CENG325 Program #3 (YB-60) – Blake Mommaerts
Due December 4

1. **What this is:**
   This program is an implementation of the RISC-V instruction set Architecture (ISA), which can perform tasks on memory structure, its monitor, and some simple program parsing. This implementation takes the name of the YB-60, a RISC-V emulator, programmed entirely in the Python programming language. It uses a 32-bit byte-addressable base integer architecture. It supports the RV32I base integer instruction set and the RV32M. The emulator also has 2^20 (1048576) bytes of memory, as well as 33 registers, one being the program counter, another being the zero register, and the remaining 31 being general-purpose registers.

   Pc: Program counter (32 bit) [Only the lower 20 bits are used]
   x0: zero register (32 bit) [Constant 0]
   x1-x31: GP registers (32 bit) [x2/sp constant 000FFFFF{HEX}]

   To ensure that memory is stored correctly as bytes, the 1MB of memory is initialized using bytearray(1048576). To ensure that a register's values are 32 bits, I intend to do a comparison to see if a value is greater than 2^31 when being inserted into a register, if so, it throws an error, and the program exits, OR it will trim to 32 bits depending on the instruction called.

   Now, in the third phase of the program, we are now executing 32-bit instructions provided through the data bytes stored in memory. The RISC-V machines are little-endian, so the least significant bit in the 32-bit instruction is the LSB in the lowest memory byte. Decoding/executing ends at an EBREAK statement, or the 00100073 in HEX. We can distinguish what instruction is provided by reading the opcode. For instance, decoding bits [6:2] of the opcode tells us it is a LOAD instruction if OPC[6:2] = 0. Another example is an OP instruction if OPC[6:2] = 12. The instructions we are supporting in this program are LOAD (I), STORE (S), BRANCH (B), JAL(I), JALR(J), OP(R), OPIMM(I), AUIPC(U), and LUI(U). Note that all six types of instructions are supported, if they are one of those listed above.

2. **How to run it:**
   This Python3 program uses command line arguments, so ensure that you are in its correct file path before attempting to run it. It will always begin with "python3 YB-60.py", which on its own starts the emulator with no pre-programming. If you wish to pre-program the emulator before booting it up, you can do so using Intel HEX in an x.obj file, with "x" being any valid file names you have saved in the same file path the program file. For instance, it can be run in these two ways:
   No file loaded:
   C: […]ceng325f23programs\Project1> python3 YB-60.py

With file loaded:
C: […]ceng325f23programs\Project1> python3 YB-60.py x.obj

Assuming you (a) provided NO file or (b) provided a VALID file, the emulator boots up, and assuming no additional errors were triggered, you are given a ">" prompt to start out with.
Your options from here are:

- Entering a hexadecimal address, which will display that byte tied to that hex address. **(displayMemory)**
- Entering a hexadecimal starting address, followed by a period, then an ending hexadecimal address, which will display all the bytes in between the two memory addresses. **(displayMemoryRange)**
- Entering a hexadecimal address, followed by a colon, then new values (bytes) for the memory locations. This updates the current memory locations. **(editMemory)**
- Entering a starting address in HEX, followed by an r (case sensitive), which the monitor will decode and execute instructions starting at the address (pc) and store instructions in memory until EBREAK is reached. **(runProgram)**
- Entering a starting address in HEX, followed by an s (case sensitive), which the monitor will decode and execute instructions starting at the address (pc) and store that instruction in memory. This function only does one instruction at a time, so to continue you will use the same starting address followed by an s until EBREAK is reached. [Example: If you start with 300s, you should only continue to type "300s" or "info" until an EBREAK is reached.] **(runProgramStep)**
- Entering a starting address in HEX, followed by a t (case sensitive), which the monitor will decode data bytes starting at the address (pc) and print out the instruction in the form of assembly. **(disassembleCode)**
- Entering "info" will display registers x0-x31 and their contents. **(displayInfo)**
- Entering "exit" will close the monitor and exit the program. Ctrl-C and Ctrl-Z+Enter are also meant to exit the program. EOF was noted as Ctrl-D, but it is Ctrl-Z+Enter on Windows. **(exitMonitor)**

If there are any additional questions, please contact me at blake.mommaerts@mines.sdsmt.edu.

