ECE 411

Spring 2020

Machine Problem 3 Final Report

A Pipelined Implementation of the RV32I Processor

Adit Umakanth(adityau2), Zhifeng Ou(zo2), Hyun Do Jung(hjung35)

ItWillWorkProcessor

Jian Huang, Sean Ngo

I. Introduction:

Building a basic ISA is never an imaginable idea for us to think of easily as students. MP3 design is a team-based project that should build up a basic RISC-V pipelined datapath CPU with a level-2 or above cache hierarchy, and a few selective advanced features along with basic functions already implemented in previous MPs. This report includes the whole timeline of the project, and specific functions we have focused on and their background idea, with true reflection and our opinion about the project itself with advice toward future students.

II. Project Overview:

This project is to build a basic RISC-V ISA with a basic L1 and L2 cache hierarchy using a basic pipelined datapath. Our goal here was to finish all the basic features that should be implemented by each given checkpoint and have a few advanced features on top of it such that the project can somewhat hit the competition baseline. The reason behind we tried to focus on basic functionality is mainly because we thought eventually working processor that we should make a result from this MP. As a result, the majority of basic and advanced features that we focused to implement were successfully implemented at least by the final due date.

III. Milestones:

Design description: Our project is implemented with a basic cache frame and memory handling that are mainly sourced from previous. On top of that, pipelining CPU design was the first one that we had to deal with by making a regular CPU into 5 stages as IF, ID, EXE, MEM, and WB stages. Two-level cache hierarchy contains two L1 caches specifically for data and instruction separately. An L2 cache similar to the L1 data-cache was added to the cache hierarchy along with the arbiter, mainly a combinational logic which decides what to choose for specific instructions to be fetched. A data forwarding unit, static-branch predictor, and a data hazard detection system was also added. More details will be described in the sections below.

Checkpoint one. First checkpoint is implementing the rudimentary pipelined processor that supports the fundamental instructions of the RISC-V ISA. Following instructions are supported by this processor at this checkpoint: LUI, AUIPC, LW, SW, ADDI, XORI, ORI, ANDI, SLLI, SRLI, ADD, SLL, XOR, SRL, OR, BEQ, BNE, BLT, BGE, BLTU, BGEU, and AND. The pipelined processor is composed of total five stages: fetch (IF), decode (ID), execute (EX), memory (MEM), and writeback (WB), and stage registers in between these stages serving as an internal register that stores intermediate values of data calculation of instruction operations. The design part was referred back to our textbook design and it firstly gives us a bit of confusion since MIPS has different architecture than ours, but we get through with other resources. ISA support was run through individual unit tests in order to check the functionality of each instruction, and with a provided test code for this checkpoint and a signal monitoring using Modelsim, we could see all functionality works as we expected.

Each stage is having an individual module itself with intermediate registers between modules serving as stage divisions in the pipeline. Figures 1 is the diagram of the five-stage pipeline we have designed. Design choices such as instruction splitter and efficient Control ROM made at this checkpoint was organized from the consideration of the efficiency of our design.

At conclusion, we recognized our need for our future work to prioritize the full support of the RISC-V ISA along with making a solid interaction between pipelined CPU and physical memory.

Checkpoint two. In this checkpoint, we had to further improve our RV32I pipelined datapath by supporting all of RISC-V ISA excluding FENCE, ECALL, EBREAK, and CSRR. To do this, we have to make a little change to our datapath. Also, considering the theoretical and expected latency from our design's interaction with physical memory, we began the implementation of a separate, one-level cache hierarchy system. Testing the cache system consisted of close attention to the cache contents during memory interaction using RVFI monitor and Modelsim.

This cache system in detail includes two 8-set, 2-way L1 caches, one for storing instruction lines and the other for storing data. The design of the L1 cache may be observed in figure 2. Each cache line holds 256 bits of data, allowing for 256-bit reads from and writes to physical memory that here, 32 bits of information may be sent to and from the pipeline. The two L1 caches, respectively named I-cache and D-cache, are working through a controller unit called an arbiter module. This module would decide which cache to write pipeline data to based on read/write signals sent from the pipeline. Instruction fetching was prioritized.

The theoretical latency should be existing throughout test cases because of the high number of memory access that will be dealt better with the addition of the L1 I-cache and D-cache. In order to improve upon this, an unified L2 cache would be beneficial that is certainly our main point of the design in the upcoming checkpoint. We also need to add data forwarding unit and hazard detection functionality to become a more conventional supportive RV32I processor.

Checkpoint three. We intended to have a second level in our cache hierarchy along with implementing data forwarding and a hazard detection. All of the features safely get done by the checkpoint due date while we certainly struggle to fit in these major features into one single checkpoint. Ultimately, this checkpoint focuses on developing logic and combining features upon what we have implemented so far to have a data forwarding and hazard detection.

