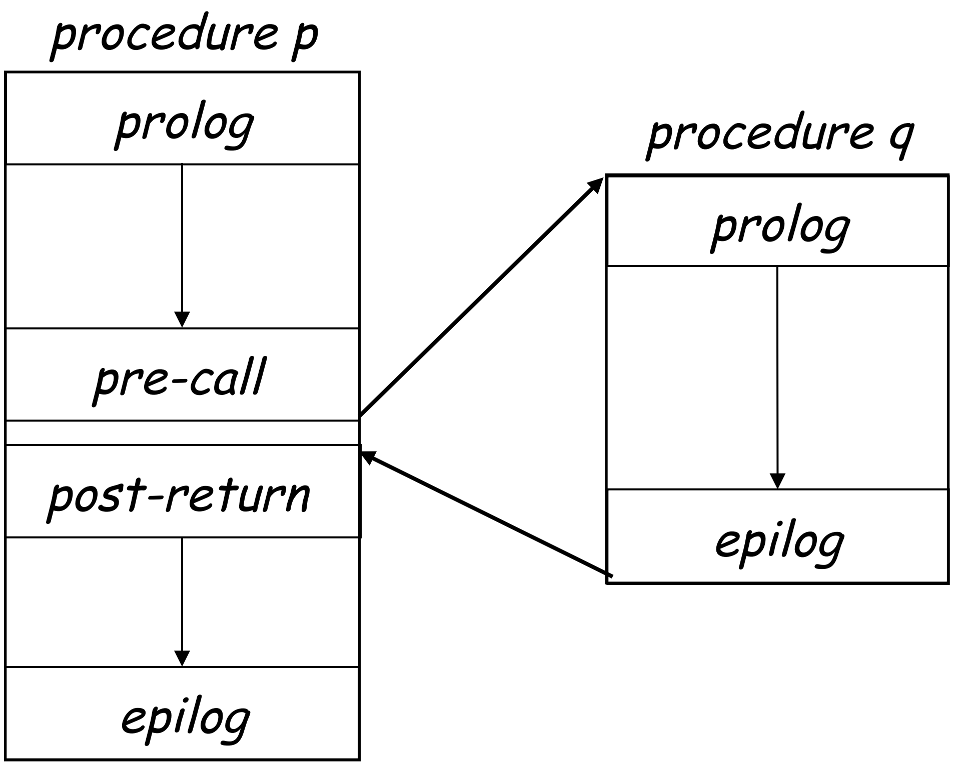
Code Shape
Procedure Calls, Dispatch, Booleans, Relationals, & Control flow

Procedure Linkages

Standard procedure linkage



Procedure has
- standard prolog
- standard epilog

Each call involves a
- pre-call sequence
- post-return sequence

These are completely predictable from the call site ⇒ depend on the number & type of the actual parameters

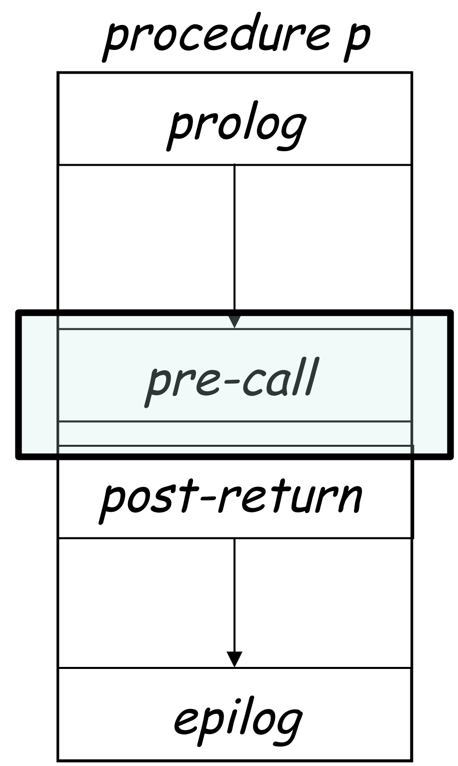
Procedure Linkages

Pre-call Sequence

- Sets up callee's basic AR
- Helps preserve its own environment

The Details

- Allocate space for the callee's AR
  - → except space for local variables
- Evaluates each parameter & stores value or address
- Saves return address, caller's ARP into callee's AR
- Save any caller-save registers
  - → Save into space in caller's AR
- Jump to address of callee's prolog code



