**RV32I Pipelined Processor**

**Risc-y\_Business0**

**Atsuhiro Suzuki**

**Bryce Tharp**

**Dylan Lim**

TA: Greg Bae

**Introduction**

The goal of our project was to design a fully pipelined RISC-V processor in SystemVerilog. Additionally, the processor must handle a variety of hazards and contain data forwarding capabilities. In a broader sense, this project serves as a capstone project demonstrating our understanding of computer architecture principles learned throughout the semester. This report will walk through the technical aspects of our design, the milestone achievements made along the way, and describe design decisions that led to the final product.

**Project Overview**

The primary goal of this project was to create a five-stage, pipelined RISC-V processor complete with hazard detection and data-forwarding. Secondary goals included designing and implementing a suite of advanced features to optimize the performance of our processor, as well as working together as a team to simulate an experience one would face in industry. The decision to implement a RISC-V processor was heavily influenced by the direction that the computing industry is headed. RISC-V is becoming a more popular computer architecture, in large part due to its open source nature. Many companies in the industry are using RISC-V, so by also implementing RISC-V we are maximizing our industry preparedness. Additionally, almost all modern day processors are pipelined, and the five-stage design is a very common form of the pipeline, so, again, it comes down to trying to attain experience that matches the industry as much as possible. Furthermore, most industry work revolves around trying to make optimizations to improve performance, and by challenging ourselves with a design problem and trying to implement our own design, we are replicating a real life work scenario.

As much as this project was about showcasing our technical ability to design a processor, the teamwork portion of this project can not be ignored. Our team had to decide how we were going to work together remotely, create guidelines for submitting work, schedule meetings, and split up tasks. For each checkpoint, the team would use Messenger and Discord voice chat to allocate workloads (e.g. Dylan handles writing the monolithic registers while Atsu handles the control unit). This was a critical line of communication that was necessary for meeting all of the deadlines we encountered. GitHub was used as our repository for sharing and saving our work. There were instances where we had to debug as a group to locate and fix bugs, we used CodeShare and screen-share to work together in a remote development environment. Everyone contributed and communicated effectively for us to meet deadlines. Complications included trying to understand each other's code after we came back from individually working on our delegated parts as we debugged the processor, as well as failing to have a unified naming convention and the bugs that followed.

**Design Description**

We chose to make a five stage processor (Figure 1). In between each stage, data and other signals are passed through monolithic registers. Control signals are passed between stages using control words. The fetch (IF) stage holds the PC and fetches the instructions from the instruction cache. The decode (ID) stage decodes the instruction based on opcodes and sets control signals in a control word to be passed on and used in the other stages. This stage also contains the register file (regfile). The execute (EX) stage handles execution of opcodes using an arithmetic logic unit (ALU) and comparator (CMP). Inputs for the ALU and CMP are forwarded from other stages. The memory (MEM) stage handles reading to and writing from the data cache. Memory addresses for this stage are passed in from the ALU in EX. The writeback (WB) stage determines the RD value for the regfile using a mux. An L1 cache is used for memory between the CPU and physical memory. The arbiter for the cache uses an FSM to handle cache misses and thus physical memory access.

