7 Code Shape

Chapter Contents

1. Assigning Storage Locations
2. Arithmetic Operators
3. Boolean and Relational Operators
4. Storing and Accessing Arrays
5. Character Strings
6. Structure References
7. Control-Flow Constructs
8. Procedure Calls

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.

```
1  char ch = ...;
2  switch(ch) {
3     key₁: ... break;
4     ...
5     key₂₅₆: ... break;
6     default: ...;
7  }
```

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

Compiler Construction

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
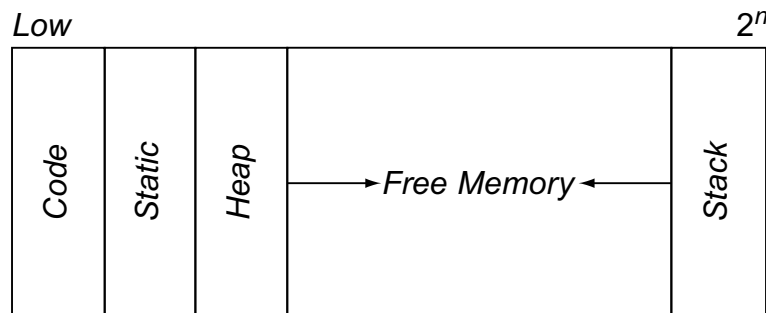
Typical layout for the address space used by a single compiled program
- **code** sits at the bottom of the address space
- **static area** holds both static and global data areas, along with any fixed-size data created by the compiler, *e.g.*, jump tables, debug information
- **heap** contains dynamically 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.
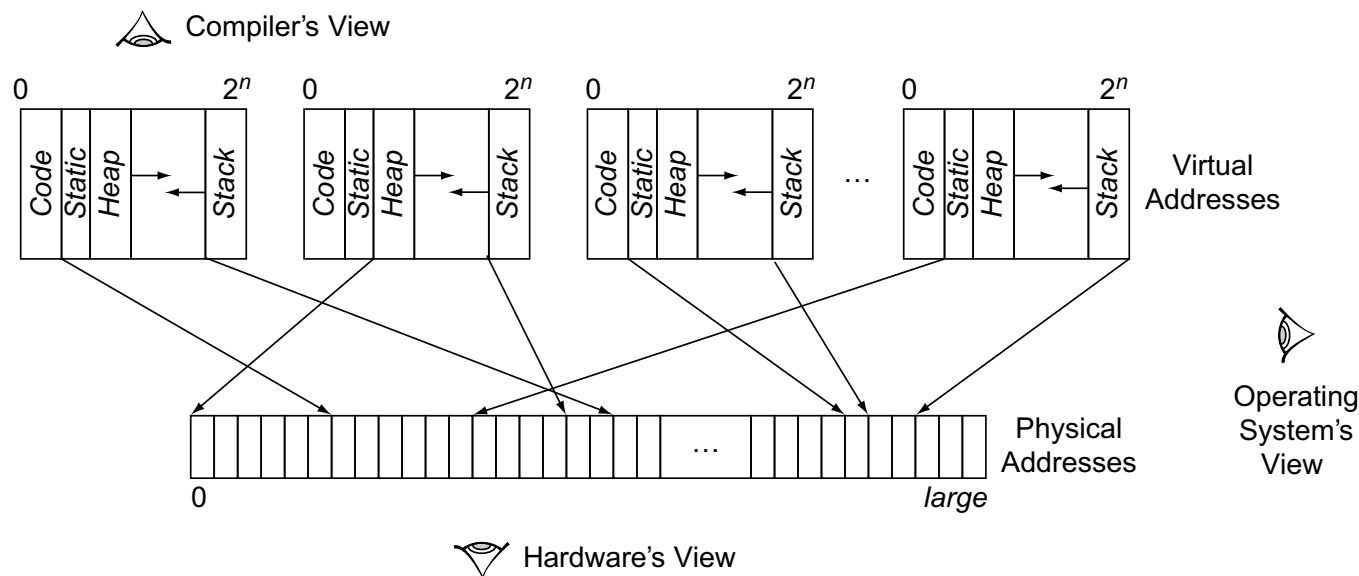
Layout for Data Areas

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

$$a \leftarrow m + n$$
$$b \leftarrow 13$$
$$c \leftarrow a + b$$

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 $m + n + 13$
- 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.

```
loadI   @a          ⇒  r_1
loadAO  r_arp,  r_1  ⇒  r_a
loadI   @b          ⇒  r_2
loadAO  r_arp,  r_2  ⇒  r_b
add     r_a,  r_b    ⇒  r_t
```

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`

```
1  procedure expr(n)
2     if n ∈ {+, −, ×, ÷} then
3        t₁ ← expr(n.left)
4        t₂ ← expr(n.right)
5        r ← NextRegister()
6        emit(n, t₁, t₂, r)
7     else if n = ident then
8        t₁ ← base(n)
9        t₂ ← offset(n)
10       r ← NextRegister()
11       emit(loadAO, t₁, t₂, r)
12    else if n = num then
13       r ← NextRegister()
14       emit(loadI, n, _, r)
15    return r
```

Arithmetic Operators

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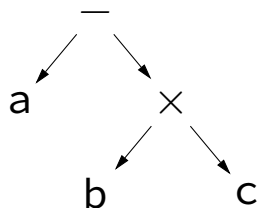

```
loadI   @a           ⇒  r₁
loadAO  r_arp, r₁  ⇒  r₂
loadI   @b           ⇒  r₃
loadAO  r_arp, r₃  ⇒  r₄
loadI   @c           ⇒  r₅
loadAO  r_arp, r₅  ⇒  r₆
mult    r₄, r₆      ⇒  r₇
add     r₂, r₇      ⇒  r₈
```

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           ⇒ r₁
loadAO  rarp, r₁      ⇒ r₁
loadI   @b           ⇒ r₂
loadAO  rarp, r₂      ⇒ r₂
loadI   @c           ⇒ r₃
loadAO  rarp, r₃      ⇒ r₃
mult    r₂, r₃        ⇒ r₂
add     r₁, r₂        ⇒ r₂
```

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         ⇒ r₁          loadI   @c         ⇒ r₁
loadAO  r_arp, r₁  ⇒ r₂          loadAO  r_arp, r₁  ⇒ r₁
loadI   @b         ⇒ r₃          loadI   @b         ⇒ r₂
loadAO  r_arp, r₃  ⇒ r₄          loadAO  r_arp, r₂  ⇒ r₂
mult    r₂, r₄     ⇒ r₅          mult    r₁, r₂     ⇒ r₁
loadI   @a         ⇒ r₆          loadI   @a         ⇒ r₂
loadAO  r_arp, r₆  ⇒ r₇          loadAO  r_arp, r₂  ⇒ r₂
add     r₇, r₅     ⇒ r₂          add     r₂, r₁     ⇒ r₁
```

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.

```
loadI   @d          ⇒  r₁
loadAO  r_arp, r₁  ⇒  r₂
load    r₂          ⇒  r₃
```

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

0	*Expr*	$\rightarrow$	*Expr* $\vee$ *AndTerm*
1		|	*AndTerm*
2	*AndTerm*	$\rightarrow$	*AndTerm* $\wedge$ *RelExpr*
3		|	*RelExpr*
4	*RelExpr*	$\rightarrow$	*RelExpr* $<$ *NumExpr*
5		|	*RelExpr* $\leq$ *NumExpr*
6		|	*RelExpr* $=$ *NumExpr*
7			*RelExpr* $\neq$ *NumExpr*
8		|	*RelExpr* $\geq$ *NumExpr*
9		|	*RelExpr* $>$ *NumExpr*
10		|	*NumExpr*
11	*NumExpr*	$\rightarrow$	*NumExpr* $+$ *Term*
12		|	*NumExpr* $-$ *Term*
13		|	*Term*
14	*Term*	$\rightarrow$	*Term* $\times$ *Value*
15		|	*Term* $\div$ *Value*
16		|	*Factor*
17	*Value*	$\rightarrow$	$\neg$ *Factor*
18		|	*Factor*
19	*Factor*	$\rightarrow$	(*Expr*)
20		|	num
21		|	ident

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 ∨ c ∧ ¬d.

