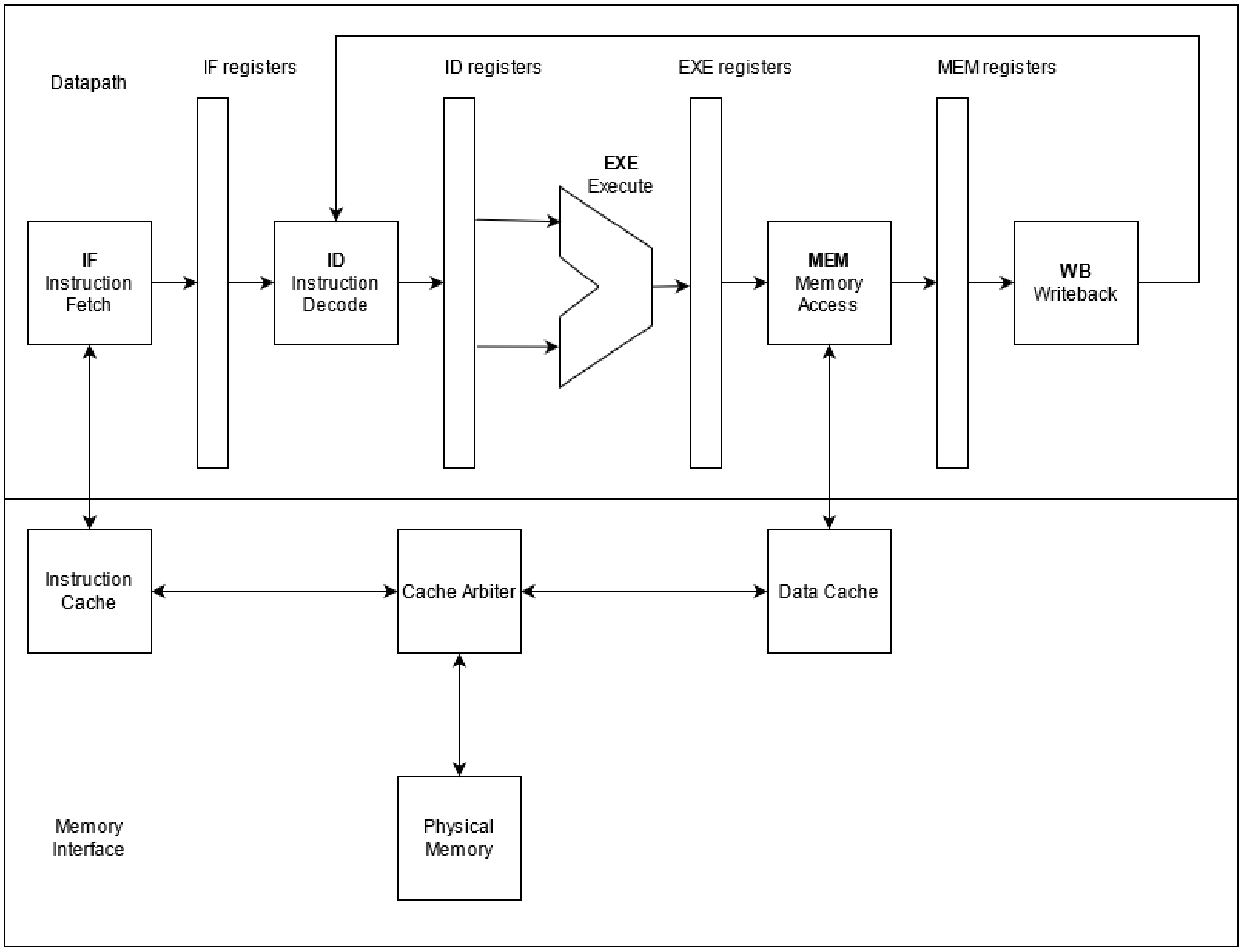
MP4 Design Document

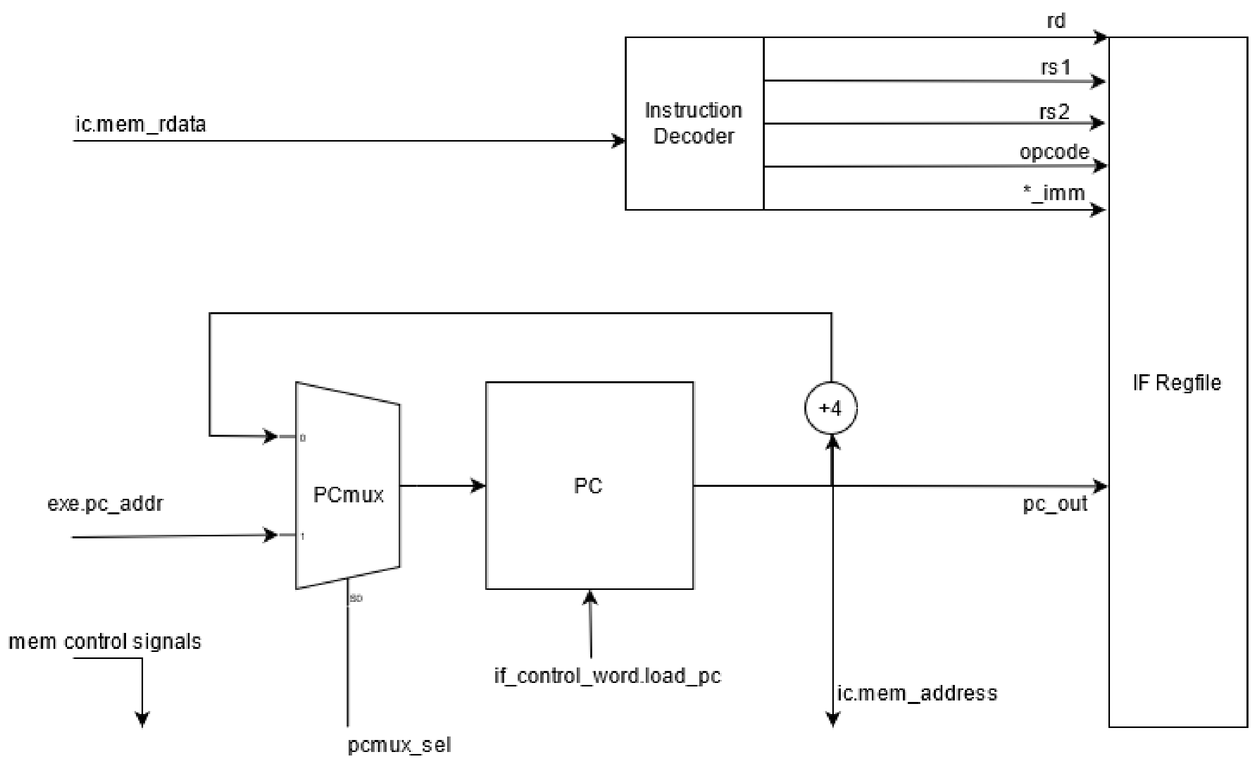
Group Name: Team ATM
Timothy Chan (tjchan2)
Matthew Nolan (mrnolan3)
Andrew Gacek (andrewg3)

