Assembly Language Specification
2024 Spring, SWPP

Updates.

1. Architecture Overview

This 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.
- Also, there are 16 vector registers. They are named `v1, v2, …, v16`.
- `v1, v2, …, v16` are initialized to 0.
- Registers are *big-endian*.
- A register can be assigned multiple times (it isn't SSA).

(2) Memory

Loads and stores.
- The memory is accessed via `load`/`store` instructions with 64-bit pointers.
- The exact formula for the cost calculation will be described later.

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 read-only registers `arg1`.. `arg16`.

Calling convention.
- When a `call` or `rcall` instruction is executed,
    - `r1` ~ `r32`, `v1` ~ `v16`, `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`.
    - Values in registers do not change (they don't get initialized to 0).
- After the call returns, register values are automatically restored to the time before the call.

(4) Cost

- The execution cost of a program can be calculated as 'program-wide instruction execution cost + maximum heap memory usage (in bytes)' * 1024.
- The code size is irrelevant to the total cost.

Memory usage cost.
- The memory usage cost is 1024 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 1024 * 8 = 8192, 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, pointer types, and vector 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) Basic Block

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 spaces are allowed before the semicolon in the line.

4. Instructions

Syntax:

```
op_name <reg1> .. <regN>
<reg> = op_name <reg1> .. <regN>
```

- `<reg>`, `<regN>`, `<reg_*>` stands for a scalar register.
- `<vr>`, `<vrN>`, `<vr_*>` stands for a vector register.
- `<cst>`, `<cstN>`, `<cst_*>` stands for a nonnegative constant integer (< 2^64).
- Argument registers (e.g. `arg1`) cannot be placed at the LHS.

(1) Control Flow

Kind	Syntax	Cost
Return Value - `ret` is equivalent to `ret 0`.	`ret` `ret <reg>`	1
Unconditional Branch	`br <bbname>`	30
Conditional Branch	`br <reg_cond> <true_bb> <false_bb>`	90 for `true_bb` 30 for `false_bb`
Switch Instruction	`switch <reg_cond> <cst1> <bb1> .. <default_bb>`	60
Function call	`call <fname> <reg1> .. <regN>` `<reg_ret> = call <fname> <reg1> .. <regN>`	30
Recursive function call	`rcall <reg1> .. <regN>` `<reg_ret> = rcall <reg1> .. <regN>`	10
Assertion An assertion fail is an error. Used for testing	`assert_eq <reg1> <reg2>` `assert_eq <reg> <cst>`	0

- `ret, br, switch` instructions should come at the end of a basic block only.
- `<bbname>, <bbN>, <*_bb>` is the name of a basic block to jump to.
- `<fname>` is the name of a function to call.
- `rcall` calls the function that this instruction resides in.
- `br / switch` cannot jump to a block in another function.

(2) Memory Allocation/Deallocation

Kind	Syntax	Cost
Heap Allocation	`<reg_ptr> = malloc <reg_size>`	150
Deallocation	`free <reg_ptr>`	150

malloc
- The size of `malloc` should be non-zero & a multiple of 8.
- 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 <sz> <reg_ptr>` `<sz> := 1|2|4|8`	Stack area: 30 Heap area: 50
Store	`store <sz> <reg> <reg_ptr>` `<sz> := 1|2|4|8`	Stack area: 30 Heap area: 50
Vector Load	`<vr> = vload <reg_ptr>`	Stack area: 60 Heap area: 100
Vector Store	`vstore <vr> <reg_ptr>`	Stack area: 60 Heap area: 100

- Value in `<reg_ptr>` should be multiple of `<sz>`, or multiple of `8` for vector memory address. Unaligned memory access will crash the interpreter.

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

store
- The `store` instruction truncates the value `<reg>` to an `<sz>`*8-bit integer and writes it at [`<reg_ptr>`, `<reg_ptr>+<sz>`).

vector load
- The `vload` instruction reads the data at [`<reg_ptr>` , `<reg_ptr>+32`) and writes it into destination `<vr>`

vector store
- The `vstore` instruction writes the value of `<vr>` at [`<reg_ptr>`, `<reg_ptr>+32`).

(4) Scalar Arithmetic

Kind	Name	Cost

Integer Multiplication/Division	`<reg> = udiv <reg1> <reg2> <bw>`
	`<reg> = sdiv <reg1> <reg2> <bw>`
	`<reg> = urem <reg1> <reg2> <bw>`
	`<reg> = srem <reg1> <reg2> <bw>`
	`<reg> = mul  <reg1> <reg2> <bw>`
	`<bw> := 1|8|16|32|64`
		1

Integer Shift
- shl: shift-left
- lshr: logical shift-right
- ashr: arithmetic shift-right
	`<reg> = shl  <reg1> <reg2> <bw>`
	`<reg> = lshr <reg1> <reg2> <bw>`
	`<reg> = ashr <reg1> <reg2> <bw>`
	`<bw> := 1|8|16|32|64`
		4

Bitwise Operation	`<reg> = and  <reg1> <reg2> <bw>`
	`<reg> = or   <reg1> <reg2> <bw>`
	`<reg> = xor  <reg1> <reg2> <bw>`
	`<bw> := 1|8|16|32|64`
		4

Integer Add/Sub	`<reg> = add <reg1> <reg2> <bw>`
	`<reg> = sub <reg1> <reg2> <bw>`
	`<bw> := 1|8|16|32|64`
		5

Integer increment `<reg>` = `<reg>` + 1	`<reg> = incr <reg1> <bw>` `<bw> := 1|8|16|32|64`	1
Integer decrement `<reg>` = `<reg>` - 1	`<reg> = decr <reg1> <bw>` `<bw> := 1|8|16|32|64`	1

Comparison
- `<cond>` is equivalent to the `cond` of LLVM IR's icmp
	`<reg> = icmp <`
	`cond> <reg1> <reg2> <bw>`
	`<bw> := 1|8|16|32|64`
		1

Ternary operation	`<reg> = select <reg_cond>` `<reg_true> <reg_false>`	1
Move constant	`<reg> = const <cst>`	1

- 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.
- For select, if <reg_cond> is not one of 0 or 1, the interpreter will crash.

