It is highly recommended to start the assignment early!

Objective:

The main objective of this assignment is to help you become familiar with CSL Machines, Modelsim, and Verilog HDL. It will help you become comfortable with the tools that you will use for the rest of the semester.

Before you start:

Please go through the following documents before starting the assignment. You can find all these documents in canvas under Files Tabs, they will be either in “Verilog Tools, Scripts and Guides” directory or “Discussion Section” directory:

1. Work Environment Setup CS552 (Link)
2. Verilog Slides (Link); Verilog Cheatsheet (Link)
3. Modelsim Tutorial (Link); Command-Line Verilog simulation tutorial (Link)
4. Verilog rules for this course (Link) & rule checking script (Link)
5. Verilog naming convention for this course (Link)

In this assignment, you will design digital components and simulate them to verify their correctness. This assignment will show you how to build hierarchically, i.e. breaking down a large design into smaller blocks, which can be composed together to build the larger system. For e.g. you can design a 2-to-1 mux from simple gates, and then use those 2-to-1 muxes to design a 4-to-1 mux. This “divide and conquer” or hierarchical design approach is very helpful when designing complex designs. (which you will experience to some extent with the course project and future assignments)
The assignment will take significant time if you are new to Modelsim and Hardware Description Languages. So start early!

Problem-1: (20 Points)

1. Design a 1-bit 1-to-2 demultiplexer module called `demux1_2` using only NAND, NOR, and NOT gates.

- Basic NAND, NOR, and NOT gate modules are provided with the homework files. Instantiate them in your solution to implement the circuit.
- Following is the IO:
  - Inputs: `Inp` (Input data), `S` (Control input)
  - Outputs: `OutA, OutB`
- Truth Table for Output:

S	OutA	OutB
0	Inp	0
1	0	Inp

2. Use the 1-to-2 demux designed in part 1.) to hierarchically design a 1-to-4 demux. Name the module `demux1_4`.

- Use the module `demux1_2` to create a 1-to-4 demux.
- Following the IO:
  - Inputs: `Inp (Input Data), S(1:0) (Control input)`
  - Outputs: `OutA, OutB, OutC, OutD`
- Truth Table for Output:

S	OutA	OutB	OutC	OutD
00	Inp	0	0	0
01	0	Inp	0	0
10	0	0	Inp	0
11	0	0	0	Inp

3. Hierarchically using `demux1_4` create a quad 1-to-4 demux called `quaddemux1_4`.

- Following will be the IO:
  - Inputs: I`np(3:0) (a 4 bit input bus), S(1:0) (Control Input)`
  - Outputs: `OutA(3:0), OutB(3:0), OutC(3:0), OutD(3:0)`
- Truth Table for Output:

S	OutA	OutB	OutC	OutD
00	Inp	0000	0000	0000
01	0000	Inp	0000	0000
10	0000	0000	Inp	0000
11	0000	0000	0000	Inp

4. Use the testbench provided for testing the solution.

What to submit:

1. Make directory `hw1` and inside it make another directory `hw1_1`.
2. Submit all the Verilog files in `hw1_1`.
3. Also run the Verilog rules checking script and submit the results in `hw1_1`.

```
hw1 ->
    hw1_1 ->
        All Verilog files
        All Verilog rules checking output files
```

Problem-2: (40 Points)

1. Design a 1-bit full adder called `fulladder1`, using only NOT, NAND, NOR & XOR gates
   - Following is the IO:
     - Inputs: `A, B, Cin (Carry-in)`
     - Outputs: `S (Sum), Cout (Carry-out)`

2. [Optional] Create your own testbench (using the provided testbench as a guide) for verifying the `fulladder1` design. For creating this testbench, you are allowed to use prohibited Verilog keywords as it will not be possible otherwise.
   - No need to submit the testbench, it is for your own practice.
   - It is a good idea to practice making testbench early as you may require this in future assignments or the course project.

3. Using 1-bit full adder design a 4-bit ripple carry adder named `fulladder4`.
   - Following is the IO:
     - Inputs: `A(3:0), B(3:0), Cin (Carry-in)`
     - Outputs: `S(3:0) (Sum), Cout (Carry-out)`

4. Using the 4-bit ripple carry adder, i.e. `fulladder4`, design a ripple carry adder that will add 2 16-bit binary numbers. Name the module `fulladder16`.
   - Following is the IO:
     - Inputs: `A(15:0), B(15:0)`
     - Outputs: `S(15:0), Cout (Carry-out)`
   - Note: `fulladder16` does not have `Cin (Carry-in)` as the input.

5. Run the provided testbench and verify the correctness.

6. In the above-designed ripple carry adder we have used a series of 1-bit full adders to generate the sum of two 16-bit binary numbers. In terms of timing, how will this adder perform if we scale it up from 16-bit to a very large number of bits? What will be the potential complications? (Hint: Think in terms of delay)
   - Submit your answer along with Verilog files as `hw1_2-answer.txt`.

What to submit:

1. Under the same directory `hw1`,make another directory `hw1_2`.
2. Submit all the Verilog files in `hw1_2`. Submit `hw1_2-answer.txt` in the same directory.
3. Also run the Verilog rules checking script and submit the results in `hw1_2`.

```
hw1 ->
    hw1_1 ->
        ............
    hw1_2 ->
        All Verilog files & hw1_2-answer.txt
```

```
All Verilog rules checking output files
```

Problem-3: (40 Points)

A sequence detector outputs “1” when a particular sequence is detected & outputs “0” otherwise. E.g. assume we have a sequence detector that detects the sequence “1010”. For the input stream “101010” the output sequence will be “0000101”.

In this problem, you will design a sequence detector that will detect a specific 2 digits represents as Binary Coded Decimal (BCD). Your design should `Reset` signal to initialize the registers in proper condition. The sequence detector will sample one bit input `Inp` at every rising edge of the clock signal `Clk`. Select the sequence to detect from the table based on the last digit (least significant digit) of your UW ID number. Detector should also detect overlapping sequences.

Note: You are expected to implement a Moore Machine (and not a Mealy Machine) (reference)

Last Digit of UW ID Number	Sequence
0 or 5	26
1 or 6	52
2 or 7	45
3 or 8	93
4 or 9	78

E.g. If your UW ID number is 9023754183, then your sequence number will be 93 and the sequence to detect will be 10010011.

You should complete each of the following:

- Draw a state diagram (Moore Machine) on paper. You are not required to submit it, but this will help you.
- Implement the sequence detector using Verilog. Call it `seqdec_<your-2-digits>`. E.g. for the user with sequence number 93 the module name will be `seqdec_93`
    - Inputs: `Inp (Next bit in input stream), Clk (Clock signal), Reset`
    - Outputs: `Out`
    - Assume your detector has seem a stream of zeros in the beginning
- Simulate your design with the provided testbench corresponding to your sequence.

What to submit:

4. Under the same directory `hw1`, make another directory `hw1_3`.
5. Submit all the Verilog files in `hw1_3`.
6. Also, run the Verilog rules checking script and submit the results in `hw1_3`.

```
hw1 ->
    hw1_1 ->
        ...........
    hw1_2 ->
        ...........
    hw1_3 ->
        All Verilog files
        All Verilog rules checking output files
```

Final Submission:

- Tar the `hw1` folder which will contain 3 subdirectories, i.e. `hw1_1, hw1_2 and hw1_3`.
- Submit `hw1.tar` file only.