Controlling Program Flow

- **Conditionals** (If-statement)
- **Loops**
  (while, do-while, for-loops)
- **Switch Statements**
- **New Instructions**
  JMP
  CMP
  Conditional jumps (branches)
  Conditional MOV instruction
Conditional statements

Normally: **Sequential Execution of instructions**

**Changing the Flow of Control**

JUMP and CALL instructions

**Some jumps and calls are conditional**

Flow of control constructs in “C”:

```c
if (x) {...} else {...}
while (x) {...}
do {...} while (x)
for (i=0; i<max; i++) {...}
switch (x) {
    case 1: ...
    case 2: ...
}
```
Condition Code Register

A register in the processor

*eflags* (*extended flags*)

Each bit is a flag, or *condition code*

- **CF** Carry Flag
- **SF** Sign Flag
- **ZF** Zero Flag
- **OF** Overflow Flag

Not like general purpose register

- Do not read/write directly
- Flags are modified by hardware
- Depending on the result of an instruction

“set” (=1)    “cleared” (=0)
Condition Codes

Automatically Set/Cleared by Arithmetic and Logical Operations

Example: `addl Src, Dest`

C analog: `t = a + b`

CF (carry flag)
- set if unsigned overflow (carry out from MSB)
  `(unsigned t) < (unsigned a)`

ZF (zero flag)
- set if `t == 0`

SF (sign flag)
- set if `t < 0`

OF (overflow flag)
- set if signed (two’s complement) overflow
  `(a>0 && b>0 && t<0) || (a<0 && b<0 && t>=0)`

Set/Cleared by `compare` and `test` operations as well.

Not modified by `lea, push, pop, mov` instructions.
Condition Codes

The compare instruction sets the condition codes

```
cmpq    b, a
```

Compute \( a-b \) without altering the destination

- **CF** set if carry out from most significant bit
- Used for unsigned comparisons

- **ZF** set if \( a == b \)
- **SF** set if \( (a-b) < 0 \)
- **OF** set if two’s complement overflow
  - \( (a>0 \land b<0 \land (a-b)<0) \lor (a<0 \land b>0 \land (a-b)>0) \)

**Variations**

```
cmpb    b, a
cmpw    b, a
cmpl    b, a
cmpq    b, a
```
Condition Codes

The **test** instruction also sets the condition codes

```assembly
    testq   b, a
```

**Computes a\&b without setting destination**
- Sets condition codes based on result
- Useful to have one of the operands be a mask

**Often used to test zero, positive**

```assembly
    testq   %rax, %rax
```

- ZF set when \( a\&b == 0 \)
- SF set when \( a\&b < 0 \)

**Variations**

```assembly
    testb   b, a
    testw   b, a
    testl   b, a
    testq   b, a
```
Jump Instructions

Change sequential flow of execution

One operand: the jump target
Variations: conditional or unconditional

Unconditional Jumps

Direct jump

jmp MyLoopLabel
jmp .L1
jmp 0x0040C0

Indirect Jump

jmp *Operand
jmp *%rax

Jump target is specified by a register or memory location

Conditional Jumps

Based on the condition codes!
## Jump Instructions

### Jump 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, jz</td>
<td>ZF</td>
<td>Equal / Zero</td>
</tr>
<tr>
<td>jne, jnz</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^0F) &amp; ~ZF</td>
<td>Greater (Signed)</td>
</tr>
<tr>
<td>jge</td>
<td>~(SF^0F)</td>
<td>Greater or Equal (Signed)</td>
</tr>
<tr>
<td>jl</td>
<td>(SF^0F)</td>
<td>Less (Signed)</td>
</tr>
<tr>
<td>jle</td>
<td>(SF^0F)</td>
<td>Less or Equal (Signed)</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>

*Overflow flips result*
Jump Instructions

What's the difference between jg and ja?

