

SWPP Project:

Restrictions In the Source Program

- Apr. 23: read/write functions added, argc/argv removed

In this project, we're going to give simplified IR programs as inputs only.

1. A source program has only i1, i8, i16, i32, i64, array types, and pointer types.
2. There is no linking; the source program consists of a single IR file.
3. The IR file contains multiple functions including a main function.
4. A function can have at most 16 arguments.
5. There is no function attribute (e.g. readonly).
6. A source program takes an input through read() calls. read() returns i64 value.
7. The output of the program is done via write(i64) calls.

SWPP Project:

Backend Assembly Language

- Apr. 28: A few updates in the language syntax:
 - The name of a basic block / function should contain alphabets(a-zA-Z) + digits(0-9) + underscore(_) + hyphen(-) + dot(.) only.
 - br/switch instructions don't use comma(,) as separator anymore.
 - A comment line (주석) starts a semicolon(;) (or a space followed by a semicolon).
 - switch's case values are all constant integers.
 - Add `assert_eq x y` which checks `x == y` and aborts if fails
- Apr. 28: A few updates in the language semantics:
 - A function always returns an i64 value.
 - Bytes in the heap/stack area are initialized as 0.
- Apr. 30: Binary operators / Integer comparisons now take the size of bitwidth of inputs (needed for correct evaluation of signed operations). The result is zero-extended.
- May. 1: Minor (grammatical) fixes in `assert_eq`, integer shift operations

1. Architecture Overview

(1) Registers

- There are 17 64-bit registers. They are named as r1, r2, ..., r16, and sp.
- A register can be assigned multiple times.

(2) Memory

Loads and stores.

- The memory is accessed via load/store instructions with 64-bit pointers.
- The structure of memory is analogous to a tape. Whenever load or store is executed, there is an additional cost to move the head to the address. The cost is proportional to the distance from the previous access.

Stack.

- The stack area starts from address 10240, grows downward (-).
- The stack area is initialized as 0.

- You can use sp register to store the address of the *stack frame*, but it is not necessary to do so.

Heap.

- The heap area starts from address 20480, grows upward (+).
- Heap allocation (malloc) initializes the area as 0.
- Accessing unallocated heap raises an error.
- Accessing the area between [10240, 20480) raises an error.

Global Variables.

- Global variables are placed at the beginning of the heap.
- They are initialized as 0.

(3) Function calls

- Function arguments can be accessed via read-only registers arg1.. arg16.
- When a call instruction is executed, r1 ~ r16, sp registers are automatically saved and restored.
- When a call instruction is executed, the arguments are automatically assigned to the registers arg1 ~ arg16.

2. Function & Basic Block Definition & Comments

- As in LLVM IR, a function consists of one or more basic blocks.
- 'start <funcname> <argN>:' starts a function. argN describes the number of arguments.
- 'end <funcname>' finishes the function.
- <funcname> is valid only if it is a non-empty string consisting of alphabets(a-zA-Z) + digits(0-9) + underscore(_) + hyphen(-) + dot(.).
- A function always return i64.
- When a function is called, caller's register's values are preserved.

Description	Syntax
Start function definition	start <funcname> <argN>:
End function definition	end <funcname>

- A basic block BBNAM starts with '. BBNAM:'.
- A basic block should end with a terminator instruction, which will be described later.

- BBNAME is valid only if it is a non-empty string consisting of alphabets(a-zA-Z) + digits(0-9) + underscore(_) + hyphen(-) + dot(.).

Description	Syntax
Basic block start	.BBNAME :

- There should be a linebreak after all of these commands.
- There is no nested function.

- A comment starts with a semicolon(;).
- Before semicolon, only space can come.

Description	Syntax
Comment	; Here is a comment.

3. Instructions

Syntax:

$$\langle \text{reg} \rangle = \text{op_name } \langle \text{val1} \rangle \dots \langle \text{valN} \rangle$$

- $\langle \text{reg} \rangle$ is the name of a register to assign the result.
- op_name is the name of the assembly instruction.
- $\langle \text{val} \rangle$ is either an integer constant or a register. k -th operand of an instruction is described as $\langle \text{val}k \rangle$.
- For some instructions, $\langle \text{ptr} \rangle$ is used instead of $\langle \text{val} \rangle$ to clarify the meaning of the operand.
- If an argument register ($\text{arg1} \sim \text{arg16}$) is used, the cost increases by 1.

(1) Memory instructions

Kind	Syntax	Cost
Heap Allocation	$\langle \text{reg} \rangle = \text{malloc } \langle \text{val} \rangle$	1
Deallocation	$\text{free } \langle \text{ptr} \rangle$	1
Load $\langle \text{ofs} \rangle$ should be a decimal constant.	$\langle \text{reg} \rangle = \text{load } \langle \text{size} \rangle \langle \text{ptr} \rangle \langle \text{ofs} \rangle$ $\langle \text{size} \rangle := 1 2 4 8$	Stack area: 2 Heap area: 4
Store $\langle \text{ofs} \rangle$ should be a decimal constant.	$\text{store } \langle \text{size} \rangle \langle \text{val} \rangle \langle \text{ptr} \rangle \langle \text{ofs} \rangle$ $\langle \text{size} \rangle := 1 2 4 8$	Stack area: 2 Heap area: 4
Reset tape access pin	$\text{reset } [\text{stack} \text{heap}]$	2

load/store.

