The Iloc Language Definition

1 Purpose

This document describes the Iloc intermediate language supported by our software.

2 Registers

Virtual register are represented, for example, using the syntax %vr1 for virtual register 1. There are four reserved virtual registers: %vr0 is the frame pointer, %vr1 is the stack pointer and %vr2 and %vr3 are used for function call linkage.

3 Instructions

3.0.1 Integer Arithmetic Instructions

To support arithmetic, Iloc supports the following operations:

Opcode	Sources	Target	Meaning	Example Syntax
i2i	vr_1	vr_2	$vr_2 \leftarrow vr_1$	i2i %vr5 => %vr6
add	vr_1, vr_2	vr_3	$vr_3 \leftarrow vr_1 + vr_2$	add %vr5, %vr6 => %vr7
sub	vr_1, vr_2	vr_3	$vr_3 \leftarrow vr_1 - vr_2$	sub %vr5, %vr6 => %vr7
mult	vr_1, vr_2	vr_3	$vr_3 \leftarrow vr_1 * vr_2$	mult %vr5, %vr6 => %vr7
lshift	vr_1, vr_2	vr_3	$vr_3 \leftarrow vr_1 \ll vr_2$	lshift %vr5, %vr6 => %vr7
rshift	vr_1, vr_2	vr_3	$vr_3 \leftarrow vr_1 \gg vr_2$	rshift %vr5, %vr6 => %vr7
mod	vr_1, vr_2	vr_3	$vr_3 \leftarrow vr_1 \bmod vr_2$	mod %vr5, %vr6 => %vr7
and	vr_1, vr_2	vr_3	$vr_3 \leftarrow vr_1 \,\&\&\, vr_2$	and %vr5, %vr6 => %vr7
or	vr_1, vr_2	vr_3	$vr_3 \leftarrow vr_1 \parallel vr_2$	or %vr5, %vr6 => %vr7
not	vr_1	vr_2	$vr_2 \leftarrow 1\text{s} - \text{complement}(vr_1)$	not %vr5 => %vr6
addI	vr_1, c_1	vr_2	$vr_2 \leftarrow vr_1 + c_1$	addI %vr5, 1 => %vr7
subI	vr_1, c_1	vr_2	$vr_2 \leftarrow vr_1 - c_1$	subI %vr5, 1 => %vr7
multI	vr_1, c_1	vr_2	$vr_2 \leftarrow vr_1 * c_1$	multI %vr5, 1 => %vr7
lshiftI	vr_1, c_1	vr_2	$vr_2 \leftarrow vr_1 \ll c_1$	lshiftI %vr5, 1 => %vr7
rshiftI	vr_1, c_1	vr_2	$vr_2 \leftarrow vr_1 \gg c_1$	rshiftI %vr5, 1 => %vr7

3.1 Integer Memory Operations

To support memory references, Iloc includes the following operations:

Opcode	Sources	Target	Meaning	Example Syntax
loadI	c_1	vr_1	$vr_1 \leftarrow c_1$	loadI 1 => %vr5
load	vr_1	vr_2	$vr_2 \leftarrow$MEMORY(vr_1)	load %vr5 => %vr6
loadAI	vr_1, c_1	vr_2	$vr_2 \leftarrow$MEMORY$(vr_1 + c_1)$	loadAI %vr5, 8 => %vr6
loadAO	vr_1, vr_2	vr_3	$vr_3 \leftarrow$MEMORY$(vr_1 + vr_2)$	loadAO %vr5, %vr6 => %vr7
store	vr_1	vr_2	MEMORY$(vr_2) \leftarrow vr_1$	store %vr5 => %vr6
storeAI	vr_1	vr_2, c_1	MEMORY$(vr_2 + c_1) \leftarrow vr_1$	storeAI %vr5 => %vr6, 8
storeAO	vr_1	vr_2, vr_3	MEMORY$(vr_2 + vr_3) \leftarrow vr_1$	storeAO %vr5 => %vr6, %vr7

3.2 Compare Instructions

We support a set of operations that compare registers and output true or false into another register.

