7 Code Shape

Chapter Contents

Code Shape

Typically, there are **many different ways** how a compiler can map a source-language construct into a sequence of operations in the target instruction set of a given processor.

These variations use different operations and different approaches

- some of these implementations are faster than others
- some use less memory
- some use fewer registers
- some might consume less energy during execution

The concept of **code shape** encapsulates all of the decisions, large and small, that the compiler writer makes on how to represent the computation in both IR and assembly code.

Code shape has a **strong impact** both on the behavior of the compiled code and on the ability of the optimizer and back end to improve it.

Code Shape

Example Consider the way that a C compiler might implement a `switch` statement that switched on a single-byte character value.

```
char ch = ...;
switch(ch) {
    key_1: ... break;
    ...
    key_256: ... break;
    default: ...;
}
```

Alternatives

1. use a cascaded series of `if–then–else` statements to implement the `switch` statement
2. use tests that perform a binary search
3. construct a table of 256 labels and interpret the character by loading the corresponding table entry and jumping to it

Which implementation is best for a particular `switch` statement depends on many factors

- number of cases and their relative execution frequencies
- cost structure for branching on the processor

Assigning Storage Locations

The compiler must **assign a storage location** to each value produced by the code, taking the following factors into account.

- the value's type, its size, its visibility, and its lifetime
- runtime layout of memory
- source-language constraints on the layout of data areas and data structures
- target-processor constraints on placement or use of data

We can distinguish **two types** of values

- **named value**: lifetime is defined by source-language rules and actual use in the code
- **unnamed values**: must be handled consistently with the meaning of the program, but the compiler has great leeway in determining where these values reside and how long to retain them.

Assigning Storage Locations

For each value, the compiler must also decide whether to keep it in a **register** or to keep it in **memory**. The compiler's **memory model** guides it choice of locations for values.

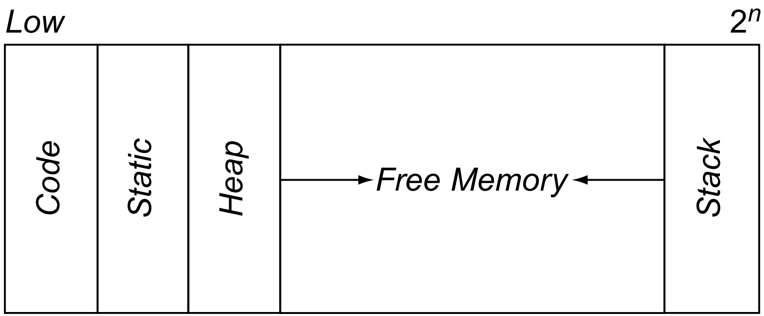
Memory-to-Memory Model

- compiler assumes that all values reside in memory
- values are loaded into registers as needed and stored to memory after each definition
- IR typically uses **physical register** names

Register-to-Register Model

- compiler assumes that it has enough registers to express the computation
- use a distinct **virtual register** for each value that can legally reside in a register
- store a virtual register's value to memory only when absolutely necessary

