Appendix A

ILOC

CHAPTER OVERVIEW

ILOC is the assembly code for a simple abstract machine. It was originally designed as a low-level, linear IR for use in an optimizing compiler. We use it throughout the book as an example IR. We also use it as a simplified target language in the chapters that discuss code generation. This appendix serves as a reference on ILOC.

Keywords: Intermediate Representation, Three-Address Code, ILOC

A.1 INTRODUCTION

ILOC is the linear assembly code for a simple abstract RISC machine. The ILOC used in this book is a simplified version of the intermediate representation that was used in the Massively Scalar Compiler Project at Rice University. For example, ILOC as presented here assumes one generic data type, an integer without a specific length; in the compiler, the IR supported a broad variety of data types.

The ILOC abstract machine has an unlimited number of registers. It has three-address, register-to-register operations, load and store operations, comparisons, and branches. It supports just a few simple addressing modes—direct, address + offset, address + immediate, and immediate. Source operands are read at the beginning of the cycle when the operation issues. Result operands are defined at the end of the cycle in which the operation completes.

Other than its instruction set, the details of the machine are left unspecified. Most of the examples assume a simple machine, with a single functional unit that executes ILOC operations in their order of appearance. When other models are used, we discuss them explicitly.

Engineering a Compiler. DOI: 10.1016/B978-0-12-088478-0.00014-1

An ILOC program consists of a sequential list of instructions. Each instruction may be preceded by a label. A label is just a textual string; it is separated from the instruction by a colon. By convention, we limit ourselves to labels of the form [*a*–*z*] ([*a*–*z*] | [0–9] | –)*. If some instruction needs more than one label, we insert an instruction that only contains a `nop` before it, and place the extra label on the `nop`. To define an ILOC program more formally,

IlocProgram	→	*InstructionList*
InstructionList	→	*Instruction*
	|	`label` : *Instruction*
	|	*Instruction InstructionList*

Each instruction contains one or more operations. A single-operation instruction is written on a line of its own, while a multioperation instruction can span several lines. To group operations into a single instruction, we enclose them in square brackets and separate them with semicolons. More formally,

Instruction	→	*Operation*
	|	`[` *OperationList* `]`
OperationList	→	*Operation*
	|	*Operation* ; *OperationList*

An ILOC operation corresponds to a machine-level instruction that might be issued to a single functional unit in a single cycle. It has an opcode, a sequence of comma-separated source operands, and a sequence of comma-separated target operands. The sources are separated from the targets by the symbol ⇒, pronounced "into."

Operation	→	*NormalOp*
	|	*ControlFlowOp*
NormalOp	→	*Opcode OperandList* ⇒ *OperandList*
OperandList	→	*Operand*
	|	*Operand* , *OperandList*
Operand	→	`register`
	|	`num`
	|	`label`

The nonterminal *Opcode* can be any ILOC operation, except `cbr`, `jump`, and `jumpI`. Unfortunately, as in a real assembly language, the relationship between an opcode and the form of its operands is less than systematic. The easiest way to specify the form of the operands for each opcode is in

a tabular form. The tables that occur later in this appendix show the number of operands and their types for each ILOC opcode used in the book.

*Operand*s may be one of three types: `register`, `num`, and `label`. The type of each operand is determined by the opcode and the position of the operand in the operation. In the examples, we use both numerical (r_{10}) and symbolic (r_i) names for registers. Numbers are simple integers, signed if necessary. We always begin a label with an `l` to make its type obvious. This is a convention rather than a rule. ILOC simulators and tools should treat any string of the form described above as a potential label.

Most operations have a single target operand; some of the `store` operations have multiple target operands, as do the branches. For example, `storeAI` has a single source operand and two target operands. The source must be a register, and the targets must be a register and an immediate constant. Thus, the ILOC operation

$$\texttt{storeAI } r_i \Rightarrow r_j, 4$$

computes an address by adding 4 to the contents of r_j and stores the value found in r_i into the memory location specified by the address. In other words,

$$\text{MEMORY } (r_j + 4) \leftarrow \text{CONTENTS } (r_i)$$

Control-flow operations have a slightly different syntax. Since these operations do not define their targets, we write them with the single arrow $\rightarrow$, instead of $\Rightarrow$.

ControlFlowOp	$\rightarrow$	`cbr`	`register`	$\rightarrow$	`label , label`
	|	`jumpI`		$\rightarrow$	`label`
	|	`jump`		$\rightarrow$	`register`

The first operation, `cbr`, implements a conditional branch. The other two operations are unconditional branches, called jumps.

