AVES INTERMEDIATE REPRESENTATION

CS 473: Compiler Design / Fall 2025

The Aves (the scientific name of the class of all birds) Intermediate Representation (IR) is a stack-based IR, in which all operands are taken from the stack, and all results are left on the stack for the next operands. Some IR operations use immediate values to specify exactly what to do. Below is a list of all possible operations, and how they manipulate the stack (S denotes "the rest of the stack"):

Constants and arithmetic

Operation	Immediates	Stack before	Stack after	Description
ICONST	int	S	<int> S	Pushes the integer specified as an immediate
SCONST	string	S	<address> S	Pushes the string specified as an immediate, leaves its address on top of the stack.
OPS		<int2> <int1> S	<int3> S	Performs a binary operation on the two values on top of the stack, leaves the result on top of the stack <int3> = <int1> OP <int2>
OPS		<int1> S	<int2> S	Performs a unary operation on the value on top of the stack, leaves the result on top of the stack <int2> = OP <int1>

Aves supports binary and unary operations. Both types of operations leave the result on top of the stack. Binary operations take two operands from the stack, and unary operations take their single argument from the stack. All operations are signed unless told otherwise. Overflow behavior is undefined.

Binary operations

Name	Stack before	Stack after	Notes
ir_add	S <A> <B>	S <A+B>	
ir_sub	S <A> <B>	S <A-B>	
ir_mul	S <A> <B>	S <A*B>	
ir_div	S <A> <B>	S <A/B>	
ir_mod	S <A> <B>	S <A%B>	
ir_bor	S <A> <B>	S <A|B>	Bitwise OR

ir_band	S <A> <B>	S <A&B>	Bitwise AND
ir_xor	S <A> <B>	S <A^B>	
ir_or	S <A> <B>	S <A||B>	Logical OR
ir_and	S <A> <B>	S <A&&B>	Logical AND
ir_eq	S <A> <B>	S <A==B>	Not zero if A has the same value as B, zero otherwise
ir_lt	S <A> <B>	S <A<B>	Not zero if A is less than B, zero otherwise
ir_gt	S <A> <B>	S <A>B>	Not zero if A is greater than B, 0 otherwise

Unary operations

Name	Stack before	Stack after	Notes
ir_not	S <A>	S <!A>	Zero if A is not zero, Not zero if A is zero

Control flow

Operation	Immediates	Stack before	Stack after	Description
NOP		S	S	No-operation, does perform any computation, does not modify the stack
BRANCH	ir_label *	<int> S	S	Jumps to the label specified in the immediate if the value on top of the stack is zero
JUMP	ir_label *	S	S	Jumps to the label specified in the immediate. Leaves the stack unchanged
LABEL	ir_label *	S	S	Adds a label, specified in the immediate, that other IR instructions can jump. Leaves the stack unchanged

Here is an example of conditional control flow:

Source code	**IR**	**Stack Before**	**Stack After**	**Notes**
`if (<cond>)` `  <then>`	`<cond>`	`S`	`int` `S`	`Should leave the result of evaluating the condition on top of the stack, either zero or non-zero`
	`Branch(L0)`	`int` `S`	`S`	`Jumps to the label L0 if the result on top of the stack is zero, otherwise continues to execute`
	`<then>`	`S`	`S`	
	`Label(L0)`	`S`	`S`	`Target of the branch, needs to be added to the IR to denote where the <then> part of the program ends.`

Intrinsics

Operation	Immediates	Stack before	Stack after	Description
INTRINSIC	enum intrinsic	<arg> S	S	Performs the intrinsic operation specified in the immediate (see operations below). Takes one argument from the top of the stack.

Aves supports 3 types of intrinsic (built-in) functions:

Name	Stack before	Stack after	Notes
INTRINSIC_EXIT	S <int>		Exits with the exit code on top of the stack
INTRINSIC_PRINT_INT	S <int>	S	Prints the int on top of the stack
INTRINSIC_PRINT_STRING	S <addr>	S	Prints the zero-terminated string which address is on top of the stack

Variables

Operation	Immediates	Stack before	Stack after	Description
RESERVE	int size char * name	S	S	Reserves memory for a global variable. Leaves the stack unchanged. The immediate values specify the name, and the size in bytes.
ADDR_GLOBAL	char * name	S	<addr> S	Pushes the address of the global variable, referenced in the immediate char*, to the top of the stack
READ		<addr> S	<int> S	Reads the integer value of the variable specified with an address on top of the stack, leaving the value on top of the stack.
WRITE		<newval> <addr> S	S	Writes the value on top of the stack to the variable specified by the address on the second position of the stack.

