The Iloc Simulator
User Documentation

Comp 506, Spring 2016

The Iloc instruction set is taken from the book, *Engineering A Compiler*, published by the Morgan-Kaufmann imprint of Elsevier [1]. The simulator itself was written over a period of many years by Tim Harvey, Todd Waterman, Keith Cooper, and, perhaps, others. The intent was to provide a useful tool for programming exercises in Rice's compiler courses. The authors' intent was always that this software be distributable without cost for educational use.

1 Introduction and Roadmap

The **ILOC** simulator implements a subset of the **ILOC** operations described in Appendix A of *Engineering a Compiler, Second Edition* [1]. The simulator takes as input a file of **ILOC** operations. It simulates the execution of those operations. It reports on the results of that execution.

The simulator was designed as a target for the programming exercises in c ompiler courses at Rice. It has internal configuration parameters that change the number and type of operations allowed in a given cycle, that change the latencies of opertions, and that change the set of allowed **ILOC** operations. These parameters allow one code base to generate the simulator for multiple programming assignments, with minimal source-level changes. Refer to Section 6 for the specific configuration parameters compiled into the simulator provided for your class.

Roadmap   This document describes the operation of the simulator. Section 2 describes the various command-line options that can control the simulator's behavior. Section 3 describes the **ILOC** virtual machine, the operations that it supports, and their formats and effects. Section 4 explains the command-line options to control instruction synchronization: stalls and interlocks. Section 5 describes the execution tracing facility and the information that it provides. Finally, Section 6 describes the **ILOC** virtual machine configurations supported for different programming assignments, including machine parameters, interlocks, and operation latencies.

2 Command-Line Syntax

Command-line flags and arguments control the simulator's behavior. The command-line syntax for `sim` is as follows:

      sim [options] filename

where the options are drawn from:

`-h`	Prints a list of all the command-line options.
`-d`	Specifies an optional file for input to `read` and `cread` operations. If no file is specified, those operations provide prompts to `stdout` and take their input from `stdin`.
`-m NUM`	Sets the number of *bytes* of available data memory to `NUM`.
`-r NUM`	Sets the number of available registers to `NUM`.
`-s NUM`	Sets the simulator's stall mode based on the value of `NUM`.

Mode 0 has no interlocks.
Mode 1 interlocks on branches.
Mode 2 interlocks on branches. & memory
Mode 3 interlocks on branches, memory, & registers.

The default stall mode is 3.

`-t`	Turns on execution tracing, which prints a record of each operation. Trace output is explained in § 5.
`-i NUM`$_0$ `NUM`$_1$ ... `NUM`$_n$	Treats `NUM`$_1$ through `NUM`$_n$ as integer data items and initializes data memory by writing them into consecutive words of memory, starting at the address `NUM`$_0$.
`-c NUM`$_0$ `NUM`$_1$ ... `NUM`$_n$	Operates in the same manner as `-i`, except that it treats `NUM`$_1$ through `NUM`$_n$ as bytes.
`-v`	Prints the version number of the simulator executable.

The simulator expects that `filename` is a valid **ILOC** source file. It will read the file and execute the code, starting at the first operation in the code.

The `-m` and `-r` options must occur before any `-i` or `-c` option. Either `-i` and `-c` will cause the simulator to allocate its data structures so that it can initialize them. Once the data structures have been allocated, the `-m` and `-r` options have no effect.

The `-h`, `-d`, `-m`, `-r`, `-s`, and `-t` options should occur only once on the command line. Both the `-i` and `-c` options can occur multiple times, to initialize different address ranges in memory.

3   The ILOC Virtual Machine

The simulator implements a simle virtual machine that supports supports a subset of **ILOC** [1, App. A]. Some properties of the virtual machine are fixed. For example, it features separate address spaces for code and data; a running program cannot read or write code memory. Some properties of the virtual machine are configurable from the command line. For example, the size of data memory, the number of registers, and the set of interlocks enforced on operations are all governed by command-line flags. Finally, some properties are determined when the simulator is, itself, compiled. For example, restrictions on the number of operations that execute per cycle (roughly speaking, the number of funcitonal units that it emulates) and operation latencies are easily configurable.

The **ILOC** operations supported in the simulator fall into four basic categories: computational operations, data movement operations, control-flow operations, and output operations. Tables 1, 2, 3, and 4 describe each group of operations.