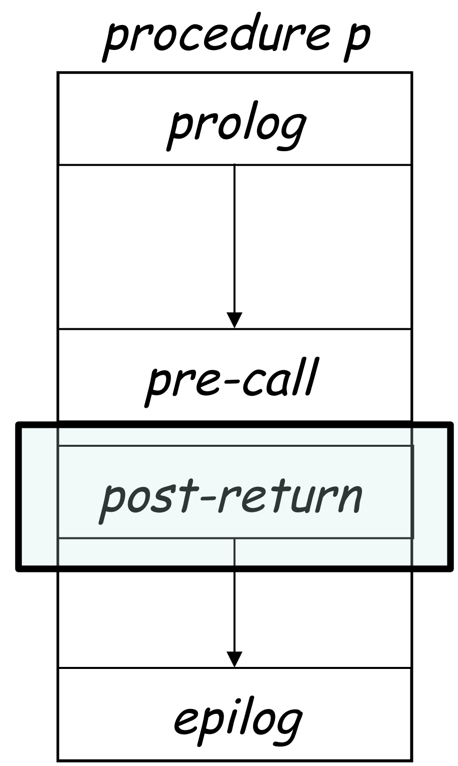
Procedure Linkages

Post-return Sequence

- Finish restoring caller's environment
- Place any value back where it belongs

The Details

- Copy return value from callee's AR, if necessary
- Free the callee's AR
- Restore any caller-save registers
- Restore any call-by-reference parameters to registers, if needed
  - → Also copy back call-by-value/result parameters
- Continue execution after the call



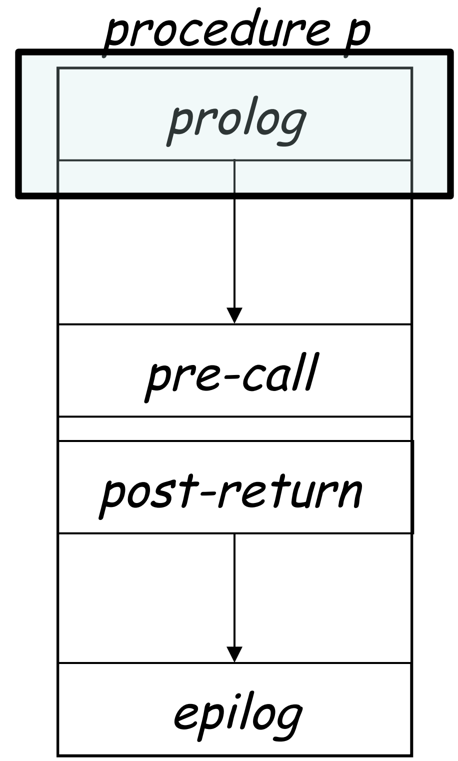
Procedure Linkages

Prolog Code

- Finish setting up the callee's environment
- Preserve parts of the caller's environment that will be disturbed

The Details

- Preserve any callee-save registers
- Allocate space for local data
  - → Easiest scenario is to extend the AR
- Find any static data areas referenced in the callee
- Handle any local variable initializations



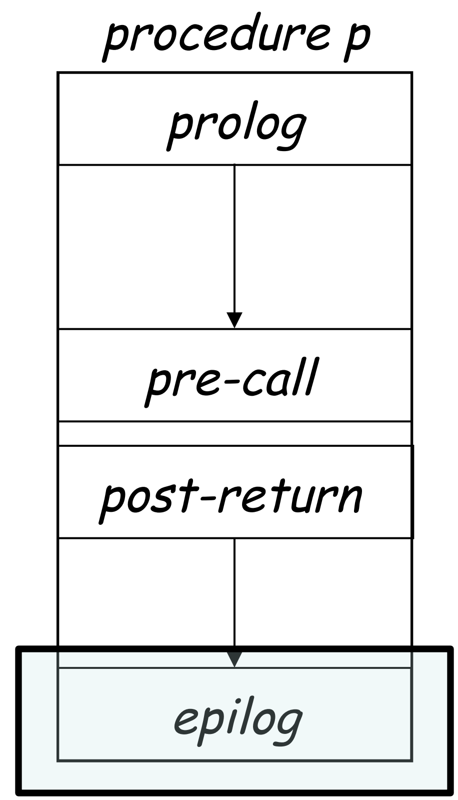
Procedure Linkages

Epilog Code

- Wind up the business of the callee
- Start restoring the caller's environment

The Details

- Store return value? No, this happens on the return statement
- Restore callee-save registers
- Free space for local data, if necessary (on the heap)
- Load return address from AR
- Restore caller's ARP
- Jump to the return address