3. **Additional Documentation**

My implementation of the YB-60 uses a class to achieve its functionality. This class consists of all the monitor functions of this program (aside from main). Now on to the functions:

__init__

- Parameters:
  - self – Pointer towards YB-60 object.
- Purpose: Sets up the current YB-60 emulator by initializing 1 MB of memory, setting all the registers to zero, as well as the program counter.
- How it was tested: Through the display functions, they display that the bytes of the memory are initialized to "00" assuming no programs have been loaded yet.

loadProgram

- Parameters:
  - self – Pointer towards YB-60 object.
  - fileName – The name of the file provided in the command line (if one exists).
- Purpose: When more than 1 sys command is provided in the command line, it will attempt to read this argument then open a file using the argument as its file name. Upon successful opening of the file, it will then attempt to parse it via Intel HEX and load information into memory. If the file consists of an invalid type of file, HEX, or HEX record, errors are thrown, and startup is cancelled. Otherwise, the program successfully reads and loads the Intel HEX data into memory-the emulator boots up with "> ", ready for other functions.
- How it was tested: Using files provided in our project's handout, I entered their file names in the command line and compared the expected output in the PDF to my output. I also made a few custom test files myself (some valid, some not) and got the correct output for those as well.

displayMemory

- Parameters:
  - self – Pointer towards YB-60 object.
  - adrStr – The HEX address string, typed into the command prompt.
- Purpose: The user types in the memory address in HEX in the command prompt, and the monitor returns the byte (also in HEX) at that location.
- How it was tested: Functionality was tested by comparing the expected output provided in the PDF (testX.obj, with X being a number 1-4) to my own output. Starting the monitor without a file and editing memory locations was also performed – the addresses that were edited in were displayed correctly. Once I achieved identical input and output compared to the PDF, the code for **displayMemory** seemed to be working as intended.

displayMemoryRange

- Parameters:
  - self – Pointer towards YB-60 object.
  - startAdrStr – The starting HEX address string, typed into the command prompt, before the '.'
  - endAdrStr – The ending HEX address string, typed into the command prompt, after the '.'
- Purpose: The user types in the starting address in HEX, followed by a period and then the ending address in HEX in the command prompt. The monitor returns the bytes of data between those two locations.
- How it was tested: Functionality was tested by comparing the expected output provided in the PDF (testX.obj, with X being a number 1-4) to my own output. Starting the monitor without a file and editing memory locations was also performed – the addresses that were edited in were displayed correctly. Once I achieved identical input and output compared to the PDF, the code for **displayMemoryRange** seemed to be working as intended.

editMemory

- Parameters:
  - self – Pointer towards YB-60 object.
  - editCmd – string of the starting address to be edited, followed by a colon, then bytes of data to be placed into the starting address and subsequent memory addresses.
- Purpose: By typing in the starting address in HEX, followed by a colon, and then the new values for the memory locations at the monitor prompt, the monitor updates the starting address and subsequent locations to the provided values, assuming they are valid.
- How it was tested: Testing for this file involves using the function through the command prompt. Using the same input as the PDF (Appendix B) and then comparing the outputs, they appear to be identical. This was performed both with and without files.

