COMP 412
FALL 2010

Boolean & Relational Values
Control-flow Constructs

Comp 412

Boolean & Relational Values

How should the compiler represent them?

- Answer depends on the target machine

Implementation of booleans, relational expressions & control flow constructs varies widely with the ISA

Two classic approaches

- Numerical (explicit) representation
- Positional (implicit) representation

Best choice depends on both context and ISA

This material is drawn from §7.4 in EaC, where it is presented in more depth with more examples. You should read that section, along with the rest of Chapter 7. Lecture will not cover all the material in Chapter 7, but you are responsible for it.

Boolean & Relational Expressions

First, we need to recognize boolean & relational expressions

Expr	$\rightarrow$	*Expr* $\vee$ *AndTerm*	*NumExpr*	$\rightarrow$	*NumExpr* + *Term*
	$\mid$	*AndTerm*		$\mid$	*NumExpr* - *Term*
AndTerm	$\rightarrow$	*AndTerm* $\wedge$ *RelExpr*		$\mid$	*Term*
	$\mid$	*RelExpr*	*Term*	$\rightarrow$	*Term* $\times$ *Value*
RelExpr	$\rightarrow$	*RelExpr* < *NumExpr*		$\mid$	*Term* $\div$ *Value*
	$\mid$	*RelExpr* $\leq$ *NumExpr*		$\mid$	*Value*
	$\mid$	*RelExpr* = *NumExpr*	*Value*	$\rightarrow$	$\neg$ *Factor*
	$\mid$	*RelExpr* $\neq$ *NumExpr*		$\mid$	*Factor*
	$\mid$	*RelExpr* $\geq$ *NumExpr*	*Factor*	$\mid$	(*Expr*)
	$\mid$	*RelExpr* > *NumExpr*		$\mid$	number
				$\mid$	*Reference*

Reference derives a name, a subscripted name, a structure reference, a string reference, …

Boolean & Relational Values

Next, we need to represent the 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 $(x < y)$ then $stmt_1$ else $stmt_2$	becomes	cmp_LT cbr	r_x, r_y r_1	$\Rightarrow r_1$ $\rightarrow$ $\_stmt_1, \_stmt_2$

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	⇒	CC_1
			cbr_LT	CC_1	→	L_T, L_F
		L_T:	loadI	1	⇒	r_2
			br		→	L_E
		L_F:	loadI	0	⇒	r_2
		L_E:		*… other statements …*		

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

> Editorial comment: (KDC)
> CC's are an evil, seductive idea

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		
		L_E:	… *other statem*			

This "positional representation" is much mo

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

Boolean & Relational Values

The last example actually encoded result in the PC

If result is used to control an operation, that may suffice

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

Boolean version does

Still, there is no significant difference in the code produced

Example
if (x < y)
then a $\leftarrow$ c + d
else a $\leftarrow$ e + f

Straight Condition Codes				*Boolean Comparisons*			
	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		$\rightarrow L_1,L_2$
L_1:	add	r_c,r_d	$\Rightarrow r_a$	L_1:	add	r_c,r_d	$\Rightarrow r_a$
	br		$\rightarrow L_{OUT}$		br		$\rightarrow L_{OUT}$
L_2:	add	r_e,r_f	$\Rightarrow r_a$	L_2:	add	r_e,r_f	$\Rightarrow r_a$
	br		$\rightarrow L_{OUT}$		br		$\rightarrow L_{OUT}$
L_{OUT}:	nop			L_{OUT}:	nop		

Boolean & Relational Values

Other Architectural Variations

Conditional move & predication both simplify this code

Example
if (x < y)
then a $\leftarrow$ c + d
else a $\leftarrow$ e + f

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_LT	CC_1, r_1, r_2	$\Rightarrow r_a$				

Both versions avoid the branches

Both are shorter than cond'n codes or Boolean-valued compare

Are they better?

Boolean & Relational Values

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

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	$\rightarrow L_1, L_2$	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	$\rightarrow L_3, L_2$			
L_2:	loadI	0	$\Rightarrow r_x$			
	br		$\rightarrow L_{OUT}$			
L_3:	loadI	1	$\Rightarrow r_x$			
	br		$\rightarrow L_{OUT}$			
L_{OUT}:	nop					

Here, Boolean compare produces much better code

Boolean & Relational Values

Conditional move & predication help here, too

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

Conditional Move			*Predicated Execution*		
comp	r_a,r_b	$\Rightarrow CC_1$	cmp_LT	r_a,r_b	$\Rightarrow r_1$
i2i_LT	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_LT	CC_2,r_T,r_F	$\Rightarrow r_2$			
and	r_1,r_2	$\Rightarrow r_x$			

Conditional move is worse than Boolean compare
Predication is identical to Boolean compares

The bottom line:
$\Rightarrow$ Context & hardware determine the appropriate choice

Chapter 7 goes into greater depth; read it.

