Handling Assignment *(just another operator)*

lhs ← *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*

Unambiguous scalars go into registers

Ambiguous scalars or aggregates go into memory

Let hardware sort out the addresses !

Handling Assignment

What if the compiler cannot determine the type of the rhs?

- This is a property of the language & the specific program
- If type-safety is desired, compiler must insert run-time checks
- Add a *tag* field to the data items to hold type information

Code for assignment becomes more complex

```
evaluate rhs
if type(lhs) ≠ rhs.tag
   then
      convert rhs to type(lhs)  or
      signal a run-time error
lhs ← rhs
```

Handling Assignment

Compile-time type-checking

- Goal is to eliminate both the check & the tag
- Determine, at compile time, the type of each subexpression
- Use compile-time types to determine whether check is needed

Optimization strategy

- If compiler knows the type, move the check to compile-time
- If tags are not needed for garbage collection, eliminate the tags
- If check is unavoidable, try to overlap it with other computation

Can design the language so all checks are static

Handling Assignment *(with reference counting)*

The problem with reference counting

- Must adjust the count on each pointer assignment
- Overhead is significant, relative to assignment

Code for assignment becomes

```
evaluate rhs
lhs→count ← lhs →count - 1
lhs ← addr(rhs)
rhs →count ← rhs →count + 1
```

Plus a check for zero at the end

This adds *1 +, 1 -, 2 loads, & 2 stores*

With extra functional units & large caches, this may become free …

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
- Righmost 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

Row-major order

A

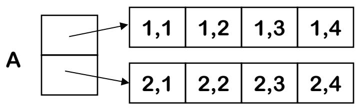
1,1	1,2	1,3	1,4	2,1	2,2	2,3	2,4

Column-major order

A

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

Indirection vectors



These have distinct & different cache behavior

Computing an Array Address

A[i]

- @A + (i – low) x sizeof(A[1])
- In general: base(A) + (i – low) x sizeof(A[1])

int A[1:10] ⇒ low is 1
Make low 0 for faster
access (save a –)

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) x ($high_2$ – low_2 + 1) + i_2 – low_2) x sizeof(A[1])

Column-major order, two dimensions

@A + ((i_2 – low_2) x ($high_1$ – low_1 + 1) + i_1 – low_1) x sizeof(A[1])

Indirection vectors, two dimensions

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

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

Optimizing Address Calculation for A[i][j]

In row-major order

@A + (i–low_1)($high_2$–low_2+1) x w + (j – low_2) x sizeof(A[1][1])

Which can be factored into

@A + i x ($high_2$–low_2+1) x sizeof(A[1][1]) + j x sizeof(A[1][1])
– (low_1 x ($high_2$–low_2+1) x sizeof(A[1][1] + low_2 x sizeof(A[1][1]))

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

Define $@A_0$ as

@A – (low_1 x ($high_2$–low_2+1) x sizeof(A[1][1]) + low_2 x sizeof(A[1][1]))

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

Then, the address expression becomes

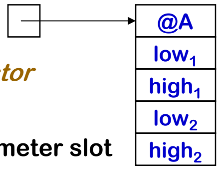
$@A_0$ + (i x len_2 + j) x sizeof(A[1][1])

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

- **Save len_i and low_i rather than low_i and $high_i$**
- **Pre-compute the fixed terms in prolog sequence** (a win if used)

What about call-by-value?

- **Most c-b-v languages pass arrays by reference**
- **This is a language design issue**

Array References

What about A[12] as an actual parameter?

If corresponding parameter is a scalar, it's easy

- **Pass the address or value, as needed**
- **Must know about both formal & actual parameter**
- **Language definition must force this interpretation**

What is corresponding parameter is an array?

- **Must know about both formal & actual parameter**
- **Meaning must be well-defined and understood**
- **Cross-procedural checking of conformability**

⇒ Again, we're treading on language design issues

Array References

What about variable-sized arrays?