Implementing Procedure Calls

If *p* calls *q*, one of them must

- Preserve register values (*caller-saves versus callee saves*)
  - → Caller-saves registers stored/restored by *p* in *p* 's AR
  - → Callee-saves registers stored/restored by *q* in *q* 's AR
- Allocate the AR
  - → Heap allocation ⇒ callee allocates its own AR
  - → Stack allocation ⇒ caller & callee cooperate to allocate AR

Space tradeoff

- Pre-call & post-return occur on every call
- Prolog & epilog occur once per procedure
- More calls than procedures
  - → Moving operations into prolog/epilog saves space

Implementing Procedure Calls

Evaluating parameters

- Call by reference ⇒ evaluate parameter to an lvalue
- Call by value ⇒ evaluate parameter to an rvalue & store it

Aggregates (structs), arrays, & strings are usually c-b-r

- Language definition issues
- Alternatives
  - → Small structures can be passed in registers
  - → Can pass large c-b-v objects c-b-r and copy on modification

Procedure-valued parameters

- Must pass starting address of procedure

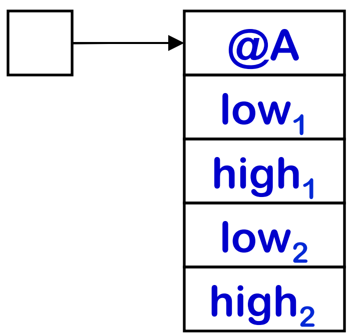
Implementing Procedure Calls

What about arrays as actual parameters?

Whole arrays, as call-by-reference parameters

- Callee needs dimension information
  - → Builds a descriptor called a *dope vector*
- Store the values in the calling sequence
- Pass the address of the dope vector in the parameter slot
- Generate complete address polynomial at each reference

dope vector



Implementing Procedure Calls

What about A[12] as an actual parameter?

If corresponding parameter is a scalar, it's easy

- Pass the address or value, as needed

What if corresponding parameter is an array?

- See previous slide

Implementing Procedure Calls

What about a string-valued argument?

- Call by reference ⇒ pass a pointer to the start of the string
  - → Works with either length/contents or null-terminated string
- Call by value ⇒ copy the string & pass it
  - → Can store it in caller's AR or callee's AR
  - → Can pass by reference & have callee copy it if necessary …

Pointer of string serves as "descriptor" for the string, stored in the appropriate location (register or slot in the AR)

Implementing Procedure Calls

What about a structure-valued parameter?

- Again, pass a handle
- Call by reference ⇒ descriptor (pointer) refers to original
- Call by value ⇒ create copy & pass its descriptor
  - → Can allocate it in either caller's AR or callee's AR
  - → Can pass by reference & have callee copy it if necessary …

If it is actually an array of structures, then use a dope vector

What About Calls in an OOL (Dispatch)?

In an OOL, most calls are indirect calls

- Compiled code does not contain address of callee
  - → Finds it by indirection through class' method table
  - → Required to make subclass calls find right methods
  - → Code compiled in class *C* cannot know of subclass methods that override methods in *C* and *C* 's superclasses
- In the general case, need dynamic dispatch
  - → Map method name to a search key
  - → Perform a run-time search through hierarchy
    - ♦ Start with object's class, search for 1st occurrence of key
    - ♦ This can be expensive
  - → Use a method cache to speed search
    - ♦ Cache holds < ***key, class, method pointer*** >

How big?
Bigger ⇒ more hits & longer search
Smaller ⇒ fewer hits, faster search

What About Calls in an OOL (Dispatch)?

Improvements are possible in special cases

- If class has no subclasses, can generate direct call
  - → Class structure must be static or class must be **FINAL**
- If class structure is static
  - → Can generate complete method table for each class
  - → Single indirection through class pointer (*1 or 2 operations*)
  - → Keeps overhead at a low level
- If class structure changes infrequently
  - → Build complete method tables at run time
  - → Initialization & any time class structure changes

What About Calls in an OOL (Dispatch)?

