Machine-level representation of programs: Bomblab, addressing mode recap, arithmetic

Yipeng Huang

Rutgers University

March 22, 2022

Table of contents

Announcements

PA4 bomb lab

- PA4 bomb lab out and live. Due Tuesday, April 5.
- Due dates for rest of semester up to date on class syllabus. `https://yipenghuang.com/teaching/2022-spring/`

Short quiz next week

Short quiz on assembly basics and control spanning Tuesday 3/29 to Thursday 3/31.

Class session plan

- Today, 3/22: Bomb lab demo, recap addressing modes, wrap up arithmetic.
- Thursday, 3/24: Control flow (conditionals, if, for, while, do loops) in assembly. (Book chapter 3.6)
- Tuesday, 3/29: Function calls in assembly. (Book chapter 3.7)
- Thursday, 3/31: 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

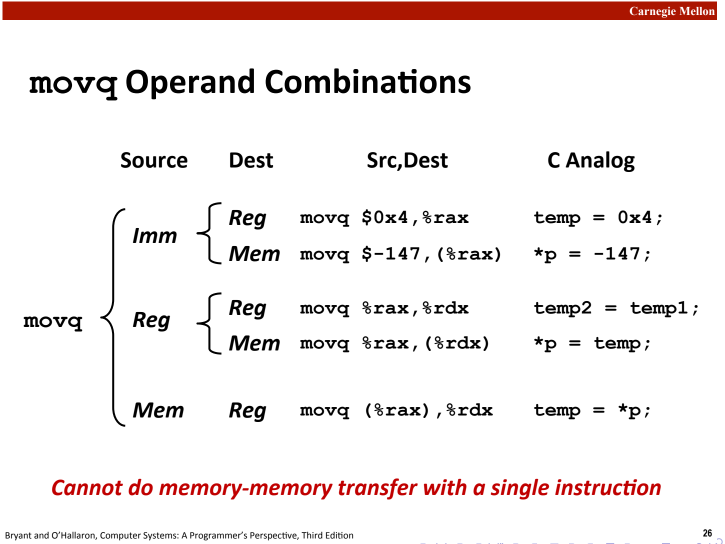
Constant integer values. Example: 2_address-ing_modes.c immediate()

Register

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

Memory

Access to memory at calculated



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
```


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 × value stored in %rsi register + 8

Table of contents

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

Carnegie Mellon

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

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

```
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 $y + ax + b$.

Table of contents

Sign extension due to unsigned and signed data types

Converting to a data type with more bits

```
unsigned short uc_to_us (
    unsigned char input
) {
    return input;
}
```

```
signed short sc_to_ss (
    signed char input
) {
    return input;
}
```

$$
\begin{aligned}
255 &= 1111\_1111_2 \\
&= 0000\_0000\_1111\_1111_2 \\
&= 255
\end{aligned}
$$

$$
\begin{aligned}
127 &= 0111\_1111_2 \\
&= 0000\_0000\_0111\_1111_2 \\
&= 127
\end{aligned}
$$

$$
\begin{aligned}
-128 &= 1000\_0000_2 \\
&= 1111\_1111\_1000\_0000_2 \\
&= -128
\end{aligned}
$$

Sign extension due to unsigned and signed data types

Converting to a data type with more bits

```
unsigned short uc_to_us (
    unsigned char input
) {
    return input;
}
```

```
signed short sc_to_ss (
    signed char input
) {
    return input;
}
```

function signature	assembly code
unsigned short uc_to_us (unsigned char input);	movzbl %dil, %eax
signed short uc_to_ss (unsigned char input);	movzbl %dil, %eax
unsigned short sc_to_us (signed char input);	movsbw %dil, %ax
signed short sc_to_ss (signed char input);	movsbw %dil, %ax

- movz: zero extension in the MSBs
- movs: signed extension in the MSBs

Table of contents

Left shift operation

```
unsigned long sl_ul (
    unsigned long in0,
    unsigned long in1
) {
    return in0<<in1;
}
```

```
signed long sl_sl (
    signed long in0,
    signed long in1
) {
    return in0<<in1;
}
```

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

```
sl_ul:
sl_sl:
    movq %rdi, %rax
    movb %sil, %cl
    salq %cl, %rax
    ret
```

Explanation

- `movq`: in0 $\rightarrow$ %rdi $\rightarrow$ %rax
- `movb`: in1 $\rightarrow$ %sil $\rightarrow$ %cl
- `salq src,dest`:
  (dest << src) $\rightarrow$ dest
- Why only use movb for in1?