Local arrays dimensioned by actual parameters

- Same set of problems as parameter arrays
- Requires dope vectors (or equivalent)
  - dope vector at fixed offset in activation record

  ⇒ Different access costs for textually similar references

This presents a lot of opportunity for a good optimizer

- Common subexpressions in the address polynomial
- Contents of dope vector are fixed during each activation
- Should be able to recover much of the lost ground

⇒ Handle them like parameter arrays

Example: Array Address Calculations in a Loop

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

- Naïve: Perform the address calculation twice

```
DO J = 1, N
   R1 = @A0 + (J x len1 + I ) x floatsize
   R2 = @B0 + (J x len1 + I ) x floatsize
   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
```

- Sophisticated: Move comon calculations out of loop

```
R1 = I x floatsize
c = $len_1$ x floatsize  ! 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
```

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 (Strength Reduction)

```
R1 = I x floatsize
c = $len_1$ x floatsize  ! 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
```

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***

$$
\begin{array}{lll}
 & \text{comp} & r_x, \\
 & r_y \Rightarrow cc_1 & \\
 & \text{cbr\_LT} & cc_1 \rightarrow L_T, L_F \\
L_T: & \text{loadI} & 1 \Rightarrow r_2 \\
 & \text{br} & \rightarrow L_E \\
L_F: & \text{loadI} & 0 \Rightarrow r_2 \\
L_E: & \text{...other stmts...} &
\end{array}
$$

This “positional representation” is much more complex

Boolean & Relational Values

What if the ISA uses a condition code? **(KDC: a seductive, evil idea)**

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

Example:

x < y ***becomes***

$$
\begin{array}{lll}
 & \text{cmp} & r_x, r_y \Rightarrow cc_1 \\
 & \text{cbr\_LT} & cc_1 \rightarrow L_T, L_F \\
L_T: & \text{loadI} & 1 \Rightarrow r_2 \\
 & \text{br} & \rightarrow L_E \\
L_F: & \text{loadI} & 0 \Rightarrow r_2 \\
L_E: & \text{...other stmts...} &
\end{array}
$$

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

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	$cc_1 \rightarrow L_1,L_2$	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

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

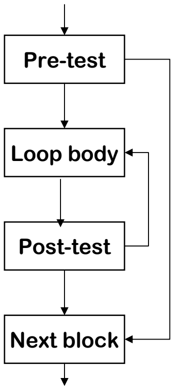
If-then-else

- Follow model for evaluating relationals & booleans with branches

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


Pre-test
Loop body
Post-test
Next block

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

Loop Implementation Code

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

loadI $1 \Rightarrow r_1$
loadI $1 \Rightarrow r_2$
loadI $100 \Rightarrow r_3$
cmp_GT $r_1, r_3 \Rightarrow r_4$
cbr $r_4 \Rightarrow L_2, L_1$
L_1: *body*
add $r_1, r_2 \Rightarrow r_1$
cmp_LE $r_1, r_3 \Rightarrow r_5$
cbr $r_5 \Rightarrow L_1, L_2$
L_2: *next statement*

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

Surprisingly many compilers do this for all cases!

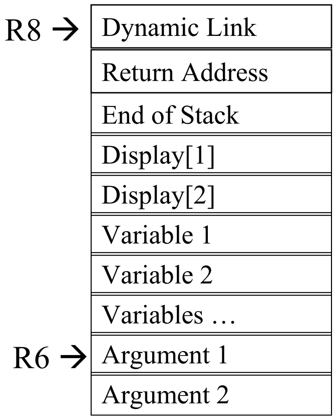
Strategies

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

Activation record structure

- R9 Link register
- R8 Activation Record ptr
- R7 Temporary Act Rec ptr
- R6 End of stack
- R2-R5 Four arithmetic Regs
- R0-R1 Mult-Div registers

- Registers 2-5 used for + and –
- Multiply code pattern:
  L R0, arg1
  M R0, arg2
  LI Ri, (R1)0 where Ri = allocated reg.



