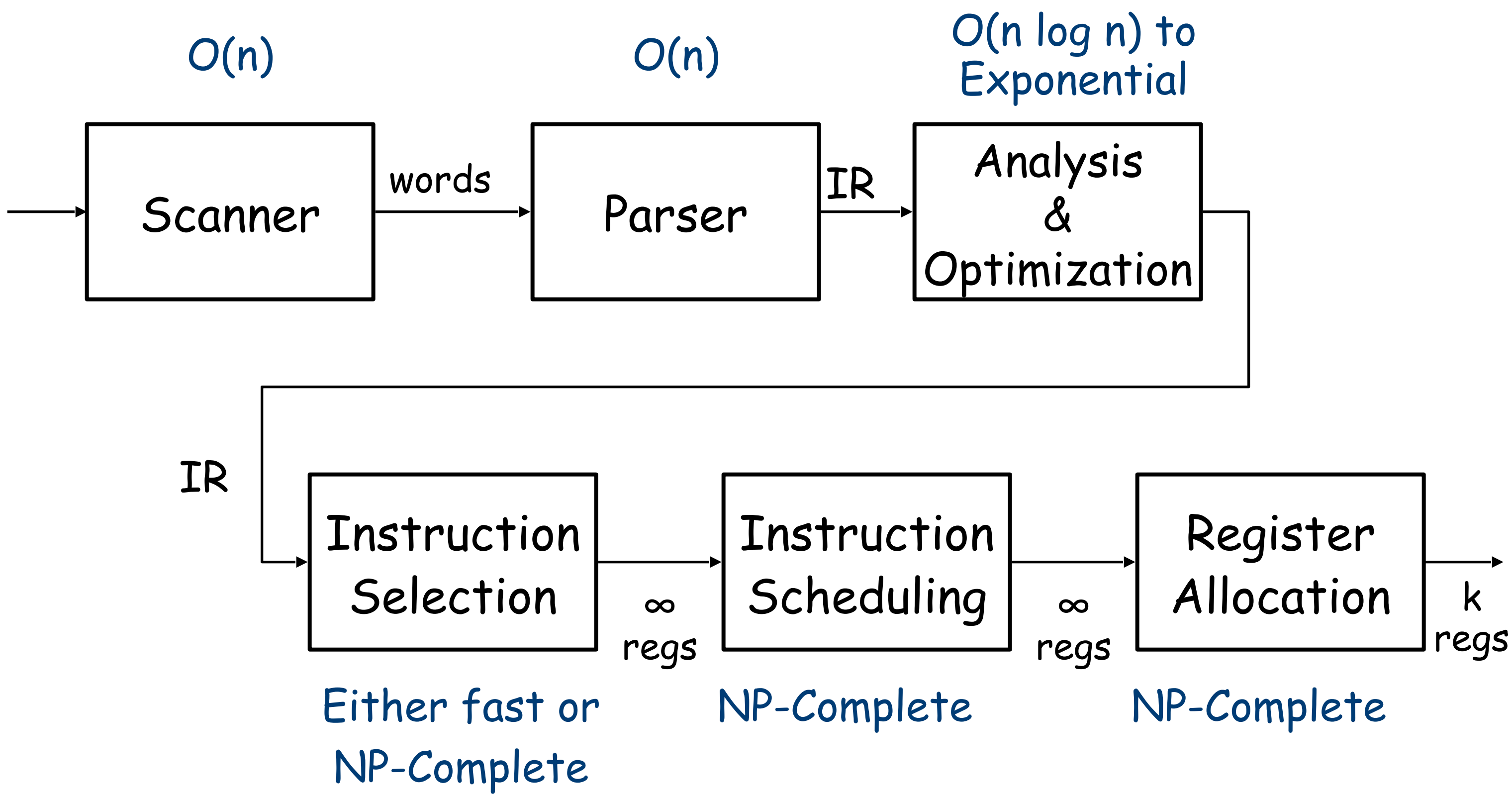
# Introduction to Code Generation

# Structure of a Compiler

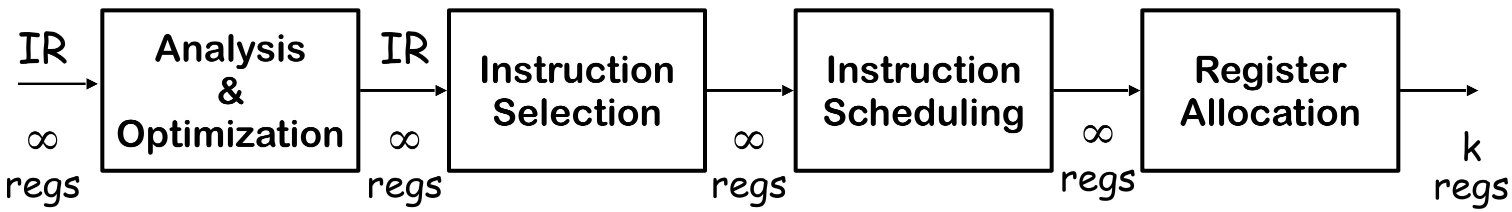



A compiler is a lot of fast stuff followed by some hard problems

- The hard stuff is mostly in code generation and optimization
- For multicores, we need to manage parallelism & sharing
- For unicore performance, allocation & scheduling are critical

# Structure of a Compiler

We assume the following model




- Selection can be fairly simple (problem of the 1980s)
- Allocation & scheduling are complex
- Operation placement is not yet critical

    we assumed a unified register set

# What about the IR ?

- Low-level, R**ISC**-like IR such as I**LOC**
- Has “enough” registers
- I**LOC** was designed for this stuff with:
    - Branches, compares, & labels
    - Memory tags
    - Hierarchy of loads & stores
    - Provision for multiple ops/cycle

# Analysis & Optimization

- The translation of the front end was obtained by considering the statements one of the time as they were encountered
- This initial IR contains general implementation strategies that will work in any surrounding context
- At run time the code will be executed in a more constrained and predictable context
- The optimizer analyses the IR form of the code to discover facts about the context and use them to rewrite (transform) the code so that it will compute the same answer in a more efficient way

# The Back End

The compiler back end traverses the IR form and emits the code for the target machine

- It selects target-machine operations to implement each IR operation (Instruction selection)
- It chooses an order in which the operations will execute efficiently (Instruction scheduling)
- It will decide which values will reside in registers and which in memory (Register allocation)

# Memory Models

Two major models

- Register-to-register model
  - Keep all values that can legally be stored in a register in registers
