Accessing the Condition Codes

3 common ways to use condition codes:

- **SET**
  - Set a single byte to 0 or 1 depending on some combination of the condition codes

- **JMP**
  - Conditionally jump to some other part of the program

- **CMOV**
  - Conditionally transfer data
Set instructions

- Sets a single byte to 0 or 1 based on combinations of condition codes
- Each set instruction has a designated destination:
  - Byte register
    - One of 8 addressable byte registers embedded within first 4 integer registers
    - Does not alter remaining 3 bytes
    - Typically use movzbl to finish the job
  - Single-byte memory location
## SET instruction options

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Condition</th>
<th>Synonym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sete</td>
<td>D ← ZF</td>
<td>setz</td>
<td>equal / zero</td>
</tr>
<tr>
<td>setne</td>
<td>D ← ~ZF</td>
<td>setnz</td>
<td>not equal / not zero</td>
</tr>
<tr>
<td>sets</td>
<td>D ← SF</td>
<td></td>
<td>negative</td>
</tr>
<tr>
<td>setns</td>
<td>D ← ~SF</td>
<td></td>
<td>nonnegative</td>
</tr>
<tr>
<td>setg</td>
<td>D ← ~(SF ^ OF) &amp; ~ZF</td>
<td>setnle</td>
<td>greater (signed &gt;)</td>
</tr>
<tr>
<td>setge</td>
<td>D ← ~(SF ^ OF)</td>
<td>setnl</td>
<td>greater or equal (signed &gt;=)</td>
</tr>
<tr>
<td>setl</td>
<td>D ← SF ^ OF</td>
<td>setnle</td>
<td>less (signed &lt;)</td>
</tr>
<tr>
<td>settle</td>
<td>D ← (SF ^ OF)</td>
<td>setng</td>
<td>less or equal (signed &lt;=)</td>
</tr>
<tr>
<td>seta</td>
<td>D ← ~CF &amp; ~ZF</td>
<td>setnbe</td>
<td>above (unsigned &gt;)</td>
</tr>
<tr>
<td>setb</td>
<td>D ← CF</td>
<td>setnae</td>
<td>below (unsigned &lt;)</td>
</tr>
</tbody>
</table>

Multiple possible names for the instructions called synonyms. Compilers and disassemblers make arbitrary choices of which names to use. Note CF only on unsigned options.
Set instruction examples

// is a < b?
   // a = %edx, b = %eax
   cmpl %eax, %edx  // a-b i.e. %edx - %eax
   // flags set by cmpl
   setl %al        // D ← SF ^ OF
   movzbl %al, %eax // clear high order 3 bytes
   // if %al has a 1 in it, then the answer is yes
   // if %al has a 0 in it, then the answer is no

FLAGS:
If a = b then ZF = 1 → a-b=0
If a < b then SF = 1 → a-b<0 (#2)
If a > b then SF = 0 → a-b>0
If a<0, b>0, t>0 then OF=1 (#1)
If a>0, b<0, t<0 then OF=1
If unsigned... CF (not interested)

SF ^ OF → D
0 0  =  0
0 1  =  1  (see #1 below)
1 0  =  1  (see #2 below)
1 1  =  0

So, a < b when D = 1
#1 a is neg, b is pos, t is pos
#2 a-b<0 means a<b

notice cmpL and setL are NOT the same thing

// another example
movl 12(%ebp), %eax  // eax = y
   cmpl %eax, 8(%ebp) // compare x:y (x-y)
   setg %al          // al = x > y
   movzbl %al, %eax  // zero rest of eax
## Jump instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Condition</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>jmp</td>
<td>1</td>
<td>unconditional</td>
</tr>
<tr>
<td>je label</td>
<td>ZF</td>
<td>equal</td>
</tr>
<tr>
<td>jne label</td>
<td>~ZF</td>
<td>not equal</td>
</tr>
<tr>
<td>js label</td>
<td>SF</td>
<td>negative</td>
</tr>
<tr>
<td>jns label</td>
<td>~SF</td>
<td>nonnegative</td>
</tr>
<tr>
<td>jg label</td>
<td>~(SF ^ OF) &amp; ~ZF</td>
<td>greater (signed)</td>
</tr>
<tr>
<td>jge label</td>
<td>~(SF ^ OF)</td>
<td>greater or equal (signed)</td>
</tr>
<tr>
<td>jl label</td>
<td>SF ^ OF</td>
<td>less (signed)</td>
</tr>
<tr>
<td>jle label</td>
<td>(SF ^ OF)</td>
<td>less or equal (signed)</td>
</tr>
<tr>
<td>ja label</td>
<td>~CF &amp; ~ZF</td>
<td>above (unsigned)</td>
</tr>
<tr>
<td>jb label</td>
<td>CF</td>
<td>below (unsigned)</td>
</tr>
</tbody>
</table>