Figure 3 in Appendix shows our design with data forwarding. Our implementation consists of two modules respectively. Data forwarding unit which is located in the EX stage along with the data hazard detection module. The forwarding module checks whether the destination register of an instruction conflicted with any source registers of other instructions. In other words, It handles specifically write-after-read (WAR) errors by forwarding relevant data to the ALU accordingly before it gets overwritten. The hazard detection module handles other existing hazards as well as any existing edge-case situations which could have potentially lead to error state within the pipeline; upon a detection, most hazards cause the pipeline stall.

While at this stage of our design we began to see many issues with time and work conflict, we could successfully implement essential parts of listed design features as a result. Forwarding and hazard detection allow us to support more conventional code. L2 Cache, unlike our worries, could be added at the last minute by passing given testcodes without generating errors. However, we all recognize that we may not have found enough edge cases that may occur in upcoming checkpoints or near future so we had to keep in mind that we will face such invisible errors soon or later.

Checkpoint four. Last but not least, merging everything into one single master piece of code with several advanced features definitely overwhelmed us. While our original goal as a team encompasses putting one more level in cache hierarchy in between L2 cache and physical memory with an eviction write buffer, developing basic hardware prefetcher and a C-Extension, we found that we were very short of time in addition to situation where we can see and detect only few visible errors that have never been out from previous test cases.

We implemented basic one block lookahead prefetching in our I-cache where the next cacheline is fetched in the event of a cache miss.

The reason behind choosing an eviction write buffer was because adding another simple cacheline buffer would be efficient in terms of latency such that we may not have enough efficiency controlling factor within our design, especially in between physical memory and L2 cache. Then, designing an eviction write buffer was actually not too hard compared to implementation of caches in previous checkpoints. However, we suddenly get data mismatching issues with given test code for this checkpoint specifically from the loading data. Meanwhile, any addition of cache related functionality may cause more problems in the Writeback stage which we thought to hold the eviction write buffer, but may continue with its own unit test. Besides, too many system errors caused on macbook and fastx stability shut out numerous trials which delayed the process in this checkpoint.

Eventually, we were only able to implement prefetching. We recognized that our writeback stage had met edge cases that we haven't tested yet while passing all given testcodes previously. We finalize that implementing upon this current status, may hurt us more so decide to have it organized and get it ready to be run on the competition codes with stability. Following section describes more about how we thought of such designs and the background planning and designs respectively.

IV. Advanced Design Features

Going into the last checkpoint, the final stage of this MP, we considered the following points in terms of how we make a better and efficient pipelined processor design: how could we further decrease memory-access latency, how could we handle branch-prediction better, and how could we have nice modifications that eliminate unnecessary or redundant features we may not have taken out yet. Ultimately, we decided to have a better cache hierarchy along with adding another level of simple cache called eviction write buffer in between L2 cache and physical memory, simple hardware prefetcher to solve latency issues better. Due to complications with given test code, testing and debugging was certainly brutal and yet, we could not finish all implementation of advanced features we have designed.

Eviction Write Buffer. This is known to be a single cacheline buffer which paralyzes the WB stage. Upon a writeback, writeback request should be routed to Eviction

Write Buffer, not our next level of memory, that here is our physical memory. Then it services the writeback at the next available time when the next level cache is free, again for here our physical memory. We thought the load should only stall if there is a Writeback performed while Eviction Write buffer is full. It certainly has been written and attached in the figure 4 of Appendix, but we could not manage it to be debugged and merged into our design.

3rd Advanced Features we have tried. Description

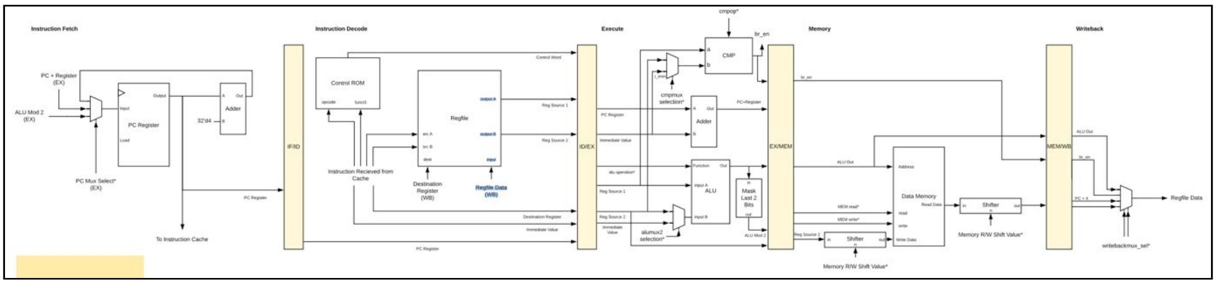
V. Conclusion

Towards the end, we thought we could have a grandiose finale with solid fancy advanced features that we could prove ourselves we can do this. However, not only because we simply did not have enough time to actually implement functionalities after designing those features, but also, the situation without school resources such as access to computer labs and actual face-to-face office hours did not make things better. Overall, it surely was a great experience having a team and learning a lot how to communicate better with each other especially under the limited resources and limited communication methods and time under the situation of the sudden outbreak of COVID-19. Scheduled weekly meetings with TA and our team was not an issue but, having our own meeting was pretty difficult due to the individual schedule. However, not only the project was time-sinking, but also, debugging and implementation through limited resources such as unstable internet connection and using laggy Modelsim and Quartus along with fastx was just tough enough to be honest to get some helps from peers and mentors if we were not under the situation, at last, we thought this was a valuable experience as an ECE student. If we have chances to talk or leave some short advices to future students, we will definitely say that learn more and better in depth in ECE 385 which is the one and only class that actually deals with systemverilog almost a whole semester so students taking this class do not get confused at the starting of MP that may trap you at the beginning of the semester. If not, rather ask the department about making another class dealing with systemverilog, maybe a single credit lab course so we get to use systemverilog better and not forget the majority of syntax and its usage since majority of students do not take ECE 411 right after taking ECE 385.