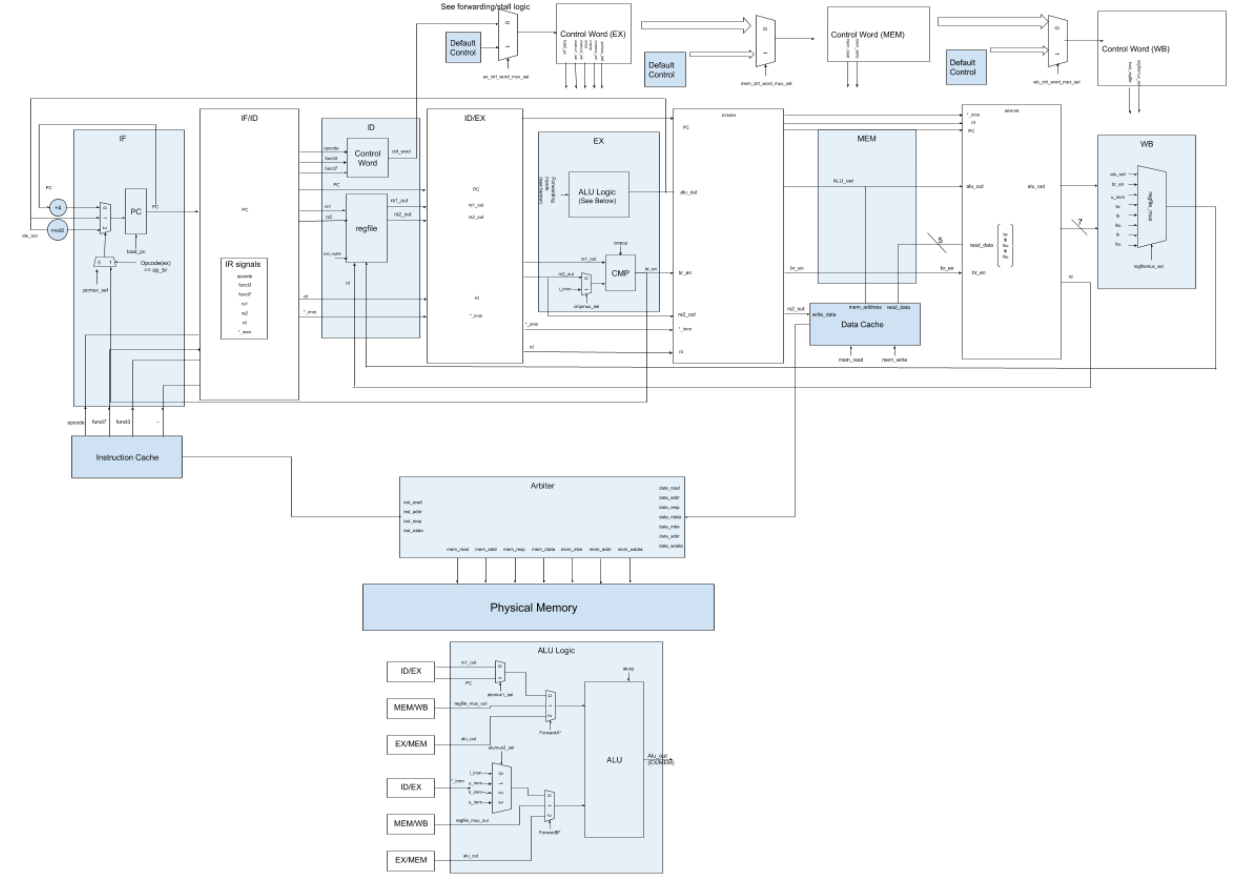


Figure 1. Processor Datapath

The processor was designed using milestones set by the checkpoint requirements. In checkpoint 1, we implemented the basic pipeline design to include basic functionality. In checkpoint 2, we updated the processor to allow for data forwarding between stages, hazard control, static branch prediction, and cache integration. In checkpoint 3, we attempted to implement dynamic branch prediction and L2 cache integration.

For checkpoint 1, we designed the pipeline based on the datapath from MP2 (Figure 2) and the pipeline design from Lecture 5 (Figure 3). We integrated the components from MP2 by separating them into their respective stages (Figure 4). Control words are used to input the control signals for muxes, registers, and other design units in each stage (Figure 5). The control word is decoded from the instruction in ID. A mux is used to set the control signals with selection from the opcode. We chose to pass data between stages using four monolithic registers. For example, the IF/ID register takes inputs such as PC values from the PC register, and instruction from the instruction cache and outputs them to the ID stage. In this checkpoint, we used the dual-port magic memory provided instead of a cache, which essentially abstracted out memory and we acted as if we would never have a cache miss. The purpose of that was because since there are two caches (instruction and data), we would need some sort of arbiter to interface with a singular memory IO, and we decided to push that implementation until later. The design was tested using the some given testcode for this checkpoint and other code we wrote for unit testing. Verification was done using ModelSim to check register values and to track functionality during debugging, where we essentially tracked the values in the registers to see if they matched the expected value when written back to the register file in the write back stage, and also if the program flow matched the program (i.e. we branched to the correct location). An issue we encountered was that it took too many clock cycles to execute each instruction. We initially updated PC using the value of PC at the end of WB stage, as seen in Figure 4, thus the pipeline would not continue to the next instruction until PC was updated 5 clock cycles later. We then changed it so that the PC we used was from execute, however, that would still result in 3 cycles per instruction, so finally we made it so that PC would be updated from the immediate value of PC in IF, as seen in Figure 1, so every clock cycle it would change to the pc of the next instruction.

