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

Yipeng Huang

Rutgers University

March 22, 2022
Table of contents

Announcements

Programming Assignment 4: Defusing a Binary Bomb
   Unpacking your bomb
   Using GDB

2_addressing_modes.s: Understanding source dest operands and memory addressing modes

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

MOV instruction sign extension

Arithmetic instructions
   Shift operations
   Bitwise operations
   Integer arithmetic operations

Comparisons and program control flow
   What is control flow?
   Condition codes
   Comparison and set instructions
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

Announcements

Programming Assignment 4: Defusing a Binary Bomb
  Unpacking your bomb
  Using GDB

2_addressing_modes.s: Understanding source dest operands and memory addressing modes

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

MOV instruction sign extension

Arithmetic instructions
  Shift operations
  Bitwise operations
  Integer arithmetic operations

Comparisons and program control flow
  What is control flow?
  Condition codes
  Comparison and set instructions
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

```bash
objdump -t bomb > symbolTable.txt
```

- `000000000040143a g F .text 0000000000000022 explode_bomb`

```bash
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**

```bash
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

Announcements

Programming Assignment 4: Defusing a Binary Bomb
  Unpacking your bomb
  Using GDB

2_addressing_modes.s: Understanding source dest operands and memory addressing modes

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

MOV instruction sign extension

Arithmetic instructions
  Shift operations
  Bitwise operations
  Integer arithmetic operations

Comparisons and program control flow
  What is control flow?
  Condition codes
  Comparison and set instructions
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
Addressing modes

Simple Memory Addressing Modes

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

  \[
  \text{movq } (\%rcx), \%rax
  \]

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

  \[
  \text{movq } 8(\%rbp), \%rdx
  \]
Addressing modes

Complete Memory Addressing Modes

- **Most General Form**
  
  \[ D(R_b,R_i,S) \quad \text{Mem}[\text{Reg}[R_b]+S^*\text{Reg}[R_i]+D] \]
  
  - **D:** Constant “displacement” 1, 2, or 4 bytes
  - **R_b:** Base register: Any of 16 integer registers
  - **R_i:** Index register: Any, except for %\text{rsp}
  - **S:** Scale: 1, 2, 4, or 8 (why these numbers?)

- **Special Cases**
  
  - **(R_b,R_i)**  \[ \text{Mem}[\text{Reg}[R_b]+\text{Reg}[R_i]] \]
  - **D(R_b,R_i)**  \[ \text{Mem}[\text{Reg}[R_b]+\text{Reg}[R_i]+D] \]
  - **(R_b,R_i,S)**  \[ \text{Mem}[\text{Reg}[R_b]+S^*\text{Reg}[R_i]] \]

Indexed

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

Address Computation Examples

<table>
<thead>
<tr>
<th>Expression</th>
<th>Address Computation</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x8(%rdx)</td>
<td>0xf000 + 0x8</td>
<td>0xf008</td>
</tr>
<tr>
<td>(%rdx,%rcx)</td>
<td>0xf000 + 0x100</td>
<td>0xf100</td>
</tr>
<tr>
<td>(%rdx,%rcx,4)</td>
<td>0xf000 + 4*0x100</td>
<td>0xf400</td>
</tr>
<tr>
<td>0x80(,%rdx,2)</td>
<td>2*0xf000 + 0x80</td>
<td>0x1e080</td>
</tr>
</tbody>
</table>
C code

```c
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
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)

<table>
<thead>
<tr>
<th>function signature</th>
<th>assembly code</th>
</tr>
</thead>
<tbody>
<tr>
<td>void displacement_c ( char * ptr );</td>
<td>movb $-1, 1(%rdi)</td>
</tr>
<tr>
<td>void displacement_s ( short * ptr );</td>
<td>movw $-1, 2(%rdi)</td>
</tr>
<tr>
<td>void displacement_i ( int * ptr );</td>
<td>movl $-1, 4(%rdi)</td>
</tr>
<tr>
<td>void displacement_l ( long * ptr );</td>
<td>movq $-1, 8(%rdi)</td>
</tr>
</tbody>
</table>
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)

<table>
<thead>
<tr>
<th>function signature</th>
<th>assembly code</th>
</tr>
</thead>
<tbody>
<tr>
<td>void index_c ( char * ptr, long index );</td>
<td>movb $-1, (%rdi,%rsi)</td>
</tr>
<tr>
<td>void index_s ( short * ptr, long index );</td>
<td>movw $-1, (%rdi,%rsi,2)</td>
</tr>
<tr>
<td>void index_i ( int * ptr, long index );</td>
<td>movl $-1, (%rdi,%rsi,4)</td>
</tr>
<tr>
<td>void index_l ( long * ptr, long index );</td>
<td>movq $-1, (%rdi,%rsi,8)</td>
</tr>
</tbody>
</table>
C code

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

Assembly code

