.cs.cmu.edu/~213/faq.html)
  - E.g. Why do I get “permission denied” error?
  - Read through before you ask questions
  - Check back for any future updates

- **Style guide:** [http://www.cs.cmu.edu/~213/codeStyle.html](http://www.cs.cmu.edu/~213/codeStyle.html)
  - consistent indentation
  - reasonable variable names
  - document important parts of the code
  - fit within 80 character line width
  - ~/.vimrc or ~/.emacs

- **TA Feedback form:** [https://www.ugrad.cs.cmu.edu/ta/feedback](https://www.ugrad.cs.cmu.edu/ta/feedback)
  - open until Feb 17
Registers (32 & 64 bit)

- 16 bits = word
- 32 bits = long = doubleword
- 64 bits = 8 bytes = quad (four words)
Registers

- General purpose registers:
  - eax (accumulator), ebx (base), ecx (counter), edx
  - esi, edi (used for string manipulation by convention)
  - rax, rbx, rcx, rdx, rsi, rdi, r8 - r15, rbp (x86_64)

- Special purpose registers:
  - EIP – instruction pointer register
  - EBP – branch pointer register
  - ESP – stack pointer register
  - eflags (carry, parity, zero, sign, overflow)

- x86 → x86_64
  - ebp/rbp is no longer special
  - 8 new registers
  - 64 bit registers, subsuming 32 bit registers
  - full-width instructions : L → Q (e.g., movl → movq)
Assembly Instructions: Arithmetic + Logic

- **Binary operators:**  \( OP \text{ SRC}, \text{ DST} \iff \text{ DST} = OP(\text{ DST}, \text{ SRC}) \)
  - addl, subl, imull, sarl, sall, shrl, shll, xorl, andl, orl
  - operand order matters:
    - e.g., SARL %eax, %ebx \(\rightarrow\) ebx = ebx >> eax

- **Unary operators:**  \( OP \text{ DST} \iff \text{ DST} = OP(\text{ DST}) \)
  - incl, decl, negl, notl
Assembly Operands

- **Immediate**
  - $0xff, $5, $-1
  - decimal, octal, or hexadecimal (as in C)
  - size: byte, word, long, or quad

- **Register**
  - %rax, %r14
  - e.g., XORL %ebx, %ebx (Q: what does this do?)
  - e.g., MOVL %ecx, %eax : eax ← ecx

- **Memory**
  - D(Rb, Ri, S)
  - 8(%ebx), 12(%ebx, %ecx, 4)
  - e.g., MOVL 8(%ebx), %eax : move 1st arg from stack to %eax
Special Operations

- **CLTQ** *(may see it in bomblab):*
  - convert long to quad: `%rax = SignExtend(%eax)`
  - `eax = 0x05 → rax = 0x05 ; eax = 0x80000000 → rax=0xffffffff 80000000`

- **LEAL**: load effective address
  - does not dereference memory
  - used as an arithmetic operator to perform memory address calculations
    - commonly used for calculating offsets into an array in a loop
  - example: LEAL (%eax, %eax, 4): `eax = eax * 5`

- **MOVZBL**
  - move byte to long with zero fill
Memory Addressing Modes

Most general form: Offset(Base, Index, Scale)
- D(Rb, Ri, S) \rightarrow Mem[Rb + Ri*S + D]
- Offset (D) can be any signed integer
- Scale is 1,2,4,8 (for byte, word, long, quad strides)
  - if omitted, assume 1
- Base: if omitted, assume 0x0

Special forms:
- (%eax) \rightarrow \text{contents of memory at address stored in } %eax
- (%ebx, %ecx) \rightarrow Mem[Reg[%ebx] + Reg[%ecx]]
- (%ebx, %ecx, 8) \rightarrow Mem[Reg[%ebx] + 8*Reg[%ecx]]
## Memory Addressing: Example

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>0xFF</td>
</tr>
<tr>
<td>0x104</td>
<td>0xAB</td>
</tr>
<tr>
<td>0x108</td>
<td>0x13</td>
</tr>
<tr>
<td>0x10C</td>
<td>0x11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>0x100</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x1</td>
</tr>
<tr>
<td>%edx</td>
<td>0x3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operand</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td></td>
</tr>
<tr>
<td>0x104</td>
<td></td>
</tr>
<tr>
<td>$0x108</td>
<td></td>
</tr>
<tr>
<td>4(%eax)</td>
<td></td>
</tr>
<tr>
<td>9(%eax,%edx)</td>
<td></td>
</tr>
<tr>
<td>0xfc( , %ecx, 4)</td>
<td></td>
</tr>
<tr>
<td>(%eax, %edx, 4)</td>
<td></td>
</tr>
</tbody>
</table>
# Memory Addressing: Example

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x100</td>
<td>0xFF</td>
</tr>
<tr>
<td>0x104</td>
<td>0xAB</td>
</tr>
<tr>
<td>0x108</td>
<td>0x13</td>
</tr>
<tr>
<td>0x10C</td>
<td>0x11</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>0x100</td>
</tr>
<tr>
<td>%ecx</td>
<td>0x1</td>
</tr>
<tr>
<td>%edx</td>
<td>0x3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Operand</th>
<th>Value</th>
<th>Why?</th>
</tr>
</thead>
<tbody>
<tr>
<td>%eax</td>
<td>0x100</td>
<td>register contents</td>
</tr>
<tr>
<td>0x104</td>
<td>0xAB</td>
<td>absolute addr</td>
</tr>
<tr>
<td>$0x108</td>
<td>0x108</td>
<td>immediate operand</td>
</tr>
<tr>
<td>4(%eax)</td>
<td>0xAB</td>
<td>contents of 0x104</td>
</tr>
<tr>
<td>9(%eax,%edx)</td>
<td>0x11</td>
<td>contents of 0x10c</td>
</tr>
<tr>
<td>0xfc(,%ecx,4)</td>
<td>0xff</td>
<td>contents of 0xfc+4</td>
</tr>
<tr>
<td>(%eax,%edx,4)</td>
<td>0x11</td>
<td>contents of 0x10c</td>
</tr>
</tbody>
</table>

note the difference
Control: Compare, Flags, and Jumps

- Explicit flag setting instructions:
  - `cmpl b, a` → execute `a – b` and set `eflags`
  - `testl b, a` → execute `a&b` and set `eflags`