runProgram

- Parameters:
  - self – Pointer towards YB-60 object.
  - startAdrStr - The starting HEX address string, typed into the command prompt, followed by an ‘r’.
- Purpose: After typing in the starting address in HEX, followed by an r at the monitor prompt, this function will execute all code starting at the address. It starts at the program counter, clear the registers, decodes the instructions, prints out the program counter, instruction HEX, instruction name plus its registers/immediate, and stops at an EBREAK statement. Since each supported instruction in RISC-V is four bytes (32 bits), it increments the program counter by 4 per instruction unless certain instructions alter the program counter. RISC-V machines are little-endian, so LSB (least significant bit) in the 32-bit instruction is the LSB in the lowest memory byte.
- How it was tested: Using the same input as the PDF (Appendix B) and then comparing the outputs, they appear to be identical. Tests were performed both with and without files. Additionally, I used my own file (blakeTest_1.obj) that contained at least 1 type of each instruction, so at least one R, I, S, B, J, and U function were tested and displayed the instructions/register values I expected. I also tested out test cases provided by others such as Tim, Alex, and Dr. Karlsson from within the discussion forum.

```
python3 YB-60.py blakeTest_1

> 300r
  PC      OPC         INST      rd      rs1     rs2/imm
 00300 F973C403    LBU    01000 00111 111110010111
 00304 F2A4A623     SW             01001 01010 111100101100
 00308 FFDB75B7    LUI    01011          11111111110110110111
 0030C 00FAD617  AUIPC 01100          00000000111110101101
 00310 40E68733    SUB     01110 01101 01110
 00314 FB7FF06F    JAL     00000          11111111111110110110
 00318 FE708067   JALR    00000 00001 111111100111
 0031C 807360E3   BLTU    00110 00111 1100000000000
 00320 00100073 EBREAK
```

runProgramStep

- Parameters:
  - self – Pointer towards YB-60 object.
  - startAdrStr - The starting HEX address string, typed into the command prompt, followed by an ‘s’.
- Purpose: After typing in the starting address in HEX, followed by an s at the monitor prompt. To begin, this function will clear the registers, then execute one instruction starting at the program counter. It will print out the program counter, instruction HEX, instruction name plus its registers/immediate, and return to the monitor. Since each supported instruction in RISC-V is four bytes (32 bits), it increments the program counter by 4 per instruction. RISC-V machines are little-endian, so LSB (least significant bit) in the 32-bit instruction is the LSB in the lowest memory byte. To run the next instruction, you will still type the original starting address followed by an s even if that is not the starting address for the next instruction. If you wish to complete execution step-by-step, keep typing “xxxxxs” (x’s being the starting address) or “info” into the monitor until you reach the EBREAK statement. Any other functions will reset the step function’s progress.
- How it was tested: I tested this function primarily through trial and error. I would attempt running the function with one of my test cases (also some provided by colleagues in the discussion forum) and kept entering 300s as it progressed along the instructions. I noticed that it would keep its progress after running functions like 300r and 300t which was not desirable, so I edited my handling of commands to reset the steps, should other functions be called. Once it reaches EBREAK, assuming a user tries to keep using the step function, it will loop on the EBREAK statement. It is advised to use a different function or exit after this occurs if there is no more testing or usage necessary.

disassembleCode

- Parameters:
  - self – Pointer towards YB-60 object.
  - startAdrStr – The starting HEX address string, typed into the command prompt, followed by a ‘t’.
- Purpose: After typing in the starting address string in HEX, followed by a t at the monitor prompt, this function decodes instructions stored as data bytes in memory in the form of assembly. This is to make the instruction more humanly readable. Since each supported instruction in RISC-V is four bytes (32 bits), it increments the program counter by 4 per instruction. RISC-V machines are little-endian, so LSB (least significant bit) in the 32-bit instruction is the LSB in the lowest memory byte. All possible formats for supported instructions are listed below:
  - Register and Register: mnemonic, rd, rs1, rs2
  - Register and Immediate: mnemonic, rd, rs1, imm
  - Load format: mnemonic, rd, imm(rs1)
  - Store format: mnemonic, rs2 imm(rs1)
  - Jal format: mnemonic, rd, imm
  - Jalr format: mnemonic, rd, imm(rs1)
  - Branch format: mnemonic, rs1, rs2, imm
- How it was tested: To start, I compared my output to that of the PDF’s. Then I created my own test cases by writing out 32-bit instructions on paper, converted them to HEX, and inserted the instructions as data bytes using **editMemory** (now also in blakeTest_1.obj). I compared the output to what I had written down and compared it to the decoded instruction.

