Computing an Array Address of an array A[low:high]

A[i]

- @A + (i - low) x sizeof(A[i])
- In general: base(A) + (i - low) x sizeof(A[i])

Color Code:

Invariant

Varying

Depending on how A is declared, @A may be

•an offset from the ARP,

•an offset from some global label, or

•an arbitrary address.

The first two are compile time constants.

Computing an Array Address A[low:high]

where w = sizeof(A[i])

A[i]

- @A + (i - low) x w
- In general: base(A) + (i - low) x w

Almost always a power of 2, known at compile-time
⇒ use a shift for speed

If the compiler knows low it can fold the subtraction into @A

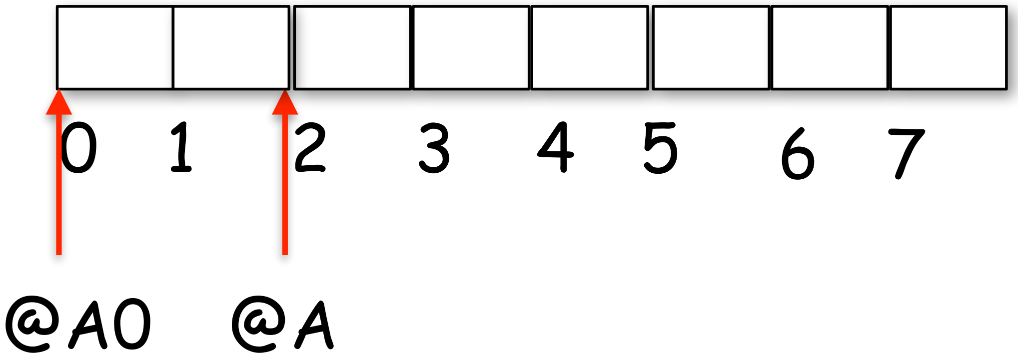
$$A_0 = @A - (low * w)$$

The false zero of A

The False Zero

$$A_0 = @A - (low * w)$$

A[2..7]



computing A[i] with A

$$\begin{array}{lll} loadI & @A & \Rightarrow r_{@A} \\ subI & r_i, 2 & \Rightarrow r_1 \\ lshiftI & r_1, 2 & \Rightarrow r_2 \\ loadA0 & r_{@A}, r_2 & \Rightarrow r_v \end{array}$$

computing A[i] with A0

$$\begin{array}{lll} loadI & @A_0 & \Rightarrow r_{@A_0} \\ lshiftI & r_i, 2 & \Rightarrow r_1 \\ loadA0 & r_{@A_0}, r_1 & \Rightarrow r_v \end{array}$$

How does the compiler handle A[i,j] ?

First, must agree on a storage scheme

Row-major order (most languages)

- Lay out as a sequence of consecutive rows
- Rightmost subscript varies fastest
- A[1,1], A[1,2], A[1,3], A[2,1], A[2,2], A[2,3]

Column-major order (Fortran)

- Lay out as a sequence of columns
- Leftmost subscript varies fastest
- A[1,1], A[2,1], A[1,2], A[2,2], A[1,3], A[2,3]

Indirection vectors (Java)

- Vector of pointers to pointers to … to values
- Takes much more space, trades indirection for arithmetic
- Not amenable to analysis

Laying Out Arrays

The Concept

A

1,1	1,2	1,3	1,4
2,1	2,2	2,3	2,4

These can have distinct & different cache behavior

Row-major order

A

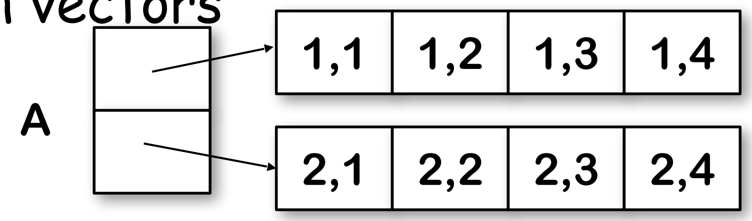
1,1	1,2	1,3	1,4	2,1	2,2	2,3	2,4

Column-major order

A

1,1	2,1	1,2	2,2	1,3	2,3	1,4	2,4

Indirection vectors



Computing an Array Address

where w = sizeof(A[1,1])

A[i]

- $@A + (i - low) \times w$
- In general: base(A) + (i – low) × w

low_1 low_2 $hight_2$

$hight_1$

1,1	1,2	1,3	1,4
2,1	2,2	2,3	2,4

What about $A[i_1,i_2]$?