- The load instruction reads the data at $[\langle \text{ptr} \rangle + \langle \text{ofs} \rangle, \langle \text{ptr} \rangle + \langle \text{ofs} \rangle + \langle \text{size} \rangle)$, zero-extends it to an 64-bit integer, and returns it.
- The store instruction truncates the value $\langle \text{val} \rangle$ to an $\langle \text{size} \rangle * 8$ -bit integer and writes it at $[\langle \text{ptr} \rangle + \langle \text{ofs} \rangle, \langle \text{ptr} \rangle + \langle \text{ofs} \rangle + \langle \text{size} \rangle)$.
- load and store has an additional cost for moving the head to the address.
$$\text{cost} = 0.0004 * |\text{current address} - \text{previous address}|$$
- $\langle \text{ptr} \rangle$ and $\langle \text{ofs} \rangle$ should be a multiple of $\langle \text{size} \rangle$.
- The memory is little-endian; store writes the 8 least significant bits to $\langle \text{ptr} \rangle + \langle \text{ofs} \rangle$. Similarly, the 8 least significant bits of the value read by load is the byte at $\langle \text{ptr} \rangle + \langle \text{ofs} \rangle$.

malloc.

- It allocates a space of the given size to the heap.

- It does not move the head.
- The size of malloc should be a multiple of 8.
- malloc finds an empty space with the smallest address in the heap area & allocates it.
- The returned address is a multiple of 8.

reset.

- It moves the head to the beginning of stack or heap.
- Its cost is fixed (it's not proportional to the distance from the previous address).

(2) Terminator instructions

Terminator instructions should come at the end of a basic block only.

<bb> stands for a basic block name to jump to.

Terminator cannot jump to a block in other functions.

Kind	Syntax	Cost
Return Value - ret is equivalent to ret 0.	ret ret <val>	1
Unconditional Branch	br <bb>	1
Conditional Branch	br <condition> <>true_bb> <>false_bb>	1
Switch Instruction - <val_1>, ... should be constant integers.	switch <cond_val> <val1> <bb1> .. <default_bb>	1

(3) Other instructions

Kind	Name	Cost
Integer Multiplication/Division	$\langle \text{reg} \rangle = \text{udiv } \langle \text{val1} \rangle \langle \text{val2} \rangle \langle \text{bw} \rangle$ $\langle \text{reg} \rangle = \text{sdiv } \langle \text{val1} \rangle \langle \text{val2} \rangle \langle \text{bw} \rangle$ $\langle \text{reg} \rangle = \text{urem } \langle \text{val1} \rangle \langle \text{val2} \rangle \langle \text{bw} \rangle$ $\langle \text{reg} \rangle = \text{srem } \langle \text{val1} \rangle \langle \text{val2} \rangle \langle \text{bw} \rangle$ $\langle \text{reg} \rangle = \text{mul } \langle \text{val1} \rangle \langle \text{val2} \rangle \langle \text{bw} \rangle$ $\langle \text{bw} \rangle := 1 8 16 32 64$	0.6
Integer Shift/Logical Operations - shl: shift-left	$\langle \text{reg} \rangle = \text{shl } \langle \text{val1} \rangle \langle \text{val2} \rangle \langle \text{bw} \rangle$ $\langle \text{reg} \rangle = \text{lshr } \langle \text{val1} \rangle \langle \text{val2} \rangle \langle \text{bw} \rangle$ $\langle \text{reg} \rangle = \text{ashr } \langle \text{val1} \rangle \langle \text{val2} \rangle \langle \text{bw} \rangle$	0.8

- lshr: logical shift-right - ashr: arithmetic shift-right	<code><reg> = and <val1> <val2> <bw></code> <code><reg> = or <val1> <val2> <bw></code> <code><reg> = xor <val1> <val2> <bw></code> <code><bw> := 1 8 16 32 64</code>	
Integer Add/Sub	<code><reg> = add <val1> <val2> <bw></code> <code><reg> = sub <val1> <val2> <bw></code> <code><bw> := 1 8 16 32 64</code>	1.2
Comparison - <cond> is equivalent to the cond of LLVM IR's icmp	<code><reg> = icmp <cond> <val1> <val2></code> <code><bw> := 1 8 16 32 64</code>	1
Ternary operation	<code><reg> = select <val_cond></code> <code><val_true> <val_false></code>	1.2
Function call	<code>call <fname> <val1> .. <valN></code> <code><reg> = call <fname> <val1> ..</code> <code><valN></code>	2 * (1 + arg #)
Assertion	<code>assert_eq <val1> <val2></code>	0

- For integer arithmetic and comparison operations, <size> is the size of bitwidth of inputs that the operation assumes. For example, `ashr 522 2 8` takes the lowest 8-bits from inputs (which is 255 = -1), performs arithmetic right shift, and zero-extends it to 64 bits. So, its result is 255.