Appendix

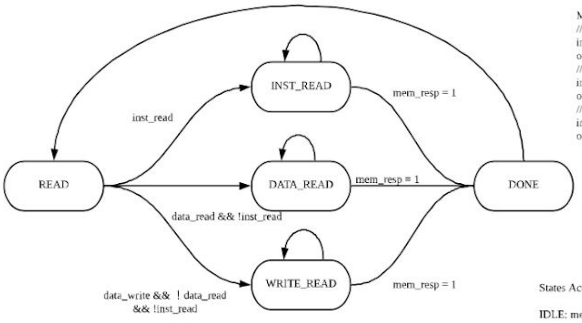
RV32I Pipeline Design

Figure 1. Basic Design of 5-Stage Pipelined Datapath



Continues at next page

Figure 2. Basic L1 Cache Design: Control and Datapath



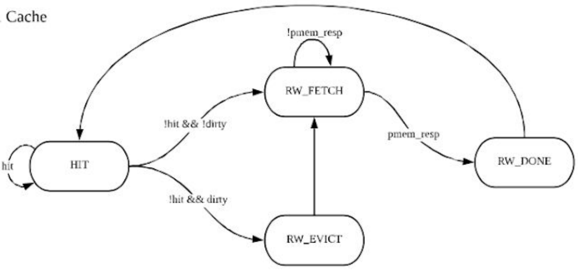
L1 Cache Arbiter

```
Module Ports:
// from Icache
input inst_read, inst_addr, inst_wdata
output inst_rdata, inst_resp
//from Dcache
input data_read, data_write, data_addr, data_wdata
output data_rdata, data_resp
//from pmem
input mem_rdata, mem_resp
output mem_read, mem_write, mem_addr, mem_wdata
```

```
States Action:

IDLE: mem_read = mem_write = 0

INST_READ: mem_read = 1, mem_addr = inst_addr
           done_mux_sel = IR

DATA_READ: mem_read = 1, mem_addr= data_addr
           done_mux_sel = DR

DATA_WRITE: mem_read = 1, mem_addr= data_addrmem_wdata = data_wdata
            done_mux_sel = DW

DONE: case(done_mux_sel):
        IR: inst_resp = 1, inst_rdata = mem_rdata
        DR: data_resp = 1, data_rdata = mem_rdata
        DW: data_resp = 1
```

Module Description:
We are palnning to implement 2-way, 8-words set-associative caches the same as MP2.The SM and module description are the ones for data cache, while instruction cache will not need any components or design that's used for WRITE operation, such as RW_EVICT state and dirty metaarray. (Eviction is never needed if WRITE is never performed) Also note that in the following paragraph, all data and metaarray are assumed to be read/written from the proper sets (decided by addr[7:5]) and proper way (decided by the output of LRU register). For simplicity, we will not reiterate on the value of set index and way index.

