Gage Jager
CENG 325 – CPU and GPU Organization and Architecture
10/25/2023
YB-60 Emulator Documentation

This document is up to date as of the end of Part 3 of the Project, originally due on 12/4/2023.

How to Run the Program

1. This program was written using Python 3.10.11, and includes the sys and string modules, which are built-in to the interpreter. As such, you do not need to install any additional libraries/modules.
2. Ensure that the Python script (main.py) and any object files you wish to input are located in the same directory.
3. Open a terminal / CMD window to the directory the files are located in.
4. In the terminal, use the following command, where objectFile.obj is the name of the object file you wish to input. Note that providing an object file is optional.
    a. python main.py [objectFile.obj]

YB-60 Emulator Information

The YB-60 is an implementation of the RISC-V Instruction Set Architecture, specifically RV32I and RV32M, featuring a 32-bit byte-addressable base integer architecture, 1 Megabyte of memory, and 33 32-bit registers. The registers include a Program Counter, “pc”, of which only the lower 20 bits are used, a Zero Register, “x0”, which holds a constant zero value, and 31 General Purpose registers, “x1” to “x31”.

Unlike the YB-60, our Emulator implementation will feature a Monitor that is not run from memory, unlike the OS that would be running on an actual YB-60. This allows us to use the entire contents of memory for holding data from the currently loaded object file, instead of having to make sure we have enough memory to hold the OS additionally.

When run, the YB-60 Emulator will read the data from the provided object file, if one was given, into the memory locations specified by object file. When the end of the object file is reached, or if no file was given, the emulator will output the menu prompt “> “. Currently, the functions of the emulator include:

1. Display the Contents of a Specific Memory Address
    a. This is done by entering a single memory address in hexadecimal, such as “> 12200”.
2. Display the Contents of a Range of Memory Addresses
    a. This is done by entering two memory addresses separated by a “.”, such as “> 300.31F”.
3. Edit Memory Locations
    a. This is done by entering a starting memory address, a “:” and a space, and entering a string of byte values, in hexadecimal, such as “> 300: A9 04 88”
4. Disassemble Object Code
    a. This is done by entering a starting memory address followed by the letter “T”, such as “> 300T”. This will output the RISC-V code that would have generated the data in memory at your specified location. It will continue disassembling the code in memory until it reaches the end of memory, or an “EBREAK” instruction.
5. Run Program Starting at Specified Address
    a. This is done by entering a starting memory address followed by the letter “R”, such as “> 300R”. This function will output the program counter, the hex representation of each line of code, the name of the instruction, and any registers or immediate values in the instruction, in binary. Simultaneously, the registers will be updated correspondingly to the instructions being ran. This will continue interpreting instructions until the program reaches the end of memory, or an “EBREAK” instruction.
6. Run Program Starting at Specified Address, with Breaks
    a. This is done by entering a starting memory address followed by the letter “S”, such as “> 300S”. This function will output the program counter, the hex representation of a line of code, the name of the instruction, and any registers or immediate values in the instruction, in binary. Simultaneously, the registers will be updated correspondingly to the instruction being ran. After each line, the user will be asked if they want to continue running the instructions, show the contents of the registers, or quit. This will continue until an “EBREAK” instruction is reached, the

end of memory is reached, or when the users specifies they want to stop.

7. Display Content of Registers
    a. This is done by entering “info” at the command prompt. The function outputs the contents of all 32 “x” registers in hexadecimal. As we do not currently run any instructions during function 5, the values outputted will always be “00000000”.
8. Exit the Program
    a. This is done by entering “exit” at the command prompt. It closes the emulator.

Function Descriptions

In main.py:

