Project #5: Pipelined Processor

ECE 552: Fall 2025

Project Introduction

In this project phase, you will create a **fully functional 5-stage pipelined processor with basic optimizations, extending the single-cycle processor you created in Project 4**. 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.

```
# pipelined_proc.c (pseudocode; not to be taken literally)
#####################################
int main (int argc, char *argv[]) {
  for (;;) {
    if (MEM_WB.regwrite) WRITEBACK(result);
    MEM_WB = MEMORY(EX_MEM);
    EX_MEM = EXECUTE(ID_EX);
    ID_EX = DECODE(IF_ID);
    IF_ID = FETCH(pc);
    pc += 4;
    if (halt_detected(MEM_WB.instruction) break;
  }
}
#####################################
```

Problem 1: Pipelining

Please update your processor to implement a 5-stage pipeline with the following stages: fetch (IF), decode (ID), execute (EX), memory (MEM), and writeback (WB). **Branches should be resolved in the execute (EX) stage**. **Instructions with dependencies should be stalled in the decode (ID) stage.**

Please use **a single clock signal** for the entire datapath to keep timing simple; do not divide your clock or use multiple clock signals (FAQ2 at the end of this document).

We recommend the following three sequential steps when implementing your pipelined processor. In the first step, you will be adding pipeline registers to your datapath. In the second step, you will be adding stalling logic (hazard detection unit) to stall any instruction with dependency in decode (ID) stage. Finally, in the third step, you will be adding forwarding logic (forwarding unit and branch predictor) to eliminate some of the stalls. *Please update your schematic to reflect your changes in each step*.

Step 1: inserting the pipeline registers (IF/ID, ID/EX, EX/MEM, MEM/WB)

Update your schematic to add these pipeline registers and label each stage. Clearly show the signals carried through each stage.

For this problem, start off by not implementing forwarding or branch prediction. Since the processor does not include stalling or forwarding logic, to handle RAW dependencies you need to modify the program to insert nop[1] instructions. (You will eventually need to implement these in the next steps, but it is easier to start without them.)

A quick note on register file bypassing: Recall from Project 3, you implemented bypassing logic in the register file. Starting from this step, you can enable it. See FAQ1 at the end of this write-up for implementation of register file bypassing.

Consider the following simple program:

```
L0: addi x7, x0, 0xf
L1: addi x1, x0, 10
L2: addi x2, x1, 1
L3: addi x3, x2, 1
L4: addi x4, x3, 1
L5: addi x5, x4, 1
```

[1] nop is pseudo instruction, which has no effect and can be used as pipeline stall or “bubble”. Think of it as “addi x0, x0, 0”

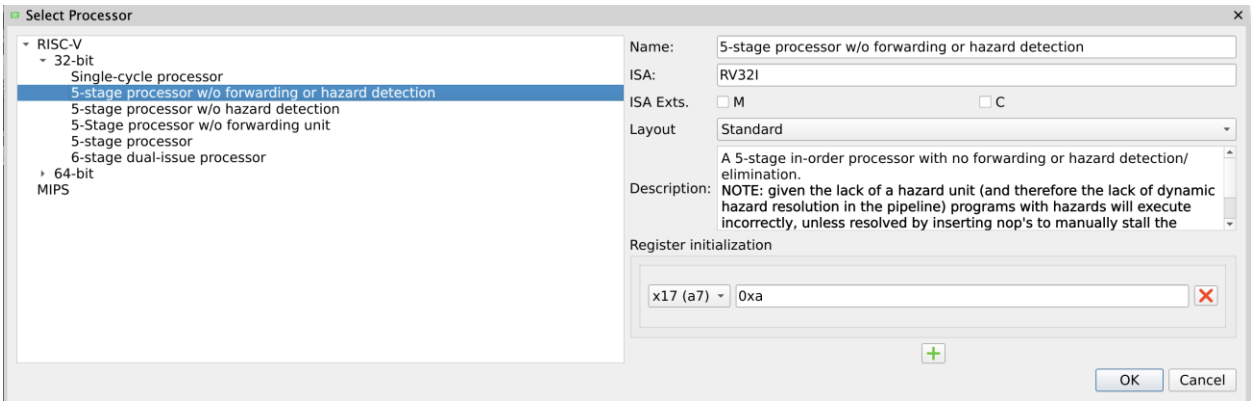
```
L6: addi x6, x5, 1
L7: beq x6, x7, L10
L8: lui a0, 0xdead
L9: ebreak
L10: lui a0, 0x1
L11: ebreak
```

How would you modify this program so that it is executed correctly on your simplistic pipelined processor (no forwarding, no hazard detection, register file bypassing enabled, branch resolved in EX)? More specifically, where and how many nop instructions do you need to add?

Once you insert nop instructions, take this modified program and run it with your simplistic pipeline processor to verify that it is executed correctly. You are also encouraged to write more programs like above with various dependencies. For example, also consider RAW dependency established by load instructions. These programs can be used as simple test cases in the next steps too.

