CSCE-312 | Project 5

BITBOT Computer

Project Submission: 100 points

Grading

Project Submission [100%]:

You will be graded for correct functioning of the computer. TA's will run tests on the submitted HDL files downloaded from your Canvas submission using Nand2tetris software (Hardware Simulator). Please make sure to test and verify your codes before finally submitting on Canvas.

Deliverables & Submission

You are required to turn in **the completed HDL files for the following three chips**: **Computer**, **CPU and Memory.** Put your **full name** in the introductory comment present in each file. You do not need to turn in any .tst or .cmp files. They are already provided for your use in testing the computer components. Use relevant code comments and indentation. Zip all the required files into a compressed file ***FirstName-LastName-UIN_P5.zip***. **Submit this zip file on Canvas.**

Background

In the previous projects we've built a computer's basic *processing* and *storage* devices (*ALU* and *RAM*, respectively). In the labs, we also gained firsthand exposure to building the TOY computer in the labs and HACK Computer in the lectures with their underlying components. In this project we will put our knowledge and skills to work in constructing our very own BITBOT computer by putting everything together, yielding the complete BITBOT *Hardware Platform*. The result will be a general-purpose computer that can run many general-purpose programs that you fancy.

Below we will describe the overall design objective, the chips needing to be implemented, and recommended testing. Then we will dive into the BITBOT computer specifications. Please read the specs and examine the schematics very carefully before creating your implementation plan.

As demonstrated in lecture and lab using the HACK and TOY computers as a teaching vehicle, we recommend using a similar approach to build out the complete set of schematics on paper before diving into the respective HDL implementations.

Objective

Complete the construction of the BITBOT CPU and computer platform memory component. These components will integrate in the top-most BITBOT Computer chip. The CPU and Data Memory are already instantiated (connected) in the Computer chip but you will be required to instantiate Instruction ROM to complete the Computer chip design. Once you have *designed* your CPU and Data Memory, and instantiated the Instruction ROM inside the Computer, you will be able to exercise the Computer chip.

Chips

Chip (HDL)	Description	Testing
Memory.hdl	Entire **Data Memory** address space	Test this chip using Memory.tst and Memory.cmp
CPU.hdl	The BITBOT **CPU**	Test this chip using CPU.tst and CPU.cmp.
Computer.hdl	The BITBOT **Computer**	Test this chip using the BITBOT machine language programs noted below.

Contract

The computer platform that you build should be capable of executing programs written in the BITBOT *machine (assembly) language*, specified below.

Testing

Testing the Data Memory and CPU chips: It's important to unit-test these chips before proceeding to build the overall BITBOT Computer chip. **Starter test collateral is provided. *You are advised to extend them to perform robust testing.***

Testing the Computer chip: A natural way to test the Computer chip implementation is to have it execute some sample programs written in the BITBOT machine (assembly) language. In order to perform such a test, one can write a test script that (i) loads the Computer.hdl chip description into the supplied *Hardware Simulator*, (ii) loads a machine-level program from an external file containing the program instructions into the ROM chip-part of the loaded Computer.hdl chip, and then (iii) runs the clock for enough number of cycles to execute the loaded instructions. Three such programs are provided:

Program	Comments
Add.hack	Adds two numbers at RAM[0] and RAM[1] and writes the result in RAM[2].
Mult.hack	Computes the multiplication of RAM[0] and RAM[1] and writes the result in RAM[2].
Rect.hack	Draws a rectangle of width 16 pixels and length RAM[1] at the top left of the screen.

The test collateral for these exercises is also included that you may use as is to test your Computer implementation. Unlike the CPU and Memory chips where we have only provided the starter collateral to be extended by you for comprehensive testing, the testing of the Computer chip will be done with the test collateral **as is** (i.e. you do not need to extend the Computer-level test collateral).

Finally, a nifty test collateral is provided for you to test your KEYBOARD USE (Keyboard.hack, Keyboard.tst, and Keyboard.cmp). You also do not need to modify this and may use as is.

BITBOT Computer Design Specifications

[A] BITBOT Computer Instructions

The BITBOT computer supports FIVE types of instructions as follows:

(1) **Arithmetic**: Two arithmetic operations are supported - **ADD** (addition) and **SUB** (subtraction). Both operations support *immediate* and *register* operands.

