Logical & Relational Operators
How to represent the following in assembly?

\(x < 10 \land \land y > 3\)

Answer: it depends on the target machine.

Several approaches:

- Numerical representation
- Positional Encoding (e.g. MIPS assembly)
- Conditional Move and Predication

Correct choice depends on both context and ISA (Instruction Set Architecture)
Assign numerical values to \texttt{true} and \texttt{false}.

- In C, \texttt{false} = 0 and \texttt{true} = anything else.

Use comparison operator from the ISA to get a value from a relational operator:

- MIPS has \texttt{SLT} instruction (Set Less Than);
- and \texttt{SLTU} instruction (Set Less Than Unsigned)
### Examples

Assuming $x$ and $y$ are in registers $x$ and $y$.

<table>
<thead>
<tr>
<th>Expression</th>
<th>Machine Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x &lt; y$</td>
<td><code>slt $t0, $x, $y</code></td>
</tr>
</tbody>
</table>
| $x \leq y$  | `slt $t0, $y, $x`
                  | `xori $t1, $t0, 0x1` |
| $x == y$    | `xor $t0,$x,$y`
                  | `sltu $t1,$t0,1`   |
| $x \neq y$  | `xor $t0,$x,$y`
                  | `sltu $t1,$zero,$t0` |

For the other two missing relational operators, swap the arguments.
Positional Encoding

What if the ISA does not provide comparison operators?

- Use conditional branch to interpret the result of a relational operator.

**Example:** \( x < y \)

```
blt $x, $y, LT
li $t0, 0
j END
LT : li $t0, 1
END : ...
```
The absence of comparison instructions is not as bad as you think.

Most boolean expressions are used with branching anyway.

### Example

```c
if (x < y)
    z = 3;
else
    z = 4;
```

### Corresponding assembly code

```
bge $x, $y, ELSE
li $z, 3
j END
ELSE:
    li $z, 4
END:
...```

What about logical operators `&&` and `||`?

In the general case, **must** use branching!

<table>
<thead>
<tr>
<th>Example with function calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>`foo()</td>
</tr>
</tbody>
</table>

If `foo()` evaluates to true, `bar` is never called!

<table>
<thead>
<tr>
<th>Simpler example</th>
</tr>
</thead>
<tbody>
<tr>
<td>`x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Corresponding assembly code</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>bne $x, $zero, TRUE</code></td>
</tr>
<tr>
<td><code>bne $y, $zero, TRUE</code></td>
</tr>
<tr>
<td><code>li $t0, 0</code></td>
</tr>
<tr>
<td><code>j END</code></td>
</tr>
<tr>
<td><code>TRUE: li $t0, 1</code></td>
</tr>
<tr>
<td><code>END: ...</code></td>
</tr>
</tbody>
</table>
If supported by your ISA, simple approach consists of using numerical encoding for relational operators and positional for logical operators.

**Example**

\[ x < 4 \quad || \quad y < 6 \]

**Corresponding assembly code**

```assembly
li $t0, 4
slt $t1, $x, $t0
bne $t1, $zero, TRUE

li $t2, 6
slt $t3, $y, $t2
bne $t3, $zero, TRUE

li $t4, 0
j END

TRUE: li $t4, 1
END: ...
```
Conditional move and predication can simplify code (if ISA supports it!).

**Example**

```plaintext
if (x < y)
  z = 3;
else
  z = 4;
```

**Corresponding (naive) assembly code**

<table>
<thead>
<tr>
<th>Conditional Move</th>
<th>Predicated Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>slt $t0, $x, $y</td>
<td>slt $t0, $x, $y</td>
</tr>
<tr>
<td>li $t1, 3</td>
<td>$t0?li $z, 3</td>
</tr>
<tr>
<td>li $t2, 4</td>
<td>$t0?li $z, 4</td>
</tr>
<tr>
<td>cmov $z, $t0, $t1, $t2</td>
<td></td>
</tr>
</tbody>
</table>

Unfortunately, these instructions are not available on MIPS.
Last words on logical and relational operators

Best choice depends on two things

- ISA instructions available
- Context
Logical & Relational Operators

Implementation
Labels

Need to have unique labels that we can emit.

**Label class**