- Flags: OF, SF, ZF, CF
  - **OF**: overflow (\(a<0 \&\& b>0 \&\& \text{diff} > 0\)) | | (\(a>0 \&\& b<0 \&\& \text{diff}<0\))
  - **SF**: sign bit set (\(a-b < 0\))
  - **ZF**: \(a==b\)
  - **CF**: previous operation resulted in a carry out from MSB

- Conditional jumps:
  - `je/jne (ZF), js/jns (SF),`
  - `jg/jge/jl/jle (SF, OF, ZF): if a >, >= , <, <= b then jump`
    - e.g., JLE: jumps iff (SF^OF) | ZF (why?)
    - `cmpl $5, $6 ; JLE .Lexit` (Q: will this jump to .Lexit?)
Control Flow Example

```c
void cond(int a, int *p) {
    if (p && a > 0)
        *p += a;
}
```

```
MOVL 8(%ebp), %edx
MOVL 12(%ebp), %eax
TESTL %eax, %eax
JE .LEXIT
TESTL %edx, %edx
JLE .LEXIT
ADDL %edx, (%eax)
.LEXIT
```
Control Flow Example: Test & JLE

OF, SF, ZF not set => JLE won’t jump
Control Flow Example

```c
void cond(int a, int *p) {
    if (p && a > 0)
        *p += a;
}
```

```assembly
MOVL 8(%ebp), %edx
MOVL 12(%ebp), %eax
TESTL %eax, %eax
JE .LEXIT
TESTL %edx, %edx
JLE .LEXIT
ADDL %edx, (%eax)
.LEXIT
```

```c
void goto_cond(int a, int *p) {
    if (p == 0) goto LEXIT;
    if (a<=0) goto LEXIT;
    *p += a;
    LEXIT:
    return;
}
```

<table>
<thead>
<tr>
<th>jX</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JE</td>
<td>ZF</td>
<td>Equal/Zero</td>
</tr>
<tr>
<td>JLE</td>
<td>(SF^OF)</td>
<td>ZF</td>
</tr>
</tbody>
</table>
Bomblab Overview

- Series of stages asking for an input string
  - input string can be anything: e.g., an array of chars or a number
  - ./bomb < mysolutions  # one input string per line

- An incorrect input string causes the bomb to “explode”
  - ½ point deducted for each explosion – they cannot be “undone”
  - it is possible to try multiple incorrect answers without explosion (carefully)

- What’s provided: ./bomb , bomb.c (partial source)
  - full C source is NOT provided

- What to do:
  - figure out the expected input string for each phase

- This must be done on shark machines

- Exercise great care
  - e.g., don’t just enter garbage and see what happens
Toolz: GDB

- `gdb <binary>`
- Ultra useful gdb cheatsheet
  - http://csapp.cs.cmu.edu/public/docs/gdbnotes-ia32.txt
- Useful commands:
  - advance EIP: `s`, `n`, `si`, `ni`, `c`, `run`
  - print things out:
    - print `<expression>`: e.g., print `variable_name`
    - `x` `/format` `<address>`: print contents of memory
      - how much to print and how to print is dictated by format
  - disassemble: `disas` address
  - layout:
    - layout split, layout regs, layout src, layout asm
    - focus next (fs n)
Toolz: GDB (continued)

- querying state:
  - info break
  - info frame (highly useful summary of the current activation record)
  - info program
  - info registers

- working with breakpoints:
  - break <address> ; break <function_name>
  - enable <breakpoint_number>
  - disable <breakpoint_number>
  - delete <breakpoint_number>

- start/stop execution:
  - run [command line args]: e.g., run partial_answer_file
  - quit (and confirm, if quitting a running executable)
GNU Debugger

Turning C into Object Code

Compiling

Download:

- esxserver_install_combined.txt
- freecolor_debug.tgz

google-analyze.py

grados_projideas.pdf

hotos11_comments_intro_onur.pdf

hotos11-paper79.pdf

hotos11.pdf

image.pdf

info

intern_agreement.pdf

text

Compiler (gcc -S)

Assembler (gcc or aas)

Program (p1.s p2.s)

Linker (gcc or ld)

makeAllDistances.sh

docs

Projects

Documents

http://www.cs.cmu.edu/~gdb

Carnegie Mellon

 GNU Debugger
GNU Debugger

Turning C into Object Code

`0x40128b <phase_1+27> return program (pl.c p2.c)`
GNU Debugger
Toolz: objdump, strings

- **objdump –d <binary>**
  - disassembles the <binary>, dumps assembly output
  - redirect stdout to an output file: `objdump –d ./bomb > bomb.S`
  - redirect stdout to a pipe: `objdump –d ./bomb | less`
  - `objdump –M intel` # if you want intel syntax
    - hint: stick with default

- **strings <binary>**
  - dumps collections of printable characters from <binary>
  - may include:
    - function names, string literals
    - password hashes? <grin>
Bomb Walkthrough

- How to get started with defusing the bomb
- How to blow up the bomb