Code Shape
Booleans, Relationals, & Control flow

Boolean & Relational Values

How should the compiler represent them?

- Answer depends on the target machine

Two classic approaches

- Numerical representation
- Positional (implicit) representation

Correct choice depends on both context and ISA

Boolean & Relational Values

Numerical representation

- Assign values to TRUE and FALSE
- Use hardware AND, OR, and NOT operations
- Use comparison to get a boolean from a relational expression

Examples

x < y becomes cmp_LT $r_x, r_y \rightarrow r_1$

if (l < r) becomes cmp_LT $r_l, r_r \rightarrow r_1$

then stm t_1 cbr $r_1 \rightarrow$ stm t_1, stm t_2

else stm t_2

Boolean & Relational Values

What if the ISA uses a condition code?

- Must use a conditional branch to interpret result of compare
- Necessitates branches in the evaluation

Example:

x < y *becomes*

cmp $r_x, r_y \Rightarrow cc_1$
cbr_LT $cc_1 \rightarrow L_T, L_F$
L_T: loadI $1 \Rightarrow r_2$
br $\rightarrow L_E$
L_F: loadI $0 \Rightarrow r_2$
L_E: …other stmts…

This “positional representation” is much more complex

Boolean & Relational Values

What if the ISA uses a condition code?

- Must use a conditional branch to interpret result of compare
- Necessitates branches in the evaluation

Example:

x < y *becomes*

```
        cmp     $r_x, r_y \Rightarrow cc_1$
        cbr_LT  $cc_1 \rightarrow L_T, L_F$
$L_T$:  loadI   $1 \Rightarrow r_2$
        br       $\rightarrow L_E$
$L_F$:  loadI   $0 \Rightarrow r_2$
$L_E$:  …other stmts…
```

Condition codes

- **are an architect's hack**
- **allow ISA to avoid some comparisons**
- **complicates code for simple cases**

This "positional representation" is much more complex

Boolean & Relational Values

The last example actually encodes result in the PC

If result is used to control an operation, this may be enough

Example

if (x < y)
 then a ← c + d
 else a ← e + f

Variations on the Iloc Branch Structure					
Straight Condition Codes			*Boolean Compares*		
	comp	$r_x, r_y \Rightarrow cc_1$		cmp_LT	$r_x, r_y \Rightarrow r_1$
	cbr_LT	$cc_1 \rightarrow L_1, L_2$		cbr	$r_1 \rightarrow L_1, L_2$
L_1:	add	$r_c, r_d \Rightarrow r_a$	L_1:	add	$r_c, r_d \Rightarrow r_a$
	br	□L_{OUT}		br	□L_{OUT}
L_2:	add	$r_e, r_f \Rightarrow r_a$	L_2:	add	$r_e, r_f \Rightarrow r_a$
	br	□L_{OUT}		br	□L_{OUT}
L_{OUT}:	nop		L_{OUT}:	nop	

Condition code version does not directly produce (x < y)

Boolean version does

Still, there is no significant difference in the code produced

Boolean & Relational Values

Conditional move & predication both simplify this code

Example

if (x < y)
then a ← c + d
else a ← e + f

Other Architectural Variations				
Conditional Move		*Predicated Execution*		
comp	$r_x, r_y \Rightarrow cc_1$		cmp_LT	$r_x, r_y \Rightarrow r_1$
add	$r_c, r_d \Rightarrow r_1$	(r_1)?	add	$r_c, r_d \Rightarrow r_a$
add	$r_e, r_f \Rightarrow r_2$	$(\neg r_1)$?	add	$r_e, r_f \Rightarrow r_a$
i2i_<	$cc_1, r_1, r_2 \Rightarrow r_a$			

Both versions avoid the branches

Both are shorter than CCs or Boolean-valued compare

Are they better?

Boolean & Relational Values

Consider the assignment $x \leftarrow a < b \wedge c < d$

Variations on the Iloc Branch Structure				
Straight Condition Codes			*Boolean Compare*	
	comp	$r_a,r_b \Rightarrow cc_1$	cmp_LT	$r_a,r_b \Rightarrow r_1$
	cbr_LT	$cc_1 \quad \rightarrow L_1,L_2$	cmp_LT	$r_c,r_d \Rightarrow r_2$
L_1:	comp	$r_c,r_d \Rightarrow cc_2$	and	$r_1,r_2 \Rightarrow r_x$
	cbr_LT	$cc_2 \quad \rightarrow L_3,L_2$		
L_2:	loadI	$0 \quad \Rightarrow r_x$		
	br	□L_{OUT}		
L_3:	loadI	$1 \quad \Rightarrow r_x$		
	br	□L_{OUT}		
L_{OUT}:	nop			

Here, the boolean compare produces much better code

Boolean & Relational Values

Conditional move & predication help here, too

$x \leftarrow a < b \wedge c < d$

Other Architectural Variations				
Conditional Move			*Predicated Execution*	
comp	r_a, r_b	$\Rightarrow cc_1$	cmp_LT	$r_a, r_b \Rightarrow r_1$
i2i_<	cc_1, r_T, r_F	$\Rightarrow r_1$	cmp_LT	$r_c, r_d \Rightarrow r_2$
comp	r_c, r_d	$\Rightarrow cc_2$	and	$r_1, r_2 \Rightarrow r_x$
i2i_<	cc_2, r_T, r_F	$\Rightarrow r_2$		
and	r_1, r_2	$\Rightarrow r_x$		

