Compiler Design

Lecture 13: Code generation : Logical & Relational Operators, and Control Flow

Christophe Dubach

Winter 2024

Timestamp: 2024/02/13 14:10:00

Logical & Relational Operators

How to represent the following in assembly?

```
x<10 && y>3
```

Answer: it depends on the target machine.

Several approaches:

- Numerical representation
- Positional Encoding
- Conditional Move and Predication

Correct choice depends on both context and ISA (Instruction Set Architecture)

Numerical Representation

Assign numerical values to true and false

- In C, false = 0 and true = anything else.

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

- MIPS has SLT instruction (Set Less Than);
- and SLTU instruction (Set Less Than Unsigned)
- `slt $1, $2, $3 # if ($2<$3) $1=1 else $1=0`

Examples

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

`x < y`	`slt $t0, $x, $y`
`x <= y`	`slt  $t0, $y, $x   # y<x` `xori $t1, $t0, 0x1 # reverse result`
`x == y`	`xor  $t0,$x,$y  # which bits different?` `sltu $t1,$t0,1  # no difference if 0`
`x != y`	`xor  $t0,$x,$y     # which bits different?` `sltu $t1,$zero,$t0 # different if 0 <`

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

Positional Encoding

What if the ISA does not provide comparison operators?

- Branch conditionally to the position that loads true or false.

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

```
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!

Example with function calls

```
foo() || bar()
```

If `foo()` returns true, `bar` is never called! This is called a short-circuit.

Simpler example

```
x || y
```

Corresponding assembly code

```
      bne $x, $zero, TRUE
      bne $y, $zero, TRUE
      li $t0, 0
      j END
TRUE: li $t0, 1
END: ...
```

Combining Logical and Relational Operators

If supported by the ISA, simplest approach consists of using numerical encoding for relational operators and positional for logical operators.

Example

```
x<4 || y<6
```

Corresponding assembly code

```
      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

Conditional move and predication can simplify code if available

Example

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

Corresponding (naive) assembly code

Conditional Move	Predicated Execution

`li   $t1, 3`
`li   $t2, 4`
`slt  $t0, $x, $y`
`cmov $z, $t0, $t1, $t2`
	`    slt $t0, $x, $y`
	`$t0?li  $z, 3`
	`$t0?li  $z, 4`

These instructions are not available on MIPS, but they are on:

- ARM: condition flags
- X86: conditional move

Last words on logical and relational operators

Best choice depends on two things

- ISA instructions/features available, *e.g.*:
    - SLT instruction;
    - Predication support.
- Context, *e.g.*:
    - Assignment of same value in each branch of an if-then-else;
    - Presence of short-circuit logical operators.

Logical & Relational Operators

Code Generation

Labels

Need to have unique labels that we can emit.

Label class

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

Pattern-Matching Expressions

Expression code generator class

```
class ExprCodeGen {

  Register visit(Expr expr) {
    return switch(expr) {
      case ... ->
      case ... ->
    }
  }
}
```

Pattern-Matching Expressions

LT Expression

```
case BinOp bo -> {
   Register lhsReg = visit(bo.lhs);
   Register resReg = newVirtualRegister();

   switch(bo.op) {
     ...
     case LT:
       Register rhsReg = visit(bo.rhs);
       emit("slt", resReg, lhsReg, rhsReg);
       break;
     ...

  }

   yield resReg;
}
```

Logical OR || Expression

```
case BinOp bo -> {
   Register lhsReg = visit(bo.lhs);
   Register resReg = newVirtualRegister();

   switch(bo.op) {
     ...
     case OR:
       Label trueLbl = new Label();
       Label endLbl  = new Label();
       emit("bne", lhsReg, zeroReg, trueLbl);

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

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

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

       emit(endLbl);
     ...
   }
   yield resReg;
 }
```

Control-Flow

Control-Flow

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

If-then-else

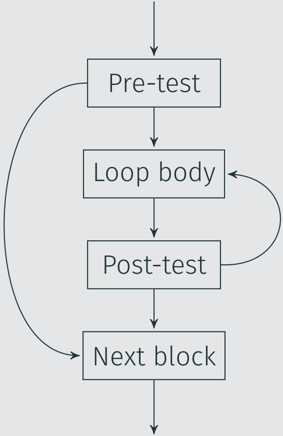
Follow the model for evaluating relational and boolean with branches.

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

Basic pattern

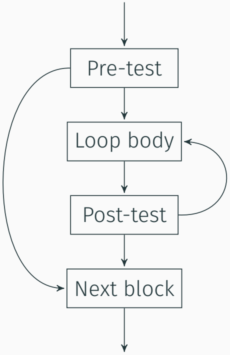


- evaluate condition before the loop (if needed)
- evaluate condition after the loop
- branch back to the top (if needed)

while, for and do while loops follow a very similar pattern.

Example: for loop

```
for (i=0; i<100; i++) {
  body
}
next stmt
```



Corresponding assembly

```
      li $t0, 0

      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:

```
while (x >= y) {
  body
}
next stmt
```

Break/continue

Most modern programming languages include a break statement (loops, switch statements)

```
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/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.

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

Food for thoughts: on the dangers of fallthrough switch cases

With C (and many other languages), default behaviour is to fallthrough to next case, unless `break` is used. This behaviour directly matches assembly code.

This is often a source of bugs if programmers forget to use `break` !

To prevent this:

- Some languages makes it mandatory to have a `break` (*e.g.* C#).
- Many languages (*e.g.* Scala, Pascal, Ada) opt to have an implicit break by default and don't allow fallthrough.
- Others don't fallthrough unless `next` (continue) is used at the end of the case.

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.

```
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.

We can now find the matching case in $O(1)$!

Control-Flow

Code Generation

Pattern-Matching Statements

No register to return this time.

Statement code generator class

```
class StmtCodeGen {

  void visit(Stmt stmt) {
    switch(stmt) {
      case ... ->
      case ... ->
    }
  }
}
```

Pattern-Matching Statements

If statement

```
case If ifStmt -> {
  Register cond = (new ExprCodeGen()).visit(ifStmt.cond);

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

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

  visit(ifStmt.then);
  emit("j", endLbl);

  emit(elseLbl);
  visit(ifStmt.els); // assumes 'else' always presents

  emit(endLbl);
}
```

Next lecture

More code generation:

- Memory Allocation
- Function Call
- References vs. Values