An **ILOC** instruction is either a single operation, or a group of operations enclosed in square brackets and separated by semicolons, as in [op_1 ; op_2]. An instruction label in **ILOC** consists of an alphabetic character followed by zero or more alphanumeric characters. Any **ILOC** instruction may be labeled; the label precedes the instruction and is followed by a

Opcode	Format			Meaning	
nop	nop			no change to registers or memory	
				used as a placeholder or to cause a delay	
add	add	r_0, r_1	$\Rightarrow r_2$	$r_2 \leftarrow r_0 + r_1$	
addl	addl	r_0, c_1	$\Rightarrow r_2$	$r_2 \leftarrow r_0 + c_1$	
sub	sub	r_0, r_1	$\Rightarrow r_2$	$r_2 \leftarrow r_0 - r_1$	
subl	subi	r_0, c_1	$\Rightarrow r_2$	$r_2 \leftarrow r_0 - c_1$	
mult	mult	r_0, r_1	$\Rightarrow r_2$	$r_2 \leftarrow r_0 \times r_1$	
multl	multl	r_0, c_1	$\Rightarrow r_2$	$r_2 \leftarrow r_0 \times c_1$	
div	div	r_0, r_1	$\Rightarrow r_2$	$r_2 \leftarrow r_0 \div r_1$	
divl	divl	r_0, c_1	$\Rightarrow r_2$	$r_2 \leftarrow r_0 \div c_1$	
lshift	lshift	r_0, r_1	$\Rightarrow r_2$	$r_2 \leftarrow r_0 \ll r_1$	
lshiftl	lshiftl	r_0, c_1	$\Rightarrow r_2$	$r_2 \leftarrow r_0 \ll c_1$	
rshift	rshift	r_0, r_1	$\Rightarrow r_2$	$r_2 \leftarrow r_0 \gg r_1$	
rshiftl	rshiftl	r_0, c_1	$\Rightarrow r_2$	$r_2 \leftarrow r_0 \gg c_1$	
and	and	r_0, r_1	$\Rightarrow r_2$	$r_2 \leftarrow r_0 \ \&\& \ r_1$	(logical and)
andl	andl	r_0, c_1	$\Rightarrow r_2$	$r_2 \leftarrow r_0 \ \&\& \ c_1$	(logical and)
or	or	r_0, r_1	$\Rightarrow r_2$	$r_2 \leftarrow r_0 \ \|\| \ r_1$	(logical or)
orl	orl	r_0, c_1	$\Rightarrow r_2$	$r_2 \leftarrow r_0 \ \|\| \ c_1$	(logical or)
not	not	r_0	$\Rightarrow r_1$	$r_1 \leftarrow \ ! \ r_0$	(logical complement)

Table 1: The ILOC Computational Operations

Opcode	Format			Meaning
loadI	loadI	c_0	$\Rightarrow r_1$	$r_1 \leftarrow c_0$
load	load	r_0	$\Rightarrow r_1$	$r_1 \leftarrow \text{WORD}[\,r_0\,]$
loadAI	loadAI	r_0, c_1	$\Rightarrow r_2$	$r_1 \leftarrow \text{WORD}[\,r_0 + c_1\,]$
loadAO	loadAI	r_0, r_1	$\Rightarrow r_2$	$r_1 \leftarrow \text{WORD}[\,r_0 + r_1\,]$
cload	cload	r_0	$\Rightarrow r_1$	$r_1 \leftarrow \text{BYTE}[\,r_0\,]$
cloadAI	cloadAI	r_0, c_1	$\Rightarrow r_2$	$r_1 \leftarrow \text{BYTE}[\,r_0 + c_1\,]$
cloadAO	cloadAI	r_0, r_1	$\Rightarrow r_2$	$r_1 \leftarrow \text{BYTE}[\,r_0 + r_1\,]$
store	store	r_0	$\Rightarrow r_1$	$\text{WORD}[\,r_1\,] \leftarrow r_0$
storeAI	store	r_0	$\Rightarrow r_1, c_2$	$\text{WORD}[\,r_1 + c_2\,] \leftarrow r_0$
storeAO	store	r_0	$\Rightarrow r_1, r_2$	$\text{WORD}[\,r_1 + r_2\,] \leftarrow r_0$
cstore	cstore	r_0	$\Rightarrow r_1$	$\text{BYTE}[\,r_1\,] \leftarrow r_0$
cstoreAI	cstore	r_0	$\Rightarrow r_1, c_2$	$\text{BYTE}[\,r_1 + c_2\,] \leftarrow r_0$
cstoreAO	cstore	r_0	$\Rightarrow r_1, r_2$	$\text{BYTE}[\,r_1 + r_2\,] \leftarrow r_0$
i2i	i2i	r_0	$\Rightarrow r_1$	$r_1 \leftarrow r_0$, as an integer
c2c	c2c	r_0	$\Rightarrow r_1$	$r_1 \leftarrow r_0$, as a character
i2c	i2c	r_0	$\Rightarrow r_1$	$r_1 \leftarrow r_0$, as a character
c2i	c2i	r_0	$\Rightarrow r_1$	$r_1 \leftarrow r_0$, as a character

