Project I - RISC-V RV32I Simulator

CSCE 2303 - 01

Merna Wael Abdelbadie - 900203731

Maya Hussein - 900201198

Farah Fathy - 900201155

The American University in Cairo

Implementation Description:

First, an unordered map for the memory that holds the address as well as the value each address contains is created. Also,another unordered_map is created for registers, which holds the register name along with the value in each register. Both unordered maps are global variables.

Main Function:

In the main function, the user is asked to enter the starting address of the assembly code, which is then given to the program counter (pc). Then, a function called `readFromRegister()` is called. This void function simply reads a file that has the 32 registers all initialized to zero and stores them in the registers unordered_map. Then, `readFromRegister()` calls another void function called `readFromProgram()`. `readFromProgram()` reads the file that contains all the initial data and then updates the register unordered map accordingly. To read the assembly instructions, `readFromAssembly()` gets the text line by line and stores it in other unordered_map instructions. Initially, after reading every line and checking if a label exists, it is stored into another `unordered_map <string, int> label` and is simultaneously deleted from the instruction before insertion.

`selectInstruction()` is a category checklist for the instruction formats, where every word calls its corresponding function along with `printMemory&Register()`. If the instruction is either `ECALL`, `EFENCE`, or `EBREAK`, then the memory and register are printed out, and the program exits.

Some helping functions:

1. `getPartsforLoading()`: simply takes the instruction line from the instructions unordered map and gets the source and destination registers and offset and returns them by reference to the caller function (Callers are `loadWord, loadHalfWord, loadByte, loadByteUnsigned, loadHalfWordUnsigned`).
2. `getPartsforStoring()`: simply takes the instruction line from the instructions unordered map and gets the source and destination registers and offset and returns them by reference to the caller function (Callers are `storeWord, storeHalfWord, storeByte`).
3. `getPartsforRegtoReg()`: Based on the instruction line stored in the instructions unordered map, the function extracts the first source, second source, and the destination and returns them by reference to the caller function. The caller functions are `ADD, SUB, AND, OR, XOR, SLL, SRL, SRA, SLT.`
4. `getPartsforRegtoReg_Imm()`: Based on the instruction line stored in the instructions unordered map, the function extracts the first source, immediate, and the destination and returns them by reference to the caller function. The caller functions are `ADDI, ANDI, ORI, XORI, SLLI, SRLI, SRAI, SLTI.`

Instruction Functions:

1. `loadWord()`: this function simply calls `getsPartsforLoading()`, then it gets the source value from the `unordered_map <string, long long> registers` by searching for the value of the source register. After that it gets the value stored in the memory starting from the `sourceValue + offset` then moving three more places into the memory to get the whole word. It then concatenates all the bits received from the

memory. After that, a simple check of whether the number is positive or negative occurs. If the number is negative (MSB in the string is 1), then all zeros are changed to ones and all ones are changed to zeros. The modified string then is changed to a decimal value which is then changed to a long long called `decimalDestinationValue`. `decimalDestinationValue` is then incremented by one and negated and saved to the corresponding place of the destination register in the `unordered_map <string, long long> registers`. If the number is positive, it is simply changed from binary to decimal then to long long called `decimalDestinationValue` which is then saved to the corresponding place of the destination register in the registers unordered map.

2. `loadHalfWord()`: this function simply calls `getsPartsforLoading()`, then it gets the source value from the `unordered_map <string, long long> registers` by searching for the value of the source register. After that it gets the value stored in the memory starting from the `sourceValue + offset` then moving one more place into the memory to get the half word. It then concatenates all the bits received from the memory. After that, a simple check of whether the number is positive or negative occurs. If the number is negative (MSB in the string is 1), then all zeros are changed to ones and all ones are changed to zeros. The modified string then is changed to a decimal value which is then changed to an long long called `decimalDestinationValue`. `decimalDestinationValue` is then incremented by one and negated and saved to the corresponding place of the destination register in the `unordered_map <string, long long> registers`. If the number is positive, it is simply changed from binary to decimal then to long long called `decimalDestinationValue` which is then saved to the corresponding place of the destination register in the registers unordered_map.

