Machine-level representation of programs: control, comparisons, branching, loops

Yipeng Huang

Rutgers University

March 24, 2022
Table of contents
Announcements
Comparisons and program control flow
  What is control flow?
  Condition codes
  Comparison and set instructions
Modifying control flow via conditional branch statements
  Jump instructions
  Conditional branch statements
Programming Assignment 4: Defusing a Binary Bomb
Modifying data flow via conditional move statements
  Conditional move statements
Loop statements
  Compiling for loops to while loops
  Compiling while loops to do-while loops
  Compiling do-while loops to goto statements
  Compiling goto statements to assembly conditional jump instructions
Switch statements
Announcements

PA4 bomb lab
- PA4 bomb lab out and live. Due Tuesday, April 5.

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

Class session plan
- Today, 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
Comparisons and program control flow
  What is control flow?
  Condition codes
  Comparison and set instructions
Modifying control flow via conditional branch statements
  Jump instructions
  Conditional branch statements
Programming Assignment 4: Defusing a Binary Bomb
Modifying data flow via conditional move statements
  Conditional move statements
Loop statements
  Compiling for loops to while loops
  Compiling while loops to do-while loops
  Compiling do-while loops to goto statements
  Compiling goto statements to assembly conditional jump instructions
Switch statements
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
- **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
Perform source2 − source1, and sets the condition codes without setting any destination register.
**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>sete / setz</td>
<td>ZF</td>
<td>Equal / zero</td>
</tr>
<tr>
<td>Signed and unsigned</td>
<td>setne / setnz</td>
<td>~ZF</td>
<td>Not equal / not zero</td>
</tr>
<tr>
<td>Unsigned</td>
<td>setb / setnae</td>
<td>CF</td>
<td>Below</td>
</tr>
<tr>
<td>Unsigned</td>
<td>setbe / setna</td>
<td>CF</td>
<td>ZF</td>
</tr>
<tr>
<td>Unsigned</td>
<td>seta / setnbe</td>
<td>~CF &amp; ~ZF</td>
<td>Above</td>
</tr>
<tr>
<td>Unsigned</td>
<td>setnb / setae</td>
<td>~CF</td>
<td>Above or equal</td>
</tr>
<tr>
<td>Signed</td>
<td>sets</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>Signed</td>
<td>setns</td>
<td>~SF</td>
<td>Nonegative</td>
</tr>
<tr>
<td>Signed</td>
<td>setl / setnge</td>
<td>SF ^ OF</td>
<td>Less than</td>
</tr>
<tr>
<td>Signed</td>
<td>setle / setng</td>
<td>(SF ^ OF)</td>
<td>ZF</td>
</tr>
<tr>
<td>Signed</td>
<td>setg / setnle</td>
<td>~(SF ^ OF) &amp; ~ZF</td>
<td>Greater than</td>
</tr>
<tr>
<td>Signed</td>
<td>setge / setnle</td>
<td>~(SF ^ OF)</td>
<td>Greater than or equal</td>
</tr>
</tbody>
</table>

**Table: Set instructions**
Side review: De Morgan’s laws

- \( \neg A \land \neg B \iff \neg (A \lor B) \)
- \( (\sim A) \& (\sim B) \iff \sim (A\vert B) \)
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.

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

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.
Table of contents

Announcements

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

Modifying control flow via conditional branch statements
  Jump instructions
  Conditional branch statements

Programming Assignment 4: Defusing a Binary Bomb

Modifying data flow via conditional move statements
  Conditional move statements

Loop statements
  Compiling for loops to while loops
  Compiling while loops to do-while loops
  Compiling do-while loops to goto statements
  Compiling goto statements to assembly conditional jump instructions

Switch statements
Jump instructions

Jumping

- jX Instructions
  - Jump to different part of code depending on condition codes

<table>
<thead>
<tr>
<th>jX</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmp</td>
<td>1</td>
<td>Unconditional</td>
</tr>
<tr>
<td>je</td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>jne</td>
<td>~ZF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>js</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>jns</td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>jg</td>
<td>~(SF^OF) &amp; ~ZF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>jge</td>
<td>~(SF^OF)</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>jl</td>
<td>(SF^OF)</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>jle</td>
<td>(SF^OF)</td>
<td>Less or Equal (Signed)</td>
</tr>
<tr>
<td>ja</td>
<td>~CF &amp; ~ZF</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>jb</td>
<td>CF</td>
<td>Below (unsigned)</td>
</tr>
</tbody>
</table>
Branch statements

```c
1 unsigned long absdiff_ternary (  
2     unsigned long x, unsigned long y ){
3         return x<y ? y-x : x-y;
4 }
```

```c
1 unsigned long absdiff_if_else (  
2     unsigned long x, unsigned long y ){
3         if (x<y) return y-x;
4         else return x-y;
5 }
```

```c
1 unsigned long absdiff_goto (  
2     unsigned long x, unsigned long y ){
3         if (!(x<y)) goto Else;
4         return y-x;
5     Else:
6         return x-y;
7 }
```

All C functions above translate (-fno-if-conversion) to assembly at right.

absdiff_if_else:
```assembly
    cmpq %rsi, %rdi
    jnb .ELSE
    movq %rsi, %rax
    subq %rdi, %rax
    ret
.ELSE:
    movq %rdi, %rax
    subq %rsi, %rax
    ret
```