This stuff looks expensive!
Lots of implicit +, -, x ops

Row-major order, two dimensions

$$@A + ((i_1 - low_1) \times (high_2 - low_2 + 1) + i_2 - low_2) \times w$$

A[2,3] @A+(2-1)x 4+(3-1)

Column-major order, two dimensions

$$@A + ((i_2 - low_2) \times (high_1 - low_1 + 1) + i_1 - low_1) \times w$$

Indirection vectors, two dimensions

$*(A[i_1])[i_2]$ — where $A[i_1]$ is, itself, a 1-d array reference

e.g., $@A + (i_1 - low) \times w$

Optimizing Address Calculation for A[i,j]

In row-major order

$$@A + (i - low_1) \times (high_2 - low_2 + 1) \times w + (j - low_2) \times w$$

	low_1 low_2			$hight_2$
A	1,1	1,2	1,3	1,4
$hight_1$	2,1	2,2	2,3	2,4

Which can be factored into

$$@A + i \times (high_2 - low_2 + 1) \times w + j \times w$$
$$- (low_1 \times (high_2 - low_2 + 1) \times w) - (low_2 \times w)$$

If low_i, $high_i$, and w are known, the last term is a constant

Define $@A_0$ as

$$@A - (low_1 \times (high_2 - low_2 + 1) \times w - low_2 \times w$$

If $@A$ is known, $@A_0$ is a known constant.

And len_2 as $(high_2 - low_2 + 1)$

Then, the address expression becomes

$$@A_0 + (i \times len_2 + j) \times w$$

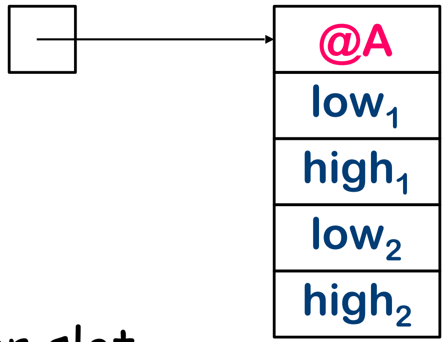
Compile-time constants

Array References

What about arrays as actual parameters?

Whole arrays, as call-by-reference parameters

- Need dimension information ⇒ build 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



Some improvement is possible

- Choose the address polynomial based on the false zero
- Pre-compute the fixed terms in prologue sequence