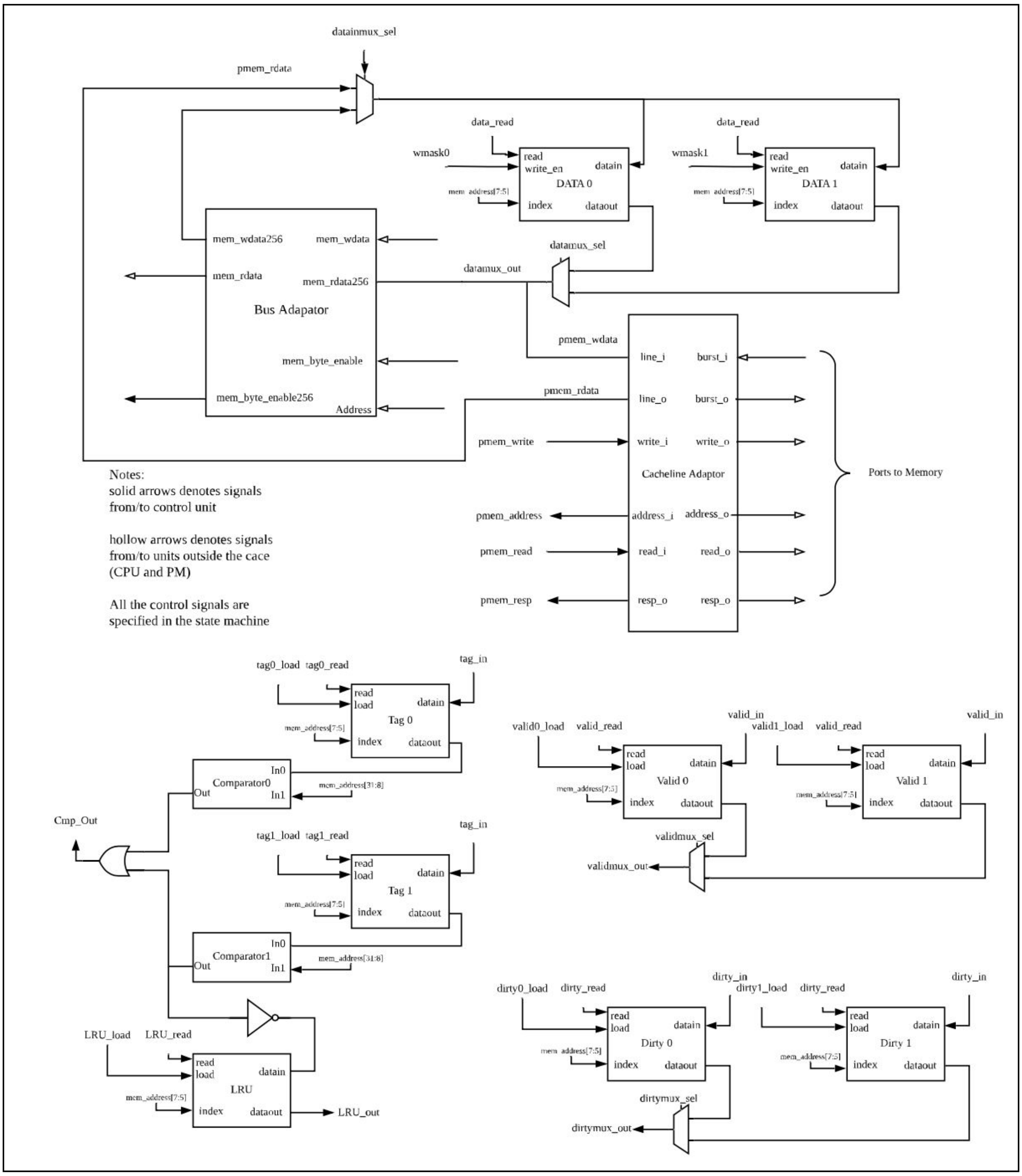
L1 Cache



State Transition Description:
hit = (addr[31:8] == tag) && valid
mem_read and mem_write are inputs.
dirty is simply the ouput of the dirty metaarray.

```
Simplifed Ports List:
    //Ports to CPU
    input mem_read, mem_write, mem_wdata, mem_addr
    output mem_resp, mem_rdata
    //Ports to physical memory
    input pmem_resp, pmem_rdata
    output pmem_read, pmem_write, pmem_wdata, pmem_addr
```

```
State Action:
HIT: if (hit)
        if (mem_read)
            mem_rdata = data_arr_out,
            update lru, mem_resp = 1
        else if (mem_write)
            data_arr_in = mem_wdata
            update LRU, raise response to cpu, set dirty array

RW_EVICT: pmem_write = 1, pmem_wdata = data_array[LRU_out]

RW_FETCH:  pmem_read = 1, data_arr_in = pmem_rdata,
           updatetag and valid array

READ_SEND: mem_resp = 1, update LRU
           if (mem_read)
               mem_rdata = data_arr_out, clear dirty
           else if(mem_write)
               data_arr_in = mem_wdata, set dirty
```



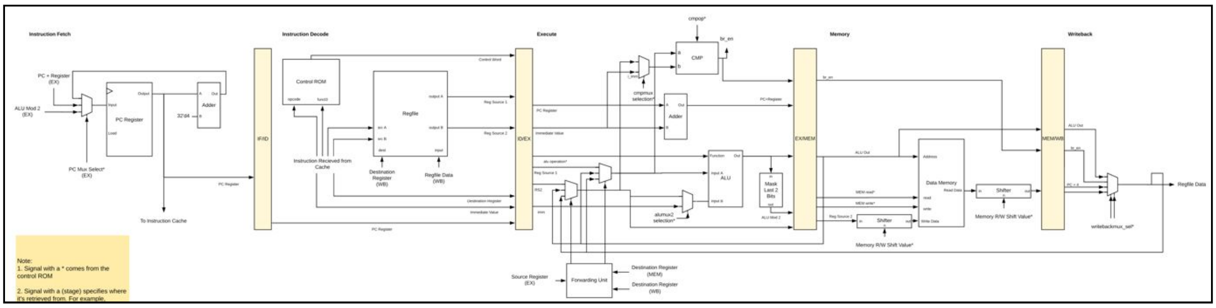
Notes:
solid arrows denotes signals
from/to control unit

hollow arrows denotes signals
from/to units outside the cace
(CPU and PM)