  - Ignore machine limitations on number of registers
  - Compiler back-end must insert loads and stores

use virtual registers!

- Memory-to-memory model
  - Keep all values in memory
  - Only promote values to registers directly before they are used
  - Compiler back-end can remove loads and stores
- Compilers for RISC machines usually use register-to-register
  - Easier to determine when registers are used

# Definitions

## Instruction selection

- Mapping IR into assembly code
- Assumes a fixed memory model & code shape
- Combining operations, using address modes (instr. reg+offset or reg to reg mode)

## Instruction scheduling

- Reordering operations to hide latencies
- Assumes a fixed program (set of operations)
- Changes demand for registers

## Register allocation

- Deciding which values will reside in registers
- Changes the storage mapping, may add false sharing
- Concerns about placement of data & memory operations

These 3 problems are tightly coupled and need static analysis

# Code Shape (Chapter 7)

## Definition

- The compiler must choose among many alternative ways to implement each construct on a given processor
- Those choices have a strong and direct impact on the quality of the final produced code
- Code shape is the end product of many decisions (big & small)

## Impact

- Code shape has a strong impact on the behaviour of the compiled code and on the ability of the optimizer and back end to improve it
- Code shape can encode important facts, or hide them

# Code Shape

## Example -- the case statement on a character value

- Implement it as cascaded if-then-else statements
  - Cost depends on where your case actually occurs
  - O(256)
- Implement it as a binary search
  - Need a dense set of conditions to search
  - Uniform (log 256) cost
- Implement it as a jump table
  - Lookup address in a table & jump to it
  - We trade data space for speed
  - Uniform (constant) cost

Performance depends on order of cases!