Which one would you use to compare two pointers?
Conditional Branch Example

gcc -Og -S -fno-if-conversion control.c

long absdiff
  (long x, long y)
{
  long result;
  if (x > y)
    result = x-y;
  else
    result = y-x;
  return result;
}
Expressing with Goto Code

if (condition) {
    Then Statements
    Then Statements
    Then Statements
} else {
    Else Statements
    Else Statements
    Else Statements
}

if (not condition) goto Else;
    Then Statements
    Then Statements
    Then Statements
goto Done;
Else:
    Else Statements
    Else Statements
    Else Statements
Done:
Expressing with Goto Code

C allows goto statement

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;
}
The SetXX Instructions

Set low-order byte of destination to 0x00 or 0x01 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>settle</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>
The SetXX Instructions

Set low-order byte of destination to 0x00 or 0x01 based on combinations of condition codes

Does not alter remaining 7 bytes.

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

Typically use movzbl to finish job

(The 32-bit instructions also zero out the upper 32-bits.)

```
gt:
    cmpq  %rsi, %rdi  # Compare x:y
    setg  %al         # Set when >
    movzbl %al, %eax  # Zero rest of %rax
    ret
```
Conditional Expressions

An expression operator in “C”

\[(Test \ ? \ Then\_Expr \ : \ Else\_Expr)\]

Example

\[val = x>y \ ? \ x-y \ : \ y-x;\]

Translation, using goto code:

\[
\begin{align*}
\text{n} & \text{test} = \neg Test; \\
\text{if} (\text{n} & \text{test}) & \text{goto Else;} \\
\text{val} & = Then\_Expr; \\
\text{goto} & \text{ Done;} \\
\text{Else:} & \\
 & \text{val} = Else\_Expr; \\
\text{Done:} & \\
\ldots \\
\end{align*}
\]

Create separate code regions for then & else expressions
Execute the appropriate one
The Conditional Move Instructions

Problem:
Branches are very disruptive to instruction flow through pipelines!

A group of instructions:
What they do:
if (Test) Dest ← Src
Example:
cmovge %rax, %rbx

Benefits:
Conditional moves do not require control transfer!
GCC tries to use them (Not always possible)

C Code
```c
result = Test
? Then_Expr
: Else_Expr;
```

Goto Version
```c
result = Then_Expr;
temp = Else_Expr;
nt = !Test;
if (nt) result = temp;
```
The Conditional Move Instructions

```c
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
```

```
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
```
Bad Cases for Conditional Move

Expensive Computations

\[ \text{val} = \text{Test}(x) \ ? \ \text{Hard1}(x) \ : \ \text{Hard2}(x); \]

- Both values get computed
- Only makes sense when computations are very simple

Risky Computations

\[ \text{val} = p \ ? \ *p \ : \ 0; \]

- Both values get computed
- May have undesirable effects

Computations with side effects

\[ \text{val} = x > 0 \ ? \ x*=7 \ : \ x+=3; \]

- Both values get computed
- Must be side-effect free
Loops

Implemented in assembly via tests and jumps

Compilers implement most loops as do-while

- Add additional check at beginning to get “while-do”

Convenient to write using “goto” in order to understand assembly implementation

```
do {                    while (test-expr) {
    body-statements     body-statements
} while (test-expr);    }
```

goto version

```
loop:                 t = test-expr
    body-statements if (not t) goto exit
    t = test-expr    loop:
    body-statements if (t) goto loop
exit:                 
```
Loops

Implemented in assembly via tests and jumps

Compilers implement most loops as do-while
  ● Add additional check at beginning to get “while-do”

Convenient to write using “goto” in order to understand assembly implementation

do-while

do {
  body-statements
} while (test-expr);

while-do

while (test-expr) {
  body-statements
}

goto version

loop:
  body-statements
  t = test-expr
  if (t) goto loop

goto version

goto test

loop:
  body-statements

test:
  t = test-expr
  if (t) goto loop
C examples

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

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

```
factorial_goto:
    movl    $1, %eax       ; eax = result = 1
    .L2:
        imull   %edi, %eax     ; result = result*x
        subl    $1,%edi        ; x--
        cmpl    $1,%edi        ; if x > 1
        jg      .L2            ;    goto .L2
        rep ret                ; return
```
“do-while” example revisited