(5) Elementwise Vector Arithmetic

Kind	Name	Cost

Integer Multiplication/Division	`<vr> = vudiv <vr1> <vr2> <bw>`
	`<vr> = vsdiv <vr1> <vr2> <bw>`
	`<vr> = vurem <vr1> <vr2> <bw>`
	`<vr> = vsrem <vr1> <vr2> <bw>`
	`<vr> = vmul  <vr1> <vr2> <bw>`
	`<bw> := 32|64`
		2

Integer Shift
- shl: shift-left
- lshr: logical shift-right
- ashr: arithmetic shift-right
	`<vr> = vshl  <vr1> <vr2> <bw>`
	`<vr> = vlshr <vr1> <vr2> <bw>`
	`<vr> = vashr <vr1> <vr2> <bw>`
	`<bw> := 32|64`
		8

Bitwise Operation	`<vr> = vand  <vr1> <vr2> <bw>`
	`<vr> = vor   <vr1> <vr2> <bw>`
	`<vr> = vxor  <vr1> <vr2> <bw>`
	`<bw> := 32|64`
		8

Integer Add/Sub	`<vr> = vadd <vr1> <vr2> <bw>`
	`<vr> = vsub <vr1> <vr2> <bw>`
	`<bw> := 32|64`
		10

Integer Increment	`<vr> = vincr <vr1> <bw>` `<bw> := 32|64`	2
integer Decrement	`<vr> = vdecr <vr1> <bw>` `<bw> := 32|64`	2
Comparison - `<cond>` is equivalent to the `cond` of LLVM IR's icmp	`<vr> = vicmp <cond> <vr1> <vr2> <bw>` `<bw> := 32|64`	2

Ternary operation	`<vr> = vselect <vr_cond>`
	`<vr_true> <vr_false> <bw>`
	`<bw> := 32|64`
		2

- All arithmetic operations described above are executed element-wise.
They execute scalar arithmetic operations between elements of the same index from two vector registers, or repeat the operation for every element if the operation is unary.
- For ternary operation, if each element in <vr_cond> is not one of 0 or 1, the interpreter will crash.

(6) Parallel Vector Arithmetic

Kind	Name	Cost

Integer Multiplication/Division	`<vr> = vpudiv <vr1> <vr2> <bw>`
	`<vr> = vpsdiv <vr1> <vr2> <bw>`
	`<vr> = vpurem <vr1> <vr2> <bw>`
	`<vr> = vpsrem <vr1> <vr2> <bw>`
	`<vr> = vpmul  <vr1> <vr2> <bw>`
	`<bw> := 32|64`
		2

Bitwise Operation	`<vr> = vpand  <vr1> <vr2> <bw>`
	`<vr> = vpor   <vr1> <vr2> <bw>`
	`<vr> = vpxor  <vr1> <vr2> <bw>`
	`<bw> := 32|64`
		8

Integer Add/Sub	`<vr> = vpadd <vr1> <vr2> <bw>`
	`<vr> = vpsub <vr1> <vr2> <bw>`
	`<bw> := 32|64`
		10

Comparison - `<cond>` is equivalent to the `cond` of LLVM IR's icmp	`<vr> = vpicmp <cond> <vr1> <vr2> <bw>` `<bw> := 32|64`	2
Ternary operation	`<vr> = vpselect <vr_cond> <vr1> <vr2> <bw>` `<bw> := 32|64`	2

- All arithmetic operations described above are executed in a 'parallel' manner.
They execute scalar arithmetic operations between each even-index element and adjacent odd-index element on the right.
For example, `v3 = vpadd v1 v2 64` executes the following:

```
v3[0] <- v1[0] + v1[1]
v3[1] <- v1[2] + v1[3]
v3[2] <- v2[0] + v2[1]
v3[3] <- v2[2] + v2[3]
```

- For ternary operation, if each element in <vr_cond> is not one of 0 or 1, the interpreter will crash.

(7) Vector Manipulation

Kind	Name	Cost
Broadcast	`<vr> = vbcast <reg> <bw>` `<bw> := 32|64`	4
Extract	`<reg> = vextct <vr1> <reg_idx> <bw>` `<bw> := 32|64`	4
Update	`<vr> = vupdate <vr1> <reg> <reg_idx> <bw>` `<bw> := 32|64`	4

- For extract and update, `<reg_idx>` must be less than 8 if `<bw>` is 32, and less than 4 if `<bw>` is 64. Otherwise the interpreter will crash.
- Broadcast writes `<reg>` to every element in `<vr>`, with each element size assumed as `<bw>`.
- Extract reads the `<reg_idx>`th element of `<vr1>` and write the result at `<reg>`
- Update changes the `<reg_idx>`th element of `<vr1>` to `<reg>` and write the result at `<vr>`