A.2 NAMING CONVENTIONS

The ILOC code in the text examples uses a simple set of naming conventions.

1. Memory offsets for variables are represented symbolically by prefixing the variable name with the @ character.

2. The user can assume an unlimited supply of registers. These are named with simple integers, as in r_{1776}, or with symbolic names, as in r_i.
3. The register r_{arp} is reserved as a pointer to the current activation record. Thus, the operation

   $$\texttt{loadAI}\ r_{arp}, @x \Rightarrow r_1$$

   loads the contents of the variable x, stored at offset @x from the ARP, into r_1.

ILOC comments begin with the string // and continue until the end of a line. We assume that these are stripped out by the scanner; thus, they can occur anywhere in an instruction and are not mentioned in the grammar.

A.3 INDIVIDUAL OPERATIONS

The examples in the book use a limited set of ILOC operations. The tables at the end of this appendix shows the set of all ILOC operations used in the book, except for the alternate branch syntax used in Chapter 7 to discuss the impact of different forms of branching constructs.

A.3.1 Arithmetic

To express arithmetic, ILOC has three-address, register-to-register operations.

Opcode	Sources	Targets	Meaning
add	r_1, r_2	r_3	$r_1 + r_2 \Rightarrow r_3$
sub	r_1, r_2	r_3	$r_1 - r_2 \Rightarrow r_3$
mult	r_1, r_2	r_3	$r_1 \times r_2 \Rightarrow r_3$
div	r_1, r_2	r_3	$r_1 \div r_2 \Rightarrow r_3$
addI	r_1, c_2	r_3	$r_1 + c_2 \Rightarrow r_3$
subI	r_1, c_2	r_3	$r_1 - c_2 \Rightarrow r_3$
rsubI	r_1, c_2	r_3	$c_2 - r_1 \Rightarrow r_3$
multI	r_1, c_2	r_3	$r_1 \times c_2 \Rightarrow r_3$
divI	r_1, c_2	r_3	$r_1 \div c_2 \Rightarrow r_3$
rdivI	r_1, c_2	r_3	$c_2 \div r_1 \Rightarrow r_3$

All these operations read their source operands from registers or constants and write their result back to a register. Any register can serve as a source or destination operand.

The first four operations are standard register-to-register operations. The next six operations specify an immediate operand. The noncommutative operations, sub and div, have two immediate forms to allow the immediate operand on either side of the operator. The immediate forms are

useful to express the results of certain optimizations, to write down examples more concisely, and to record obvious ways to reduce demand for registers.

Note that a real ILOC-based processor would need more than one data type. This would lead to typed opcodes or to polymorphic opcodes. We would prefer a family of typed opcodes—an integer add, a floating-point add, and so on. The research compiler where ILOC originated has distinct arithmetic operations for integer, single-precision floating-point, double-precision floating-point, complex, and pointer data, but not for character data.

A.3.2 Shifts

ILOC supports a set of arithmetic shift operations—to the left and to the right, in both register and immediate forms.

Opcode	Sources	Targets	Meaning
`lshift`	r_1, r_2	r_3	$r_1 \ll r_2 \Rightarrow r_3$
`lshiftI`	r_1, c_2	r_3	$r_1 \ll c_2 \Rightarrow r_3$
`rshift`	r_1, r_2	r_3	$r_1 \gg r_2 \Rightarrow r_3$
`rshiftI`	r_1, c_2	r_3	$r_1 \gg c_2 \Rightarrow r_3$

A.3.3 Memory Operations

To move values between memory and registers, ILOC supports a full set of load and store operations. The `load` and `cload` operations move data items from memory to registers.

Opcode	Sources	Targets	Meaning
`load`	r_1	r_2	MEMORY $(r_1) \Rightarrow r_2$
`loadAI`	r_1, c_2	r_3	MEMORY $(r_1 + c_2) \Rightarrow r_3$
`loadAO`	r_1, r_2	r_3	MEMORY $(r_1 + r_2) \Rightarrow r_3$
`cload`	r_1	r_2	character `load`
`cloadAI`	r_1, c_2	r_3	character `loadAI`
`cloadAO`	r_1, r_2	r_3	character `loadAO`

The operations differ in the addressing modes that they support. The `load` and `cload` forms assume that the full address is in the single register operand. The `loadAI` and `cloadAI` forms add an immediate value to the contents of the register to form an immediate address before performing the load. We call these *address-immediate* operations. The `loadAO` and `cloadAO`

forms add the contents of two registers to compute an effective address before performing the load. We call these *address-offset* operations.

