Condition Codes
Processor State (x86-64, first look)

Information about currently executing program
- Temporary data ( %rax, ... )
- Location of runtime stack ( %rsp )
- Location of current code control point ( %rip, ... )
- Status of recent tests ( CF, ZF, SF, OF )

<table>
<thead>
<tr>
<th>Registers</th>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>%r8</td>
</tr>
<tr>
<td>%rbx</td>
<td>%r9</td>
</tr>
<tr>
<td>%rcx</td>
<td>%r10</td>
</tr>
<tr>
<td>%rdx</td>
<td>%r11</td>
</tr>
<tr>
<td>%rsi</td>
<td>%r12</td>
</tr>
<tr>
<td>%rdi</td>
<td>%r13</td>
</tr>
<tr>
<td>%rsp</td>
<td>%r14</td>
</tr>
<tr>
<td>%rbp</td>
<td>%r15</td>
</tr>
</tbody>
</table>

Instruction pointer: %rip
Condition codes: CF, ZF, SF, OF

Current top of the stack
Condition Codes (Implicit Setting)

- Single bit registers
  - CF  Carry Flag (for unsigned)
  - SF  Sign Flag (for signed)
  - ZF  Zero Flag
  - 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 from most significant bit (unsigned overflow)
- ZF set if $t = 0$
- SF set if $t < 0$ (as signed)
- OF set if two’s-complement (signed) overflow
  
  \[(a>0 && b>0 && t<0) \lor (a<0 && b<0 && t>=0)\]

- Not set by `leaq` instruction
Condition Codes (Explicit Setting - `cmpq`) 

- 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 (as signed)
- OF set    if two’s-complement (signed) overflow
- \((a>0 && b<0 && (a-b)<0) || (a<0 && b>0 && (a-b)>0)\)
Condition Codes (Explicit Setting - `testq`)

- Explicit Setting by Test instruction

\[ \text{testq } \text{Src2, Src1} \]

- `testq b, a` like computing a\&b without setting destination
- Sets condition codes based on value of Src1 & Src2
- Useful to have one of the operands be a mask

- ZF set when a\&b == 0
- SF set when a\&b < 0
Reading Condition Codes

- **SetX family of instructions**
  - Set low-order byte of destination to 0 or 1 based on combinations of condition codes
  - Does not alter remaining 7 bytes

<table>
<thead>
<tr>
<th>SetX</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sete</td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>setne</td>
<td>~ZF</td>
<td>Not Equal / Not Zero</td>
</tr>
<tr>
<td>sets</td>
<td>SF</td>
<td>Negative</td>
</tr>
<tr>
<td>setns</td>
<td>~SF</td>
<td>Nonnegative</td>
</tr>
<tr>
<td>setg</td>
<td>~(SF^OF) &amp; ~ZF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>setge</td>
<td>~(SF^OF)</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>setl</td>
<td>(SF^OF)</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>setle</td>
<td>(SF^OF)</td>
<td>Less or Equal (Signed)</td>
</tr>
<tr>
<td>seta</td>
<td>~CF &amp; ~ZF</td>
<td>Above (unsigned)</td>
</tr>
<tr>
<td>setb</td>
<td>CF</td>
<td>Below (unsigned)</td>
</tr>
</tbody>
</table>
### x86-64 Integer Registers

<table>
<thead>
<tr>
<th>Register</th>
<th>Low-Order Byte</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rax</td>
<td>%al</td>
</tr>
<tr>
<td>%rbx</td>
<td>%bl</td>
</tr>
<tr>
<td>%rcx</td>
<td>%cl</td>
</tr>
<tr>
<td>%rdx</td>
<td>%dl</td>
</tr>
<tr>
<td>%rsi</td>
<td>%sil</td>
</tr>
<tr>
<td>%rdi</td>
<td>%dil</td>
</tr>
<tr>
<td>%rsp</td>
<td>%spl</td>
</tr>
<tr>
<td>%rbp</td>
<td>%bpl</td>
</tr>
<tr>
<td>%r8</td>
<td>%r8b</td>
</tr>
<tr>
<td>%r9</td>
<td>%r9b</td>
</tr>
<tr>
<td>%r10</td>
<td>%r10b</td>
</tr>
<tr>
<td>%r11</td>
<td>%r11b</td>
</tr>
<tr>
<td>%r12</td>
<td>%r12b</td>
</tr>
<tr>
<td>%r13</td>
<td>%r13b</td>
</tr>
<tr>
<td>%r14</td>
<td>%r14b</td>
</tr>
<tr>
<td>%r15</td>
<td>%r15b</td>
</tr>
</tbody>
</table>

We can reference low-order byte.
Reading Condition Codes

- **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)

```c
int gt (long x, long y)
{
    return x > y;
}
```

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument y</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>

```assembly
cmpq   %rsi, %rdi   # Compare x:y
setg   %al          # Set when >
movzbl %al, %eax    # Zero rest of %rax
ret
```
Conditional Branches
Jumping (in the code)

**jX** family of 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>ZF</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>
Re-Writing Code with `goto` Statements