decodeInstruction

- Parameters:
  - self – Pointer towards YB-60 object.
  - inst – The current 32-bit instruction passed into the function.
- Purpose: A helper function to both **runProgram** and **disassembleCode**, it parses a single instruction to determine what type of instruction it is, what specific instruction within the type (ex: I-type, addi), and which registers are involved (could be a specific combination of rd, rs1, rs2).
- How it was tested: As a starting measure, I made sure my output matched the PDF's. Afterwards, I made my own test cases by writing out 32-bit instructions on paper, converted them to HEX, and inserted the instructions as data bytes using **editMemory**. I compared the output to what I had written down to the decoded instruction. My test cases include at least 1 of each instruction type (see below).

```
Python3 YB-60.py blakeTest_1.obj
>300t
    lbu x8, -105(x7)
     sw x10, -212(x9)
     lui x11, -585
  auipc x12, 4013
    sub x14, x13, x14
     jal x0, -74
    jalr x0, -25(x1)
   bltu x6, x7, -2048
ebreak
```

HEX: 03 C4 73 F9 23 A6 A4 F2 B7 75 DB FF 17 D6 FA 00 33 87 E6 40 6F F0 7F FB 67 80 70 FE E3 60 73 80 73 00 10 00

R_format

- Parameters:
  - self – Pointer towards YB-60 object.
  - inst – The current 32-bit instruction passed into the function.
- Purpose: A helper function that parses a known R-type instruction to obtain the values of each register. This helper function is used in **decodeInstruction**.
- How it was tested: Used file blakeTest_1. See **decodeInstruction** above.

I_format

- Parameters:
  - self – Pointer towards YB-60 object.
  - inst – The current 32-bit instruction passed into the function.
- Purpose: A helper function that parses a known I-type instruction to obtain the values of the registers and immediate. This helper function is used in **decodeInstruction**. If the immediate is negative (greatest bit being 1), the function performs sign extension.
- How it was tested: Used file blakeTest_1. See **decodeInstruction** above.

S_format

- Parameters:
  - self – Pointer towards YB-60 object.
  - inst – The current 32-bit instruction passed into the function.
- Purpose: A helper function that parses a known S-type instruction to obtain the values of each register and immediate. This helper function is used in **decodeInstruction**. If the immediate is negative (greatest bit being 1), the function performs sign extension.
- How it was tested: Used file blakeTest_1. See **decodeInstruction** above.

B_format

- Parameters:
  - self – Pointer towards YB-60 object.
  - inst – The current 32-bit instruction passed into the function.
- Purpose: A helper function that parses the now known B-type instruction to obtain the values of each register and immediate. This helper function is used in **decodeInstruction**. If the immediate is negative (greatest bit being 1), the function performs sign extension.
- How it was tested: Used file blakeTest_1. See **decodeInstruction** above.

U_format

- Parameters:
  - self – Pointer towards YB-60 object.
  - inst – The current 32-bit instruction passed into the function.
- Purpose: A helper function that parses the now known U-type instruction to obtain the values of the destination register and immediate. This helper function is used in **decodeInstruction**. If the immediate is negative (greatest bit being 1), the function performs sign extension.
- How it was tested: Used file blakeTest_1. See **decodeInstruction** above.

J_format

- Parameters:
  - self – Pointer towards YB-60 object.
  - inst – The current 32-bit instruction passed into the function.
- Purpose: A helper function that parses the now known J-type instruction to obtain the values of the destination register and immediate. This helper function is used in **decodeInstruction**. If the immediate is negative (greatest bit being 1), the function performs sign extension.
- How it was tested: See **decodeInstruction** above.

funct3Load

- Parameters:
  - self – Pointer towards YB-60 object.
  - funct3 – Bits 14 through 12 of the instruction; three bits that determine which type of LOAD instruction is called.
- Purpose: A helper function that determines which LOAD instruction will be called by reading the funct3 value within the instruction.
- How it was tested: Aside from comparisons with the examples provided to us, I made some 32-bit instructions with funct3s that did/didn't exist on the PDF. As expected, the program filtered out which instructions were supported and not supported.