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

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**

```
cmp_LT  r_a, r_b  ⇒  r_1
```

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

```
        comp    r_a, r_b  ⇒  cc_1
        cbr_LT  cc_1        →  L_1, L_2
   L_1: loadI   true      ⇒  r_1
        jumpI              →  L_3
   L_2: loadI   false     ⇒  r_1
        jumpI              →  L_3
   L_3: nop
```

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.

```
1  cmp     %rdi, %rsi
2  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	$\neg(\text{SF} \oplus \text{OF}) \wedge \neg\text{ZF}$	greater (signed $>$)
ge	$\neg(\text{SF} \oplus \text{OF})$	greater or equal (signed $\geq$)
l	$\text{SF} \oplus \text{OF}$	less (signed $<$)
le	$(\text{SF} \oplus \text{OF}) \vee \text{ZF}$	less or equal (signed $\leq$)
a	$\neg\text{CF} \wedge \neg\text{ZF}$	above (unsigned $>$)
ae	$\neg\text{CF}$	above or equal (unsigned $\geq$)
b	CF	below (unsigned $<$)
be	$\text{CF} \vee \text{ZF}$	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.

$$\forall x \ \ \texttt{false} \wedge x = \texttt{false}$$
$$\forall x \ \ \ \texttt{true} \vee x = \texttt{true}$$

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 $a < b \vee c < d \wedge e < f$ that a naive code generator would emit.

```
        comp    ra, rb  ⇒ cc1          L5: loadI   false ⇒ r2
        cbr_LT cc1      → L1, L2           jumpI          → L6
    L1: loadI   true  ⇒ r1            L6: comp    re, rf ⇒ cc3
        jumpI           → L3               cbr_LT cc3      → L7, L8
    L2: loadI   false ⇒ r1            L7: loadI   true  ⇒ r3
        jumpI           → L3               jumpI           → L9
    L3: comp    rc, rd  ⇒ cc2          L8: loadI   false ⇒ r3
        cbr_LT cc2      → L4, L5           jumpI           → L9
    L4: loadI   true  ⇒ r2            L9: and     r2, r3 ⇒ r4
        jumpI           → L6               or      r1, r4 ⇒ r5
```

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.

```
        comp    ra, rb  ⇒  cc1
        cbr_LT  cc1       →  L3, L1
L1:     comp    rc, rd  ⇒  cc2
        cbr_LT  cc2       →  L2, L4
L2:     comp    re, rf  ⇒  cc3
        cbr_LT  cc3       →  L3, L4
L3:     loadI   true    ⇒  r5
        jumpI             →  L5
L4:     loadI   false   ⇒  r5
        jumpI             →  L5
L5:     nop
```

Positional Encoding

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

```
1  if (a < b)
2  {
3      statement₁ ;
4  }
5  else
6  {
7      statement₂ ;
8  }
```

$$
\begin{aligned}
&\text{comp} \quad r_a, r_b \Rightarrow cc_1 \\
&\text{cbr\_LT } cc_1 \quad \rightarrow L_1, L_2 \\
L_1: \ &\text{code for } statement_1 \\
&\text{jumpI} \quad \rightarrow L_3 \\
L_2: \ &\text{code for } statement_2 \\
&\text{jumpI} \quad \rightarrow L_3 \\
L_3: \ &\text{nop}
\end{aligned}
$$

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`

<table>
<tr>
<td rowspan="2">ILOC Code</td>
<td>

```
        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
```
Straight Condition Codes
</td>
<td>

```
        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
```
Boolean Compare
</td>
</tr>
<tr>
<td>

```
comp   rx,ry       ⇒ cc1
add    rc,rd       ⇒ r1
add    re,rf       ⇒ r2
i2i_LT cc1,r1,r2   ⇒ ra
```
Conditional Move
</td>
<td>

```
        cmp_LT rx,ry ⇒ r1
        not    r1    ⇒ r2
(r1)? add    rc,rd ⇒ ra
(r2)? add    re,rf ⇒ ra
```
Predicated Execution
</td>
</tr>
</table>

Source Code	$x \leftarrow a < b \wedge c < d$	
ILOC Code	`comp  ` r_a, r_b $\Rightarrow cc_1$ `cbr_LT ` cc_1 $\rightarrow L_1, L_2$ L_1: `comp  ` r_c, r_d $\Rightarrow cc_2$ `cbr_LT ` cc_2 $\rightarrow L_3, L_2$ L_2: `loadI  false` $\Rightarrow r_x$ `jumpI ` $\rightarrow L_{out}$ L_3: `loadI  true ` $\Rightarrow r_x$ `jumpI ` $\rightarrow L_{out}$ L_{out}: `nop` **Straight Condition Codes**	`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 $\Rightarrow r_x$ **Conditional Move**
		`cmp_LT ` r_a, r_b $\Rightarrow r_1$ `cmp_LT ` r_c, r_d $\Rightarrow r_2$ `and    ` r_1, r_2 $\Rightarrow r_x$ **Boolean Compare**
		`cmp_LT ` r_a, r_b $\Rightarrow r_1$ `cmp_LT ` r_c, r_d $\Rightarrow r_2$ `and    ` r_1, r_2 $\Rightarrow r_x$ **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.

