

GFSEMBLY

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

	r0	r1	r2	r3	memory	output
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]	