# Introduction

Our virtual machine consists of three main components. [Memory](#_cogoqu6gzpdc), also called Random Access Memory (RAM), since it enables reading and writing into any location. The [Central Processing Unit](#_58vuetgb9kon) that executes the program from memory, and the [Beeper](#_grhcpsn2snjr). Download the starter code at <https://www.cs.unh.edu/~cs416/public/2P/>

# Memory

This virtual machine has 16 addressable memory locations (address 0 through 15), each of which holds a value that fits into 4 bits.One way to think of memory is as an array of numbers, where the array index represents a memory address.

| 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0  ^ ^ ^  | | |  Position 0 | Position 15  Position 7 |
| --- |

## Memory.java

We will model memory usage using a class **Memory.java**. Use an int array as an instance variable to represent the memory Implement the following:

* **Memory()** - default, no argument constructor
  + Creates an array of integers to represent the VM memory and initializes each memory cell to 0
* **Memory( Scanner s)** - one argument constructor
  + Creates an array of integers to represent the VM memory and initializes each memory cell to a value in from the Scanner.
  + For example, if the Scanner is reading a program file with the following contents:

| 1 2 3 4 5 6 7 8  8 7 6 5 4 3 2 1 |
| --- |

Then the internal array is initialized as

| 1 2 3 4 5 6 7 8 8 7 6 5 4 3 2 1 |
| --- |

* **Int read(int address) throws IllegalStateException**
  + Returns the value stored at the specified address, or throws an IllegalStateException if the address is outside the valid range
  + This [link](https://math.hws.edu/javanotes/c8/s3.html) provides more explanation on try/catch blocks and throwing exceptions
* **void write(int address, int value) throws IllegalStateException**
  + Writes the value to the specified address, or throws an IllegalStateException if the address is is outside the valid range
  + This [link](https://math.hws.edu/javanotes/c8/s3.html) provides more explanation on try/catch blocks and throwing exceptions
* **String toString()**
  + Returns the String representation of Memory. Should produce a String in the following format. Two lines, the first line showing the addresses and the second line showing the values at those addresses
  + **NOTE:** This returns a **single** String consisting of both lines

| address: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |  value: | 3 | 6 | 1 | 11 | 6 | 8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | |
| --- |

* **Int[] rawMemory()**
  + Return the internal array for debugging and testing purposes.

# CPU

The CPU purpose is to load instructions from memory and execute them. This virtual machine consists of 4 registers, each of which can hold a 4 bit number from 0 to 15.. Registers are memory that the CPU can access directly.

Our CPU has the following registers:

* IP (Instruction Pointer)
* IS (Instruction store)
* R0 (General purpose register)
* R1 (General purpose register)

## Typical Scenario

The CPU fetches an instruction from the memory address pointed by IP (Instruction pointer) and puts it IS (instruction store) and executes that instruction. This then moves the IP to the next instruction or sets it based on the previously executed instruction.

## Instructions

| **Instruction** | **Operation** | **Details** |
| --- | --- | --- |
| 0 | Halt |  |
| 1 | R0 = R0 + R1 | Add R0 and R1 and store the result in R0 |
| 2 | R0 = R0 - R1 | Subtract R0 And R1 and store the result in R0 |
| 3 | R0 = R0 + 1 | Increment R0 |
| 4 | R1= R1 + 1 | Increment R1 |
| 5 | R0 = R0 - 1 | Decrement R0 |
| 6 | R1 = R1 - 1 | Decrement R1 |
| 7 | BEEP |  |
| 8 X | PRINT X | Print the next memory cell |
| 9 X | R0 = MEMORY[X] | Load the value at address X into R0 |
| 10 X | R1 = MEMORY[X] | Load the value at address X into R1 |
| 11 X | MEMORY[X] = R0 | Store the value of R0 into address X |
| 12 X | MEMORY[X] = R1 | Store the value of R1 into address X |
| 13 X | JUMP X | Jump to the value in the next memory cell. E.g. 13 7 means set the IP to 7 |
| 14 X | JUMP X IF R0 == 0 | Jump to X if R0 is equal to 0. E.g. 14 7 means jump to address 7 if R0 is equal to 0, otherwise proceed with the next instruction |
| 15 X | JUMP X if R0 != 0 | Jump to X if R0 is **not** equal to 0 |