All these are legal (and reasonable) implementations of the switch statement

# Which implementation for switch?

The one that is the best for a particular switch statement depends on many factors such as:

-The number of cases and their relative executions frequencies
-The knowledge of the cost structure for branching on the processor

Even when the compiler does not have enough information to choose it must choose an implementation strategy

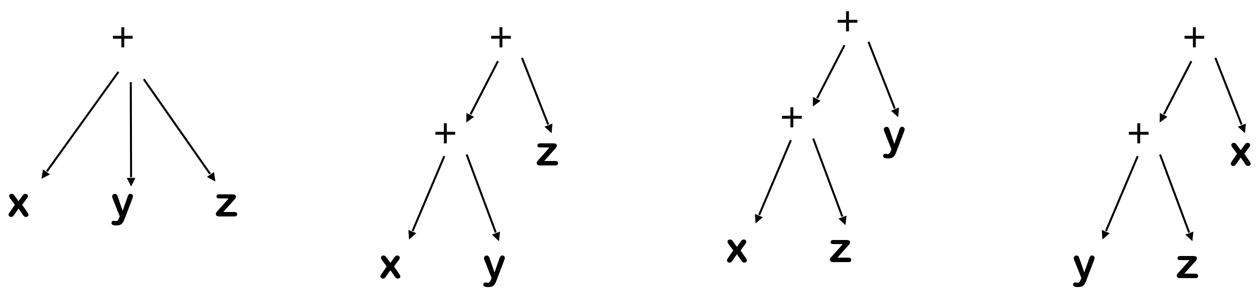
No amount of massaging or transforming will convert one into another

# Code Shape: the ternary operation x+y+z

## Several ways to implement x+y+z

Addition is commutative & associative for integers

| $x + y + z$ | $x + y \rightarrow t1$ | $x + z \rightarrow t1$ | $y + z \rightarrow t1$ |
|---|---|---|---|
| | $t1 + z \rightarrow t2$ | $t1 + y \rightarrow t2$ | $t1 + x \rightarrow t2$ |




- What if the compiler knows that x is constant 2 and z is 3?

The compiler should detect 2+3 evaluates and fold it into the code

- What if y+z is evaluated earlier?

The "best" shape for x+y+z depends on contextual knowledge

– There may be several conflicting options

# Code Shape

Why worry about code shape? Can't we just trust the optimizer and the back end?

- Optimizer and back end approximate the answers to many hard problems
- The compiler's individual passes must run quickly
- It often pays to encode useful information into the IR
  - Shape of an expression or a control structure
  - A value kept in a register rather than in memory
- Deriving such information may be expensive, when possible
- Recording it explicitly in the IR is often easier and cheaper

# How to generate ILOC code

- The three-address form lets the compiler name the result of any operation and preserve it for later reuse

- It uses always new register and leave to the allocator the duty of reduce them

- To generate code for a trivial expression a+b the compiler emits code to ensure that the values of a and b are in registers

- If a is stored in memory at offset $@a$ in the current Activation Record (AR), the code is

$$\begin{array}{lll} loadI & @a & \Rightarrow r_1 \\ loadA0 & r_{arp}, r_1 & \Rightarrow r_a \end{array}$$

# Generating Code for Expressions

## The idea

- Assume an AST as input and ILOC as output
- Use a postorder treewalk evaluator
  - Visits & evaluates children
  - Emits code for the op itself
  - Returns register with result
- Bury complexity of addressing names in routines that it calls
  - base(), offset() and val()
- Works for simple expressions
- Easily extended to other operators

the node of the AST

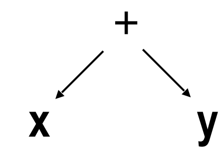
```
expr(node) {
  register result, t1, t2;
  switch (type(node)) {
      case ×,÷,+,– :
          t1← expr(left child(node));
          t2← expr(right child(node));
          result ← NextRegister();
          emit (op(node), t1, t2, result);
          break;
      case IDENTIFIER:
          t1← base(node);
          t2 ← NextRegister();
          emit (loadI, offset(node), none, t2);
          result ← NextRegister();
          emit (loadAO, t1, t2, result);
          break;
      case NUMBER:
          result ← NextRegister();
          emit (loadI, val(node), none, result);
          break;
  }
  return result;
}
```

# Generating Code for Expressions (a naive translation)

```
expr(node) {
  register result, t1, t2;
  switch (type(node)) {
      case ×,÷,+,– :
          t1← expr(left child(node));
          t2← expr(right child(node));
          result ← NextRegister();
          emit (op(node), t1, t2, result);
          break;
      case IDENTIFIER:
          t1← base(node);
           t2 ← NextRegister();
          emit (loadI, offset(node), none, t2);
          result ← NextRegister();
          emit (loadAO, t1, t2, result);
          break;
      case NUMBER:
          result ← NextRegister();
          emit (loadI, val(node), none, result);
          break;
    }
    return result;
}
```

base(id) loads the right pointer to the AR where id is defined in register $r_{arp}$

Example:




