Assembly: Control flow and loops.

Yipeng Huang

Rutgers University

March 9, 2021

Table of contents

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`
4. Today, Tuesday, 3/9: Assembly control flow and loops.
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

What is control flow?

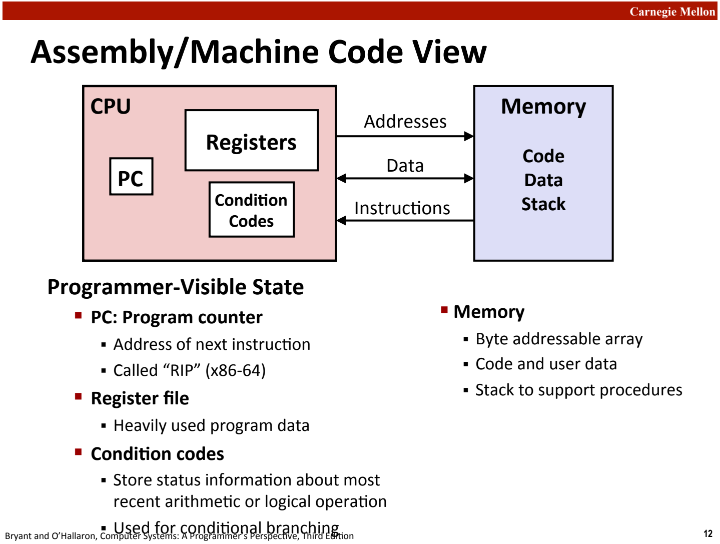
Control flow is:

- 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



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

```
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;
}
```

x = 127
y = 128
x-y = 127-128 = -1 = overflow
CF = 1
returns 1

```
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`	$\sim$ `ZF`	Not equal / not zero
Unsigned	`setb` / `setnae`	`CF`	Below
Unsigned	`setbe` / `setna`	`CF`|`ZF`	Below or equal
Unsigned	`seta` / `setnbe`	$\sim$ `CF`& $\sim$ `ZF`	Above
Unsigned	`setnb` / `setae`	$\sim$ `CF`	Above or equal
Signed	`sets`	`SF`	Negative
Signed	`setns`	$\sim$ `SF`	Nonegative
Signed	`setl` / `setnge`	`SF ^ OF`	Less than
Signed	`setle` / `setng`	(`SF ^ OF`)|`ZF`	Less than or equal
Signed	`setg` / `setnle`	$\sim$ (`SF ^ OF`)& $\sim$ `ZF`	Greater than
Signed	`setge` / `setnl`	$\sim$ (`SF ^ OF`)	Greater than or equal

Table: Set instructions

Table of contents

Jump instructions

Jumping

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

jX	Condition	Description
`jmp`	`1`	Unconditional
`je`	`ZF`	Equal / Zero
`jne`	`~ZF`	Not Equal / Not Zero
`js`	`SF`	Negative
`jns`	`~SF`	Nonnegative
`jg`	`~(SF^OF)&~ZF`	Greater (Signed)
`jge`	`~(SF^OF)`	Greater or Equal (Signed)
`jl`	`(SF^OF)`	Less (Signed)
`jle`	`(SF^OF)|ZF`	Less or Equal (Signed)
`ja`	`~CF&~ZF`	Above (unsigned)
`jb`	`CF`	Below (unsigned)

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.

```
absdiff_if_else:
absdiff_goto:
    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

Deep CPU pipelines

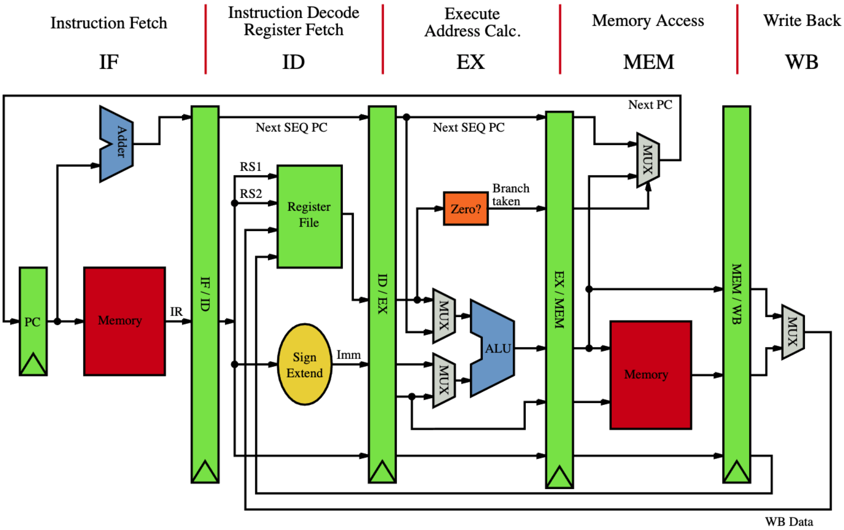


Figure: Pipelined CPU stages. Image credit wikimedia