

15-213 Recitation 3

Introduction to Computer Systems

Fahim Dalvi

12 September, 2013

Today

- Assembly Review
 - Basics
 - Operations
- Bomblab!
 - Basics
 - Tools
 - Walkthrough

Assembly – Architecture!

- Program counter
 - Always contains the address of the next instruction
 - *eip* (x86), *rip* (x86-64)
- Stack registers
 - Contains address of the bottom and the top of the stack
 - *esp* and *ebp* (x86)
- General purpose registers
 - *eax, ebx, ecx, edx, esi, edi* (x86)
 - *rax, rbx, rcx, rdx, rsi, rdi, r8, r9, r10, r11, r12, r13, r14, r15* and sometimes *rbp* (x86-64)
- Condition codes

Data types

- Integer data – 1,2 or 4 bytes (x86) or 8 bytes (x86-64)
- Addresses – 4 bytes (x86) or 8 bytes (x86-64)
- Floating point data
 - 4, 8 or 10 bytes
- No aggregate data types!
 - Means no arrays, no strings!

Operations - Accessing

- Registers
 - %eax, %ebx
 - Some instructions, as you will learn, use and store results in special registers
- Memory
 - If %eax contains an address that you want to deference, you can use (%eax)
 - General memory addressing format is D(Rb, Ri, S)
 - Rb is the base address register
 - Ri is the index address register
 - S is the index scale (1, 2, 4 or 8)
 - D is the constant offset
 - Equivalent in C style: $Rb[Ri * S + D]$
- Immediate values
 - Examples: \$0x15213, \$-11228
 - Like a C constant, except its prefixed by a \$
 - 1, 2 or 4 bytes (or 8 on x86-64)

Memory Operations

- `movl src,dest`
 - Example: `movl $0x15213, %eax`
 - Can move data between registers and memory
 - Only prohibited operation → Memory to Memory
- `leal src,dest`
 - Example: `leal (%eax,%eax,2),%eax`
 - Computes an address specified by `src` and saves it in `dst`
 - Does not actually dereference `src`!
 - Sometimes used by compilers as a fast alternative to `imul`
 - Examples above triples `%eax`

Arithmetic Operations

- Two operand commands → Always src,dest

<u>Format</u>	<u>Result</u>
<code>addl src,dest</code>	<code>dst+=src</code>
<code>subl src,dest</code>	<code>dst-=src</code>
<code>imull src,dest</code>	<code>dst*=src</code>
<code>sall src,dest</code>	<code>dst<<=src</code>
<code>sarl src,dest</code>	<code>dst>>=src</code>
<code>xorl src,dest</code>	<code>dst^=src</code>
<code>andl src,dest</code>	<code>dst&=src</code>
<code>orl src,dest</code>	<code>dst =src</code>

Arithmetic Operations

- One operand commands

<u>Format</u>	<u>Result</u>
<code>incl dst</code>	<code>dst++</code>
<code>decl dst</code>	<code>dst--</code>
<code>negl dst</code>	<code>dst = -dst</code>
<code>notl dst</code>	<code>dst = ~dst</code>

- Equivalent 64-bit operations are also available (e.g. `addq`)

Lets look at a complete example

```
void foo ( ) {
```

```
    int a = 0;
```

```
    int b = 2;
```

```
    int c = a - b ;
```

```
    int d = c << 2 ;
```

```
}
```

```
    pushq %rbp
```

```
    movq %rsp, %rbp
```

```
    movl $0, -16(%rbp)
```

```
    movl $2, -12(%rbp)
```

```
    movl -12(%rbp), %edx
```

```
    movl -16(%rbp), %eax
```

```
    subl %edx, %eax
```

```
    movl %eax, -8(%rbp)
```

```
    movl -8(%rbp), %eax
```

```
    sall $2, %eax
```

```
    movl %eax, -4(%rbp)
```

```
    leave
```

```
    ret
```

Condition codes

- Set as side-effect of arithmetic operations in the eflags register
- CF set on unsigned integer overflow
- ZF set if result was 0
- SF set if result was negative
- OF set on signed integer overflow
- *testl a,b* and *cmpl a,b* are similar to *andl a,b* and *subl a,b* but only set condition codes
- Use *set* reg* instructions to set register *reg* based on state of condition codes.

Conditionals

- Change the instruction pointer with the j* operations
 - jmp dst unconditionally jumps to the address dst
 - Use other jump variants (e.g. jne or jg) to conditionally jump
 - Usually a testl or cmpl followed by a conditional jump

Lets look at (another) complete example

```
void bar ( ) {  
  
    int a = 2;  
    int b = 0;  
    if (a > 7) {  
        b++;  
    }  
}
```

```
pushq %rbp  
movq %rsp, %rbp  
movl $2, -8(%rbp)  
movl $0, -4(%rbp)  
cmpl $7, -8(%rbp)  
jle .L3  
addl $1, -4(%rbp)  
.L3:  
leave  
ret
```

Bomblab!

- Series of stages, all asking for a password
- Give the wrong password and the bomb explodes
 - You lose a half point every time your bomb explodes
 - The bomb should never explode if you're careful
- We give you the binary, you have to find the passwords
- The binary ONLY runs on the shark machine

Bomblab - Tools

- Syntax: `$> gdb ./bomb`
- Useful commands
 - `run <args>` : Runs the bomb with specified command line arguments
 - `break <location>` : Stops the bomb just **before** the instruction at the specified location is about to be run
 - `info functions` : Lists the names of all functions.
 - `stepi` : Steps the program one instruction. `nexti` will do the same, but skipping over function calls.
 - `print <variable>` : Prints the contents of a variable
 - `x/<format> <address>` : Prints contents of the memory area starting at the address in a specified format
 - `disassemble <address>` : Displays the assembly instructions near the specified address
 - `layout <type>` : Changes the layout of GDB.
 - `help` and `help <command>` : Explains GDB usage.

Bomblab - Tools

- Strings
 - Dumps all strings in the binary
 - Function names, string literals, etc
- objdump
 - The -d option disassembles the bomb and outputs the assembly to the terminal
 - The -t option dumps the symbol table (all function and global variable names) to the terminal
 - You probably want to redirect the output into a file
 - `objdump -d ./bomb > bomb.asm`

Bomblab

Lets check it out!