ECE 552 Final Report

Coolerst Guys - Thomas Mandel, Gokce Pilli, Mayank Katwal

Overview

With the accumulation of the work done on the three project phases, we have implemented a 5-stage pipelined processor that can fully handle caching, data forwarding, hazard detection and flushing on branches. Our development of the *WISC-F18* instruction set architecture can handle 16 instructions that include stores and loads from memory to registers. For the store and load instructions if the data is not present in the cache we can grab half-word size data insuccesion for 8 consecutive cycle. Due to 4 cycle memory access delay it would in total take 12 cycles to grab 8 half-word data.

The module hierarchy can be seen below:

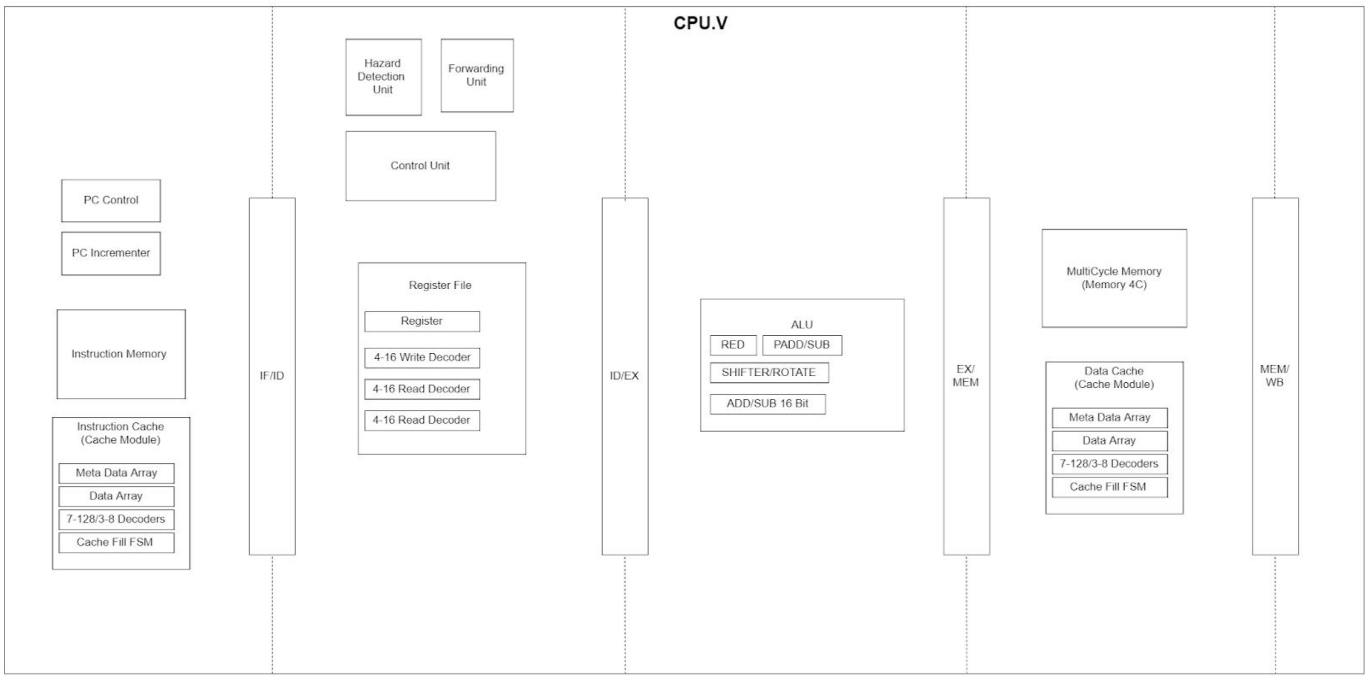

Fig. 1 *Module Hierarchy*

Smaller modules such as d-flip-flops and multiplexers were left out of the diagram for simplicity. A larger copy of the image is also in the zipped folder that is submitted.

Task Breakdown

Responsibilities Breakdown

	Thomas Mandel	**Gokce Pilli**	**Mayank Katwal**
Phase 1	- Shifter Module - CPU.V Module - Debugging CPU.V - Branches	- Test Benches - Debugging CPU.V - Branches	- ALU Module - RED Module - Debugging CPU.V - Flag modules
Phase 2	- Wiring CPU.V module - Forwarding Unit - Debugging CPU	- Pipeline Register implementation - Debugging CPU	- Forwarding Unit - Hazard Detection Unit - Pipeline Register implementation
Phase 3	- Wiring Cache to CPU - Debugging Cache fill FSM module - Debugging Cache module - Debugging CPU.V	- Cache fill FSM implementation - Cache.V implementation - Debugging Cache - Debugging CPU.V - Debugging Cache fill FSM module	- 3-8,7-128 decoder implementation - Debugging Cache module - Report

