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

Yipeng Huang

Rutgers University

October 30, 2025

Table of contents

Announcements

Class session plan

- Today, 10/30: addressing modes (Book chapter 3.4), arithmetic (Book chapter 3.5). Bomblab phase_1.
- Tuesday, 11/4: Control flow (conditionals, if, for, while, do loops, switch statements) in assembly. (Book chapter 3.6). Bomblab phase_2, phase_3.
- Thursday, 11/6: Function calls in assembly. (Book chapter 3.7). Bomblab phase_4.
- Tuesday, 11/11: Arrays and data structures in assembly. (Book chapter 3.8). Bomblab phase_5, phase_6.

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

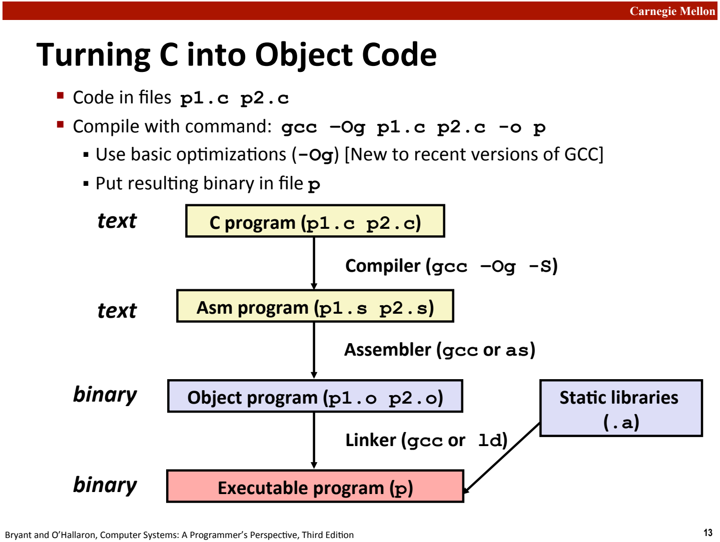
`gdb bomb`

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

Unraveling the compilation chain



- gcc -0g -S swap.c
- objdump -d swap

Let's go to CS:APP textbook lecture slides (05-machine-basics.pdf) slide 28

Data movement instructions

Does unsigned / signed matter?

1. `void swap_uc ( unsigned char*a, unsigned char*b );`
2. `void swap_sc ( signed char*a, signed char*b );`

Swapping different data sizes

1. `void swap_c ( char*a, char*b );`
2. `void swap_s ( short*a, short*b );`
3. `void swap_i ( int*a, int*b );`
4. `void swap_l ( long*a, long*b );`

Table of contents

Data size and x86 / x86-64 registers

Assembly syntax

Instruction Source, Dest

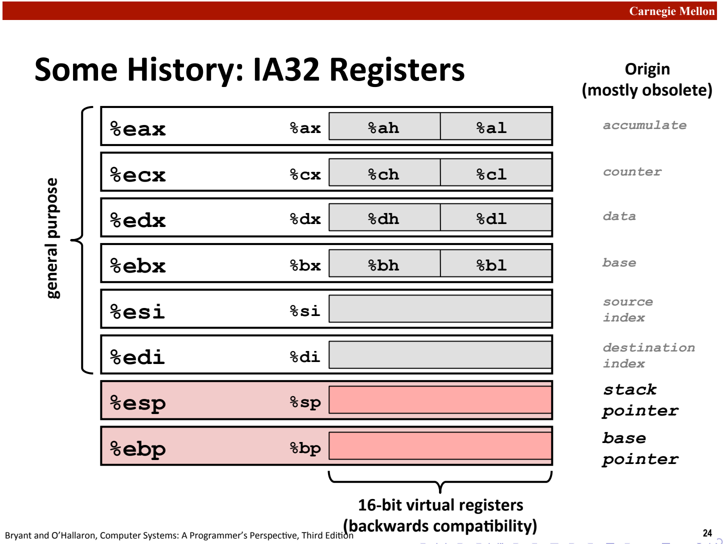
```
swap_l:
 movq (%rsi), %rax
 movq (%rdi), %rdx
 movq %rdx, (%rsi)
 movq %rax, (%rdi)
 ret
```