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 $\Rightarrow$ 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 construct complicates the predicates and makes branches attractive

Short-circuit Evaluation

Optimize boolean expression evaluation

- Once value is determined, skip rest of the evaluation

  if (x or y and z) then …

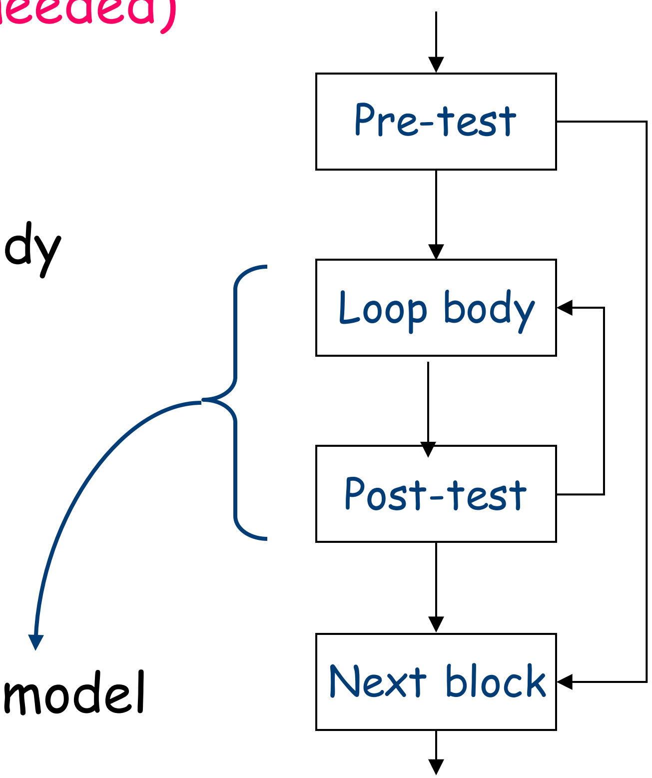
  – If x is true, need not evaluate y or z
    - → Branch directly to the "then" clause
  – On a PDP-11 or a VAX, short circuiting saved time
- Modern architectures may favor evaluating full expression
  – Rising branch latencies make the short-circuit path expensive
  – Conditional move and predication may make full path cheaper
- Past: compilers analyzed code to insert short circuits
- Future: compilers analyze code to prove legality of full path evaluation where language specifies short circuits

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

Implementing Loops

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

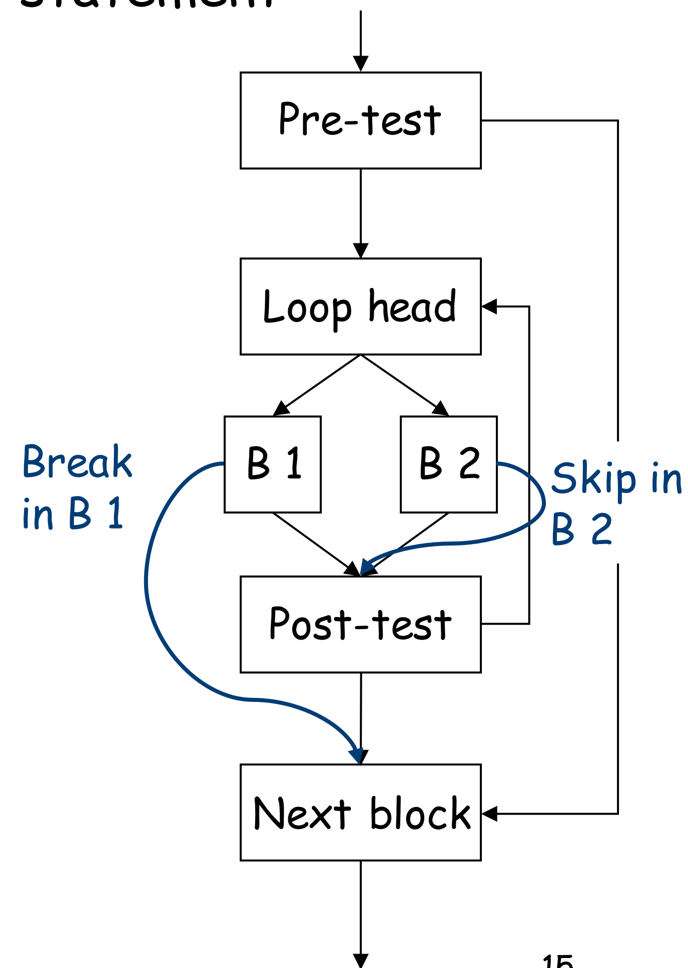
	loadI	1	$\Rightarrow r_1$	
	loadI	1	$\Rightarrow r_2$	Initialization
	loadI	100	$\Rightarrow r_3$	
	cmp_GE	r_1,r_3	$\Rightarrow r_4$	Pre-test
	cbr	r_4	$\rightarrow L_2,L_1$	
L_1:	*loop body*			
	add	r_1,r_2	$\Rightarrow r_1$	
	cmp_LT	r_1,r_3	$\Rightarrow r_5$	Post-test
	cbr	r_5	$\rightarrow L_1,L_2$	
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

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)

Case statements are a place where attention to code shape pays off handsomely.