Placing Runtime Data Structures

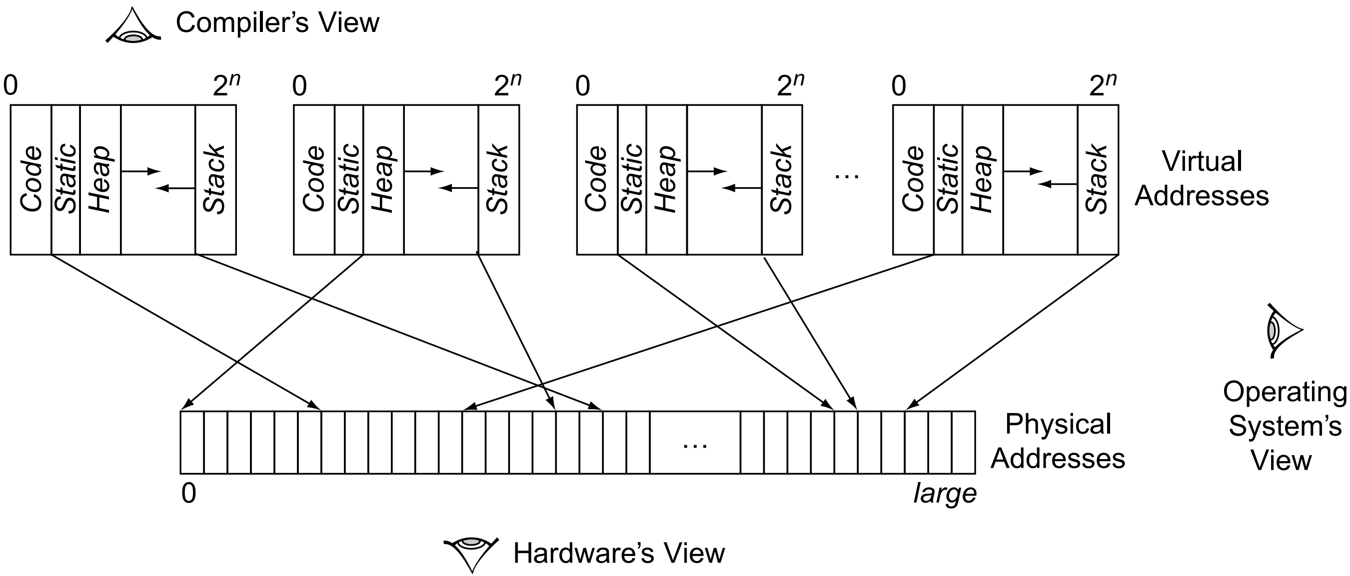


Typical layout for the address space used by a single compiled program

- **code** sits at the bottom of the address space
- **static area** holds both static and global data areas, along with any fixed-size data created by the compiler, *e.g.*, jump tables, debug information
- **heap** contains dynamically allocated data structures
- runtime **stack** is used to stack-allocate activation records, if possible

Placing Runtime Data Structures

The operating system maps multiple **logical address spaces** into the single **physical address space** supported by the processor.



Layout for Data Areas

For convenience, the compiler groups together the storage for values with the **same lifetimes** and **visibility**. It creates distinct **data areas** for them.

Typical placement rules for Algol-like languages

- if x is declared locally in procedure p
  - if its value is not preserved across distinct invocations of p
    $\longrightarrow$ assign it to procedure-local storage
  - if its value is preserved across invocations of p
    $\longrightarrow$ assign it to procedure-local static storage
- if x is declared as globally visible
  $\longrightarrow$ assign it to global storage
- if x is allocated under program control
  $\longrightarrow$ assign it to the runtime heap

Assigning Offsets

In the case of local, static, and global data areas, the compiler must assign each name an **offset** inside the data area. Target ISAs constrain the placement of data items in memory.

Typical alignment rules

- 32-bit integers and floating-point numbers begin on word (32-bit) boundaries
- 64-bit integer and floating-point data begin on doubleword (64-bit) boundaries
- string data begin on halfword (16-bit) boundaries

To **minimize wasted space**, the compiler should order the variables into groups, from those with the most restrictive alignment rules to those with the least.

Note Doubleword alignment is more restrictive than word alignment.

Keeping Values in Registers

The register-to-register memory model has **three** principal advantages

- it is simple
- it can improve the results of analysis and optimization
- it enhances portability by postponing processor-specific constraints until optimization

However, only **unambiguous values** can be kept in registers. **Ambiguous values** cannot be kept in a register across either a definition or a use of another ambiguous value.

Ambiguous and Unambiguous Values

Any value that can be accessed by multiple names is **ambiguous**. In contrast, a value that can be accessed with just one name is **unambiguous**.

Keeping Values in Registers

Ambiguity arises in several ways

- values stored in pointer-based variables are often ambiguous
- call-by-reference formal parameters and name scoping rules can make the formal parameters ambiguous
- array-element values can be ambiguous values since two references could refer to the same location

$$
\begin{array}{lcl}
\mathsf{a} & \leftarrow & \mathtt{m} + \mathsf{n} \\
\mathsf{b} & \leftarrow & 13 \\
\mathsf{c} & \leftarrow & \mathsf{a} + \mathsf{b}
\end{array}
$$

With careful analysis, the compiler can **disambiguate** some of these cases.

