Lab 1: Single Cycle RISC-V

In this Lab assignment, you will implement an instruction-level simulator for a single cycle RISC-V processor in C++ step by step. The simulator supports a subset of the RISC-V instruction set and can model the execution of each instruction.

A RISC-V program you are expected to write is provided for the simulator as a text file "imem.txt" file, which is used to initialize the Instruction Memory. Each line of the file corresponds to a Byte stored in the Instruction Memory in binary format, with the first line at address 0, the next line at address 1 and so on. Four contiguous lines correspond to a whole instruction. Note that the words stored in memory are in "Big-Endian" format, meaning that the most significant byte is stored first.

We have defined a "halt" instruction as 32'b1 (0xFFFFFFFF), which is the last instruction in every "imem.txt" file. As the name suggests, when this instruction is fetched, the simulation is terminated. You are encouraged to generate "imem.txt" files to test your simulator.

The Data Memory is initialized using the "dmem.txt" file. The format of the stored words is the same as the Instruction Memory. As with the instruction memory, the data memory addresses also begin at 0 and increment by one in each line.

The instructions that the simulator supports and their encodings are shown in Table 1. Note that all instructions, except for "halt", exist in the RISC-V ISA. The *RISC-V Instruction Set Manual* in **Assignments** defines the semantics of each instruction (page 130).

Name	Format Type	Opcode (Binary)	Func 3(Binary)	Func 7(Binary)
add	R-Type	0110011	000	0000000
sub	R-Type	0110011	000	0100000
addi	I-Type	0010011	000	
and	R-Type	0110011	111	0000000
or	R-Type	0110011	110	0000000
xor	R-Type	0110011	100	0000000
beq	SB-Type	1100011	000	
jal	UJ-Type	1101111		
ld	I-Type	0000011	011	
sd	S-Type	0100011	011	

Table 1. Instruction encodings for a reduced RISC-V ISA

Skeleton Code

The file "RISC-V.cpp" contains a skeleton code for the assignment. In this section, we provide descriptions for each of the components in the skeleton code. You are required to *Fill in* the missing code (**\\ TODO: implement!**).

Classes

We have defined four C++ classes that each implement one of the four major blocks in a single cycle RISC-V, namely RF (to implement the register file), ALU (to implement the ALU), INSMem (to implement instruction memory), and DataMem (to implement data memory).

1. **RF class:** contains 32 64-bit registers defined as a private member. Remember that register $0 is always 0. Your job is to implement the implement the *ReadWrite()* member function that provides read and write access to the register file.
2. **ALU class:** implements the ALU. Your job is to implement *ALUOperation()* member function that performs the appropriate operation on two 64 bit operands based on ALUOP. See Table 1 for more details.
3. **INSMem class:** a Byte addressable memory that contains instructions. The constructor *InsMem()* initializes the contents of instruction memory from the file imem.txt (you are expected to write a program). Your job is to implement the member function *ReadMemory()* that provides read access to instruction memory. An access to the instruction memory class returns 4 bytes of data; i.e., the byte pointed to by the address and the three subsequent bytes.
4. **DataMem class:** is similar to the instruction memory, except that it provides both read and write access, and an access to data memory class returns 8 bytes of data.

Main Function

The main function defines a 32 bit program counter (PC) that is initialized to zero. The RISC-V simulation routine is carried out within a while loop. In each iteration of the while loop, you will fetch one instruction from the instruction memory, and based on the instruction, make calls to the register file, ALU and data memory classes (in fact, you might need to make two calls to the register file class, once to read and a second time to write back). Finally you will update the PC so as to fetch the next instruction. When the halt instruction is fetched, you are to break out of the while loop and terminate the simulation.

Make sure that the architectural state is updated correctly after execution of each instruction. The architectural state consists of the Program Counter (PC), the Register File (RF) and the Data Memory (DataMem). We will check the correctness of the architectural state after *each* instruction.

Specifically, the OutputRF() function is called at the end of each iteration of the while loop, and will add the new state of the Register File to "RFresult.txt". Therefore, at the end of the program execution "RFresult.txt" contains all the intermediate states of the Register File. Once the program terminates, the OutputDataMem() function will write the final state of the Data Memory to "dmem.txt". These functions have been implemented for you.
(Note: **You should delete the "RFresult.txt" file before re-executing your program**, otherwise the new results will append to the previous results.)

Test case:

Assume that the variables i and j are assigned to registers x28, and x29, respectively. Assume that the base address of the arrays A and B are in registers x10 and x11, respectively. The C code is:
B[1] = A[i−j];
The corresponding RISC-V code is:

```
sub x30, x28, x29  // compute i-j
add x30, x30, x30        // multiply by 8 to convert the double word offset to a byte offset
add x30, x30, x30
add x30, x30, x30
add x10, x10, x30
ld x30, 0(x10)                // load A[i-j]
sd x30, 8(x11)           // store in B[1]
```

What You Have to Submit

1. We have provided skeleton code in the file RISC-V.cpp. Finish the code. Code that does not compile will automatically be given a 0.
2. We have provided "dmem.txt" file of initialized data.
3. You will have to write your own RISC-V programs in "imem.txt" to check your design for different cases, we have provided a testcase for you to verify your code.

Some useful references for this lab:

1. A brief introduction to C++ (https://web.eecs.umich.edu/~sugih/pointers/c++.pdf)
2. A reference for the C++ bitset class (http://www.cplusplus.com/reference/bitset/bitset/)
3. A reference to the C++ string class (http://www.cplusplus.com/reference/string/string/)