

# Introduction to Computer Science

## HW #2

Due: 2018/04/11

---

### Homework Rules:

Hand-written homework can be handed in **before lecture starts**. Otherwise, you may contact the TA in advance and then bring the hardcopy to the TA in MD-631 (please send e-mail in advance).

As for the programming part, you need to upload it to CEIBA before the deadline (2018/04/11 3am). You can choose to program in C++ or in Python. The file you upload must be a **.zip** file that contains the following files:

**README.txt**

**HW01\_b05901XXX** (a folder that contains all .cpp & .h (or .py) as required),

If you program in C/C++:

1. Do not submit executable files (.exe) or objective files (.o, .obj). Files with names in wrong format will not be graded. You must **remove any system calls**, such as system("pause"), in your code if any.
2. In README.txt, you need to describe which compiler you used in this homework and how to compile or execute it (if it is in a "project" form).
3. In your .cpp files, we suggest you write comments as detailed as you can. If your code does not work properly, code with comments earns you more partial credits.

If you program in Python:

1. Do not submit .pyc files. Files with names in wrong format will not be graded.
2. In README.txt, you need to describe which Python version you used in this homework (e.g. Python 3.5) and how to execute it.
3. In your .py files, we suggest you write comments as detailed as you can. If your code does not work properly, code with comments earns you more partial credits.

# Introduction to Computer Science

## HW #2

Due: 2018/04/11

---

### Review Problems (50%, each 5%)

#### Chapter 2

Problems 1, 23, 24(a), 28, 35, 44.


#### Chapter 3

Problems 26, 35, 42, 47.

### Programming Problem (50%)

#### 1. Machine Instruction Simulator

We are going to write a simulator to simulate how the CPU works with machine instructions. The instructions are defined the Appendix C in the textbook.



### A Simple Machine Language

In this appendix we present a simple but representative machine language. We begin by explaining the architecture of the machine itself.

#### The Machine's Architecture

The machine has 16 general-purpose registers numbered 0 through F (in hexadecimal). Each register is one byte (eight bits) long. For identifying registers within instructions, each register is assigned the unique four-bit pattern that represents its register number. Thus register 0 is identified by 0000 (hexadecimal 0), and register 4 is identified by 0100 (hexadecimal 4).

There are 256 cells in the machine's main memory. Each cell is assigned a unique address consisting of an integer in the range of 0 to 255. An address can therefore be represented by a pattern of eight bits ranging from 00000000 to 11111111 (or a hexadecimal value in the range of 00 to FF).

Floating-point values are assumed to be stored in an eight-bit format discussed in Section 1.7 and summarized in Figure 1.24.

#### The Machine's Language

Each machine instruction is two bytes long. The first 4 bits provide the op-code; the last 12 bits make up the operand field. The table that follows lists the instructions in hexadecimal notation together with a short description of each. The letters R, S, and T are used in place of hexadecimal digits in those fields representing a register identifier that varies depending on the particular application of the instruction. The letters X and Y are used in lieu of hexadecimal digits in variable fields not representing a register.

Op-code	Operand	Description
1	RXY	LOAD the register R with the bit pattern found in the memory cell whose address is XY. <i>Example:</i> 14A3 would cause the contents of the memory cell located at address A3 to be placed in register 4.
2	RXY	LOAD the register R with the bit pattern XY. <i>Example:</i> 20A3 would cause the value A3 to be placed in register 0.

