Assembly Language Specification

2021 Spring, SWPP

1. Architecture Overview

- An architecture consists of a single-core CPU and 64-bit memory space.

(1) Registers

- There are 33 64-bit general registers. They are named `r1`, `r2`, .., `r32`, and `sp`.
- `r1`, `r2`, .., `r32` are initialized to 0 and `sp` is initialized to 102400.
- A register can be assigned multiple times (it isn't SSA).

(2) Memory

Loads and stores.
- The memory is accessed via `load`/`store`/`vload`/`vstore` instructions with 64-bit pointers.
- Frequently accessing a memory location will increase the temperature of its and nearby locations.
- An access to a hot location has a bigger cost than an access to a cool location.
- The exact formula for its calculation is described later.

Stack.
- The stack area starts from address 102400, grows downward (-), and is initialized as 0 in the beginning of the program execution.
- You can use `sp` to store the address of the current stack frame, but it is not necessary to do so.

Heap.
- The heap area starts from address 204800, grows upward (+).
- Heap allocation (`malloc`) initializes the area as zero.
- Accessing an unallocated heap raises an error.
- Accessing the area between [102400, 204800) raises an error.

Global Variables.
- Syntactically, there is no difference between global variables and heap-allocated blocks.
- The project skeleton lowers a global variable to a heap allocation (malloc call) at 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`.

Calling convention.

- When a `call` instruction is executed,
  - `r1` ~ `r32`, `sp` registers are automatically saved in an invisible space (you don't need to manually spill them).
  - Values of the arguments are automatically assigned to the registers `arg1` ~ `arg16`.
  - The values of `r1` ~ `r32` are unchanged (not initialized to 0).
- After the call returns, `r1` ~ `r32`, sp registers are automatically restored.

(4) Cost

- The execution cost of a program can be calculated as 'program-wide instruction execution cost + maximum heap memory usage (in bytes)'.
- The code size is irrelevant to the total cost.

Memory usage cost.

- The memory usage cost is the maximum 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.

Compile time.

- Compile time should be less than 1 minute.

2. Input Program

Structure.

- The source program consists of a single IR file; There is no linking.
- The IR file consists of one or more functions, including the main function.
- A source program only uses i1, i8, i16, i32, i64, array types, and pointer types.

Function.

- A function can have at most 16 arguments.
- There is no function attribute (e.g. read-only).
- `main()` is never called recursively.

Standard I/O.

- A source program takes input through `read()` calls. `read()` reads an integer and returns it as a 64 value.
- The output of the program is done via `write(i64)` calls. It writes the output as an unsigned integer in a new line.
- `read()` / `write()` calls are connected to the standard input/output.

Misc.

- The test programs will never raise out-of-memory or stack overflow with the given inputs if compiled with the project skeleton.

3. Function & Basic Block

(1) Function

Syntax:

```
start <funcname> <Narg>:
... (basic blocks)
end <funcname>
```

- A function contains one or more basic blocks.
- `<funcname>` is a non-empty string consisting of alphabets(a-zA-Z), digits(0-9), underscore(_), hyphen(-), or dot(.).
- `<Narg>` describes the number of arguments.
- A function's return type is always i64.
- There is no variadic function.
- There is no nested function.

(2) BasicBlock

Syntax:

```
<bbname>:
... (instructions)
```

- A basic block consists of one or more instructions.
- A basic block must end with a terminator instruction (see below for more details)
- `<bbname>` is a non-empty string, starting with a dot(.) and consists of alphabets(a-zA-Z) + digits(0-9) + underscore(_) + hyphen(-) + dot(.).

(3) Comment

Syntax:

```
; <comment>
```

- A comment starts with a semicolon(;).
- Only space characters are allowed before the semicolon in the line.

4. Instructions

Syntax:

```
<reg1> .. <regM> = op_name <val1> .. <valN>
```

- `<reg`*k*`>` is the name of a register to assign the result.
- Vector instructions can have more than one register in the LHS whereas non-vector instructions can have at most one register in the LHS[1].
- `<val>` is either an integer constant or a register. `<val`*k*`>` is the *k*-th operand of the instruction.
- Argument registers (e.g. `arg1`) cannot be placed at the LHS.

(1) Terminator instructions

Kind	Syntax	Cost
Return Value - `ret` is equivalent to `ret 0`.	`ret` `ret <val>`	1
Unconditional Branch	`br <bbname>`	1
Conditional Branch	`br <condition> <true_bb> <false_bb>`	4 for `true_bb` 1 for `false_bb`
Switch Instruction - `<val1>`, … should be constant integers.	`switch <cond_val> <val1> <bb1> .. <default_bb>`	2

- Terminator instructions should come at the end of a basic block only.
- `<bbname>` stands for a basic block name to jump to.
- Branches/switch cannot jump to a block in another function.
- `ret` does not reset the temperature of the previously used stack area.

