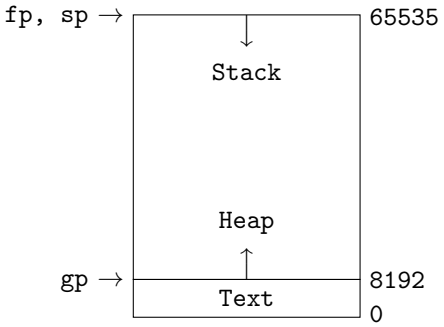
Marvin is a hypothetical computer with sixteen 32-bit registers and 65,536 32-bit words of main memory (aka RAM). In addition to the sixteen registers, Marvin has a 16-bit program counter `pc` and a 32-bit instruction register `ir`.

Registers: Marvin specifies the following conventions for its sixteen registers:

- `r0` – `r2` (`a0` – `a2`) are argument registers.
- `r3` – `r10` are general purpose registers.
- `r11` (`ra`) is reserved to store the return address of the calling subroutine (aka function).
- `r12` (`rv`) is reserved to store the return value of a subroutine.
- `r13` (`fp`), called the frame pointer is reserved to store the base address of the most recent frame on the stack. It is initialized to 65,535 when the computer boots.
- `r14` (`sp`), called the stack pointer, is reserved to store the address of the top of the stack. It is initialized to 65,535 when the computer boots.
- `r15` (`gp`), called the global pointer, is reserved for to store the address of the top of the heap. It is initialized to 8,192 when the computer boots.

Main Memory: Marvin's main memory is divided into a text segment, a stack segment, and a heap segment, as shown below:

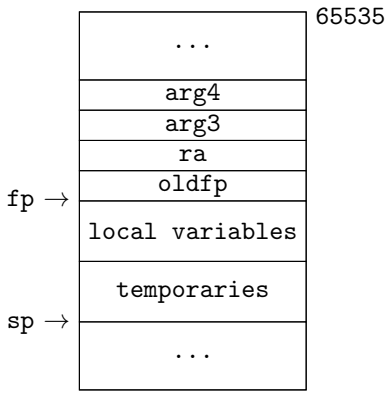


The text segment extends from address 0 to address 8192. A Marvin program, which is a `.marv` file, is assembled and loaded into this segment starting at address 0.

The dynamic stack segment starts at address 65535 and grows downwards. This is where a subroutine's arguments, its local variables, and temporaries are stored.

The dynamic heap segment starts at address 8192 and grows upwards. This is where dynamically created objects are stored.

When a subroutine is called, a stack frame must be created for it on the stack, and it must be organized as shown below:



Instructions: Marvin supports 53 instructions, each of which accepts between 0 and 3 arguments (aka inputs). Each category of instruction begins with a designated nibble (first four bits) followed by an operation specifier nibble.

System instructions (starting with the nibble `0000`):

Opcode	32-bit Machine Code	Description
`halt`	`00000000 00000000 00000000 00000000`	stops the machine
`readi rX`	`00000001 00000000 00000000 0000XXXX`	sets `rX = N`, where `N` $\in [-2^{31}, 2^{31}-1]$ read from stdin
`readf rX`	`00000011 00000000 00000000 0000XXXX`	sets `rX = N`, where `N` is a float read from stdin
`readc rX`	`00000101 00000000 00000000 0000XXXX`	sets `rX = N`, where `N` is a character read from stdin
`writei rX`	`00000010 00000000 00000000 0000XXXX`	writes integer `rX` to stdout
`writef rX`	`00000100 00000000 00000000 0000XXXX`	writes float `rX` to stdout
`writec rX`	`00000110 00000000 00000000 0000XXXX`	writes character `rX` to stdout
`seed rX`	`00000111 00000000 00000000 0000XXXX`	seeds the random number generator with `rX`
`rand rX rY rZ`	`00001000 00000000 0000XXXX YYYYZZZZ`	sets `rX` to a random number in the range `[rY, rZ]`
`time rX`	`00001001 00000000 00000000 0000XXXX`	sets `rX` to the time midnight
`date rX`	`00001010 00000000 00000000 0000XXXX`	sets `rX` to the current date
`nop`	`00001111 00000000 00000000 00000000`	does nothing

Arithmetic instructions (starting with the nibble `0001`):