*C code: do-while*

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

*C code: while-do*

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

*Are these equivalent?*
Assembly:  do-while

```assembly
factorial_do:
    movl $1, %eax

.L2:
    imull %edi, %eax
    subl $1, %edi
    cmpl $1, %edi
    jg .L2
rep ret
```

while-do

```assembly
factorial_while:
    movl $1, %eax
    cmpl $1, %edi
    jle .L6

.L2:
    imull %edi, %eax
    subl $1, %edi
    cmpl $1, %edi
    jg .L2

.L6:
    rep ret
```
“For” Loop Example

```c
int factorial_for(int x)
{
    int result;
    for (result=1; x > 1; x=x-1) {
        result *= x;
    }
    return result;
}
```

Init

- `result = 1`

Test

- `x > 1`

Update

- `x = x - 1`

Body

```c
{ result *= x;
}
```

Is this code equivalent to the do-while version or the while-do version?
For Loop Example

```c
int factorial_for(int x)
{
    int result;
    for (result=1; x > 1; x=x-1) {
        result *= x;
    }
    return result;
}
```

General Form

```
for (Init; Test; Update )
{
    Body
    Init;
    if (not Test) goto exit;
    loop:
    Body;
    Update;
    if (Test) goto loop;
exit:
```

- **Init**
  - `result = 1`

- **Test**
  - `x > 1`

- **Update**
  - `x = x - 1`

- **Body**
  ```
  { result *= x; }
  ```

Is this code equivalent to the do-while version or the while-do version?
### “For” Loop Example

#### factorial_for:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>movl $1,%eax</td>
<td>Init;</td>
</tr>
<tr>
<td>cmpl $1,%edi</td>
<td>Body;</td>
</tr>
<tr>
<td>jle .L6</td>
<td>Update;</td>
</tr>
<tr>
<td>imull %edi, %eax</td>
<td></td>
</tr>
<tr>
<td>subl $1,%edi</td>
<td></td>
</tr>
<tr>
<td>cmpl $1,%edi</td>
<td></td>
</tr>
<tr>
<td>jg .L2</td>
<td></td>
</tr>
<tr>
<td>rep ret</td>
<td></td>
</tr>
</tbody>
</table>

#### Code:

```
For Loop Example
Init;
if (not Test) goto exit;
loop:
Body;
Update;
if (Test) goto loop;
exit:
```
### “For” Loop Example

**factorial_for:**

```
.movl    $1,%eax
.cmpl    $1,%edi
.jle     .L6

.L2:
.imull   %edi, %eax
.subl    $1,%edi
.cmpl    $1,%edi
.jg      .L2

.L6:
.rep ret
```

**factorial_while:**

```
.movl    $1,%eax
.cmpl    $1,%edi
.jle     .L6

.L2:
.imull   %edi, %eax
.subl    $1,%edi
.cmpl    $1,%edi
.jg      .L2

.L6:
.rep ret
```
Reverse Engineer This!

```assembly
loop:
    subl $1, %edx
    js .L18
    imull %edi, %esi
    movl $0, %eax
.L17:
    addl %esi, %eax
    subl %edi, %edx
    jns .L17
    rep ret
.L18:
    movl $0, %eax
    ret
```

```c
int loop(int x, int y, int z)
{
    int result=0;
    int i;
    for (i = ____ ; i ___ ; i = ____ )
    {
        result += ___ ;
    }
    return result;
}
```

What registers hold result and i?
What is the initial value of i?
What is the test condition on i?
How is i updated?
What instructions increment result?
Reverse Engineer This!

What registers hold result and i? %eax = result, %edx = i
What is the initial value of i? i = z-1
What is the test condition on i? i >= 0
How is i updated? i = i - x
What instructions increment result? addl (x*y)
C Switch Statements

Test whether an expression matches one of a number of constant integer values.

Branch accordingly.

Missing “break”?

Fall through to the next case.

No matching case?