(2) Memory allocation/deallocation

Kind	Syntax	Cost
Heap Allocation	`<reg> = malloc <val>`	8
Deallocation	`free <reg>`	8

[1] It is allowed for call to not have a LHS.

malloc.
- malloc allocates a space of the given size to the heap. The space is initialized to zero and has zero temperature.
- The size of malloc should be non-zero & a multiple of 8.
- malloc finds an empty consecutive space with the smallest address in the heap area & allocates it.
- The returned address by malloc is a multiple of 8.

free.
- free deallocates a space associated with the given pointer.
- The pointer passed to free should point to the beginning of an allocated heap space.

(3) Memory access (non-vector)

Kind	Syntax	Base Cost
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

load.
- The load instruction reads the data at [<ptr>+<ofs> , <ptr>+<ofs>+<size>), zero-extends it to 64 bits, and returns it.
- <ptr> and <ofs> should be multiple of <size>.
- The memory is *little-endian*. The least significant byte of the value read by load is from <ptr>+<ofs>, and the most significant byte is from <ptr>+<ofs>+<size>-1.

store.
- The store instruction truncates the value <val> to an <size>*8-bit integer and writes it at [<ptr>+<ofs> , <ptr>+<ofs>+<size>).
- <ptr> and <ofs> should be multiple of <size>.

The cost of load and store.
- The cost of accessing the location is a summation of the base cost (which is 2 if stack and 4 if heap) and the cost from the temperature of the location (<ptr>+<ofs>).
- Given T(L) which is the temperature at location L before executing an instruction i, the cost of a load/store whose <ptr>+<ofs> is L is calculated as follows:

```
base_cost + 0.1 * T(L)
```

- The temperature T'(L) after executing i is calculated as follows:

```
T'(L) = min(T(L)+25, 200)   if i is a load/store to L
        max(T(L)-1,  0)     otherwise
```

- The temperature is controlled in a coarse-grained manner.
    - The memory is divided into 8 bytes.
    - For any address i, [8*⌊i/8⌋, 8*⌊i/8⌋+7] has the same temperature
    - Access to any location in the range will equally increase the temperatures.

(4) Memory access (vector)

<table>
<tr><th>Kind</th><th>Syntax</th><th>Base Cost</th></tr>
<tr><td>Vector Load
<ofs> should be a decimal constant.</td><td>(<reg1>|_) .. (<regN>|_) = vload <N> <ptr> <ofs>
<N> := 2|4|8</td><td>Stack area: 2.8
Heap area: 4.8</td></tr>
<tr><td>Vector Store
<ofs> should be a decimal constant.</td><td>vstore <N> (<val1>|_) .. (<valN>|_) <ptr> <ofs>
<N> := 2|4|8</td><td>Stack area: 2.8
Heap area: 4.8</td></tr>
</table>

vload.

- The vload instruction reads <N> words (of 8 bytes) at [<ptr>+<ofs>, <ptr>+<ofs>+<N>*8) and stores the results at the registers accordingly.
- vload can accept placeholder (_) instead of a register at LHS. In this case, vload does not touch the memory location. For example,

```
r1 _ r3 r4 = vload 4 arg1 0
```

accesses arg1, arg1+16, arg1+24, but not arg1+8.
- <ptr> and <ofs> should be multiple of 8.

vstore.

- The vstore instruction writes <N> words (of 8 bytes) at [<ptr>+<ofs>, <ptr>+<ofs>+<N>*8).
- vstore can accept placeholder (_) instead of a value to store. In this case, vstore does not touch the memory location. For example,

```
vstore 4 10 _ 30 40 arg1 0
```

stores 10 at arg1, 30 at arg1+16, 40 at arg1+24, but writes nothing to arg1+8.

The cost of vload and vstore.

- When vload/vstore accesses multiple memory lines simultaneously
    - the highest temperature from the touched locations is used to calculate the cost (which is base_cost + 0.1 * T(L)).

- The touched memory locations have increased temperature.

(5) Temperature manipulation

Kind	**Syntax**	**Cost**
Cool down memory	cool <val>	10

cool.

- cool initializes the temperature of [<val>, <val>+8) into zero.
- <val> should be multiple of 8.

(6) 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	1
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	2.8
Integer Add/Sub	<reg> = add <val1> <val2> <bw> <reg> = sub <val1> <val2> <bw> <bw> := 1|8|16|32|64	3.2
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>	2 + arg #

	<reg> = call <fname> <val1> .. <valN>	
Assertion	assert_eq <val1> <val2>	

- 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.
- assert_eq does not drop temperature.