Note: Please start this assignment early as it will take more effort than hw1 and hw2.

Objective:

From this homework onwards we will start building the basic blocks of a processor. This assignment will slowly help you to build a small processor which can run a few instructions. It will also help you to get familiar with the tools that you will use for your project.
Before you start please see these guidelines:

1. Please review the Verilog rules for this course (Link) - Understand what you are allowed to use and what is restricted. You are only supposed to do either dataflow or structural modeling and are not allowed to use sequential modeling in this course (i.e. no always blocks with posedge, negedge or clk).
2. Start using wsrun.pl from now onwards. wsrun.pl will help you automate the majority of checking parts in your project. It is highly recommended.

As usual, the general suggestion is to use hierarchical modeling, i.e. break down your module into smaller submodules. Draw the modules by hand to get a picture of what is happening. Build smaller modules and verify them. Reuse as much as possible.

Problem 1:

Design a register file `rf.v`. It will have a total of 8 registers, each of size 16 bits. Following is the I/O:

Inputs:

```
clk - Clock input for register files
rst - Reset signal. Will initialize all the registers as 0
[3:0] read1regsel - 3 bit register selector for read1data
[3:0] read2regsel - 3 bit register selector for read2data
[3:0] writeregsel - 3 bit register selector for writing to that register
[15:0] writedata - data to be written to register
write - write enable signal. If 1 we will write the writedata to
        writeregsel at rising clk end.
```

Outputs:

```
[15:0] read1data - Value of the register corresponding to read1regsel
[15:0] read2data - Value of the register corresponding to read2regsel
err - err signal will assert high when we reach any state which is
      impossible to reach.
```

Additional info:

- The `write` input is a write enable which when high (1'b1) will write the value of writedata to the register specified by writeregsel at the next rising clock edge.

- There is no read enable signal. You will always have `read1data` show the value of the register specified by `read1regsel` and `read2data` show value of the register specified by `read2regsel`.
- The read logic is combinational logic, i.e.whenever `read1regsel` is updated you should see changes in `read1data` irrespective of clock edge (same for `read2data`).
- The `err` output is a standard way of indicating hardware errors or illegal states. Assert it (1'b1) for states which are supposed to be impossible to get into.

First try to make a single 16-bit register. Then combine 8 of those together with logic of reading and writing to make the register file.

The rf_bench.v will check your register file with random inputs. If you are passing all the tests you will see a “TEST PASSED” message in wsrun.pl output (same will be visible in ModelSim Transcript). But if you fail any test case you will see “TEST FAILED” and “ERRORCHECK: …” messages in the output. To help you debug your code, the ERRORCHECK messages will show the inputs provided to the module, your output, and the expected output for a particular cycle. Make sure that you have “TEST PASSED” in your testbench.

What to submit:

```
hw3 ->
    hw3_1 ->
        All Verilog files
        All Verilog rules checking output files
```

Problem 2:

In problem 2 you will be designing a FETCH unit called `fetch.v`. This will be a pretty basic fetch unit and at every clock edge it will fetch the instr present at PC and also update the PC to PC+2. (Note the 2 here not 4, its because the **instruction size in this ISA is only 16 bits** unlike MIPS which is 32-bits)

Inputs:

```
clk - Input clock
rst - Reset will initialize the PC to 0 and trigger other necessary initial
setup
```

Outputs:

```
err - Assert high when you reach any state that is impossible to get into
[15:0] instr - The instruction fetched from the memory
```

Do the following parts step by step:

a) Take a look at the Memory Module Specification Document [Link] & `memory2c.v` module provided in the tar file. Try and understand how it is working. For this problem we will not need signals `data_in, wr and createdump,` so you may assign them as 0.
b) From the given `fetch.asm` file generate a bytecode image using the assembler. You will find the details in Using the Assembler [Link] pdf. Basically you will run need to run the following command to generate the `loadfile_all.img`:

```
assemble.sh fetch.asm
```

c) Once you have built the `loadfile_all.img` it will automatically get loaded in the memory at the first clk edge during rst. Now you can use this in your fetch unit and generate the necessary output. So build the `fetch.v` module.
d) Finally run the provided testbench (`fetch_hier_bench.v`) and ensure that it is passing properly.

If you are passing all the tests in the testbench you will see a “TEST PASSED” message in wsrun.pl output (same will be visible in ModelSim Transcript). But if you fail any test case you will see “TEST FAILED” and “ERROCHECK: …” messages in the output. Make sure that you have “TEST PASSED” in your testbench.

