Control Flow

Condition Codes (implicit setting)

- single bit registers
  - CF (carry flag for unsigned)
  - ZF (zero flag)
  - SF (sign flag (for signed))
  - OF (overflow flag (for signed))
- implicitly set (think of it as side effect) by arithmetic operations
  - Example: **addq** *Src, Dest* <-> t = a+b
  - CF set if carry out form most significant bit (unsigned overflow)
  - ZF set if t == 0
  - SF set if t < 0 (as signed)
  - OF set if 2's-complement (signed) overflow
    - `(a > 0 && b > 0 && t < 0) || (a < 0 && b < 0 && t >= 0)`
- not set by **leaq** instruction

Condition Codes (explicit setting)

- does not overwrite destination
- just writes condition codes
- explicit setting by compare instruction
  - **cmpq** *src2, src1*
  - **cmpq** b, a like computing a-b without setting destination
  - CF set if carry out from most significant bit (used for unsigned comparisons)
  - ZF set if a == b
  - SF set if (a-b) < 0
- explicity setting by test instruction
  - testq src2, src1
    - testq b, a like computing a & b without setting destination
  - sets conition codes based ...

Reading Condition Codes

- setX instructions
  - set low-order byte of destination to 0 or 1 based on combinations of condition codes
  - does not alter remaining 7 bytes

SetX	Condition	Description
sete	ZF	Equal / Zero
setne	~ZF	Not Equal / Not Zero
sets	SF	Negative
setns	~SF	Nonnegative
setg	~(SF^OF)&~ZF	Greater (Signed)
setge	~(SF^OF)	Greater or Equal (Signed)
setl	(SF^OF)	Less (Signed)
setle	(SF^OF)|ZF	Less or Equal (Signed)
seta	~CF&~ZF	Above (unsigned)
setb	CF	Below (unsigned)

- setX instructions
  - set single byte based on combination of condition codes
- one of addressable byte registers
  - does not alter remaining bytes
  - typically use **movzbl** to finish job
    - 32-bit instructions also set upper 32 bits to 0

Register	Use(s)
%rdi	Argument **x**
%rsi	Argument **y**
%rax	Return value

```
1 int gt (long x, long y) {
2     return x > y;
3 }
```

```
1 cmpq    %rsi, %rdi  # Compare x:y
2 setq    %al         # Set when >
3 movzbl  %al, %eax   # Zero rest of %rax
4 ret
```

Conditional Branches

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)

Conditional Branch Example (Old Style)

- basic if-else branching in assembly uses a jump
  - the greater than (~ <=) requires the use of `jle`

Register	Use(s)
%rdi	Argument **x**
%rsi	Argument **y**
%rax	Return value

```
1 long absdiff (long x, long) {
2     long result;
3     if (x > y)
4         result = x-y;
5     else                    // jump to L4 in assembly
6         result = y-x;
7     return result;
8 }
```

```
1 absdiff:
2     cmpq    %rsi, %rdi  # x:y
3     jle .L4             # jump to .L4 if less than or equal to
4     movq    %rdi, %rax  # move x argument into %rax
5     subq    %rsi, %rax  # subtract y from x
6     ret
7 .L4:                    # x <= y
8     movq    %rsi, %rax  # move y argument into %rax
9     subq    %rdi, %rax  # subtract x from y
```

```
10      ret
```

Using Conditional Moves

Conditional Move Instructions

- instruction supports:
    - `if (test) dest <- src`
- GCC tries to use them
    - but only when known to be safe

Why?

- branches are very disruptive to instruction flow through pipelines
- conditional moves do not require control transfer

```
1 // C Code
2 val = Test
3     ? Then_Expr
4     : Else_Expr;
```

Conditional Move Example

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rax	Return value

```
1 long absdiff (long x, long y) {
2     long result;
3     if (x > y)
4         result = x-y;
5     else
6         result = y-x;
7     return result;
8 }
```

```
1 absdiff:
2     movq    %rdi, %rax  # x
3     subq    %rsi, %rax  # result = x-y
4     movq    %rsi, %rdx
5     subq    %rdi, %rdx  # eval = y-x
6     cmpq    %rsi, %rdi  # x:y
7     cmovle  %rdx, %rax  # if <=, result = eval
8     ret
9
10 # Do both x-y and y-x and compare the results afterwards.
```

Bad Cases for Conditional Move

- **expensive computations**
  - `val = Test(x) ? Hard1(x) : Hard2(x);`
  - both values get computed
  - only makes sense when computations are very simple
- **risky computations**
  - `val = p ? *p : 0;`
  - both values get computed
  - may have undesirable effects
- **computations with side effects**
  - `val = x > 0 ? x*=7 : x+=3;`
  - both values get computed
  - must be side-effect free

Loops

“do-while” Loop Example

- count number of 1’s in argument (“popcount”)
- use conditional branc to either continue looping or to exit loop

```
long pcount_do (unsigned long x) {
    long result = 0;
    do {
        result += x & 0x1;
        x >>= 1;
    } while (x);
    return result;
}
```

```
    movl    $0, %eax    ; result = 0
.L2:                    ; loop:
    movq    %rdi, %rdx  ; rdi to rdx
    andl    $1, %edx    ; t = x & 0x1
    addq    %rdx, %rax  ; result += t
    shrq    %rdi        ; x >>= 1
    jne     .L2         ; if (x) goto loop, checks 0F flag from shift
                        ; as long as rdi is not 0
    ret
```

“for” Loop Example

```
long pcount_for (unsigned long x) {
```

```
    size_t i;
    long result = 0;
    for (i = 0; i < WSIZE; i++) {
        unsigned bit = (x >> i) & 0x1;
        result += bit;
    }
    return result;
}
```

```
long pcount_for_goto_dw
  (unsigned long x) {
  size_t i;
  long result = 0;
  i = 0;                          Init
  if (!(i < WSIZE))               ! Test
    goto done;
 loop:
  {
    unsigned bit =
       (x >> i) & 0x1;            Body
    result += bit;
  }
  i++;                            Update
  if (i < WSIZE)                  Test
    goto loop;
 done:
  return result;
}
```

Switch Statement Example

Register	Use(s)
%rdi	Argument x
%rsi	Argument y
%rdx	Argument z
%rax	Return value

Jump table

```
.section    .rodata
  .align 8
.L4:
  .quad     .L8   # x = 0
  .quad     .L3   # x = 1
  .quad     .L5   # x = 2
  .quad     .L9   # x = 3
  .quad     .L8   # x = 4
  .quad     .L7   # x = 5
  .quad     .L7   # x = 6
```

```
1 long switch_eg (long x, long y, long z) {
2     long 2 = 1;
3     switch (x) {
4         ...
5     }
6     return w;
7 }
```

```
1 switch_eg:
2     movq    %rdx, %rcx
3     cmpq    $6, %rdi    #x:6
4     ja      .L8
5     jmp     *.L4(,%rdi,8)
```