Machine-Level Representation of Programs: Bomblab, addressing modes, arithmetic

Yipeng Huang

Rutgers University

March 23, 2023

Table of contents

Announcements

PA4 bomb lab

- PA4 bomb lab out and live. Due Wednesday, April 5.
- Due dates for rest of semester up to date on class syllabus. `https://yipenghuang.com/teaching/2023-spring/`

Short quiz next week

Short quiz on assembly basics and control spanning Monday 3/27 to Friday 3/31.

Class session plan

- Today, 3/23: Bomb lab demo, addressing modes (Book chapter 3.4), control flow.
- Monday, 3/27: Control flow (conditionals, if, for, while, do loops) in assembly. (Book chapter 3.6)
- Thursday, 3/30: Function calls in assembly. (Book chapter 3.7)
- Monday, 4/3: Arrays and data structures in assembly. (Book chapter 3.8)

Table of contents

Programming Assignment 4: Defusing a Binary Bomb

Goals

- Learning to learn to use important tools like GDB.
- Understand how high level programming constructs compile down to assembly instructions.
- Practice reverse engineering and debugging.

Setup

- Programming assignment description PDF on Canvas.
- Web interface for obtaining bomb and seeing progress.
- Unpacking.

Unpacking and gathering information about your bomb

What comes in the package

- `bomb.c`: Skeleton source code
- `bomb`: The executable binary

`objdump -t bomb > symbolTable.txt`

- 000000000040143a g F .text 0000000000000022 explode_bomb

`objdump -d bomb > bomb.s`

Different phases correspond to different topics about assembly programming in the CS211 lecture slides, in the CS:APP slides, and in the CS:APP book.

- phase_1
- phase_2
- explode_bomb

`strings -t x bomb > strings.txt`

Example phase_1 in example bomb from CS:APP website

```
0000000000400ee0 <phase_1>:
  400ee0: 48 83 ec 08              sub    $0x8,%rsp
  400ee4: be 00 24 40 00           mov    $0x402400,%esi
  400ee9: e8 4a 04 00 00           callq  401338 <strings_not_equal>
  400eee: 85 c0                    test   %eax,%eax
  400ef0: 74 05                    je     400ef7 <phase_1+0x17>
  400ef2: e8 43 05 00 00           callq  40143a <explode_bomb>
  400ef7: 48 83 c4 08              add    $0x8,%rsp
  400efb: c3                       retq
```

Understanding what we're seeing here

- Don't let `callq` to `explode_bomb` at instruction address `400ef2` happen...
- so, must ensure `je` instruction does jump, so we want `test` instruction to set ZF condition code to 0.
- so, must ensure `callq` to `strings_not_equal()` function returns 0.

Using GDB to carefully step through execution of the bomb program

```
gdb bomb
```

Finding help in GDB

- `help`: Menu of documentation.
- `help layout`: Useful tip to use either `layout asm` or `layout regs` for this assignment.
- `help aliases`
- `help running`
- `help data`
- `help stack`

Using GDB to carefully step through execution of the bomb program

`gdb bomb`

Setting breakpoints and running / stepping through code

- `break explode_bomb` or `b explode_bomb`: Pause execution upon entering explode_bomb function.
- `break phase_1` or `b phase_1`: Pause execution upon entering phase_1 function.
- `run mysolution.txt` or `r mysolution.txt`: Run the code passing the solution file.
- `continue` or `c`: Continue until the next breakpoint.
- `nexti` or `ni`: Step one instruction, but proceed through subroutine calls.
- `stepi` or `si`: Step one instruction exactly. Steps into functions / subroutine calls.

Using GDB to carefully step through execution of the bomb program

`gdb bomb`

Printing and examining registers and memory addresses

- `print /x $eax` or `p /x $eax`: Print value of %eax register as hex.
- `print /d $eax` or `p /d $eax`: Print value of %eax register as decimal.
- `x /s 0x402400`: Examine memory address 0x402400 as a string.