Opcode	Sources	Target	Meaning	Example Syntax
`cmp_LT`	vr_1,vr_2	vr_3	if $vr_1 < vr_2$, $vr_3 \leftarrow$ true, else $vr_3 \leftarrow$ false	`cmp_LT %vr5, %vr6 => %vr7`
`cmp_LE`	vr_1,vr_2	vr_3	if $vr_1 <= vr_2$, $vr_3 \leftarrow$ true, else $vr_3 \leftarrow$ false	`cmp_LE %vr5, %vr6 => %vr7`
`cmp_GT`	vr_1,vr_2	vr_3	if $vr_1 > vr_2$, $vr_3 \leftarrow$ true, else $vr_3 \leftarrow$ false	`cmp_GT %vr5, %vr6 => %vr7`
`cmp_GE`	vr_1,vr_2	vr_3	if $vr_1 >= vr_2$, $vr_3 \leftarrow$ true, else $vr_3 \leftarrow$ false	`cmp_GE %vr5, %vr6 => %vr7`
`cmp_EQ`	vr_1,vr_2	vr_3	if $vr_1 = vr_2$, $vr_3 \leftarrow$ true, else $vr_3 \leftarrow$ false	`cmp_EQ %vr5, %vr6 => %vr7`
`cmp_NE`	vr_1,vr_2	vr_3	if $vr_1 \neq vr_2$, $vr_3 \leftarrow$ true, else $vr_3 \leftarrow$ false	`cmp_NE %vr5, %vr6 => %vr7`
`comp`	vr_1, vr_2	vr_3	compare vr_1 and vr_2 and put result in vr_3	`comp %vr5, %vr6 => %vr7`
`testeq`	vr_1	vr_2	if vr_1 is set to $=$, $vr_2 \leftarrow$ true, else $vr_2 \leftarrow$ false	`testeq %vr5 => %vr6`
`testne`	vr_1	vr_2	if vr_1 is set to $\neq$, $vr_2 \leftarrow$ true, else $vr_2 \leftarrow$ false	`testne %vr5 => %vr6`
`testgt`	vr_1	vr_2	if vr_1 is set to $>$, $vr_2 \leftarrow$ true, else $vr_2 \leftarrow$ false	`testgt %vr5 => %vr6`
`testge`	vr_1	vr_2	if vr_1 is set to $\geq$, $vr_2 \leftarrow$ true, else $vr_2 \leftarrow$ false	`testge %vr5 => %vr6`
`testlt`	vr_1	vr_2	if vr_1 is set to $<$, $vr_2 \leftarrow$ true, else $vr_2 \leftarrow$ false	`testlt %vr5 => %vr6`
`testle`	vr_1	vr_2	if vr_1 is set to $\leq$, $vr_2 \leftarrow$ true, else $vr_2 \leftarrow$ false	`testle %vr5 => %vr6`

3.3 Floating-point Operations

The following floating-point exist in `Iloc`.

Opcode	Sources	Target	Meaning	Example Syntax
`f2i`	vr_1	vr_2	$vr_2 \leftarrow \text{int}(vr_1)$	`f2i %vr5 => %vr6`
`i2f`	vr_1	vr_2	$vr_2 \leftarrow \text{float}(vr_1)$	`i2f %vr5 => %vr6`
`f2f`	vr_1	vr_2	$vr_2 \leftarrow vr_1$	`f2f %vr5 => %vr6`
`fadd`	vr_1, vr_2	vr_3	$vr_3 \leftarrow vr_1 + vr_2$	`fadd %vr5, %vr6 => %vr7`
`fsub`	vr_1, vr_2	vr_3	$vr_3 \leftarrow vr_1 - vr_2$	`fsub %vr5, %vr6 => %vr7`
`fmult`	vr_1, vr_2	vr_3	$vr_3 \leftarrow vr_1 \times vr_2$	`fmult %vr5, %vr6 => %vr7`
`fdiv`	vr_1, vr_2	vr_3	$vr_3 \leftarrow vr_1 / vr_2$	`fdiv %vr5, %vr6 => %vr7`
`fcomp`	vr_1, vr_2	vr_3	compare vr_1 and vr_2 and put result in vr_3	`fcomp %vr5, %vr6 => %vr7`
`fload`	vr_1	vr_2	$vr_2 \leftarrow$`MEMORY`(vr_1)	`fload %vr5 => %vr6`
`floadAI`	vr_1, c_1	vr_2	$vr_2 \leftarrow$`MEMORY`$(vr_1 + c_1)$	`floadAI %vr5, 8 => %vr6`
`floadAO`	vr_1, vr_2	vr_3	$vr_3 \leftarrow$`MEMORY`$(vr_1 + vr_2)$	`floadAO %vr5, %vr6 => %vr7`
`fstore`	vr_1	vr_2	`MEMORY`$(vr_2) \leftarrow vr_1$	`fstore %vr5 => %vr6`
`fstoreAI`	vr_1	vr_2, c_1	`MEMORY`$(vr_2 + c_1) \leftarrow vr_1$	`fstoreAI %vr5 => %vr6, 8`
`fstoreAO`	vr_1	vr_2, vr_3	`MEMORY`$(vr_2 + vr_3) \leftarrow vr_1$	`fstoreAO %vr5 => %vr6, %vr7`