```java
class Label {
    static counter = 0;
    String name;
    Label() { name = "label"+counter++; }
}
```
Visitor Implementation (Expression)

ExpressionVisitor

```java
Register visitBinOp(BinOp bo) {
    Register lhsReg = bo.lhs.accept(this);
    Register resReg = new VirtualRegister();

    switch (bo.op) {
        ...
        case LT:
            Register rhsReg = bo.rhs.accept(this);
            emit("slt", resReg, lhsReg, rhsReg);
            break;
        ...
```
ExpressionVisitor (cont.)

```java
case OR:
    Label trueLbl = new Label();
    Label endLbl = new Label();
    emit("bne", lhsReg, zeroReg, trueLbl);

    Register rhsReg = bo.rhs.accept(this);
    emit("bne", rhsReg, zeroReg, trueLbl);

    emit("li", resReg, 0);
    emit("j", endLbl);

    emit(trueLbl);
    emit("li", resReg, 1);

    emit(endLbl);
    ...
}
return resReg;
```
Control-Flow
Control-Flow

- If-then-else
- Loops (for, while, ...)
- Switch/case statements
If-then-else

Follow the model for evaluating relational and boolean with branches.

Branching versus predication (e.g. IA-64, ARM ISA) trade-off:

- Frequency of execution:
  uneven distribution, try to speedup common case
- Amount of code in each case:
  unequal amounts means predication might waste issue slots
- Nested control flow:
  any nested branches complicates the predicates and makes branching attractive
Loops

- evaluate condition before the loop (if needed)
- evaluate condition after the loop
- branch back to the top (if needed)

**while, for and do while** loops all fit this basic model.
**Example: for loop**

```c
for (i=1; i<100; i++) {
    body
}
next stmt
```

**Corresponding assembly**

```assembly
li $t0, 1
li $t1, 100
bge $t0, $t1, NEXT

BODY: body
    addi $t0, $t0, 1
    blt $t0, $t1, BODY

NEXT: next stmt
```
Exercise

Write the assembly code for the following while loop:

```plaintext
while (x >= y) {
    body
}
next stmt
```
Most modern programming languages include a `break` statement (loops, switch statements)

```plaintext
for (....) {
    ...
    if (....)
        break;
    ...
}
```

In such cases, use an unconditional branch to the next statement following the control-flow construct (loop or case statement).

For `skip` or `continue` statement, branch to the next iteration (loop start)
Case Statement (switch)

Case statement

switch (c) {
    case 'a': stmt1;
    case 'b': stmt2; break;
    case 'c': stmt3;
}

1. Evaluate the controlling expression
2. Branch to the selected case
3. Execute the code for that case
4. Branch to the statement after the case

Part 2 is key!

Strategies:

- Linear search (nested if-then-else)
- Build a table of case expressions and use binary search on it
- Directly compute an address (requires dense case set)
Exercise

Knowing that the character 'a' corresponds to the decimal value 97 (ASCII table), write the assembly code for the example below using linear search.

```c
char c;
...
switch (c) {
    case 'a': stmt1;
    case 'b': stmt2; break;
    case 'c': stmt3; break;
    case 'd': stmt4;
}
stmt5;
```

Exercise: can you do it without any conditional jumps?

Hint: use the JR MIPS instruction which jumps directly to an address stored in a register.
Control-Flow

Implementation
If statement

StmtCodeGenVisitor implements Visitor<Void> {

Void visitIf(If ifStmt) {
    Register cond = ifStmt.cond.accept(new ExprVisitor());

    Label elseLbl = new Label();
    Label endLbl = new Label();

    emit("beq", cond, zeroReg, elseLbl);

    ifStmt.then.accept(this);
    emit("j", endLbl);

    emit(elseLbl);
    ifStmt.els.accept(this); // assumes else is present

    emit(endLbl);

    return null;
}

Next lecture

More code generation:

- Memory Allocation
- Function Call
- References vs. Values