All the control signals are
specified in the state machine

Continues at next page

Figure 3. 5-Stage Pipelined Datapath with Data forwarding unit

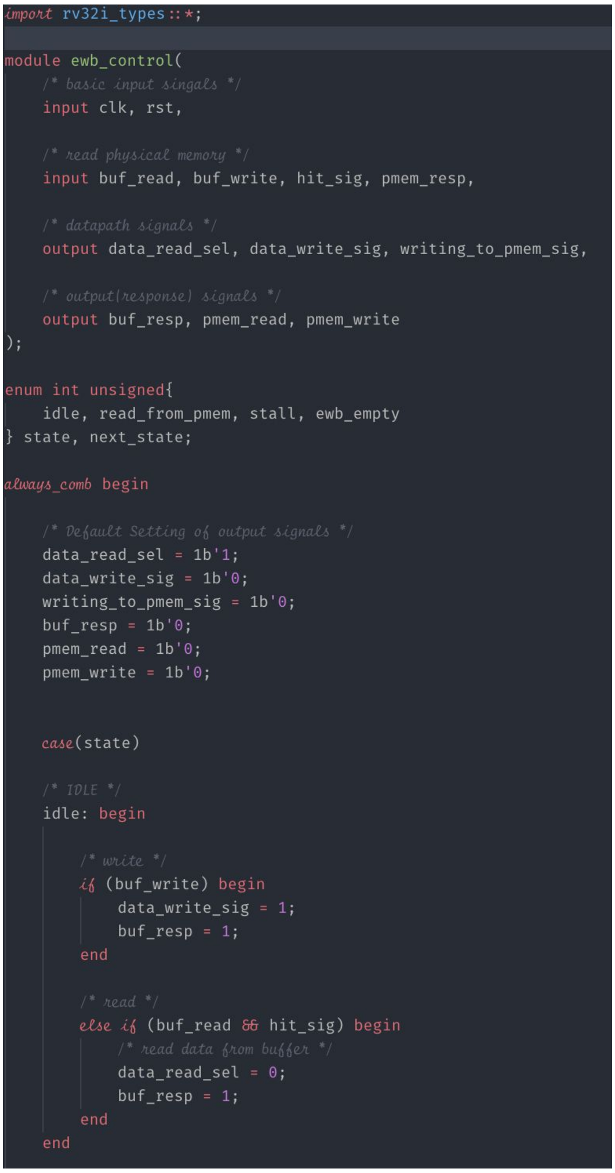


Note:
1. Signal with a * comes from the control ROM

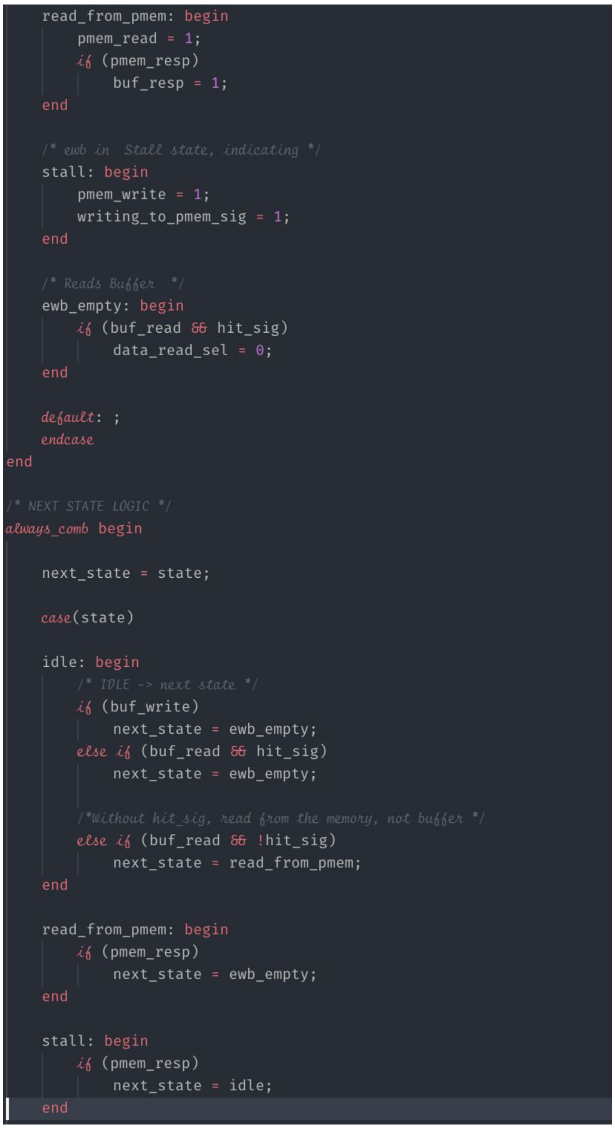
2. Signal with a (stage) specifies where it's retrieved from. For example,

Continues at next page

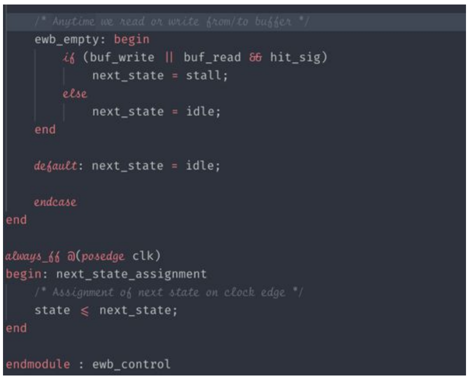
Figure 4. Eviction Write Buffer Codes: Control and Datapath



