# Final Project Report

**ECE 411**

Pied Piper: 5-Stage Pipelined CPU

**Steven Phan, Megh Shah, Ishaan Patel**

(sphan5, mshah201, ipatel28)

# Table of Contents

## Introduction and Project Overview

Generational technological improvements can only speed up CPUs so much. When this limit is hit, which we are already slowly approaching, CPU performance increases must originate from other sources such as better circuit design techniques, deeper pipelines, and non-traditional designs such as out-of-order CPUs and superscalar CPUs. This report explores one such approach to improving CPU performance: pipelining.

Before discussing pipelining, the traditional approach for implementing the RISC-V instruction set is a 5-cycle multi-cycle CPU. One cycle to fetch the instruction, one to decode, one to execute, one to access memory if needed, and another for writing back to the register file. The RISC-V's instruction set implemented in this project supports memory instructions like load and store, simple arithmetic instructions that can operate on register-register/register-immediate modes, six branch instructions, and two jump instructions.

Our project aims to speed up the 5-cycle multi-cycle design by pipelining the CPU. In essence, the datapath is divided into five separate stages that can concurrently operate on up to 5 different instructions at a time. Each stage's control signals originate from a set of registers, and the output of the logical components in each stage are also registered. By registering the inputs and outputs for each stage and passing these along the datapath, we can isolate the datapath into chunks that can be concurrently run and eventually committed into the register file and reduce critical paths. This implementation allows the CPU to see speedup mainly from a higher throughput (more instructions executing at a given instant in time). Additionally, this project adds to the simple RISC-V instruction set by adding support for RISC-V multiply extension (multiply instruction, divide instruction, and remainder instruction).

We will first discuss how the 5-stage pipeline was implemented: who worked on what part at each project milestone, how each checkpoint was tested, and general design decisions that had to be made when encountering issues in the datapath. Furthermore, beyond the standard pipelined CPU, certain advanced features were added to the datapath that increased design complexity and greatly improved the CPU's performance.

# Milestones

## Milestone 1: Basic RISC-V32 I Datapath

By the end of the first milestone, the datapath for our CPU supported all functionalities required for a basic RISCV-32 I CPU. Support was added for all the instructions except FENCE*, ECALL, EBREAK, and CSRR instructions. As mentioned above in the introduction, the datapath was split into "stages," with each stage pulling data and outputting data into specific "stage registers."

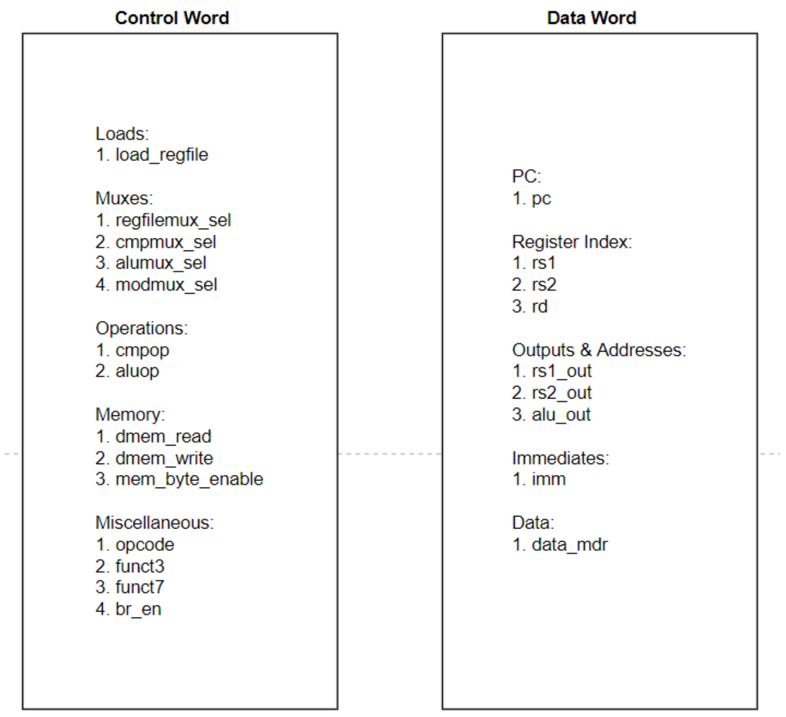


Figure 1.1: Stage Registers

For example, in the instruction fetch stage, some aspects like load_regfile, the select lines, the ops used, dmem_read/write, opcode, funct3, funct7, pc, rs1/rs2, rd, and immediate are set. In contrast, the others are still ambiguous and are decided (if used) at later stages, such as the execute, decode, and memory stages. The five stages are: instruction fetch (IF), instruction decode (ID), execute (EX), memory (MEM), and writeback (WB). With five stages, there are four pipeline stages, each with an input and output stage register denoted by currentStage_nextStage. For example, if an instruction is in the execute stage, the inputs will come from ID_EX and be outputted (registered) to EX_MEM. Please see the figures below for the complete datapath by the end of this milestone (except for the memory interface, our diagrams at the end of this milestone included what we would have done for the memory interface in the next milestone).