```
1         cmpq   %rdi, %rsi
2         jl     _L1
3         leaq   (%rdx, %rcx), %rax
4         jmp    _L2
5  _L1:   leaq   (%r8, %r9), %rax
6  _L2:   nop
```

```
1         cmpq   %rdi, %rsi
2         setl   %al
3         cmpq   %rdx, %rcx
4         setl   %bl
5         and    %al, %bl
6         movzbq %bl, %rax
```

Storing and Accessing Arrays

So far, we have assumed that variables stored in memory contain scalar values
- many programs need arrays or similar structures
- locating and referencing an element of an array can be **surprisingly complex**

We will begin by looking at **one-dimensional** arrays, *i.e.*, vectors, and then generalize the approach to **multi-dimensional** arrays.

Referencing a Vector Element

One-dimensional arrays, or vectors, are typically stored in **contiguous memory**, so that the ith element immediately precedes the $i + 1^{st}$ element.

A vector `V[3...10]` leads to the following memory layout, where the number below a cell indicates its index in the vector.
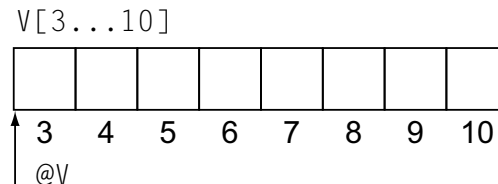
When the compiler encounters a reference, it must use the **index** into the vector, along with **facts** available from the declaration of `V`, to generate an offset for the reference.

The actual address is then computed as the **sum** of the offset and a pointer to the start of `V`, which we write as `@V`.

Referencing a Vector Element

Assume that a vector V has been declared as V[*low...high*], where *low* and *high* are the vector's lower and upper bounds.

To translate the reference V[i], the compiler needs both a pointer to the start of storage for V and the offset of element i within V.

The offset is simply $(\texttt{i} - \textit{low}) \times w$, where w is the length of a single element of V.

Example   If w is 4, the offset of V[6] in V[3...10] is: $(6 - 3) \times 4 = 12$.

Referencing a Vector Element

Assuming that r_i holds the value of i, the following code fragment computes the address of $V[i]$ into r_3 and loads its value into r_V.

```
loadI  @V         ⇒ r@V      …get V's address
subI   ri, 3      ⇒ r1       …offset − lower bound
multI  r1, 4      ⇒ r2       …× element length (4)
add    r@V, r2    ⇒ r3       …address of V[i]
load   r3         ⇒ rV       …value of V[i]
```

Notice that the simple reference $V[i]$ introduces **three** arithmetic operations. The compiler can improve this sequence.

Referencing a Vector Element

If w is a power of two, the multiply can be replaced with an **arithmetic shift**. Many base types in real programming languages have this property.

Even though adding the address and offset is unavoidable, most processors include an **addressing mode** specifically designed for this use case.

```
loadI    @V         ⇒ r@V      …get V's address
subI     ri, 3      ⇒ r1       …offset − lower bound
lshiftI  r1, 2      ⇒ r2       …× element length (4)
loadAO   r@V, r2    ⇒ rV       …value of V[i]
```

Example   In the Intel x86-64 ISA, move has an **offset-scaled-base-relative** addressing mode. The instruction `movq -16(%rbx, %rcx, 8), %rax` loads the value at the address $-16 + \%rbx + \%rcx \times 8$ into register `%rax`.

Referencing a Vector Element

Using a lower bound of zero eliminates the subtraction

- if the compiler knows the lower bound of V, it can fold the subtraction into @V
- instead of using @V as the base address for V, it can also use $V_0 = @V - low \times w$
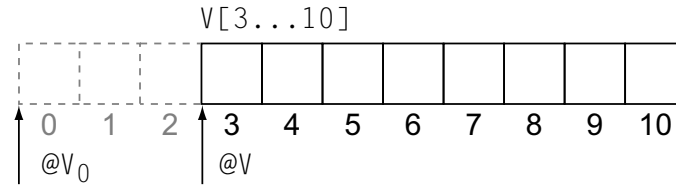
False Zero

The **false zero** of a vector V is the address where V[0] would be. In multiple dimensions, it is the location of a zero in each dimension.

Referencing a Vector Element

Using $@V_0$ and assuming that i is in r_i, the code for accessing $V[i]$ becomes shorter and, presumably, faster.

$$
\begin{array}{llll}
\texttt{loadI} & \texttt{@V}_0 & \Rightarrow r_{@V_0} & \text{...adjusted address of V} \\
\texttt{lshiftI} & r_i, 2 & \Rightarrow r_1 & \text{...} \times \text{ element length (4)} \\
\texttt{loadAO} & r_{@V_0}, r_1 \Rightarrow r_V & \text{...value of } V[i]
\end{array}
$$

Note   In a compiler, the longer sequence may produce better results by exposing details such as the multiply and add instruction to optimization.

An alternative strategy, employed in languages like C or Java, forces the use of zero as a lower bound, which ensures that $@V_0 = @V$ and simplifies all array-address calculations.

Array Storage Layout

Accessing an element of a multi-dimensional array requires more work. The code that the compiler needs to generate depends on the mapping of array indices to memory locations.

Most implementations use one of three schemes
- row-major order
- column-major order
- indirection vectors

The source-language definition usually specifies one of these mappings.

Array Storage Layout

Example Consider the array `A[1...2,1...4]`. Conceptually, it looks as follows.

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

In **row-major order**, the elements of a are mapped onto consecutive memory locations so that adjacent elements of a single row occupy consecutive memory locations.

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

Array Storage Layout

The array storage layout has an impact on **memory access patterns** and, thus, run-time performance.

Example Consider the following nested loops.

```
for i ← 1 to 2
    for j ← 1 to 4
        A[i,j] ← A[i,j] + 1
```

In row-major order, the assignment statement steps through memory in **sequential order**, beginning with A[1,1], A[1,2], A[1,3], and on through A[2,4].

Switching the two loops produces an access pattern that jumps between rows. For arrays larger than the cache, lack of sequential access can lead to poor run-time performance.

Array Storage Layout

The obvious alternative to row-major order is **column-major order**. It keeps the columns of a in contiguous locations, producing the following layout.

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

While row-major order leads to sequential access if the **rightmost** subscript varies fastest, column-major order leads to sequential access if the **leftmost** subscript varies fastest.

> ### Row-major vs. column-major order
>
> For languages that store arrays in contiguous storage, row-major order has been the typical choice. The one notable exception is FORTRAN, which uses column-major order.

Array Storage Layout

A third alternative are **indirection vectors**. This scheme reduces all multi-dimensional arrays to a set of (one-dimensional) vectors.
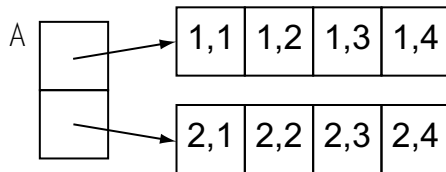
Indirection vectors appear simple, but they introduce their own complexity
- need more storage than either of the contiguous storage schemes
- require that application initializes all of the indirection pointers at runtime

An advantage of indirection vectors is that they can easily represent **ragged arrays**.

Note Java supports indirection vectors.

Referencing an Array Element

Programs that use arrays typically contain **references** to individual array elements.

As with vectors, the compiler must translate an array reference into a base address for the array's storage and an offset where the element is located relative to the starting address.

We will look at the following cases
- row-major order
- indirection vectors
- array-valued parameters

Note   The calculations for column-major order follow the same basic scheme as those for row-major order, with the dimensions reversed.

Row-Major Order

Before deriving the corresponding formula, we introduce the following notation
- low_i and $high_i$ denote *low* and *high* of the i-th dimension, respectively
- $len_i = high_i - low_i + 1$ denotes the length of the i-th dimension

In row-major order, the address calculation must find the start of the row and then generate an offset within the row as if it were a vector.

To access element `A[i,j]` of a two-dimensional array the compiler therefore must emit code that computes the…
- address of row `i`: $(i - low_1) \times len_2 \times w$
- offset of element `j`: $(j - low_2) \times w$

Putting all parts together, the resulting address computation is as follows.

$$@A + (i - low_1) \times len_2 \times w + (j - low_2) \times w$$

Row-Major Order

Example In the array `A[1...2,1...4]`, element `A[2,3]` lies at offset

$$(2 - 1) \times (4 - 1 + 1) \times 4 + (3 - 1) \times 4 = 24$$

from `A[1,1]`, assuming that `@A` points at `A[1,1]` at offset 0.

Looking at `A` in memory, we find that the address of `A[1,1]` + 24 is, in fact, the address of `A[2,3]`.

0	4	8	12	16	20	24	28
1,1	1,2	1,3	1,4	2,1	2,2	2,3	2,4

`@A`   `A[2,3]`

Row-Major Order

As in the vector case, we can simplify the address calculation if upper and lower bounds are known at compile time by introducing a false zero.

$$@A + (i - low_1) \times len_2 \times w + (j - low_2) \times w$$

$$@A + (i \times len_2 \times w) - (low_1 \times len_2 \times w) + (j \times w) - (low_2 \times w)$$

$$@A + (i \times len_2 \times w) + (j \times w) - (low_1 \times len_2 \times w + low_2 \times w)$$

The term $(low_1 \times len_2 \times w + low_2 \times w)$ is independent of i and j. Thus, it can be factored directly into the base address: $@A_0 = @A - (low_1 \times len_2 \times w + low_2 \times w) = @A - 20$.

Now, the array reference is simply $@A_0 + i \times len_2 \times w + j \times w$. Finally, we can refactor and move w outside, saving an extraneous multiplication.

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

Row-Major Order

Assuming that `i` and `j` are in r_i and r_j, and that len_2 is a constant, this form of the polynomial leads to the following code sequence.