As a final form of load, ILOC supports a simple load immediate operation. It takes an integer from the instruction stream and places it in a register.

Opcode	Sources	Targets	Meaning
loadI	c_1	r_2	$c_1 \Rightarrow r_2$

A complete, ILOC-like IR should have a load immediate for each distinct kind of value that it supports.

The store operations match the load operations. ILOC supports both numerical stores and character stores in its simple register form, in the address-immediate form, and in the address-offset form.

Opcode	Sources	Targets	Meaning
store	r_1	r_2	$r_1 \Rightarrow$ MEMORY (r_2)
storeAI	r_1	r_2, c_3	$r_1 \Rightarrow$ MEMORY ($r_2 + c_3$)
storeAO	r_1	r_2, r_3	$r_1 \Rightarrow$ MEMORY ($r_2 + r_3$)
cstore	r_1	r_2	character store
cstoreAI	r_1	r_2, c_3	character storeAI
cstoreAO	r_1	r_2, r_3	character storeAO

There is no store immediate operation.

A.3.4 Register-to-Register Copy Operations

To move values between registers, without going though memory, ILOC includes a set of register-to-register copy operations.

Opcode	Sources	Targets	Meaning
i2i	r_1	r_2	$r_1 \Rightarrow r_2$ for integers
c2c	r_1	r_2	$r_1 \Rightarrow r_2$ for characters
c2i	r_1	r_2	convert character to integer
i2c	r_1	r_2	convert integer to character

The first two operations, `i2i` and `c2c`, copy a value from one register to another, with no conversion. The former is for use with integer values, while the latter is for characters. The last two operations perform conversions between characters and integers, replacing a character by its ordinal position in the ASCII character set and replacing an integer with the corresponding ASCII character.

A.4 CONTROL-FLOW OPERATIONS

In general, the ILOC comparison operators take two values and return a boolean value. If the specified relationship holds between its operands, the comparison sets the target register to the value true; otherwise the target register receives false.

Opcode	Sources	Targets	Meaning	
`cmp_LT`	r_1, r_2	r_3	$\texttt{true} \Rightarrow r_3$	if $r_1 < r_2$
			$\texttt{false} \Rightarrow r_3$	otherwise
`cmp_LE`	r_1, r_2	r_3	$\texttt{true} \Rightarrow r_3$	if $r_1 \leq r_2$
			$\texttt{false} \Rightarrow r_3$	otherwise
`cmp_EQ`	r_1, r_2	r_3	$\texttt{true} \Rightarrow r_3$	if $r_1 = r_2$
			$\texttt{false} \Rightarrow r_3$	otherwise
`cmp_GE`	r_1, r_2	r_3	$\texttt{true} \Rightarrow r_3$	if $r_1 \geq r_2$
			$\texttt{false} \Rightarrow r_3$	otherwise
`cmp_GT`	r_1, r_2	r_3	$\texttt{true} \Rightarrow r_3$	if $r_1 > r_2$
			$\texttt{false} \Rightarrow r_3$	otherwise
`cmp_NE`	r_1, r_2	r_3	$\texttt{true} \Rightarrow r_3$	if $r_1 \neq r_2$
			$\texttt{false} \Rightarrow r_3$	otherwise
`cbr`	r_1	l_2, l_3	$l_2 \rightarrow \texttt{PC}$	if $r_1 = \texttt{true}$
			$l_3 \rightarrow \texttt{PC}$	otherwise

The conditional branch operation, `cbr`, takes a boolean as its argument and transfers control to one of two target labels. The first label is selected if the boolean is true; the second is selected if the boolean is false. Because the two branch targets are not "defined" by the instruction, we change the syntax slightly. Rather than use the arrow $\Rightarrow$, we write branches with the single arrow $\rightarrow$.

All branches in ILOC have two labels. This approach eliminates a branch followed by a jump and makes the code more concise. It also eliminates any "fall-through" paths; by making those paths explicit, it removes any positional dependence and simplifies construction of the control-flow graph.

A.4.1 Alternate Comparison and Branch Syntax

To discuss code shape on processors that use a condition code, we must introduce an alternate comparison and branch syntax. The condition code scheme simplifies the comparison and pushes the complexity into the conditional branch operation.