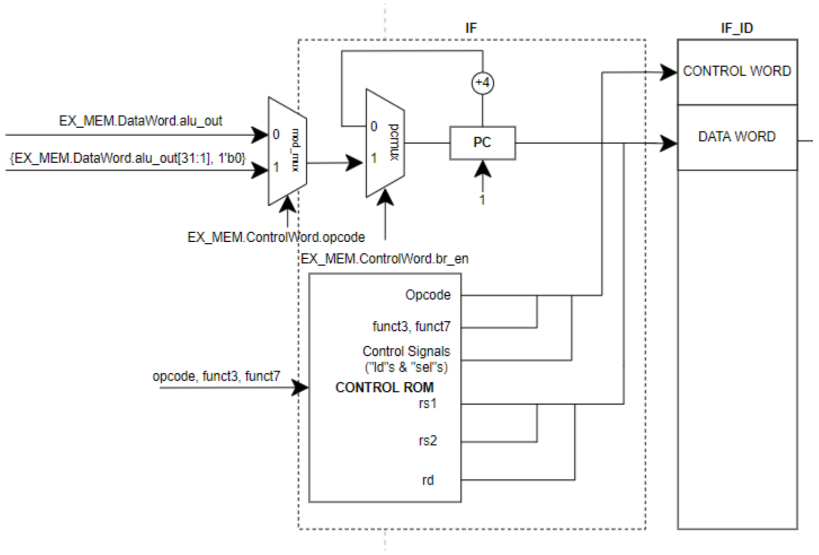


Figure 1.2: Instruction Fetch

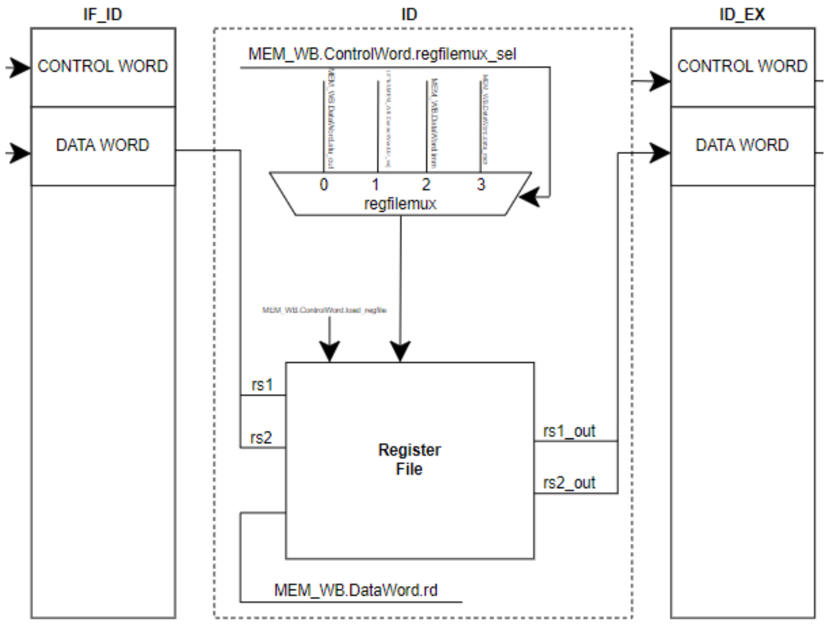


Figure 1.3: Instruction Decode

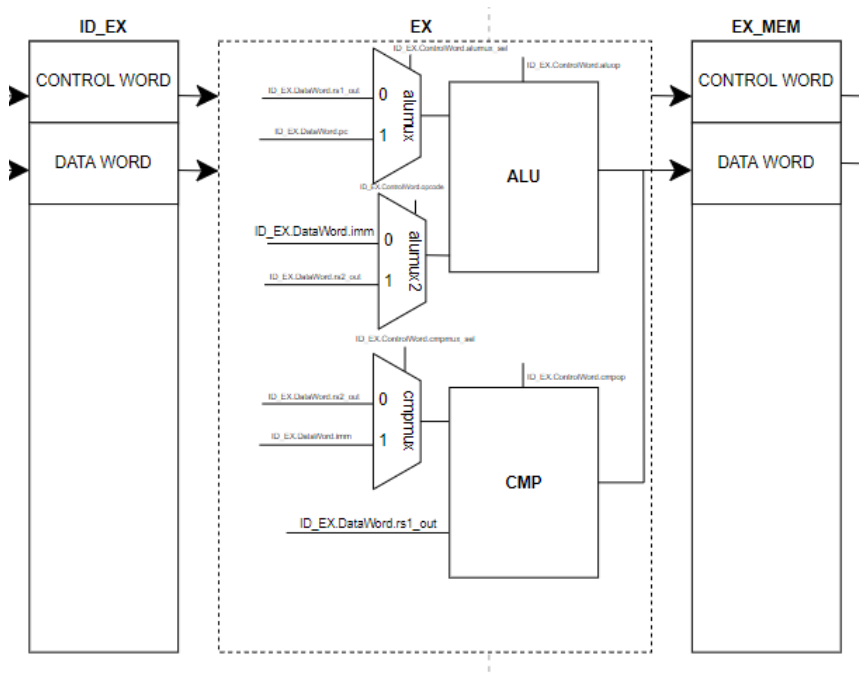
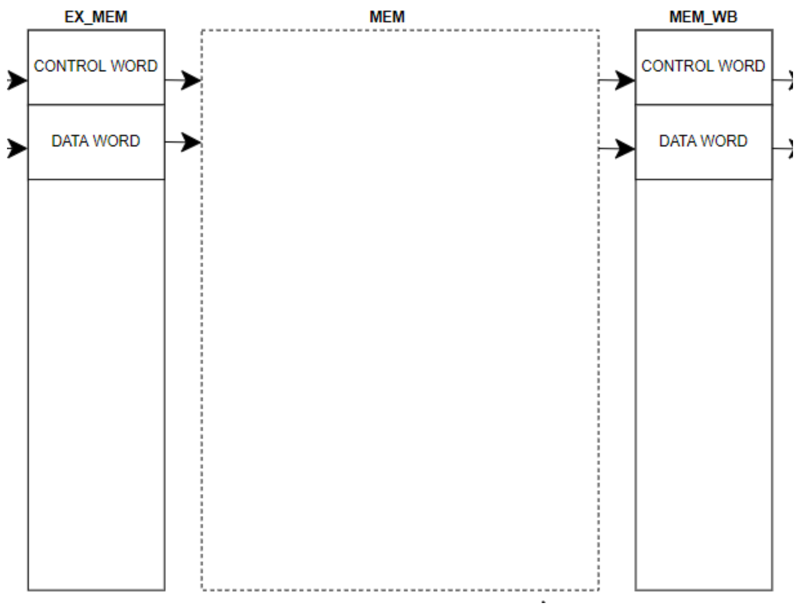


Figure 1.4: Execute



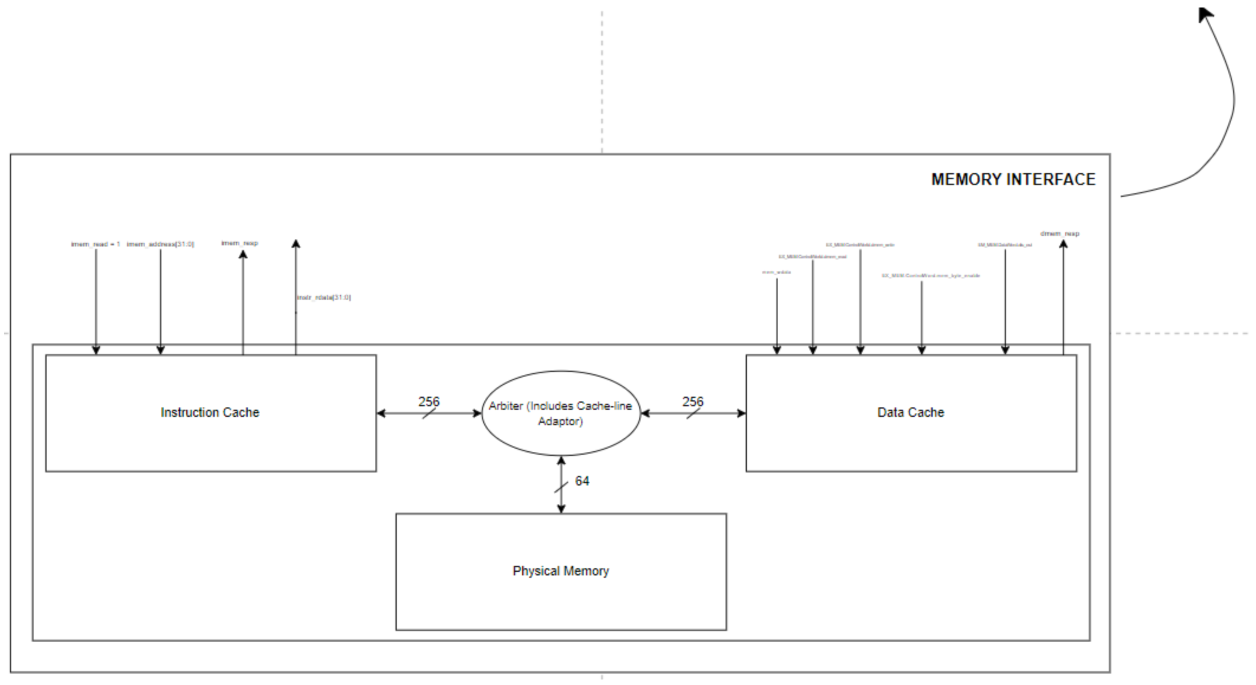


Figure 1.5: Memory Interface

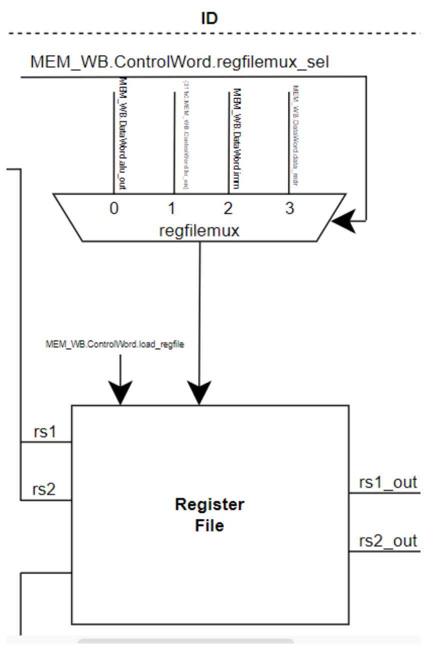


Figure 1.6: Writeback

In terms of who worked on what, we developed every part of the processor together (except for advanced features) because we thought it would be better if we all knew exactly what we were implementing. With this pair (three people) programming style, we could mitigate minor bugs. Furthermore, we made important decisions together much faster than usual with this development style.

As for testing in this milestone, we ran the mp4 checkpoint one code on a CPU we know works (from mp2) and compared the register file values with our outputs. Seeing the mismatches and going through the disassembled code allowed us to narrow down which instructions were faulty, and signal searching in ModelSim allowed us to narrow down the signal(s) causing issues in the datapath. Furthermore, before doing the ModelSim signal search, we would try to recreate specific problems with a smaller set of instructions to verify that the issue we thought was an issue is an issue. Doing this saved us a lot of time debugging something that did not have to be debugged.

## Milestone 2: L1 + Arbiter + Hazards + Forwarding + Static Branch Predictor

After milestone 1, we added L1 caches for both instruction and data memory (separating them instead of having a unified cache and dual-port magic memory like in the previous milestone), an arbiter to handle simultaneous physical memory requests from both instruction and data caches, hazard detection to help detect and prevent memory hazards, data forwarding, and a static non-taken branch predictor.

The way we implemented this was to first start with the memory interface. Our thought process was that since the memory interface is isolated from the rest of the datapath, getting this to work first before implementing and debugging the rest of the datapath would make our lives easier. We used the given cache module. For the data cache, we hooked up the outputs of the EX_MEM register into the cache and the outputs of the cache into MEM_WB. As for the instruction memory, the read signal was held high, the output of the PC was fed into the instruction cache's memory address, and the outputs (instructions) were fed into the control rom in the instruction fetch phase. See Figures 1.5 and 1.2 for more details regarding these connections.

The arbiter is a simple three-state state machine that handles data and instruction cache requests. More specifically, it handles requests to read or write from and to physical memory. There is only one physical memory, but our caches are non-unified, so some mechanism is required to handle concurrent requests. The way it works is that the arbiter starts in an idle state: it waits for a request from either instruction cache or data cache (inst_pmem_read, data_pmem_read, or data_pmem_write). It then transitions to either instr_mem or data_mem based on which signal arrived first and waits in that state until physical memory sends a resp. Upon a resp, instr_mem can transition to data_mem if one of the data_pmem_xxxx signals was raised while in instr_mem.