Produces for register counter 0 :

espr("x"):

NextRegister(): r1    loadI @x    -> r1

NextRegister(): r2    loadAO    $r_{arp}$, r1    -> r2

espr("y"):

NextRegister(): r3    loadI @y    -> r3

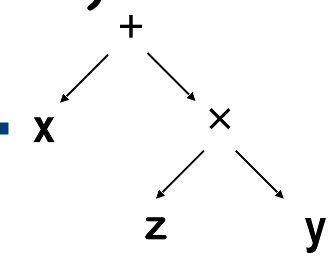
NextRegister(): r4    loadAO    $r_{arp}$, r3    -> r4

NextRegister() : r5

Emit(add, r2,r4,r5) :

add    r2, r4    -> r5

# Generating Code for Expressions (a naive translation)




```
expr(node) {
  register result, t1, t2;
  switch (type(node)) {
      case ×,÷,+,– :
          t1← expr(left child(node));
          t2← expr(right child(node));
          result ← NextRegister();
          emit (op(node), t1, t2, result);
          break;
      case IDENTIFIER:
           t1← base(node);
           t2 ← NextRegister();
          emit (loadI, offset(node), none, t2);
          result ← NextRegister();
          emit (loadAO, t1, t2, result);
          break;
      case NUMBER:
          result ← NextRegister();
          emit (loadI, val(node), none, result);
          break;
       }
       return result;
 }
```

Produces for register counter 0 :

espr("x"):

NextRegister():r1, loadI @x -> r1

NextRegister():r2 loadAO rarp, r1 -> r2

espr("z"):

NextRegister():r3 loadI @z -> r3

NextRegister():r4 loadAO rarp, r3 -> r4

espr("y"):

NextRegister():r5 loadI @y -> r5

NextRegister():r6 loadAO rarp, r5 -> r6

NextRegister():r7

Emit(mul, r4,r6,r7) :

mult r4, r6 -> r7

NextRegister():r8

Emit(add, r2,r7,r8) :

add r2, r7 -> r8

# Effects of code shape on the demand of registers

- Code shape decisions encoded into the tree walk code generator have an effect on the demand of registers

- The previous naive code uses 8 registers + $r_{arp}$

- The register allocator (later in compilation) can reduce the demand for register to 3 + $r_{arp}$

```
loadI   @x          -> r1
loadAO  rarp, r1    -> r1
loadI   @z          -> r2
loadAO  rarp, r2    -> r2
loadI   @y          -> r3
loadAO  rarp, r3    -> r3
mult    r2,   r3    -> r2
add     r1,   r2    -> r2
```

# The best solution: alternate right and left children

## evaluating z×y first

$$
\begin{array}{lll}
load & @z & \Rightarrow r_1 \\
loadA0 & r_{arp}, r_1 & \Rightarrow r_2 \\
load & @y & \Rightarrow r_3 \\
loadA0 & r_{arp}, r_3 & \Rightarrow r_4 \\
mult & r_2, r_4 & \Rightarrow r_5 \\
load & @x & \Rightarrow r_6 \\
loadA0 & r_{arp}, r_6 & \Rightarrow r_7 \\
add & r_7, r_5 & \Rightarrow r_8
\end{array}
$$

## after register allocation

$$
\begin{array}{lll}
load & @z & \Rightarrow r_1 \\
loadA0 & r_{arp}, r_1 & \Rightarrow r_1 \\
load & @y & \Rightarrow r_2 \\
loadA0 & r_{arp}, r_2 & \Rightarrow r_2 \\
mult & r_1, r_2 & \Rightarrow r_1 \\
load & @x & \Rightarrow r_2 \\
loadA0 & r_{arp}, r_2 & \Rightarrow r_2 \\
add & r_2, r_1 & \Rightarrow r_1
\end{array}
$$

General rule: evaluate first the
child that has more demand for registers

Code shape!