581

# Introduction to Computer Science

## HW #2

Due: 2018/04/11

582 Appendixes

Op-code	Operand	Description
3	RXY	STORE the bit pattern found in register R in the memory cell whose address is XY. <i>Example:</i> 35B1 would cause the contents of register 5 to be placed in the memory cell whose address is B1.
4	ORS	MOVE the bit pattern found in register R to register S. <i>Example:</i> 40A4 would cause the contents of register A to be copied into register 4.
5	RST	ADD the bit patterns in registers S and T as though they were two's complement representations and leave the result in register R. <i>Example:</i> 5726 would cause the binary values in registers 2 and 6 to be added and the sum placed in register 7.
6	RST	ADD the bit patterns in registers S and T as though they represented values in floating-point notation and leave the floating-point result in register R. <i>Example:</i> 634E would cause the values in registers 4 and E to be added as floating-point values and the result to be placed in register 3.
7	RST	OR the bit patterns in registers S and T and place the result in register R. <i>Example:</i> 7C84 would cause the result of ORing the contents of registers B and 4 to be placed in register C.
8	RST	AND the bit patterns in registers S and T and place the result in register R. <i>Example:</i> 8045 would cause the result of ANDing the contents of registers 4 and 5 to be placed in register 0.
9	RST	EXCLUSIVE OR the bit patterns in registers S and T and place the result in register R. <i>Example:</i> 95F3 would cause the result of EXCLUSIVE ORing the contents of registers F and 3 to be placed in register 5.
A	ROX	ROTATE the bit pattern in register R one bit to the right X times. Each time place the bit that started at the low-order end at the high-order end. <i>Example:</i> A403 would cause the contents of register 4 to be rotated 3 bits to the right in a circular fashion.
B	RXY	JUMP to the instruction located in the memory cell at address XY if the bit pattern in register R is equal to the bit pattern in register number 0. Otherwise, continue with the normal sequence of execution. (The jump is implemented by copying XY into the program counter during the execute phase.) <i>Example:</i> B43C would first compare the contents of register 4 with the contents of register 0. If the two were equal, the pattern 3C would be placed in the program counter so that the next instruction executed would be the one located at that memory address. Otherwise, nothing would be done and program execution would continue in its normal sequence.
C	000	HALT execution. <i>Example:</i> C000 would cause program execution to stop.

Here is an example.

<b>Program counter</b>	00
<b>Register 0</b>	00
<b>Register 1</b>	00
<b>Register 2</b>	00
<b>Register 3</b>	00
<b>Register 4</b>	00
<b>Address</b>	<b>Cells</b>
00	14
01	02
02	34
03	00
04	C0
05	00

After the simulation, the result should be as follows.

# Introduction to Computer Science

## HW #2

Due: 2018/04/11

---

<b>Program counter</b>	06
<b>Register 0</b>	00
<b>Register 1</b>	00
<b>Register 2</b>	00
<b>Register 3</b>	00
<b>Register 4</b>	34
<b>Address</b>	<b>Cells</b>
00	34
01	02
02	34
03	00
04	C0
05	00

### Input/ Output format:

The input file is a 256-bytes binary file containing data stored in memory address 00 to FF. The simulation should start with program counter starts at 00, with 16 registers, denoted 0 to F respectively, containing 00 in them. The output file should be stored in the same format containing the data in memory address 00 to FF.

### Important rules:

- A class named Simulator, TA will use it for grading.
- You need to implement the following member functions as interfaces:

In C++:

```
bool/void loadMemory (string/char* memoryPath); //Loading input file
bool/void simulate (); //Simulate the machine instructions
bool/void storeMemory(string/char* outputPath); //Store output file
```

In Python:

```
def loadMemory (self, memoryPath):
    #Loading input file
def simulate (self):
    #Simulate the machine instructions
def storeMemory (outputPath):
    #Store output file
```

# Introduction to Computer Science

## HW #2

Due: 2018/04/11

---

- TA will test your code in a way like this:

C++:

```
#include "Simulator.h"
int main(){
    Simulator sim;
    sim.loadMemory("input");
    sim.storeMemory("result1");
    sim.simulate();
    sim.storeMemory("result2");
    return 0;
}
```

Python:

```
import Simulator;
def main():
    sim = Simulator.Simulator();
    img.loadMemory("input");
    img.storeMemory("result1");
    img.simulate();
    img.storeMemory("result2");
```

## 2. Bonus (5%): Assembly Simulator

This time, we write another simulator to simulate assembly code with the same instruction set as the above one. The mapping is shown below:

Machine code	Assembly code
1RXY	lw R XY
2RXY	lb R XY
3RXY	sw R XY
4ORS	mv R S
5RST	add R S T
6RST	addf R S T
7RST	or R S T
8RST	and R S T
9RST	xor R S T
AROX	srl R R X

# Introduction to Computer Science

## HW #2

Due: 2018/04/11

---

BRXY	beq 0 R XY
C000	halt

The input will be a text file with each line meaning an instruction. When simulating, start from the first line to the last one. Also need to be mentioned, the input instruction may be in higher case or lower case and you have to handle the memory management for variable address like the instruction "sw A X". Variables will only represent addresses and they will be any string except 00 to FF. When 00 to FF is in memory address slot, it means the absolute address.

You should modify the Simulator class to enable it to do the job by adding member functions, the input file will be given in file path just like loadMemory(). The output function should be storeMemory().

**If you meet the bonus requirements, write "I finished the bonus part." with further details (at least how to do the job ) in the readme file to let TA know.**