```assembly
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
Announcements

Programming Assignment 4: Defusing a Binary Bomb
  Unpacking your bomb
  Using GDB

2_addressing_modes.s: Understanding source dest operands and memory addressing modes
3_leaq.s: Borrowing memory address calculation to efficiently implement arithmetic

MOV instruction sign extension

Arithmetic instructions
  Shift operations
  Bitwise operations
  Integer arithmetic operations

Comparisons and program control flow
  What is control flow?
  Condition codes
  Comparison and set instructions
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**

```c
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
```
Load effective address

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

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

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

**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
  Announcements
  Programming Assignment 4: Defusing a Binary Bomb
    Unpacking your bomb
    Using GDB
  2_addressing_modes.s: Understanding source dest operands and memory addressing modes
  3_leaq.s: Borrowing memory address calculation to efficiently implement arithmetic
  MOV instruction sign extension
  Arithmetic instructions
    Shift operations
    Bitwise operations
    Integer arithmetic operations
  Comparisons and program control flow
    What is control flow?
    Condition codes
    Comparison and set instructions
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;
}

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

---

\[ 255 = 1111_{1111} \]
\[ = 0000_{0000}1111_{1111} \]
\[ = 255 \]

\[ 127 = 0111_{1111} \]
\[ = 0000_{0000}0111_{1111} \]
\[ = 127 \]

\[ -128 = 1000_{0000} \]
\[ = 1111_{1111}1000_{0000} \]
\[ = -128 \]
Sign extension due to unsigned and signed data types

Converting to a data type with more bits

```
1 unsigned short uc_to_us ( unsigned char input ) {
2       return input;
3 } }

1 signed short sc_to_ss ( signed char input ) {
2       return input;
3 } }
```

<table>
<thead>
<tr>
<th>function signature</th>
<th>assembly code</th>
</tr>
</thead>
<tbody>
<tr>
<td>unsigned short uc_to_us ( unsigned char input );</td>
<td>movzbl %dil, %eax</td>
</tr>
<tr>
<td>signed short uc_to_ss ( unsigned char input );</td>
<td>movzbl %dil, %eax</td>
</tr>
<tr>
<td>unsigned short sc_to_us ( signed char input );</td>
<td>movsbw %dil, %ax</td>
</tr>
<tr>
<td>signed short sc_to_ss ( signed char input );</td>
<td>movsbw %dil, %ax</td>
</tr>
</tbody>
</table>

- movz: zero extension in the MSBs
- movs: signed extension in the MSBs
Table of contents

Announcements
Programming Assignment 4: Defusing a Binary Bomb
  Unpacking your bomb
  Using GDB
2_addressing_modes.s: Understanding source dest operands and memory addressing modes
3_leaq.s: Borrowing memory address calculation to efficiently implement arithmetic
MOV instruction sign extension
Arithmetic instructions
  Shift operations
  Bitwise operations
  Integer arithmetic operations
Comparisons and program control flow
  What is control flow?
  Condition codes
  Comparison and set instructions
Left shift operation

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

---

`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; }`

---

`sl_ul:`

```
movq %rdi, %rax
movb %sil, %cl
salq %cl, %rax
ret
```

**Explanation**

- **movq:** `in0 → %rdi → %rax`
- **movb:** `in1 → %sil → %cl`
- **salq src,dest:** `(dest ▼ src) → dest`
- **Why only use movb for in1?**
Right shift operation

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

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

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

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

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

```assembly
sr_sl:
    movq %rdi, %rax
    movb %sil, %cl
    sarq %cl, %rax
    ret
```
## Bitwise operations

<table>
<thead>
<tr>
<th>Assembly instruction</th>
<th>Instruction effect</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>notq dest</code></td>
<td><code>~ dest → dest</code></td>
</tr>
<tr>
<td><code>andq src,dest</code></td>
<td><code>src&amp;dest → dest</code></td>
</tr>
<tr>
<td><code>orq src,dest</code></td>
<td>`src</td>
</tr>
<tr>
<td><code>xorq src,dest</code></td>
<td><code>src ∧ dest → dest</code></td>
</tr>
</tbody>
</table>
### Integer arithmetic operations