- Outside any Function
    - At the top of the file, I import sys to have access to sys.argv, so I can determine if I need to load from the object file or not (as well as have access to the object file in the first place). I also import string to have access to the hexdigits constant.
    - Following that, I have global variable declarations.
        - "labels" is the string that is outputted when "runProgram" is called, and is used to label the output formatting columns.
        - "EBREAK" is a string containing the hexadecimal representation of the EBREAK instruction. It is used to compare the current instruction to EBREAK, to determine if we should stop interpreting code.
        - "registers" holds the emulator's registers, and are all initially set to 0. The index corresponds to the register number, so x0 is index 0, and x31 is index 31. Index 32 corresponds to the program counter register.
        - "memory" is a bytearray with 2^{20} indexed locations, and serves as the 1MB of byte-addressable memory for the emulator.
    - At the bottom of the file, after all the functions, I call "main" to truly begin the program.
- clearRegisters()
    - "clearRegisters" is called inside "runProgram" and initializes all the emulator's registers to 0, except x2, the stack pointer, which is initialized to "FFFFF", or the maximum physical memory address.
    - This function was easily debugged, as the debugger can show me the values of every register as they are set during runtime. Additionally, the only register currently used anywhere in the program is "pc".
- loadProgram(objectFile)
    - "loadProgram" is called in the event that an object file is provided as a command line argument. "objectFile" is a read-only file input using the file name provided on the command line.
    - This function reads lines from "objectFile" one at a time using a while loop, verifies the line's checksum using "checksum", and loads data into memory based on the I16HEX data on each line.
        - Since the Record Type byte location is static, we can extract that data and perform the correct operations.
        - If the Record Type is "01", we have reached the EOF data, and can break out of the loop.
        - If the Record Type is "02", we extract the extended address data, and save it for later.
        - If the Record Type is "00", we extract the number of data bytes, the address data, and the data bytes themselves, calculate the full address, and begin storing the data bytes in the "memory" bytearray, indexed by the full address.
    - The function was primarily debugged by using the test object files provided, and using the same inputs as in the assignment's sample output, in order to see if the same file and input combinations produced the same results as the sample output.
- checksum(currentLine)

    - "checksum" is utilized during "loadProgram" in order to verify that a line of I16HEX has not been changed. "currentLine" is a String of the current I16HEX line, stripped of the leading ":" and ending "\n".
    - The checksum byte is separated from the data bytes, and the data bytes are summed. Following this, the last byte of the sum is isolated. We take the 2's Compliment of the isolated byte, and compare it to the given checksum byte. The function returns True or False depending on if the checksums match or not.
- RFormat(bitString)
    - "RFormat" is utilized during "disassembleCode" and "runProgram" to extract the values for rd, rs1, and rs2 from an R Format RISC-V instruction, represented as a string of binary values (bitString).
- IFormat(bitString)
    - "IFormat" is utilized during "disassembleCode" and "runProgram" to extract the values for rd, rs1, and imm from an I Format RISC-V instruction, represented as a string of binary values (bitString).
- SFormat(bitString)
    - "SFormat" is utilized during "disassembleCode" and "runProgram" to extract the values for rs1, rs2, and imm from an S Format RISC-V instruction, represented as a string of binary values (bitString).
- SBFormat(bitString)
    - "SBFormat" is utilized during "disassembleCode" and "runProgram" to extract the values for rs1, rs2, and imm from an SB Format RISC-V instruction, represented as a string of binary values (bitString).
- UFormat(bitString)
    - "UFormat" is utilized during "disassembleCode" and "runProgram" to extract the values for rd and imm from a U Format RISC-V instruction, represented as a string of binary values (bitString).
- UJFormat(bitString)
    - "UJFormat" is utilized during "disassembleCode" and "runProgram" to extract the values for rd and imm from a UJ Format RISC-V instruction, represented as a string of binary values (bitString).
- determineLoad(bitString, rd, rs1, imm)
    - "determineLoad" is utilized during "diassembleCode" and "runProgram" to determine which Load instruction is being used in a given RISC-V instruction, represented as a string of binary values (bitString).
    - The function extracts the value of func3 from the bitString and compares it to the func3 values of the Load instructions to determine which instruction is in use, and will return the name of the instruction.
    - Additionally, upon determining the instruction, the memory array, rd, rs1, and imm values are passed into the specific function in order to perform its operation.
- determineStore(bitString, rs1, rs2, imm)
    - "determineStore" is utilized during "diassembleCode" and "runProgram" to determine which Store instruction is being used in a given RISC-V instruction, represented as a string of binary values (bitString).