Opcode	Sources	Targets	Meaning	
comp	r_1, r_2	cc_3	sets cc_3	
cbr_LT	cc_1	l_2, l_3	$l_2 \rightarrow PC$	if $cc_3 = LT$
			$l_3 \rightarrow PC$	otherwise
cbr_LE	cc_1	l_2, l_3	$l_2 \rightarrow PC$	if $cc_3 = LE$
			$l_3 \rightarrow PC$	otherwise
cbr_EQ	cc_1	l_2, l_3	$l_2 \rightarrow PC$	if $cc_3 = EQ$
			$l_3 \rightarrow PC$	otherwise
cbr_GE	cc_1	l_2, l_3	$l_2 \rightarrow PC$	if $cc_3 = GE$
			$l_3 \rightarrow PC$	otherwise
cbr_GT	cc_1	l_2, l_3	$l_2 \rightarrow PC$	if $cc_3 = GT$
			$l_3 \rightarrow PC$	otherwise
cbr_NE	cc_1	l_2, l_3	$l_2 \rightarrow PC$	if $cc_3 = NE$
			$l_3 \rightarrow PC$	otherwise

Here, the comparison operator, comp, takes two values and sets the condition code appropriately. We always designate the target of comp as a condition-code register by writing it as cc_i. The corresponding conditional branch has six variants, one for each comparison result.

A.4.2 Jumps

ILOC includes two forms of the jump operation. The form used in almost all of the examples is an immediate jump that transfers control to a literal label. The second, a jump-to-register operation, takes a single register operand. It interprets contents of the register as a runtime address and transfers control to that address.

Opcode	Sources	Targets	Meaning
jumpI	—	l_1	$l_1 \rightarrow PC$
jump	—	r_1	$r_1 \rightarrow PC$

The jump-to-register form is an ambiguous control-flow transfer. Once it has been generated, the compiler may be unable to deduce the correct set of

target labels for the jump. For this reason, the compiler should avoid using jump to register, if possible.

Sometimes, the gyrations needed to avoid a jump to register are so complex that jump to register becomes attractive, despite its problems. For example, FORTRAN includes a construct that jumps to a label variable; implementing it with immediate branches would require logic similar to a case statement—a series of immediate branches, along with code to match the runtime value of the label variable against the set of possible labels. In such circumstances, the compiler should probably use a jump to register.

To reduce the loss of information from jump to register, ILOC includes a pseudo-operation that lets the compiler record the set of possible labels for a jump to register. The `tbl` operation has two arguments, a register and an immediate label.

Opcode	Sources	Targets	Meaning
`tbl`	r_1, l_2	—	r_1 might hold l_2

A `tbl` operation can occur only after a `jump`. The compiler interprets a set of one or more `tbl`s as naming all the possible labels for the register. Thus, the following code sequence asserts that the jump targets one of `L01`, `L03`, `L05`, or `L08`:

```
jump          →ri
tbl   ri, L01
tbl   ri, L03
tbl   ri, L05
tbl   ri, L08
```

A.5 REPRESENTING SSA FORM

When a compiler constructs the SSA form of a program from its IR version, it needs a way to represent ϕ-functions. In ILOC, the natural way to write a ϕ-function is as an ILOC operation. Thus, we will sometimes write

```
phi ri, rj, rk ⇒ rm
```

for the ϕ-function $r_m \leftarrow \phi(r_i, r_j, r_k)$. Because of the nature of SSA form, the `phi` operation may take an arbitrary number of sources. It always defines a single target.

