1 Problem statement

You have used the RIPES simulator for RISC-V in the first assignment. In this second assignment, you will implement your own RISC-V simulator (just the processor part of it). The processor should have 2 variants, as seen in the following screenshots from RIPES (a) 5 stage pipelined processor with no forwarding (b) 5 stage pipelined processor with forwarding, the five stages being Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory operations (MEM) and WriteBack (WB).

Your implementation needs to be object oriented in C++, without copy pasting any code between the two processor variants. The 32 registers can be a data structure. The control signals, input and output for each pipeline stage can be implemented as a struct, with a separate function to implement the working of the stage. The function for EX stage can have support for some basic ALU operations (to support the examples mentioned in next section). Cycles can be implemented as loop iterations. Branches should be decided after the ID stage.

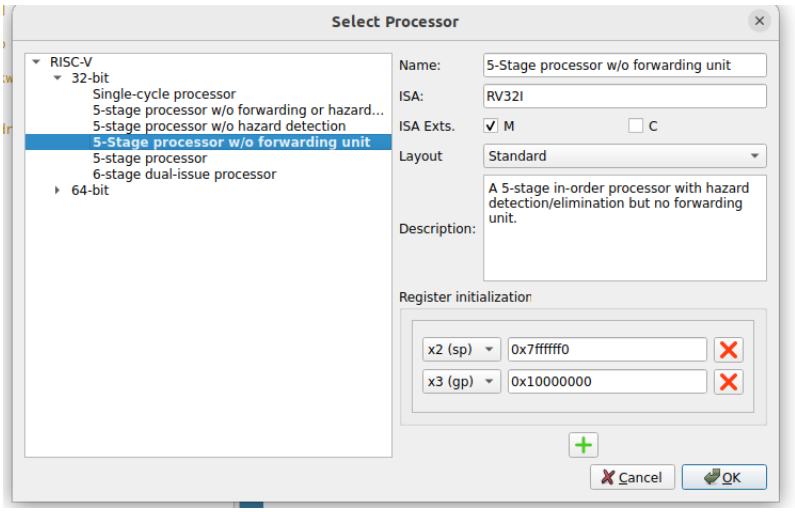

Figure 1: 5 stage pipelined processor with no forwarding

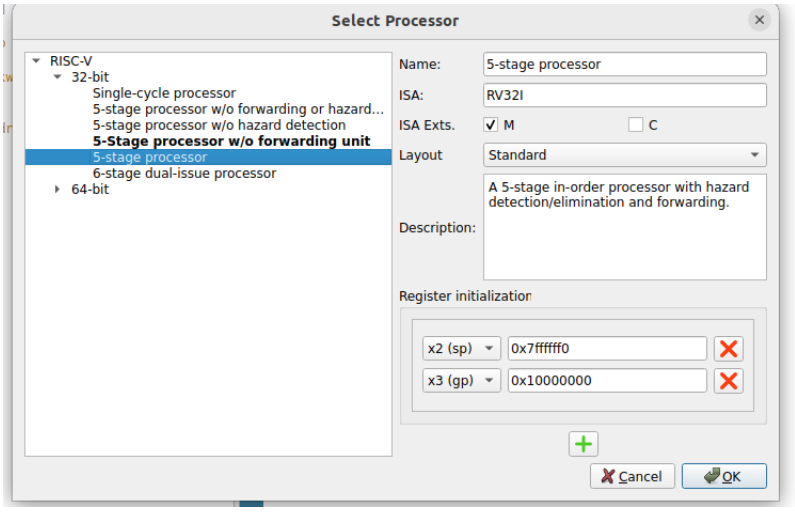

Figure 2: 5 stage pipelined processor with forwarding

2 Getting RISC-V instructions to test your processor

Check https://marz.utk.edu/my-courses/cosc230/book/example-risc-v-assembly-programs/ for example functions given in C++ and the corresponding RISC-V assembly. Import these example functions in assembly into the RIPES simulator, and get the 32 bit machine codes corresponding to each instruction in the assembly programs.

```
strlen:
    # a0 = const char *str
    li      t0, 0           # i = 0
1: # Start of for loop
    add     t1, t0, a0      # Add the byte offset for str[i]
    lb      t1, 0(t1)       # Dereference str[i]
    beqz    t1, 1f          # if str[i] == 0, break for loop
    addi    t0, t0, 1       # Add 1 to our iterator
    j       1b              # Jump back to condition (1 backwards)
1: # End of for loop
    mv      a0, t0          # Move t0 into a0 to return
    ret                     # Return back via the return address register
```