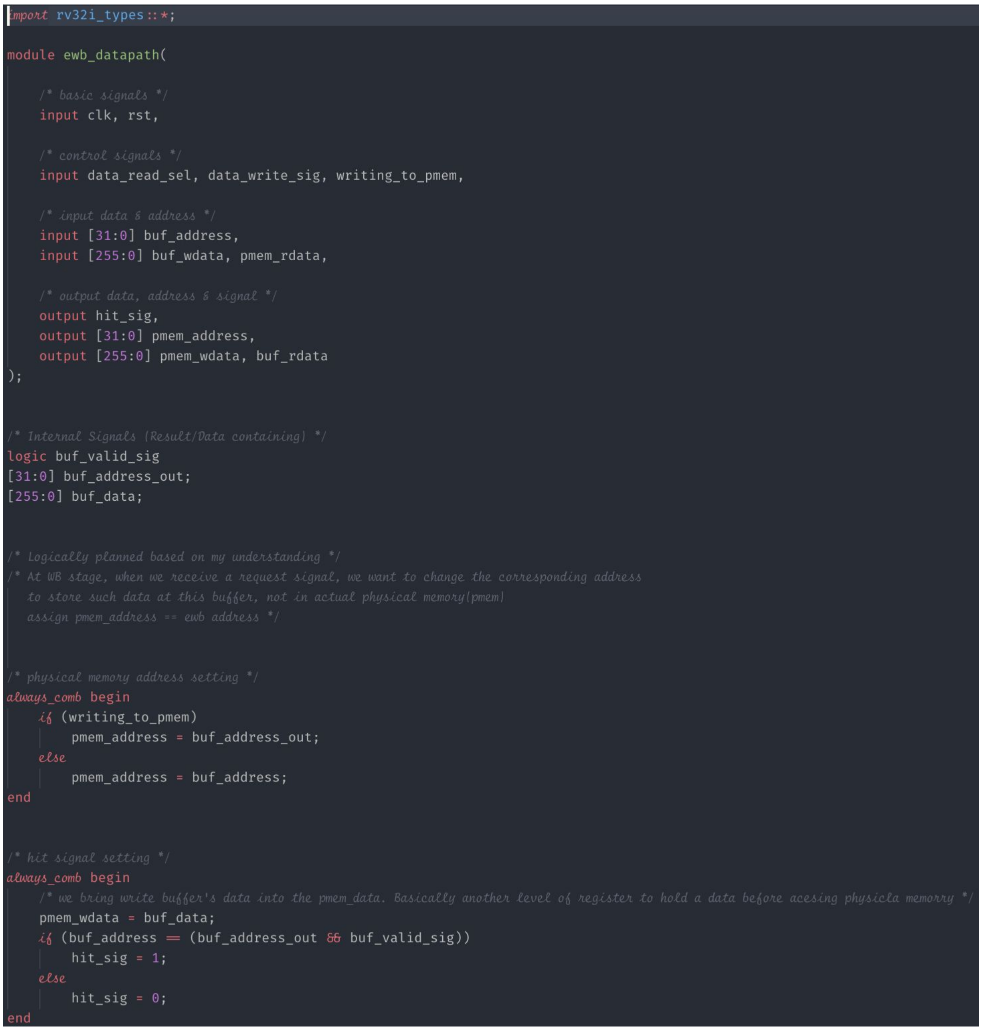
```
import rv32i_types::*;

module ewb_control(
    /* basic input singals */
    input clk, rst,

    /* read physical memory */
    input buf_read, buf_write, hit_sig, pmem_resp,

    /* datapath signals */
    output data_read_sel, data_write_sig, writing_to_pmem_sig,

    /* output(response) signals */
    output buf_resp, pmem_read, pmem_write
);

enum int unsigned{
    idle, read_from_pmem, stall, ewb_empty
} state, next_state;

always_comb begin

    /* Default Setting of output signals */
    data_read_sel = 1b'1;
    data_write_sig = 1b'0;
    writing_to_pmem_sig = 1b'0;
    buf_resp = 1b'0;
    pmem_read = 1b'0;
    pmem_write = 1b'0;


    case(state)

    /* IDLE */
    idle: begin

        /* write */
        if (buf_write) begin
            data_write_sig = 1;
            buf_resp = 1;
        end

        /* read */
        else if (buf_read && hit_sig) begin
            /* read data from buffer */
            data_read_sel = 0;
            buf_resp = 1;
        end
    end
```