Example, in which each address is color-coded with the read/write operation that uses it:

Source code	IR	Stack Before	Stack After
`a = a + 1;`	`addr_global "a"`	`S`	`<addr &a>` `S`

	`addr_global "a"`	`<addr &a>`
		`S`
			`<addr &a>`
			`<addr &a>`
			`S`

(continues on next page)

`a = a + 1;`
 	
`(continued)`
	`read`	`<addr &a>`
		`<addr &a>`
		`S`
			`<int>`
			`<addr &a>`
			`S`

	`iconst 1`	`<int>`
		`<addr &a>`
		`S`
			`<int>`
			`<int>`
			`<addr &a>`
			`S`

	`ops(ir_add)`	`<int>`
		`<int>`
		`<addr &a>`
		`S`
			`<int>`
			`<addr &a>`
			`S`

	`write`	`<int>`
		`<addr &a>`
		`S`
			`S`

Functions

Operation	Immediates	Stack before	Stack after	Description
FUNCION	ir_label * int	S	S	Defines where a function starts, named using the label in the immediate, and reserves space for N local variables, specified in the immediate
CALL	ir_label * int	<argN> … <arg2> <arg1> <int1> S	<int2> ret	Calls the function specified by the label in the immediates, consuming N+1 elements from the stack where N is the number of arguments provided to the instruction. Leaves a return integer on top of the stack.
RET		<int> S	S	Returns from the given function with the value on top of the stack. It is not possible to return from top-level code.
ADDR_LOCAL	Int	S	<addr> S	Reads the address of a local variable or argument specified by its index as an immediate, leaves it on top of the stack.

Function arguments are passed via the stack. The first argument on the stack should be an arbitrary number (to reserve stack space for the return value). Then, the arguments must be on the stack, in the same order in which the function declares them. Finally, a CALL instruction specifies how many arguments to pass to the function, and which function to call. Upon return from a CALL instruction, the return is left on top of the stack. In the IR, all functions return an integer.

Inside a function, each argument can be accessed by their index. The first argument is at index 0, the second argument is at index 1, etc. Reading an argument pushes it to the top of the stack. The return instruction returns the value on top of the stack, and jumps to the instruction immediate after the CALL that resulted in invoking the function. As such, it stops executing the current function.

Example:

Source code	Function IR	Call IR

`int add3(int a, int b, int c)`
`{`
`  return a+b+c;`
`}`
 	
`print_int(add3(400,70,3));`
	`Function("add3", 0);`
	`ArgLocal(0);`
	`ArgLocal(1);`
	`Op(ir_add);`
	`ArgLocal(2);`
	`Op(ir_add);`
	`Return();`
		`Iconst(<any number>);`
		`Iconst(400);`
		`Iconst(70);`
		`Iconst(3);`
		`Call("add3",3);`
		`Intrinsic(intrinsic_print_int);`

Stack manipulation

Operation	Immediates	Stack before	Stack after	Description
POP		<int> S	S	Discards the value on top of the stack
DUP		<val> S	<val> <val> S	Duplicates the value on top of the stack
A2I		<addr> S	<int> S	Turns an address on top of the stack into an integer, representing the same value, also left of top of the stack
I2A		<int> S	<addr> S	Turns an integers on top of the stack into an address, representing the same value, also left on top of the stack

(continues on next page)

AVES interpreter

The provided AVES interpreter can execute a human readable AVES file, and it supports all the operations above with the same names. The interpreter uses 8 bytes to represent integers and IR programs that reserve less memory will result in a crash flagged by the address sanitizer (example shown below).

Bugs

Most crashes of the interpreter are likely to be a bug in the IR program (as described above, not reserving enough memory for a variable) and very unlikely to be a bug in the interpreter itself. When reporting "bugs", please include the minimal IR program that shows the "bug".

```
=================================================================
==764040==ERROR: AddressSanitizer: heap-buffer-overflow on address 0x7bb5875e0070 at
pc 0x55885d3e68b2 bp 0x7ffc414492d0 sp 0x7ffc414492c0
WRITE of size 8 at 0x7bb5875e0070 thread T0
    #0 0x55885d3e68b1 in interp_list interpret.c:519
    #1 0x55885d3e7ed3 in interpret interpret.c:825
    #2 0x55885d3e4665 in main main_interp.c:27
    #3 0x7f95888376b4  (/usr/lib/libc.so.6+0x276b4)
    #4 0x7f9588837768 in __libc_start_main (/usr/lib/libc.so.6)
    #5 0x55885d3e4394 in _start (interp)
```

Writing strings to global variables

When writing a string, represented as an address on top of the stack, to a global variable, the AVES interpreter will actually copy the contents of the string to the memory reserved for that global variable. It will copy all the string, which will trigger an address sanitizer error if there is not enough memory reserved on the global variable.

The following program will write "CS473" to variable "s":

Source code	**IR**	**Stack After**	**Notes**
`var string s := "CS473";`	`Reserve("s", 6)`	`S`	`strlen("CS473")=5 because it doesn't count the` `\0` `character`
`printstring(s);`	`Addr_global("a")`	`<addr &a>` `S`	

	`Sconst("CS473")`	`<addr &"CS473">`
		`<addr &a>`
		`S`	

	`Write`	`S`	`Copies the contents of "CS473" to s`
	`Addr_global("a")`	`<addr &a>` `S`	
	`Intrinsic PRINT_STRING`	`S`	`Prints "CS473"`