Format:

ADD Rx, Ry, Rz //Rx = Ry+Rz, where Rx, Ry, Rz are registers
SUB Rx, Ry, Rz // Rx = Ry-Rz, where Rx, Ry, Rz are registers
ADDI Rx, Ry, Constant //Rx = Ry+Constant, where Rx, Ry are registers and *Constant* is a 6-bit positive number between 0 and 63
SUBI Rx, Ry, Constant ////Rx = Ry-Constant, where Rx, Ry are registers and *Constant* is a 6-bit positive number between 0 and 63

Examples:

ADD R0, R1, R2 - Operation is *R0=R1+R2*
ADDI R0, R1, 7 - Operation is *R0 = R1+7*
SUB R0, R1, R2 - Operation is *R0=R1-R2*
SUBI R0, R1, 6 - Operation is *R0 = R1-6*

(2) **Logical**: Two logical bitwise operations are supported - **NAND** and **NOR**

Format:

NAND Rx, Ry, Rz //Rx = Ry NAND Rz, where Rx, Ry, Rz are registers
NOR Rx, Ry, Rz // Rx = Ry NOR Rz, where Rx, Ry, Rz are registers

Examples:

NAND R0, R1, R2 - Operation is *R0=R1 NAND R2* done in bitwise fashion
NOR R0, R1, R2 - Operation is *R0=R1 NOR R2* done in bitwise fashion

(3) Memory: Two memory operations are supported - **READ and WRITE:**

Format:

READ Rx, Ry //Rx = MEM[Ry], where Rx and Ry are registers
WRITE Rx, Ry //MEM[Rx] = Ry, where Rx and Ry are registers

Examples:

READ R0, R1 - Operation is R0 = MEM[R1]
WRITE R4, R5 - Operation is MEM[R4] = R5

(4) Branch: Two branch operations are supported - **Conditional and Unconditional**

Format:

BEQ Rx, Ry //Conditional: Fetch Next Instruction from ROM[Rx] if (Ry == 0)
JMP Rx //Unconditional: Fetch Next Instruction from ROM[Rx]

Examples:

BEQ R0, R5 //Fetch Next Instruction from ROM[R0] if (R5 == 0)
JMP R7 //Fetch Next Instruction from ROM[R7]

(5) Input/Output: One Input and One Output Instruction are supported - **INP and OUT**

Format:

OUT Rx, Ry *//MEM[Rx] = Ry*
//Assume: Rx contents are within SCREEN Memory range
INP *Rx // Rx = MEM[hardwired address for Keyboard]*
//Assume: Only one location for Keyboard Input will be tested

Examples:

OUT R0, R5 //MEM[R0] = R5, //R0 content is within SCREEN Memory range
INP R0 // R0 = contents of Memory address dedicated to Keyboard m

[B] BITBOT Computer Instruction Encodings

Our BITBOT computer uses a Register Indirect Mode of Addressing. Unlike the HACK Computer where only the A-Register could be used to address data memory, we have a larger number of choices here. There is also no distinction made as to which register may be used to hold data versus address.

The 16-bit Instruction In[15:0] is designed to be interpreted as follows (examples are also provided along with bit encodings):

a. **In[15:14] are OPCODE Bits**
   - In[15:14] = 00 for *Arithmetic* operations
   - In[15:14] = 01 for *Logical* operations
   - In[15:14] = 10 for *Memory or Branch* operations
   - In[15:14] = 11 for I/O operations (OUT to the SCREEN or INP from the KEYBOARD)

b. In[13] & In[12] are OPTYPE Bits

- *Arithmetic*: In[13] = 0 is *ADD*, In[13] = 1 is *SUB*
  - *Immediate Data:* In[12] = 0 when no Immediate data is provided in the instruction, In[12] = 1 when Immediate data is provided in the instruction
- *Logical*: In[13] = 0 is *NAND*, In[13] = 1 is *NOR*
  - Immediate Data is NOT SUPPORTED for logical operations hence In[12] is UNUSED
- *Memory*: In[13:12] = 00 is *READ*, In[13:12] = 10 is *WRITE*
- *Branch:* In[13:12] = 01 for *Conditional Branch* and In[13:12] = 11 for *Unconditional Branch*
- *I/O:* In[13:12] = 00 is *OUT* (*SCREEN)*, In[13:12] = 10 is *INP* (*KEYBOARD)*

