Code Shape II
Booleans, Relationals, & Control flow

Handling Assignment (just another operator)

$$lhs \leftarrow rhs$$

Strategy

- Evaluate *rhs* to a value (an rvalue)
- Evaluate *lhs* to a location (an lvalue)
  - → *lvalue* is a register ⇒ move rhs
  - → *lvalue* is an address ⇒ store rhs
- If *rvalue* & *lvalue* have different types
  - → Evaluate *rvalue* to its "*natural*" type
  - → Convert that value to the type of **lvalue*

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

Column-major order (Fortran)

Lay out as a sequence of columns

Indirection vectors (Java)

Vector of pointers to pointers to … to values

Takes more space, trades indirection for arithmetic

Hard to analyze

Laying Out Arrays : Row-major Order

The Concept

A

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

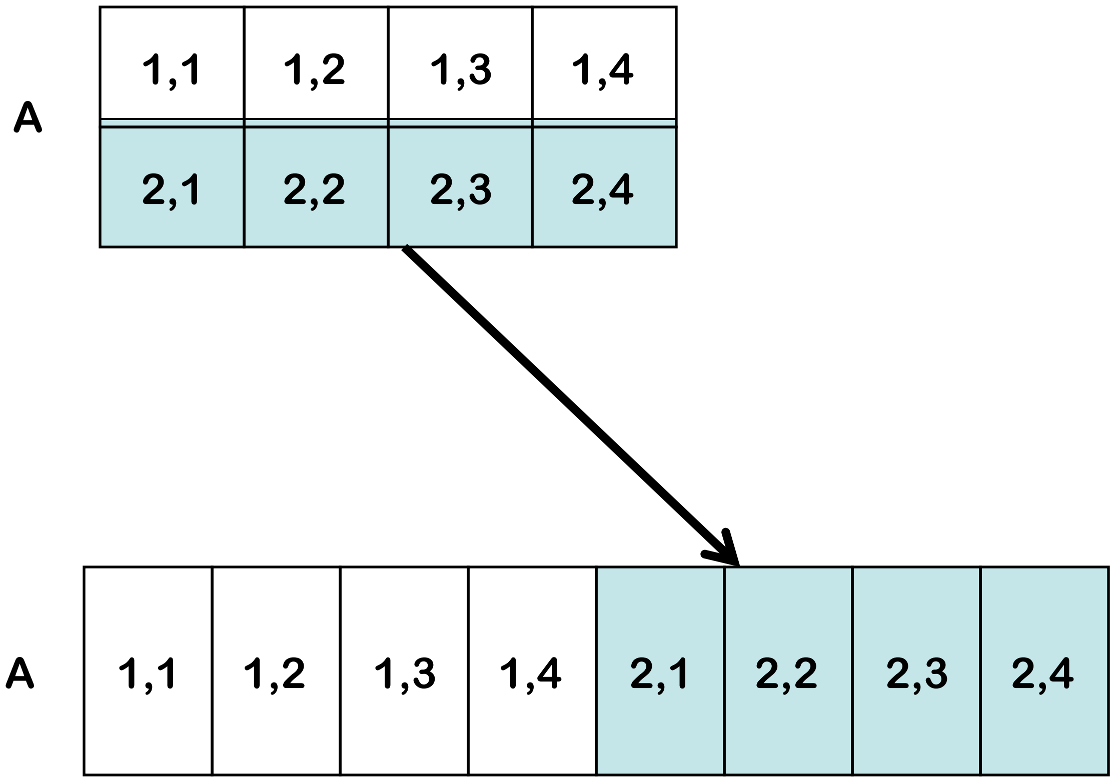
A

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

Row-major Order

Laying Out Arrays : *Row-major Order*

The Concept



Row-major Order

Laying Out Arrays : *Column-major order*

The Concept

A

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

A

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

Column-major Order

Laying Out Arrays : *Column-major order*

The Concept

A

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

A

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

Column-major Order

Laying Out Arrays : *Column-major order*

The Concept

A

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

A

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

Column-major Order

Laying Out Arrays : *Column-major order*

The Concept

A

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

A

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

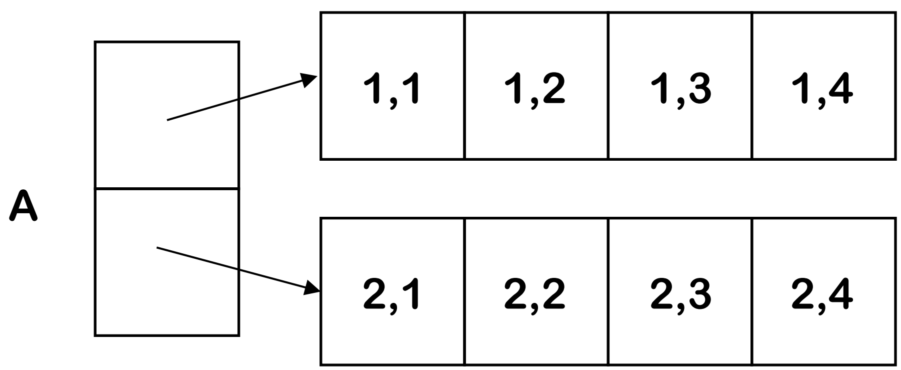