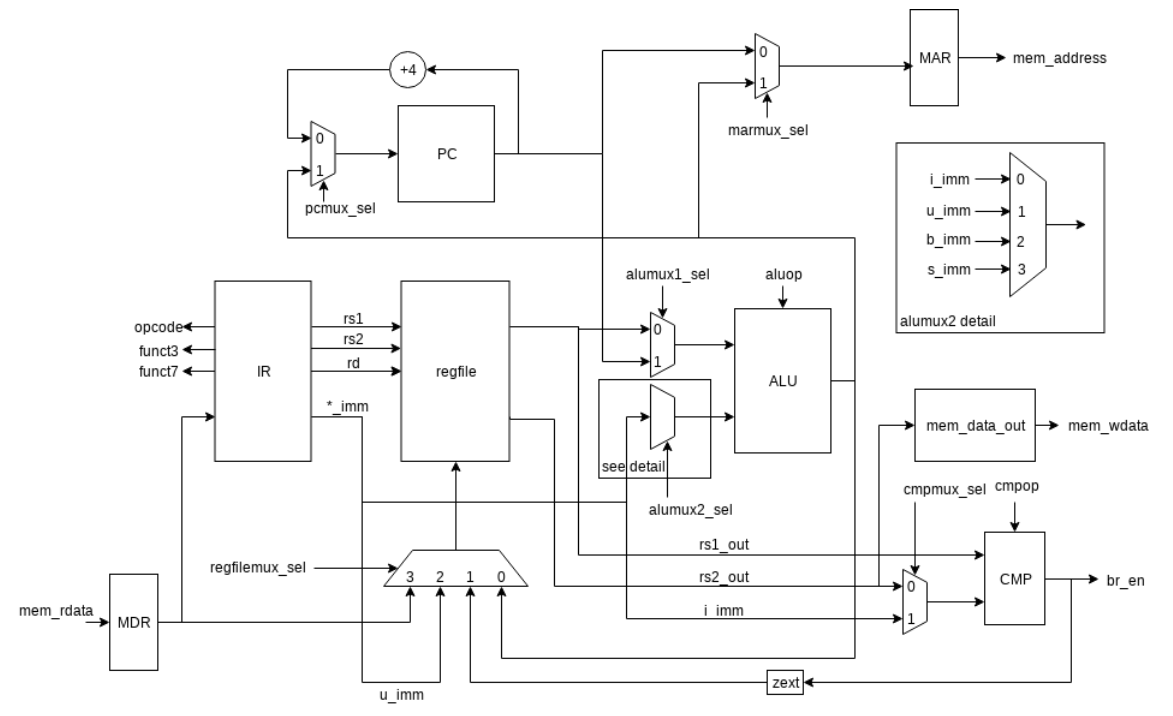


Figure 2. RV32I Datapath from MP2

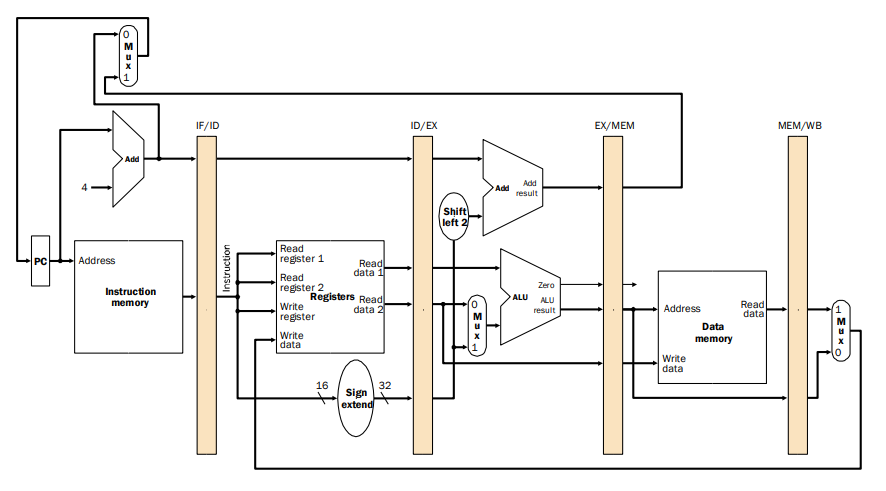


Figure 3. Pipeline Design from Lecture 5 Slides

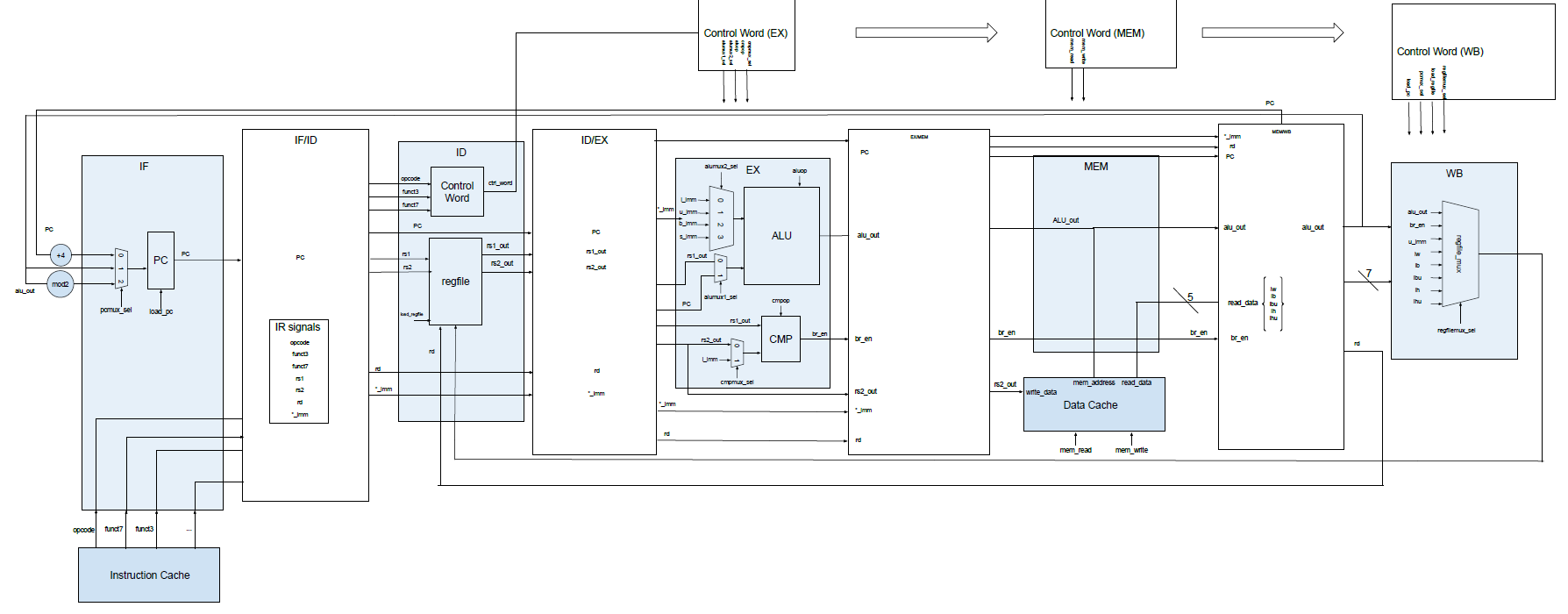


Figure 4. CP1 Processor Datapath

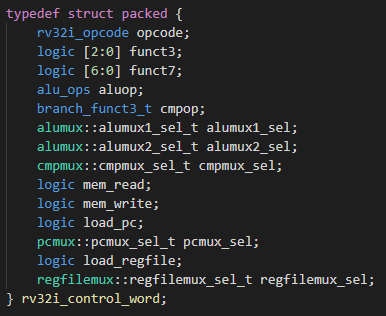


Figure 5. Control Word Data

In checkpoint 2, we implemented static branch prediction, data forwarding, hazard detection, and we integrated the cache into the design (Figure 1). For a branch instruction, normally you would have to wait until the EX stage to see if you should take the branch or not from the output of the comparator. However, using branch prediction, when we receive a branch instruction, we assume that the branch is not taken and when we get to execute we can confirm whether or not the prediction was correct based on the br\_en signal in EX stage (Figure 6). We continue to update PC to PC + 4 for two clock cycles so that IF and ID continue to receive instructions. If the prediction is correct, we continue as normal and save two clock cycles, but if the prediction is incorrect then IF and ID are flushed (Figure 7).

The purpose of data forwarding is to eliminate stalling the pipeline on hazards. For example, if we write to a register and the following instruction uses that register as a source register, the following instruction would not receive the correct register value because it hasn’t been written back to the register file yet. One solution is to simply stall the pipeline until the register has been written back, but a better solution would be to forward the value that is going to be written back to the input of the execute stage before it gets written back. Below you can find the forwarded ALU (Figure 8) and also the logic that determines what to forward (Figure 9). One problem that we ran into was that we were sometimes forwarding when unneeded, and the reason was we weren’t checking whether or not an instruction actually used a source register or a destination register, and some data that in the instruction that wasn’t meant to be interpreted as a register number was interpreted as one.

Even with forwarding, there are still some hazards that necessitate a pipeline stall or a flush (without the addition of extra advanced logic). Hazards are detected using the logic in Figure 10. As stated before, when there is a branch mispredict, we need to flush the pipeline. Flush here means we set the IF\_ID buffer and ID\_EX buffer to 0 and set the control word in execute to a nop instruction, essentially erasing the instructions that were loaded in previously. We also made the decision here to apply the same flushing logic for jump instructions. While this is technically not necessary as for jump instructions we are always going to that location, it was easier to implement and we were short on time. If we had more time when the jump instruction is put into the fetch stage, we would update PC there.