3.4 I/O Instructions

The following instructions have been added to support simple I/O. *All read instructions expect input to be separated by lines, not white space.*

Opcode	Sources	Target	Meaning	Example Syntax
`fread`		vr_1	read a float from stdin and store it at `MEMORY`(vr_1)	`fread %vr5`
`iread`		vr_1	read an integer from stdin and store it at `MEMORY`(vr_1)	`iread %vr5`
`fwrite`	vr_1		write a float in vr_1 to stdout	`fwrite %vr5`
`iwrite`	vr_1		write an integer in vr_1 to stdout	`iwrite %vr5`
`swrite`	vr_1		write a '`\0`'-terminated string stored at `MEMORY`(vr_1) to stdout	`swrite %vr5`

3.5 Branch Instructions

These are the unconditional and conditional branch operations.

Opcode	Sources	Target	Meaning	Example Syntax
jumpI	none	l_1	branch to l_1	jumpI -> .L0
jump	none	vr_1	branch to MEMORY(vr_1)	jump -> %vr5
ret	none	none	exit from program	ret
cbr	vr_1	l_1	if vr_1 is true, branch to l_1, else fall through	cbr %vr5 -> .L0
cbrne	vr_1	l_1	if vr_1 is false, branch to l_1, else fall through	cbrne %vr5 -> .L0
cbr_LT	vr_1,vr_2	l_1	if $vr_1 < vr_2$, branch to l_1, else fall through	cbr_LT %vr5, %vr6 -> .L0
cbr_LE	vr_1,vr_2	l_1	if $vr_1 <= vr_2$, branch to l_1, else fall through	cbr_LE %vr5, %vr6 -> .L0
cbr_GT	vr_1,vr_2	l_1	if $vr_1 > vr_2$, branch to l_1, else fall through	cbr_GT %vr5, %vr6 -> .L0
cbr_GE	vr_1,vr_2	l_1	if $vr_1 >= vr_2$, branch to l_1, else fall through	cbr_GE %vr5, %vr6 -> .L0
cbr_EQ	vr_1,vr_2	l_1	if $vr_1 != vr_2$, branch to l_1, else fall through	cbr_EQ %vr5, %vr6 -> .L0
cbr_NE	vr_1,vr_2	l_1	if $vr_1 = vr_2$, branch to l_1, else fall through	cbr_NE %vr5, %vr6 -> .L0

Note that there are no specific condition registers. All registers are virtual registers.

3.6 Stack Instructions

There are number of instructions that allow one to use Iloc as a stack-based architecture.