Unusual issues in OOL call

- Need to pass receiver's object record as (1st) parameter
  - → Becomes self or this
- Method needs access to its class
  - → Object record has static pointer to superclass, and so on …
- Method is a full-fledged procedure
  - → It still needs an AR …
  - → Can often stack allocate them (*HotSpot does …*)

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 $(x < y)$ then $stmt_1$ else $stmt_2$	*becomes*	cmp_LT $r_x, r_y \Rightarrow r_1$ cbr $r1 \rightarrow$ _$stmt_1$, _$stmt_2$

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      rx, ry ⇒ cc1
      cbr_LT   cc1 → LT, LF
LT:   loadI    1 ⇒ r2
      br       → LE
LF:   loadI    0 ⇒ r2
LE:   ...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	$\rightarrow L_{OUT}$		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	

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

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	$cc_1 \rightarrow L_1, L_2$	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, 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)*

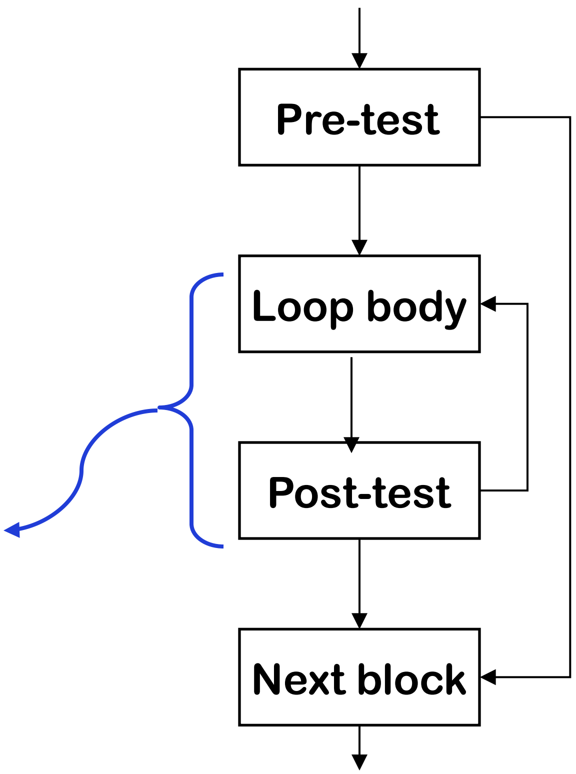
- Frequency of execution
  - → Uneven distribution ⇒ do what it takes to speed common case
- Amount of code in each case
  - → Unequal amounts means predication may waste issue slots
- Control flow inside the construct
  - → Any branching activity within the case base complicates the predicates and makes 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	$1 \Rightarrow r_1$	Initialization
loadI	$1 \Rightarrow r_2$	
loadI	$100 \Rightarrow r_3$	Pre-test
cmp_GE	$r_1, r_3 \Rightarrow r_4$	
cbr	$r_4 \Rightarrow L_2, L_1$	
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

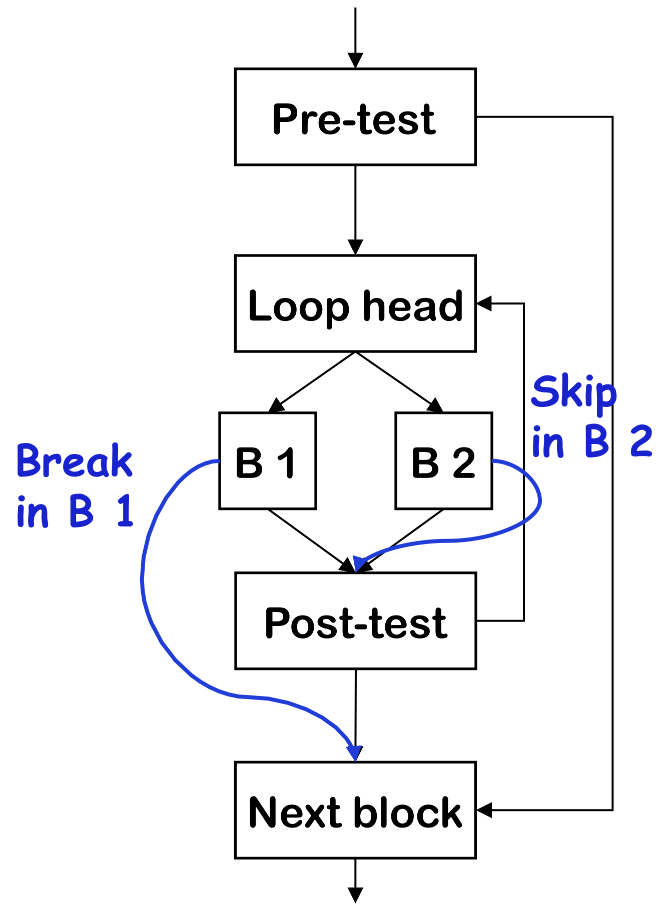
Many modern programming languages include a break

- 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)