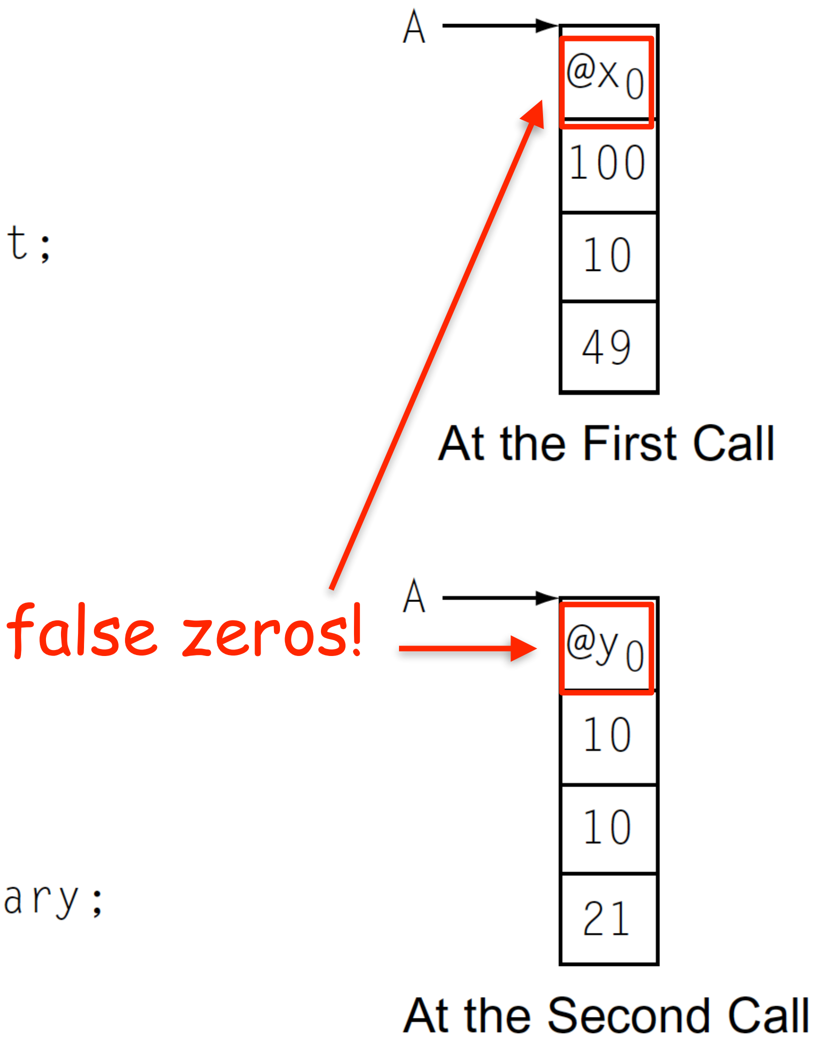
What about call-by-value?

- Most languages pass arrays by reference
- This is a language design issue

The Dope vector

```
program main;
  begin;
    declare x(1:100,1:10,2:50),
        y(1:10,1:10,15:35) float;
    ...
    call fee(x)
    call fee(y);
  end main;

procedure fee(A)
  declare A(*,*,*) float;
  begin;
    declare x float;
      declare i, j, k fixed binary;
      ...
      x = A(i,j,k);
      ...
  end fee;
```



Range checking

A program that refers out-of-the-bound array elements is not well formed.

Some languages like Java requires out-of-the-bound accesses be detected and reported.

In other languages compilers have included mechanisms to detect and report out-of-the-bound accesses.

The easy way is to introduce is to introduce a runtime check that verifies that the index value falls in the array range

Expensive!!

the compiler has to prove
that a given reference cannot
generate an out-of-bounds reference

Information on the bounds in the dope vector

Array Address Calculations

Array address calculations are a major source of overhead

- Scientific applications make extensive use of arrays and array-like structures
  - Computational linear algebra, both dense & sparse
- Non-scientific applications use arrays, too
  - Representations of other data structures
    - → Hash tables, adjacency matrices, tables, structures, …

Array calculations tend iterate over arrays

- Loops execute more often than code outside loops
- Array address calculations inside loops make a huge difference in efficiency of many compiled applications

Reducing array address overhead has been a major focus of optimization since the 1950s.

Example: Array Address Calculations in a Loop

A, B are declared as conformable floating-point arrays

```
DO J = 1, N
    A[I,J] = A[I,J] + B[I,J]
END DO
```

In column-major order

$@A_0 + (j \times len_1 + i) \times w$

number of rows!

Naïve: Perform the address calculation twice

DO J = 1, N

R1 = $@A_0 + (J \times \boxed{len_1 + I) \times w}$

R2 = $@B_0 + (J \times \boxed{len_1 + I) \times w}$

MEM(R1) = MEM(R1) + MEM(R2)

END DO

Example: Array Address Calculations in a Loop

```
DO J = 1, N
      A[I,J] = A[I,J] + B[I,J]
END DO
```

More sophisticated: Move common calculations out of loop

R1 = I x w

c = len_1 x w   ! Compile-time constant

R2 = $@A_0$ + R1

R3 = $@B_0$ + R1

DO J = 1, N

  a = J x c

  R4 = R2 + a

  R5 = R3 + a

  MEM(R4) = MEM(R4) + MEM(R5)

END DO

Loop-invariant code motion

Example: Array Address Calculations in a Loop

```
DO J = 1, N
      A[I,J] = A[I,J] + B[I,J]
END DO
```