Opcode	Sources	Meaning
push	c_0	%vr1 $-= 4$, MEMORY(%vr1) $= c_0$
pushr	vr_0	%vr1 $-= 4$, MEMORY(%vr1) $= vr_0$
pop	none	%vr1 $+= 4$
stadd	none	MEMORY(%vr1+4) = MEMORY(%vr1+4) + MEMORY(%vr1), %vr1 $+= 4$
stsub	none	MEMORY(%vr1+4) = MEMORY(%vr1+4) − MEMORY(%vr1), %vr1 $+= 4$
stmul	none	MEMORY(%vr1+4) = MEMORY(%vr1+4) × MEMORY(%vr1), %vr1 $+= 4$
stdiv	none	MEMORY(%vr1+4) = MEMORY(%vr1+4) / MEMORY(%vr1), %vr1 $+= 4$
stlshift	none	MEMORY(%vr1+4) = MEMORY(%vr1+4) $\ll$ MEMORY(%vr1), %vr1 $+= 4$
strshift	none	MEMORY(%vr1+4) = MEMORY(%vr1+4) $\gg$ MEMORY(%vr1), %vr1 $+= 4$
stcomp	none	compare MEMORY(%vr1+4) and MEMORY(%vr1) and put result in MEMORY(%vr1+4), %vr1 $+= 4$
stand	none	MEMORY(%vr1+4) = MEMORY(%vr1+4) & MEMORY(%vr1), %vr1 $+= 4$
stor	none	MEMORY(%vr1+4) = MEMORY(%vr1+4) | MEMORY(%vr1), %vr1 $+= 4$
stnot	none	MEMORY(%vr1) = MEMORY(%vr1)
stload	none	MEMORY(%vr1) = MEMORY(MEMORY(%vr1))
ststore	none	MEMORY(MEMORY(%vr1)) = MEMORY(%vr1+4)), %vr1 $+= 8$
sttesteq	none	if MEMORY(%vr1) is set to $=$, MEMORY(%vr1) = true, else MEMORY(%vr1) = false
sttestne	none	if MEMORY(%vr1) is set to $\neq$, MEMORY(%vr1) = true, else MEMORY(%vr1) = false
sttestgt	none	if MEMORY(%vr1) is set to $>$, MEMORY(%vr1) = true, else MEMORY(%vr1) = false
sttestge	none	if MEMORY(%vr1) is set to $\geq$, MEMORY(%vr1) = true, else MEMORY(%vr1) = false
sttestlt	none	if MEMORY(%vr1) is set to $<$, MEMORY(%vr1) = true, else MEMORY(%vr1) = false
sttestle	none	if MEMORY(%vr1) is set to $\leq$, MEMORY(%vr1) = true, else MEMORY(%vr1) = false
stfadd	none	MEMORY(%vr1+4) = MEMORY(%vr1+4) + MEMORY(%vr1), %vr1 $+= 4$
stfsub	none	MEMORY(%vr1+4) = MEMORY(%vr1+4) − MEMORY(%vr1), %vr1 $+= 4$
stfmul	none	MEMORY(%vr1+4) = MEMORY(%vr1+4) × MEMORY(%vr1), %vr1 $+= 4$
stfdiv	none	MEMORY(%vr1+4) = MEMORY(%vr1+4) / MEMORY(%vr1), %vr1 $+= 4$
stfcomp	none	compare MEMORY(%vr1+4) and MEMORY(%vr1) and put result in MEMORY(%vr1+4), %vr1 $+= 4$
stfload	none	MEMORY(%vr1) = MEMORY(MEMORY(%vr1))
stfstore	none	MEMORY(MEMORY(%vr1)) = MEMORY(%vr1+4)), %vr1 $+= 8$
stfread	none	read a float from stdin and store it at MEMORY(MEMORY(%vr1))
stiread	none	read an integer from stdin and store it at MEMORY(MEMORY(%vr1))
stfwrite	none	write a float in MEMORY(%vr1) to stdout
stiwrite	none	write an integer in MEMORY(%vr1) to stdout
stswrite	none	write a '\0'-terminated string stored at MEMORY(%vr1) to stdout
stjump	none	branch to MEMORY(%vr1)

3.7 Pseudo-ops

The following pseudo-ops have been added.

Opcode	Operands	Meaning
`.data`		declare the data segment
`.text`		declare the text segment
`.frame`	*name*, *size* $[, vr_i]^*$	declare procedure *name* with a stack of *size* bytes and optional params vr_i^*
`.global`	*name*, *size*, *align*	declare a global variable *name* of size *size* bytes with alignment *align*
`.string`	*name*, *string-literal*	declare a string *name* with value *string-literal*
`.float`	*name, float-val*	declare a floating-point constant *name* with value *float-val*

Note that the virtual machine memory is in little endian format.

4 Example

Below is an example `Iloc` program.

```
.data
.string .string_const_0, "Hello world!"
.text
.frame main, 0
loadI .string_const_0  => %vr4
swrite %vr4
ret
```