Column-major Order

Laying Out Arrays

The Concept

A

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



Indirection vectors

Computing an Array Address

A[i]

$$@A + (i - low) \times \text{sizeof}(A[1])$$

- In general:

$$\text{base}(A) + (i - low) \times \text{sizeof}(A[1])$$

Computing an Array Address

A[i]

$$@A + (i - low) \times sizeof(A[1])$$

- In general:

$$base(A) + \underline{(i - low)} \times sizeof(A[1])$$

int A[1:10] ⇒ low is 1
Make low 0 for faster access
(saves a sub operation)

Computing an Array Address

A[i]

$$@A + (\, i - low \,) \times \text{sizeof}(A[1])$$

- In general:

$$\text{base}(A) + \underline{(\, i - low \,)} \times \underline{\text{sizeof}(A[1])}$$

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

Computing an Array Address

What about $A[i_1,i_2]$?

Row-major order, two dimensions

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

$$@A + (\boxed{(i1 - low1)} \times (high1 - low1 + 1) + i2 - low2) \times sizeof(A[1])$$

A[2,3]

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

(i1 - low1)

What row do we want?

Computing an Array Address

What about $A[i_1,i_2]$?

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

Row-major order, two dimensions

$$@A + ((i1 - low1) \times \boxed{(high1 - low1 + 1)} + i2 - low2) \times sizeof(A[1])$$

A[2,3]

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

$$\underbrace{\text{1,1 \quad 1,2 \quad 1,3 \quad 1,4}}_{high - low + 1}$$

high - low + 1

4 - 1 + 1 = 4

Computing an Array Address

What about $A[i_1,i_2]$?

Row-major order, two dimensions

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

$$@A + ((i1 - low1) \times (high1 - low1 + 1) + \boxed{i2 - low2}) \times sizeof(A[1])$$

What column do we want?

A[2,3]