Conditional move is worse than Boolean compares

Predication is identical to Boolean compares

Context & hardware determine the appropriate choice

Control Flow

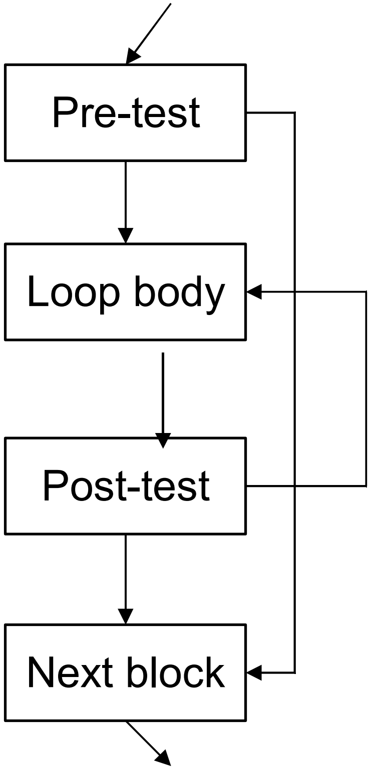
If-then-else

- Follow model for evaluating relationals & booleans with branches

Branching versus predication *(e.g., IA-64)*

- Frequency of execution
  - → Uneven distribution ⇒ do what it takes to speed common case
- Amount of code in each case
  - → Unequal amounts means predication may waste issue slots
- Control flow inside the construct
  - → Any branching activity within the case base complicates the predicates and makes branches attractive

Control Flow



Loops

- Evaluate condition before loop (if needed)
- Evaluate condition after loop
- Branch back to the top (if needed)

Merges test with last block of loop body

while, for, do, & until all fit this basic model

Loop Implementation Code

for (i = 1; i< 100; i++) { *body* }

next statement

loadI $1 \Rightarrow r_1$
loadI $1 \Rightarrow r_2$

Initialization

loadI $100 \Rightarrow r_3$
cmp_GE $r_1, r_3 \Rightarrow r_4$
cbr $r_4 \Rightarrow L_2, L_1$

Pre-test

L_1: *body*

add $r_1, r_2 \Rightarrow r_1$
cmp_LT $r_1, r_3 \Rightarrow r_5$

Post-test

cbr $r_5 \Rightarrow L_1, L_2$

L_2: *next statement*

Break statements

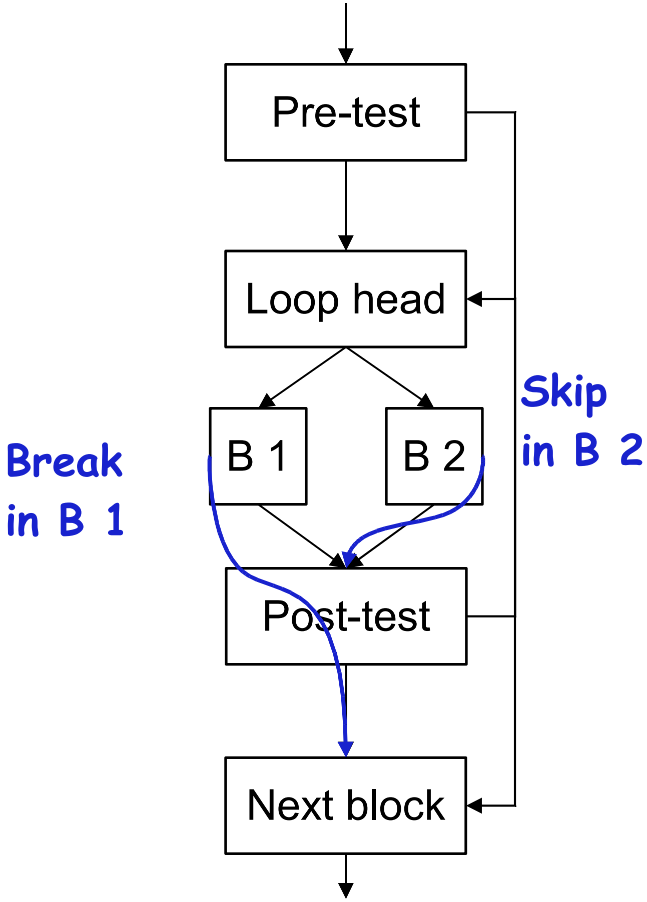
Many modern programming languages include a break

- Exits from the innermost control-flow statement
  - → Out of the innermost loop
  - → Out of a case statement

Translates into a jump

- Targets statement outside control-flow construct
- Creates multiple-exit construct
- Skip in loop goes to next iteration

Only make sense if loop has > 1 block



Control Flow

Case Statements

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

Parts 1, 3, & 4 are well understood, part 2 is the key

Control Flow

Case Statements

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 (*use break*)

Parts 1, 3, & 4 are well understood, part 2 is the key

Surprisingly many compilers do this for all cases!

Strategies

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