functOpImm

- Parameters:
  - self – Pointer towards YB-60 object.
  - funct3 – Bits 14 through 12 of the instruction; three bits that determine which type of OPIMM instruction is called.
  - funct7 – Bits 31 through 25 of the instruction; seven bits that further determine which type of OPIMM instruction is called.
- Purpose: A helper function that determines which OPIMM instruction will be called by reading the funct3 and funct7 value within the instruction.
- How it was tested: I compared output with the examples provided to us, and it seemed correct. Additionally, I created some 32-bit instructions with funct3s/funct7s that did and did not exist on the PDF. The program was able to decipher which instructions were supported and not supported.

funct3Store

- Parameters:
  - self – Pointer towards YB-60 object.
  - funct3 – Bits 14 through 12 of the instruction; three bits that determine which type of STORE instruction is called.
- Purpose: A helper function that determines which STORE instruction will be called by reading the funct3 value within the instruction.
- How it was tested: After making some comparisons with the examples provided to us, it seemed the correct instructions were being inserted. To check further, I made some 32-bit instructions with funct3s that did/didn't exist on the PDF. As expected, the program filtered out which instructions were supported and not supported.

funct3Branch

- Parameters:
  - self – Pointer towards YB-60 object.
  - funct3 – Bits 14 through 12 of the instruction; three bits that determine which type of BRANCH instruction is called.
- Purpose: A helper function that determines which BRANCH instruction will be called by reading the funct3 value within the instruction.
- How it was tested: I made some comparisons with the examples provided to us; my BRANCH instructions were generated as expected. To ensure all my BRANCH instructions are called correctly, I created some 32-bit instructions with funct3s that did and didn't exist as codes for our given instructions. The program successfully filtered out the not supported instructions and kept the supported ones.

functOp

- Parameters:
  - self – Pointer towards YB-60 object.
  - funct3 – Bits 14 through 12 of the instruction; three bits that determine which type of OP instruction is called.
  - funct7 – Bits 31 through 25 of the instruction; seven bits that further determine which type of OP instruction is called.
- Purpose: A helper function that determines which OP instruction will be called by reading the funct3 and funct7 value within the instruction.
- How it was tested: I made some comparisons with the examples provided to us, and all OP instruction appeared as it did on the PDF. To ensure all my branch instructions are called correctly, I created some 32-bit instructions with funct3s that did/didn't exist on the PDF. The program successfully filtered out the not supported instructions and kept the supported ones.

getInstandReg

- Parameters:
  - self – Pointer towards YB-60 object.
  - decoded – A string of the decoded instruction to be parsed for its register/immediate values.
  - inst – The current 32-bit instruction passed into the function.
- Purpose: A helper function that parses the decoded output string to obtain the register/immediate values. Then, these values are reformatted to binary to be printed in **runProgram**.
- How it was tested: This function required intense trial and error, as the methodology of extracting the instruction/register values was string parsing. I placed print statements throughout the function to ensure what was being fed into the function and had to decipher if what came out was my intended output. A benefit that came out of this rigorous testing process was figuring out how to remove the parenthesis from the LOAD/STORE/JALR format using a built-in string.replace() function within Python. It made it much easier to extract registers values this way.

runInstruction

- Parameters:
  - self – Pointer towards YB-60 object.
  - decoded2 – A string of the decoded instruction that has been parsed for its register/immediate values.
  - inst – The current 32-bit instruction passed into the function.
- Purpose: A helper function that parses the decoded instruction and figures out what type of instruction it is. It will pass that information into one of 7 functions.
- How it was tested: Since this function just parses the opcode to determine which helper function to execute, testing was not difficult for this function. As long as the 32-bit instruction gets passed in, this function selects one of its helper functions, as the instruction has already been confirmed to be supported through the decoding phase. For the helper functions: by using my own and some provided test cases in the discussion forum (credit to Tim, Alex, and Dr. Karlsson for using some of theirs), I ran these test cases, compared register values, and got the expected results for each.

runROp

- Parameters:
  - self – Pointer towards YB-60 object.
  - decoded2 – A string of the decoded instruction that has been parsed for its register/immediate values.
- Purpose: A function that uses the decoded instruction to execute whichever R-type instruction's name was parsed.
- How it was tested: See **runInstruction** above.