Similarly, data_mem can transition to instr_mem upon physical memory response and inst_pmem_read being asserted. If none of these cases are met, the arbiter goes back to idle to wait for another request. See the figure below for a more explicit explanation of the states.

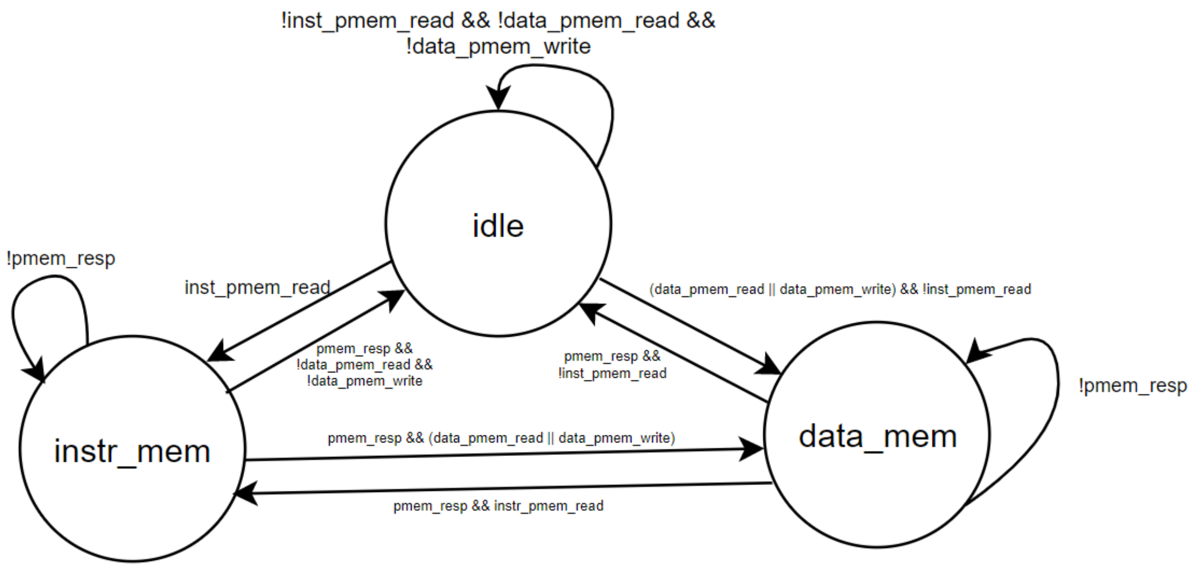
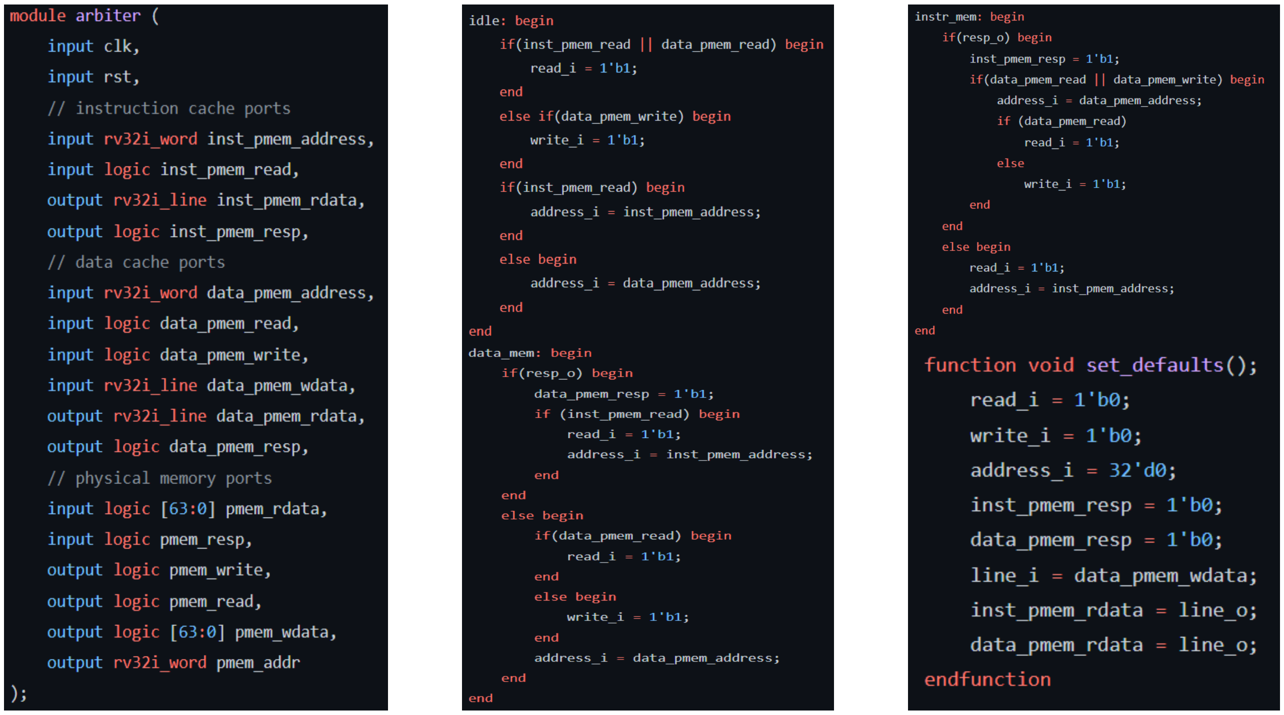

!inst_pmem_read && !data_pmem_read && !data_pmem_write
idle
!pmem_resp
inst_pmem_read
pmem_resp && !data_pmem_read && !data_pmem_write
pmem_resp && !inst_pmem_read
(data_pmem_read || data_pmem_write) && !inst_pmem_read
!pmem_resp
instr_mem
pmem_resp && (data_pmem_read || data_pmem_write)
data_mem
pmem_resp && instr_pmem_read

Figure 2.1: Arbiter State Diagram

Then, depending on which state the arbiter is in, it redirects the signals sent to physical memory from cache and from physical memory to cache.



```
module arbiter (
    input clk,
    input rst,
    // instruction cache ports
    input rv32i_word inst_pmem_address,
    input logic inst_pmem_read,
    output rv32i_line inst_pmem_rdata,
    output logic inst_pmem_resp,
    // data cache ports
    input rv32i_word data_pmem_address,
    input logic data_pmem_read,
    input logic data_pmem_write,
    input rv32i_line data_pmem_wdata,
    output rv32i_line data_pmem_rdata,
    output logic data_pmem_resp,
    // physical memory ports
    input logic [63:0] pmem_rdata,
    input logic pmem_resp,
    output logic pmem_write,
    output logic pmem_read,
    output logic [63:0] pmem_wdata,
    output rv32i_word pmem_addr
);
```

```
idle: begin
    if(inst_pmem_read || data_pmem_read) begin
        read_i = 1'b1;
    end
    else if(data_pmem_write) begin
        write_i = 1'b1;
    end
    if(inst_pmem_read) begin
        address_i = inst_pmem_address;
    end
    else begin
        address_i = data_pmem_address;
    end
end
data_mem: begin
    if(resp_o) begin
        data_pmem_resp = 1'b1;
        if (inst_pmem_read) begin
            read_i = 1'b1;
            address_i = inst_pmem_address;
        end
    end
    else begin
        if(data_pmem_read) begin
            read_i = 1'b1;
        end
        else begin
            write_i = 1'b1;
        end
        address_i = data_pmem_address;
    end
end
```

```
instr_mem: begin
    if(resp_o) begin
        inst_pmem_resp = 1'b1;
        if(data_pmem_read || data_pmem_write) begin
            address_i = data_pmem_address;
            if (data_pmem_read)
                read_i = 1'b1;
            else
                write_i = 1'b1;
        end
    end
    else begin
        read_i = 1'b1;
        address_i = inst_pmem_address;
    end
end
 function void set_defaults();
     read_i = 1'b0;
     write_i = 1'b0;
     address_i = 32'd0;
     inst_pmem_resp = 1'b0;
     data_pmem_resp = 1'b0;
     line_i = data_pmem_wdata;
     inst_pmem_rdata = line_o;
     data_pmem_rdata = line_o;
 endfunction
```

Figure 2.2: Arbiter Signal Redirection

At this point, the way we debugged was running code we know should work: mp4 cp1 code. We followed a similar procedure every time we debug: run code on a CPU we know works, then compare register file values to see what went wrong.

The next feature we implemented was the hazard detection unit. As mentioned earlier, this module should essentially either insert nops (instructions that do nothing) or stall (i.e., set load signals = 0 for certain stages) depending on the situation. For example, loads take multiple clock cycles. There may be instructions dependent on this load, so all the instructions before it in the pipeline must be stalled until the load is complete for correct execution. Similarly, the pipeline cannot expect to insert correct instructions into the pipeline if the instruction cache results in a miss. All these cases are handled in the hazard detection unit. See the figure below for a high-level overview of the inputs and outputs.