```
loadI   @A₀          ⇒ r@A₀      …adjusted base address of A
multI   rᵢ, len₂     ⇒ r₁        …i × len₂
add     r₁, rⱼ       ⇒ r₂        …+ j
multI   r₂, 4        ⇒ r₃        …× element length (4)
loadAO  r@A₀, r₃     ⇒ rₐ        …value of A[i,j]
```

Note   The computation takes two multiplications and two additions (one in the `loadAO`). The second multiply can be rewritten as a shift.

Row-Major Order

If the compiler does not have access to the array bounds
- compute the false zero at runtime
- use the more complex polynomial with subtractions to adjust for lower bounds

The first option only makes sense if the elements of the array are accessed multiple times in a procedure.

Note   These ideas behind address computation for two-dimensional arrays generalize to **arrays of higher dimension**. The address polynomial for an array stored in **column-major order** can be derived in a similar fashion.

Indirection Vectors

Using indirection vectors simplifies the code generated to access an individual element. For an n-dimensional array, the first $n-1$ steps navigate the indirection vectors, while the last step performs a regular (one-dimensional) vector lookup.

Example Accessing element `B[i,j,k]` in the array `B[1...2,1...3,1...4]`

1. use `@B0`, `i`, and the length of a pointer, to find the vector for the subarray `B[i,*,*]`
2. use result, along with `j` and the length of a pointer to find the vector for the subarray `B[i,j,*]`
3. use base address in the vector-address computation with `k` and element length *w* to find the address of `B[i,j,k]`
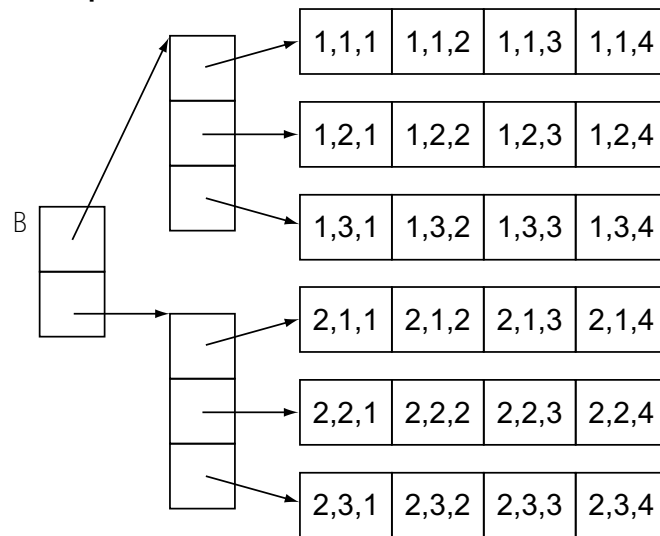
Indirection Vectors

If the values for i, j, and k exist in registers r_i, r_j, and r_k, respectively, and $@B_0$ is the zero-adjusted address of the first dimension, then $B[i,j,k]$ can be loaded as follows.

```
loadI   @B₀              ⇒ r@B₀    …adjusted base address of B
multI   rᵢ, 4            ⇒ r₁      …pointer size (4)
loadAO  r@B₀, r₁         ⇒ r₂      …load @B[i,*,*]

multI   rⱼ, 4            ⇒ r₃      …pointer size (4)
loadAO  r₂, r₃           ⇒ r₄      …load @B[i,j,*]

multI   rₖ, 4            ⇒ r₅      …element size (4)
loadAO  r₄, r₅           ⇒ rᵦ      …value of B[i,j,k]
```

Note   This code assumes that the pointers in the indirection structure have already been adjusted to account for non-zero lower bounds.

Accessing Array-Valued Parameters

Note   Even in languages that use call by value for all other parameters, arrays are usually passed by reference, for the sake of runtime performance.

To generate array references in the callee, we need information about the dimensions of the array that is bound to the parameter.

- some languages, *e.g.*, FORTRAN, Oberon-0, give the **programmer** responsibility for passing the information to the callee needed to address a parameter array correctly
- other languages leave the task of collecting, organizing, and passing the necessary information to the **compiler**

Accessing Array-Valued Parameters

> **Dope Vector**
>
> A descriptor for an actual parameter array that contains both a pointer to the **start of the array** and the **necessary information for each dimension**. Dope vectors may also be used for arrays whose bounds are determined at runtime.

Caller

- builds the dope vector and stores it in callee's AR
- value passed in the array's parameter slot is a pointer to the dope vector

Callee

- loads values out of the dope vector as needed
- generates array reference using the same polynomial as for a local array

Range Checking

A program that references an out-of-bounds array element is not well formed
- some languages, *e.g.*, Ada, Java, require such accesses to be detected and reported
- compilers for other languages may include optional mechanisms for range checking

The simplest implementation of **range checking** inserts a check before each array access
- introduces additional overhead, which may be significant in some programs
- may need more information in dope vector, *e.g.*, explicit bounds rather than length

An optimizing compiler can improve on this naive implementation in several ways
- combine checks
- move checks out of loops
- prove checks redundant

Character Strings

The operations that programming languages provide for **character data** are different from those provided for numerical data.

Programming language support for character strings varies
- in C, most manipulation takes the form of calls to library routines
- PL/I provides first-class mechanisms to assign individual characters, specify arbitrary substrings, and concatenate strings

CISC architectures provide extensive support for string manipulation, whereas RISC machines rely on the compiler to code these operations using a set of simpler instructions.
- string assignment
- string concatenation
- string-length computation

String Representation

The string representation chosen by the compiler writer has a strong impact on the cost of string operations.

a	~~∅~~	s	t	r	i	n	g	\0	

↑
@b

8	a	~~∅~~	s	t	r	i	n	g

↑
@b
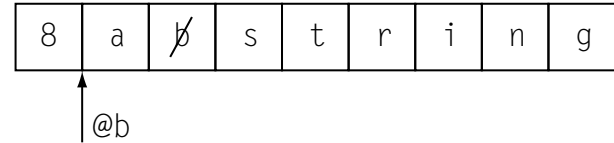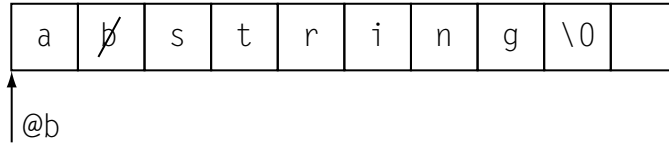
- C traditionally uses **null termination** (left) with a designated character ($\backslash 0$) serving as a terminator
- many other language implementations use an **explicit length field** (right) that stores the length of the string (8) alongside its contents

If the length field takes more space than the null terminator, then storing the length will marginally increase the size of the string in memory. However, storing the length simplifies several operations on strings.

String Assignment

String assignment is conceptually simple. The complexity of its implementation depends on whether or not the target machine supports **character-sized memory operations**.

Example   In C, an assignment from the third character of b to the second character of a can be written as `a[1] = b[2];`.

With support for character-sized memory operations, this assignment translates into the following simple code.