    - The function extracts the value of func3 from the bitString and compares it to the func3 values of the Store instructions to determine which instruction is in use, and will return the name of the instruction.
    - Additionally, upon determining the instruction, the memory array, rs1, rs2, and imm values are passed into the specific function in order to perform its operation.
- determineBranch(bitString, rs1, rs2, imm)
    - "determineBranch" is utilized during "diassembleCode" and "runProgram" to determine which Branch instruction is being used in a given RISC-V instruction, represented as a string of binary values (bitString).
    - The function extracts the value of func3 from the bitString and compares it to the func3 values of the Branch instructions to determine which instruction is in use, and will return the name of the instruction.
    - Additionally, upon determining the instruction, the rs1, rs2, and imm values are passed into the specific function in order to perform its operation.
- determineOPImm(bitString, rd, rs1, imm)
    - "determineOPImm" is utilized during "diassembleCode" and "runProgram" to determine which OP Imm instruction is being used in a given RISC-V instruction, represented as a string of binary values (bitString).
    - The function extracts the value of func3 and func7 from the bitString and compares it to the func3 and func7 values of the OP Imm instructions to determine which instruction is in use, and will return the name of the instruction.
    - Additionally, upon determining the instruction, the rd, rs1, and imm values are passed into the specific function in order to perform its operation.
- determineOP(bitString, rd, rs1, rs2)
    - "determineOP" is utilized during "diassembleCode" and "runProgram" to determine which OP instruction is being used in a given RISC-V instruction, represented as a string of binary values (bitString).
    - The function extracts the value of func3 and func7 from the bitString and compares it to the func3 and func7 values of the OP instructions to determine which instruction is in use, and will return the name of the instruction.
    - Additionally, upon determining the instruction, the rd, rs1, and rs2 values are passed into the specific function in order to perform its operation.
- displayAddress(userInput)
    - "displayAddress" is one of the potential user input functions, called when the input matches no other function. "userInput" is the user's input from the command line, following the "> ".
    - This function converts the user's string input into an int in order to access the proper memory location. The data at the memory address is converted into a hex string, before trimming off the leading "0x", leaving just hex characters. The characters are made uppercase for output uniformity, and a leading '0' is appended to single digit hex strings (0-F become 00-0F).
    - Following the data formatting, the inputted memory address is outputted along with the data from that address.
    - Debugging this function was straightforward, since the user input was able to be turned directly to an int to access memory. Additionally, due to how I formatted the function selection, this function is the default behavior after user input, meaning that every other function had to catch their inputs, leaving only leftovers for this function. So, if the command was not caught elsewhere, if ended up in the right place.
- displayAddressRange(userInput)

    - "displayAddressRange" is one of the potential user input functions, called when the input contains a '.' character. "userInput" is the user's input from the command line, following the "> ".
    - This function partitions the user input into a starting address and an ending address. The difference between the ending and starting addresses gives the number of bytes we need to output from memory, and can be used as an iterator in a while loop.
    - The while loop serves to output the data. First, the current address is added to a string before entering a second while loop that makes sure each line of output has no more than 8 bytes of data on it. Inside the second loop, bytes of data are pulled from memory and added to the string until there are either no more bytes to read or until 8 bytes have been read to one line. In whichever case, the string of output is printed to the console. If there are remaining bytes to read, the inner loop is repeated. Otherwise, the outer loop exits and returns to main.
    - This function was also fairly simple to debug, as it would either show the correct range of data or not (having non-zero data helped immensely in ensuring that the correct data was being pulled). It was mostly tested during the debugging of "loadProgram".
- editMemory(userInput)
    - "editMemory" is one of the potential user input functions, called when the input contains a ':' character. "userInput" is the user's input from the command line, following the "> ".
    - This function first partitions the user input into the starting address and the data bytes.
    - Next, a while loop is run, with the condition being that there is still a data byte to read in the string. Inside the loop, the first hex character pair is partitioned from the rest of the data, converted to an actual int instead of 2 char's, and stored in "memory" at the current address. The address is incremented and the loop will continue, if there was more data bytes to change.
    - This function was debugged primarily during the process of debugging "loadProgram", as multiple of the sample outputs made use of editing memory, which I made sure to match.
- disassembleCode(userInput)
    - "disassembleCode" is one of the potential user input functions, called when the input contains a 'T' character. "userInput is the user's input from the command line, following the "> ".
    - The function is a giant while loop, governed by the current memory location. We begin the loop at the memory location specified by "userInput", and will continue until we hit the end of memory. Additionally, we will break out of the loop if the current instruction is "EBREAK".
    - Once in the loop, we read 4 consecutive bytes from memory, starting at the current memory address, and combine the bytes using shifts and adds to form the hexadecimal representation of a single RISC-V instruction. It is at this point that we determine if we have an "EBREAK" instruction, and handle that case before breaking out of the loop.
    - If the instruction is not an "EBREAK", we convert the instruction to a binary string and extract the OPCode bits from the string. We compare the OPCode bits in an if-else chain to determine if the instruction is a Load, Store, Branch, Jalr, Jal, OP Imm, OP, AUIPC, or LUI instruction.
    - We then pass the binary string to the "xFormat" function corresponding to each instruction type to extract additional data from the instruction, then pass the binary

