Final Project & Research Report:
Speculative, Pipelined RISC-V Processor

Three-fold Objective: To implement a pipelined, RISC-V processor, with data forwarding capabilities and speculative execution. The student will experiment with different types of branch predictors and report the performance benefits for the design enhancements made to the CPU. An in-depth analytical report on the RISC-V CPU's microarchitectural features and performance will be provided by the student.

1. Pipelined RISC-V Processor – Base Design

Use your processor from the last lab to implement the pipelined processor shown in Fig. 1. The architecture remains roughly the same, however explicit pipeline buffers must be added between stages. Since explicit pipeline buffers will be implemented, the temporary registers which were used in the previous CPU version may now be eliminated. Your pipeline will now consist of six pipeline stages: IF/ID/Rd/EXE/MEM/WB to register.

The pipeline registers will temporarily store both instruction control signals and data required for the next clock cycle. You are free to implement these buffering registers as you see fit – one big register which buffers all data and control per pipeline stage, separate control registers and data registers per pipeline stage, etc. Note that the larger the number of inputs per register, the longer the propagation delay, and variation of setup and hold requirements. Ensure that you test your CPU sufficiently to manage your delays and guarantee a proper frequency of operation.

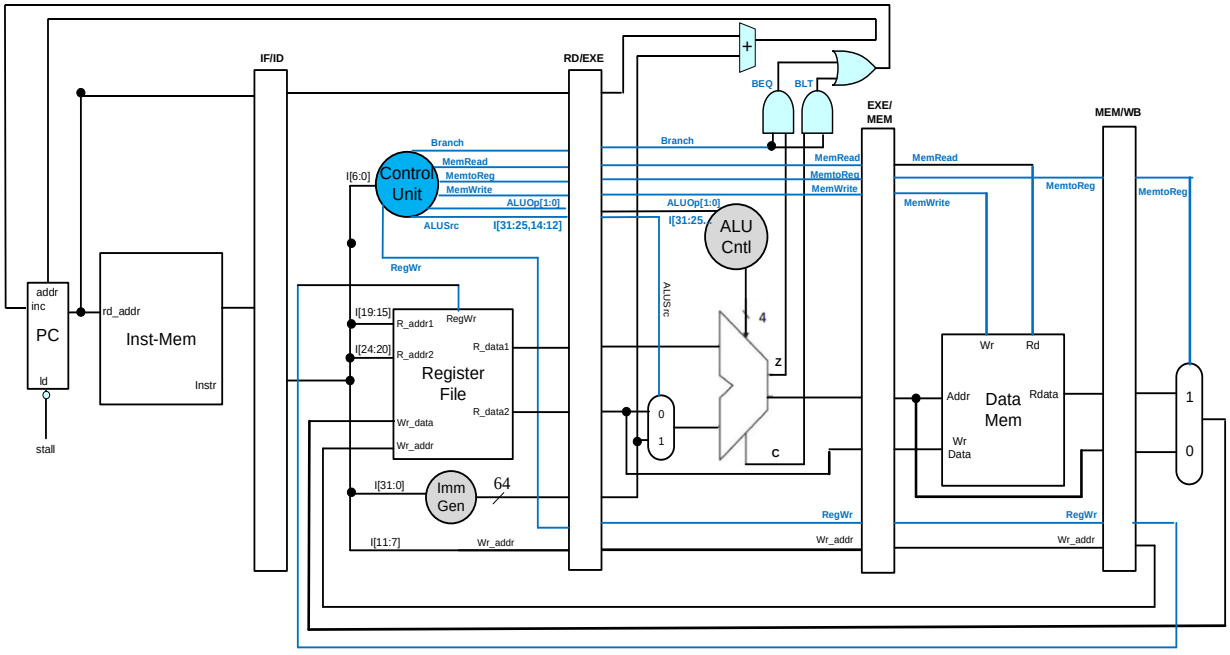

Fig. 1: Preliminary Pipelined RISC-V Processor

2. Data Forwarding

Data hazards, such as Read-After-Write (RAW) hazards, lead to pipeline stalls. Such a hazard occurs when an instruction produces a result (producer instruction) that is required by a

subsequent instruction (consumer instruction). For instance, the producer's destination register is the consumer's source operand. Considering a very simple pipelined processor, if the consumer instruction directly follows the producer instruction in the flow, the consumer operation must **stall** and wait for the earlier (producer) instruction to execute and write back the result to the register file, before it is able to read its source operand from the register file.