Furthermore, we handled the hazard edge cases and incorporated the static not taken branch predictor. Essentially, the pipeline always assumes the branch is not taken and fetches instructions as usual until the branch is computed. By the time it is calculated (the execute stage), if it turns out the branch was taken, then the instructions in the instruction fetch and instruction decode are flushed (reset asserted for those pipeline registers), and the remaining instructions finish execution.

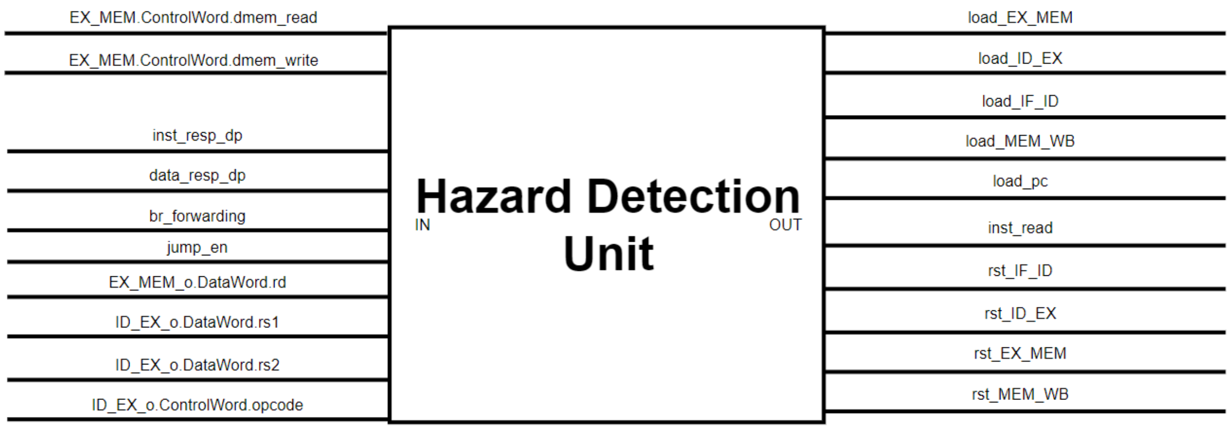
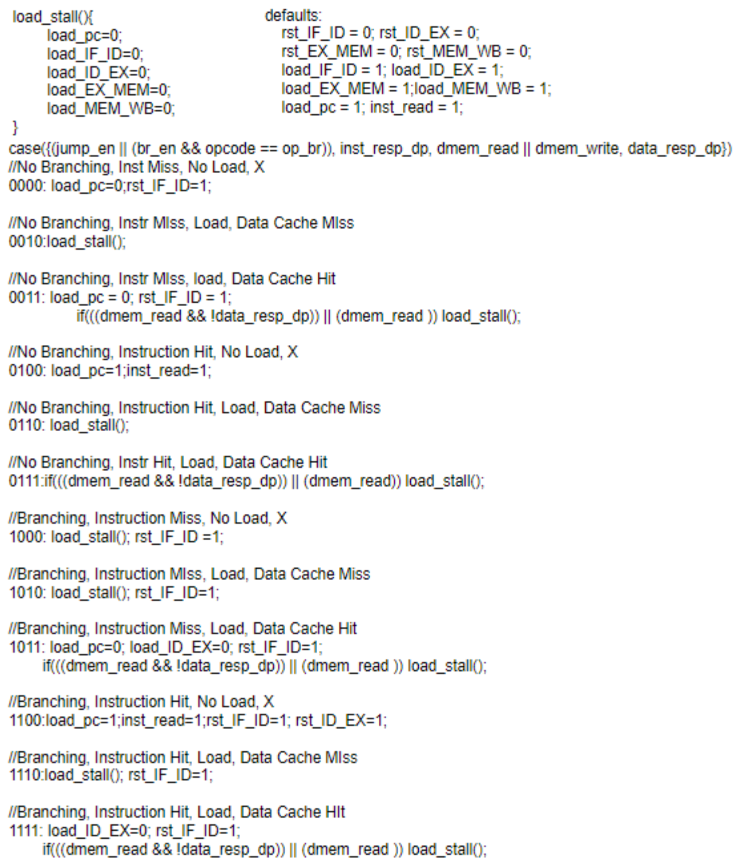


Figure 2.3: High-Level Hazard Detection Unit



```
 load_stall(){                          defaults:
        load_pc=0;                         rst_IF_ID = 0; rst_ID_EX = 0;
        load_IF_ID=0;                      rst_EX_MEM = 0; rst_MEM_WB = 0;
        load_ID_EX=0;                      load_IF_ID = 1; load_ID_EX = 1;
        load_EX_MEM=0;                     load_EX_MEM = 1;load_MEM_WB = 1;
        load_MEM_WB=0;                     load_pc = 1; inst_read = 1;
 }
case({(jump_en || (br_en && opcode == op_br)), inst_resp_dp, dmem_read || dmem_write, data_resp_dp})
//No Branching, Inst Miss, No Load, X
0000: load_pc=0;rst_IF_ID=1;

//No Branching, Instr MIss, Load, Data Cache MIss
0010:load_stall();

//No Branching, Instr MIss, load, Data Cache Hit
0011: load_pc = 0; rst_IF_ID = 1;
            if(((dmem_read && !data_resp_dp)) || (dmem_read )) load_stall();

//No Branching, Instruction Hit, No Load, X
0100: load_pc=1;inst_read=1;

//No Branching, Instruction Hit, Load, Data Cache Miss
0110: load_stall();

//No Branching, Instr Hit, Load, Data Cache Hit
0111:if(((dmem_read && !data_resp_dp)) || (dmem_read)) load_stall();

//Branching, Instruction Miss, No Load, X
1000: load_stall(); rst_IF_ID =1;

//Branching, Instruction MIss, Load, Data Cache Miss
1010: load_stall(); rst_IF_ID=1;

//Branching, Instruction Miss, Load, Data Cache Hit
1011: load_pc=0; load_ID_EX=0; rst_IF_ID=1;
      if(((dmem_read && !data_resp_dp)) || (dmem_read )) load_stall();

//Branching, Instruction Hit, No Load, X
1100:load_pc=1;inst_read=1;rst_IF_ID=1; rst_ID_EX=1;

//Branching, Instruction Hit, Load, Data Cache MIss
1110:load_stall(); rst_IF_ID=1;

//Branching, Instruction Hit, Load, Data Cache HIt
1111: load_ID_EX=0; rst_IF_ID=1;
      if(((dmem_read && !data_resp_dp)) || (dmem_read )) load_stall();
```

Figure 2.4: Hazard Detection Cases

Then, the only remaining component to implement was the forwarding unit. The way the forwarding unit works is that it checks if an instruction in the execute or writeback stage will write back to the register file (load_regfile is asserted). If so, it checks the destination register (rd). If rd matches any of the sources of the instruction in the execute stage, it will pass the appropriate data back (i.e., data_mdr that originates from memory, br_en that stores the comparison instructions, or alu_out for all other instructions). If there’s a double dependency (a match in both the writeback and the memory stage), it will prioritize the memory stage dependency since it is more recent than the writeback dependency. At this point, we introduced many more muxes to account for all the priority cases and the forwarded data.