- if a and b refer to the same location, c gets the value 26, otherwise it gets $\mathtt{m} + \mathsf{n} + 13$
- to keep a in a register across assignment to another ambiguous value, the compiler needs to prove that the sets of locations to which a and b can refer are **disjoint**

Since pairwise ambiguity analysis is expensive, compilers typically relegate ambiguous values to memory, with a load before each use and a store after each definition.

Arithmetic Operators

Modern processors provide a full complement of operations to evaluate expressions

- arithmetic operators, *e.g.*, `add`, `sub`, `imul`, and `idiv`
- shift and rotate operators, *e.g.*, `shl`, `shr`, `rol`, and `ror`
- boolean operators, *e.g.*, `and`, `or`, `xor`, and `not`

To generate code for a **trivial expression**, such as $a + b$, the compiler emits code to load the values of a and b into registers, followed by an instruction to perform the addition.

$$
\begin{array}{lll}
\texttt{loadI} & \texttt{@a} & \Rightarrow r_1 \\
\texttt{loadAO} & r_{arp}, r_1 & \Rightarrow r_a \\
\texttt{loadI} & \texttt{@b} & \Rightarrow r_2 \\
\texttt{loadAO} & r_{arp}, r_2 & \Rightarrow r_b \\
\texttt{add} & r_a, r_b & \Rightarrow r_t
\end{array}
$$

If the value of a or b is already in a register, the compiler can avoid the load instructions and simply use that register in place of r_a or r_b, respectively.

Arithmetic Operators

If the expression is represented in a tree-like IR, this process fits into a **postorder** tree walk.

`base`

- returns the name of a register holding the base address for an identifier
- if needed, it emits code to get that address into a register

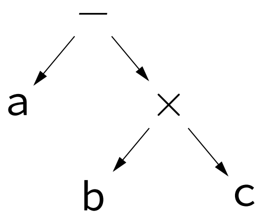
`offset`

- returns the name of a register holding the identifier's offset
- offset is relative to the address returned by `base`

```
procedure expr(n)
    if n ∈ {+, −, ×, ÷} then
        t1 ← expr(n.left)
        t2 ← expr(n.right)
        r ← NextRegister()
        emit(n, t1, t2, r)
    else if n = ident then
        t1 ← base(n)
        t2 ← offset(n)
        r ← NextRegister()
        emit(loadAO, t1, t2, r)
    else if n = num then
        r ← NextRegister()
        emit(loadI, n, _, r)
    return r
```

Arithmetic Operators

Example Invoking the routine `expr` on the AST for $\texttt{a} - \texttt{b} \times \texttt{c}$ shown on the left produces the results shown on the right.



$$
\begin{array}{llcl}
\texttt{loadI} & \texttt{@a} & \Rightarrow & r_1 \\
\texttt{loadAO} & r_{arp},\ r_1 & \Rightarrow & r_2 \\
\texttt{loadI} & \texttt{@b} & \Rightarrow & r_3 \\
\texttt{loadAO} & r_{arp},\ r_3 & \Rightarrow & r_4 \\
\texttt{loadI} & \texttt{@c} & \Rightarrow & r_5 \\
\texttt{loadAO} & r_{arp},\ r_5 & \Rightarrow & r_6 \\
\texttt{mult} & r_4,\ r_6 & \Rightarrow & r_7 \\
\texttt{add} & r_2,\ r_7 & \Rightarrow & r_8
\end{array}
$$

The example assumes that `a`, `b`, and `c` are not already in registers and that each resides in the current AR.

Reducing Demand for Registers

The choice of storage locations has a direct impact on the **quality** of the generated code

- if a was in a global data area, we need another `loadI` and register to load it
- if a was already in a register, the two instructions to load it could be omitted

Code-shape decisions encoded into the treewalk code generator have an effect on demand for registers

- the naive code uses **eight** registers plus r_{arp}
- a register allocator could rewrite the code as shown on the right
- the rewritten code uses **three** registers plus r_{arp}

`loadI`	@a	$\Rightarrow$	r_1
`loadAO`	r_{arp}, r_1	$\Rightarrow$	r_1
`loadI`	@b	$\Rightarrow$	r_2
`loadAO`	r_{arp}, r_2	$\Rightarrow$	r_2
`loadI`	@c	$\Rightarrow$	r_3
`loadAO`	r_{arp}, r_3	$\Rightarrow$	r_3
`mult`	r_2, r_3	$\Rightarrow$	r_2
`add`	r_1, r_2	$\Rightarrow$	r_2