Execute “default” case.

```c
int switch_eg (int x) {
    int result = x;
    switch (x) {
        case 100:
            result *= 13;
            break;
        case 102:
            result += 10;
            /* Fall through */
        case 103:
            result += 11;
            break;
        case 104:
        case 106:
            result *= result;
            break;
        default:
            result = 0;
    }
    return result;
}
```
C Switch Statements

Implementation options

- Series of conditionals
  testl/cmpl followed by je
  Good, if only a few cases
  Slow, if many cases

- Jump table (example below)
  Build a table of addresses
  Use the index value as an offset into this table
  Each table entry points to the right chunk of code
  Do an “indirect jump” through the table
  Possible with a small range of integer constants

GCC picks implementation based on the actual switch values.
C Switch Statements

Example:

```c
switch (x) {
    case 1:
    case 5:
        code at L0
    case 2:
    case 3:
        code at L1
    default:
        code at L2
}
```

JumpTable:

- Code for cases 1, 5
- Code for cases 2, 3
- Code for default case

Check that $0 \leq x \leq 5$
if not, goto .L2

```asm
%rax = .L3 + (4 *x)
jmp * %rax
```

```
.L3:
    .quad .L2
    .quad .L0
    .quad .L1
    .quad .L1
    .quad .L2
    .quad .L0
```
int switch_eg (int x) {
    int result = x;
    switch (x) {
        case 100:
            result *= 13;
            break;
        case 102:
            result += 10;
            /* Fall through */
        case 103:
            result += 11;
            break;
        case 104:
        case 106:
            result *= result;
            break;
        default:
            result = 0;
    }
    return result;
}
int switch_eg (int x) {
    int result = x;
    switch (x) {
        case 100:
            result *= 13;
            break;
        case 102:
            result += 10;
            /* Fall through */
        case 103:
            result += 11;
            break;
        case 104:
        case 106:
            result *= result;
            break;
        default:
            result = 0;
    }
    return result;
}
Reverse Engineering Challenge

The body of the switch statement has been omitted in the above C program. The code has case labels that did not span a contiguous range, and some cases had multiple labels. GCC generates the code shown when compiled. Variable x is initially at offset 8 relative to register %ebp.

a) What were the values of the case labels in the switch statement body?
b) What cases had multiple labels in the C code?
Reverse Engineering Challenge

```c
int switch2(int x) {
    int result = 0;
    switch (x) {
        ...???...
    }
    return result;
}
```

Sets start range to -2
Top range is 4

```asm
addl $2, $edi
cmpl $6, %edi
ja .L25
movl %edi,%edi
jmp *.L27(,%rdi,8)
.align 8

.L27:
.quad .L26 -2
.quad .L25 -1
.quad .L32 0
.quad .L29 1
.quad .L30 2
.quad .L30 3
.quad .L31 4
```

```c
int switch2(int x) {
    int result = 0;
    switch (x) {
        case -2:
            /* Code at .L26 */
        case 0:
            /* Code at .L32 */
        case 1:
            /* Code at .L29 */
        case 2,3:
            /* Code at .L30 */
        case 4:
            /* Code at .L31 */
        case -1:
            default:
            /* Code at .L25 */
    }
    return result;
}
```
“For” Loop Example: ipwr

\[ 3^{11} = 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \]

10 multiplications

\[ 3^{47} = 3 \times 3 \times 3 \times 3 \times 3 \times \ldots \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \]

n-1 multiplications

Is there a better algorithm?
“For” Loop Example: \( ipwr \)

\[ 3^{11} = 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \]

10 multiplications

\[ 3^{47} = 3 \times 3 \times 3 \times 3 \times 3 \times \ldots \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \]

n-1 multiplications

Is there a better algorithm?

\[ 3^{11} = 3^{1+2+8} = 3^{1+2+0+8+0+\ldots} = 3^1 \times 3^2 \times 3^8 \]
“For” Loop Example: ipwr

\[ 3^{11} = 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \]

10 multiplications

\[ 3^{47} = 3 \times 3 \times 3 \times 3 \times 3 \times \ldots \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \times 3 \]

n-1 multiplications

Is there a better algorithm?

