

## Learning Outcomes

Upon completing this assignment, students should be able to

- Use their `BitString` and `Instruction` objects to implement simulated RAM and ROM.
- Use features of the `Instruction` to interpret instructions in the Nand2Tetris Hack CPU/Machine Language
- Use the `invert`, `add`, and `and` methods of the `BitString` to *implement* the Nand2Tetris CPU

## Introduction

The `BitString` and `Instruction` classes we have implemented recently represent the 16-bit *word* (memory cell) in the RAM of a Hack computer and the 16-bit *instruction word* in the ROM. Since the `BitString` can be inverted, anded and added, it will be easy to implement a Nand2Tetris CPU and then a computer simulator.

## Method

### Overview

The main Java program, `Hack`, will take a single file name on the command line. It is assumed that this file contains lines of sixteen 1 and 0 *characters* representing Hack instructions.

The program will instantiate a ROM, a big list of `Instruction`. The ROM (which you will write) will be loaded from the machine language file. The program will then instantiate a large RAM, a list of `BitString` that is initialized with all zero values. Finally, the program will create a CPU object with the loaded ROM and zeroed RAM.

Finally, `Hack` will run the following loop:

```
while (!cpu.halt()) {  
    cpu.fetch();  
    cpu.decode();  
    cpu.execute();  
}
```

**Notice** that the simulator, `Hack`, does not provide any addresses, instructions, or values to the CPU: that object has a program counter, the A and D registers along with a IDR. Those registers are used to read and write into the RAM/ROM to execute the Hack machine language program that was loaded.

When the program ends, the Hack simulator will print out the values of the RAM that were changed so that you can see how the program changed the “computer’s” memory.

loading it from the given file (you will write ROM to some interfaces). It will then instantiate a RAM

## The Packages

### bits

This is where your `BitString` and `Instruction` classes live along with a new class provided with this assignment.

You will need a couple of new methods in `BitString` and `Instruction`:

```
class BitString {  
    // take a string (perhaps from a file?) of 0/1 and initialize  
    public BitString(String stringOfZerosAndOnes) { ... }  
  
    // like toString but without the spaces  
    public String stringOfZerosAndOnes() { ... }
```

```

    // return int (always non-negative) interpretation of BitString
    public int toInt() { ... }
}

class Instruction {
    // just use the super constructor to copy the bits so they can
    // be interpreted as an Instruction
    Instruction(BitString bitstring) { ... }
}

```

InstructionToASM provides static functions to convert an Instruction into ASM and *vice versa*. This **could** be part of Instruction but I moved it out so that I could write it for you.

## hardware\_interface

This package is full of *interfaces* that your classes will implement.

**PrintableHardware** interface that returns arrays of BitString or short from a memory. Both data and instruction memories implement this interface.

**Note** the two rangeTo\* routines take the **last** index to return as their second parameter, not *one past the last* as with arrays and substrings. This is due to the limitations of short values. routines that return arrays take the *valid* indices of the two endpoints. The last spot is **not** given as one past. This is because of the limitations of short.

To get *all* the bit strings in a RAM(100) (a RAM with 100 words in it), use

```

BitString array[] =
    ram.rangeToBitString(new BitString((short)0), new BitString((short)99));

```

the last value is the hundredth index.

**LoadableHardware** interface that includes methods for loading the memory from a file or a scanner. Uses a new feature of Java, default function bodies. The **file** based methods are implemented for you in the interface: they forward the work to the scanner-based version. **You** will have to provide the scanner-based version for both your ram and rom.

**Memory** This is the interface your RAM must implement. It extends the two \*Hardware interfaces so you need to provide their functions, too. This has get and set, as you would expect for reading and writing to memory.

There are two other methods, capacity() and size(). capacity returns the total number of memory words in the Memory (the size it was constructed with, probably); size returns the number of elements, from the beginning of the memory, that have been read or written since the memory was constructed. So, if I get location 7, the size returns at least 8. This is used for printing the results of the program.

**InstructionMemory** Has size and capacity as with Memory and a get that returns an Instruction rather than a BitString. No set because instruction memory is read-only.

**N2TCPU** This is the interface for your CPU. It has a state method that returns a string with the values of the CPU's registers, a halt method that should return true when the CPU runs out of loaded program, and the standard fetch, decode, and execute methods.

fetch uses your CPU's program counter to get an instruction from the ROM. Before the next call to fetch, your CPU will need to update the PC for the next instruction (add one) or to take a jump.

decode can do anything to prepare for execution.

execute is a lot like the test code in the Instruction lab: get the x and y inputs for the **arithmetic logic unit** (one comes from the D register, the other from either A or RAM[A]). Then use the c bits in the instruction to invert/zero the inputs, add or and the two values, and invert the output.

The *dst* bits determine where the output is copied (RAM[A], D, A or any subset of this set) and then the zero and negative bits in out are checked according to the *jmp* bits in the instruction to see if A gets copied into PC (the jump is taken).

### **simulation**

Provides the driver program, *CPUClient*. When it compiles, it is run with a Hack file name (required) and an optional switch to get it to print out CPU state every n clock cycles.

### **Documentation**

#### **README**

Must document how you tested. How do you know that it is right?

Must include instructions on how to **compile** and how to **run** the program as submitted.

### **Deliverables**

**Submission medium:** git to Gitea at `cs-devel.potsdam.edu`.