

SWPP Project:

Restrictions In the Source Program

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 argc & argv arguments of the main function.
7. The output of the program is the return value of the main function, which is i64 type.

SWPP Project:

Backend Assembly Language

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 (-).
- 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 (+).
- 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.

(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

- 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.

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

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

Description	Syntax
Basic block start	.BBNAME:

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

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 ret <val>	1
Unconditional Branch	br <bb>	1
Conditional Branch	br <condition>, <true_bb>, <false_bb>	1
Switch Instruction	switch <cond_val>, <val_1>, <bb1>, <val_2>, <bb2>, ..., <default_bb>	1

(3) Other instructions

Kind	Name	Cost
Integer Multiplication/Division	<reg> = udiv <val1> <val2> <reg> = sdiv <val1> <val2> <reg> = urem <val1> <val2> <reg> = srem <val1> <val2> <reg> = mul <val1> <val2>	0.6
Integer Shift/Logical	<reg> = shl <val1> <val2>	0.8

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