Assembly: Control flow and loops.

Yipeng Huang

Rutgers University

March 9, 2021
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

Modifying data flow via conditional move statements
  Conditional move statements
Looking ahead

Class plan

1. PA2 grades released last night. Contact TAs Prince and Azita for questions and concerns.
2. Code review session for PA2 ongoing this week. TAs will take attendance to assign participation points.
3. Provide mid-semester course feedback at: http://sirs.ctaar.rutgers.edu/blue
5. Thursday, 3/11: Assembly loops and function calls.
6. Reading assignment for next two weeks: CS:APP Chapter 3.
7. Programming Assignment 3 on bits, bytes, integers, floats out. Due Monday March 22.
Programming Assignment 3: binToFloat

General reminders

▶ PA3 is structured in terms of difficulty similarly to PA1 and PA2.
▶ The assignment rewards you for starting early.
▶ Use Piazza; We rely on it to gauge what needs further explanation.
▶ Later parts (parts 4 and 5) are more open-ended.

binToFloat

▶ How to read in the characters.
▶ How to accumulate the representation in the binary number.
▶ You are not allowed to use pointer casting to directly convert binary representation to float.
▶ You may find the %e or %E printf format specifiers useful at some point in PA3. https://www.cplusplus.com/reference/cstdio/printf/
▶ How to shift and mask for the sign bit.
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

Modifying data flow via conditional move statements
  Conditional move 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**
  - 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

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

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

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;
}
```

```c
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

- \( \neg A \land \neg B \iff \neg (A \lor B) \)
- \((\sim A) \land (\sim B) \iff \sim (A \lor B)\)
**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</code> / <code>setz</code></td>
<td><code>ZF</code></td>
<td>Equal / zero</td>
</tr>
<tr>
<td>Signed and unsigned</td>
<td><code>setne</code> / <code>setnz</code></td>
<td><code>∼ ZF</code></td>
<td>Not equal / not zero</td>
</tr>
<tr>
<td>Unsigned</td>
<td><code>setb</code> / <code>setnae</code></td>
<td><code>CF</code></td>
<td>Below</td>
</tr>
<tr>
<td>Unsigned</td>
<td><code>setbe</code> / <code>setna</code></td>
<td>`CF</td>
<td>ZF`</td>
</tr>
<tr>
<td>Unsigned</td>
<td><code>seta</code> / <code>setnbe</code></td>
<td><code>∼ CF &amp; ∼ ZF</code></td>
<td>Above</td>
</tr>
<tr>
<td>Unsigned</td>
<td><code>setnb</code> / <code>setae</code></td>
<td><code>∼ CF</code></td>
<td>Above or equal</td>
</tr>
<tr>
<td>Signed</td>
<td><code>sets</code></td>
<td><code>SF</code></td>
<td>Negative</td>
</tr>
<tr>
<td>Signed</td>
<td><code>setns</code></td>
<td><code>∼ SF</code></td>
<td>Nonegative</td>
</tr>
<tr>
<td>Signed</td>
<td><code>setl</code> / <code>setnge</code></td>
<td><code>SF ^ OF</code></td>
<td>Less than</td>
</tr>
<tr>
<td>Signed</td>
<td><code>setle</code> / <code>setng</code></td>
<td>`(SF ^ OF</td>
<td>ZF`</td>
</tr>
<tr>
<td>Signed</td>
<td><code>setg</code> / <code>setnle</code></td>
<td><code>(SF ^ OF) &amp; ∼ ZF</code></td>
<td>Greater than</td>
</tr>
<tr>
<td>Signed</td>
<td><code>setge</code> / <code>setnl</code></td>
<td><code>(SF ^ OF)</code></td>
<td>Greater than or equal</td>
</tr>
</tbody>
</table>

**Table: Set instructions**
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

Modifying data flow via conditional move statements
  Conditional move 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>

Figure: Jump instructions. Image credit CS:APP
Branch statements

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

```c
unsigned long absdiff_if_else (unsigned long x, unsigned long y)
{
    if (x < y) return y - x;
    else return x - y;
}
```

```c
unsigned long absdiff_goto (unsigned long x, unsigned long y)
{
    if (!(x < y)) goto Else;
    return y - x;

Else:
    return x - y;
}
```

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

```assembly
absdiff_if_else:
    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

Modifying data flow via conditional move statements
  Conditional move statements
Deep CPU pipelines

Figure: Pipelined CPU stages. Image credit wikimedia