Table of contents

Immediate, register, and memory

Immediate

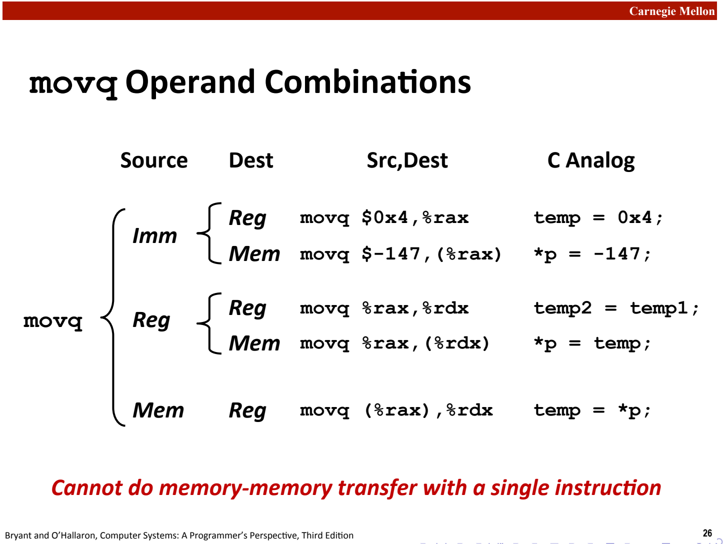
Constant integer values. Example: 2_addressing_modes.c immediate()

Register

One of the registers of appropriate size for data type. Example: 1_swap.c

Memory

Access to memory at calculated



Carnegie Mellon

movq Operand Combinations

	Source	Dest	Src,Dest	C Analog
movq	*Imm*	*Reg*	`movq $0x4,%rax`	`temp = 0x4;`
		Mem	`movq $-147,(%rax)`	`*p = -147;`
	Reg	*Reg*	`movq %rax,%rdx`	`temp2 = temp1;`
		Mem	`movq %rax,(%rdx)`	`*p = temp;`
	Mem	*Reg*	`movq (%rax),%rdx`	`temp = *p;`

Cannot do memory-memory transfer with a single instruction

Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

26

Addressing modes

Carnegie Mellon

Simple Memory Addressing Modes

- **Normal (R) Mem[Reg[R]]**
  - Register R specifies memory address
  - Aha! Pointer dereferencing in C

```
movq (%rcx),%rax
```

- **Displacement D(R) Mem[Reg[R]+D]**
  - Register R specifies start of memory region
  - Constant displacement D specifies offset

```
movq 8(%rbp),%rdx
```

Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition 27

Normal

Simple pointers. Example: 2_addressing_modes.c immediate()

Displacement

Array access with constant index. Example: 2_addressing_modes.c displacement()

Addressing modes

Complete Memory Addressing Modes

- **Most General Form**

  D(Rb,Ri,S) Mem[Reg[Rb]+S*Reg[Ri]+ D]

  - D: Constant "displacement" 1, 2, or 4 bytes
  - Rb: Base register: Any of 16 integer registers
  - Ri: Index register: Any, except for `%rsp`
  - S: Scale: 1, 2, 4, or 8 (*why these numbers?*)

- **Special Cases**

  (Rb,Ri) Mem[Reg[Rb]+Reg[Ri]]

  D(Rb,Ri) Mem[Reg[Rb]+Reg[Ri]+D]

  (Rb,Ri,S) Mem[Reg[Rb]+S*Reg[Ri]]

Indexed

Array access with variable index. Example: 2_addressing_modes.c index()

Addressing modes

Address Computation Examples

%rdx	0xf000
%rcx	0x0100

Expression	Address Computation	Address
0x8(%rdx)	0xf000 + 0x8	0xf008
(%rdx,%rcx)	0xf000 + 0x100	0xf100
(%rdx,%rcx,4)	0xf000 + 4*0x100	0xf400
0x80(,%rdx,2)	2*0xf000 + 0x80	0x1e080