swap	data type	mov operation	registers
swap_uc	unsigned char	movb (move byte)	%al, %dl
swap_sc	signed char	movb (move byte)	%al, %dl
swap_c	char	movb (move byte)	%al, %dl
swap_s	short	movw (move word)	%ax, %dx
swap_i	int	movl	%eax, %edx
swap_l	long	movq	%rax, %rdx

Data size and IA32, x86, and x86-64 registers

data type	registers
char	%al, %dl
short	%ax, %dx
int	%eax, %edx
long	%rax, %rdx

Note the backward compatibility.



Data size and IA32, x86, and x86-64 registers

data type	registers
char	%al, %dl
short	%ax, %dx
int	%eax, %edx
long	%rax, %rdx

Note the backward compatibility.

x86-64 Integer Registers

%rax	%eax	%r8	%r8d
%rbx	%ebx	%r9	%r9d
%rcx	%ecx	%r10	%r10d
%rdx	%edx	%r11	%r11d
%rsi	%esi	%r12	%r12d
%rdi	%edi	%r13	%r13d
%rsp	%esp	%r14	%r14d
%rbp	%ebp	%r15	%r15d

- Can reference low-order 4 bytes (also low-order 1 & 2 bytes)

Data size and IA32, x86, and x86-64 registers

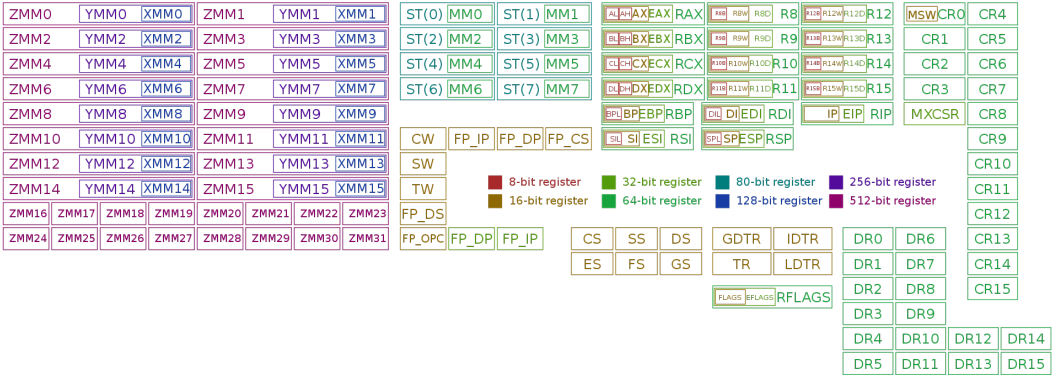


Figure: x86-64 with SIMD extensions registers. Image credit: `https://commons.wikimedia.org/wiki/File:Table_of_x86_Registers_svg.svg`

Table of contents

Sign extension due to unsigned and signed data types

Converting to a data type with more bits

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

```c
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

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

```c
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

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

```c
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`	$\text{dest} + 1 \to \text{dest}$
`decq dest`	$\text{dest} - 1 \to \text{dest}$
`negq dest`	$-\text{dest} \to \text{dest}$
`addq src,dest`	$\text{src} + \text{dest} \to \text{dest}$
`subq src,dest`	$\text{src} - \text{dest} \to \text{dest}$
`imulq src,dest`	$\text{src} \times \text{dest} \to \text{dest}$

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

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

Addressing modes

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

Carnegie Mellon

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]]**

Bryant and O’Hallaron, Computer Systems: A Programmer’s Perspective, Third Edition 36

Indexed

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

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

```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 $y + ax + b$.