What to submit:

```
hw3 ->
    hw3_1 ->
        ...........
    hw3_2 ->
        All Verilog files
        All Verilog rules checking output files
```

Problem 3:

Now that you have already built a fetch unit, let's try to expand our fetch unit so we can run some basic instructions. In this problem you will build the capability to run 3 instructions only and you will also learn how to run your assembly program with just these 3 instructions and verify the output.

Inputs:

```
clk - Clock input
rst - Reset signal
```

Outputs:

```
err - Assert high when reaching any states impossible to get into
```

Instructions:

Instruction Format	Syntax	Semantics
00000 xxxxxxxxxxx	HALT	Assert high `halt` signal
01000 sss ddd iiiii	ADDI Rd, Rs, immediate	Rd <- Rs + I(sign ext.)
11011 sss ttt ddd 10	XOR Rd, Rs, Rt	Rd <- Rs XOR Rt

Additional Info:

- Your design of `proc.v` must have a `halt` signal at the top-level module and it must be asserted high when you receive the halt signal. For hw3 we are only using HALT to stop the testbench. (HALT will have other additional constraints in your project but for the scope of the assignment 3 we are limiting it to this)
- Note that each register is 16 bits long and ADDI uses a sign extender, so your sign extender will be 16-bits long.
- You will find more details in the WISC ISA Specification document (which will be released with the project design review phase).

a) Use your already existing fetch and register file, and make this processor. It will be called `proc.v`

b) Once you have made the design you will need to **update some names in the testbench module**. It will be marked clearly in the module which names need to be updated. Please update them or else the testbench will not work.

c) Refer to the document "**WISC-SP13 Simulator-debugger**" [Link] to understand what it is. The command below will automatically run both your design and wiscalculator. (Note: It is possible that you may have to do steps from the Troubleshooting section in the document. Please do it if needed)

d) Run the processor using the following command:

```
wsrun.pl -prog proc.asm proc_hier_bench *.v
```

It will do the following:

- Go to the directory which has the Verilog code of all your modules.
- At the prompt, run wsrun.pl with the -prog flag. This will assemble the program, compile your Verilog design, and then run the program on your Verilog design. A "trace" of changes to the architecture state for each instruction's execution is written to a file called verilogsim.trace.
- Next the command will run the wiscalculator and generate its trace as archsim.trace.
- Lastly it will compare both the traces and if they are same you will see success in summary.log file

e) Verify that the **summary.log** output shows **success**.

NOTE: Please don't modify the existing proc.asm. If you want to try and run different assembly

code with your design, copy proc.asm into another file and edit that. While submitting, please submit original proc.asm only.

What to submit:

```
hw3 ->
    hw3_1 ->
        ............
    hw3_2 ->
        ............
    hw3_3 ->
        All Verilog files
        All Verilog rules checking output files
```

[Bonus] Problem 4:

For the bonus part, let's try making the processor more interesting. You will be making `proc_beqz_added.v` where you will add one more instruction which is BEQZ as written below.

Instruction Format	Syntax	Semantics
01100 sss iiiiiiii	BEQZ Rs, immediate	if (Rs == 0) then PC <- PC + 2 + I(sign ext.)

Inputs:

```
clk - Clock input
rst - Reset signal
```

Outputs:

```
err - Assert high when reaching any states impossible to get into
```

a) Design the new module with capabilities to run BEQZ.
b) Prepare your own asm code with 4 instructions (ADDI, XOR, HALT and BEQZ) to verify your own code. Name it `proc_beqz_added.asm`. For problem 4 we are not providing the asm file and you should verify it yourself by writing your own code. (TAs will also check with their own asm file during grading so make sure to be thorough). (See `fetch.asm` from problem 3 to see how to write the asm code)
c) Verify that your design is working with `proc_beqz_added.asm` file using the command and steps as mentioned above: (Note testbench file names are same but we have change the instance name so it should not be an issue)

```
wsrun.pl -prog proc_beqz_added.asm proc_hier_bench *.v
```

d) Verify that everything is working fine according to steps mentioned in problem 3.

What to submit:

```
hw3 ->
    hw3_1 ->
        ............
    hw3_2 ->
        ............
    hw3_3 ->
        ............
    hw3_4 ->
        All Verilog files
        All Verilog rules checking output files
```

Final Submission:

- Tar the `hw3` folder which will contain 4 (or 3) subdirectories, i.e. `hw3_1, hw3_2, hw3_3 (and hw3_4 if you attempted bonus question).`
- Upload the `hw3.tar` file on canvas to submit the assignment.