Right shift operation

Right shift of unsigned types yields logical (zero-filled) right shift

```
unsigned long sr_ul (
    unsigned long in0,
    unsigned long in1
) {
    return in0>>in1;
}
```

```
sr_ul:
    movq %rdi, %rax
    movb %sil, %cl
    shrq %cl, %rax
    ret
```

Right shift of signed types yields arithmetic (sign-extended) right shift

```
signed long sr_sl (
    signed long in0,
    signed long in1
) {
    return in0>>in1;
}
```

```
sr_sl:
    movq %rdi, %rax
    movb %sil, %cl
    sarq %cl, %rax
    ret
```

Bitwise operations

Assembly instruction	Instruction effect
`notq dest`	$\sim$ dest $\rightarrow$ dest
`andq src,dest`	src&dest $\rightarrow$ dest
`orq src,dest`	src$\vert$dest $\rightarrow$ dest
`xorq src,dest`	src $\wedge$ dest $\rightarrow$ dest

Integer arithmetic operations

Assembly instruction	Instruction effect
`incq dest`	dest $+ 1 \rightarrow$ dest
`decq dest`	dest $- 1 \rightarrow$ dest
`negq dest`	$-$dest $\rightarrow$ dest
`addq src,dest`	src $+$ dest $\rightarrow$ dest
`subq src,dest`	src $-$ dest $\rightarrow$ dest
`imulq src,dest`	src $\times$ dest $\rightarrow$ dest

Table of contents

What is control flow?

Control flow is:

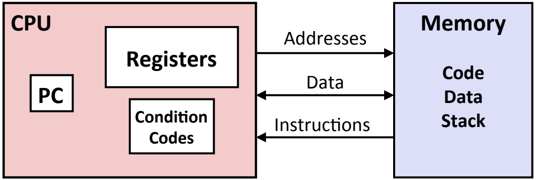
- Change in the sequential execution of instructions.
- Change in the steady incrementation of the program counter / instruction pointer (%rip register).

Control primitives in assembly build up to enable C and Java control statements:

- if-else statements
- do-while loops
- while loops
- for loops
- switch statements

Condition codes

Assembly/Machine Code View



Programmer-Visible State

- **PC: Program counter**
  - Address of next instruction
  - Called “RIP” (x86-64)
- **Register file**
  - Heavily used program data
- **Condition codes**
  - Store status information about most recent arithmetic or logical operation
  - Used for conditional branching

- **Memory**
  - Byte addressable array
  - Code and user data
  - Stack to support procedures

Condition codes

Automatically set by most arithmetic instructions.

Applicable types	Condition code	Name	Use
Signed and unsigned	`ZF`	Zero flag	The most recent operation yielded zero.
Unsigned types	`CF`	Carry flag	The most recent operation generated a carry out of the most significant bit. Used to detect overflow for unsigned operations
Signed types	`SF`	Sign flag	The most recent operation yielded a negative value.
Signed types	`OF`	Overflow flag	The most recent operation yielded a two's complement positive or negative overflow.

Table: Condition codes important for control flow

Comparison instructions

`cmpq source1, source2`

Performs source2 − source1, and sets the condition codes without setting any destination register.

Test for equality

```c
short equal_sl (
    long x,
    long y
) {
    return x==y;
}
```

C code function above translates to the assembly on the right.

```
equal_sl:
    xorl %eax, %eax
    cmpq %rsi, %rdi
    sete %al
    ret
```

Explanation

- `xorl %eax, %eax`: Zeros the 32-bit register %eax.
- `cmpq %rsi, %rdi`: Calculates %rdi − %rsi ($x - y$), sets condition codes without updating any destination register.
- `sete %al`: Sets the 8-bit %al subset of %eax if op yielded zero.

Test if unsigned x is below unsigned y

```
short below_ul (
    unsigned long x,
    unsigned long y
) {
    return x<y;
}
```

```
short nae_ul (
    unsigned long x,
    unsigned long y
) {
    return !(x>=y);
}
```

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

```
below_ul:
nae_ul:
    xorl %eax, %eax
    cmpq %rsi, %rdi
    setb %al
    ret
```

Explanation

- `xorl %eax, %eax`: Zeros %eax.
- `cmpq %rsi, %rdi`: Calculates %rdi − %rsi ($x - y$), sets condition codes without updating any destination register.
- `setb %al`: Sets %al if CF flag set indicating unsigned overflow.

Side review: De Morgan's laws

- $\neg A \wedge \neg B \iff \neg(A \vee B)$
- $(\sim A)\&(\sim B) \iff \sim (A|B)$

Set instructions

`cmp source1, source2` performs source2 – source1, sets condition codes.

Applicable types	Set instruction	Logical condition	Intutive condition
Signed and unsigned	`sete` / `setz`	`ZF`	Equal / zero
Signed and unsigned	`setne` / `setnz`	~ `ZF`	Not equal / not zero
Unsigned	`setb` / `setnae`	`CF`	Below
Unsigned	`setbe` / `setna`	`CF`|`ZF`	Below or equal
Unsigned	`seta` / `setnbe`	~ `CF`& ~ `ZF`	Above
Unsigned	`setnb` / `setae`	~ `CF`	Above or equal
Signed	`sets`	`SF`	Negative
Signed	`setns`	~ `SF`	Nonegative
Signed	`setl` / `setnge`	`SF ^ OF`	Less than
Signed	`setle` / `setng`	(`SF ^ OF`)|`ZF`	Less than or equal
Signed	`setg` / `setnle`	~ (`SF ^ OF`)& ~ `ZF`	Greater than
Signed	`setge` / `setnl`	~ (`SF ^ OF`)	Greater than or equal

Table: Set instructions