3. loadByte(): this function simply calls getsPartsforLoading, then it gets the source value from the unordered_map <string, long long> registers by searching for the value of the source register. After that it gets the value stored in the memory from the sourceValue + offset. After that, a simple check of whether the number is positive or negative occurs. If the number is negative (MSB in the string is 1), then all zeros are changed to ones and all ones are changed to zeros. The modified string then is changed to a decimal value which is then changed to an long long called decimalDestinationValue. decimalDestinationValue is then incremented by one and negated and saved to the corresponding place of the destination register in the registers unordered map. If the number is positive, it is simply changed from binary to decimal then to long long called decimalDestinationValue which is then saved to the corresponding place of the destination register in the registers unordered map.
4. loadByteUnsigned(): this function simply calls getsPartsforLoading(), then it gets the source value from the unordered_map <string, long long> registers by searching for the value of the source register. After that it gets the value stored in the memory from the sourceValue + offset. After that, a simple check of whether the number is positive or negative occurs. If the number is negative (MSB in the string is 1), then all zeros are changed to ones and all ones are changed to zeros. The modified string then is changed to a decimal value which is then changed to an long long called decimalDestinationValue. decimalDestinationValue is then incremented by one and saved to the corresponding place of the destination register in the unordered_map <string, long long> registers. If the number is positive, it is simply changed from binary to decimal then to long long called

decimalDestinationValue which is then saved to the corresponding place of the destination register in the `unordered_map <string, long long> registers`.

5. `loadHalfWordUnsigned()`: this function simply calls `getsPartsforLoading()`, then it gets the source value from the registers unordered map by searching for the value of the source register. After that it gets the value stored in the memory starting from the sourceValue + offset then moving 1 more place into the memory to get the half word. It then concatenates all the bits received from the memory. After that, a simple check of whether the number is positive or negative occurs. If the number is negative (MSB in the string is 1), then all zeros are changed to ones and all ones are changed to zeros. The modified string then is changed to a decimal value which is then changed to an long long called decimalDestinationValue. decimalDestinationValue is then incremented by one and saved to the corresponding place of the destination register in the registers unordered map. If the number is positive, it is simply changed from binary to decimal then to long long called `decimalDestinationValue` which is then saved to the corresponding place of the destination register in the registers unordered map.
6. `StoreWord()`: this function simply calls `getPartsforStoring` function, then gets the source value from the `unordered_map <string, long long> registers` by searching for the value of the source register. Then bitset is used to change the sourceValue into binary instead of decimal. The destination value is also retrieved from the registers unordered map. As this is a word (32 bits) and the memory is byte addressable, 4 places in the memory will be needed to store the word. So, first of all if the memory does not contain the destinationValue + offset (memory address for storing), then it will be added along with the least significant 8 bits of the word. If the memory

address exists, then the program will overwrite the previous value with the least significant 8 bits. This procedure is repeated three more times with each time the memory address to store in increases by one and the value stored in the memory space is the next 8 bits.

7. `StoreHalfWord()`: this function simply calls `getPartsforStoring()` function, then gets the source value from the `unordered_map <string, long long> registers` by searching for the value of the source register. Then bitset is used to change the sourceValue into binary instead of decimal. The destination value is also retrieved from the `unordered_map <string, long long> registers`. As this is a half word (16 bits) and the memory is byte addressable, 2 places in the memory will be needed to store the word. So, first of all if the memory does not contain the destinationValue + offset (memory address for storing), then it will be added along with the least significant 8 bits of the half word. If the memory address exists, then the program will overwrite the previous value with the least significant 8 bits. This procedure is repeated one more time with the memory address to store in increases by one and the value stored in the memory space is the next 8 bits.
8. `StoreByte()`: this function simply calls getPartsforStoring function, then gets the source value from the `unordered_map <string, long long> registers` by searching for the value of the source register. Then bitset is used to change the sourceValue into binary instead of decimal. The destination value is also retrieved from the `unordered_map <string, long long> registers` As this is a Byte (8 bits) and the memory is byte addressable, one place in the memory will be needed to store the word. So, first of all if the memory does not contain the destinationValue + offset

(memory address for storing), then it will be added along with the byte from the sourceValue. If the memory address exists, then the program will overwrite the previous value with the byte in sourceValue.

9. `loadUpperImmidate()`: this function simply gets the instruction line from the instructions unordered map, then breaks down the instruction line into destination register and immediate, then equates the destination register value in the unordered map to the upper 20 bits of the immediate using bitExtracted function.
10. `add()`: The function gets the instruction line from the `unordered_map <string, long long> instructions`, then it calls getPartsforRegtoReg to get the rs1, rs2, rd names. Then, it gets the source1 and source2 values from the `unordered_map <int, string> registers`. It computes the destination value by adding the source1 value to the source2 value. Lastly, it stores the destination value in the destination register.
11. `addImmediate()`: The function gets the instruction line from the instructions unordered map, then it calls `getPartsforRegtoReg_Imm()` to get the `rs1` and `rd` names. It also gets the `immediate` as a string. Then, it gets the source1 value from the `unordered_map <string, long long> registers`. It converts the immediate from string to long long data type. It computes the destination value by adding the source1 value to the immediate value. Lastly, it stores the destination value in the destination register.
12. `sub()`: The function gets the instruction line from the `unordered_map <string, long long> instructions`, then it calls `getPartsforRegtoReg()` to get the `rs1, rs2, rd` names. Then, it gets the source1 and source2 values from the registers

unordered map. It computes the destination value by subtracting the source2 value from the source1 value. Lastly, it stores the destination value in the destination register.