\[ 3^{11} = 3^{1+2+8} = 3^{1+2+0+8+0+\ldots} = 3^1 \times 3^2 \times 3^4 \times 3^8 \times 3^{16} \times \ldots \]
\[ = 3 \times 3^2 \times (3^2)^2 \times ((3^2)^2)^2 \times (((3^2)^2)^2)^2 \times \ldots \]
“For” Loop Example: ipwr

Algorithm

Exploit property that \( p = p_0 + 2p_1 + 4p_2 + \ldots + 2^{n-1}p_{n-1} \)

Gives: \( x^p = z_0 \cdot z_1^2 \cdot (z_2^2)^2 \cdot \ldots \cdot (\ldots((z_{n-1}^2)^2\ldots)^2)^2 \)

- \( z_i = 1 \) when \( p_i = 0 \)
- \( z_i = x \) when \( p_i = 1 \)

Complexity \( O(\log p) \)

\[
3^{11} = 3^{1+2+8} = 3^{1+2+0+8+0+\ldots} = 3^1 \times 3^2 \times 3^4 \times 3^8 \times 3^{16} \times \ldots \\
= 3 \times 3^2 \times (3^2)^2 \times ((3^2)^2)^2 \times (((3^2)^2)^2)^2 \times \ldots
\]
"For" Loop Example: ipwr

/* Compute x raised to 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;
}

Algorithm

Exploit property that $p = p_0 + 2p_1 + 4p_2 + \ldots + 2^{n-1}p_{n-1}$

Gives: $x^p = z_0 \cdot z_1^2 \cdot (z_2^2)^2 \cdot \ldots \cdot (\ldots((z_{n-1}^2)^2)^2\ldots)^2$

$z_i = 1$ when $p_i = 0$
$z_i = x$ when $p_i = 1$

Complexity $O(\log p)$

$3^{11} = 3^{1+2+8} = 3^{1+2+0+8+0+\ldots} = 3^1 \times 3^2 \times 3^4 \times 3^8 \times 3^{16} \times \ldots$

$= 3 \times 3^2 \times (3^2)^2 \times ((3^2)^2)^2 \times (((3^2)^2)^2)^2 \times \ldots$
"For" Loop Example: ipwr

/* Compute x raised to 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;
}

<table>
<thead>
<tr>
<th>result</th>
<th>x</th>
<th>p</th>
<th>p</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>11</td>
<td>1011</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>5</td>
<td>101</td>
</tr>
<tr>
<td>27</td>
<td>81</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>27</td>
<td>6561</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>177,147</td>
<td>43,046,721</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

$$3^{11} = 3^{1+2+8} = 3^{1+2+0+8+0+\ldots} = 3^1 \times 3^2 \times 3^4 \times 3^8 \times 3^{16} \times \ldots$$
$$= 3 \times 3^2 \times (3^2)^2 \times ((3^2)^2)^2 \times (((3^2)^2)^2)^2 \times \ldots$$
“For” Loop Example: ipwr

```c
/* Compute x raised to 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;
}
```

<table>
<thead>
<tr>
<th>result</th>
<th>1</th>
<th>3</th>
<th>27</th>
<th>177,147</th>
<th>43,046,721</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
</table>

3^{11} = 3^{1+2+8} = 3^{1+2+0+8+0+...} = 3 \times 3^2 \times 3^4 \times 3^8 \times 3^{16} \times ...
= 3 \times 3^2 \times (3^2)^2 \times ((3^2)^2)^2 \times (((3^2)^2)^2)^2 \times ...
Summary

C Control
- if-then-else
- do-while
- while
- switch

Assembler Control
- Jump
- Conditional Jump

Compiler
- Must generate assembly code to implement more complex control

Standard Techniques
- All loops converted to do-while form
- Large switch statements use jump tables

Conditions in CISC
- CISC machines generally have condition code registers

Conditions in RISC
- Use general registers to store condition information
- Special comparison instructions
- E.g., on Alpha:
  \[ \text{cmple} \; 16, 1, 1 \]
  Sets register $1$ to $1$ when Register $16 \leq 1$