## Notes

* Some instructions only involve one memory address(e.g. add, subtract, beep, etc), others take two memory addresses (print or jump)
* X in the table means the following memory address. So if you read instruction 11 at memory address 3, X would be the value stored in memory address 4. Increase the IP to read the value stored in memory address 4.
* CPU can’t go directly to memory to add/subtract. It needs to load values from memory to registers, do operation, and store back into memory
* **ALL values of registers/memory need to be an unsigned 4 bit value (i.e. 0 to 15)**
  + Values wrap around. For example, if R0 is the value 15 and we execute instruction 3 (increment), the result in R0 is now 0
  + Conversely, this works for subtraction too. If R1 is the value0 and we execute instruction 6 (decrement), the result in R1 is now 15.
  + There needs to be a way to *force* all values of operations, changes to registers to always be between 0 and 15 and exhibit this “wrap around behavior” (also called overflow)
    - One approach is lots of careful logic
    - The other is to use a **bitwise and** operation
      * If we *bitwise and* calculated values with 0xF (1111) then this only keeps the lower 4 bits, effectively guaranteeing a value between 0 and 15
      * Examples:
        + 3 & 0xF => 3
        + 16 & 0xF => 0
        + -2 & 0xF => 14
      * <https://en.wikipedia.org/wiki/Bitwise_operation>
      * <https://docs.oracle.com/javase/tutorial/java/nutsandbolts/op3.html>

## CPU.java

We will model CPU usage using a class, **CPU.java.** Implement the following methods

* **CPU()** - default, no argument constructor
  + Initializes the CPU object state. Sets the 4 registers (a.k.a, instance variables) to 0
* **void run(Memory mem, boolean debugMode) throws IllegalStateException**
  + While the program is Running (not halted). Execute the 3 step Fetch-Execute Cycle over and over again.

1. Read the value from memory using the value stored in the IP register and then store the read value from memory into the IS register
2. Execute the instruction stored in the IS register
3. Increase the count of the IP register by 1 or 2 or set directly
   * Debug mode enables additional information to be printed during the CPU execution, such as, the current cycle (step) of the program, the CPU registers, and Memory

| STEP 0:  IP= 0, IS= 9, R0= 0, R1= 0  address: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |  value: | 9 | 15 | 11 | 5 | 8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 6 |  ----------  STEP 1:  IP= 2, IS=11, R0= 6, R1= 0  address: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |  value: | 9 | 15 | 11 | 5 | 8 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 6 |  ----------  ... |
| --- |

* **String toString()**
  + Returns the String representation of the CPU. It should returns a String with the register names and current values.
  + For example,

| IP= 2, IS=11, R0= 6, R1= 0 |
| --- |

# Beeper

As part of the starter code, you have been provided an implementation of a Beep class which will play a tone for some amount of time. To have your program play a short beep, call the beep static function of the Beep class. Example:

| Beep.beep(); // play a tone  …  Beep.beep(); // play a second tone |
| --- |

# Starter Code and Submission

The starter code provides a fully working **Beep.java** to play simple tones, a fully working **VM.java** driver for loading and running programs from the command line, and skeleton code for **Memory.java** and **CPU.java.**  Submit **Memory.java** and **CPU.java** on Gradescope.

You are also provided ~50 small programs written in the VM programming language described above in the [Instructions](#_qm78q9hp0abw) section above, as well as, the expected debug output of each of those programs.

Feel free to write and test your own small programs too.

# 

# Example

Let's look at the execution of the VM once memory has been loaded with the following program

| 3 11 4 8 0 0 0 0 0 0 0 0 0 0 0 0 |
| --- |

Once the CPU starts running it:

* FETCH the value at address IP into IS
* EXECUTE the instruction from IS
* UPDATE the IP location accordingly
  + e.g. if the instruction is 13 7 (JUMP TO 7), it will just do IP = 7 or if it is just 3 (INCREMENT R0) it will do IP = IP + 1
* GOTO FETCH, forever and ever

So the program above does the following:

1. IP = 0, IS = 3
   1. increment the value of R0
   2. IP = IP + 1
2. IP = 1, IS = 11 4
   1. store the value of R0 on address 4
   2. IP = IP + 2
3. IP = 3, IS = 8 1
   1. PRINT 1, you see when we started on address 4 we had the value 0 but after the previous instruction, we put 1 there, so when we execute the print instruction it will print 1
4. IP = 5, IS = 0
   1. the value at address 5 is 0, so the CPU will HALT and not update the instruction pointer(IP) anymore

# Acknowledgements

This assignment is heavily influenced by the following project: <https://punkx.org/4917/guide.html>