2_addressing_modes.c: Imm→Mem

C code

```
void immediate ( long * ptr ) {
    *ptr = 0xFFFFFFFFFFFFFFFF;
}
```

Assembly code

```
immediate:
    movq $-1, (%rdi)
    ret
```

- $ indicates the immediate value; corresponds to literals in C
- (%rdi) indicates memory location at address stored in %rdi register

2_addressing_modes.c: Imm→Mem (with displacement)

C code

```
void displacement_l ( long * ptr ) {
  ptr[1] = 0xFFFFFFFFFFFFFFFF;
}
```

Assembly code

```
displacement_l:
    movq $-1, 8(%rdi)
    ret
```

- 8(%rdi) indicates memory location at address stored in %rdi register + 8

2_addressing_modes.c: Imm→Mem (with displacement)

function signature	assembly code
void displacement_c (char * ptr);	movb $-1, 1(%rdi)
void displacement_s (short * ptr);	movw $-1, 2(%rdi)
void displacement_i (int * ptr);	movl $-1, 4(%rdi)
void displacement_l (long * ptr);	movq $-1, 8(%rdi)

2_addressing_modes.c: Imm→Mem (with index)

C code

```
void index_l ( long * ptr, long index ) {
  ptr[index] = 0xFFFFFFFFFFFFFFFF;
}
```

Assembly code

```
index_l:
    movq $-1, (%rdi,%rsi,8)
    ret
```

- (%rdi,%rsi,8) indicates memory location at address stored in %rdi register + 8 × value stored in %rsi register

2_addressing_modes.c: Imm→Mem (with index)

function signature	assembly code
void index_c (char * ptr, long index);	movb $-1, (%rdi,%rsi)
void index_s (short * ptr, long index);	movw $-1, (%rdi,%rsi,2)
void index_i (int * ptr, long index);	movl $-1, (%rdi,%rsi,4)
void index_l (long * ptr, long index);	movq $-1, (%rdi,%rsi,8)

2_addressing_modes.c: Imm→Mem (with displacement and index)

C code

```
void displacement_and_index ( long * ptr, long index ) {
  ptr[index+1] = 0xFFFFFFFFFFFFFFFF;
}
```

Assembly code

```
displacement_and_index:
    movq $-1, 8(%rdi,%rsi,8)
    ret
```

- 8(%rdi,%rsi,8) indicates memory location at address stored in %rdi register + 8 $\times$ value stored in %rsi register + 8

Table of contents

`3_leaq.s`: Borrowing memory address calculation to efficiently implement arithmetic

Address Computation Instruction

- **`leaq` Src, Dst**
  - Src is address mode expression
  - Set Dst to address denoted by expression
- **Uses**
  - Computing addresses without a memory reference
    - E.g., translation of `p = &x[i];`
  - Computing arithmetic expressions of the form x + k*y
    - k = 1, 2, 4, or 8
- **Example**

```
long m12(long x)
{
  return x*12;
}
```

Converted to ASM by compiler:

```
leaq (%rdi,%rdi,2), %rax # t <- x+x*2
salq $2, %rax            # return t<<2
```

Example: 3_leaq.c

Load effective address

```c
long * leaq (
    long * ptr, long index
) {
    return &ptr[index+1];
}
```

```c
long mulAdd (
    long base, long index
) {
    return base+index*8+8;
}
```

Both C code functions above translate to the assembly on the right.

```
leaq:
mulAdd:
    leaq 8(%rdi,%rsi,8), %rax
    ret
```

Explanation

- `leaq src,dest` takes the effective address of the memory (index, displacement) expression of src and puts it in dest.
- `leaq` has shorter latency (takes fewer CPU cycles) than `imulq`, so GCC will use leaq whenever it can to calculate expressions like $y + ax + b$.