runIOp

- Parameters:
  - self – Pointer towards YB-60 object.
  - decoded2 – A string of the decoded instruction that has been parsed for its register/immediate values.
- Purpose: A function that uses the decoded instruction to execute whichever I-type instruction's name was parsed.
- How it was tested: See **runInstruction** above.

runLoadOp

- Parameters:
  - self – Pointer towards YB-60 object.
  - decoded2 – A string of the decoded instruction that has been parsed for its register/immediate values.
- Purpose: A function that uses the decoded instruction to execute whichever Load-type instruction's name was parsed.
- How it was tested: See **runInstruction** above.

runSOp

- Parameters:
  - self – Pointer towards YB-60 object.
  - decoded2 – A string of the decoded instruction that has been parsed for its register/immediate values.
- Purpose: A function that uses the decoded instruction to execute whichever S-type instruction's name was parsed.
- How it was tested: See **runInstruction** above.

runBOp

- Parameters:
  - self – Pointer towards YB-60 object.
  - decoded2 – A string of the decoded instruction that has been parsed for its register/immediate values.
- Purpose: A function that uses the decoded instruction to execute whichever B-type instruction's name was parsed.
- How it was tested: See **runInstruction** above.

runJOp

- Parameters:
  - self – Pointer towards YB-60 object.
  - decoded2 – A string of the decoded instruction that has been parsed for its register/immediate values.
- Purpose: A function that uses the decoded instruction to execute whichever R-type instruction's name was parsed.
- How it was tested: See **runInstruction** above.

runUOp

- Parameters:
  - self – Pointer towards YB-60 object.
  - decoded2 – A string of the decoded instruction that has been parsed for its register/immediate values.
- Purpose: A function that uses the decoded instruction to execute whichever R-type instruction's name was parsed.
- How it was tested: See **runInstruction** above.

startMonitor

- Parameters:
  - self – Pointer towards YB-60 object.
- Purpose: This function will boot up the emulator when it is called. It provides the user with "> " to indicate when it is ready for user input. Assuming an exit command or fatal error is not caused, it will keep looping and providing the monitor prompt for each command given.
- How it was tested: Since the "> " prompt occurs upon startup of the monitor; testing was quite simple. All I had to do was enter the startup command in the command line. Trying some commands, such as **editMemory**, **displayMemory**, **displayMemoryRange** all executed and once again I was provided with "> ".

commandList

- Parameters:
  - self – Pointer towards YB-60 object.
  - cmd – The current command string for the monitor to handle.
- Purpose: This helper function parses the command/string provide to the monitor and determines what function needs to be called. It will detect a singular HEX address (**displayMemory**), period (**displayMemoryRange**), colon (**editMemory**), R (**runProgram**), T (**disassembleCode**), info (**displayInfo**), and exit (**exitMonitor**). Otherwise, it will throw an error.
- How it was tested: To test this helper function, I would keep executing random words, combinations of letters and numbers (not HEX), and numbers. Of course, random words are likely to not be HEX, combinations may or may not work, and numbers will just be an address. Anything that may slip by this function will more likely be caught in one of the other functions' error checks.

displayInfo

- Parameters:
  - self – Pointer towards YB-60 object.
- Purpose: Displaying the values currently stored in the registers after decoding and running instructions (running is currently not implemented).
- How it was tested: Just typed "info", then registers x0-x31 are displayed on the monitor, and the user is returned to the cmd. (Again, running instructions is not currently implemented, so all registers will hold only zeroes).

exitMonitor

- Parameters:
  - self – Pointer towards YB-60 object.
- Purpose: Exiting the monitor/program. Prints ">>>" indicating the exiting of the program.
- How it was tested: Just typed "exit", then ">>>" is displayed, and the user is returned to the cmd. CTRL-C and CTRL-Z + Enter have been tested and have the same intended result.

Main

- System Argument(s):
  - python3 YB-60.py (1 argument)
  - python3 YB-60.py testX.obj (2 arguments)
- Purpose: To create an instance of the YB-60 and execute it depending on what was provided in the command prompt.
- How it was tested: By typing both types of system arguments into the command prompt, the monitor booted up correctly in both instances. It will even take more than 2 arguments, only for it to look at the provided filename and ignore the rest.