13. `and()`: The function gets the instruction line from the `unordered_map <string, long long> instructions`, then it calls `getPartsforRegtoReg()` to get the `rs1, rs2, rd` names. Then, it gets the source1 and source2 values from the registers unordered map. It computes the destination value by performing a bitwise and operation between the source1 and source2 values. Lastly, it stores the destination value in the destination register.
14. `andImmediate()`: The function gets the instruction line from the instructions unordered map, then it calls `getPartsforRegtoReg_Imm` to get the `rs1` and `rd` names. It also gets the immediate as a string. Then, it gets the source1 value from the registers unordered map. It converts the immediate from string to long long data type. It computes the destination value by performing a bitwise and operation between the source1 and the immediate values. Lastly, it stores the destination value in the destination register.
15. `or():` The function gets the instruction line from the instructions unordered map, then it calls `getPartsforRegtoReg` to get the `rs1, rs2, rd` names. Then, it gets the source1 and source2 values from the registers unordered map. It computes the destination value by performing a bitwise or operation between the source1 and source2 values. Lastly, it stores the destination value in the destination register.
16. `orImmediate()`: The function gets the instruction line from the instructions unordered map, then it calls `getPartsforRegtoReg_Imm()` to get the rs1 and rd names. It also gets the immediate as a string. Then, it gets the source1 value from the registers

unordered map. It converts the immediate from string to long long data type. It computes the destination value by performing a bitwise or operation between the source1 and the immediate values. Lastly, it stores the destination value in the destination register.

17. `xor()`: The function gets the instruction line from the instructions unordered map, then it calls getPartsforRegtoReg to get the `rs1, rs2, rd` names. Then, it gets the source1 and source2 values from the registers unordered map. It computes the destination value by performing a bitwise xor operation between the source1 and source2 values. Lastly, it stores the destination value in the destination register.
18. `xorImmediate()`: The function gets the instruction line from the instructions unordered map, then it calls `getPartsforRegtoReg_Imm()` to get the `rs1` and `rd` names. It also gets the immediate as a string. Then, it gets the source1 value from the registers unordered map. It converts the immediate from string to long long data type. It computes the destination value by performing a bitwise xor operation between the source1 and the immediate values. Lastly, it stores the destination value in the destination register.
19. `shiftLeftLogical()`: The function gets the instruction line from the instructions unordered map, then it calls getPartsforRegtoReg to get the `rs1, rs2, rd` names. Then, it gets the source1 and source2 values from the registers unordered map. It computes the destination value by performing a shift left logical operation on the source1 value with steps equal to the source2 value. Lastly, it stores the destination value in the destination register.
20. `shiftLeftLogicalImmediate()`: The function gets the instruction line from the instructions unordered_map, then it calls `getPartsforRegtoReg()` to get the rs1 rd

names. Then, it gets the source1 and immediate values from the registers unordered_map. It computes the destination value by performing a shift left logical operation on the source1 value with steps equal to the immediate value. Lastly, it stores the destination value in the destination register.

21. `shiftRightLogical()`: The function gets the instruction line from the instructions unordered map, then it calls getPartsforRegtoReg to get the `rs1, rs2, rd` names. Then, it gets the source1 and source2 values from the registers unordered map. It computes the destination value by performing a shift right logical operation on the source1 value with steps equal to the source2 value. Lastly, it stores the destination value in the destination register.
22. `shiftRightLogicalImmediate()`: The function gets the instruction line from the instructions unordered map, then it calls `getPartsforRegtoReg()` to get the rs1 rd names. Then, it gets the source1 and immediate values from the registers unordered map. It computes the destination value by performing a shift right logical operation on the source1 value with steps equal to the immediate value. Lastly, it stores the destination value in the destination register.
23. `shiftRightAri()`: The function gets the instruction line from the instructions unordered map, then it calls `getPartsforRegtoReg()` to get the rs1, rs2, rd names. Then, it gets the source1 and source2 values from the registers unordered map. It computes the destination value by performing a shift right arithmetic operation on the source1 value with steps equal to the source2 value. Lastly, it stores the destination value in the destination register.