string to the "determineX" function for a Load, Store, Branch, OP Imm, or OP instruction to determine which specific instruction is in use. Jalr, Jal, AUIPC, and LUI are already specific, so we do not need a "determineX" function for those 4.
    - Lastly, we format and print the instruction's data as it would be in RISC-V assembly code, prior to being turned into an object file. We then increment the current memory address by 4, and continue the while loop.
- runProgram(userInput)
    - "runProgram" is one of the potential user input functions, called when the input contains an 'R' character. "userInput" is the user's input from the command line, following the "> ".
    - This function first clears the emulator's registers before formatting the user input to obtain the starting memory address. Afterwards, it prints the "labels" string, and enters a while loop, governed by the current memory address. The loop will continue until we hit the end of memory. Additionally, we will break out of the loop if the current instruction is "EBREAK".
    - Once in the loop, we read 4 consecutive bytes from memory, starting at the current memory address, and combine the bytes using shifts and adds to form the hexadecimal representation of a single RISC-V instruction. It is at this point that we determine if we have an "EBREAK" instruction, and handle that case before breaking out of the loop.
    - If the instruction is not an "EBREAK", we convert the instruction to a binary string and extract the OPCode bits from the string. We compare the OPCode bits in an if-else chain to determine if the instruction is a Load, Store, Branch, Jalr, Jal, OP Imm, OP, AUIPC, or LUI instruction.
    - We then pass the binary string to the "xFormat" function corresponding to each instruction type to extract additional data from the instruction, then pass the binary string to the "determineX" function for a Load, Store, Branch, OP Imm, or OP instruction to determine which specific instruction is in use. Jalr, Jal, AUIPC, and LUI are already specific, so we do not need a "determineX" function for those 4.
    - Lastly, we format and print the instruction's data, showing the "PC" and "OPC" values in hexadecimal, followed by the instruction name in text and the rd, rs1, rs2, and immediate values in binary, if applicable to the specific instruction. We then increment the current memory address by 4, and continue the while loop.
- stepThroughProgram(userInput)
    - "stepThroughProgram" is identical to "runProgram", bar one important distinction.
    - The instruction running loop in "stepThroughProgram" is additionally governed by a user input called "cont".
    - After each instruction is ran, the user is asked if they would like to continue running instructions, output the registers, or stop running the instructions. Selecting the register output will call the "printRegisters" function, then show the input prompt again. As long as the user does not enter "n" or "N", the next instruction will be ran, unless there is an "EBREAK".
- printRegisters()
    - This function extends the stored value in each "x" register to a full 8 hex characters

and outputs them all individually.

- main()
    - This function first checks “sys.argv” to determine if an object file was provided as a command argument. If one was, the file is opened, read-only, and passed to “loadProgram”, before being closed once it is finished reading the data.
    - Whether an object file was provided or not, the function will then enter a while loop that outputs “> “ and waits for user input. Depending on the input, one of the above functions may be called, or the program may exit. The while loop will act infinitely until the program is exited or it crashes.