Situations where we need to stall the pipeline are for a load conflict and for cache misses. Stalling the pipeline here means that we stop PC from being updated, stop the IF\_ID buffer from being updated, and insert a nop into the pipeline (set the EX control word to 0). A load conflict is when there is a load instruction right before another instruction that needs to use the value from that load. We won’t know the value from the load until after the MEM stage, so we can’t forward from the execute stage, so we just stall by one cycle. We know if there is a memory cache miss if the data cache has not responded and we are also doing a write or read in MEM stage, and we know if there is an instruction cache miss if the instruction cache has not responded because we should be reading from the instruction cache every cycle.

Our team used the provided, one-hit cache for the design. However, we designed and implemented the cache arbiter (Figure 12) - a control unit that interfaces with memory. The arbiter determines which cache requests are served first between the instruction cache and the data cache. This is especially important when both caches miss, and thus due to the one hit cycle requirement, need to access memory at the same time. One notable feature of our arbiter design is that we used only three states, when typically more is used. This simplified our arbiter logic.

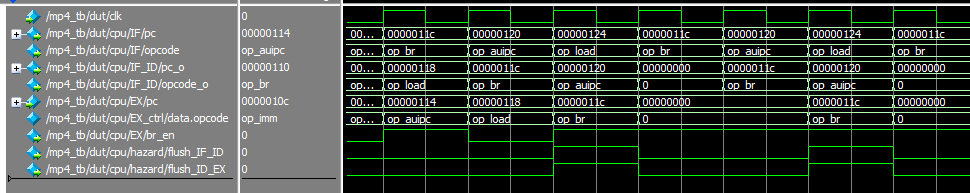