# Some observations

What if our IDENTIFIER is

- already in a register?
- in a global data area?
- a parameter value?
  - call by value
  - call by reference

# Extending the Simple Treewalk Algorithm

It assumes a single case for id, more cases for IDENTIFIER

- What about values that reside in registers?
  - Modify the IDENTIFIER case
    - Already in a register ⇒ return the register name
    - Not in a register ⇒ load it as before, but record the fact
  - Choose names to avoid creating false dependences
- What about parameter values ?
  - Call-by-value ⇒ it can be handled as it was a local variable as before
  - Call-by-reference ⇒ extra indirection 3 instructions. The value may not be kept in a register across an assignment (see next slide)
- What about function calls in expressions?
  - Generate the calling sequence & load the return value
  - Severely limits compiler's ability to reorder operations

# Keeping values in registers

- In a register-to register memory model, the compiler tries to assigns many values as possible to virtual registers

- Then the register allocator will map the set of virtual to physical registers inserting the spills

- However, the compiler can keep values in a register only for unambiguous value:
  a value that can be accessed with just one name is unambiguous

# The problem with ambiguous values

- Consider a and b ambiguous and the following code

```
a := m+n;
b := 13;
c:= a+b;
```

If a and b refers to the same location c gets value 26, otherwise c gets value m+n+13;

The compiler cannot keep a in a register during the assignment of b unless it proves that the set of location that the two name refer to are disjoint. This analysis can be expensive!

sharing analysis !

# Where do ambiguous values arise?

Ambigous values may arise in several ways :

- values stored in a pointer based variable
- call by reference formal parameter
- many compilers treat array element values as ambiguous because they can not tell if two references A[i,j] e A[n,m] refer to the same location

for safety the compiler has to consider that values as **ambiguous**

# Extending the Simple Treewalk Algorithm

## Adding other operators

- Evaluate the operands, then perform the operation
- Complex operations may turn into library calls (exp. and trig fun.)

## Mixed-type expressions

- Insert conversion code as needed from conversion table
- Most languages have symmetric & rational conversion tables

Typical Table for Addition

| + | Integer | Real | Double | Complex |
|---|---|---|---|---|
| Integer | Integer | Real | Double | Complex |
| Real | Real | Real | Double | Complex |
| Double | Double | Double | Double | Complex |
| Complex | Complex | Complex | Complex | Complex |

If the type cannot be inferred at compile time, the compiler must insert code for run-time checks that test for illegal cases!

# Extending the Simple Treewalk Algorithm

What about evaluation order?

Can use commutativity & associativity to improve code for integers

- For recognising that already computed that value

  a+b = b+a

- For recognising that it can compute subexpressions

  a+b+d  and c+a+b

(it does not if it evaluates the expressions in strict left right order!)

It should not reorder floating point expressions!

- The subset of reals represented on a computer does not preserve associativity

  a-b-c   the results may depend on the evaluation order!

# 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 $\Rightarrow$ move rhs
  - lvalue is an address $\Rightarrow$ 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

# Handling Assignment

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

- It is a property of the language & the specific program
- For type-safety, compiler must insert a run-time check
  - Some languages & implementations ignore safety (bad idea)
- Add a tag field to the data items to hold type information
  - Explicitly check tags at runtime