<table>
<thead>
<tr>
<th>Assembly instruction</th>
<th>Instruction effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>incq dest</td>
<td>dest + 1 → dest</td>
</tr>
<tr>
<td>decq dest</td>
<td>dest − 1 → dest</td>
</tr>
<tr>
<td>negq dest</td>
<td>−dest → dest</td>
</tr>
<tr>
<td>addq src,dest</td>
<td>src + dest → dest</td>
</tr>
<tr>
<td>subq src,dest</td>
<td>src − dest → dest</td>
</tr>
<tr>
<td>imulq src,dest</td>
<td>src × dest → dest</td>
</tr>
</tbody>
</table>
Table of contents
Announcements

Programming Assignment 4: Defusing a Binary Bomb
  Unpacking your bomb
  Using GDB
2_addressing_modes.s: Understanding source dest operands and memory addressing modes
3_leaq.s: Borrowing memory address calculation to efficiently implement arithmetic
MOV instruction sign extension
Arithmetic instructions
  Shift operations
  Bitwise operations
  Integer arithmetic operations
Comparisons and program control flow
  What is control flow?
  Condition codes
  Comparison and set instructions
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

Figure: Assembly language view of CPU and memory. Image credit CS:APP
## Condition codes

Automatically set by most arithmetic instructions.

<table>
<thead>
<tr>
<th>Applicable types</th>
<th>Condition code</th>
<th>Name</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signed and unsigned</td>
<td>ZF</td>
<td>Zero flag</td>
<td>The most recent operation yielded zero.</td>
</tr>
<tr>
<td>Unsigned types</td>
<td>CF</td>
<td>Carry flag</td>
<td>The most recent operation generated a carry out of the most significant bit. Used to detect overflow for unsigned operations</td>
</tr>
<tr>
<td>Signed types</td>
<td>SF</td>
<td>Sign flag</td>
<td>The most recent operation yielded a negative value.</td>
</tr>
<tr>
<td>Signed types</td>
<td>OF</td>
<td>Overflow flag</td>
<td>The most recent operation yielded a two’s complement positive or negative overflow.</td>
</tr>
</tbody>
</table>

**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

```c
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:
```
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

- $\neg A \land \neg B \iff \neg(A \lor B)$
- $(\sim A) \& (\sim B) \iff \sim (A \mid B)$
### Set instructions

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

<table>
<thead>
<tr>
<th>Applicable types</th>
<th>Set instruction</th>
<th>Logical condition</th>
<th>Intuitive condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>Signed and unsigned</td>
<td><code>sete / setz</code></td>
<td>ZF</td>
<td>Equal / zero</td>
</tr>
<tr>
<td>Signed and unsigned</td>
<td><code>setne / setnz</code></td>
<td>~ZF</td>
<td>Not equal / not zero</td>
</tr>
<tr>
<td>Unsigned</td>
<td><code>setb / setnae</code></td>
<td>CF</td>
<td>Below</td>
</tr>
<tr>
<td>Unsigned</td>
<td><code>setbe / setna</code></td>
<td>CF</td>
<td>ZF</td>
</tr>
<tr>
<td>Unsigned</td>
<td><code>seta / setnbe</code></td>
<td>~ CF &amp; ~ ZF</td>
<td>Above</td>
</tr>
<tr>
<td>Unsigned</td>
<td><code>setnb / setae</code></td>
<td>~ CF</td>
<td>Above or equal</td>
</tr>
<tr>
<td>Signed</td>
<td><code>sets</code></td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>Signed</td>
<td><code>setns</code></td>
<td>~ SF</td>
<td>Nonegative</td>
</tr>
<tr>
<td>Signed</td>
<td><code>setl / setnge</code></td>
<td>SF ^ OF</td>
<td>Less than</td>
</tr>
<tr>
<td>Signed</td>
<td><code>setle / setng</code></td>
<td>(SF ^ OF)</td>
<td>ZF</td>
</tr>
<tr>
<td>Signed</td>
<td><code>setg / setnle</code></td>
<td>~ (SF ^ OF)&amp; ~ ZF</td>
<td>Greater than</td>
</tr>
<tr>
<td>Signed</td>
<td><code>setge / setnl</code></td>
<td>~ (SF ^ OF)</td>
<td>Greater than or equal</td>
</tr>
</tbody>
</table>

#### Table: Set instructions