c. **In[11:9] act as follows**

- In[11:9] act as a 3-bit destination register address
  - if (*OPCODE* is *Arithmetic*) OR (*OPCODE* is *Logical*), or
  - if (*OPCODE* is *Memory*) AND (*OPTYPE* is *READ*), or
  - if (*OPCODE* is I/O) AND (*OPTYPE* is *INP*)
- In[11:9] act as a 3-bit "pointer register" Rx address
  - if (*OPCODE* is *Memory* AND *OPTYPE* is *WRITE)*, or
  - (*OPCODE* is I/O) AND (*OPTYPE* is *OUT*)
- In[11:9] act as Address of Rx if (*OPCODE* is *Branch)*

d. In[8:6] and In[5:3] act as follows:

- In[8:6] and In[5:3] act as *SOURCE1* and *SOURCE2* Register Addresses respectively if (*OPCODE* is *Arithmetic*) OR (*OPCODE* is *Logical*)
- In[8:6] acts as Address of Ry if (*OPCODE* is *Branch) and (OPTYPE is CONDITIONAL BRANCH)*.
- In[8:6] is the address of Ry if the instruction is Memory READ or WRITE.

e. In[5:0] act as follows:

- In[5:0] act as Immediate operand input if *OPCODE* is *Arithmetic.*
- In[5:0] are unused otherwise

Examples *(Note: Spaces in bit encoding are there for readability only, and X represents don't care or unused bit)*

Arithmetic

1. ***ADD R0, R1, R2*** - Operation is *R0=R1+R2* and bit encoding = 0000 000 001 010 XXX
2. ***ADDI R0, R1, 7*** - Operation is *R0 = R1+7* and bit encoding = 0001 000 001 000111
3. ***SUB R0, R1, R2*** - Operation is *R0=R1-R2* and bit encoding = 0010 000 001 010 XXX
4. ***SUBI R0, R1, 7*** - Operation is *R0 = R1-7* and bit encoding = 0011 000 001 000111

Logical

5. ***NAND R0, R1, R2*** - Operation is *R0=R1 NAND R2* and bit encoding = 0100 000 001 010 XXX
6. ***NOR R0, R1, R2*** - Operation is *R0=R1 NOR R2* and bit encoding = 0110 000 001 010 XXX

Memory

7. ***READ R0, R1*** - Operation is R0=RAM[R1] and bit encoding = 1000 000 001 XXXXXX
8. ***WRITE R0, R1*** - Operation is RAM[R0]=R1 and bit encoding = 1010 000 001 XXXXXX

Branch

9. ***BEQ R0, R1*** - Operation is to set the address of next instruction to ROM[R0] if (R1 == 0) and bit encoding = 1001 000 001 XXXXXX
10. ***JMP R2*** - Operation is to set the address of next instruction to ROM[R2] (unconditional JUMP) and bit encoding = 1011 010 XXX XXXXXX

I/O

11. ***OUT R0, R1*** - Operation is RAM[R0] = R1. This instruction should only access the Screen Memory MAP. Bit Encoding = 1100 000 001 XXXXXX
12. ***INP R0*** - Operation is R0 = RAM[<Keyboard Register Location>]. This instruction should only access the Keyboard Memory Map. Bit Encoding = 1110 000 XXX XXXXXX