This stalling technique is achieved by inserting "bubbles" or "no operations" (NOPS) into the pipeline at decode. NOPS are instructions which do not perform a function and are solely used to stall the pipeline (no reads/writes or operation execution). Stalling however causes performance degradation as the processor may be doing other useful work as opposed to waiting for the producer's result. To minimize the effects of hazards, data forwarding is used in pipelined CPUs to forward results to consumer instructions and improve performance. Note that results are still written back to the register file regardless of the data forwarding.

As expected, the data forwarding technique requires additional CPU hardware to detect hazards, and datapath amendments to redirect the producer's result in the execution stage and the writeback stage to the potential consumer in the decode stage. The forwarding unit that will be implemented in this project is demonstrated in Fig. 2.

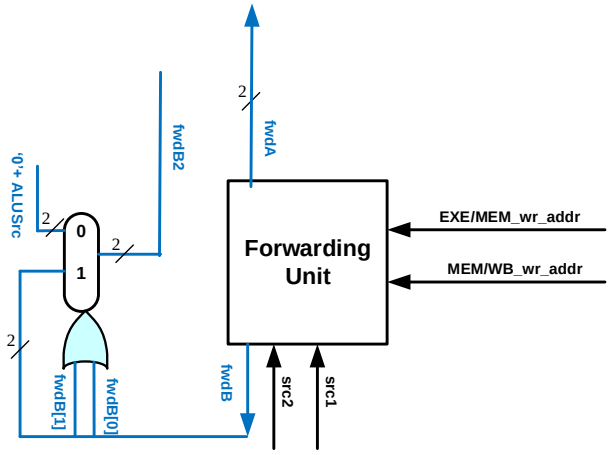

Fig. 2: Forwarding Unit

The forwarding unit accepts 4 input signals: (1-2) the decoded instruction's two source operands (src1 and src2), 3) the destination register (exe/mem_wr_addr) of the instruction currently in the EXE/MEM stage, and 4) the destination register of the instruction currently in the MEM/WB stage (mem/wb_wr_addr). If src1 and/or src2 matches one of the destination registers, then this result data should forwarded to the respective source input(s) of the ALU.

As seen in Fig. 2 and 3, the two output signals, fwdA, and fwdB are 2bit signals that determine the necessary multiplexer control signals to forward src1 or src2 upon a match. The truth table for these signals are given in Table 1. If **no match exists** between src1/src2 and the registers of EXE/MEM or MEM/WB stages, then the operand may be safely obtained from the register file as performed in our previous version of the processor (select "00"). If there **is a match** between src1/src2 and one of the wr_addr input, then the corresponding control signal bits, FwdA /fwdB, must be asserted according to Table 1 to forward the correct data. Furthermore, if src1 or src2

matches the wr_addr operands from **both** the exe/mem and mem/wb stage, then the latest version of the result should be obtained i.e. the result produced by the exe/mem stage since the mem/wb represents an older version of the result.

Since the second input operand of our ALU already possesses two input possibilities, immediate or register file, we must adjust this to a 4:1 multiplexer to support the two new possibilities: register file operand, immediate, exe/mem result, or mem/wb result. The new 4:1 multiplexer is shown in Fig. 3. To determine the correct result to forward from the writeback stages, the **fwdB** control signal requires an additional level of interpretation, as illustrated in Table 1:

- If either fwdB(0) or fwdB(1) is asserted (1), the operand **must be forwarded**. This control signal will generate a 01 or 10.
- If fwdB(0) and fwdB(1) are both 0, then the operand source will be a register file or immediate value. In this case, the ALUSrc must be appended with '0' ('0' + ALUSrc) and sent to the second operand's multiplexer.