- C allows `goto` statement
- Jump to position designated by label

```c
long absdiff (long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}

long absdiff_j (long x, long y)
{
    long result;
    int ntest = x <= y;
    if (ntest) goto Else;
    result = x-y;
    goto Done;
Else:
    result = y-x;
Done:
    return result;
}
```

Why do that?
- Because the "goto" code is closer to the assembly instructions.
General Conditional Expression Translation

C code:

```c
val = Test ? Then Expr : Else Expr;
```

for example: `val = x>y ? x-y : y-x;`

```c
ntest = !Test;
if (ntest) goto Else;
val = Then_Expr;
goto Done;
Else:
  val = Else_Expr;
Done:
  ...
```

- Create separate code regions for then & else expressions
- Execute appropriate one
Using Conditional Moves

- Conditional Move Instructions
  - Instruction supports:
    
    ```
    if (Test) Dest ⇐ Src
    ```
  - Supported in post-1995 x86 processors
  - 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

```
C Code:
val = Test
? Then_Expr : Else_Expr;
```

```
goto_version:
result = Then_Expr;
eval = Else_Expr;
nt = !Test;
if (nt) result = eval;
return result;
```
Conditional Move Example

```c
long absdiff
 (long x, long y)
{
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}
```

### absdiff:
- `movq %rdi, %rax # x`
- **subq** `%rsi, %rax # result = x-y`
- `movq %rsi, %rdx`
- **subq** `%rdi, %rdx # eval = y-x`
- `cmpq %rsi, %rdi # x:y`
- `cmovle %rdx, %rax # if <=, result = eval`
- `ret`

### Register Use(s)

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rsi</td>
<td>Argument y</td>
</tr>
<tr>
<td>%rax</td>
<td>Return value</td>
</tr>
</tbody>
</table>
Bad Cases of Conditional Move

- **expensive computations:**
  - both values get calculated
  - only makes sense when computations are very simple

- **risky computations**
  - both values get calculated
  - may have undesirable side effects (above it is dereferencing a pointer that may be 0)

- **computations with side effects**
  - both values get calculated
  - must be side-effect free (unlike the example above)
Loops
do...while... loop example

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

long pcount_goto (unsigned long x)
{
    long result = 0;
    loop:
    result += x & 0x1;
    x >>= 1;
    if(x) goto loop;
    return result;
}

- Count number of 1’s in argument x
- Use conditional branch to either continue looping or to exit loop
do...while... loop compilation

long pcount_goto (unsigned long x) 
{ 
    long result = 0;
    loop: 
        result += x & 0x1;
        x >>= 1;
        if(x) goto loop;
    return result;
}

movl    $0, %eax  # result = 0
.L2: 
    movq    %rdi, %rdx  # loop:
    movl    $1, %edx  # t = x & 0x1
    addq    %rdx, %rax  # result += t
    shrq    %rdi  # x >>= 1
    jne     .L2  # if (x) goto loop
rep; ret

<table>
<thead>
<tr>
<th>Register</th>
<th>Use(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>%rdi</td>
<td>Argument x</td>
</tr>
<tr>
<td>%rax</td>
<td>result</td>
</tr>
</tbody>
</table>
General do...while... Translation

```
do
  Body
  while (Test);
```

```
loop:
  Body
  if (Test)
    goto loop
```
General **while** Loop Translation (ver. 1)

- “Jump-to-middle” translation
- Used with -Og option to gcc
General **while** Loop Translation (ver. 1)

- Compare to do-while version of function
- Initial goto starts loop at test
General **while** Loop Translation (ver. 2)

- “Do-while” conversion
- Used with -O1

```
while (Test)
  Body
```

convert to do.. while.. first

```
if (!Test) 
goto done;
do 
  Body
while (Test);
done:
```

```
  if (!Test) 
goto done;
loop:
  Body
  if (Test) 
goto loop;
done:
```
General **while** Loop Translation (ver. 1)

- Compare to do-while version of function
- Initial conditional guards entrance to loop

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

```c
long pcount_goto_dw (unsigned long x) {
    long result = 0;
    if (!x) goto done;
    loop:
    result += x & 0x1;
    x >>= 1;
    if(x) goto loop;
    done:
    return result;
}
```
**for Loop Form**

**General form:**

```
for (Init; Test; Update )
  Body
```

```c
#define WSIZE 8*sizeof(int)
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;
}
```
for Loop ⇒ while loop

\[
\text{for (Init; Test; Update )} \\
\text{Body}
\]

\[
\text{Init;} \\
\text{while (Test )} \{ \\
\text{Body} \\
\text{Update} \\
\}
\]
for Loop ⇒ while loop

```c
long pcount_for_while (unsigned long x) {
    size_t i;
    long result = 0;
    i = 0;
    while (i < WSIZE) {
        unsigned bit =
            (x >> i) & 0x1;
        result += bit;
        i++;
    }
    return result;
}
```
for Loop ⇒ while loop

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;
    if (!(i < WSIZE))
        goto done;
    loop:
    {
        unsigned bit =
        (x >> i) & 0x1;
        result += bit;
    }
    i++;
    if (i < WSIZE)
        goto loop;
    done:
    return result;
}

Initial test can be optimized away