Very sophisticated: Convert multiply to add

$R1 = I \times w$

$c = len_1 \times w$! Compile-time constant

$R2 = @A_0 + R1$;

$R3 = @B_0 + R1$;

```
DO J = 1, N
      R2 = R2 + c
      R3 = R3 + c
      MEM(R2) = MEM(R2) + MEM(R3)
END DO
```

J is now bookkeeping

Operator Strength Reduction (§ 10.4.2 in EaC)

Representing and Manipulating Strings

Character strings differ from scalars, arrays, & structures

- Languages support can be different:
  - In C most manipulations takes the form of calls to library routines
  - Other languages provvide first-class mechanism to specify substrings or concatenate them

- Fundamental unit is a character
  - Typical sizes are one or two bytes
  - Target ISA may (or may not) support character-size operations

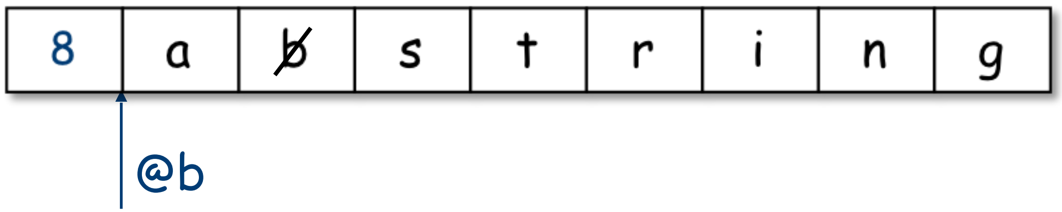
String operation can be costly

- Older CISC architectures provide extensive support for string manipulation
- Modern RISC architectures rely on compiler to code this complex operations using a set a of simpler operations

Representing and Manipulating Strings

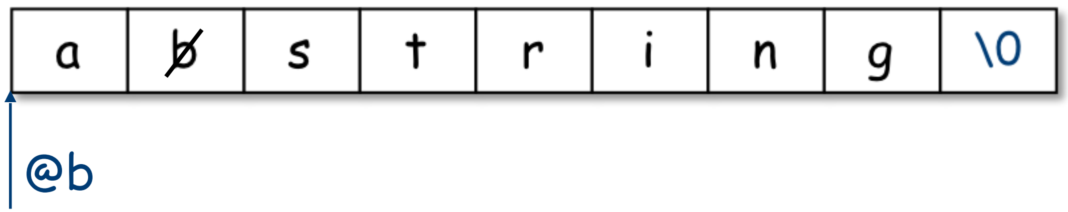
Two common representations of string “a string”

- Explicit length field



Length field may take more space than terminator

- Null termination



- Language design issue

Representing and Manipulating Strings

Each representation as advantages and disadvantages

Operation	Explicit Length	Null Termination
Assignment	Straightforward	Straightforward
Checked Assignment	Checking is easy	Must count length
Length	O(1)	O(n)
Concatenation	Must copy data	Length + copy data

Unfortunately, null termination is almost considered normal

- Hangover from design of C
- Embedded in OS and API designs

Manipulating Strings

Single character assignment

a[1]=b[2]

- With character operations
  - Compute address of rhs, load character
  - Compute address of lhs, store character
- With only word operations (>1 char per word)
  - Compute address of word containing rhs & load it
  - Move character to destination position within word
  - Compute address of word containing lhs & load it
  - Mask out current character & mask in new character
  - Store lhs word back into place

Manipulating Strings

Multiple character assignment

Two strategies

1. Wrap a loop around the single character code, or
2. Work up to a word-aligned case, repeat whole word moves, and handle any partial-word end case

With character operations

With only word operations

Manipulating Strings

Concatenation

- String concatenation is a length computation followed by a pair of whole-string assignments
  - Touches every character
  - There can be length problems!

Manipulating Strings

Length Computation

- Representation determines cost
- Length computation arises in other contexts
  - Whole-string or substring assignment
  - Checked assignment (buffer overflow)
  - Concatenation
  - Evaluating call-by-value actual parameter

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

Some cases works better with the first representation other ones with the second!