In Instructions.py:

- add(registers, rd, rs1, rs2)
    - “add” obtains the values from the rs1- and rs2-indexed registers, adds them together, and stores the result in the rd-indexed register.
- addi(registers, rd, rs1, imm)
    - “addi” obtains the value from the rs1-indexed register, adds it to the immediate value, and stores the result in the rd-indexed register.
- andFunc(registers, rd, rs1, rs2)
    - “andFunc” (and is a keyword in Python) performs a bitwise and between the values in the rs1- and rs2-indexed registers, and stores the results in the rd-indexed register.
- andi(registers, rd, rs1, imm)
    - “andi” performs a bitwise and between the value in the rs1-indexed register and the immediate value, and stores the result in the rd-indexed register.
- auipc(registers, rd, imm)
    - “auipc” shifts the immediate value left 12 bits, adds the value to the Program Counter’s value, then stores the result in the rd-indexed register.
- beq(registers, rs1, rs2, imm)
    - “beq” compares the values in the rs1- and rs2-indexed registers, and if they are equal, the immediate value is added to the program counter.
- bge(registers, rs1, rs2, imm)
    - “bge” compares the values in the rs1- and rs2-indexed registers, and if rs1’s value is greater than or equal to rs2’s value, the immediate value is added to the program counter.
- bgeu(registers, rs1, rs2, imm)
    - “bgeu” compares the unsigned values in the rs1- and rs2-indexed registers, and if rs1’s value is greater than or equal to rs2’s value, the immediate value is added to the program counter.
- blt(registers, rs1, rs2, imm)
    - “blt” compares the values in the rs1- and rs2-indexed registers, and if rs1’s value is less than rs2’s value, the immediate value is added to the program counter.
- bltu(registers, rs1, rs2, imm)
    - “bltu” compares the unsigned values in the rs1- and rs2-indexed registers, and if rs1’s value is less than rs2’s value, the immediate value is added to the program counter.
- bne(registers, rs1, rs2, imm)

    - "bne" compares the values in the rs1- and rs2-indexed registers, and if the values are not equal, the immediate value is added to the program counter.
- jal(registers, rd, imm)
    - "jal" first saves the location of the next instruction (PC + 4) into the rd-indexed register, then updates the program counter by adding the immediate value to it.
- jalr(registers, rd, rs1, imm)
    - "jalr" first saves the location of the next instruction (PC + 4) into the rd-indexed register, then updates the program counter to the rs1-indexed register's value plus the immediate value.
- lb(registers, memory, rd, rs1, imm)
    - "lb" obtains the value stored in the memory array at the location of the rs1-indexed register's value plus the immediate value, and puts that obtained value in the rd-indexed register.
- lbu(registers, memory, rd, rs1, imm)
    - "lbu" obtains the value stored in the memory array at the location of the rs1-indexed register's value plus the unsigned immediate value, and puts that obtained value in the rd-indexed register.
- lh(registers, memory, rd, rs1, imm)
    - "lh" obtains the 2 values stored in the memory array at the location of the rs1-indexed register's value plus the immediate value, and the sequential value in the memory array. Then it combines the obtained values and puts that value in the rd-indexed register.
- lhu(registers, memory, rd, rs1, imm)
    - "lhu" obtains the 2 values stored in the memory array at the location of the rs1-indexed register's value plus the unsigned immediate value, and the sequential value in the memory array. Then it combines the obtained values and puts that value in the rd-indexed register.
- lui(registers, rd, imm)
    - "lui" shifts the immediate value to the left 12 bits, then stores the value in the rd-indexed register.
- lw(registers, memory, rd, rs1, imm)
    - "lw" obtains the 4 values stored in the memory array at the location of the rs1-indexed register's value plus the immediate value, and the 3 sequential values in the memory array. Then it combines the obtained values and puts that value in the rd-indexed register.
- orFunc(registers, rd, rs1, rs2)
    - "orFunc" (or is a keyword in Python) performs a bitwise or between the values in the rs1- and rs2-indexed registers, and stores the result in the rd-indexed register.
- ori(registers, rd, rs1, imm)
    - "ori" performs a bitwise or between the value in the rs1-indexed register and the immediate value, and stores the result in the rd-indexed register.
- sb(registers, memory, rs1, rs2, imm)
    - "sb" stores the lowest byte in the rs2-indexed register, into memory at the rs1-indexed register's value plus the immediate value.