```
loadI     @b      ⇒ r@b
cloadAI   r@b, 2  ⇒ r1
loadI     r@a     ⇒ r@a
cstoreAI  r1      ⇒ r@a, 1
```

Without hardware support for character-sized memory operations, the compiler must generate more complex code that **masks** the characters explicitly.

String Assignment

Assumptions Strings a and b begin on word boundaries, a character occupies 1 byte, and that a word is 4 bytes.

```
loadI    0x0000FF00  ⇒  r_C2       …mask for the character 2
loadI    0xFF00FFFF  ⇒  r_C124     …mask for characters 1, 2, and 4

loadI    @b          ⇒  r_@b       …load address of b
load     r_@b        ⇒  r_1        …load first word of b
and      r1, r_C2    ⇒  r_2        …mask away other characters
lshiftI  r_2, 8      ⇒  r_3        …move character to first byte
loadI    @a          ⇒  r_@a       …load address of a
load     r_@a        ⇒  r_4        …load first word of a
and      r_4, r_C124 ⇒  r_5        …mask away second character
or       r_3, r_5    ⇒  r_6        …insert new second character
store    r_6         ⇒  r_@a       …store result
```

String Assignment

The code is similar for longer strings. The following code for `a = b;` assumes explicit length representation and hardware support for character-sized memory operations.

```
        loadI    @b        ⇒  r_@b           …load address of b
        loadAI   r_@b, -4  ⇒  r_1            …load b's length
        loadI    @a        ⇒  r_@a           …load address of a
        loadAI   r_@a, -4  ⇒  r_2            …load a's length
        cmp_LT   r_2, r_1  ⇒  r_3            …does b fit into a?
        cbr      r_3       →  L_SOV, L_1     …raise string overflow
L_1:    loadI    0         ⇒  r_4            …initialize counter
        cmp_LT   r_4, r_1  ⇒  r_5            …more to copy?
        cbr      r_5       →  L_2, L_3       …jump
L_2:    cloadAO  r_@b, r_4 ⇒  r_6            …load character from b
        cstoreAO r_6       ⇒  r_@a, r_4      …store character in a
        addI     r_4, 1    ⇒  r_4            …increment counter
        cmp_LT   r_4, r_1  ⇒  r_7            …more to copy?
        cbr      r_7       →  L_2, L_3       …jump
L_3:    storeAI  r_1       ⇒  r_@a, -4       …set length of a
```

String Assignment

In C, which uses null termination for strings, the same assignment would be written as a character-copying loop.

```
1  t1 = a;
2  t2 = b;
3  do {
4      *t1++ = *t2++;
5  } while (*t2 != '\0')
```

	loadI	@b	$\Rightarrow$	$r_{@b}$	…load address of b
	loadI	@a	$\Rightarrow$	$r_{@a}$	…load address of a
	loadI	NULL	$\Rightarrow$	r_1	…load terminator
L_1:	cload	$r_{@b}$	$\Rightarrow$	r_2	…load character
	cstore	r_2	$\Rightarrow$	$r_{@a}$	…store character
	addI	$r_{@b}$, 1	$\Rightarrow$	$r_{@b}$	…move pointer
	addI	$r_{@a}$, 1	$\Rightarrow$	$r_{@a}$	…move pointer
	cmp_NE	r_1, r_2	$\Rightarrow$	r_3	…more to copy?
	cbr	r_3	$\rightarrow$	L_1, L_2	…jump
L_2:	nop				…next statement

String Assignment

Further Possibilities

- if the target machine supports **auto-increment** on load and store operations, the two adds in the loop can be performed in the cload and cstore operations
- the compiler could generate **whole-word** loads and stores, followed by a character-oriented loop to handle any leftover characters at the end of the string
- the compiler might also call a carefully optimized **library routine** to implement the nontrivial cases

String Concatenation

Concatenation is simply a shorthand for a sequence of one or more assignments. It comes in **two basic forms**.

1. Appending string b to string a
- compiler emits code to determine the length of a
- space permitting, it assigns b to the space that immediately follows a

2. Creating a new string that contains a followed immediately by b
- requires copying each character in a and each character in b
- can be treated as a pair of assignments in code generation

The compiler should ensure that **enough space** is allocated to hold the result. If the lengths of a and b are unknown at compile time, runtime code needs to be generated
- to compute the lengths of the strings
- to perform corresponding test and branch

String Length

Programs that manipulate strings often need to compute a character string's length.

Different string representations lead to different costs for the length computation
- a **null-terminated string** requires time proportional to the length of the string
- with an **explicit length field** the cost is constant and small

Tradeoff   Null termination saves a small amount of space, but requires more code and more time for the length computation. An explicit length field costs one more word per string, but makes the length computation take constant time.

Example   With explicit length fields a statement `length(a + b)` can be optimized to two `loads` and one `add`.

String Length on Intel x86-64

The Intel x86-64 ISA provides several **dedicated operations** for string manipulation.

- `movs`, `lods`, `stos`: move, load, and store strings
- `cmps`: compare strings
- `scas`: scan strings

These instructions are combined with **repetition prefixes**: `rep`, `repe`, `repne`, `repz`, and `repnz`).

```
1  movq   -1, %rcx      # "clear" counting register
2  movq   %rdi, %rsi    # backup %rdi
3  mov    0, %al        # look for \0
4  repne  scasb         # actually do the search
5  subq   %rsi, %rdi    # save the string length
6  decq   %rdi          # do not count the \0 in the string length
7  movq   %rdi, %rax    # save the return value
```

Structure References

Most programming languages provide a mechanism to aggregate data together into a **structure**.

Example   In C, we could use the following structure to create lists of integers.