Boolean & Relational Expressions

First, we need to recognize boolean & relational expressions

$$
\begin{aligned}
\textit{Expr} &\rightarrow \textit{Expr} \vee \textit{AndTerm} \\
&\mid \textit{AndTerm} \\
\textit{AndTerm} &\rightarrow \textit{AndTerm} \wedge \textit{RelExpr} \\
&\mid \textit{RelExpr} \\
\textit{RelExpr} &\rightarrow \textit{RelExpr} < \textit{NumExpr} \\
&\mid \textit{RelExpr} \le \textit{NumExpr} \\
&\mid \textit{RelExpr} = \textit{NumExpr} \\
&\mid \textit{RelExpr} \ne \textit{NumExpr} \\
&\mid \textit{RelExpr} \ge \textit{NumExpr} \\
&\mid \textit{RelExpr} > \textit{NumExpr}
\end{aligned}
$$

$$
\begin{aligned}
\textit{NumExpr} &\rightarrow \textit{NumExpr} + \textit{Term} \\
&\mid \textit{NumExpr} - \textit{Term} \\
&\mid \textit{Term} \\
\textit{Term} &\rightarrow \textit{Term} \times \textit{Value} \\
&\mid \textit{Term} \div \textit{Value} \\
&\mid \textit{Value} \\
\textit{Value} &\rightarrow \neg\, \textit{Factor} \\
&\mid \textit{Factor} \\
\textit{Factor} &\mid (\ \textit{Expr}\ ) \\
&\mid \text{number}
\end{aligned}
$$

Boolean & Relational Values

Next, we need to represent the values

Numerical representation

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

If the target machine supports boolean operations that compute the boolean result   cmp_LT rx,ry-> r1   r1=True if rx<=ry, r1=False otherwise

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 target machine uses a condition code instead than boolean operations as cmp_LT?

cmp r1,r2 -> cc sets cc with code for LT,LE,EQ,GE,GT,NE

- Must use a conditional branch to interpret result of compare

If the target machine computes a code result of the comparison and we need to store the result of the boolean operation

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 statements ...*			

cbr_LT cc l2,l3 sets
PC=l2 if CC=LT
PC=l3 otherwise

Boolean & Relational Values

The last example actually encoded result in r2

If result is used to control an operation we may not need to write explicitly the result! Positional encoding!

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

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		

Conditional move & predication both simplify this code

Example
if (x < y)
then a ← c + d
else a ← 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	$\Rightarrow r_a$				

i2i_LT cc,r1,r2->r3 copy r1 in r3 if cc matches LT, copy r2 in r3 otherwise

(r1)? add r2,r3 ->r4 the add operation executes if r1 is true

Both versions avoid the branches

Both are shorter than cond'n codes or Boolean compare

Are they equivalent to the initial code? Not always!

Are they better? does code size matter? or execution time?

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	$\Rightarrow r_x$			
L_{OUT}:	nop					

Here, Boolean compare produces much better code

Boolean & Relational Values

Conditional move help here, too

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					

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

Conditional Move		
comp	r_a,r_b	$\Rightarrow CC_1$
i2i_LT	CC_1,r_T,r_F	$\Rightarrow r_1$
comp	r_c,r_d	$\Rightarrow CC_2$
i2i_LT	CC_2,r_T,r_F	$\Rightarrow r_2$
and	r_1,r_2	$\Rightarrow r_x$

i2i_LT cc,r1,r2->r3 copy r1 in r3 if cc matches LT, copy r2 in r3 otherwise

Conditional move is worse than Boolean compare

The bottom line:

$\Rightarrow$ Context & hardware determine the appropriate choice

Control Flow

If-then-else

- Follow model for evaluating relationals & booleans with branches

Branching versus predication

- 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 (lazy 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

