

# Project #4: Single-Cycle Processor

ECE 552: Fall 2025

## Project Introduction

In this project phase, you will create a **fully functional single-cycle processor implementing the RISC-V RV32I ISA**. You will also generate a synthesized netlist of this processor and complete post-synthesis verification.

Implementations must be created **using Verilog; SystemVerilog will not be allowed and will result in a grade of 0 for Problem 2 if used**. This assignment is designed to be completed **in a group of 2-4**. Collaboration between groups is not permitted; we will perform code comparisons between groups.

You are **permitted** to use generative artificial intelligence (e.g. ChatGPT); if you choose to do so, please indicate that in your submission. Please ensure your usage of LLMs adheres to the course policies. For this assignment, however, **LLMs will not be helpful and overly relying on them may make this project phase and subsequent phases significantly harder to complete**.





## Problem 1: Schematic

**Before writing a single line of code**, you must complete a schematic representing the data and control paths of your processor. You must use graphic design software, e.g. Figma (web-based), MS paint (convenient for windows machines), draw.io (web-based), inkscape (available on CAE Linux machines), etc., to create this schematic. It is recommended to create your schematic in the following order:

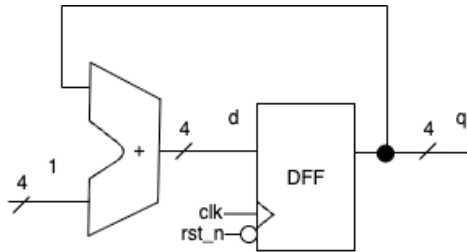
1. Full **data path** of processor; including but not limited to:
  - a. Functional modules such as the arithmetic logic unit (ALU), register file, instruction decoder, program counter (PC) register, etc.
  - b. Multiplexers for selection logic for ALU operands, write back data, register selection, etc.
  - c. Any other logic required (CLAs, zero/sign extenders, etc.)
2. Full **control path** of processor (outputs from instruction decoder); **including but not limited to:**
  - a. ALU signal operand logic
  - b. Memory address selection
  - c. write back selection
  - d. Branch handling

**NOTE: NOT ALL THE COMPONENTS REQUIRED HAVE BEEN DESCRIBED; FILL IN YOUR SCHEMATIC WITH THE DETAILS OF YOUR IMPLEMENTATION. ALSO, YOU ARE NOT REQUIRED TO DRAW BOUNDARIES BETWEEN PIPELINE STAGES BUT IT IS RECOMMENDED.**

Additional requirements:

1. **Show the bit-width of each signal in your processor** (e.g. instructions fetched produce a 32-bit ([31:0]) signal).
2. **Differentiate between combinational (stateless) and sequential (flip-flop) logic by drawing a small triangle** at the bottom of any sequential logic modules (PC, RF, etc.)
3. For each instruction type, **trace the execution path of a sample instruction in a bright color**. For example, for R-type instructions, which modules are required?

## 1.1 Example Schematic of 4-bit Counter



For assistance completing this problem, please consult the textbook. Your schematic should resemble the one from the textbook, but you must determine which logic and control signals are required for your implementation. The simulator described in Problem 2 may also help you with this problem.

Your final schematic must reflect any changes made when implementing your processor in Verilog in Problem 2.