Reducing Demand for Registers

A different code shape can reduce the demand for registers. Using a **right-to-left** tree walk instead of a **left-to-right** tree walk produces the code shown below on the left.

loadI	@c	$\Rightarrow$	r_1	loadI	@c	$\Rightarrow$	r_1
loadAO	r_{arp}, r_1	$\Rightarrow$	r_2	loadAO	r_{arp}, r_1	$\Rightarrow$	r_1
loadI	@b	$\Rightarrow$	r_3	loadI	@b	$\Rightarrow$	r_2
loadAO	r_{arp}, r_3	$\Rightarrow$	r_4	loadAO	r_{arp}, r_2	$\Rightarrow$	r_2
mult	r_2, r_4	$\Rightarrow$	r_5	mult	r_1, r_2	$\Rightarrow$	r_1
loadI	@a	$\Rightarrow$	r_6	loadI	@a	$\Rightarrow$	r_2
loadAO	r_{arp}, r_6	$\Rightarrow$	r_7	loadAO	r_{arp}, r_2	$\Rightarrow$	r_2
add	r_7, r_5	$\Rightarrow$	r_2	add	r_2, r_1	$\Rightarrow$	r_1

After register allocation, the code shown above on the right only uses **two** registers plus r_{arp}.

Reducing Demand for Registers

Of course, right-to-left evaluation is not a general solution. To generate an evaluation order that reduces demand for registers, the compiler must chose between right and left children.

> **Rule**
>
> The compiler can **minimize** register use by evaluating first, at each node, the sub-tree that needs the most registers.

This approach requires **two** passes over the AST

- the first pass computes the demand for registers
- the second pass emits the actual code

Accessing Parameter Values

Formal parameters need may a different treatment

- a **call-by-value** parameter passed in the AR can be treated as if it was a local variable
- a **call-by-reference** parameter passed in the AR requires one additional indirection

Example For the call-by-reference parameter d, the compiler might generate the following instructions to obtain d's value.

$$
\begin{array}{lll}
\texttt{loadI} & \texttt{@d} & \Rightarrow r_1 \\
\texttt{loadAO} & r_{arp}, r_1 & \Rightarrow r_2 \\
\texttt{load} & r_2 & \Rightarrow r_3
\end{array}
$$

The first two instructions move the address of d into r_2, the third instruction loads its value.

Accessing Parameter Values

Many linkage conventions pass the first few parameters in registers. So far, our code generation algorithm cannot handle a value that is **permanently** kept in a register.

The necessary extensions are easy to implement

- **Call-by-Value Parameters**
  The `ident` case must check if the value is already in a register
  - if so, it just assigns the register number to `r`
  - otherwise, it uses the standard mechanisms to load the value from memory
- **Call-by-Reference Parameters**
  - if the address resides in a register, simply load the value into a register
  - if the address resides in the AR, must load the address before loading the value

The Small Print Note that the compiler cannot keep the value of a call-by-reference parameter in a register across an assignment, unless the compiler can prove that the reference is unambiguous, across all calls to the procedure.

Function Calls in an Expression

Apart from variables, constants, and temporary values produced by other subexpressions, **function calls** also occur as operands in expressions.

To evaluate a function call, the compiler performs the following steps

- generate the calling sequence needed to invoke the function
- emit the code necessary to move the returned value to a register

Recall The linkage convention limits the callee's impact on the caller.

The presence of a function call may restrict the compiler's ability to **change** the evaluation order of an expression

- function may have **side effects** that modify values of variables used in the expression
- without further analysis, compiler must emit code that is correct in the **worst case**

Other Arithmetic Operators

To handle other arithmetic operations, we can extend the treewalk model.

The basic scheme remains the same

- get the operands into registers
- perform the operation
- store the result

Note Operator precedence, encoded into the expression grammar, ensures the correct evaluation order.

Some operators require complex **multioperation sequences** for their implementation, *e.g.*, exponentiation and trigonometric functions

- expand operation sequence inline
- emit call to a library routine

Mixed-Type Expressions

Many programming languages support operations with operands of **different types**. The compiler must recognize this situation and insert the **conversion code**.