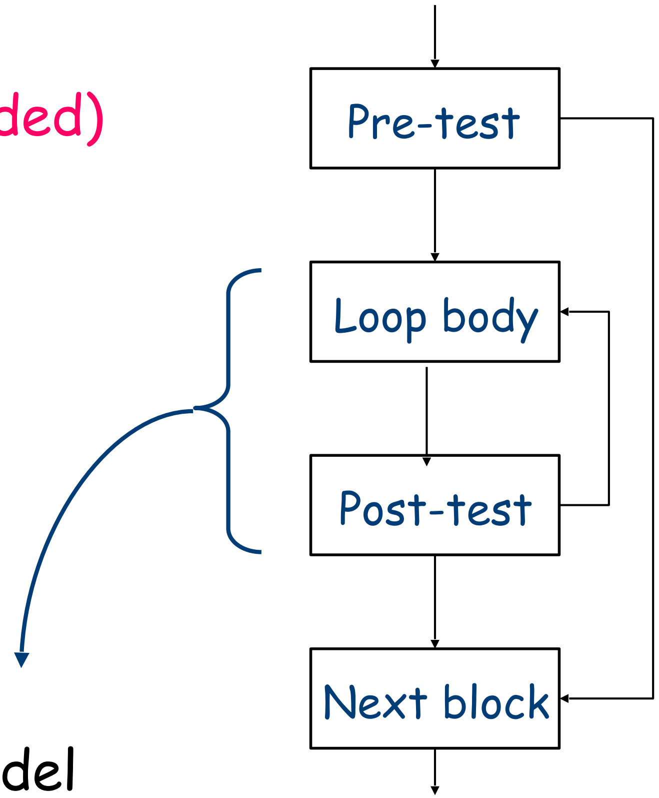
- Modern architectures may favor evaluating full expression
  - Rising branch latencies make the short-circuit path expensive
  - Conditional move and predication may make full path cheaper

- 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)



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

Implementing Loops

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

	loadI	1	$\Rightarrow r_1$	Initialization
	loadI	1	$\Rightarrow r_2$	
	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	$\Rightarrow r_1$	Post-test
	cmp_LT	r_1,r_3	$\Rightarrow r_5$	
	cbr	r_5	$\rightarrow L_1,L_2$	
L_2:	*next statement*			

Case (switch) 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:

need an efficient method to locate the designated code

many compilers provvide several different search schemas each one can be better in some cases.

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 address (requires dense case set)

Linear Search

```
switch (e1)  {
   case 0:   block0;
             break;
   case 1:   block1;
             break;
   case 3:   block3;
             break;
   default:  blockd;
             break;
}
```

Switch Statement

```
t1 ← e1
if (t1 = 0)
   then block0
   else if (t1 = 1)
      then block1
      else if (t1 = 2)
         then block2
         else if (t1 = 3)
            then block3
            else blockd
```

Implementing as a Linear Search

Binary Search

```
switch (e1)  {
  case 0:   block0
            break;
  case 15:  block15
            break;
  case 23:  block23
            break;
  ...
  case 99:  block99
            break;
  default:  blockd
            break;
}
```

Switch Statement

Value	Label
0	LB_0
15	LB_{15}
23	LB_{23}
37	LB_{37}
41	LB_{41}
50	LB_{50}
68	LB_{68}
72	LB_{72}
83	LB_{83}
99	LB_{99}

Search Table

```
t1 ← e1
down ← 0    // lower bound
up ← 10     // upper bound + 1
while (down + 1 < up)  {
   middle ← (up + down) ÷ 2
   if (Value [middle] ≤ t1)
      then down ← middle
      else up ← middle
}
if (Value [down] = t1)
   then jump to Label[down]
   else jump to LBd
```

Code for Binary Search

Direct Address Computation

- requires dense case set

```
switch (e1) {
   case 0:  block0
            break;
   case 1:  block1
            break;
   case 2:  block2
            break;
   ...
   case 9:  block9
            break;
   default: blockd
            break;
}
```

Switch Statement

Label
LB_0
LB_1
LB_2
LB_3
LB_4
LB_5
LB_6
LB_7
LB_8
LB_9

Jump Table

```
t1 ← e1
if (0 > t1 or t1 > 9)
   then jump to LBd
   else
      t2 ←@Table + t1 x 4
      t3 ← memory(t2)
      jump to t3
```

Code for Address Computation