```
1  struct node {
2      int value;
3      struct node *next;
4  };
5  struct node NILNode = { 0, (struct node*) 0 };
6  struct node *NIL = &NILNode;
```

The introduction of structures and pointers creates two distinct problems for the compiler: **anonymous values** and **structure layout**.

Understanding Structure Layouts

To emit code for structure references, the compiler needs to know
- **starting address** of the structure instance
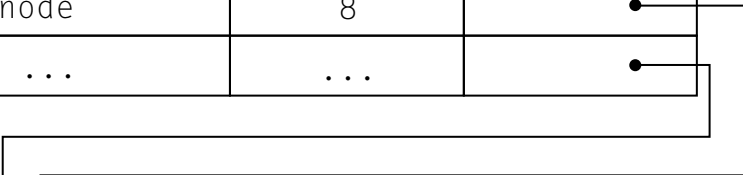- **offset** and **length** of each structure element

The compiler can build a separate **table of structure layouts** to maintain this information.
- textual name for each structure element
- its offset within the structure
- its source-language data type

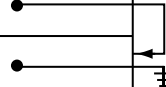# Understanding Structure Layouts

Structure Layout Table

Name	Length	1st Element
node	8	●
...	...	●

Structure Element Table

Name	Length	Offset	Type	Next
node.value	4	0	int	●
node.next	4	4	struct node *	●
...	...	...	...	...

Understanding Structure Layouts

Note Entries in the **structure element table** (shown on the previous slide) use fully qualified names to avoid conflicts due to reuse of a name in several distinct structures.

With this information, the compiler can easily generate code for structure references.

Example The compiler might translate the reference `p1->next`, for a pointer to node `p1`, into the following ILOC code.

```
loadI   4          ⇒  r₁    …offset of next
loadAO  r_p1, r₁ ⇒  r₂    …value of p1->next
```
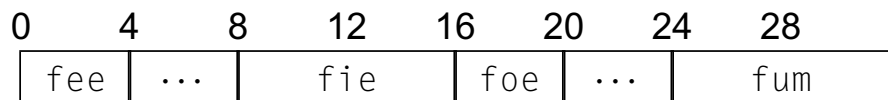
Understanding Structure Layouts

In laying out a structure and assigning offsets to its elements, the compiler must obey the **alignment rules** of the target architecture.

```
1  struct example {
2      int fee;
3      double fie;
4      int foe;
5      double fum;
6  };
```

0	4	8	12	16	20	24	28
fee	···	fie		foe	···	fum	

Elements in Declaration Order

0	4	8	12	16	16
fie		fum		fee	foe
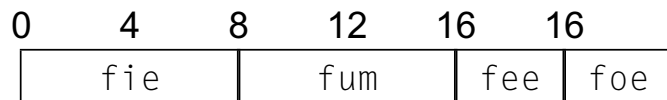
Elements Ordered by Alignment

Note Whether the compiler can reorder structure elements in memory arbitrarily depends whether the source language definition exposes the layout of a structure to the user.

Array of Structures

Many programming languages allow the user to declare an **array of structures**. The compiler has two options how to layout such data in memory.
1. a structure-valued array with multiple copies of the structure layout
2. a structure composed of elements that are arrays

Note   The second option is only possible if the programmer cannot take the address of a structure-valued element of an array.

Depending on how the surrounding code accesses the data, these two strategies may have strikingly **different performance** on a system with cache memory.

Unions and Runtime Tags

Many languages support structures with **multiple**, **data-dependent** interpretations.

Unions and variants present one additional complication
- the possibility exists that element names are **not unique**
- compiler must resolve each reference to a unique offset and type in the runtime object

```
1  struct n1 {
2     int kind;
3     int value;
4  };
```

```
1  struct n2 {
2     int kind;
3     float value;
4  };
```

```
1  union one {
2     struct n1 inode;
3     struct n2 fnode;
4  } u1;
```

This problem has a **linguistic solution**: the programming language can force the programmer to make the reference unambiguous.

Unions and Runtime Tags

Example   To reference an integer value, the programmer specifies `u1.inode.value`, whereas to reference a floating-point value, the programmer specifies `u1.fnode.value`.

The **fully qualified name** resolves any ambiguity.

The same mechanism is used to disambiguate references to implicitly defined structure-valued union elements.

Example   Union `two` has the same properties as `one`.

Again, the programmer has to specify a fully qualified name such as `u2.inode.value` to reference an integer value and `u2.fnode.value` to reference a floating-point value.

```
1  union two {
2     struct {
3        int kind;
4        int value;
5     } inode;
6     struct {
7        int kind;
8        float value;
9     } fnode;
10 } u2;
```

Unions and Runtime Tags

As an alternative to linguistic solutions, some systems rely on **runtime discrimination**.

- each variant in the union has a field (tag) that distinguishes it from all other variants
- compiler emits code to check the value of the tag field and handle object accordingly

```
1  type shapeKind = (square, rectangle, circle);
2      shape = record
3                  centerx : integer;
4                  centery : integer;
5                  case kind : shapeKind of
6                      square : (side : integer);
7                      rectangle : (length, height : integer);
8                      circle : (radius : integer);
9              end;
```

Pointers and Anonymous Values

A C program creates an instance of a structure in one of two ways
- declare a structure instance, *e.g.*, `NILNode` in the earlier example
- allocate a structure instance explicitly

Example For a variable declared as a pointer to `node`, the allocation looks as follows.

```
1  fee = (struct node*) malloc(sizeof(node));
```

The only access to this new node is through the pointer `fee`. Thus, we think of it as an **anonymous value**, since it has no permanent name.

Because the only name for an anonymous value is a pointer, the compiler cannot easily determine if two pointer references specify the **same** memory location.

Pointers and Anonymous Values

Example   Consider the following code fragment.

```
1  p1 = (node*) malloc(sizeof(node));
2  p2 = (node*) malloc(sizeof(node));
3  if (...) {
4      p3 = p1;
5  } else {
6      p3 = p2;
7  }
8  p1->value = ...;
9  p3->value = ...;
10 ... = p1->value;
```

The first two lines create anonymous nodes.

Line 8 writes through p1, while line 9 writes through p3.

On line 9, p3 can refer to either the node allocated in line 1 or in line 2.

Finally, line 10 references p1->value. The compiler cannot easily determine at this point which value is used.

The uncertainty introduced by pointers **prevents** the compiler from keeping values used in pointer-based references in registers.

Pointers and Anonymous Values

Anonymous objects further complicate the problem because they introduce an **unbounded set** of objects to track.

A similar effect occurs with arrays: the compiler would need to perform in-depth analysis of array subscripts to determine whether two array references overlap.

While challenging, the problem of disambiguating array references is easier than the problem of disambiguating pointer references.

Analysis to disambiguate pointer references and array references is a **major** source of potential improvement in program performance.
- interprocedural data-flow analysis to discover all objects each pointer can point to
- data-dependence analysis to understand the patterns of array references

Control-Flow Constructs

As the compiler generates code, it can build up **basic blocks** by simply aggregating consecutive, unlabeled, non-control-flow operations.

To tie a set of blocks together so that they form a procedure, the compiler must insert code that implements the **control-flow operations** of the source program.
- build a **control-flow graph** and use it for analysis, optimization, and code generation
- code for control-flow constructs resides **at or near the end** of each basic block

While many different syntactic conventions have been used to express control flow, the number of underlying concepts is **small**.

Conditional Execution

Most programming languages provide some version of an `if-then-else` construct.