Built-in Types

- definition of programming language specifies a formula for each conversion
- some processors provide explicit conversion operators
- others expect the compiler to generate complex, machine-dependent code

Programmer-defined Types

- compiler has no conversion tables that define each specific case
- source language still defines the meaning of the expression
- compiler must implement this meaning and reject illegal expressions

Assignment as an Operator

Most Algol-like languages implement assignment with the following simple rules

1. evaluate the right-hand side of the assignment to a **value**
2. evaluate the left-hand side of the assignment to a **location**
3. store the right-hand side value into the left-hand side location

Distinguishing between these modes of evaluation

- **rvalue**: result (value) of evaluation on the right-hand side of an assignment
- **lvalue**: result (address) of evaluation on the left-hand side of an assignment

In an assignment, the type of the lvalue can differ from the type of the rvalue, which may require either a compiler-inserted **conversion** or an **error message**.

Note The typical rule for such a conversion is to evaluate the rvalue to its natural type and then convert the result to the type of the lvalue.

Boolean and Relational Operators

Most programming languages can operate on the results of **Boolean** and **relational operators**, both of which produce Boolean values.

To support such values, a compiler writer must…

- augment standard expression grammar with Boolean and relational operators
- define and enforce typing and inference rules for these operators
- decide how to represent Boolean values and how to compute them

Most architectures provide a **rich set** of Boolean operations, but support for relational operators **varies widely** from one architecture to another.

The compiler writer must find an evaluation strategy that matches the needs of the language to the available instruction set.

Boolean and Relational Operators

$$
\begin{array}{r|rcl}
0 & \mathit{Expr} & \rightarrow & \mathit{Expr} \vee \mathit{AndTerm} \\
1 & & | & \mathit{AndTerm} \\
2 & \mathit{AndTerm} & \rightarrow & \mathit{AndTerm} \wedge \mathit{RelExpr} \\
3 & & | & \mathit{RelExpr} \\
4 & \mathit{RelExpr} & \rightarrow & \mathit{RelExpr} < \mathit{NumExpr} \\
5 & & | & \mathit{RelExpr} \leq \mathit{NumExpr} \\
6 & & | & \mathit{RelExpr} = \mathit{NumExpr} \\
7 & & & \mathit{RelExpr} \neq \mathit{NumExpr} \\
8 & & | & \mathit{RelExpr} \geq \mathit{NumExpr} \\
9 & & | & \mathit{RelExpr} > \mathit{NumExpr} \\
10 & & | & \mathit{NumExpr}
\end{array}
$$

$$
\begin{array}{r|rcl}
11 & \mathit{NumExpr} & \rightarrow & \mathit{NumExpr} + \mathit{Term} \\
12 & & | & \mathit{NumExpr} - \mathit{Term} \\
13 & & | & \mathit{Term} \\
14 & \mathit{Term} & \rightarrow & \mathit{Term} \times \mathit{Value} \\
15 & & | & \mathit{Term} \div \mathit{Value} \\
16 & & | & \mathit{Factor} \\
17 & \mathit{Value} & \rightarrow & \neg\ \mathit{Factor} \\
18 & & | & \mathit{Factor} \\
19 & \mathit{Factor} & \rightarrow & (\ \mathit{Expr}\ ) \\
20 & & | & \mathtt{num} \\
21 & & | & \mathtt{ident}
\end{array}
$$

Representations

Traditionally, two representations have been proposed for boolean values.

Numerical Encoding

- assign specific values to `true` and `false`
- manipulate them using the target machine's arithmetic and logical operations

Positional Encoding

- encode the value of the expression as a position in the executable code
- use comparisons and conditional branches to evaluate the expression
- different control-flow paths represent the result of evaluation

$\longrightarrow$ Each approach works well for some examples, but not for others…

Numerical Encoding

When the program **stores** the result of a Boolean or relational operation into a variable, the compiler must assign **numerical values** to `true` and `false`.