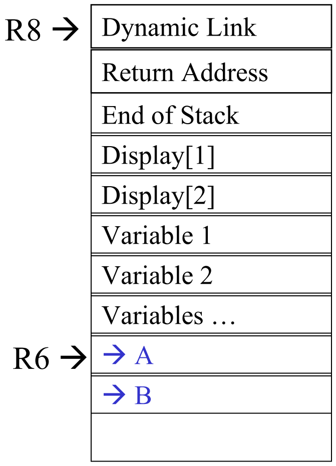
R8 → Dynamic Link
Return Address
End of Stack
Display[1]
Display[2]
Variable 1
Variable 2
Variables …
R6 → Argument 1
Argument 2

Assume a procedure at level 3 calls a function with 2 arguments at static level 3:
X(A,B)

Activation record structure

- **Function call code: X(A,B) Code:**

```
LI    R0, A        Get first arg address
ST    R0, (R6)0    Save address in stack
LI    R0, B        Get second arg address
ST    R0, (R6)1    Save address in stack
JSRI  R9,X         Goto Function R9=retn addr
ST    R0,X         Save function value in Stack
```

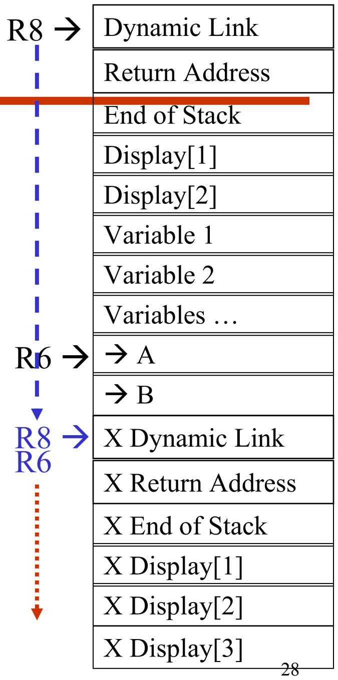


R8 →
Dynamic Link
Return Address
End of Stack
Display[1]
Display[2]
Variable 1
Variable 2
Variables …
R6 →
→ A
→ B

Activation record structure

- **Function X(integer: I, J) Prolog Code:**

```
ST   R8, (R6)2        New Dyn Link
ST   R9, (R6)3        New Retn Addr
L    R0, (R6)0        Get arg 1
ST   R0, (R6)I+2      Store in I in new AR
L    R0, (R6)1        Get arg 2
ST   R0, (R6)J+2      Store in J in new AR
AI   R6, 2            New AR start
L    R0, (R8)3        Calling Disp[1]
ST   R0, (R6)3        New Disp[1]
L    R0, (R8)4        Calling Disp[2]
ST   R0, (R6)4        New Disp[2]
ST   R6, (R6)5        New Disp[3]
LI   R8, (R6)0        Set new Act Rec Ptr
AI   R6,sizeof(X)     Set new end stack
ST   R6, (R8)2        Save end stack
```

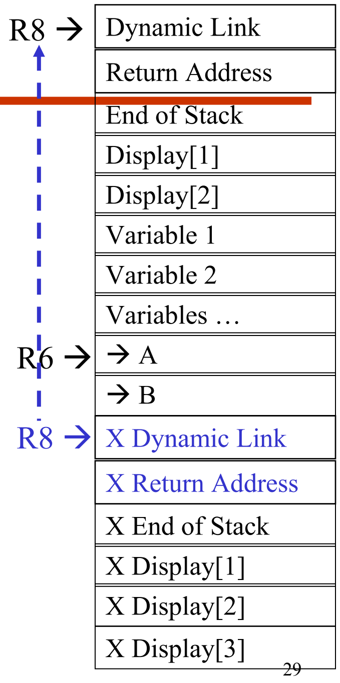


R8 →
Dynamic Link
Return Address
End of Stack
Display[1]
Display[2]
Variable 1
Variable 2
Variables …
R6 →
→ A
→ B
R8 →
R6
X Dynamic Link
X Return Address
X End of Stack
X Display[1]
X Display[2]
X Display[3]

Activation record structure

- **Function X(A,B) Epilog Code:**

```
L    R0, fcn-value       Set value of X
L    R9, (R8)1Return Address
L    R8, (R8)0Load Dyn Link
L    R6, (R8)2Reset end of stack
JI   0, (R9)0            Return
```



R8 → Dynamic Link
Return Address
End of Stack
Display[1]
Display[2]
Variable 1
Variable 2
Variables …
R6 → → A
→ B
R8 → X Dynamic Link
X Return Address
X End of Stack
X Display[1]
X Display[2]
X Display[3]