```
if expr
    then statement₁
    else statement₂
statement₃
```

The compiler needs to generate code that
- evaluates *expr* and branches to *statement₁* or *statement₂*, accordingly
- implements the two statements and at the end jumps to *statement₃*

As we already saw on Slide 355, the compiler has many options for implementing `if-then-else` constructs.

Conditional Execution

With trivial `then` and `else` parts the primary consideration for the compiler is matching the expression evaluation to the underlying hardware.

As the `then` and `else` parts grow, the importance of efficient execution inside the `then` and `else` parts begins to **outweigh** the cost of executing the controlling expression.

Example   On a machine that supports **predicated execution**, using predicates for large blocks in the `then` and `else` parts can waste execution cycles.

With large blocks of code under both the `then` and `else` parts, the cost of unexecuted instructions may outweigh the overhead of using a conditional branch.

Unit 1	Unit 2
comparison $\Rightarrow r_1$	
(r_1) op_1	$(\neg r_1)$ op_{11}
(r_1) op_2	$(\neg r_1)$ op_{12}
(r_1) op_3	$(\neg r_1)$ op_{13}
(r_1) op_4	$(\neg r_1)$ op_{14}
(r_1) op_5	$(\neg r_1)$ op_{15}
(r_1) op_6	$(\neg r_1)$ op_{16}
(r_1) op_7	$(\neg r_1)$ op_{17}
(r_1) op_8	$(\neg r_1)$ op_{18}
(r_1) op_9	$(\neg r_1)$ op_{19}
(r_1) op_{10}	$(\neg r_1)$ op_{20}

Unit 1	Unit 2
compare and branch	
L_1: op_1	op_2
op_3	op_4
op_5	op_6
op_7	op_8
op_9	op_{10}
jumpI $\rightarrow$ L_3	
L_2: op_{11}	op_{12}
op_{13}	op_{14}
op_{15}	op_{16}
op_{17}	op_{18}
op_{19}	op_{20}
jumpI $\rightarrow$ L_3	
L_3: nop	

Conditional Execution

Choosing between branching and predication to implement an `if-then-else` requires some care. Several issues need to be considered.

1. **Expected frequency of execution**: if one path executes significantly more often, techniques that speed up its execution (*e.g.*, branch prediction, speculative execution, and instruction reordering) may produce faster code
2. **Uneven amounts of code**: if one path contains many more instructions, this may weigh against predication or for a combination of predication and branching
3. **Control flow inside the construct**: if either path contains nontrivial control flow, in particular nested `if`-statements, then predication may be a poor choice

Conditional Execution

Predication in the Real World

Nvidia's CUDA parallel computing platform 32 threads are executed together as a **warp**. All threads in a warp execute the **same** instruction at the **same** time.

In order to support **different** threads doing **different** things, CUDA has predicated instructions that are executed only if a logical flag is `true`.

```
if (x < 0.0)                          p = (x < 0.0);
    r = -1.0;                      p:  r = -1.0;
else                              !p:  r = sqrt(x);
    r = sqrt(x);
```

This is called **warp divergence**. The performance of the code generated by the `nvcc` compiler is impacted by the same issues as outlined on the previous slide.

Loops and Iteration

Most programming languages include `loop` constructs to perform iteration.

The first FORTRAN compiler introduced the `do` loop to perform iteration. Today, loops are found in many forms. For the most part, they have a similar structure.

Example   Consider the C `for` loop, which has three controlling expressions.

e_1 performs initialization

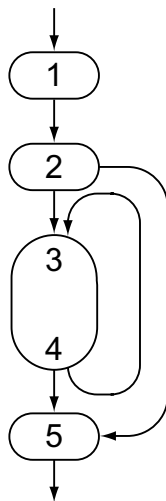e_2 evaluates to a Boolean and governs execution of the loop

e_3 executes at the end of each iteration and, potentially, updates the values used in e_2

```
for (e₁; e₂; e₃) {
    loop body
}
```

Loops and Iteration

The following schema shows how the compiler might lay out the code.

```
for (e₁; e₂; e₃) {
    loop body
}
```


Step	Action
1	evaluate e_1
2	if ($\neg e_2$) then goto 5
3	*loop body*
4	evaluate e_3 if (e_2) then goto 5
5	*code following loop*

We will use this basic schema to explain the implementation of several kinds of loops.

Loops and Iteration

If the loop body consists of a single basic block, *i.e.*, contains no other control flow, then the loop that results from this schema has **an initial branch** plus **one branch per iteration**.

The compiler can hide the latency of this branch in several ways
- **branch hint prefix**: mark the branch in Step 4 as likely to be taken
- **delay slot**: move instructions from loop body into the branch delay slot(s)
- **loop unrolling**: if number of iterations is predictable, combine multiple iterations

Delay Slot

Some computer architectures feature **delay slots** to mask the latency of time-consuming instructions, such as loads and branches. A delay slot is an instruction that gets executed without the effects of a preceding instruction.

Loops and Iteration

Branch Prediction in Intel x86-64

In the past, Intel's ISA supported static branch prediction prefixes (2EH and 3EH), a feature that has been abandoned since.

Modern Intel CPUs use a Branch Target Buffer (BTB) to manage branching history dynamically. If this information is not available, the CPU uses a simple heuristic:
- conditional forward branches are predicted as **not being taken**
- conditional backward branches are predicted as **being taken**

This heuristic fits well with the basic schema that we will use to implement loops.

For Loops

To map a `for` loop into code, the compiler follows the general schema from before.

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

```
        loadI   1          ⇒  rᵢ          …Step 1
        loadI   100        ⇒  r₁          …Step 2
        cmp_GT  rᵢ, r₁     ⇒  r₂
        cbr     r₂          →  L₁, L₂
L₁:   loop body                            …Step 3
        addI    rᵢ, 1      ⇒  rᵢ          …Step 4
        cmp_LE  rᵢ, r₁     ⇒  r₃
        cbr     r₃          →  L₁, L₂
L₂:   next statement                       …Step 5
```

The compiler can also shape the loop to have only **one copy** of the test. In this form, Step 4 evaluates e_3 and then jumps to Step 2, *i.e.*, replace `cmp_LE` and `cbr` with `jumpI`.

For Loops

Overall, this form of the loop is **one operation smaller** than the two-test form.

However, it creates a **two-block loop** for even the simplest loops, and it **lengthens the path** through the loop by at least one operation.

```
        loadI   1       ⇒ rᵢ          …Step 1
        loadI   100     ⇒ r₁          …Step 2
    L₁: cmp_GT  rᵢ, r₁  ⇒ r₂
        cbr     r₂      → L₁, L₂
        loop body                     …Step 3
        addI    rᵢ, 1   ⇒ rᵢ          …Step 4
        jumpI           → L₁
    L₂: next statement                …Step 5
```

This more compact loop form is only appropriate if **code size** is a serious consideration.

While Loops

A `while` loop can also be implemented with the loop schema. Since it has no initialization, the code is even more compact.