The test and cmp instructions are combined with the **conditional and unconditional jmp instructions** to implement most relational and logical expressions and all control structures.

Set allows us to know what the condition evaluates to if something other than jmp to be done.

There are synonyms for jump instructions as well.
Example overview

if ( a == b ) x = 1;

cmpl a, b // (b-a) == 0
jne skip // not equal, so skip
movl $1, x // since a == b, x = 1
skip:
nop // no operation...???

if ( a > b ) x = 1;

cmpl b, a // (a-b) > 0
jle skip // skip if a <= b
movl $1, x
skip:
cmpl a,b
jge skip

// Counts the number of bits set to 1
int count = 0;
int loop = 32;
do {
    if ( x & 1 ) count++;
    x >>= 1;
    loop--;
} while ( loop != 0 )

movl $0, count
movl $32, loop
.L2:
movl x, %eax
andl $1, %eax
testl %eax, %eax
je .L5
    incl count
.L5:
sarlx
decl loop
cmpl $0, loop
jne .L2
Conditional branch example

```c
int max(int x, int y)
{
    if (x > y)
        return x;
    else
        return y;
}
```

```c
int goto_max(int x, int y)
{
    int rval = y;
    int ok = (x <= y);
    if (ok)
        goto done;
    rval = x;
    done:
    return rval;
}
```

C allows “goto” as means of transferring control
Closer to machine-level programming style
Generally considered bad coding style

```asm
movl 8(%ebp),%edx   # edx = x
movl 12(%ebp),%eax  # eax = y
cmpl %eax,%edx      # x : y
jle L9              # if <= goto L9
movl %edx,%eax      # eax = x  }  Skipped when x <= y
L9:                   # Done:
```
General “do while” translation

**C Code**

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

**Goto Version**

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

*Body* can be any C statement
Typically compound statement:

```c
{
    Statement_1;
    Statement_2;
    ...
    Statement_n;
}
```

**C Code**

```c
int fact_do (int x)
{
    int result = 1;
    do 
    {
        result *= x;
        x = x-1;
    } while (x > 1);
    return result;
}
```

**Goto Version**

```c
int fact_goto(int x)
{
    int result = 1;
    loop:
    result *= x;
    x = x-1;
    if (x > 1)
      goto loop;
    return result;
}
```

Reminder: “Test” is expression return an integer of 1 when true and 0 when false.

Use backward branch to continue looping

Only take branch when “while” condition holds
**“Do While” loop compilation**

**Goto Version**

```c
int fact goto (int x)
{
    int result = 1;
    loop:
        result *= x;
        x = x-1;
        if (x > 1)
            goto loop;
    return result;
}
```

**Assembly**

```assembly
fact goto:
    pushl %ebp
    movl %esp,%ebp
    movl $1,%eax
    movl 8(%ebp),%edx
    # Setup
    # edx = x
    L11:
    imull %edx,%eax
    # result *= x
    decl %edx
    # x--
    cmpl $1,%edx
    jg L11
    movl %ebp,%esp
    popl %ebp
    ret
    # Compare x : 1
    # if > goto loop
    # Finish
```

**Registers**

- `%edx` - `x`
- `%eax` - `result`
- `%ebp` - Base Pointer
- `%esp` - Stack Pointer
"While" loop translation

Is this code equivalent to the do-while version? Must jump out of loop if test fails

Uses same inner loop as do-while version; guards loop entry with extra test
"For" loop example

```c
// compute x raised to the // nonnegative power p int ipwr_for(int x, unsigned p) {
    int result;
    for (result = 1; p != 0; p = p>>1) {
        if (p & 0x1)
            result *= x;
        x = x * x;
    }
    return result;
}
```

Example walkthrough

x=2, p=4

cmov (conditional move) only transfers the data if the condition is true