Table 1: FwdA and FwdB Control Output

Mux control	Source	Explanation
ForwardA = 00	ID/EX	The first ALU operand comes from the register file.
ForwardA = 10	EX/MEM	The first ALU operand is forwarded from the prior ALU result.
ForwardA = 01	MEM/WB	The first ALU operand is forwarded from data memory or an earlier ALU result.
ForwardB = 00	ID/EX	The second ALU operand comes from the register file.
ForwardB = 10	EX/MEM	The second ALU operand is forwarded from the prior ALU result.
ForwardB = 01	MEM/WB	The second ALU operand is forwarded from data memory or an earlier ALU result.

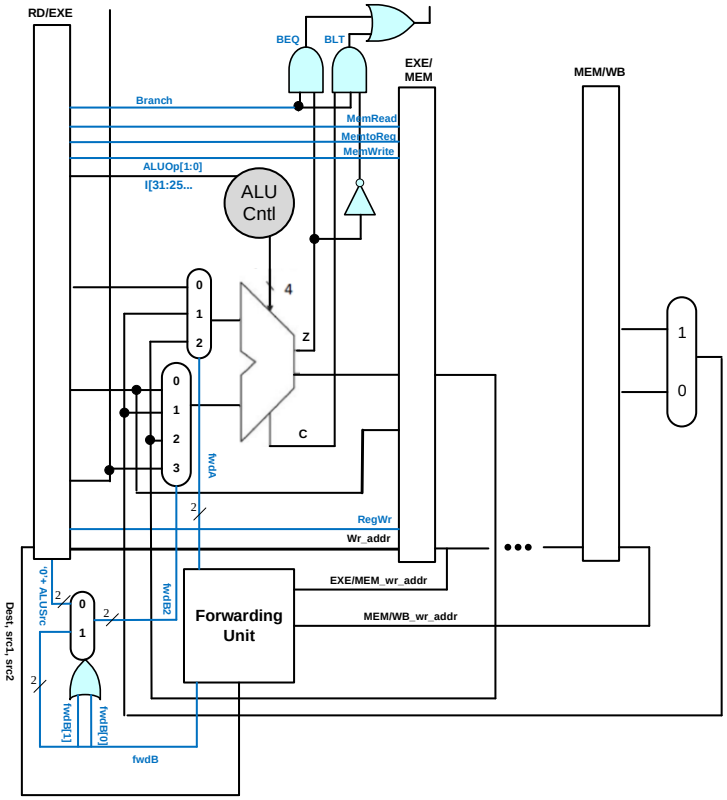

Fig. 3: Pipeline Adjustments for Data Forwarding

3. Speculative Execution and Branch prediction

Throughout the semester, you have iteratively created and optimized a simple RISC-V CPU. Although the CPU now supports forwarding to eliminate data hazards, the CPU must still stall upon a branch instruction and wait until it is resolved at the execution stage, in turn degrading application performance.

For the third part of the project, students will design three different, traditional branch predictors – two static and one dynamic branch predictor – which will be integrated into your CPU design. It is a good idea to create a separate component for the branch predictor that can easily be integrated into your IF stage for testing and implementation purposes.

When a branch is encountered in a CPU's instruction stream, a branch predictor will predict the branch direction likely taken by the instruction. The fetcher will update the PC to the branch predictor's "predicted" direction and fetch the instruction for processing. If a correct prediction was made, then the CPU was successfully redirected to the correct instruction, where performance was improved, and stalling was completely avoided. If an incorrect prediction was made, then the same penalty is endured as stalling (wasted cycles): In the case of stalling, NOOPS would be injected into the pipeline until the branch was resolved, whereas branch prediction fetches potential correct instructions into the pipeline for processing. If the branch was predicted incorrectly, then not much performance was lost.

Static branch predictors always predict the same result for a branch – taken or not taken. Hence static branch predictors are very easy to implement. On the contrary, **dynamic branch predictors** use and update branch history to make a more accurate prediction on a given branch's direction. Students will independently **research various dynamic branch predictor designs (there are many!). Implement and integrate one dynamic branch predictor, and two static predictors into your CPU.** Branch predictors will be compared for prediction accuracy and performance improvements.

Recovering from branch mispredictions: To recover from mispredictions considering an in-order pipelined CPU, the pipeline will need to be flushed and the correct instruction will need to be fetched from instruction memory. Implement the logic required to recover from a branch misprediction within your datapath. Data forwarding and the pipeline amendments of the previous parts must still be supported by the CPU.