Typical Values that work with hardware operations such as `and`, `or`, and `not`

- `false`: zero
- `true`: one, word of ones, or ¬`false`

Example If b, c, and d are all in registers, the compiler might produce the following code for the expression $b \lor c \land \neg d$.

$$
\begin{array}{llll}
\texttt{not} & r_d & \Rightarrow & r_1 \\
\texttt{and} & r_c,\ r_1 & \Rightarrow & r_2 \\
\texttt{or} & r_b,\ r_2 & \Rightarrow & r_3
\end{array}
$$

Numerical Encoding

For a **comparison**, such as $a < b$, the compiler must generate code that compares a and b and assigns the appropriate value to the result.

1. the target machine provides a comparison operation that **returns a Boolean**

   $\texttt{cmp\_LT}\ \ r_a,\ r_b \Rightarrow r_1$

2. the comparison defines a **condition code** that must be read with a branch

$$
\begin{array}{llll}
 & \texttt{comp} & r_a,\ r_b & \Rightarrow\ cc_1 \\
 & \texttt{cbr\_LT} & cc_1 & \rightarrow\ L_1,\ L_2 \\
L_1: & \texttt{loadI} & \texttt{true} & \Rightarrow\ r_1 \\
 & \texttt{jumpI} & & \rightarrow\ L_3 \\
L_2: & \texttt{loadI} & \texttt{false} & \Rightarrow\ r_1 \\
 & \texttt{jumpI} & & \rightarrow\ L_3 \\
L_3: & \texttt{nop} & &
\end{array}
$$

Condition Codes on Intel x86-64

Nearly all arithmetic instructions set condition codes based on their result.

ZF	result was **z**ero
CF	result caused **c**arry out of most significant bit
SF	result was negative (**s**ign bit was set)
OF	result caused (signed) **o**verflow

Based on these condition codes, standard condition suffixes *cc* are defined (*cf.* Slide 349) that modify the behaviur of three different kinds of instructions.

`j`*cc*	jumps to the specified label if *cc* holds
`set`*cc*	sets the given (single-byte) register to `1` or `0` depending on whether *cc* holds or not
`cmov`*cc*	performs the specified move only if *cc* holds

The table on the right shows the **standard condition suffixes** defined by the Intel x86-64 instruction set.

Example The expression a $<$ b could be implemented as follows.

```
cmp     %rdi, %rsi
setl    %al
```

Note Since `setl` needs a single-byte register, the example uses `%al` instead of `%rax`.

cc	**Condition Tested**	**Meaning**
e	ZF	equal to zero
ne	$\neg$ZF	not equal to zero
s	SF	negative
ns	$\neg$SF	not negative
g	$\neg$(SF $\oplus$ OF) $\wedge$ $\neg$ZF	greater (signed $>$)
ge	$\neg$(SF $\oplus$ OF)	greater or equal (signed $\geq$)
l	SF $\oplus$ OF	less (signed $<$)
le	(SF $\oplus$ OF) $\vee$ ZF	less or equal (signed $\leq$)
a	$\neg$CF $\wedge$ $\neg$ZF	above (unsigned $>$)
ae	$\neg$CF	above or equal (unsigned $\geq$)
b	CF	below (unsigned $<$)
be	CF $\vee$ ZF	below or equal (unsigned $\leq$)

Positional Encoding

Positional encoding makes sense if an expression's result is **never** stored

- result of subexpression evaluations
- expressions to determine control flow

Short-Circuit Evaluation

In **short-circuit evaluation**, expressions are only evaluated until their final value is determined. Short-circuit evaluation relies on two Boolean identities.

$$\forall x \;\; \mathtt{false} \wedge x = \mathtt{false}$$
$$\forall x \;\; \mathtt{true} \vee x = \mathtt{true}$$

Some programming languages, *e.g.*, C and Java, **require** the compiler to use short-circuit evaluation.