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
```

Choice between conversion & a runtime exception depends on details of language & type system

Much more complex than static checking, plus costs occur at runtime rather than compile time

# Handling Assignment

## Compile-time type-checking

- Goal is to eliminate the need for both tags & runtime checks
- Determine, at compile time, the type of each subexpression
- Use runtime check only if compiler cannot determine types

## Optimization strategy

- If compiler knows the type, move the check to compile-time
- Unless tags are needed for garbage collection, eliminate them
- If check is needed, try to overlap it with other computation

Can design the language so all checks are static

# Summary

## Code Generation for Expressions

- Simple treewalk produces reasonable code
  - Execute most demanding subtree first
  - Can implement treewalk explicitly, with an Attributed grammar

  or ad hoc Syntax directed translation …

- Handle assignment as an operator
  - Insert conversions according to language-specific rules
  - If compile-time checking is impossible, check tags at runtime

Next computing Array access!

# 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
$\Rightarrow$ use a shift for speed

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

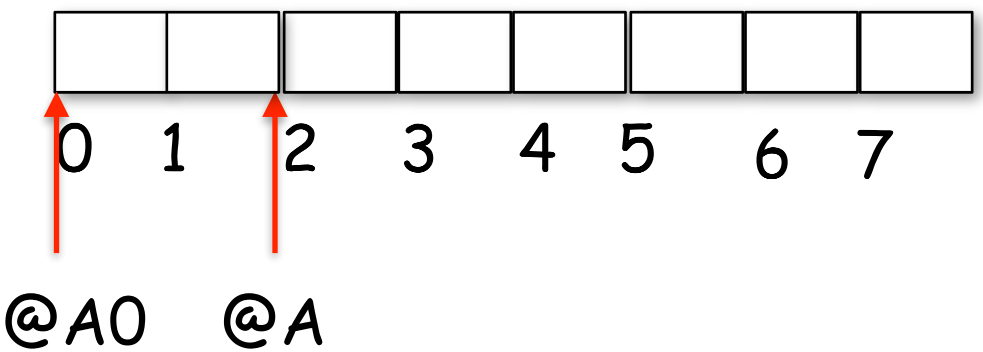
$$A_0 = @A - (low * w)$$

The false zero of A

# The False Zero

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

A[2..7]




**computing A[i] with A**

| | | |
|---|---|---|
| $loadI$ | $@A$ | $\Rightarrow r_{@A}$ |
| $subI$ | $r_i, 2$ | $\Rightarrow r_1$ |
| $lshiftI$ | $r_1, 2$ | $\Rightarrow r_2$ |
| $loadA0$ | $r_{@A}, r_2$ | $\Rightarrow r_v$ |

**computing A[i] with A0**

| | | |
|---|---|---|
| $loadI$ | $@A_0$ | $\Rightarrow r_{@A_0}$ |
| $lshiftI$ | $r_i, 2$ | $\Rightarrow r_1$ |
| $loadA0$ | $r_{@A_0}, r_1$ | $\Rightarrow r_v$ |

# 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

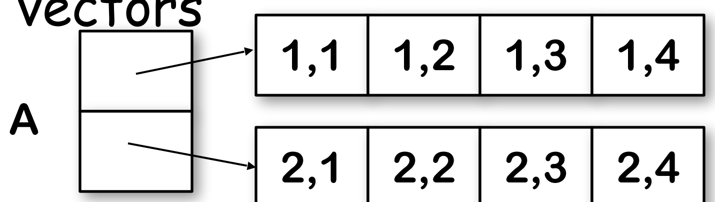
| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1,1 | 1,2 | 1,3 | 1,4 | 2,1 | 2,2 | 2,3 | 2,4 |

## 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: $base(A) + ( i - low ) \times w$

$low_1$ $low_2$ $hight_2$

$hight_1$

| | | | |
|---|---|---|---|
| 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 |
| 2,1 | 2,2 | 2,3 | 2,4 |

$hight_1$

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

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

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

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

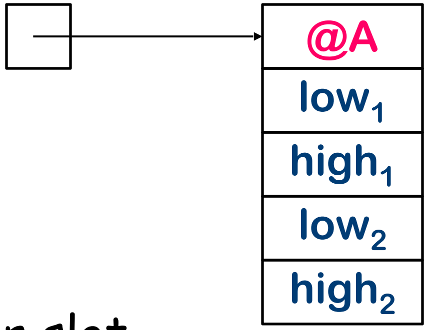
Then, the address expression becomes

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

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

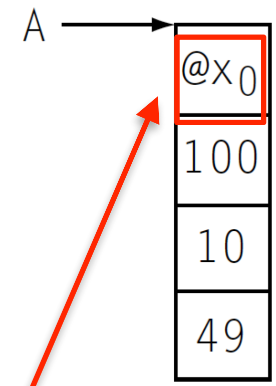
What about call-by-value?

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

# The Dope vector

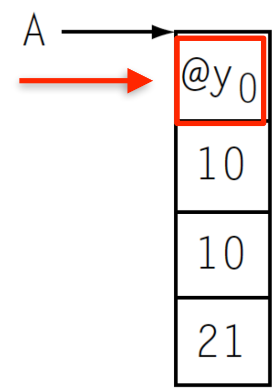
```
program main;
  begin;
    declare x(1:100,1:10,2:50),
        y(1:10,1:10,15:35) float;
    ...
    call fee(x)
    call fee(y);
  end main;

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




At the First Call

false zeros!