```
while (x < y) {
    loop body
}
next statement
```

$$\begin{array}{llll}
& \text{cmp\_LT} & r_x, r_y \Rightarrow r_1 & \dots\text{Step 2} \\
& \text{cbr} & r_1 \quad\quad \rightarrow L_1, L_2 & \\
L_1: & \textit{loop body} & & \dots\text{Step 3} \\
& \text{cmp\_LT} & r_x, r_y \Rightarrow r_2 & \dots\text{Step 4} \\
& \text{cbr} & r_2 \quad\quad \rightarrow L_1, L_2 & \\
L_2: & \textit{next statement} & & \dots\text{Step 5}
\end{array}$$

Replicating the test in Step 4 creates the possibility of a loop with a **single basic block**.

Until Loops

An `until` loop iterates as long as the controlling expression is `false`.

It checks the controlling expression **after** each iteration. Thus, it always enters the loop and performs at least one iteration and produces a particularly simple loop structure.

```
{
    loop body
} until (x < y)
next statement
```

$L_1:$ *loop body*   ...Step 3
        `cmp_LT` $r_x, r_y \Rightarrow r_1$ ...Step 4
        `cbr` r_1 $\rightarrow$ L_1, L_2
$L_2:$ *next statement*   ...Step 5

Note   The `do` loop known from C, C++, and Java is similar to the `until` loop with the difference that it iterates as long as the controlling expression is `true`.

Break and Exit

Several languages provide a **structured way** to exit a control-flow construct.

In a loop, `break` transfers control to the first statement following the loop. For nested loops, a `break` typically exits the innermost loop.

Some languages, *e.g.*, Ada and Java, use labels to control which enclosing constructs will exited by the `break` statement.

C also uses `break` in `switch` statements to transfer control to the statement that follows the `switch` statement.

All of these actions have simple implementations
- each loop and each case statement ends with a label for the statement that follows it
- a `break` can be implemented as an immediate jump to that label

Skip and Continue

Some languages include a `skip` or `continue` statement that jumps to the next iteration of a loop.

This construct can be implemented in two ways
- **immediate jump** to the code that reevaluates the controlling expression, tests its value, and branches accordingly
- **insert a copy** of the evaluation, test, and branch at the point where the skip occurs

Case Statements

Many programming languages include some variant of a case statement.

The basic strategy is straightforward
1. evaluate the controlling expression
2. branch to the selected case
3. execute the code for that case

Steps 1 and 3 are well understood. The complex part of case-statement implementation lies in choosing an efficient method to locate the designated case.

No single method works well for all case statements. We examine three strategies: a **linear search**, a **binary search**, and a **computed address**.

Linear Search

The simplest way to locate the appropriate case is to treat the case statement as the specification for a nested set of `if-then-else` statements.

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

$t_1 \leftarrow e$
if (t_1 = 0)
    then $block_0$
    else if (t_1 = 1)
       then $block_1$
       else if (t_1 = 3)
          then $block_3$
          else $block_d$

The compiler should order cases according to estimated execution frequency.

Binary Search

As the number of cases rises, the efficiency of linear search becomes a problem.

If the compiler can impose an order on the case labels, it can use **binary search** to obtain a logarithmic search rather than a linear one.
- build a compact ordered table of case labels and corresponding branch labels
- use binary search to discover a matching case label, or the absence of a match
- finally, either branch to the corresponding label or to the default case.

Binary Search

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

$t_1 \leftarrow e$

$down \leftarrow 0$ ...*lower bound*

$up \leftarrow 10$ ...*upper bound + 1*

while (down + 1 < up) {

     middle ← (up + down) ÷ 2

     if (Value[middle] $\leq$ t_1)

        then down ← middle

        else up ← middle

}

if (Value[down] = t_1)

     then *jump to* Label[down]

     else *jump to* LB_d

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}

Computing the Address Directly

If the case labels form a compact set, the compiler can do better than binary search.

In this case, the compiler can build a compact vector, or **jump table**, that contains the block labels, and find the appropriate label by index into the table.
- for a dense label set, this scheme generates compact and efficient code
- the cost is small and constant: a brief calculation, a memory reference, and a `jump`
- if a few holes exist in the label set, they can be filled with the label for the default case

Computing the Address Directly

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

$t_1 \leftarrow e$
if $(t_1 < 0$ 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

Label

| |
|---|
| LB_0 |
| LB_1 |
| LB_2 |
| LB_3 |
| LB_4 |
| LB_5 |
| LB_6 |
| LB_7 |
| LB_8 |
| LB_9 |

Procedure Calls

The implementation of procedure calls is, for the most part, straightforward.

Recall: a procedure call consists of

- a **precall** and a **postreturn** sequence in the caller
- a **prologue** and an **epilogue** sequence in the callee

In the following, we focus on issues that affect the compiler's ability to generate **efficient**, **compact**, and **consistent** code for procedure calls.

Procedure p

Prologue
Precall
Postreturn
Epilogue

Procedure q

Call

Prologue
Epilogue
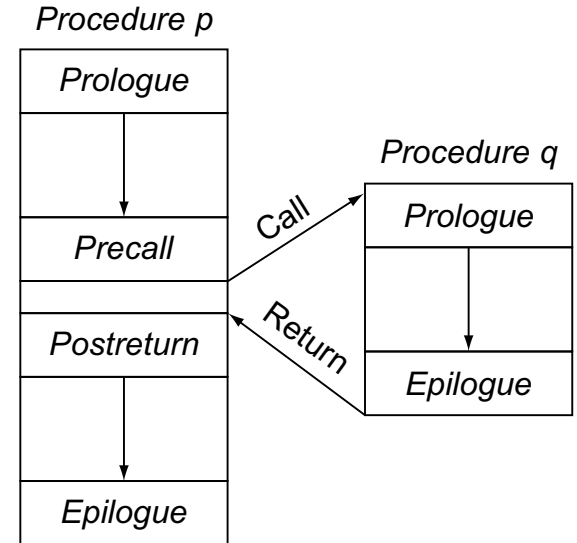
Return

As a general rule, moving operations from the precall and postreturn sequences into the prologue and epilogue sequences should **reduce the overall size** of the final code.

Evaluating Actual Parameters

When it builds the precall sequence, the compiler must emit code to evaluate the actual parameters to the call. The compiler treats each actual parameter as an expression.

- **call-by-value parameters**: evaluate the expression and store its value in location designated for that parameter, *i.e.*, a register or callee's AR
- **call-by-reference parameter**: evaluate the parameter to an address and store it in the location designated for that parameter

Note   If a call-by-reference parameter has no storage location, then the compiler may need to allocate space for that value so that it has an address to pass to the callee.

The compiler should use a consistent **evaluation order** for parameters, *i.e.*, either left to right or right to left, as evaluating parameters might have side effects.

Saving and Restoring Registers

As both the cost of memory operations and the number of registers have risen, the cost of **saving and restoring registers** has increased to the point that it needs careful attention.

1. **Using multi-register memory operations**: many ISAs support doubleword and quadword load and store operations for adjacent registers.
2. **Using a library routine**: replace the sequence of individual memory operations with a call to a compiler-supplied save or restore routine
3. **Combining responsibilities**: caller and callee pass a value back and forth that specifies which registers each must save