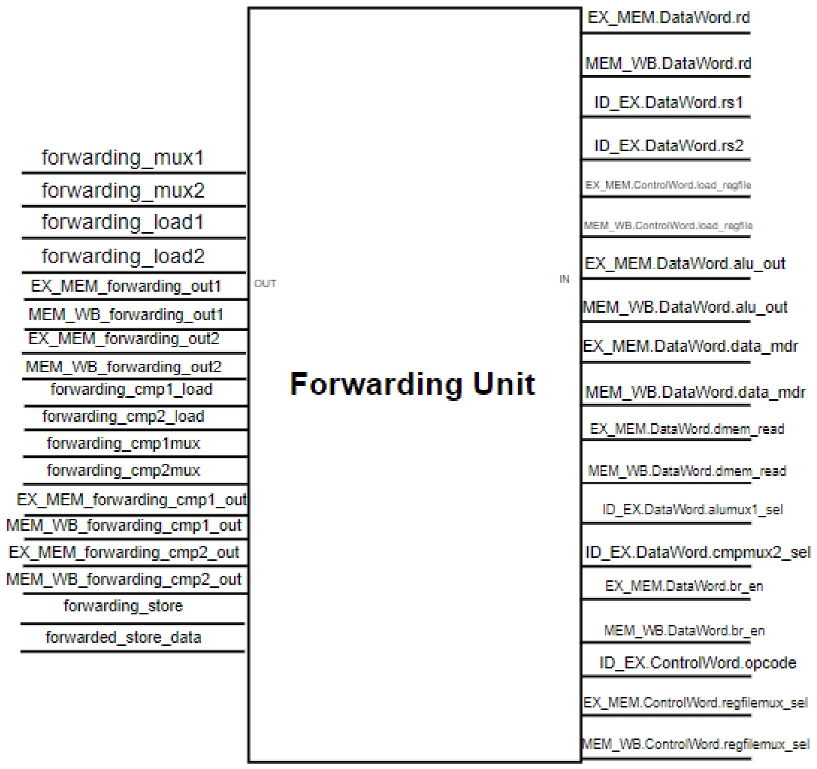


Figure 2.5: High-Level Forwarding Unit

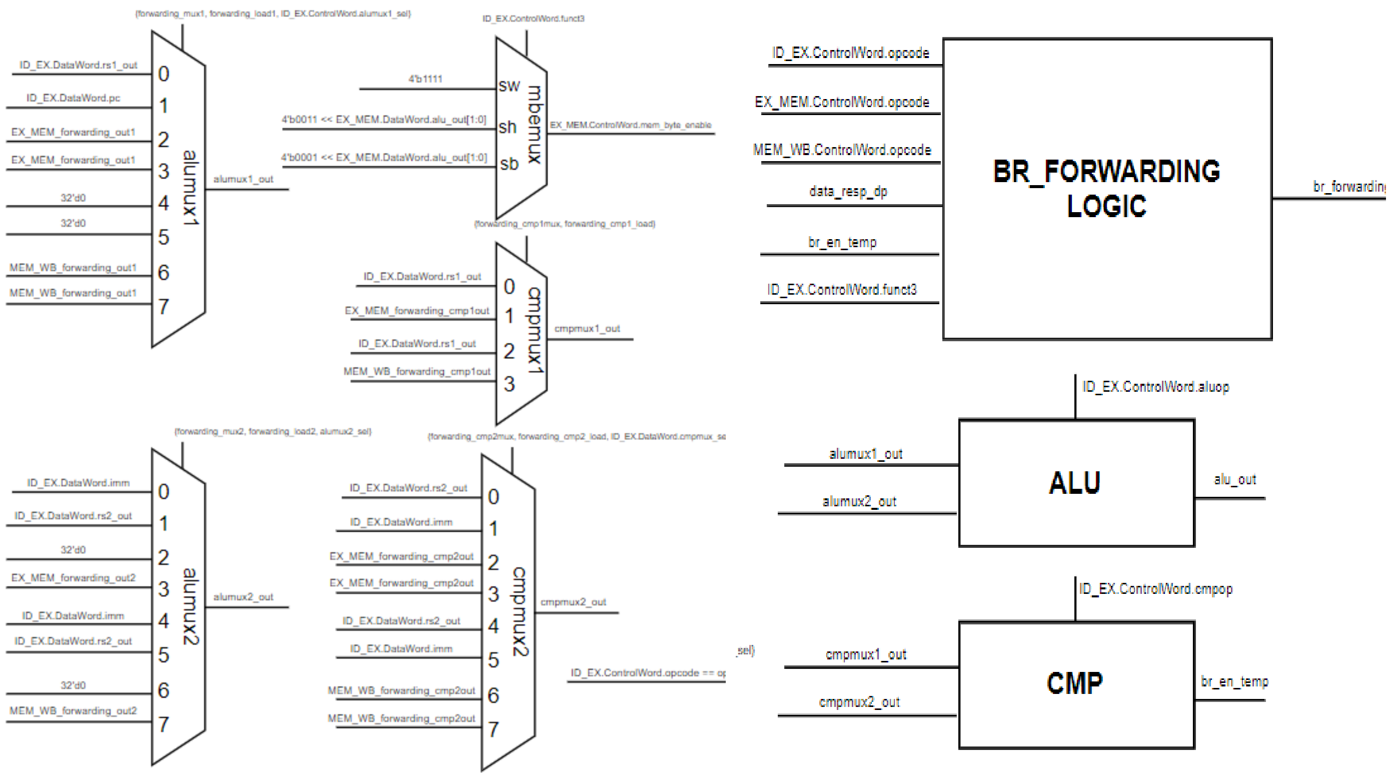


Figure 2.6: Execute Stage With Forwarding

With both the forwarding unit and hazard detection units implemented, we began debugging with the usual procedure but on mp4 cp2 code. Before running cp2, however, we wrote a piece of assembly code that would not cause hazards but only test forwarding. We did this by checking all the instructions that could forward data. We considered every combination of dependencies we could think of (rs1 vs. rs2 dependencies, duplicate dependencies in memory and writeback stage, and immediate forwarding). Once we got our miniature test codes to work, running and debugging cp2 code was smooth.

## Milestone 3: Advanced Design Features

This milestone is where we split up the work because there were so many independent features to implement. Ishaan designed, tested, and debugged all cache-related features (L2, parameterized sets, and 4 way set associative). Steven was in charge of the prefetching block for the instruction cache, and the eviction write buffer for the data cache. Megh was in charge of the RISC-V M extension.

The cache design is between the CPU and arbiter for both the instruction and data. The L1 cache, in particular, is between the CPU and L2 cache. The L2 cache is between the L1 cache and the arbiter. We tested the cache design by running the mp codes and checking the waveform to see if the behavior was correct. We looked for a few things: the four ways in the cache are appropriately used, the resp signals are being asserted at the right time, and the pseudo-lru policy is being used correctly. The diagram below shows how we hooked the caches from L1 to L2. You also will see the pseudo-lru policy that we followed.

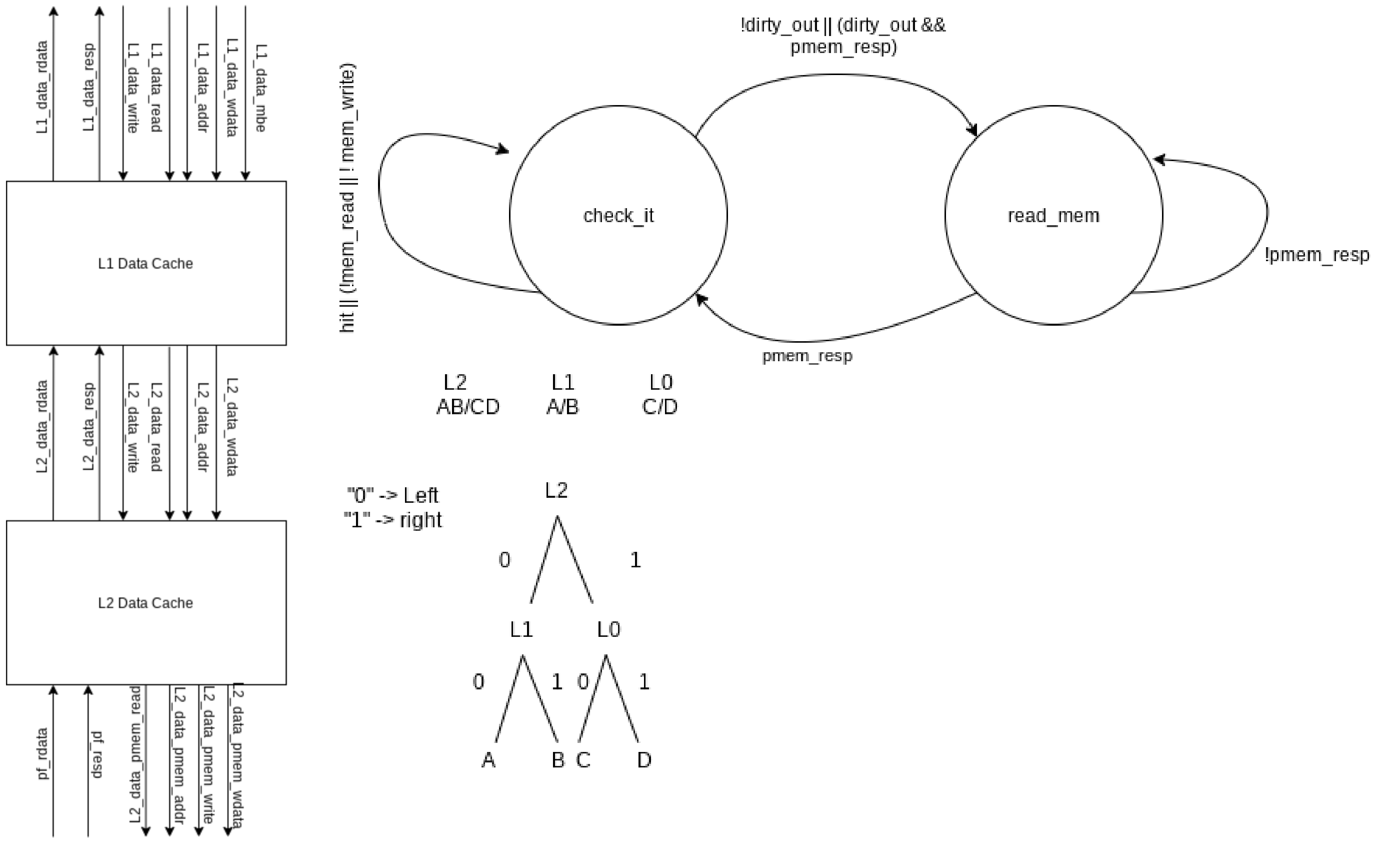


Figure 3.1: Cache Structure and Design

The pseudo-lru policy is precisely the same as the one we learned in class where the bits represent the most recently used way, and when we want to replace the least recently used, we would replace the way that is the “not” of our lru bits. For example, suppose we had the bits be 000. In that case, that means that A was the most recently used, and the way we would replace is D. The diagram above also shows the state machine used for the cache, and we decided to use the state machine that was given to us since it was much faster than the cache I built for MP3.