Explanation

- `cmpq %rsi, %rdi`: Calculates `%rdi - %rsi (x \&\& y)`, sets condition codes.
- `jnb .ELSE`: Sets program counter / instruction pointer in `%rip (.ELSE)` if CF flag not set indicating no unsigned overflow.
Table of contents

Announcements

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

Modifying control flow via conditional branch statements
  Jump instructions
  Conditional branch statements

Programming Assignment 4: Defusing a Binary Bomb

Modifying data flow via conditional move statements
  Conditional move statements

Loop statements
  Compiling for loops to while loops
  Compiling while loops to do-while loops
  Compiling do-while loops to goto statements
  Compiling goto statements to assembly conditional jump instructions

Switch statements
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.
- `stepe` or `si`: Step one instruction exactly. Steps into functions / subroutine calls.
Example phase_1 in example bomb from CS:APP website

00000000000400ee0 <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

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

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

Modifying control flow via conditional branch statements
  Jump instructions
  Conditional branch statements

Programming Assignment 4: Defusing a Binary Bomb

Modifying data flow via conditional move statements
  Conditional move statements

Loop statements
  Compiling for loops to while loops
  Compiling while loops to do-while loops
  Compiling do-while loops to goto statements
  Compiling goto statements to assembly conditional jump instructions

Switch statements
Deep CPU pipelines

Figure: Pipelined CPU stages. Image credit wikimedia
Table of contents

Announcements

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

Modifying control flow via conditional branch statements
  Jump instructions
  Conditional branch statements

Programming Assignment 4: Defusing a Binary Bomb

Modifying data flow via conditional move statements
  Conditional move statements

Loop statements
  Compiling for loops to while loops
  Compiling while loops to do-while loops
  Compiling do-while loops to goto statements
  Compiling goto statements to assembly conditional jump instructions

Switch statements
Compiling for loops to while loops

C loop statements such as for loops, while loops, and do-while loops do not exist in assembly. They are instead constructed from conditional jump statements.

```c
unsigned long count_bits_for (unsigned long number) {
    unsigned long tally = 0;
    for (int shift=0; shift<8*sizeof(unsigned long); // test
         shift++) // update
        tally += 0b1 & number>>shift;
    return tally;
}
```

```c
unsigned long count_bits_while (unsigned long number) {
    unsigned long tally = 0;
    int shift=0; // init
    while (shift<8*sizeof(unsigned long) // test
            shift++) // update
        tally += 0b1 & number>>shift;
    return tally;
}
```
Compiling while loops to do-while loops

```c
unsigned long count_bits_while (unsigned long number) {
  unsigned long tally = 0;
  int shift = 0; // init
  while (shift < 8 * sizeof(unsigned long) - test) {
    // body
    tally += 0b1 & number >> shift;
    shift++; // update
  }
  return tally;
}
```

```c
unsigned long count_bits_do_while (unsigned long number) {
  unsigned long tally = 0;
  int shift = 0; // init
  do {
    // body
    tally += 0b1 & number >> shift;
    shift++; // update
  } while (shift < 8 * sizeof(unsigned long) - test); // test
  return tally;
}
```

If initial iteration is guaranteed to run, then do one fewer test.
Compiling do-while loops to goto statements

```
unsigned long count_bits_do_while ( unsigned long number ) {
    unsigned long tally = 0;
    int shift=0; // init
    do {
        // body
        tally += 0b1 & number>>shift;
        shift++; // update
    } while (shift<8*sizeof(unsigned long)); // test
    return tally;
}
```

```
unsigned long count_bits_goto ( unsigned long number ) {
    unsigned long tally = 0;
    int shift=0; // init
    LOOP:
    // body
    tally += 0b1 & number>>shift;
    shift++; // update
    if (shift<8*sizeof(unsigned long)) { ← // test
        goto LOOP;
    }
    return tally;
}
```

Loops get compiled into goto statements which are readily translated to assembly.
Compiling goto statements to assembly conditional jump instructions

```c
1 unsigned long count_bits_goto(
2    unsigned long number
3 )
4 {
5    unsigned long tally = 0;
6    int shift=0; // init
7    LOOP:
8        // body
9        tally += 0b1 & number>>shift;
10       shift++; // update
11       if (shift<8*sizeof(unsigned long)) {
12            goto LOOP;
13       }
14    return tally;
15 }
```

All C loop statements so far translate to assembly at right.

```
count_bits_for:
    xorl %ecx, %ecx # int shift=0; // init
    xorl %eax, %eax # unsigned long tally = 0;
    .LOOP:
    movq %rdi, %rdx # number
    shrq %cl, %rdx # number>>shift
    incl %ecx # shift++; // update
    andl $1, %edx. # 0b1 & number>>shift
    addq %rdx, %rax # tally += 0b1 & number>>shift;
    cmpl $64, %ecx # shift<8*sizeof(unsigned long)
    jne .LOOP # goto LOOP;
    ret # return tally;
```
Table of contents

Announcements

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

Modifying control flow via conditional branch statements
  Jump instructions
  Conditional branch statements

Programming Assignment 4: Defusing a Binary Bomb

Modifying data flow via conditional move statements
  Conditional move statements

Loop statements
  Compiling for loops to while loops
  Compiling while loops to do-while loops
  Compiling do-while loops to goto statements
  Compiling goto statements to assembly conditional jump instructions

Switch statements