ILOC Opcode Summary			
Opcode	**Sources**	**Targets**	**Meaning**
nop	*none*	*none*	Used as a placeholder
add	r_1, r_2	r_3	$r_1 + r_2 \Rightarrow r_3$
sub	r_1, r_2	r_3	$r_1 - r_2 \Rightarrow r_3$
mult	r_1, r_2	r_3	$r_1 \times r_2 \Rightarrow r_3$
div	r_1, r_2	r_3	$r_1 \div r_2 \Rightarrow r_3$
addI	r_1, c_2	r_3	$r_1 + c_2 \Rightarrow r_3$
subI	r_1, c_2	r_3	$r_1 - c_2 \Rightarrow r_3$
rsubI	r_1, c_2	r_3	$c_2 - r_1 \Rightarrow r_3$
multI	r_1, c_2	r_3	$r_1 \times c_2 \Rightarrow r_3$
divI	r_1, c_2	r_3	$r_1 \div c_2 \Rightarrow r_3$
rdivI	r_1, c_2	r_3	$c_2 \div r_1 \Rightarrow r_3$
lshift	r_1, r_2	r_3	$r_1 \ll r_2 \Rightarrow r_3$
lshiftI	r_1, c_2	r_3	$r_1 \ll c_2 \Rightarrow r_3$
rshift	r_1, r_2	r_3	$r_1 \gg r_2 \Rightarrow r_3$
rshiftI	r_1, c_2	r_3	$r_1 \gg c_2 \Rightarrow r_3$
and	r_1, r_2	r_3	$r_1 \wedge r_2 \Rightarrow r_3$
andI	r_1, c_2	r_3	$r_1 \wedge c_2 \Rightarrow r_3$
or	r_1, r_2	r_3	$r_1 \vee r_2 \Rightarrow r_3$
orI	r_1, c_2	r_3	$r_1 \vee c_2 \Rightarrow r_3$
xor	r_1, r_2	r_3	r_1 *xor* $r_2 \Rightarrow r_3$
xorI	r_1, c_2	r_3	r_1 *xor* $c_2 \Rightarrow r_3$
loadI	c_1	r_2	$c_1 \Rightarrow r_2$
load	r_1	r_2	MEMORY $(r_1) \Rightarrow r_2$
loadAI	r_1, c_2	r_3	MEMORY $(r_1 + c_2) \Rightarrow r_3$
loadAO	r_1, r_2	r_3	MEMORY $(r_1 + r_2) \Rightarrow r_3$
cload	r_1	r_2	character load
cloadAI	r_1, c_2	r_3	character loadAI
cloadAO	r_1, r_2	r_3	character loadAO
store	r_1	r_2	$r_1 \Rightarrow$ MEMORY (r_2)
storeAI	r_1	r_2, c_3	$r_1 \Rightarrow$ MEMORY $(r_2 + c_3)$
storeAO	r_1	r_2, r_3	$r_1 \Rightarrow$ MEMORY $(r_2 + r_3)$
cstore	r_1	r_2	character store
cstoreAI	r_1	r_2, c_3	character storeAI
cstoreAO	r_1	r_2, r_3	character storeAO
i2i	r_1	r_2	$r_1 \Rightarrow r_2$ for integers
c2c	r_1	r_2	$r_1 \Rightarrow r_2$ for characters
c2i	r_1	r_2	convert character to integer
i2c	r_1	r_2	convert integer to character

ILOC Control-Flow Operations				
Opcode	**Sources**	**Targets**	**Meaning**	
jump	—	r_1	$r_1 \rightarrow$ PC	
jumpI	—	l_1	$l_1 \rightarrow$ PC	
cbr	r_1	l_2, l_3	$l_2 \rightarrow$ PC	if $r_1 =$ true
			$l_3 \rightarrow$ PC	otherwise
tbl	r_1, l_2	—	r_1 might hold l_2	
cmp_LT	r_1, r_2	r_3	true $\Rightarrow r_3$	if $r_1 < r_2$
			false $\Rightarrow r_3$	otherwise
cmp_LE	r_1, r_2	r_3	true $\Rightarrow r_3$	if $r_1 \leq r_2$
			false $\Rightarrow r_3$	otherwise
cmp_EQ	r_1, r_2	r_3	true $\Rightarrow r_3$	if $r_1 = r_2$
			false $\Rightarrow r_3$	otherwise
cmp_GE	r_1, r_2	r_3	true $\Rightarrow r_3$	if $r_1 \geq r_2$
			false $\Rightarrow r_3$	otherwise
cmp_GT	r_1, r_2	r_3	true $\Rightarrow r_3$	if $r_1 > r_2$
			false $\Rightarrow r_3$	otherwise
cmp_NE	r_1, r_2	r_3	true $\Rightarrow r_3$	if $r_1 \neq r_2$
			false $\Rightarrow r_3$	otherwise
comp	r_1, r_2	cc_3	sets cc_3	
cbr_LT	cc_1	l_2, l_3	$l_2 \rightarrow$ PC	if $cc_3 =$ LT
			$l_3 \rightarrow$ PC	otherwise
cbr_LE	cc_1	l_2, l_3	$l_2 \rightarrow$ PC	if $cc_3 =$ LE
			$l_3 \rightarrow$ PC	otherwise
cbr_EQ	cc_1	l_2, l_3	$l_2 \rightarrow$ PC	if $cc_3 =$ EQ
			$l_3 \rightarrow$ PC	otherwise
cbr_GE	cc_1	l_2, l_3	$l_2 \rightarrow$ PC	if $cc_3 =$ GE
			$l_3 \rightarrow$ PC	otherwise
cbr_GT	cc_1	l_2, l_3	$l_2 \rightarrow$ PC	if $cc_3 =$ GT
			$l_3 \rightarrow$ PC	otherwise
cbr_NE	cc_1	l_2, l_3	$l_2 \rightarrow$ PC	if $cc_3 =$ NE
			$l_3 \rightarrow$ PC	otherwise