Figure 6. Waveform when a Branch Instruction is Received

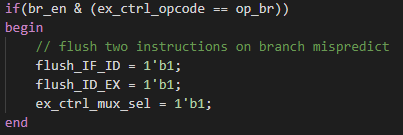


Figure 7. Flush Logic

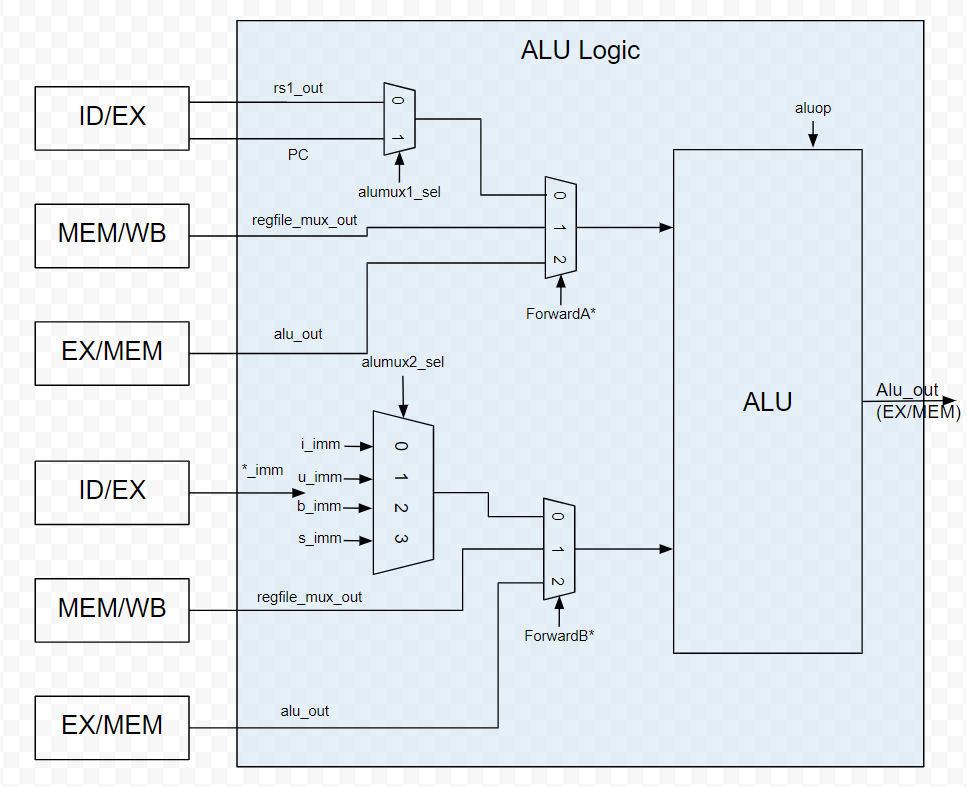


Figure 8. Forwarded ALU

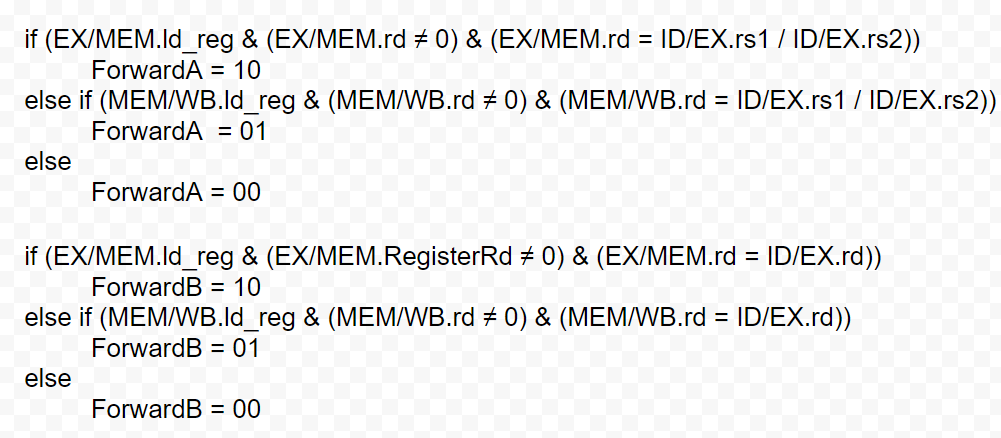


Figure 9. Forwarding Mux Logic



Figure 10. Hazard Detection Logic

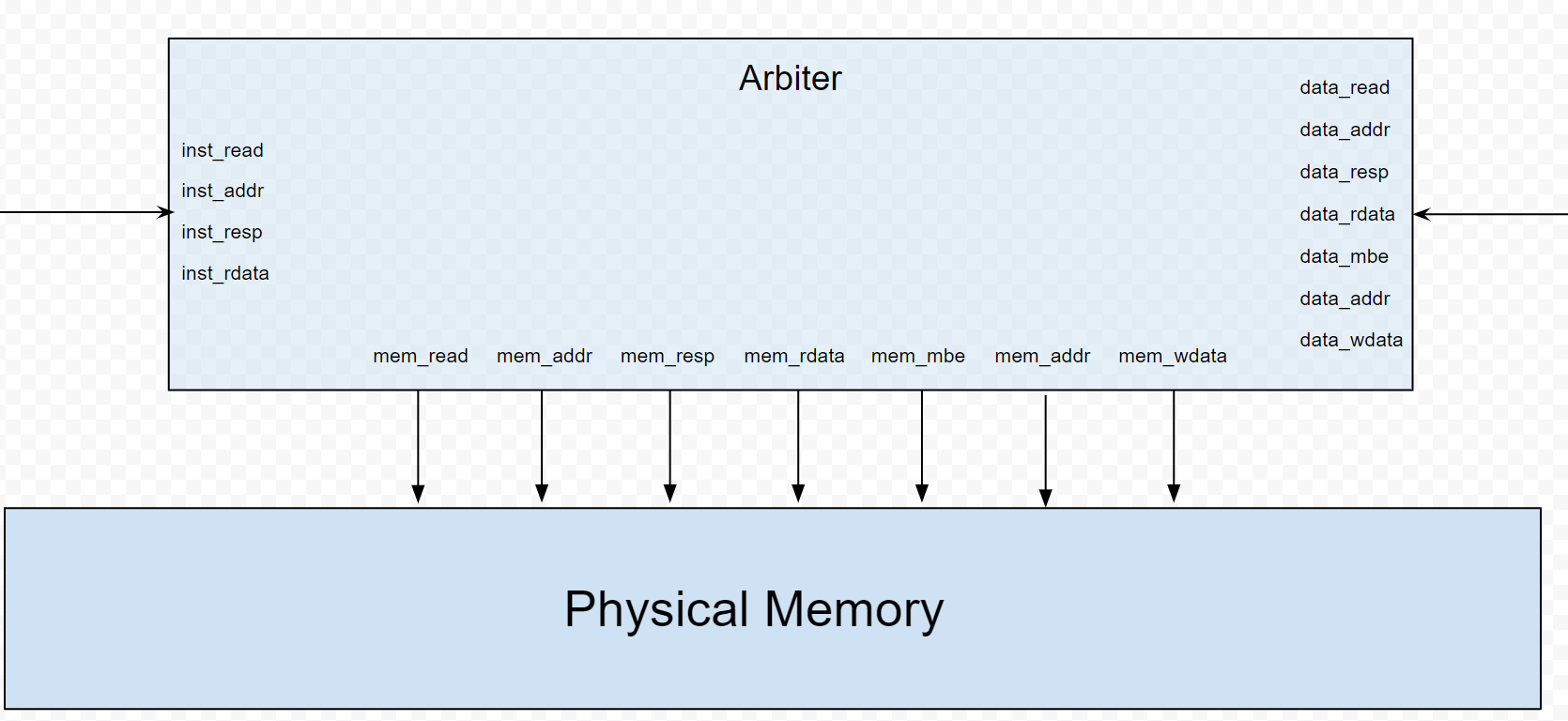


Figure 11. Arbiter Signals

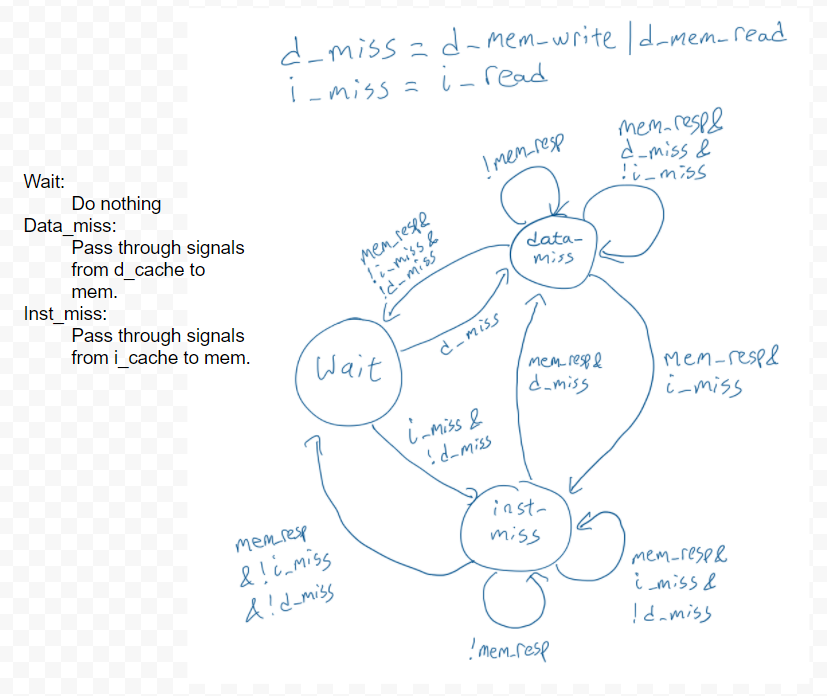


Figure 12. Arbiter FSM

In checkpoint 3, we mainly debugged issues with the pipeline and attempted to implement advanced features. The majority of the time however was spent debugging the pipeline. We encountered several issues with forwarding that prevented the completion of the CP3 testcode. These issues caused us to neglect development of the advanced features, and we were unable to integrate them properly into our processor design.

**Advanced Design Options**

Our Advanced Features did not pan out as ideally as we hoped. Our pipeline was very close to being fully functional, but there were some small, hard to find bugs that we had to spend a lot of time debugging before we could start working on advanced features. One was forgetting to pass forwarded register values to the MEM stage, so in the edge case that there was a store instruction that needed a forwarded value, it was writing the wrong thing into memory. Another was only forwarding the ALU output, when certain instructions like LUI and SLT get their value from the mux in WB, not from the ALU. By the time we got our processor fully functional with all provided testcodes, we struggled to implement the advanced feature modules into our design because of time constraints. However, what follows was our design plan and strategy for integrating Advanced Features.

1. **Branch Predictor**

In Checkpoint 2 and Checkpoint 3, the design goal was to implement a static-not-taken branch predictor, the most basic form of branch prediction. An Advanced Feature option was to design a Tournament Branch Predictor that chose either the Local Branch History Table, or the Global History Table. Leveraging this type of predictor significantly speeds up the processor and wastes less cycles per instruction.

The Local Branch History Table contains both the PC values and the history of the conditional branches. A simple FSM was created with four states: Strongly Not Taken, Weakly Not Taken, Weakly Taken, and Strongly Taken (Figure 13). The FSM uses these states to predict whether or not to take the branch. The states are then transitioned according to whether or not the branch was actually taken. The FSM stores its state in a Pattern History Table (PHT), which is indexed according to the Branch History Register (BHR) (Figure 14). The BHR takes in the actual outcome (br\_en) of each branch. Therefore we had a combination of the branch history and the predicted values to form a final prediction that theoretically has around 80% accuracy.

