GFSsembly

GFSsembly is a toy assembly language for a register-based virtual machine, made for use in CS 3.

The VM has:

- A sequence of instructions. The VM starts with the first instruction, and moves through the instructions in sequence unless a jump operation is performed.
- Four registers, which each hold a single integer, which are initialized to `0` at the start of the program.
- A fixed-length memory array (a list of numbers).
- An output stream, which can be used to log numbers or print characters.

All numbers in the VM are integers (division results are truncated).

Instructions List

Instruction	Action
`add rX n`	Add `n` to `rX` and store the result in `rX`
`sub rX n`	Subtract `n` from `rX` and store the result in `rX`
`mul rX n`	Multiply `rX` by `n` and store the result in `rX`
`div rX n`	Divide `rX` by `n` (truncating decimals) and store the result in `rX`
`mod rX n`	Compute `rX` modulo `n` and store the result in `rX`
`eq rX n`	Check if `rX` equals `n` and store the result in `rX`
`lt rX n`	Check if `rX` is less than `n` and store the result in `rX`
`gt rX n`	Check if `rX` is greater than `n` and store the result in `rX`
`and rX n`	Compute logical 'and' of `rX` and `n` and store the result in `rX`
`or rX n`	Compute logical 'or' of `rX` and `n` and store the result in `rX`
`not rX`	Compute logical 'not' of `rX` and store the result in `rX`
`jmp n`	Jump by `n` instructions
`jeq n a b`	Jump by `n` instructions if `a` equals `b`
`jne n a b`	Jump by `n` instructions if `a` does not equal `b`
`halt`	Stop program execution immediately
`set rX n`	Set the value of `rX` to `n`
`log n`	Print `n` as a number on a new line
`print n`	Print `n` as a character in the current line, converting to unicode
`load rX index`	Load the number at memory location `index` into `rX`
`store src index`	Store `src` in memory at location `index`
`mem values...`	Set memory array to a given sequence of values

Notes:

- Instructions with a `rX` parameter store their result in the register `rX`, where `rX` is `r0`, `r1`, `r2`, or `r3`. All other arguments can be register names or number literals.
- Comparison instructions produce a result of `0` if the comparison is false and `1` if the comparison is true.
- Logical instructions interpret arguments of `0` as false and arguments of any other value as true.
- Jumps are applied relative to the instruction position. Conditional jumps whose condition is false proceed to the next instruction.

`add`

Instruction: `add rX n`

Action: Add `n` to `rX` and store the result in `rX`

Example: `add r2 r1`

	r0	r1	r2	r3	memory	output
before	`0`	`3`	`4`	`0`	`[]`	
after	`0`	`3`	`7`	`0`	`[]`	

Example: `add r2 8`

	r0	r1	r2	r3	memory	output
before	`0`	`0`	`4`	`0`	`[]`	
after	`0`	`0`	`12`	`0`	`[]`	

`sub`

Instruction: `sub rX n`

Action: Subtract `n` from `rX` and store the result in `rX`

Example: `sub r2 r1`

	r0	r1	r2	r3	memory	output
before	`0`	`7`	`5`	`0`	`[]`	
after	`0`	`7`	`-2`	`0`	`[]`	

Example: `sub r2 3`

	r0	r1	r2	r3	memory	output
before	`0`	`0`	`5`	`0`	`[]`	
after	`0`	`0`	`2`	`0`	`[]`	

`mul`

Instruction: `mul rX n`

Action: Multiply `rX` by `n` and store the result in `rX`

Example: `mul r2 r1`

	r0	r1	r2	r3	memory	output
before	`0`	`3`	`4`	`0`	`[]`	
after	`0`	`3`	`12`	`0`	`[]`	

Example: `mul r2 7`

	r0	r1	r2	r3	memory	output
before	`0`	`0`	`4`	`0`	`[]`	
after	`0`	`0`	`28`	`0`	`[]`	

`div`

Instruction: `div rX n`

Action: Divide `rX` by `n` (truncating decimals) and store the result in `rX`

Example: `div r2 r1`

	r0	r1	r2	r3	memory	output
before	`0`	`8`	`20`	`0`	`[]`	
after	`0`	`8`	`2`	`0`	`[]`	

Example: `div r2 4`

	r0	r1	r2	r3	memory	output
before	`0`	`0`	`20`	`0`	`[]`	
after	`0`	`0`	`5`	`0`	`[]`	

`mod`

Instruction: `mod rX n`

Action: Compute `rX` modulo `n` and store the result in `rX`

Example: `mod r2 r1`

	r0	r1	r2	r3	memory	output
before	`0`	`7`	`20`	`0`	`[]`	
after	`0`	`7`	`6`	`0`	`[]`	

Example: `mod r2 3`

	r0	r1	r2	r3	memory	output
before	`0`	`0`	`20`	`0`	`[]`	
after	`0`	`0`	`2`	`0`	`[]`	

`eq`

Instruction: `eq rX n`

Action: Check if `rX` equals `n` and store the result in `rX`

Example: `eq r2 r1`

	r0	r1	r2	r3	memory	output
before	`0`	`7`	`10`	`0`	`[]`	
after	`0`	`7`	`0`	`0`	`[]`	

Example: `eq r2 5`

	r0	r1	r2	r3	memory	output
before	`0`	`0`	`5`	`0`	`[]`	
after	`0`	`0`	`1`	`0`	`[]`	