As in Project 4, Ripes may be useful for visualizing how your processor should execute.

In the processor settings for Ripes, you will notice that there are several 5-stage processor models available. You will first implement the model without forwarding or hazard detection, and can choose the path you would like to take to implement the final 5-stage processor in Problem 3.



Note on Ripes simulator behavior: Ripes does not support the ebreak instruction used by your processor. Replace ebreak with ecall instead. Also, initialize register x17 (a7) to 0xA, which Ripes uses to indicate the “exit” syscall for ecall. These requirements are specific to Ripes—you do not need to implement them on your own processor.

Step 2: Adding hazard detection unit

Now that your pipelined processor is working correctly with programs explicitly annotated with nop instructions. The next step is to implement a hazard detection unit that identifies all the dependencies and sends control signals to the pipeline registers. If an instruction is dependent on one or multiple older instructions inflight, **stall it in the decode (ID) stage.** Think about how you would implement a stall/pipeline bubble in your pipeline.

Once you have correctly implemented the hazard detection unit and integrated it into your processor, remove all the nop instructions you inserted previously and run them again on your processor. Verify that it still behaves as expected.

Update your schematic again to add the hazard detection unit and control signals that stall the pipeline. Your processor should now pass the accuracy tests on Gradescope, but will not pass the cpi tests.

Step 3: Adding forwarding unit and always-not-taken branch predictor

Now that your processor is working correctly with unmodified programs. It is time to think about performance. Now add in the forwarding unit and modify the pipelined datapath to support **EX-EX and MEM-EX forwarding**. Note that you don't need to implement MEM-to-MEM forwarding in this phase. The forwarding unit should be able to override the decision of the hazard detection unit. With only EX-EX and MEM-EX forwarding, think about the cases where pipeline stalls are still required; in those cases, the hazard detection should still be able to stall instructions in the decode stage.

Additionally, implement an **always-not-taken** static branch predictor. Flush the incorrect execution path if the branch **is** taken.

If you would like to implement more advanced branch predictors for extra credit (e.g. 2-bit saturating counter), you may do so in a later phase (optimization phase). It is important to have the static branch predictor as baseline to claim performance gain of the more advanced branch predictors.

You may also implement additional optimizations at a later phase if you like. This will result in extra credit proportional to the improvement of the CPI of the processor (**not for this phase**). Please note that it is unlikely that we will be able to assist in debugging any advanced microarchitectural techniques you decide to implement. More information about extra credit will be provided in the later phase; no extra credit will be given for this phase for improving CPI beyond the reference implementation.

You can compare the performance of your processor at step 2 and 3. With the forwarding logic and static branch predictor, the processor should take less (if there is dependence) or the same (if no dependency) number of cycles to execute the same program. Note that you will need to meet the minimum CPI requirement for the optimized processor (if you implement these correctly, you will get full points); unoptimized processors (just step2 or slower than the reference implementation by a certain tolerance) will be penalized.

Update your schematic to add the forwarding unit along with any additional changes you made to the pipelined datapath (e.g., control signals, muxes). Also, add any logic required for the static branch predictor. Your processor should now pass all accuracy and cpi tests on Gradescope.

Problem 2: 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 proj5_synthesis`
2. Navigate to the newly created directory: `cd proj5_synthesis`
3. Inside the `proj5_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:**
   - 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:**
- The synthesized netlist file (.vg) will be located in the **outputs/** folder.
- This netlist file will be used for **post-synthesis verification** (section 3.2).

3.2 Post–Synthesis Verification

Step 1: Setup

1. Exit the proj5_synthesis directory and create a new directory for post-synthesis verification: `mkdir proj5_post_synthesis`

2. Navigate to the newly created directory: `cd proj5_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: 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 3: 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/proj5_post_synthesis/saed32nm.v
vlog /filespace/n/nelango/proj5_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 4: 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	35 (17.5%)	See problem 1.
All Verilog files required for your processor to run	130 (80% accuracy, 20% cpi)	See problem 2 and 3.
dut.vg file (synthesized netlist)	35 (17.5%)	See problem 4.
project5.txt	0	**Submission Template** *[Group Member 1]: Name* *[Group Member 2]: Name* ... ai_statement
FPGA Extra Credit	20 (10%)	**Contact TAs for details; NOT REQUIRED**

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 project5.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.

FAQ

1. **Register file bypassing.** There are multiple ways to implement register file bypassing. The textbook implies that you can configure the register file to write on the negative clock edge, but you would need to change your DFF logic to implement negative edge trigger. The simplest way we recommend is to use a mux as the bypassing logic.
2. **Why is a single clock enough?** You don't need to match the speed of different delay paths by having multiple clocks with various speeds. As long as the single clock period is long enough for data to propagate through the stage and settle before the next edge, the design works correctly. (You might be tempted to use a faster clock for the register file. Please do not do that. Note that the reading logic from the register file is combinational. It does add propagation delay, but it is less than a full clock period.)