At the Second Call

# 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 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 = @A0 + (J x len1 + I ) x w
    R2 = @B0 + (J x len1 + I ) x w
    MEM(R1) = MEM(R1) + MEM(R2)
END DO
```

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

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

# 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 x w

c = $len_1$ x 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

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 8 | a | b̸ | s | t | r | i | n | g |

@b

Length field may take more space than terminator

- Null termination

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| a | b̸ | s | t | r | i | n | g | \0 |

@b

- 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

$$
\begin{array}{lcl}
\textit{Expr} & \rightarrow & \textit{Expr} \lor \textit{AndTerm} \\
 & | & \textit{AndTerm} \\
\textit{AndTerm} & \rightarrow & \textit{AndTerm} \land \textit{RelExpr} \\
 & | & \textit{RelExpr} \\
\textit{RelExpr} & \rightarrow & \textit{RelExpr} < \textit{NumExpr} \\
 & | & \textit{RelExpr} \leq \textit{NumExpr} \\
 & | & \textit{RelExpr} = \textit{NumExpr} \\
 & | & \textit{RelExpr} \neq \textit{NumExpr} \\
 & | & \textit{RelExpr} \geq \textit{NumExpr} \\
 & | & \textit{RelExpr} > \textit{NumExpr}
\end{array}
$$

$$
\begin{array}{lcl}
\textit{NumExpr} & \rightarrow & \textit{NumExpr} + \textit{Term} \\
 & | & \textit{NumExpr} - \textit{Term} \\
 & | & \textit{Term} \\
\textit{Term} & \rightarrow & \textit{Term} \times \textit{Value} \\
 & | & \textit{Term} \div \textit{Value} \\
 & | & \textit{Value} \\
\textit{Value} & \rightarrow & \neg\ \textit{Factor} \\
 & | & \textit{Factor} \\
\textit{Factor} & | & (\ \textit{Expr}\ ) \\
 & | & \textit{number}
\end{array}
$$

# 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)<br>then $stmt_1$<br>else $stmt_2$ | becomes | cmp_LT<br>cbr | $r_x, r_y$<br>$r_1$ | $\Rightarrow r_1$<br>$\rightarrow$ $\_stmt_1, \_stmt_2$ |

# 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 | r1 | $\rightarrow L_1, L_2$ |
| $L_1$: | 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 | $r_e, r_f$ | $\Rightarrow r_a$ |
| $L_{OUT}$: | 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 | r1 | $\rightarrow L_1,L_2$ |
| $L_1$: | 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 | $r_e,r_f$ | $\Rightarrow r_a$ |
| $L_{OUT}$: | 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

| 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 | | | | | |

## Conditional move help here, too

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

| 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

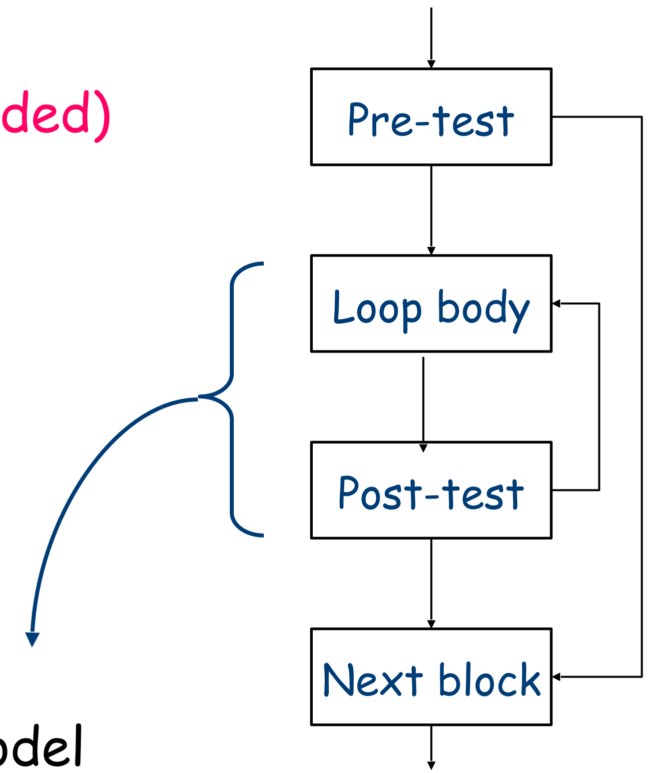
- Modern architectures may favor evaluating full expression
  - Rising branch latencies make the short-circuit path expensive
  - Conditional move and predication may make full path cheaper

- 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 (use break)

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 ($e_1$) {
  case 0: $block_0$
      break;
  case 1: $block_1$
      break;
  case 2: $block_2$
      break;
  ...
  case 9: $block_9$
      break;
  default: $block_d$
      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

$t_1 \leftarrow e_1$
if ($0 > t_1$ or $t_1 > 9$)
  then jump to $LB_d$
  else
    $t_2 \leftarrow$ @Table + $t_1 \times 4$
    $t_3 \leftarrow$ memory($t_2$)
    jump to $t_3$

Code for Address Computation