4. Project Report

Write a 10-15 page report, IEEE 2-column format, on your RISC-V CPU design. Sections should include (but are not limited to) *Introduction, Course Review, Methodology (simple CPU, pipelining, hazards, speculation and branch prediction), Experimental Setup, Results, Future work and Conclusion*. An appendix should also be included for source code, testing waveforms, and will not be considered as the 10-15page writeup. For diagrams which do not fit within the 2-column IEEE margins, please place them in the appendix and refer to them throughout the report.

Provide a **Methodology** based on your CPU design research, and on different types of branch predictors. Discuss your method and design for handling branches and recovering from

mispredictions. For CPU design, discuss the benefits, solutions, and drawbacks of data hazards and speculative execution.

For **experimental setup**, describe the tools which you have used, the predictors and different CPU models which you have designed and will be testing in the report. Describe the algorithm/benchmarks which you will be using to assess performance of the CPU models. Feel free to use Algorithm #1 from Lab 5 as one of your benchmarks.

During **experimental results**, be sure to report and compare:
-Performance of a fully-pipelined versus semi-pipelined design in Lab 5
-Performance with no branch prediction + forwarding versus branch prediction + forwarding
-Branch predictor accuracy per benchmark
-Provide diagrams of branch predictors used in your report, and their integration within your pipelined design

Future Work – take a look at superscalar processors, and Out-of-Order processors. How do these differ from the design which you implemented during this course? And how would they benefit application performance?

5. RISC-V CPU Emulation

Top-Level: To test and emulate the CPU developed in this project, certain hardware structures in the pipeline must be observable and used as the top-level inputs and outputs of your CPU. Specifically, we would like to verify the outcomes of the CPU's execute stage, which require the following signals: the **Program Counter** address (at execute), and the **ALU's two inputs, result, carry, zero and branch** flags. Accordingly, only these signals should be included as ports in your CPU's top-level file, along with **clock** and **reset.**

Overview: Once an appropriate top-level is designed with the signals specified above/ports in the entity, students will create a block symbol of their VHDL CPU using Quartus. The block will be integrated within a provided schematic test-fixture (final_project.bdf) which contains other hardware blocks and DE2 pins necessary to interface the CPU block to the Cyclone IV FPGA/DE2-115 dev-board. The test-fixture **schematic** provided is an alternative to creating a top-level VHDL that would port-map all component to the FPGA and dev-board peripherals. The schematic consists of component blocks for the LCD driver, HEX display, and FPGA pins which are graphically interconnected with appropriately named wires (STD_LOGIC) and buses (STD_LOGIC_VECTOR) to facilitate communication between all components. This file must therefore be set as the top-level to successfully port the 64-bit RISC-V CPU on the DE2-115 dev board. More specifications about generating blocks and creating interconnections is provided below.

Adjustments: To accommodate the limited datawidth supported by the dev board to display values, it is recommended that students use 16bit data values, in the data memory mif and register file, to properly display inputs and outputs to the on-board peripherals. That is, leave the CPU as a 64-bit RISC-V processor, however extract the lower bits of these signals when interfacing to the peripherals to successfully test the design and verify functionality.

CPU to Peripheral Assignments: The two ALU inputs will be displayed on the two rows of the LCD, respectively. As the LCD can only support up to 32bits per line, displayed as hex, the lower 32-bits of the 64-bit ALU inputs should be extracted and mapped to the two 32b inputs of the LCD. Since students will be working with 16bit values, only half of these bits will be shown on the 32-bit inputs to the LCD.

Students will be provided with the files necessary for the LCD controller (including clock dividers, decoding units etc). The provided schematic (final_project.bdf) integrates all logic for the LCD, hex, and pins to work properly with the board. Therefore the student must simply make the proper connections from their generated CPU symbol to the hardware components and pins present on the schematic.