24. `shiftRightAriImmediate()`: The function gets the instruction line from the instructions unordered map, then it calls `getPartsforRegtoReg()` to get the rs1 rd names. Then, it gets the source1 and immediate values from the registers unordered map. It computes the destination value by performing a shift right arithmetic operation on the source1 value with steps equal to the immediate value. Lastly, it stores the destination value in the destination register.
25. `setLessThan()`: The function gets the instruction line from the instructions unordered map, then it calls getPartsforRegtoReg to get the rs1, rs2, rd names. Then, it gets the source1 and source2 values from the registers unordered map. It computes the destination value by comparing the value of source1 to that of source2; if source1 value is less than source2 value, then destination value would be set to 1. Otherwise, it would be set to 0. Lastly, it stores the destination value in the destination register.

26. `setLessThanImm()`: The function gets the instruction line from the instructions unordered map, then it calls getPartsforRegtoReg to get the `rs1, rs2, rd` names. Then, it gets the source1 and immediate values from the registers unordered map. It computes the destination value by comparing the value of source1 to that of immediate; if source1 value is less than the immediate value, then destination value would be set to 1. Otherwise, it would be set to 0. Lastly, it stores the destination value in the destination register.
27. `setLessThanUnsigned()`: The function gets the instruction line from the instructions unordered map, then it calls `getPartsforRegtoReg()` to get the rs1, rs2, rd names. Then, it gets the source1 and source2 values from the registers unordered map. It converts

the values of rs1 and rs2 from signed to unsigned ones for the sake of performing an unsigned comparison. It computes the destination value by comparing the value of source1 to that of source2; if source1 value is less than source2 value, then destination value would be set to 1. Otherwise, it would be set to 0. Lastly, it stores the destination value in the destination register.

28. `setLessThanUnsignedImm()`: The function gets the instruction line from the instructions unordered map, then it calls `getPartsforRegtoReg()` to get the rs1 and rd names. Then, it gets the source1 and immediate values from the registers unordered map. It converts the values of rs1 and immediate from signed to unsigned ones for the sake of performing an unsigned comparison. It computes the destination value by comparing the value of source1 to that of immediate; if source1 value is less than immediate value, then destination value would be set to 1. Otherwise, it would be set to 0. Lastly, it stores the destination value in the destination register.
29. `addUpperImmediatetoPC()`: takes the corresponding instruction from the unordered_map and identifies rd and the immediate value. It then stores the sum of the program counter along with the extracted value of the immediate value. The bit extraction occurs in another function that converts the decimal value into binary, extracts the 20 bits from the immediate value, and proceeds to return it to the addUpperImmediatePC();

30. `jumpandlink()`: this function also takes the instruction from the unordered_map, and locates the destination register, label. Afterwards, the program stores the next instruction's program counter into rd for future retrieval, and updates the program counter

with that of the upcoming label's program counter. A for loop is constructed to proceed with further instructions.

31. `Jumpandlinkregister()`: this function takes once again the instruction from the unordered_map, and locates the destination register, the offset, and the base address. Afterwards, the program stores the next instruction's program counter into rd for future retrieval, and updates the program counter with that of the base address plus the offset. A for loop is constructed to proceed with further instructions.

32. `branchEqual()`: after taking the instruction and determining the corresponding registers and label, the program checks if the registers contain the same value if so the pc is reinitialized to that of the labels location and a for loop is constructed to iterate over the rest of the program.
33. `branchNotEqual()`: after taking the instruction and determining the corresponding registers and label, the program checks if the registers do not contain the same value if so the pc is reinitialized to that of the labels location and a for loop is constructed to iterate over the rest of the program.
34. `branchLessThan()`: after taking the instruction and determining the corresponding registers and label, the program checks if the first register is less than the second register if so the pc is reinitialized to that of the labels location and a for loop is constructed to iterate over the rest of the program.
35. `branchLessThanUnsigned()`: after taking the instruction and determining the corresponding registers and label, the program removes the sign of values residing inside the registers and checks if the first register is less than the second register, if so the pc is

reinitialized to that of the labels location and a for loop is constructed to iterate over the rest of the program.

36. `branchGreaterThanOrEqual()`: after taking the instruction and determining the corresponding registers and label, the program checks if the first register is greater than or equal to the the second register, if so the pc is reinitialized to that of the labels location and a for loop is constructed to iterate over the rest of the program.
37. `branchGreaterThanorEqualUnsigned()`:after taking the instruction and determining the corresponding registers and label, the program removes the sign of values residing inside the registers and checks if the first register is greater than or equal to the the second register, if so the pc is reinitialized to that of the labels location and a for loop is constructed to iterate over the rest of the program.