This is the example of the strlen function in RISC-V assembly (including pseudo instructions). The 32bit machine code, with actual non-pseduo instructions from the RIPES simulator are given below.

```
0:          00000293            addi x5 x0 0
4:          00a28333            add x6 x5 x10
8:          00030303            lb x6 0 x6
c:          00030663            beq x6 x0 12
10:         00128293            addi x5 x5 1
14:         ff1ff06f            jal x0 -16
18:         00028513            addi x10 x5 0
1c:         00008067            jalr x0 x1 0
```

The second column, containing the 32bit machine code should be input to your C++ processor simulator. The third column will be useful in printing the pipeline diagram. So save these two columns for different functions in files like strlen.txt, stringcopy.txt, strncpy.txt etc. Your executable should take the filename as commandline input and open the file at the start of the main function. Then IF stage should read a new line of the already opened file (which line to read will depend on branch vs. other instructions), ID stage should decode the 32 bit machine code, EX should do the appropriate ALU operation and so on.

You need to output the pipeline diagram on stdout for appropriate number of cycles. For the above example, the initial part of pipeline diagram for the two processor types are given below.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
addi x5 x0 0	IF	ID	EX	MEM	WB											
add x6 x5 x10		IF	ID	-	-	EX	MEM	WB							IF	ID
lb x6 0 x6			IF	-	-	ID	-	-	EX	MEM	WB					IF
beq x6 x0 12						IF	-	-	ID	-	-	EX	MEM	WB		
addi x5 x5 1									IF	-	-	ID	EX	MEM	WB	
jal x0 -16												IF	ID	EX	MEM	WB
addi x10 x5 0													IF	ID		
jalr x0 x1 0														IF		

	0	1	2	3	4	5	6	7	8	9	10
addi x5 x0 0	IF	ID	EX	MEM	WB						
add x6 x5 x10		IF	ID	EX	MEM	WB				IF	ID
lb x6 0 x6			IF	ID	EX	MEM	WB				IF
beq x6 x0 12				IF	ID	-	EX	MEM	WB		
addi x5 x5 1					IF	-	ID	EX	MEM	WB	
jal x0 -16							IF	ID	EX	MEM	WB
addi x10 x5 0								IF	ID		
jalr x0 x1 0									IF		

Figure 3: Pipeline diagram without and with forwarding

Use ; as a delimiter between cycles within a line. Thus output corresponding to the first two rows of the nonforwarding processor will look as follows:

addi x5 x0 0;IF;ID;EX;MEM;WB
add x6 x5 x10; ;IF;ID;-;-;EX;MEM;WB

3 Submission: Follow formatting instructions to avoid penalty

Try to get as many RISC-V assembly functions working on your processor simulator as possible from https://marz.utk.edu/my-courses/cosc230/book/example-risc-v-assembly-programs/. Increasing supported examples might mean extending your decode function to understand more and more machine codes, increasing your support for new ALU operations, new control signals, and so on. Your code should be modular, so that you can gradually add support to run more example programs.

Create a **private git repo**, from the start of working on this assignment, and both assignment partners should commit all code changes regularly (not all together at the end). Your last git commit should be before submission deadline. How you have gradually built/debugged/fixed your implementation will have some credits, in addition to the final running.

The files and subfolders in the git repo should be as follows.

1. **src/** should contain your .hpp files, .cpp files and a Makefile. In the src/ folder, `$make` should create executables `noforward` and `forward` for your two processor types respectively.

2. **inputfiles/** should contain the specific strlen.txt, stringcopy.txt, strncpy.txt etc. that your processor implementation supports.

3. **outputfiles/** should contain the stdout redirected to a file of the same name as the input file, with extensions _noforward_out.txt and _forward_out.txt respectively. You should submit the outputfiles/ folder with the outputs already generated when you run the examples.

4. **readme.md** documenting design decisions you took, known issues in your implementation, sources (including chatgpt) you have consulted and how.

We should be able to run the executables in the src/ folder from command line as `$./noforward ../inputfiles/filename.txt cyclecount` to generate the pipeline diagram on stdout for your nonforwarding processor, for that particular filename, for those number of cyclecounts. Similarly, `$./forward ../inputfiles/filename.txt cyclecount` should generate pipeline diagram on stdout for the forwarding processor, for that particular filename, for those number of cyclecounts.

At assignment deadline, download the zip from your git repo and submit on Moodle, renaming it as entrynum1_entrynum2.zip. The internal subfolder structure should be exactly the same for the zip submitted on Moodle, as specified above for the git repo. During demo, the TA will download from Moodle and run. We will also check the commit trail in the git repo to understand how your code gradually progressed over 4 weeks between 2 partners.