BITBOT INSTRUCTION ENCODING CHART

Arithmetic	EXAMPLE	In[15] In[14]	In[13] In[12]	In[11] In[10] In[9]	In[8] In[7] In[6]	In[5] In[4] In[3]	In[2] In[1] In[0]
		OPCODE	OPTYPE	DEST REGISTER	SRC1 REGISTER	SRC2 REGISTER	
ADD	ADD R0, R1, R2 (i.e. R0 = R1 + R2)	0 0	0 0	Any Reg: 000 to 111	Any Reg: 000 to 111	Any Reg: 000-111	
ADDI	ADDI R0, R1, 8 (i.e. R0 = R1 + 8)		0 1			Six Bit Immediate Value (0-63)	
SUB	SUB R0, R1, R2 (i.e. R0 =R1 - R2)		1 0			Any Reg: 000-111	
SUBI	SUBI R0, R1, 8 (i.e. R0 = R1 - 8)		1 1			Six Bit Immediate Value (0-63)	
Logical	EXAMPLE	In[15] In[14]	In[13] In[12]	In[11] In[10] In[9]	In[8] In[7] In[6]	In[5] In[4] In[3]	In[2] In[1] In[0]
		OPCODE	OPTYPE	DEST REGISTER	SRC1 REGISTER	SRC2 REGISTER	UNUSED
NAND	NAND R0, R1, R2 (i.e. R0 = R1 NAND R2)	0 1	0	Any Reg: 000 to 111	Any Reg: 000 to 111	Any Reg: 000 to 111	
NOR	NOR R0, R1, R2 (i.e. R0 = R1 NAND R2)		1				
Memory	EXAMPLE	In[15] In[14]	In[13] In[12]	In[11] In[10] In[9]	In[8] In[7] In[6]	In[5] In[4] In[3]	In[2] In[1] In[0]
		OPCODE	OPTYPE	LEFT SIDE	RIGHT SIDE		
READ	READ R0, R1 (i.e. R0 = MEM[R1])	1 0	0 0	DEST REGISTER	SRC PTR REGISTER		
WRITE	WRITE R0, R1 (i.e. MEM[R0] = R1)		1 0	DEST PTR REGISTER	SRC REGISTER		
Branch	EXAMPLE	In[15] In[14]	In[13] In[12]	In[11] In[10] In[9]	In[8] In[7] In[6]	In[5] In[4] In[3]	In[2] In[1] In[0]
		OPCODE	OPTYPE	TARGET ADDRESS	SRC REGISTER		
JMP	JMP R0 (i.e. PCOut = R0)	1 0	1 1	Any Reg: 000 to 111			
BEQ	BEQ R0, R1 (i.e. PCOut = R0 if R1==0)		0 1		Any Reg: 000 to 111		
INPUT/OUTPUT	EXAMPLE	In[15] In[14]	In[13] In[12]	In[11] In[10] In[9]	In[8] In[7] In[6]	In[5] In[4] In[3]	In[2] In[1] In[0]
		OPCODE	OPTYPE	TARGET ADDRESS	SRC REGISTER		
INP	INP R0 (i.e. R0 = MEM[KEYBOARD])	1 1	1 0	Any Reg: 000 to 111			
OUT	OUT R0, R1 (i.e. SCREEN[R0] = R1)		0 0		Any Reg: 000 to 111		

[C] BITBOT Computer Resources

1. **BITBOT Computer**
   a. The BITBOT Computer is a 16-bit architecture consisting of 16-bit Instructions and Data. *Here, the 16-bit unit is also referred to as WORD. All capacities are expressed in this unit.*

b. The BITBOT Computer contains a Central Processing Unit (CPU), a 32K Instruction ROM, and 40K Data Memory.
c. There is one output device in the form of Screen and one input device in the form of Keyboard. Both of these I/O devices are memory-mapped to the Data Memory.

2. BITBOT Central Processing Unit (CPU)

a. The CPU contains a total of 8 16-bit General Purpose Registers: R0 through R7
b. The CPU also contains a 16-Bit custom ALU with two 16-bit inputs XInput and YInput, and capable of performing 2s complement arithmetic. The ALU can perform arithmetic and logical operations on its data inputs. The completely functional custom ALU design is provided to you in the release collateral.
c. The 16-bit output appears on the "AluOut" port and contains the result of performing Arithmetic and Logical Instructions.
d. Output flags such as overflow are not tracked.
e. PC is the 16-bit Program Counter with output "PCOut". The PC can be reset to 0 or loaded with contents of the selected Register to implement Conditional Branch (BEQ) or Unconditional Branch (JMP) instruction.
    i. By default, the PC increments its state by 1 on every instruction boundary.
    ii. The initial state of the PC is 0.

3. BITBOT Instruction ROM

a. The 32K Instruction ROM is organized with 32K 16-bit instruction words.
b. The Instruction ROM is available as a BUILT-IN CHIP so you do not have to implement it, but simply instantiate it in your BITBOT Computer implementation.