- sh(registers, memory, rs1, rs2, imm)
    - “sh” stores the lowest 2 bytes in the rs2-indexed register, into memory at the rs1-indexed register’s value plus the immediate value, and the sequential memory value.
- sll(registers, rd, rs1, rs2)
    - “sll” shifts the value in the rs1-indexed register left a number of bits equal to the value in the rs2-indexed register, then puts the result into the rd-indexed register.
- slli(registers, rd, rs1, imm)
    - “slli” shifts the value in the rs1-indexed register left a number of bits equal to the value of the immediate, then puts the result into the rd-indexed register.
- slt(registers, rd, rs1, rs2)
    - “slt” sets the rd-indexed register to 1 if the value in the rs1-indexed register is less than the value in the rs2-indexed register, and to 0 otherwise.
- slti(registers, rd, rs1, imm)
    - “slti” sets the rd-indexed register to 1 if the value in the rs1-indexed register is less than the immediate value, and to 0 otherwise.
- sltiu(registers, rd, rs1, imm)
    - “sltiu” sets the rd-indexed register to 1 if the unsigned value in the rs1-indexed register is less than the unsigned immediate value, and to 0 otherwise.
- sltu(registers, rd, rs1, rs2)
    - “sltu” sets the rd-indexed register to 1 if the unsigned value in the rs1-indexed register is less than the unsigned value in the rs2-indexed register, and to 0 otherwise.
- sra(registers, rd, rs1, rs2)
    - “sra” performs an arithmetic right shift on the rs1-indexed register’s value by a number of bits equal to the value of the rs2-indexed register’s value, then puts the result into the rd-indexed register.
- srai(registers, rd, rs1, imm)
    - “srai” performs an arithmetic right shift on the rs1-indexed register’s value by a number of bits equal to the value of the immediate, then puts the result into the rd-indexed register.
- srl(registers, rd, rs1, rs2)
    - “srl” performs a logical right shift on the rs1-indexed register’s value by a number of bits equal to the value of the rs2-indexed register’s value, then puts the result into the rd-indexed register.
- srli(registers, rd, rs1, imm)
    - “srli” performs a logical right shift on the rs1-indexed register’s value by a number of bits equal to the value of the immediate, then puts the result into the rd-indexed register.
- sub(registers, rd, rs1, rs2)
    - “sub” obtains the values from the rs1- and rs2-indexed values, subtracts the rs2 value from the rs1 value, and puts the result into the rd-indexed register.
- sw(registers, memory, rs1, rs2, imm)
    - “sw” stores the 4 bytes in the rs2-indexed register, into the memory at the rs1-indexed register’s value plus the immediate value, and the 3 sequential memory values.

- xor(registers, rd, rs1, rs2)
    - “xor” performs a bitwise xor operation on the values in the rs1- and rs2-indexed registers, and puts the result in the rd-indexed register.
- xori(registers, rd, rs1, imm)
    - “xori” performs a bitwise xor operation on the value in the rs1-indexed register and the immediate value, and puts the result in the rd-indexed register.
- mul(registers, rd, rs1, rs2)
    - “mul” obtains the values in the rs1- and rs2-indexed registers, multiplies them, then stores the lower 32 bits into the rd-indexed register.
- mulh(registers, rd, rs1, rs2)
    - “mulh” obtains the values in the rs1- and rs2-indexed registers, multiplies them, then stores the upper 32 bits into the rd-indexed register.
- mulhsu(registers, rd, rs1, rs2)
    - “mulhsu” obtains the values in the rs1- and rs2-indexed registers, multiplies them, then stores the upper 32 bits into the rd-indexed register, knowing that one register is signed and the other is unsigned.
- mulhu(registers, rd, rs1, rs2)
    - “mulhu” obtains the values in the rs1- and rs2-indexed registers, multiplies them, then stores the upper 32 bits into the rd-indexed register, knowing that both registers are unsigned.
- div(registers, rd, rs1, rs2)
    - “div” obtains the values in the rs1- and rs2-indexed registers, performs integer division on them, then stores the result into the rd-indexed register.
- divu(registers, rd, rs1, rs2)
    - “divu” obtains the values in the rs1- and rs2-indexed registers, performs integer division on them, then stores the result into the rd-indexed register, knowing that both registers are unsigned.
- rem(registers, rd, rs1, rs2)
    - “rem” obtains the values in the rs1- and rs2-indexed registers, performs modulus on them, then stores the result into the rd-indexed register.
- remu(registers, rd, rs1, rs2)
    - “remu” obtains the values in the rs1- and rs2-indexed registers, performs modulus on them, then stores the result into the rd-indexed register, knowing that both registers are unsigned.