Opcode	32-bit Machine Code	Description
`add rX rY rZ`	`00010000 00000000 0000XXXX YYYYZZZZ`	sets `rX = rY + rZ` for integers
`sub rX rY rZ`	`00010001 00000000 0000XXXX YYYYZZZZ`	sets `rX = rY - rZ` for integers
`mul rX rY rZ`	`00010010 00000000 0000XXXX YYYYZZZZ`	sets `rX = rY * rZ` for integers
`div rX rY rZ`	`00010011 00000000 0000XXXX YYYYZZZZ`	sets `rX = rY // rZ` for integers
`mod rX rY rZ`	`00010100 00000000 0000XXXX YYYYZZZZ`	sets `rX = rY % rZ` for integers
`neg rX rY`	`00010101 00000000 00000000 XXXXYYYY`	sets `rX = -rY` for integers
`fadd rX rY rZ`	`00010110 00000000 0000XXXX YYYYZZZZ`	sets `rX = rY + rZ` for floating point numbers
`fsub rX rY rZ`	`00010111 00000000 0000XXXX YYYYZZZZ`	sets `rX = rY - rZ` for floating point numbers
`fmul rX rY rZ`	`00011000 00000000 0000XXXX YYYYZZZZ`	sets `rX = rY * rZ` for floating point numbers
`fdiv rX rY rZ`	`00011001 00000000 0000XXXX YYYYZZZZ`	sets `rX = rY // rZ` for floating point numbers
`fneg rX rY`	`00011010 00000000 00000000 XXXXYYYY`	sets `rX = -rY` for floating point numbers

Bitwise instructions (starting with the nibble `0010`):

Opcode	32-bit Machine Code	Description
`and rX rY rZ`	`00100000 00000000 0000XXXX YYYYZZZZ`	bitwise ANDs `rY` by `rZ`, placing the result in `rX`
`or rX rY rZ`	`00100001 00000000 0000XXXX YYYYZZZZ`	bitwise ORs `rY` by `rZ`, placing the result in `rX`
`xor rX rY rZ`	`00100010 00000000 0000XXXX YYYYZZZZ`	bitwise XORs `rY` by `rZ`, placing the result in `rX`
`not rX rY rZ`	`00100011 00000000 0000XXXX YYYYZZZZ`	bitwise NOTs `rY` by `rZ`, placing the result in `rX`
`lshl rX rY`	`00100100 00000000 00000000 XXXXYYYY`	logical shifts `rY` `rZ` bits to the left
`lshr rX rY`	`00100101 00000000 00000000 XXXXYYYY`	logical shifts `rX` `rY` bits to the right
`ashl rX rY`	`00100110 00000000 00000000 XXXXYYYY`	arithmetic shifts integer `rX` `rY` bits to the left
`ashr rX rY`	`00100111 00000000 00000000 XXXXYYYY`	arithmetic shifts integer `rX` `rY` bits to the right

Jump instructions (starting with the nibble `0011`):

Opcode	32-bit Machine Code	Description
`jumpn N`	`00110000 00000000 NNNNNNNN NNNNNNNN`	jumps to instruction `N`
`jumpr rX`	`00110001 00000000 00000000 0000XXXX`	jumps to instruction `rX`
`jeqzn rX N`	`00110010 0000XXXX NNNNNNNN NNNNNNNN`	jumps to instruction `N` if `rX == 0`
`jnezn rX N`	`00110011 0000XXXX NNNNNNNN NNNNNNNN`	jumps to instruction `N` if `rX != 0`
`jgen rX rY N`	`00110100 XXXXYYYY NNNNNNNN NNNNNNNN`	jumps to instruction `N` if `rX >= rY`
`jlen rX rY N`	`00110101 XXXXYYYY NNNNNNNN NNNNNNNN`	jumps to instruction `N` if `rX <= rY`
`jeqn rX rY N`	`00110110 XXXXYYYY NNNNNNNN NNNNNNNN`	jumps to instruction `N` if `rX == rY`
`jnen rX rY N`	`00110111 XXXXYYYY NNNNNNNN NNNNNNNN`	jumps to instruction `N` if `rX != rY`
`jgtn rX rY N`	`00111000 XXXXYYYY NNNNNNNN NNNNNNNN`	jumps to instruction `N` if `rX > rY`
`jltn rX rY N`	`00111001 XXXXYYYY NNNNNNNN NNNNNNNN`	jumps to instruction `N` if `rX < rY`
`calln rX N`	`00111010 0000XXXX NNNNNNNN NNNNNNNN`	sets `rX = pc + 1` and jumps to instruction `N`

Instructions for setting register data (starting with the nibble `0100`):

Opcode	32-bit Machine Code	Description
`seti rX N`	`01000000 0000XXXX NNNNNNNN NNNNNNNN`	sets `rX = N`, where `N` $\in [-2^{15}, 2^{15}-1]$
`addi rX N`	`01000001 0000XXXX NNNNNNNN NNNNNNNN`	sets `rX = rX + N`, where `N` $\in [-2^{15}, 2^{15}-1]$
`setf rX F`	`01000010 0000XXXX FFFFFFFF FFFFFFFF`	sets `rX = F`, where `F` is a floating point number
`addf rX F`	`01000011 0000XXXX FFFFFFFF FFFFFFFF`	sets `rX = rX + F`, where `F` is a floating point number
`copy rX rY`	`01000100 00000000 00000000 XXXXYYYY`	sets `rX = rY`

Instructions for interacting with the stack (starting with the nibble `0101`):