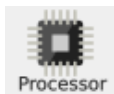
## Problem 2: Processor Implementation

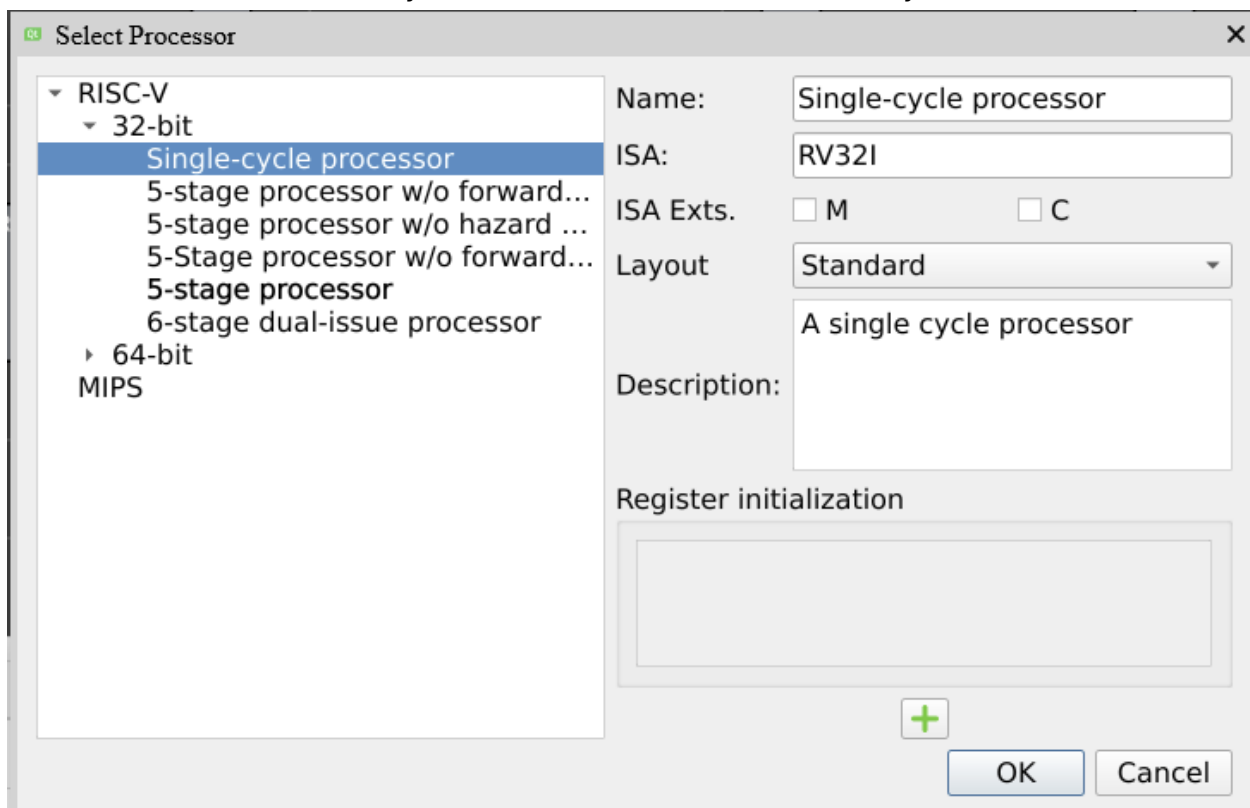
Please implement a single-cycle processor that implements the RISC-V RV32I ISA. You will be graded proportionally to the number of tests that you pass.

Here are some resources you may find helpful:

- The [WISC-F25 specification document](#) is the primary document to reference how instructions are implemented.
- The cheat sheet is helpful when writing programs or as a quick reference. You may find [this simulator](#) helpful for visualizing how your processor should execute and verifying functional correctness and performance.
- We also recommend [this decoder](#) to assemble RISC-V instructions to test and examine instructions when debugging.



Select the  icon. Make sure that you have set the processor to be the single-cycle processor, and that all extensions are disabled. You will also want to disable register initializations. You may also want to select the Extended layout.





To edit the source code to be run by the simulator, select the  icon.

The simulator will also provide information on the processor's performance. This will become more relevant in subsequent project phases.

## 2.1 Local Testing of Your Processor

To run code on your own processor, set your directory up as follows:

tb/                   -> testbench directory; where program.mem is written to  
rtl/                  -> your implementation code (all in same directory, no subdirectories)  
tests/asm/         -> assembly tests with .asm extension go here  
traces/             -> trace output will be placed here

Take a look at RV32I programs under test/asm directory. We use the ebreak instruction to halt the processor. You will also **implement the ebreak instruction** in this project. Not implementing it will cause the testbench to enter an infinite loop. See the [updated specification](#) on Canvas for more details.

To assemble a RV32I program, pull (or update) the docker container:  
**docker pull coderkalyan/ece552-tools:latest**  
**cd tests/asm/**

**Edit the Makefile in the tests/asm/ directory to remove the .txt extension.**

Run the following command to assemble the program 01add.asm under test/asm/ directory. Change 01add to the name of the program you'd like to assemble.

```
make PROGRAM=01add
```

Then run your **tb.v** and inspect the trace generated in traces/ directory to make sure your processor works as expected. Repeat the process for all the tests. In addition to our provided tests, you are also encouraged to write your own tests to test as exhaustively as possible.

**NOTE: I recommend dividing your cpu into modules by pipeline stage; this may make subsequent projects easier but is not required for this project.**

## Problem 3: Synthesis and Post-Synthesis Verification

This problem must be completed on a **Linux CAE Machine**.

### 3.1 Synthesis

#### Step 1: Synthesis Setup

1. Create a new directory for synthesis using the following command:  
`mkdir proj4_synthesis`
2. Navigate to the newly created directory: `cd proj4_synthesis`
3. Inside the `proj4_synthesis` directory, **add the following files:**
  - All the Verilog files for your project
  - `script_syn.tcl`
  - `.synopsys_dc_setup`
4. Inside the **`script_syn.tcl`** file, update the following lines to match the Verilog files included in your design.
  - The `read_file` command should list **all** Verilog source files used in your project.
  - The `set current_design` command should specify the **top-level module** of your design.

```
read_file -format verilog {hart.v decode.v alu.v rf.v} # or  
whatever your files are named
```

```
set current_design hart
```

Make sure to replace the file names and module name with those corresponding to your own design if they differ.

5. If your **top-level module** does not use the signal names `i_clk` for the clock or `i_rst` for the reset, you will need to modify the `script_syn.tcl` file accordingly.
  - Wherever the signal `i_clk` is referenced, replace it with the actual clock signal name used in your design.
  - Similarly, replace `i_rst` with your design's reset signal name.