Table 2: The ILOC Data-Movement Operations

colon, as in L01: add r1, r2 $\Rightarrow$ r3, or L02: [add r1, r2 $\Rightarrow$ r3 ; i2i r0 $\Rightarrow$ r4].

ILOC is case sensitive. The only uppercase letters in ILOC are the 'A', 'I', and 'O' used to specify address modes and immediate operations.

In the tables, r_i represents a register name; the subscripts make explicit the correspondence between operands in the "Format" column and the "Meaning" column. The notation c_i represents an integer constant, and L_i represents a label. "WORD[ex]" indicates the contents of the word of data memory at the location specified by ex. The address expression, ex, must be *word-aligned*—that is (ex MOD 4) must be 0. "BYTE[ex]" indicates the contents of the byte of data memory at the location specified by ex, without an alignment constraint on ex.

Register names have an initial r followed immediately by a non-negative integer. The 'r' is case sensitive (as is all of ILOC). Leading zeroes in the register name are not significant; thus r017 and r17 refere to the same register. Arguments that do not begin with an 'r' which appear as a c in the tables, are assumed to be positive integers constants in the range 0 to 2^{31}-1.

Blanks and tabs are treated as whitespace. All ILOC opcodes must be followed by whitespace—any combination of blanks or tabs. Whitespace preceding and following other symbols is optional. Whitespace may not appear within operation names, register names, or the assignment symbol. A double slash ("//") indicates that the rest of the line is a comment. Empty lines may appear in the input; the simulator will ignore them.

Opcode	Format			Meaning
br	br		$\rightarrow$ L$_0$	control transfers to L$_0$
cbr	cbr	r$_1$	$\rightarrow$ L$_1$, L$_2$	if r$_0$ is true, control transfers to L$_1$
				otherwise, control transfers to L$_2$
cmp_LT	cmp_LT	r$_0$, r$_1$	$\Rightarrow$ r$_2$	r$_2$ $\leftarrow$ true if r$_0 <$ r$_1$
				otherwise, r$_2$ $\leftarrow$ false
cmp_LE	cmp_LE	r$_0$, r$_1$	$\Rightarrow$ r$_2$	r$_2$ $\leftarrow$ true if r$_0$ le r$_1$
				otherwise, r$_2$ $\leftarrow$ false
cmp_GT	cmp_GT	r$_0$, r$_1$	$\Rightarrow$ r$_2$	r$_2$ $\leftarrow$ true if r$_0 >$ r$_1$
				otherwise, r$_2$ $\leftarrow$ false
cmp_GE	cmp_GE	r$_0$, r$_1$	$\Rightarrow$ r$_2$	r$_2$ $\leftarrow$ true if r$_0$ ge r$_1$
				otherwise, r$_2$ $\leftarrow$ false
cmp_EQ	cmp_EQ	r$_0$, r$_1$	$\Rightarrow$ r$_2$	r$_2$ $\leftarrow$ true if r$_0 =$ r$_1$
				otherwise, r$_2$ $\leftarrow$ false
cmp_NE	cmp_NE	r$_0$, r$_1$	$\Rightarrow$ r$_2$	r$_2$ $\leftarrow$ true if r$_0 \neq$ r$_1$
				otherwise, r$_2$ $\leftarrow$ false

Table 3: The ILOC Control-Flow Operations

Opcode	Format		Meaning
read	read	r$_0$	reads one integer from `stdin` and places its value in r$_0$
cread	cread	r$_0$	reads one character from `stdin` and places its value in r$_0$
output	output	c$_0$	writes WORD$[$c$_0]$ to `stdout`
			c$_0$ must be a word-aligned integer constant
coutput	coutput	c$_0$	writes BYTE$[$c$_0]$ to `stdout` as a character
write	write	r$_0$	writes contents of r$_0$ to `stdout` as an integer
			Each write generates a separate line.
cwrite	write	r$_0$	writes contents of r$_0$ to `stdout` as a character
			Each write generates a separate line.