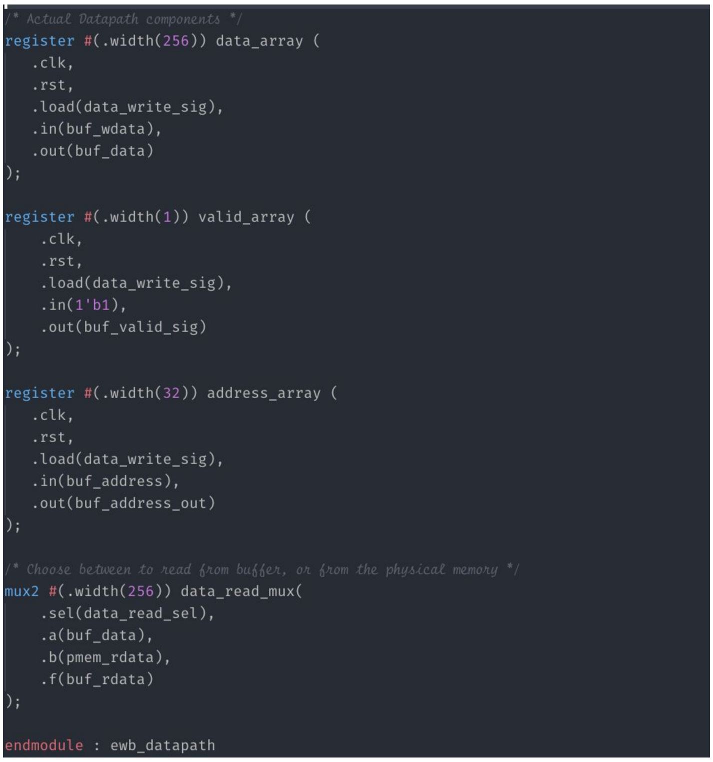
```
    read_from_pmem: begin
        pmem_read = 1;
        if (pmem_resp)
            buf_resp = 1;
    end

    /* ewb in  Stall state, indicating */
    stall: begin
        pmem_write = 1;
        writing_to_pmem_sig = 1;
    end

    /* Reads Buffer  */
    ewb_empty: begin
        if (buf_read && hit_sig)
            data_read_sel = 0;
    end

    default: ;
    endcase
end

/* NEXT STATE LOGIC */
always_comb begin

    next_state = state;

    case(state)

    idle: begin
        /* IDLE -> next state */
        if (buf_write)
            next_state = ewb_empty;
        else if (buf_read && hit_sig)
            next_state = ewb_empty;

        /*Without hit_sig, read from the memory, not buffer */
        else if (buf_read && !hit_sig)
            next_state = read_from_pmem;
    end

    read_from_pmem: begin
        if (pmem_resp)
            next_state = ewb_empty;
    end

    stall: begin
        if (pmem_resp)
            next_state = idle;
    end
```



```
    /* Anytime we read or write from/to buffer */
    ewb_empty: begin
        if (buf_write || buf_read && hit_sig)
            next_state = stall;
        else
            next_state = idle;
    end

    default: next_state = idle;

    endcase
end

always_ff @(posedge clk)
begin: next_state_assignment
    /* Assignment of next state on clock edge */
    state <= next_state;
end

endmodule : ewb_control
```