Assumptions made:

1. We made an assumption that we will store in the register data structure decimal values and in the memory we stored binary values and worked accordingly.
2. Memory is byte addressable

Any Bugs or issues in the simulator:

No bugs in the simulator.

User guide:

How to compile and run:

1. Change the directory in line 115 for assembly code file

   Line 165 for register file that contains initial values

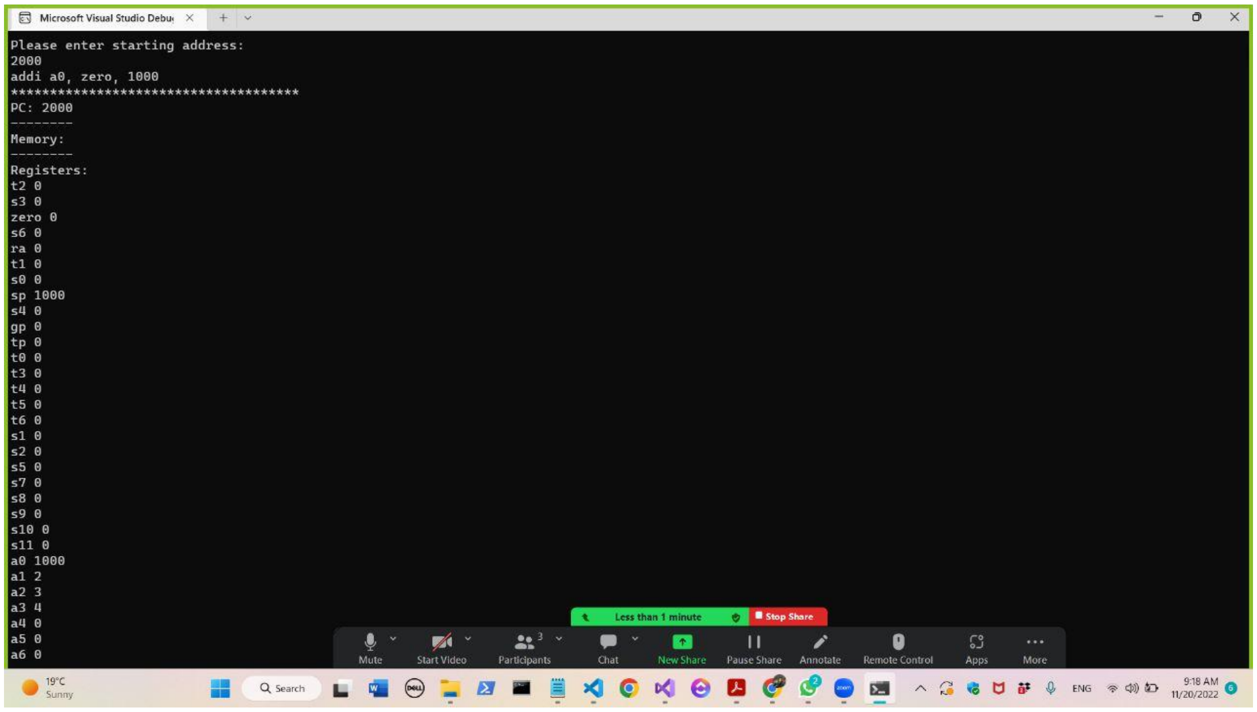
   Line 189 for program data file

2. The user is required to enter the starting address. Then the program will go through the assembly code and change in the memory and register value and output them after each instruction.

Screenshots of a simulation:

<u>Note: To make it more readable we have attached an output file that contains the simulation example (it shows the output after each instruction)</u>

Find below a snippet of the output, all the output is in the file attached in the submission.



```
Please enter starting address:
2000
addi a0, zero, 1000
*************************************
PC: 2000
--------
Memory:
--------
Registers:
t2 0
s3 0
zero 0
s6 0
ra 0
t1 0
s0 0
sp 1000
s4 0
gp 0
tp 0
t0 0
t3 0
t4 0
t5 0
t6 0
s1 0
s2 0
s5 0
s7 0
s8 0
s9 0
s10 0
s11 0
a0 1000
a1 2
a2 3
a3 4
a4 0
a5 0
a6 0
```

Programs simulated:

1. Addition and subtraction in another function
2. Find maximum integer in an array
3. Auipc and lui: In Test Case, please note that you need to start from address 1000 so that you can allocate the data stored in the memory.
4. The logical operators `AND, OR, XOR, SLL, SRL, SRA`.

5. The logical operators `SLT, SLTI.`
6. Stores byte, half word, retrieves them unsigned then compares, if not equal then `XOR`, if equal then `OR`.