Table 1. *Responsibilities Breakdown*

Work Percentage Breakdown

	Thomas Mandel	**Gokce Pilli**	**Mayank Katwal**
Phase1	35%	30%	35%
Phase 2	35%	35%	30%
Phase 3	30%	40%	30%

Table 2. *Work Percentage Breakdown*

Overall, we split up equal work equally for everyone on the team per project phase. Mayank and Gokce were mainly in charge of creating individual modules and testing them. Thomas focused on connecting all of the modules and making sure the integration went well. We would all then meet for debugging sessions and run tests until we were certain that the outcome was correct. We also used github as version control so everyone could commit their working code, making it easier for us to integrate pieces. Looking at the table above, the work was split up very evenly in summary, every one contributed to the success of the team.

Special Features

For phase one, our main optimizations focused on low area. We spent time converting decoders into shifters, and minimizing logic paths.

For phase two, we ran into issues with branching, recognizing at the end that we had zero-extended the branch immediate rather than sign-extending it. This resulted in miscalculating the branch address when jumping backwards in the program. Our optimization for this phase focused on minimizing the number of signals that traversed our implementation through the pipeline registers. This helped minimize our area throughout, reducing the number of multiplexers whenever possible.

On project phase three, we spent a majority of time debugging our program. The first, and most difficult issue to overcome was finding a way to write and read to the metadata arrays at the same time. While debugging this issue, we also noticed the need for stalling registers later in the pipeline, to ensure correct forwarding execution. Lastly, we had implemented the R0 register in such a way that it did not update the ALU flags (because R0 is hard-coded to 0). Once these three things were addressed, the programs ran smoothly.

Completeness

Phase 1	Complete and **passes all 3 tests**, RED implementation was incorrect
Phase 2	Complete and **passes all 3 tests** with new RED module
Phase 3	Complete and **passes all 4 tests** as shown in demo

Table 3. *Completeness of Project Phases*

Testing

Starting with project phase 1, we knew how modular the project was going to be. So in order to guarantee the integration was a success at the end, we wrote test benches for every module we wrote. This included testbenches for the PADD/SUB, RED, Shifter and rotator, PC control and then finally the main cpu. Since the success of each individual testbench guaranteed that the particular module was working correctly, we could narrow down our debugging to the main CPU file. Because we did initial testing, we saved a lot of time debugging at the end and had all of our tests passing.

For project phase 2 we added the four pipeline registers, hazard and forwarding unit. To test our implementation progressively, we tested each pipeline register before adding the next. Only once our entire flow was successful did we add the forwarding unit and hazarding unit into the mix. The last thing we did was add stalling for the hazard unit. To test all of the possible combinations of hazards and branching, we created 4 extra tests.

For phase three our main testing strategy was to implement the instruction cache first and guarantee its correctness when filling the cache from memory and responding to cache hits. Only then did we move on to the data cache. This proved to be more challenging as we now had to consider not only stores to memory, but additional stalls in the pipeline, as well as memory contention with the instruction cache. One of the major things we noticed was the importance of maintaining correctness for the forwarding unit when stalling registers later in the pipeline. In this stage, we struggled to overcome the issue of reading and writing from the metadata arrays in the same cycle, but solved this by implementing the LRU logic in a separate array in the cache module. All of these strategies contributed to the success of all test cases in phase three.

Results

🟩 Correct Outcome	🟥 Incorrect Outcome

Outcome and Cycle/Hit Count:

	Test 1	**Test 2**	**Test 3**	**Test 4**
Phase 1	🟩 **14 cycles**	🟩 **15 cycles**	🟩 **22 cycles**	N/A
Phase 2	🟩 **18 cycles**	🟩 **20 cycles**	🟩 **34 cycles**	N/A
Phase 3	🟩 **42 cycles** I-cache Hits: **15** D-cache Hits: **0**	🟩 **45 cycles** I-cache Hits: **18** D-cache Hits: **2**	🟩 **58 cycles** I-cache Hits: **31** D-cache Hits: **0**	🟩 **365 cycles** I-cache Hits: **290** D-cache Hits: **35**

Table 4. *Result and counts for each Project Phase*

For project phase 2 test 3, we had the correct results but our cycle count was higher than expected. This is due to extra stall cycles per each branch instruction. Also for project phase 2 we wrote our own test cases, tests 4, 5, 6, 8 to make sure we were branching correctly as mentioned above.

Extra Credit Synthesis/Features

We were not able to synthesis the design in the given time

Verilog Files

All of the verilog, testbenches and trace files are included in the zip folder and submitted on canvas