Control & Condition Codes
### Processor State (x86-64, Partial)

- Information about currently executing program...
  - temporary data (%rax, …)
  - location of runtime stack (%rsp)
  - location of current code point (%rip)
  - status of recent tests (CF, ZF, SF, OF)

```plaintext
Current stack ‘top’  Current instruction
```

### Registers

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

### Instruction

```
CF ZF SF OF
```

### Condition
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 a side effect) by arithmetic operations
  - Example: `addq src, dest ↔ b = 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
      \[(a > 0 && b > 0 && t < 0) || (a < 0 && b < 0 && t > 0)\]

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

- 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
      
      \[(a > 0 && b < 0 && (a-b) < 0) || (a < 0 && b > 0 && (a-b) > 0)\]

- Only purpose of this instruction is to set condition codes!
- There are other instructions like this.
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

<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>ZF</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

- Can reference low-order byte.
Reading Condition Codes *Con't*

- **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

```
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>

```
cmpq %rsi, %rdi          # Compare x and y
setg %al                 # Set %al 'on' when x > y
movzbl %al, %rax         # Copy and zero rest of %rax
ret                      |
```
Conditional Branches
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>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>
### Conditional Branching by Jumping

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

**absdiff:**

```assembly
    cmpq    %rsi, %rdi  # y, x
    jle     .L4
    movq    %rdi, %rax
    subq    %rsi, %rax
    ret
.L4:       # x <= y
    movq    %rsi, %rax
    subq    %rdi, %rax
    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>

- **Note:** must use `-fno-if-conversion` argument to gcc, otherwise assembly will not use jumps in this program, we’ll see why in a moment.
Rewriting C 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;
}
```

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

- `goto` form resembles assembly instructions using jumps
Rewriting C with `goto` Statements con’t

- C code
  
  ```c
  val = test ? then_expr : else_expr;
  ```

- Example
  
  ```c
  val = x > y ? x - y : y - x;
  ```

- Goto version
  
  ```c
  if (!test) goto Else;
  val = then_expr;
  goto Done:
  Else:
    val = else_Expr;
  Done:
    return val;
  ```

- Create separate code regions for then & else expressions
- Execute appropriate one
- This is how we can think about ‘jumping’ in assembly
Alternate Approach: 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

C Code

```c
val = test ? then_exp : else_exp;
```

Goto Version

```c
result = then_expr;
eval = else_expr;
neg_test = !test;
if (neg_test) result = eval;
return result;
```
long absdiff(long x, long y){
    long result;
    if (x > y)
        result = x-y;
    else
        result = y-x;
    return result;
}

Register | Use(s)
--- | ---
%rdi | Argument x
%rsi | Argument y
%rax | Return value
%rdx | Temp variable

absdiff:
    movq   %rdi, %rax  # x
    subq   %rsi, %rax  # if-val = x-y
    movq   %rsi, %rdx
    subq   %rdi, %rdx  # else-val = y-x
    cmpq   %rsi, %rdi  # %rsi = y, %rdi = x
    cmovle %rdx, %rax  # if y <= x, result = else-val
    ret
Bad Cases for Conditional Move

- **Expensive computations**
  ```c
  val = Test(x) ? Hard1(x) : Hard2(x);
  ```
  - Both values get computed
  - Only makes sense when computations are very simple

- **Risky computations**
  ```c
  val = p == 0 ? 0 : 5 / p;
  ```
  - Both values get computed
  - May have undesirable effects

- **Computations with side effects**
  ```c
  val = x > 0 ? x *= 7 : x += 3;
  ```
  - Both values get computed
  - Must be side-effect free
Loops
General “Do-While” Translation

C Code

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

Goto Version

```c
loop:
    Body
    if (Test)
        goto loop
```

Body

```c
{
    statement_1;
    statement_2;
    ...
    statement_n;
}
```
### “Do-While” Loop Example

**C Code**

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

**Goto Version**

```c
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 (‘popcount’)
- Use conditional jump to either continue looping or to exit loop
“Do-While” Loop Compilation

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

**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>%rax</td>
<td>result</td>
</tr>
</tbody>
</table>

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

- Note: some processors' branch predictors behave badly when a branch's target or fall-through is a **ret** instruction, and adding the **rep;** prefix avoids this.
General “While” Translation

- “Jump-to-middle” translation
- Used with `gcc -Og` (our setting)

While Version

```c
while (Test)  
  Body
```

Goto Version

```c
goto test;  
loop:  
  Body
  test:  
    if (Test)  
      goto loop;
  done:
```
While Loop Example

- Compare to do-while version of function
- Initial goto starts loop at test
For Loop: Derived From While

For Version

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

While Version

```
Init;
while (Test) {
  Body
  Update;
}
```
For-While Conversion

```c
#define WSIZE 8*sizeof(int)
long pcount_for(unsigned long x)
{
    int i;
    long result = 0;
    for (i = 0; i < WSIZE; i++)
    {
        unsigned bit = (x >> i) & 0x1;
        result += bit;
    }
    return result;
}

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 Do-While Conversion

Initial test may be optimized away if compiler knows its safe