`lt`

Instruction: `lt rX n`

Action: Check if `rX` is less than `n` and store the result in `rX`

Example: `lt r2 r1`

	r0	r1	r2	r3	memory	output
before	`0`	`17`	`10`	`0`	`[]`	
after	`0`	`17`	`1`	`0`	`[]`	

Example: `lt r2 5`

	r0	r1	r2	r3	memory	output
before	`0`	`0`	`10`	`0`	`[]`	
after	`0`	`0`	`0`	`0`	`[]`	

`gt`

Instruction: `gt rX n`

Action: Check if `rX` is greater than `n` and store the result in `rX`

Example: `gt r2 r1`

	r0	r1	r2	r3	memory	output
before	`0`	`17`	`10`	`0`	`[]`	
after	`0`	`17`	`0`	`0`	`[]`	

Example: `gt r2 5`

	r0	r1	r2	r3	memory	output
before	`0`	`0`	`10`	`0`	`[]`	
after	`0`	`0`	`1`	`0`	`[]`	

`and`

Instruction: `and rX n`

Action: Compute logical 'and' of `rX` and `n` and store the result in `rX`

Example: `and r2 r1`

	r0	r1	r2	r3	memory	output
before	`0`	`1`	`0`	`0`	`[]`	
after	`0`	`1`	`0`	`0`	`[]`	

`or`

Instruction: `or rX n`

Action: Compute logical 'or' of `rX` and `n` and store the result in `rX`

Example: `or r2 r1`

	r0	r1	r2	r3	memory	output
before	`0`	`1`	`0`	`0`	`[]`	
after	`0`	`1`	`1`	`0`	`[]`	

`not`

Instruction: `not rX`

Action: Compute logical 'not' of `rX` and store the result in `rX`

Example: `not r2`

	r0	r1	r2	r3	memory	output
before	`0`	`0`	`1`	`0`	`[]`	
after	`0`	`0`	`0`	`0`	`[]`	

`jmp`

Instruction: `jmp n`

Action: Jump by `n` instructions

Example: `jmp -5`

> Jump backwards by `5` instructions

`jeq`

Instruction: `jeq n a b`

Action: Jump by `n` instructions if `a` equals `b`

Example: `jeq 10 r2 r1`

> Jump forward by `10` instructions if the value in `r2` equals the value in `r1`

Example: `jeq 10 r2 0`

> Jump forward by `10` instructions if the value in `r2` equals `0`

`jne`

Instruction: `jne n a b`

Action: Jump by `n` instructions if `a` does not equal `b`

Example: `jne 10 r2 r1`

> Jump forward by `10` instructions if the value in `r2` does not equal the value in `r1`

Example: `jne 10 r2 0`

> Jump forward by `10` instructions if the value in `r2` does not equal `0`

`halt`

Instruction: `halt`

Action: Stop program execution immediately

`set`

Instruction: `set rX n`

Action: Set the value of `rX` to `n`

Example: `set r2 5`

	r0	r1	r2	r3	memory	output
before	`0`	`0`	`0`	`0`	`[]`	
after	`0`	`0`	`5`	`0`	`[]`	

Example: `set r2 r3`

	r0	r1	r2	r3	memory	output
before	`0`	`0`	`0`	`7`	`[]`	
after	`0`	`0`	`7`	`7`	`[]`	

`log`

Instruction: `log n`

Action: Print `n` as a number on a new line

Example: `log r2`

	r0	r1	r2	r3	memory	output
before	`0`	`0`	`7`	`0`	`[]`	
after	`0`	`0`	`7`	`0`	`[]`	7

Example: `log 10`

	r0	r1	r2	r3	memory	output
before	`0`	`0`	`0`	`0`	`[]`	
after	`0`	`0`	`0`	`0`	`[]`	10

`print`

Instruction: `print n`

Action: Print `n` as a character in the current line, converting to unicode

Example: `print r2`

	r0	r1	r2	r3	memory	output
before	`0`	`0`	`97`	`0`	`[]`	
after	`0`	`0`	`97`	`0`	`[]`	a

Example: `print 64`

	r0	r1	r2	r3	memory	output
before	`0`	`0`	`0`	`0`	`[]`	
after	`0`	`0`	`0`	`0`	`[]`	@

`load`

Instruction: `load rX index`

Action: Load the number at memory location `index` into `rX`

Example: `load r2 1`

	r0	r1	r2	r3	memory	output
before	`0`	`0`	`0`	`0`	`[4, 3, 0]`	
after	`0`	`0`	`3`	`0`	`[4, 3, 0]`	

Example: `load r2 r1`

	r0	r1	r2	r3	memory	output
before	`0`	`0`	`0`	`0`	`[4, 3, 0]`	
after	`0`	`0`	`4`	`0`	`[4, 3, 0]`	

`store`

Instruction: `store src index`

Action: Store `src` in memory at location `index`

Example: `store 5 r0`

	r0	r1	r2	r3	memory	output
before	`1`	`0`	`0`	`0`	`[0, 0, 0]`	
after	`1`	`0`	`0`	`0`	`[0, 5, 0]`	

Example: `store r1 2`

	r0	r1	r2	r3	memory	output
before	`0`	`8`	`0`	`0`	`[0, 0, 0]`	
after	`0`	`8`	`0`	`0`	`[0, 0, 8]`	

`mem`

Instruction: `mem values...`

Action: Set memory array to a given sequence of values

Example: `mem 1 1 2 3 5`

	r0	r1	r2	r3	memory	output
before	`0`	`0`	`0`	`0`	`[]`	
after	`0`	`0`	`0`	`0`	`[1, 1, 2, 3, 5]`	