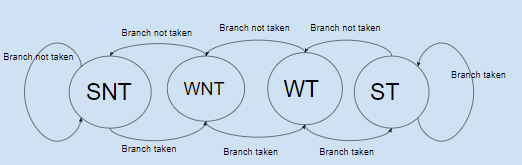


Figure 13. 2-bit Prediction FSM

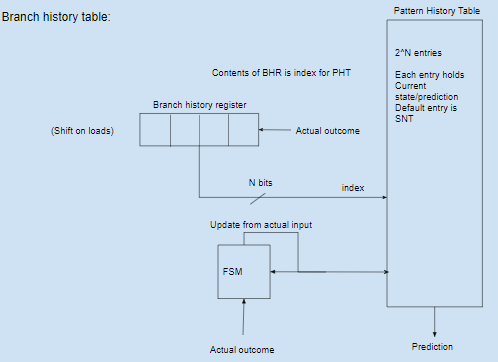


Figure 14. Branch History Register and Pattern History Table FSM

1. **L2 Cache**

The addition of the L2 Cache shortens the critical path in the processor and increases the maximum frequency of the system. Prior to Checkpoint 4, we had a large L1 cache that had to handle one cycle hits. The addition of an L2 cache allows for less physical memory accesses which increases speed on loads and stores. It also allows the L1 cache to be smaller whilst increasing hit frequency. The smaller L1 cache is then faster and the slower, larger L2 cache will make up for cache misses in L1 that occur due to its decreased size. In order to implement the L2 Cache, we change the arbiter to reconcile cache misses from L1 using the L2 cache instead of physical memory.

1. **Eviction Write Buffer**

In order to speed up cache misses, dirty evicted blocks can be relegated to an Eviction Write Buffer (Figure 15). Since cache misses are costly in terms of time, adding an EWB holds the evicted blocks between cache levels and organizes them based on the order of missed addresses. The CPU then can process the missed data much faster.

The EWB is simple to write in SystemVerilog. The more difficult part was integrating the buffer into the current cache architecture. Implementing the flush logic, load signals, and also the addresses with the one cycle hit cache did not show any noticeable improvements. However, if we were able to implement the EWB with our current cache architecture, I would predict it would have significantly sped up the processor because we lost many cycles on cache misses when analyzing our performance metrics.

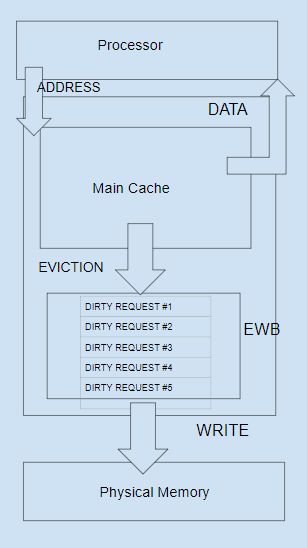


Figure 15. Eviction Write Buffer

**Additional Observations**

When running competition code, the processor achieves a maximum frequency of 100.96 MHz as well as thermal power dissipation and runtimes as seen in Figure 16 and Figure 17. These are baseline metrics, as advanced features were not able to be integrated. Significant slowdown occurs due to cache misses with each miss adding between 340 ns - 590 ns to the runtime. Other slowdowns occur due to branch mispredictions, with each taking 20 ns.

|  |  |
| --- | --- |
| **Testcode** | **Thermal Power Dissipation (mW)** |
| Comp1 | 462.15 mW |
| Comp2 | 453.51 mW |
| Comp3 | 430.60 mW |

Figure 16. Thermal Power Dissipation

|  |  |
| --- | --- |
| **Testcode** | **Runtime** |
| Comp1 | 2,135,337.064ns |
| Comp2 | 6,120,191.656ns |
| Comp3 | 4,332,856.392ns |

Figure 17. Runtime for Competition Codes

**Conclusion**

The objective of this final MP was to implement a pipelined RISC-V Processor with optimizations for speed and power. We were able to implement a working pipelined processor with data forwarding and hazard detection. Due to time constraints and issues with debugging, we were unable to finish implementing advanced features. Through designing and debugging the processor, we learned how to effectively interpret waveforms in ModelSim and how to use ModelSim as an effective debugging tool. However, a downside to this approach was that most of the time was spent stepping through each clock cycle of the waveform. A more efficient way to debug would have been to use RV32I simulators such as Spike to quickly check expected register values and to use the RVFI monitor to check the signals. Even though these problems occurred, creating a processor has led us to a greater understanding of pipelining, caching, and the core concepts of hardware design.