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

Indirection vectors, two dimensions

$$*(A[i_1])[i_2]$$

where $A[i_1]$ is a 1-d array refs

Boolean & Relational Values

Numerical representation

- Assign values to TRUE and FALSE
- Use hardware AND, OR, and NOT operations

Consider $b \lor c \land \boxed{\neg d}$

not rd ⇒ r1

and rc, r1 ⇒ r2

or rb, r2 ⇒ r3

Boolean & Relational Values

Numerical representation

- Assign values to TRUE and FALSE
- Use hardware AND, OR, and NOT operations

Consider $b \lor \boxed{c \land \neg d}$

not rd ⇒ r1

and rc, r1 ⇒ r2

or rb, r2 ⇒ r3

Boolean & Relational Values

Numerical representation

- Assign values to TRUE and FALSE
- Use hardware AND, OR, and NOT operations

Consider $\boxed{b \lor c \land \neg d}$

not rd ⇒ r1

and rc, r1 ⇒ r2

or rb, r2 ⇒ r3

Boolean & Relational Values

Numerical representation

- Use comparison to get a boolean from a relational expression

Example

x < y ***becomes*** **cmp_LT** $r_x, r_y \Rightarrow r_1$

cmp_LT = compare operation less than

Boolean & Relational Values

Numerical representation

- Use comparison to get a boolean from a relational expression

Example

```
if  (x < y)
  then stmt₁
  else stmt₂
```

becomes

cmp_LT $r_x, r_y \Rightarrow r_1$

cbr $r1 \rightarrow$ _$stmt_1$, _$stmt_2$

Source code: if $(x < y)$ then $stmt_1$ else $stmt_2$

cbr = conditional branch

Control Flow

If-then-else

- Follow model for evaluating relationals & booleans with branches

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

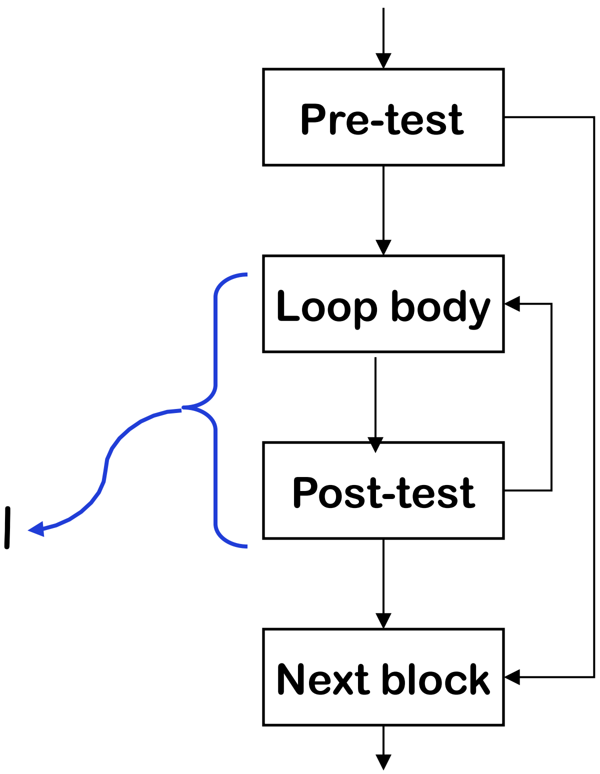
- Frequency of execution
  - → Uneven distribution ⇒ do what it takes to speed common case
- Amount of code in each case
  - → Unequal amounts means predication may waste issue slots
- Control flow inside the construct
  - → Any branching activity within the case base complicates the predicates and makes 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 ⇒ r1
     loadI    1 ⇒ r2
     loadI    100 ⇒ r3
     cmp_GE   r1, r3 ⇒ r4
     cbr      r4 ⇒ L2, L1
L1:  body
     add      r1, r2 ⇒ r1
     cmp_LT   r1, r3 ⇒ r5
     cbr      r5 ⇒ L1, L2
L2:  next statement
```

Initialization

Pre-test

Post-test

BACKUP SLIDES

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

What if the ISA uses a condition code?

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

Example:

x < y *becomes*

	$\boxed{\textbf{cmp} \quad r_x, r_y \Rightarrow cc_1}$	
	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…	

Boolean & Relational Values

What if the ISA uses a condition code?

- 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...
```

Boolean & Relational Values

What if the ISA uses a condition code?

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

Example:

			cmp	$r_x, r_y \Rightarrow cc_1$
			cbr_LT	$cc_1 \rightarrow L_T, L_F$
x < y	***becomes***	L_T:	loadI	$1 \Rightarrow r_2$
			br	$\rightarrow L_E$
		L_F:	loadI	$0 \Rightarrow r_2$
		L_E:	...other stmts...	

Boolean & Relational Values

What if the ISA uses a condition code?

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

Example:

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

Boolean & Relational Values

Conditional move & predication

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_<	$cc_1, r_1, r_2 \Rightarrow r_a$			