Opcode	32-bit Machine Code	Description
`pushrb rX rY`	`01010000 00000000 00000000 XXXXYYYY`	sets `mem[rY--] = rX`
`poprb rX rY`	`01010001 00000000 00000000 XXXXYYYY`	sets `rX = mem[++rY]`
`pushrs rX rY`	`01010010 00000000 00000000 XXXXYYYY`	sets `mem[rY--] = rX`
`poprs rX rY`	`01010011 00000000 00000000 XXXXYYYY`	sets `rX = mem[++rY]`
`pushrw rX rY`	`01010100 00000000 00000000 XXXXYYYY`	sets `mem[rY--] = rX`
`poprw rX rY`	`01010101 00000000 00000000 XXXXYYYY`	sets `rX = mem[++rY]`

Instructions for interacting with memory (starting with the nibble `0110`):

Opcode	32-bit Machine Code	Description
`loadnb rX rY N`	`01100000 XXXXYYYY NNNNNNNN NNNNNNNN`	sets `rX = mem[rY + N]`, where `N` $\in [-2^{15}, 2^{15}-1]$
`storenb rX rY N`	`01100001 XXXXYYYY NNNNNNNN NNNNNNNN`	sets `mem[rY + N] = rX`, where `N` $\in [-2^{15}, 2^{15}-1]$
`loadrb rX rY`	`01100010 00000000 00000000 XXXXYYYY`	sets `rX = mem[rY]`
`storerb rX rY`	`01100011 00000000 00000000 XXXXYYYY`	sets `mem[rY] = rX`
`loadns rX rY N`	`01100100 XXXXYYYY NNNNNNNN NNNNNNNN`	sets `rX = mem[rY + N]`, where `N` $\in [-2^{15}, 2^{15}-1]$
`storens rX rY N`	`01100101 XXXXYYYY NNNNNNNN NNNNNNNN`	sets `mem[rY + N] = rX`, where `N` $\in [-2^{15}, 2^{15}-1]$
`loadrs rX rY`	`01100110 00000000 00000000 XXXXYYYY`	sets `rX = mem[rY]`
`storers rX rY`	`01100111 00000000 00000000 XXXXYYYY`	sets `mem[rY] = rX`
`loadnw rX rY N`	`01101000 XXXXYYYY NNNNNNNN NNNNNNNN`	sets `rX = mem[rY + N]`, where `N` $\in [-2^{15}, 2^{15}-1]$
`storenw rX rY N`	`01101001 XXXXYYYY NNNNNNNN NNNNNNNN`	sets `mem[rY + N] = rX`, where `N` $\in [-2^{15}, 2^{15}-1]$
`loadrw rX rY`	`01101010 00000000 00000000 XXXXYYYY`	sets `rX = mem[rY]`
`storerw rX rY`	`01101011 00000000 00000000 XXXXYYYY`	sets `mem[rY] = rX`

Marvin Emulator: The Python program `marvin.py` serves as an emulator for the Marvin machine. Here is the usage syntax for the program:

```
× ~/workspace/marvin
$ python3 marvin.py -h

usage: marvin [-h] [-v] [-d] filename

This program serves as an emulator for a register-based machine called Marvin (named
after the paranoid android character, Marvin, from The Hitchhiker's Guide to the
Galaxy by Douglas Adams). The design of the machine was inspired by that of the
Harvey Mudd Miniature Machine (HMMM) developed at Harvey Mudd College. The program
accepts a .marv file as input, assembles and simulates the instructions within, and
prints any output to stdout. Any input to the .marv program is via stdin. If the
optional -v argument is specified, the emulator prints the assembled instructions
to stdout before simulating them.

positional arguments:
  filename       input .marv file

options:
  -h, --help     show this help message and exit
```

```
  -v, --verbose  enable verbose output
  -d, --debug    enable debug mode
$ _
```

Here is a sample Marvin program called `Coundown.marv` that accepts n (int) from standard input and writes to standard output a countdown from n to 0.

× Countdown.marv

```
# Accepts n (int) from standard input and writes a countdown from n to 0 to standard
# output.

0    read     r0          # read n
1    set0     r1          # zero = 0
2    jltn     r0 r1 6     # if n < zero jump to 6
3    write    r0          # write n
4    addn     r0 -1       # n = n - 1
5    jumpn    2           # jump to 2
6    halt                 # halt the machine
```

Here is the output from running `Countdown.marv` using `marvin.py`, which is an emulator for the Marvin machine.

× ~/workspace/marvin

```
$ python3 marvin.py -v Countdown.marv
    0: 00000001 00000000 00000000 00000000                    0: read   r0
    1: 00000100 00000000 00000000 00000001                    1: set0   r1
    2: 00011000 00000001 00000000 00000110                    2: jltn   r0 r1 6
    3: 00000010 00000000 00000000 00000000                    3: write  r0
    4: 00000111 00000000 10000000 00000001                    4: addn   r0 -1
    5: 00001111 00000000 00000000 00000010                    5: jumpn  2
    6: 00000000 00000000 00000000 00000000                    6: halt

5
5
4
3
2
1
0
$ _
```