## Step 2: Running Synthesis

1. Open the Synopsys Design Compiler by typing: `dc_shell`
2. Once inside the DC shell, execute the synthesis script using the command: `source script_syn.tcl`
3. During synthesis, the console may pause and display the prompt --More-- when showing warnings or long outputs.  
If this occurs, press **Enter** repeatedly until the synthesis process continues.

## Step 3: Post-Synthesis Reports and Outputs

1. After synthesis is complete, timing, area, and power reports will be generated and stored in the **reports/** folder.
  - a. Check the following report files:
    - i. `dut_min_delay_syn.txt`
    - ii. `dut_max_delay_syn.txt`
  - b. In both files, verify that the **Slack** value shows “**(MET)**”.  
If it shows “**(VIOLATED)**”, it indicates timing violations.
2. **Resolving Timing Violations:**
  - o Timing violations can usually be resolved by increasing the clock period in the synthesis script.
  - Open the `script_syn.tcl` file and locate the following line:  
`create_clock -name "clk" -period 7 -waveform {0 1.25} $clk_port`
  - Increase the clock period value (for example, change 7 to 7.5) and rerun the synthesis.
  - Repeat this process until both reports show **Slack (MET)**.
2. **Generated Netlist:**
  - o The synthesized netlist file (.vg) will be located in the **outputs/** folder.
  - o This netlist file will be used for **post-synthesis verification** (section 3.2).



## 3.2 Post-Synthesis Verification

### Step 1: Setup

1. Exit the proj4\_synthesis directory and create a new directory for post-synthesis verification:

```
mkdir proj4_post_synthesis
```

2. Navigate to the newly created directory:

```
cd proj4_post_synthesis
```

3. Copy or move the following files into this folder:

1. tb.v
2. Synthesized netlist file (.vg)
3. program.mem
4. Standard cell library file saed32nm.v

### Step 2: File Preparation

1. Open both the **dut.vg** file and the **tb.v** (testbench) file.
2. At the very top of each file—**before the module definition**—ensure that the following line is included:

```
`timescale 1ns/1ps
```

Note that the **dut.vg** as a part of your submission should contain the above line.

3. In the **tb.v** file, **comment out** the following line to avoid simulation issues:

```
// $display("r[a0]=%08h (%d)", dut.rf.mem[10], dut.rf.mem[10]);
```

### Step 3: Launching ModelSim

1. Open **ModelSim** by typing: **vsim**
2. Create a **new project** and add the following files:
  - dut.vg file (synthesized netlist)
  - program.mem
  - tb.v (testbench)
  - saed32nm.v

#### Step 4: Compilation

1. In the **Transcript** window of ModelSim, execute the following commands to compile the design with the provided cell library (Change file paths accordingly)

```
vdel -all
```

```
vlib work
```

```
vlog /fileSPACE/n/nelango/proj4_post_synthesis/saed32nm.v
```

```
vlog /fileSPACE/n/nelango/proj4_post_synthesis/hart.vg tb.v
```

2. If the compilation is successful, run the simulation using (change hart\_tb according to your module name): `vsim -c hart_tb`

#### Step 5: Running the Simulation

1. Once the simulation environment launches, execute: `run -all`
2. Observe the simulation output in the **Transcript** window. Verify that your post synthesis netlist should be functionally equivalent to your pre synthesis rtl.

## Submission

Submit the following files to the appropriate **GRADESCOPE** assignment:

Deliverable	Points	Notes
schematic.pdf One additional .pdf file for each execution trace	35 (17.5%)	See problem 1.
All Verilog files required for your processor to run (no folder hierarchy, just .v files; you can zip them and submit the zip file and that also works)	130 (65%)	See problem 2.
dut.vg file (synthesized netlist)	35 (17.5%)	See problem 3.
project4.txt	0	<b>Submission Template</b> [Group Member 1]: Name [Group Member 2]: Name ... ai_statement
<b>FPGA Extra Credit</b>	20 (10%)	<b>Contact TAs for details; NOT REQUIRED</b>

For information about how to submit as a group on Gradescope, see [this](#). Only one member of your group needs to submit this assignment.

If you **have used LLMs**, include the following in project4.txt:

*We certify that our usage of LLMs is consistent with UW-Madison's academic integrity policies.*

**Otherwise**, include the following:

*We certify that we have not used LLMs to complete this assignment in any shape or form.*

```
module ai_statement (input wire used_llms, output reg [31:0] ai_statement);
  always @(*) begin
    if (used_llms)
      ai_statement = "I certify that my usage of LLMs is consistent with UW-Madison's academic integrity policies.";
    else
      ai_statement = "I certify that I have not used LLMs to complete this assignment in any shape or form.";
    end
endmodule
```

The results of your submission will be made available after you submit via Gradescope and Canvas. You may submit as many times as you would like before the deadline.