Assembly Language Specification

2022 Spring, SWPP

Updates.

Apr. 19: `await` instruction is removed

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/aload` instructions with 64-bit pointers.
- The exact formula for its calculation is described later.
- An asynchronous memory load can be performed using `aload` (asynchronous load)

Stack.

- The stack area starts from address 102400, grows downward (-), and is initialized as 0 at 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 and 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 the main(). So, they are placed at the beginning of the heap area.

(3) Function calls

- Function arguments can be accessed via <u>read-only</u> 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 16 times the maximum 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 16 * 8 = 128, because the maximum memory usage is 8 bytes.

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 an i64 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(i64)` 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:

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

- `<reg>` is the name of a register to assign the result.
- `<val>` is one of integer constant, bbname, 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>`	6 for `true_bb` 1 for `false_bb`
Switch Instruction - `<val1>`, … should be constant integers.	`switch <cond_val> <val1> <bb1> .. <default_bb>`	1.2

- 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>`	16
Deallocation	`free <reg>`	16

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 allocated heap space.

(3) Memory access

Kind	**Syntax**	**Base Cost**

Load	`<reg> = load <size> <ptr>`
	`<size> := 1|2|4|8`
		Stack area: 6
		Heap area: 12

Store	`store <size> <val> <ptr>`
	`<size> := 1|2|4|8`
		Stack area: 6
		Heap area: 12

Async Load	`<reg> = aload <size> <ptr>`
	`<size> := 1|2|4|8`
		Stack area: 1
		Heap area: 1
		
		Cost until being resolved
		Stack area: 10
		Heap area: 16

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

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

asynchronous load.
- The `aload` instruction behaves exactly the same as an ordinary load, except that one should wait until the loaded value is resolved to use it.

```
    r2 = aload 8 r1
    r3 = add r2 1 64    ; waits for cost 10 (stack) or 16 (heap)
```

- After executing an `aload` instruction, it takes the cost *n* = 10 for stack area and *n* = 16 for heap area for the returning register to be ready. e.g.,

```
    r2 = aload 8 r1  ; suppose r1 contains an address to heap area
    ...              ; instructions here take cost m (no use of r2)
    call write r2    ; waits for 16 - m
```

  Total cost = 1 (for `aload`) + *m* + 3 (for `call`) + max(0, 16 - *m*) (for waiting)

- Accesses to addresses overlap an unresolved async load is allowed. e.g.,

```
    store 8 3 r1
    r2 = aload 8 r1
    store 8 42 r1
    r3 = load 8 r1    ; loads 42
    call write r2     ; prints 3 after the loaded value (r2) is resolved
```

- Overwriting a register that is waiting for an async load is allowed. e.g.,

```
    store 8 3 r1      ; suppose r1 contains an address to stack area
    r2 = aload 8 r1
    r2 = add 42 0 64  ; does not wait for aload to be resolved
    call write r2     ; prints 42
```

  Total cost = 6 (for `store`) + 1 (for `aload`) + 5 (for `add`) + 3 (for `call`)

(4) 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`
		4

Integer Add/Sub	`<reg> = add <val1> <val2> <bw>`
	`<reg> = sub <val1> <val2> <bw>`
	`    <bw> := 1|8|16|32|64`
		5

Integer Sum	`<reg> = sum <val1> … <val8> <bw>`
	`    <bw> := 1|8|16|32|64`
		5.2

Integer increment `<reg>` = `<reg>` + 1	`incr <reg> <bw>`	1
Integer decrement `<reg>` = `<reg>` - 1	`decr <reg> <bw>`	1

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 + arg #

Assertion An assertion fail is an error Used for testing	`assert_eq <val1> <val2>`	0

- For integer arithmetic and comparison operations, `<bw>` 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.