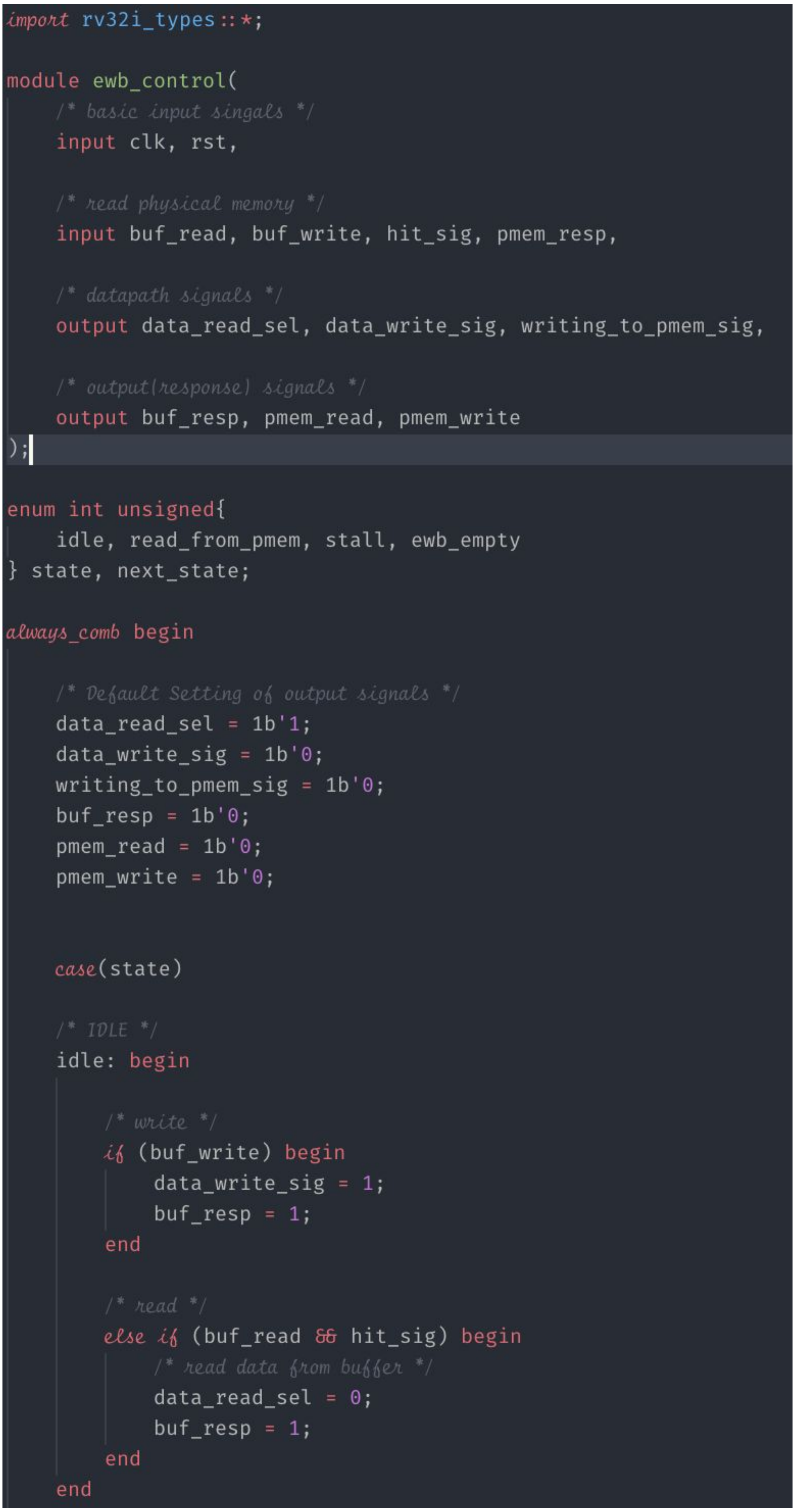


```
import rv32i_types::*;

module ewb_datapath(

    /* basic signals */
    input clk, rst,

    /* control signals */
    input data_read_sel, data_write_sig, writing_to_pmem,

    /* input data & address */
    input [31:0] buf_address,
    input [255:0] buf_wdata, pmem_rdata,

    /* output data, address & signal */
    output hit_sig,
    output [31:0] pmem_address,
    output [255:0] pmem_wdata, buf_rdata
);


/* Internal Signals (Result/Data containing) */
logic buf_valid_sig
[31:0] buf_address_out;
[255:0] buf_data;


/* Logically planned based on my understanding */
/* At WB stage, when we receive a request signal, we want to change the corresponding address
  to store such data at this buffer, not in actual physical memory(pmem)
  assign pmem_address == ewb address */


/* physical memory address setting */
always_comb begin
    if (writing_to_pmem)
        pmem_address = buf_address_out;
    else
        pmem_address = buf_address;
end


/* hit signal setting */
always_comb begin
    /* we bring write buffer's data into the pmem_data. Basically another level of register to hold a data before acesing physicla memorry */
    pmem_wdata = buf_data;
    if (buf_address == (buf_address_out && buf_valid_sig))
        hit_sig = 1;
    else
        hit_sig = 0;
end
```



```
/* Actual Datapath components */
register #(.width(256)) data_array (
   .clk,
   .rst,
   .load(data_write_sig),
   .in(buf_wdata),
   .out(buf_data)
);

register #(.width(1)) valid_array (
    .clk,
    .rst,
    .load(data_write_sig),
    .in(1'b1),
    .out(buf_valid_sig)
);

register #(.width(32)) address_array (
   .clk,
   .rst,
   .load(data_write_sig),
   .in(buf_address),
   .out(buf_address_out)
);

/* Choose between to read from buffer, or from the physical memory */
mux2 #(.width(256)) data_read_mux(
    .sel(data_read_sel),
    .a(buf_data),
    .b(pmem_rdata),
    .f(buf_rdata)
);

endmodule : ewb_datapath
```

<table>
<tr><td colspan="4"></td></tr>
<tr><td colspan="4"></td></tr>
<tr><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td></td><td></td><td></td></tr>
</table>



```
import rv32i_types::*;

module ewb_control(
    /* basic input singals */
    input clk, rst,

    /* read physical memory */
    input buf_read, buf_write, hit_sig, pmem_resp,

    /* datapath signals */
    output data_read_sel, data_write_sig, writing_to_pmem_sig,

    /* output(response) signals */
    output buf_resp, pmem_read, pmem_write
);

enum int unsigned{
    idle, read_from_pmem, stall, ewb_empty
} state, next_state;

always_comb begin

    /* Default Setting of output signals */
    data_read_sel = 1b'1;
    data_write_sig = 1b'0;
    writing_to_pmem_sig = 1b'0;
    buf_resp = 1b'0;
    pmem_read = 1b'0;
    pmem_write = 1b'0;


    case(state)

    /* IDLE */
    idle: begin

        /* write */
        if (buf_write) begin
            data_write_sig = 1;
            buf_resp = 1;
        end

        /* read */
        else if (buf_read && hit_sig) begin
            /* read data from buffer */
            data_read_sel = 0;
            buf_resp = 1;
        end
    end
```