Table 4: The ILOC Input and Output Operations

Pseudo-Operations to Define Storage

In addition to the **ILOC** operations, the simulator supports two pseudo-operations to initialize memory with known values; they are "pseudo" operations in the sense that their effects occur before the **ILOC** code starts execution.

Pseudo-Op	Format	Meaning
dcs	dcs c_0 c_1 c_2 $\ldots c_n$	c_0 is an integer address. c_1 through c_n are characters separated by blanks. c_1 through c_n are placed in consecutive bytes of memory, starting at address c_0.
dis	dis c_0 c_1 c_2 $\ldots c_3$	c_0 is a word-aligned integer address. c_1 through c_n are integers. c_1 through c_n are placed in consecutive words of memory, starting at address c_0.

Here, dcs is an acronym for *define character storage* and dis is an acronym for *define integer storage*. These pseudo-operations provide a convenient way to initialize large quantities of memory, such as an array.

The pseudo-operations must appear before the first **ILOC** operation. There is no arbitrary restriction on the number of pseudo-operations or their relative order. The obvious way to use these pseudo-operations is to create a "data" file and use the -d command-line flag to prepend it to an **ILOC** file. In this way, a single program can be easily run against multiple different data files.

4   Interlocks

The **ILOC** simulator supports four distinct levels of enforced synchronization. These levels are:

3   interlocks on branches, registers, and memory
2   interlocks on branches and memory
1   interlocks on branches
0   no interlocks

The branch interlock prevents the simulator from executing operations during the latency of a branch. Once the branch operation is issued, the simulator will stall all pending operations until the branch completes.

The memory interlock stalls operations when a load or output operation tries to access a memory location that will be modified by a pending operation—one that has been issued and has not yet completed. Similarly, the register interlock stalls operations that attempt to read a value from a register that will be modified by a pending operation.

The interlock level is set by the -s command-line flag. The default interlock levels are documented in Section 6. The trace facility lists the interlock level in use for each run at the head of the trace.

5   Understanding the Trace Output

To help the user understand the details of a specific execution, the **ILOC** simulator includes a trace facility, invoked with the -t command-line flag. The trace shows each executed instruction, its arguments and its results.

Figure 1 shows an example trace generated by the **ILOC** code for the small program shown to the right. Below the source code is the **ILOC** code that the compiler generated. While the program is simple, it highlights several of the important features of the trace facility.

In the translation, both `a` and `i` are kept in registers. Thus, the **ILOC** code contains no `load` operations. The only `store` operation is generated by the `print` statement, which must store `a`'s value to memory so that it can generate an **ILOC output** operation. Notice that the code assumes a single functional-unit configuration of the simulator; it contains no explicitly specified instruction-level parallelism.

Turning our attention to the trace in Figure 1, it begins by listing the version number of the simulator and the interlock settings in use for the run. The final line in the trace is the execution summary generated by every simulator run, whether traced or not. Between the interlock settings and the execution summary, the trace lists the operations executed at each cycle of the execution, one cycle per line. Each line begins with the cycle number.

```
procedure main {
  int a, i;
  a = 1;
  for i = 1 to 4 by 1 {
    a = a + 1;
  }
  print a;
}
```

Source Program

```
    loadI 1 => r0
    loadI 1 => r1
    loadI 4 => r2
    cmp_LE r1, r2 => r3
    cbr r3 -> L0, L1
L0: addI r0, 1 => r4
    i2i r4 => r0
    addI r1, 1 => r1
    cmp_LE r1, r2 => r5
    cbr r5 -> L0, L1
L1: loadI 0 => r6
    store r0 => r6
    output 0
```

ILOC Program

To the right of the cycle number, the operation(s) executed in that cycle are listed. For any register operand, the value of the register appears in parentheses after the register's name. Registers that are used show their values before the operation executes. Registers that are defined show their values after the operation takes effect.