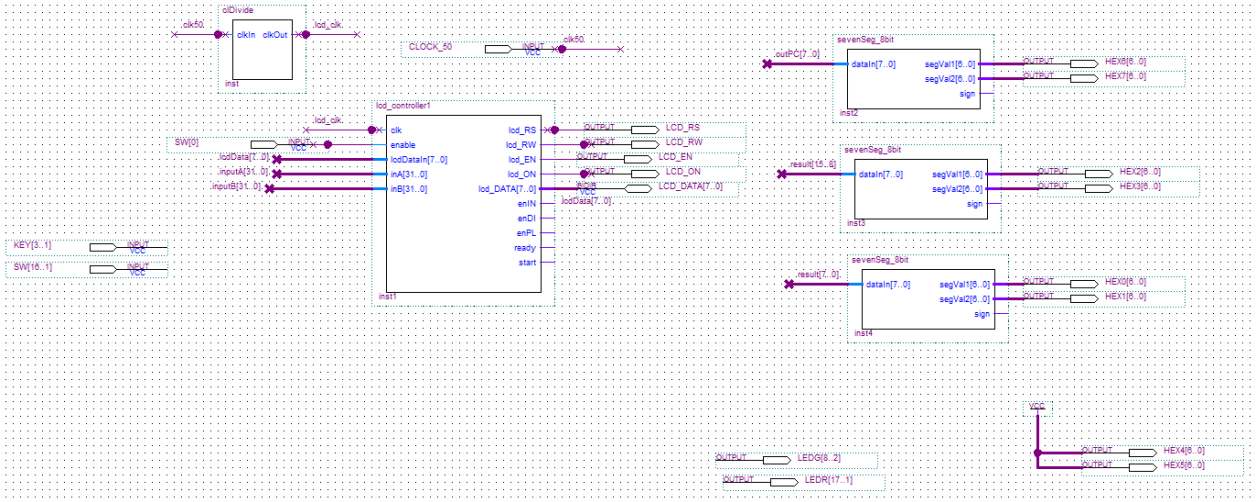

Fig. 4: final_project.bdf Schematic – Test-Fixture

Schematic connections to CPU: The 32bit ALU inputs should be connected to the two LCD data inputs, inA[31..0] and inB[31..0], as seen in Fig. 4. SW[0] should be connected to LCD_ON which enables and disables the LCD from displaying text. The result of the ALU will be displayed on the four HEX displays HEX3, HEX2, HEX1, and HEX0.The zero and carry flags should be mapped to LEDG1 and LEDG0 respectively, with the branch flag from ALUCntl connected to LEDR0. The last 8-bits of the "execution" PC should be connected to HEX7 and HEX6. Finally, the control signals clock and resets should be connected to KEY[0] and SW[17], respectively. The resulting schematic should resemble Fig, 6.

As mentioned previously, the provided final_project.bdf schematic should be set as the top-level file. To appropriately map the pins on the schematic, import the **DE2_pin_assignments.csv** file into your project by using the **Assignments > Import Assignments…** option in Quartus. It is recommended that you open the pin assignments file to observe all the pins associated with the LCD display.

Creating and Integrating the CPU block into the design:
To create a block symbol for your CPU with the appropriate entity declarations, in Quartus under the left "Project Navigator" panel, select the **files** tab, and right-click your CPU top-level file. In the pop-up menu, select "**Create Symbol Files for Current File**". Quartus will generate the symbol for you, which you may find in the folder with the extension .bsf (block symbol file).

To integrate the .bsf into the final_project.bdf schematic, go to this schematic and right-click any blank place on the canvas. In the pop-up menu, select **Insert > Symbol …** which will launch a window. Expand your **project's folder** and select the appropriate CPU block for integration. Place the symbol on the appropriate place on the schematic canvas, similar to the CPU seen in Fig. 6.

Updating your CPU file requires the regeneration of the block. When any adjustment is made in the VHDL, repeat the above steps to update and regenerate the block. Once complete, right-click the existing block in the schematic and select "update symbol or block…" > "All symbols or blocks" in this file. Re-synthesize the entire project.

Making wired connections:

Fig. 5 displays the wire/pin toolbar in Quartus used to select the necessary connections from your CPU block to the periphery and pins in the bdf. You will require the wire icon to create wires, the bus icon for bus connections, and the pin icon to insert pins for LEDs, KEYs and any necessary switches (SW).