Positional Encoding

Example Consider the following code for the expression $a < b \lor c < d \land e < f$ that a naive code generator would emit.

$$
\begin{array}{lllll}
 & \texttt{comp} & \texttt{r}_a, \texttt{r}_b & \Rightarrow & \texttt{cc}_1 \\
 & \texttt{cbr\_LT} & \texttt{cc}_1 & \rightarrow & \texttt{L}_1, \texttt{L}_2 \\
\texttt{L}_1\texttt{:} & \texttt{loadI} & \texttt{true} & \Rightarrow & \texttt{r}_1 \\
 & \texttt{jumpI} & & \rightarrow & \texttt{L}_3 \\
\texttt{L}_2\texttt{:} & \texttt{loadI} & \texttt{false} & \Rightarrow & \texttt{r}_1 \\
 & \texttt{jumpI} & & \rightarrow & \texttt{L}_3 \\
\texttt{L}_3\texttt{:} & \texttt{comp} & \texttt{r}_c, \texttt{r}_d & \Rightarrow & \texttt{cc}_2 \\
 & \texttt{cbr\_LT} & \texttt{cc}_2 & \rightarrow & \texttt{L}_4, \texttt{L}_5 \\
\texttt{L}_4\texttt{:} & \texttt{loadI} & \texttt{true} & \Rightarrow & \texttt{r}_2 \\
 & \texttt{jumpI} & & \rightarrow & \texttt{L}_6
\end{array}
$$

$$
\begin{array}{lllll}
\texttt{L}_5\texttt{:} & \texttt{loadI} & \texttt{false} & \Rightarrow & \texttt{r}_2 \\
 & \texttt{jumpI} & & \rightarrow & \texttt{L}_6 \\
\texttt{L}_6\texttt{:} & \texttt{comp} & \texttt{r}_e, \texttt{r}_f & \Rightarrow & \texttt{cc}_3 \\
 & \texttt{cbr\_LT} & \texttt{cc}_3 & \rightarrow & \texttt{L}_7, \texttt{L}_8 \\
\texttt{L}_7\texttt{:} & \texttt{loadI} & \texttt{true} & \Rightarrow & \texttt{r}_3 \\
 & \texttt{jumpI} & & \rightarrow & \texttt{L}_9 \\
\texttt{L}_8\texttt{:} & \texttt{loadI} & \texttt{false} & \Rightarrow & \texttt{r}_3 \\
 & \texttt{jumpI} & & \rightarrow & \texttt{L}_9 \\
\texttt{L}_9\texttt{:} & \texttt{and} & \texttt{r}_2, \texttt{r}_3 & \Rightarrow & \texttt{r}_4 \\
 & \texttt{or} & \texttt{r}_1, \texttt{r}_4 & \Rightarrow & \texttt{r}_5
\end{array}
$$

Note Every path takes **eleven** operations, including **three** branches and **three** jumps!

Positional Encoding

If the compiler avoids storing intermediate results of subexpressions and uses short-circuit evaluation, it can emit **much more efficient** code.

$$
\begin{array}{lllll}
 & \texttt{comp} & \texttt{r}_a, \texttt{r}_b & \Rightarrow & \texttt{cc}_1 \\
 & \texttt{cbr\_LT} & \texttt{cc}_1 & \rightarrow & \texttt{L}_3, \texttt{L}_1 \\
\texttt{L}_1: & \texttt{comp} & \texttt{r}_c, \texttt{r}_d & \Rightarrow & \texttt{cc}_2 \\
 & \texttt{cbr\_LT} & \texttt{cc}_2 & \rightarrow & \texttt{L}_2, \texttt{L}_4 \\
\texttt{L}_2: & \texttt{comp} & \texttt{r}_e, \texttt{r}_f & \Rightarrow & \texttt{cc}_3 \\
 & \texttt{cbr\_LT} & \texttt{cc}_3 & \rightarrow & \texttt{L}_3, \texttt{L}_4 \\
\texttt{L}_3: & \texttt{loadI} & \texttt{true} & \Rightarrow & \texttt{r}_5 \\
 & \texttt{jumpI} & & \rightarrow & \texttt{L}_5 \\
\texttt{L}_4: & \texttt{loadI} & \texttt{false} & \Rightarrow & \texttt{r}_5 \\
 & \texttt{jumpI} & & \rightarrow & \texttt{L}_5 \\
\texttt{L}_5: & \texttt{nop} & & &
\end{array}
$$

Positional Encoding

When the code uses the result of an expression to determine control flow, positional encoding often avoids extraneous operations.

```
if (a < b)
{
    statement₁;
}
else
{
    statement₂;
}
```

```
      comp    r_a, r_b  ⇒ cc₁
      cbr_LT  cc₁       → L₁, L₂
L₁:   code for statement₁
      jumpI             → L₃
L₂:   code for statement₂
      jumpI             → L₃
L₃:   nop
```

Note The code combines the evaluation of $a < b$ with the selection between *statement*$_1$ and *statement*$_2$. The result of $a < b$ is represented as a position, either L_1 or L_2.