4. **BITBOT Data Memory**

a. The 40K Data Memory is organized with 40K 16-bit data words.
b. The first 32K Data words are dedicated to the RAM. The next 8K Data words are assigned to Memory-Mapped Screen Output Device. The next 1 data word is assigned to Memory-Mapped Keyboard Input Device. ***So practically, the data memory is "40K plus 1" word.***

5. **BITBOT I/O Devices**

a. As noted earlier, the I/O Devices are Memory-Mapped
b. The output device SCREEN is mapped to 8K words in the Data Memory
c. The input device Keyboard is mapped to 1 Word in the Data Memory
d. The Screen and Keyboard chips are available as BUILT-IN chips that may simply be instantiated in your Data Memory chip Implementation.

[D] INTERFACES

1. CPU Interface

a. **Input Pins**:
    i. Instruction: In[15:0],
    ii. Data from Data Memory: fromM[15:0],

iii. Reset (when set to 1, resets the PC state to 0)

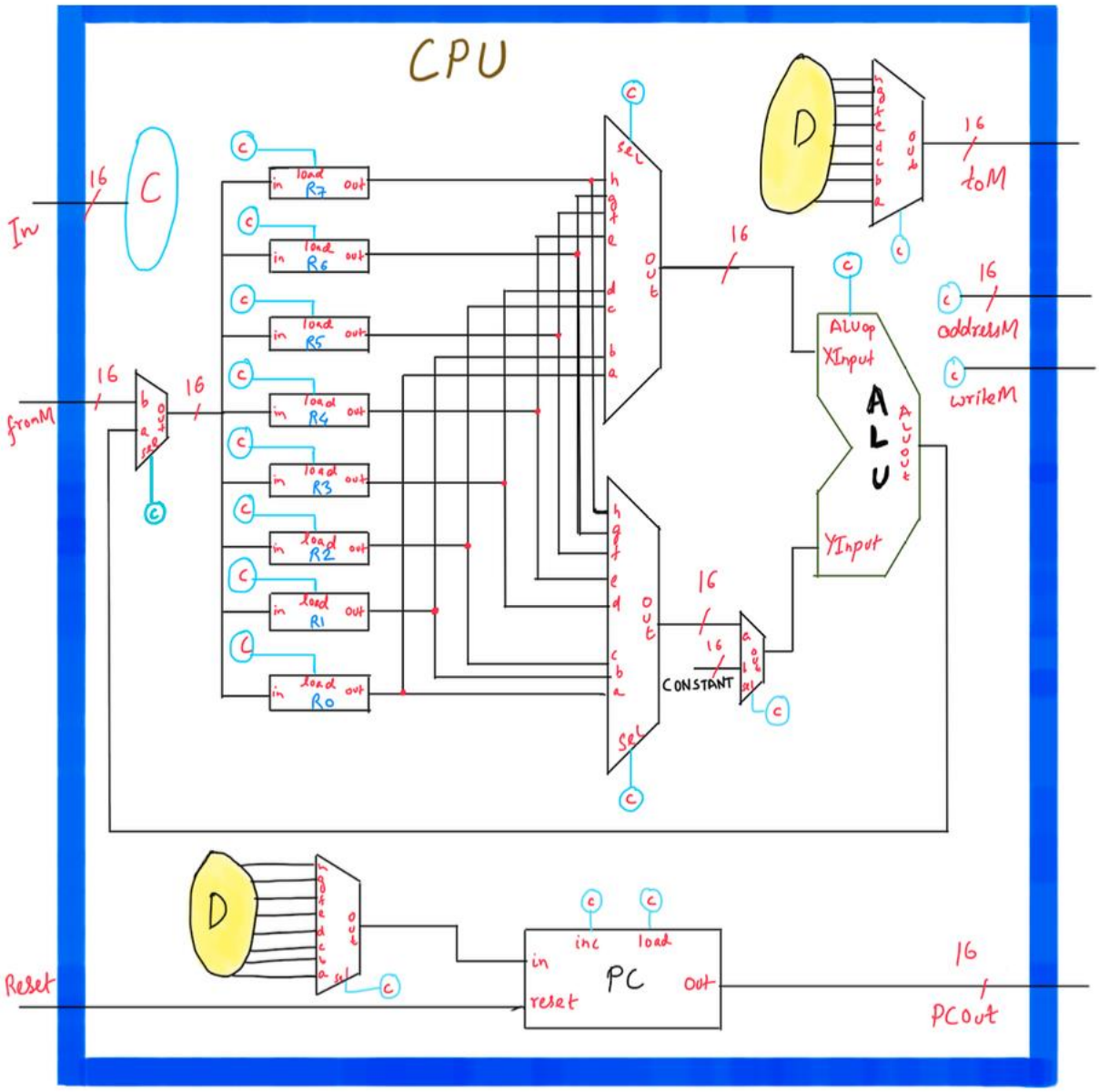
b. **Output Pins**:
   i. Instruction ROM Address: PCOut[15:0],
   ii. Data to Data Memory: toM[15:0],
   iii. Data Memory Write Enable: writeM,
   iv. Data Memory Address: addressM[15:0]

