SWPP Project:
Restrictions In the Source Program

- Apr. 23: read/write functions added, argc/argv removed
- May. 12:
    - read()/write() are stdin/stdout
    - The size given to malloc is a multiple of 8
    - The test programs will never raise out-of-memory with the test inputs if compiled with the project skeleton.
    - The main function is never called recursively.

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.
8. read() / write() calls are connected to stdin/stdout.
9. The size given to malloc() is a multiple of 8.
10. The test programs will never raise out-of-memory or stack overflow with the given inputs, if compiled with the project skeleton.
11. main() is never called recursively.

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
- May. 12: A few updates from Q&As:
    - Describe the cost of the first load/store
    - argN cannot be at LHS of an instruction
    - r1 ~ r16 are preserved at the function call,
    - There is no syntax for global variables (they are only in IR)
    - The total cost considers memory usage, it doesn't consider code size.
    - Fix typo in the ashr example
    - There is no variadic function.

1. Architecture Overview

(1) Registers

- There are 17 64-bit general 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.
- Syntactically, there is no difference between global variables and heap allocated blocks.
- The project skeleton will lower global variables to heap allocations in the beginning of main(). So, they are placed at the beginning of the heap area.

(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.
- arg1 ~ arg16 cannot be assigned in the callee.
- The value of r1 ~ r16 is preserved inside the callee.

(4) Cost

- The total cost of an execution of a program is a summation of costs of the executed instructions + maximum memory usage cost.
- The memory usage cost is the maximum value of heap allocated byte size at any moment.
- For example, the memory usage cost of `` `r1 = malloc 8; free r1; r2 = malloc 8; free r2` `` is 8, because the maximum memory usage is 8.
- The total cost does not consider the code size.

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.
- There is no variadic 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.
- 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:

```
<reg> = op_name <val1> .. <valN>
```

- `<reg>` is the name of a register to assign the result.
- `op_name` is the name of the assembly instruction.
- `<val>` is either an integer constant or a register. *k*-th operand of an instruction is described as `<val`*k*`>`.
- For some instructions, `<ptr>` is used instead of `<val>` to clarify the meaning of the operand.
- If an argument register (arg1 ~ arg16) is used at `<val>`, the cost increases by 1.
- If `<reg>` is argN, its execution raises an error.

(1) Memory instructions

Kind	**Syntax**	**Cost**
Heap Allocation	`<reg> = malloc <val>`	1
Deallocation	`free <ptr>`	1
Load <ofs> should be a decimal constant.	`<reg> = load <size> <ptr> <ofs>` `<size> := 1|2|4|8`	Stack area: 2 Heap area: 4
Store <ofs> should be a decimal constant.	`store <size> <val> <ptr> <ofs>` `<size> := 1|2|4|8`	Stack area: 2 Heap area: 4
Reset tape access pin	`reset [stack|heap]`	2

load/store.

- The `load` instruction reads the data at [`<ptr>+<ofs>` , `<ptr>+<ofs>+<size>`), zero-extends it to an 64-bit integer, and returns it.
- The `store` instruction truncates the value `<val>` to an `<size>`*8-bit integer and writes it at [`<ptr>+<ofs>` , `<ptr>+<ofs>+<size>`).
- `load` and `store` has an additional cost for moving the head to the address.

  cost = 0.0004 * |current address - previous address|

- `<ptr>` and `<ofs>` should be a multiple of `<size>`.
- The memory is little-endian; `store` writes the 8 least significant bits to `<ptr>+<ofs>`. Similarly, the 8 least significant bits of the value read by `load` is the byte at `<ptr>+<ofs>`.

- The cost of moving the head to the address is zero for the first load or store instruction during the program execution. For example, if there is a load to a stack area followed by store to a stack area, the first load only has cost 2, but the second store has cost 2 + head relocation cost.

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	`<reg> = udiv <val1> <val2> <bw>`
	`<reg> = sdiv <val1> <val2> <bw>`
	`<reg> = urem <val1> <val2> <bw>`
	`<reg> = srem <val1> <val2> <bw>`
	`<reg> = mul  <val1> <val2> <bw>`
	`<bw> := 1|8|16|32|64`
		0.6

Integer Shift/Logical Operations
- shl: shift-left
- lshr: logical shift-right
- ashr: arithmetic shift-right
	`<reg> = shl  <val1> <val2> <bw>`
	`<reg> = lshr <val1> <val2> <bw>`
	`<reg> = ashr <val1> <val2> <bw>`
	`<reg> = and <val1> <val2>  <bw>`
	`<reg> = or  <val1> <val2>  <bw>`
	`<reg> = xor <val1> <val2>  <bw>`
	`<bw> := 1|8|16|32|64`
		0.8

Integer Add/Sub	`<reg> = add <val1> <val2>  <bw>`
	`<reg> = sub <val1> <val2>  <bw>`
	`<bw> := 1|8|16|32|64`
		1.2

Comparison - `<cond>` is equivalent to the `cond` of LLVM IR's icmp	`<reg> = icmp <cond> <val1> <val2> <bw>` `<bw> := 1|8|16|32|64`	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 #)

Assertion	`assert_eq <val1> <val2>`	0

- For integer arithmetic and comparison operations, `<size>` is the size of bitwidth of inputs that the operation assumes. For example, `` `ashr 511 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.