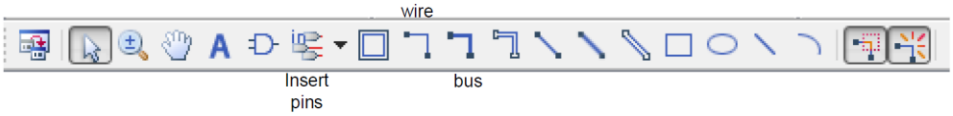

Fig. 5: Wire/Pin Toolbar

A connection may be made to a pin or a node (marked with 'x' at the end of the bus/wire) by clicking the mouse on the commencing point and releasing the mouse on the destination point. The (unconnected) node wires imply that the other end is connected elsewhere on the schematic, also marked with an 'x'. To ensure a connection between the nodes, you must name the wires appropriately by right-clicking the wire and selecting **Properties**. In the name field, write an appropriate name so that Quartus may make a connection to the other (same named) wire. An example is shown in Fig. 6 with the outPC[7..0] bus, connecting the CPU to the seven segment display. Buses are named as vectors xxxxx[#...#] and wires are named alphanumerically (xxxxx).

Pins may be named simply by double clicking the text next to the pin. Pins may a vector or single bit simply by the vector/alphanumeric naming strategy explained. Based on the connections, pins and blocks specified, the final schematic for the project should resemble Fig. 6.

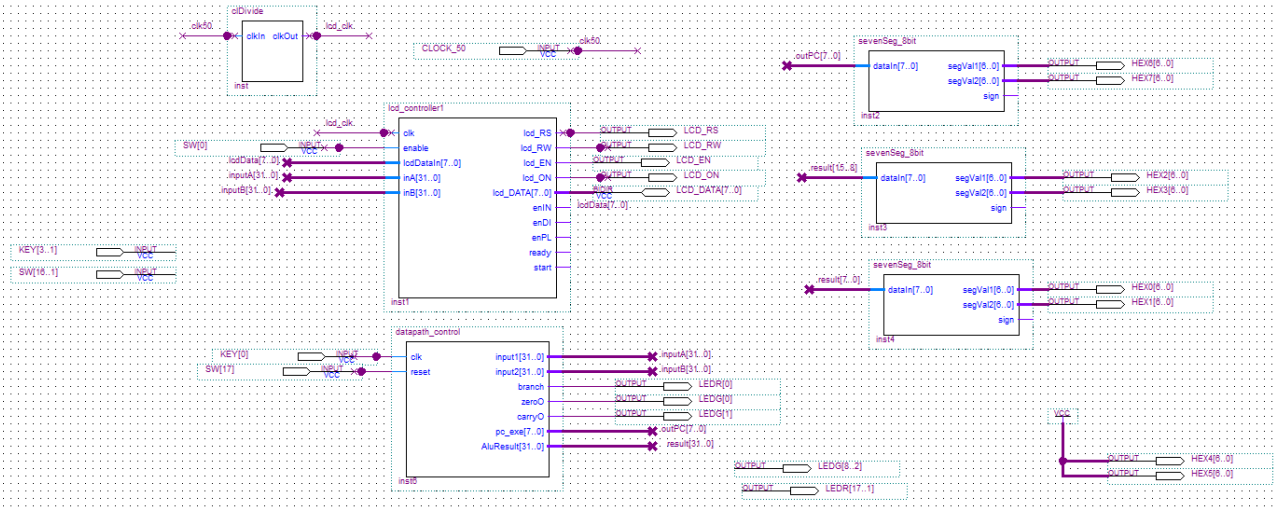

Fig. 6: Final top-level project schematic

Deliverables:

- **VHDL** –new datapath implementation for part 1, part 2, and part 3
  - Emulated project schematics necessary to replicate on DE2-115
- **Mif** for instruction memory and data memory values used for testing
- Waveforms for the datapath design, implementing the specified algorithm
  - Recommended – testbench approach in Modelsim will be easier to debug the CPU
- **Research Report** – IEEE report with specifications as outlined in Section 4.
- **Demonstration** of working project emulated on the DE2-115 according to the specifications outlined in Section 5.