Hardware Support for Relational Operations

Specific, low-level details in the target machine's instruction set strongly influence the choice of a representation for relational values.

We will consider four schemes for supporting relational expressions

- straight condition codes
- condition codes augmented with a conditional move operation
- boolean-valued comparisons
- predicated operations

For each of these, we will examine the implementation of an `if-then-else` statement and an assignment of a Boolean value.

Note The first two schemes are supported by the Intel x84-64 instruction set.

Source Code

```
if (x < y)
   then a ← c + d
   else a ← e + f
```

ILOC Code

Straight Condition Codes

```
       comp    rx, ry  ⇒ cc1
       cbr_LT  cc1     → L1, L2
L1:    add     rc, rd  ⇒ ra
       jumpI           → Lout
L2:    add     re, rf  ⇒ ra
       jumpI           → Lout
Lout:  nop
```

Boolean Compare

```
       cmp_LT  rx, ry  ⇒ r1
       cbr     r1      → L1, L2
L1:    add     rc, rd  ⇒ ra
       jumpI           → Lout
L2:    add     re, rf  ⇒ ra
       jumpI           → Lout
Lout:  nop
```

Conditional Move

```
comp    rx, ry          ⇒ cc1
add     rc, rd          ⇒ r1
add     re, rf          ⇒ r2
i2i_LT  cc1, r1, r2     ⇒ ra
```

Predicated Execution

```
       cmp_LT  rx, ry  ⇒ r1
       not     r1      ⇒ r2
(r1)?  add     rc, rd  ⇒ ra
(r2)?  add     re, rf  ⇒ ra
```

Source Code

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

ILOC Code

```
        comp    r_a, r_b   ⇒ cc_1
        cbr_LT  cc_1       → L_1,L_2
L_1:    comp    r_c, r_d   ⇒ cc_2
        cbr_LT  cc_2       → L_3,L_2
L_2:    loadI   false      ⇒ r_x
        jumpI              → L_out
L_3:    loadI   true       ⇒ r_x
        jumpI              → L_out
L_out:  nop
```

Straight Condition Codes

```
comp    r_a,r_b          ⇒ cc_1
i2i_LT  cc_1,r_T,r_F     ⇒ r_1
comp    r_c,r_d          ⇒ cc_2
i2i_LT  cc_2,r_T,r_F     ⇒ r_2
and     r_1,r_2          ⇒ r_x
```

Conditional Move

```
cmp_LT  r_a, r_b   ⇒ r_1
cmp_LT  r_c, r_d   ⇒ r_2
and     r_1, r_2   ⇒ r_x
```

Boolean Compare

```
cmp_LT  r_a, r_b   ⇒ r_1
cmp_LT  r_c, r_d   ⇒ r_2
and     r_1, r_2   ⇒ r_x
```

Predicated Execution

Hardware Support for Relational Operations

Condition Codes

- code has at least one conditional branch per relational operator
- comparison operation may be omitted if condition codes are set by default

Conditional Move

- leads to faster code by avoiding branches
- safe as long as neither operation can raise an exception

Boolean Compare

- works without a branch and without converting comparison results to Boolean values
- a weakness of this model is that it requires explicit comparisons

Predicated Execution

- code is simple and concise
- predication can lead to the same code as the boolean-comparison scheme

Relational Operations on Intel x86-64

The Intel x86-64 instruction set supports both the **straight condition codes** and **conditional move** implementation scheme.

Straight condition codes are typically used to implement control-flow constructs, whereas conditional moves are used to evaluate and assign expressions.

A compiler for Intel x86-64 might implement the two examples from before as follows.

```
      cmpq   %rdi, %rsi
      jl     _L1
      leaq   (%rdx, %rcx), %rax
      jmp    _L2
_L1:  leaq   (%r8, %r9), %rax
_L2:  nop
```

```
      cmpq   %rdi, %rsi
      setl   %al
      cmpq   %rdx, %rcx
      setl   %bl
      and    %al, %bl
      movzbq %bl, %rax
```