2. **Data Memory Interface**
   a. **Input Pins**:
      i. Data Input: toM[15:0],
      ii. Write Enable: load,
      iii. Address: addressM[15:0]
   b. **Output Pins**:
      i. Data Output: fromM[15:0]

3. **Instruction ROM Interface**
   a. **Input Pins**:
      i. Instruction ROM Address: PCOut[15:0]
   b. **Output Pins**:
      i. Instruction: In[15:0]

4. **Custom Arithmetic Logic Unit (ALU) Interface** (The fully functional ALU.hdl is provided to you in the collateral release so this information is just provided for reference)
   a. Input Pins:
      i. ALU Operation: AluOp[3:0]
      ii. ALU X Data Input: x[15:0]
      iii. ALU Y Data Input: y[15:0]
   b. **Output Pins**
      i. ALU Data Output: AluOut[15:0]
      ii. Overflow: overflow

[E] BITBOT CPU High Level Schematic



D = Datapath Source from Registers

c = Control signal generated using In and other Logic.

NOTATION FOR SYMBOLS

Implementation Tips

Complete the computer's construction in the following order:

(Data) Memory: This chip includes three chip-parts: RAM32K, Screen, and Keyboard. All of these chips are provided as built-in chips, so there is no need to implement them. All you need to do is to implement the top-level Data Memory by instantiating and connecting these three parts together.

CPU: This chip can be constructed according to the proposed CPU schematic and aligned with the definition of BITBOT Instructions. Note that the custom ALU chip is provided to you as a collateral.

Important Advisory: Please move the files RAM32K.class and RAM32K.hdl into tools/builtInChips inside nand2Tetris folder. After that you will be able to use RAM32K for your Memory.hdl. Also use the hardware simulator instead of VScode to test this. VScode has given error in the past. So you may decide to just stick to the hardware simulator if you encounter unexplained error.

If you choose to create any custom new chips, be sure to document and unit-test them carefully before you plug them into the architecture. And then please include them in your turn-in otherwise your design will fail during our testing.

Instruction memory: Use the built-in ROM32K chip.

Computer: The top-most Computer chip can be constructed according to the proposed implementation.

Tools

All the chips mentioned in this project, including the topmost Computer chip, can be implemented and tested using the supplied *Hardware Simulator*. For example, the Rect program draws a rectangle of width 16 pixels and length RAM[0] at the top-left of the screen. Now here is an interesting observation: normally, when you run a program on some computer, and you don't get the desired result, you conclude that the program is buggy. In our case though, the supplied Rect program is *bug-free*. Thus, if running this program yields unexpected results, it means that the computer platform on which it runs (Computer.hdl and/or some of its lower-level chip parts) is buggy. If that is the case, you have to debug your chips.

Reference Resources

The relevant readings for this project are Chapter 5, Appendix A, and Appendix B (as a reference, and use TAMU mail id to access). Be sure to participate in the labs to build the HACK computer. The principles you will learn in these labs will prepare you to succeed in building your own BITBOT computer! The resources that you need for this project are the supplied *Hardware Simulator* and the files provided in Project 5 collateral.

Rubrics

CPU: 60 points
Memory: 30 points
Computer: 10 points