For a long-latency operation, the result is shown in the trace for the cycle in which the instruction issues. When a long-latency operation completes, that fact is noted at the end of the trace for the cycle in which it completes. The trace for that cycle will show an asterisk ('*') followed by the cycle number in which the long-latency operation was first issued.

Look at the `store` issued in cycle 26. Because the `output` uses the same memory location (and the simulator has memory interlocks enabled), the `output` operation stalls until the `store` completes in cycle 30. The trace for cycle 30 ends with the notation "*26" to indicate that an operation issued in cycle 26 completed at the end of cycle 30. The `output` is then issued in the next cycle.

Now, look at the `cbr` operations in cycles 9, 14, 19, and 24. The asterisk in the trace for `cbr` indicates which branch was taken. In the first three `cbr` operations, the branch transferred control to L0 at the top of the loop. In the final `cbr`, it transferred control to L1, the label on the first statement after the loop.

When the simulator is generating a trace, it changes the format of the output generated by the `output` operation. In a run without tracing, the number is simply writted to the

```
ILOC Simulator, Version 512-2-0
Interlock settings: memory registers branches

0:  [loadI 1 => r0 (1)]
1:  [loadI 1 => r1 (1)]
2:  [loadI 4 => r2 (4)]
3:  [cmp_LE r1 (1), r2 (4) => r3 (1)]
4:  [cbr r3 (1) -> L0*, L1]
5:  [addI r0 (1), 1 => r4 (2)]
6:  [i2i r4 (2) => r0 (2)]
7:  [addI r1 (1), 1 => r1 (2)]
8:  [cmp_LE r1 (2), r2 (4) => r5 (1)]
9:  [cbr r5 (1) -> L0*, L1]
10: [addI r0 (2), 1 => r4 (3)]
11: [i2i r4 (3) => r0 (3)]
12: [addI r1 (2), 1 => r1 (3)]
13: [cmp_LE r1 (3), r2 (4) => r5 (1)]
14: [cbr r5 (1) -> L0*, L1]
15: [addI r0 (3), 1 => r4 (4)]
16: [i2i r4 (4) => r0 (4)]
17: [addI r1 (3), 1 => r1 (4)]
18: [cmp_LE r1 (4), r2 (4) => r5 (1)]
19: [cbr r5 (1) -> L0*, L1]
20: [addI r0 (4), 1 => r4 (5)]
21: [i2i r4 (5) => r0 (5)]
22: [addI r1 (4), 1 => r1 (5)]
23: [cmp_LE r1 (5), r2 (4) => r5 (0)]
24: [cbr r5 (0) -> L0, L1*]
25: [loadI 0 => r6 (0)]
26: [store r0 (5) => r6 (addr: 0)]
27: [ stall ]
28: [ stall ]
29: [ stall ]
30: [ stall ] *26
31: [output 0 (5)]
output generates => 5

Executed 28 instructions and 28 operations in 32 cycles.
```

Figure 1: Execution Trace for Simple Example

standard output stream, one number per line. In a run with tracing, the simulator adds the text `output generates =>` to help the user find the printed result.

6 Details of Specific Simulator Configurations

This section lists configuration details of implementations for specific classes.

6.1 COMP 506, Spring 2016, Lab 2

The simulator for COMP 506 supports the full set of **ILOC** operations. All operations execute in a single cycle (latency of one cycle), except:

- `mult` and `multI` require three cycles.

- `div` and `divI` require six cycles.

- `load`, `loadAI`, `loadAO`, `cload`, `cloadAI`, `cloadAO`, `store`, `storeAI`, `storeAO`, `cstore`, `cstoreAI`, and `cstoreAO` all require five cycles.

The simulator is configured with a single functional unit—that is, it can issue at most one operation per cycle. By default, it had 4,000,000 bytes of memory and 1,000 registers. You can change these values with command-line options. The default stall mode is 3. (Stalls and interlocks are explained in § 4).

Your compiler can lay out data in memory in any way that you see fit. The simulator has separate instruction and data memory, so your variables and temporary values cannot interfere with the representation of the code.

Other configurations omitted to avoid confusion

References

[1] Keith Cooper and Linda Torczon. *Engineering A Compiler*. Elsevier Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2011.