The given state machine is faster because it has only two states, while the cache we implemented had four.

Although it would not have been hard to implement this for both data and instruction caches, the prefetching block was added to handle just instruction prefetching. Our group decided not to have the hardware prefetcher for both instruction and data due to power consumption and possible contention within the arbiter having to handle too many prefetch requests and actual missed requests. The prefetcher lies in between the last level instruction cache and the arbiter, as seen in the figure below.

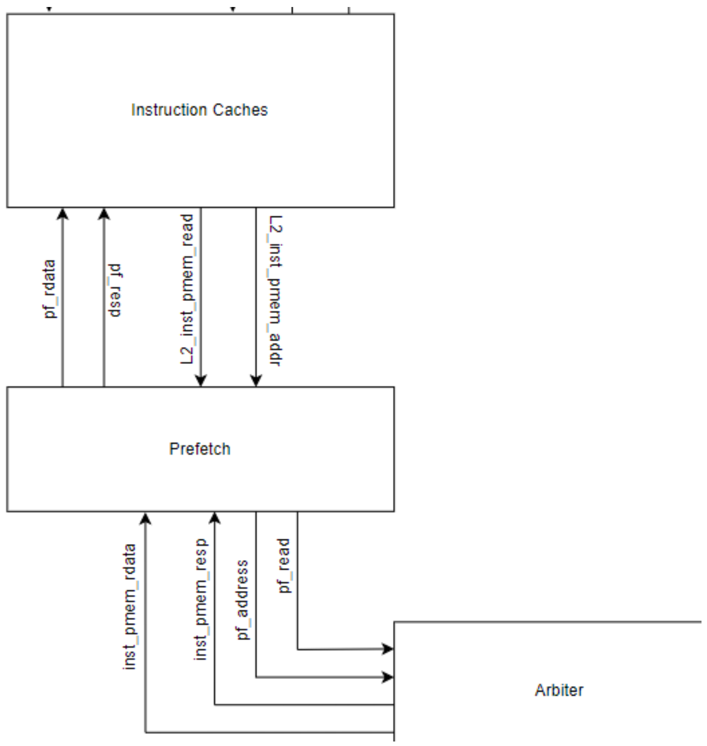


Figure 3.2: Prefetching Block

The way the prefetching is implemented is through a three-state state machine. The states are idle, read_missed, and read_prefetched. When in the idle state, every time the last-level instruction cache asserts inst_pmem_read, the prefetching block buffers the following cache line address (the address sent via inst_pmem_addr + 32) and transitions to read_miss. In this state, it performs the missed read and waits for the arbiter to send a response back. Once this response is received, the prefetching then goes to read_prefetch where it initiates a read to the arbiter with the prefetch address in hopes that by the time the next miss from the last level instruction cache occurs, the address matches the prefetched address, and we have already started the read, which saves a significant amount of clock cycles. Once read_prefetch is done, a flag is set to signify it is done so that in the idle state, it can check to see if the missed address matches the prefetched address. If so, pass the data back to the instruction cache only if the flag was set (i.e., the data is valid). Please see the figures below for a more detailed state diagram and control logic.

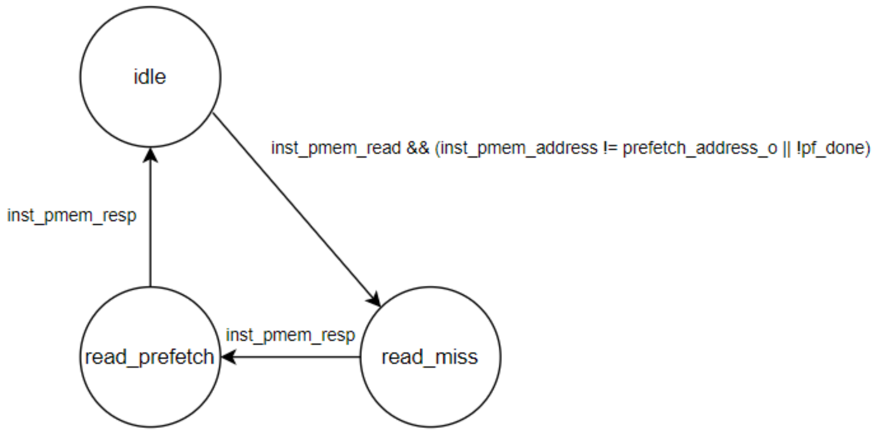
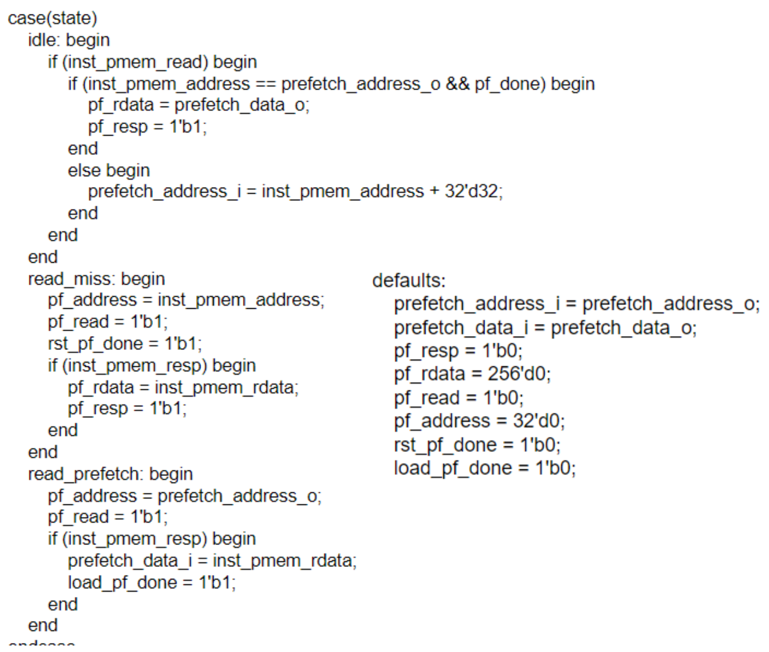

idle
inst_pmem_read && (inst_pmem_address != prefetch_address_o || !pf_done)
inst_pmem_resp
inst_pmem_resp
read_prefetch
read_miss

Figure 3.3: Prefetch State Diagram



```
case(state)
  idle: begin
    if (inst_pmem_read) begin
      if (inst_pmem_address == prefetch_address_o && pf_done) begin
        pf_rdata = prefetch_data_o;
        pf_resp = 1'b1;
      end
      else begin
        prefetch_address_i = inst_pmem_address + 32'd32;
      end
    end
  end
  read_miss: begin
    pf_address = inst_pmem_address;
    pf_read = 1'b1;
    rst_pf_done = 1'b1;
    if (inst_pmem_resp) begin
      pf_rdata = inst_pmem_rdata;
      pf_resp = 1'b1;
    end
  end
  read_prefetch: begin
    pf_address = prefetch_address_o;
    pf_read = 1'b1;
    if (inst_pmem_resp) begin
      prefetch_data_i = inst_pmem_rdata;
      load_pf_done = 1'b1;
    end
  end
```

```
defaults:
  prefetch_address_i = prefetch_address_o;
  prefetch_data_i = prefetch_data_o;
  pf_resp = 1'b0;
  pf_rdata = 256'd0;
  pf_read = 1'b0;
  pf_address = 32'd0;
  rst_pf_done = 1'b0;
  load_pf_done = 1'b0;
```

Figure 3.4: Prefetch Controls

This module was tested by writing small pieces of code and verifying through ModelSim that the prefetched address and the prefetched data were correct every time an inst_pmem_read was asserted. This data matching was done by looking at the binary file outputted by rv_load_memory.sh and figuring out where each cache line address was. Once that portion was verified, running it with all the working codes ensured that the prefetching block did not break the CPU.

The next feature is the eviction buffer. This buffer needs to be only between the last level data cache and the arbiter. This is because the instruction cache never does a dirty writeback to the main memory, so there is no point in adding it between the last-level instruction cache and the arbiter, which would only complicate the design since prefetching already exists in that area of the datapath. See the figure below for the updated memory interface, including the eviction buffer and prefetching.

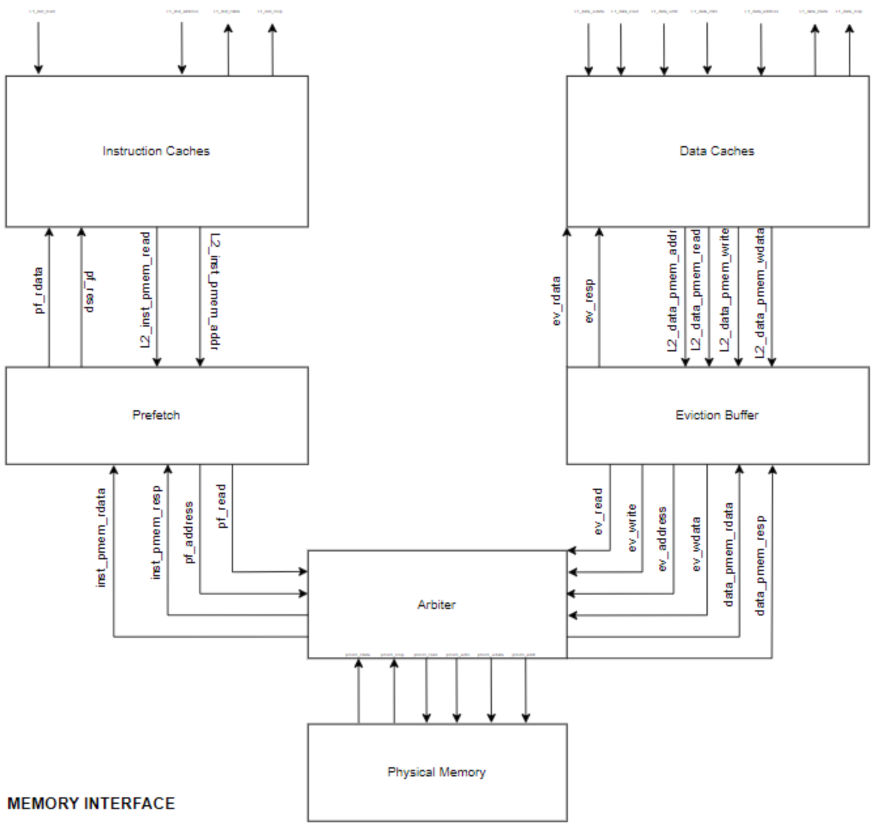


Figure 3.5: Updated Memory Interface

The eviction buffer was designed similarly to the prefetching block. There are three states: idle, read_miss, and writeback. The idle state waits for a data_pmem_write (which indicates the last-level data cache evicting dirty data), buffers the address and data, and sets a writeback flag. The response is also sent back to the data cache to transition to its internal state, which initiates the physical memory read. The idle state waits for data_pmem_read to be asserted and transitions to read_miss, which performs the missed read that caused the eviction. When the physical read is done, it transitions to the writeback state to send the buffered address and data to physical memory if the writeback flag is set. If not, the state machine transitions back to the idle state. Please see the figure below for more details.

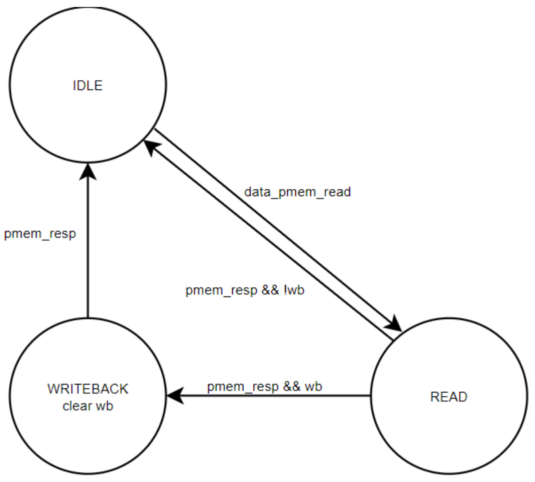
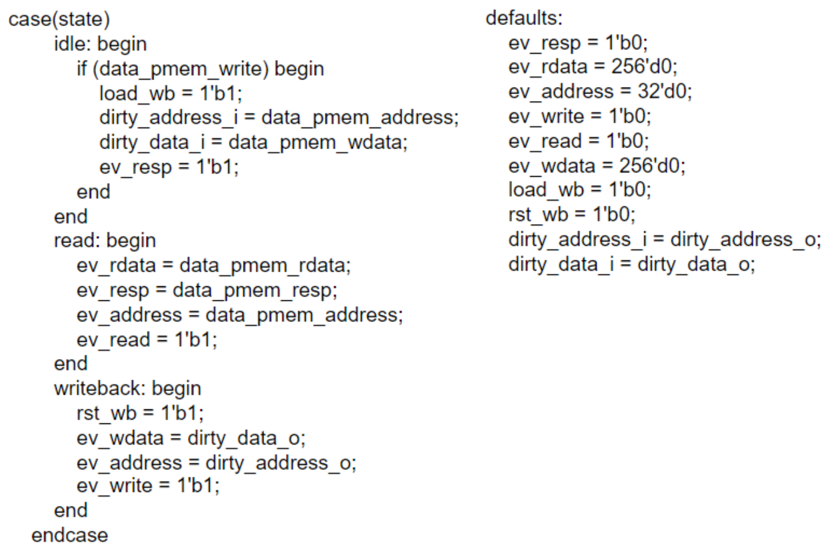


Figure 3.6: Eviction Writeback Buffer State Diagram



```
case(state)
      idle: begin
         if (data_pmem_write) begin
            load_wb = 1'b1;
            dirty_address_i = data_pmem_address;
            dirty_data_i = data_pmem_wdata;
            ev_resp = 1'b1;
         end
      end
      read: begin
         ev_rdata = data_pmem_rdata;
         ev_resp = data_pmem_resp;
         ev_address = data_pmem_address;
         ev_read = 1'b1;
      end
      writeback: begin
         rst_wb = 1'b1;
         ev_wdata = dirty_data_o;
         ev_address = dirty_address_o;
         ev_write = 1'b1;
      end
   endcase
```

```
defaults:
   ev_resp = 1'b0;
   ev_rdata = 256'd0;
   ev_address = 32'd0;
   ev_write = 1'b0;
   ev_read = 1'b0;
   ev_wdata = 256'd0;
   load_wb = 1'b0;
   rst_wb = 1'b0;
   dirty_address_i = dirty_address_o;
   dirty_data_i = dirty_data_o;
```

Figure 3.7: Eviction Writeback Buffer Controls

This module was tested in a similar way to the prefetching testing. To test this eviction buffer is to essentially fill up a set in a cache, make one of the ways dirty, and evict the dirty way. This was simulated by writing test code that fills up set zero, makes one of the ways in set zero dirty, and performs a sequence of reads to make sure this way becomes the least recently used, then loads an address that maps to set zero that is not cached. Then, we read the evicted address and ensured the data was the same before and after the eviction. Once these tests worked, running it against test code that already worked caused little to no memorable issues.

The RISC-V M extension is an advanced design feature that we chose to implement due to its promising performance gains. To begin the implementation of this feature, we had to first research and decide the different multiplication and division algorithms that we would prefer our design to use. After looking at several algorithms and techniques, such as Wallace Tree and Karatsuba, we decided to implement a simple add-shift and sub-shift methods for multiplication and division due to their simplicity and less power consumption. Regarding the actual design of our M extension, the design is split into three main components. The main muldiv module, a multiplier module, and a division module. The muldiv module is placed in the execute stage and acts as a top-level module that takes in the respective inputs from the outputs of the ALU muxes. Our decision to hook the data inputs of the muldiv module to the output of the ALU muxes was to prevent any additional changes to our forwarding unit.

Since the ALU muxes output the correct forwarded data, by having this output go into the muldiv, we are essentially ensuring that forwarding will work with the M extension without any additional changes. Furthermore, to account for the output, we hooked the output of the muldiv module to the alu_out portion of our EX_MEM pipeline register to ensure normal functionality. In addition to the data inputs, the muldiv module takes in a start signal, which is determined when the instruction in the execute stage is a mul, div, or rem type instruction. The muldiv module will only begin if the start signal is high to prevent excess power consumption. Furthermore, in addition to the resulting output, the muldiv module also sends a response signal to the hazard detection unit to stop stalling since the M extension instructions take multiple clock cycles.

Delving deeper into the muldiv module, it contains the multiplier and division module. Muldiv acts as a passthrough module – meaning that it takes in the inputs and simply passes them to the correct execution module. Once the multiplier or divider finishes execution, it returns the result. It sets the response signal to high to indicate to the muldiv module that it should notify the hazard detection unit to stop the pipeline stall. The multiplier and division module itself serves as a three-stage module. The multiplier module adds the multiplier to the multiplicand a respective number of times. The divider module subtracts the divisor from the dividend until it no longer needs to do so.

Regarding negative values, before the muldiv module passes the inputs into the execution modules, it checks how many are negative. It keeps track of this value and converts all inputs to positive before handing it off to the correct execution unit if needed. Once the muldiv module gets the result, it checks how many of the inputs were negative. If this value is even, it keeps the result positive. If odd, it will flip the bits and add one to convert the positive result to negative before passing it back to the processor. Below is a figure representing the general design of the M extension implementation.

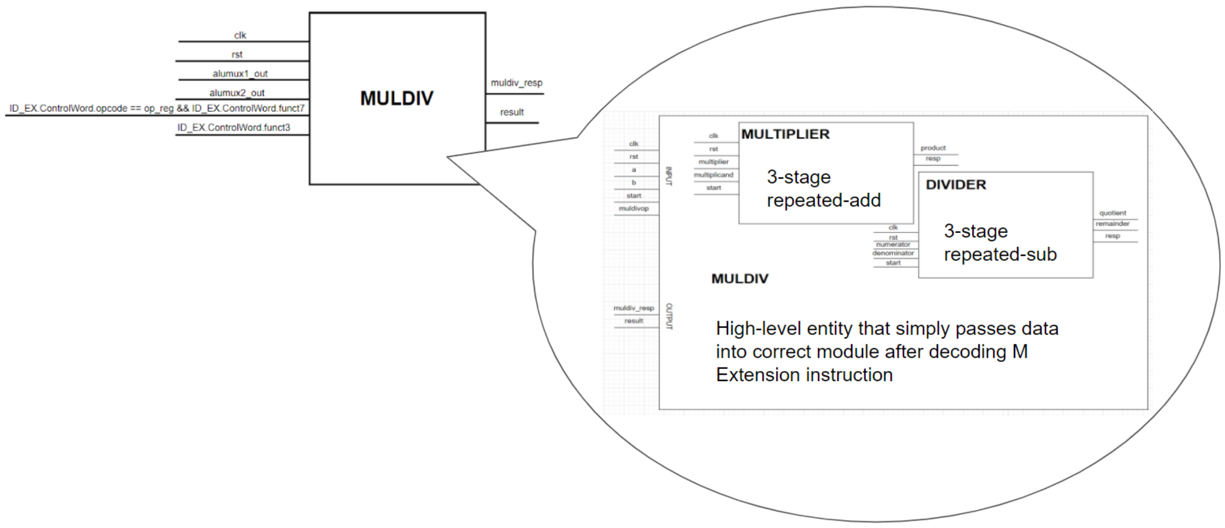


Figure 3.8: RISC-V M Extension Design (Component in Execute Stage)

## <u>Advanced Design Features</u>

### <u>Advanced Design Feature #1: L2 Caches + Parameterized Sets + 4 Way Set Associativity</u>

The given MP4 cache had one way and eight sets. Due to this configuration, we knew that one of the advanced features that needed to be upgraded was cache since we suspected that would give us a considerable performance boost. The first thing to do to the cache was to parameterize the sets and then expand the number of ways to 4. After testing out a 4-way, parameterized L1 cache, we expanded the cache to the second level. The chart below shows that our theory was correct, and we were able to cut the execution time significantly for the competition codes. Since the competition code used branches extensively that would write to the same memory address, a 4-way cache can increase the chances of a cache hit. This can be shown best in comp2_i.s, where we had 135,284 hits in our first level for instruction cache and 7090 hits for data. While a faster execution time was achieved, it did come at the cost of increased design space. We were also fortunate to have a workload that would utilize the cache extensively; however, if the workload did not utilize temporal or spatial locality effectively, our cache would not help and, in some cases, give worse performance due to the additional state machines.

| Counter/Competition | comp1.s | comp2_i.s | comp3.s |
|---|---|---|---|
| Counter (# used) | L1 Inst:<br>Hit:<br>57796<br>Miss:<br>25<br>L2 Inst:<br>Hit:<br>57821<br>Miss:<br>25<br>L1 Data:<br>Hit:<br>3845<br>Miss:<br>7<br>L2 Data:<br>Hit:<br>3852<br>Miss:<br>7 | L1 Inst:<br>Hit:<br>135284<br>Miss:<br>76<br>L2 Inst:<br>Hit:<br>135360<br>Miss:<br>76<br>L1 Data:<br>Hit:<br>7090<br>Miss:<br>24<br>L2 Data:<br>Hit:<br>17129<br>Miss:<br>24 | L1 Inst:<br>Hit:<br>66018<br>Miss:<br>33<br>L2 Inst:<br>Hit:<br>66051<br>Miss:<br>33<br>L1 Data:<br>Hit:<br>13410<br>Miss:<br>283<br>L2 Data:<br>Hit:<br>13693<br>Miss:<br>283 |
| Execution Time (ns) | 588,495 ns | 1,390,715 ns | 675,185 ns |

Figure 4.1: Cache Usage and Performance

The performance listed in the table above is for an 8-set, 4-way L1 and L2 cache. We experimented with changing the number of sets to 16 and removing the L2 cache, and both design choices didn't provide any significant performance boost. Due to this, we decided to do 8-set and 4-ways.

## Advanced Design Feature #2: Basic Hardware Prefetching

The milestone #3 section mentioned that the basic hardware prefetcher expects performance benefits when the prefetching address matches the next missed cache line address. The tradeoffs, in this case, are scenario-dependent: cycles are saved from fetching missed data if that actual prefetched address matches the requested missed address. In this case, the CPU saves several clock cycles by initiating the read ahead of time to prefetch the (assumed to be) correct data. Furthermore, additional logic elements always contribute to overall power consumption if no special energy consumption logic is added. The issue with prefetching is that when code has branch instructions (which is one of the most frequent), prefetching without some sort of predictor hurts performance because the prefetcher will read irrelevant data. This performance degradation is evident through the tables below.

| Counter/Competition | comp1.s | comp2_i.s | comp3.s |
|---|---|---|---|
| Counter (# used) | 4,105 | 10,817 | 8,003 |
| Execution Time (ns) | 3.597 million | 9.775 million | 8.101 million |

Figure 4.2: Prefetch Usage and Runtime

These runtimes are significantly worse than the baseline runtimes (no advanced features) for all the competition codes. The issue is that the prefetcher is wasting time fetching something that should not be fetched due to our lack of a branch predictor or some sort of prefetch predictor. The prefetcher (with no predictor) would be useful under sequential chunks of code with little to no branch instructions (handling large iterative loops) and performs horribly under frequent branching.

## Advanced Design Feature #3: Eviction Write Buffer

As discussed in the milestone #3 section, the eviction write buffer helps with handling evictions quicker. When a dirty writeback occurs, the traditional flow is to wait for the writeback and then wait for the miss read that caused the writeback to be serviced. However, the writeback essentially mitigates the wait time and immediately services the miss read so that the CPU can progress without waiting. While the CPU is executing and physical memory is free, the buffer does the writeback asynchronously. The main benefit comes from reducing wait time for pieces of code that cause a lot of writebacks; however, this extra layer of logic introduces more power consumption and a longer critical path which slows down our clock speed.

| Counter/Competition | comp1.s | comp2_i.s | comp3.s |
|---|---|---|---|
| Counter (# used) | 7 | 260 | 728 |
| Execution Time (ns) | 2.189 million | 6.205 million | 4.151 million |

Figure 4.3: Eviction Buffer Usage and Runtime

While the CPU does see general speedups, we noticed that our clock speed was not great with both L2 caches and the eviction buffer. Additionally, there are certain cases where the eviction buffer would hurt performance. The design immediately writes back dirty data after a physical memory read; however, what would happen if a requested read address matches the address of the data being written back? Our implementation would need the writeback to finish and then initiate a read. We would have taken this into account with more time and more careful consideration by adding an extra check in the writeback state.

## Advanced Design Feature #4: RISC-V M Extension

Upon implementation, we began testing the performance of the M extension design. We saw a significant difference when comparing comp2_i.s code with comp2_m.s code, as the M extension design decreased our runtime by nearly nine times. Our original design ran the code in a little over six million nanoseconds, whereas the M extension design completed the code in just over seven hundred thousand nanoseconds. Our performance counter also indicated that the muldiv module was used over seventeen hundred times. The figure below demonstrates our performance, speedup increase, and counter values.

| Counter/Competition | comp2_i.s | comp2_m.s |
|---|---|---|
| Counter (# used) | 0 | 1,733 |
| Execution Time (ns) | 6.214 million | 700,565 |

Figure 4.3: RISC-V M Extension Usage and Runtime

This performance increase was not only because our design can support the M extension instructions but also because the comp2_m.s code contains fewer instructions. In the original code, since no M extension instructions were supported, the code would contain repeated add instructions to execute an instruction such as multiply. This leads to a greater code size and more instructions that need to be fetched, decoded, and executed. The M extension code reduces all of these aspects as we can replace the repeated instructions with the respective M extension instruction leading to smaller code size, less use of processor resources, and fewer instructions in the pipeline overall. Branching off this, the M extension advanced feature is an excellent design implementation as now the design can now support multiply, divide, and remainders. This means that arithmetic-based workloads would run much faster on our design, and even for normal programs, such as programs that do not contain the M extension code, the M extension modules would not interfere. Thus our design would operate as normal.

## **Conclusion**

By the end of this project, our group managed to significantly speed up our original multi-cycle CPU when it comes to performance under competition code workloads. By changing from a multi-cycle dichotomy to a pipeline structure, we significantly increased the instruction throughput. We also introduced advanced features that sped up memory response times and instruction execution time (the RISC-V M extension). These additional features helped reduce stall time when dealing with hazards and greatly improved cache efficiency and memory efficiency. While these advanced techniques may be more complex, they are essential when it comes to CPU designs due to the slowing down of technological scaling.