Block Diagram



Design implementation follows the classic RISC pipeline described in lecture

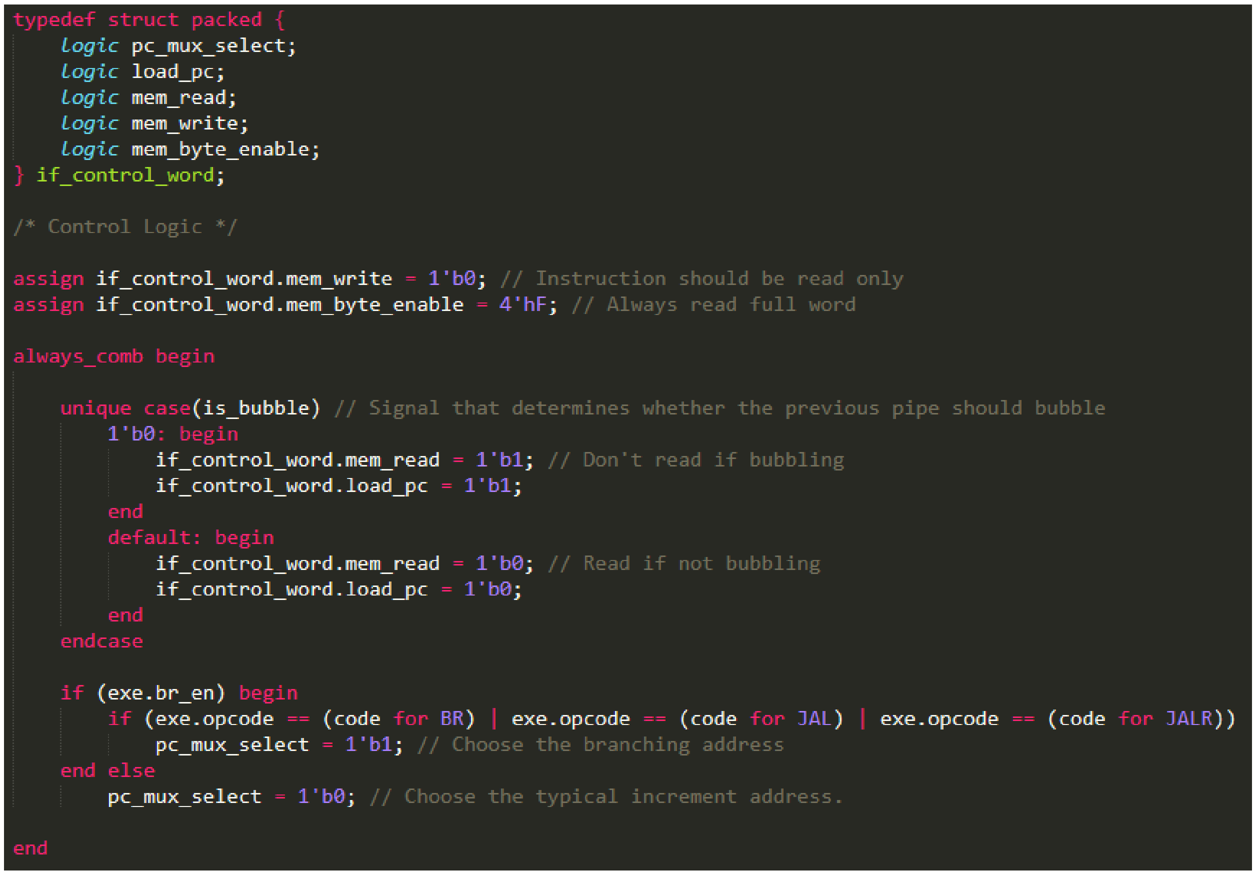
IF- Instruction Fetch



Signal descriptions

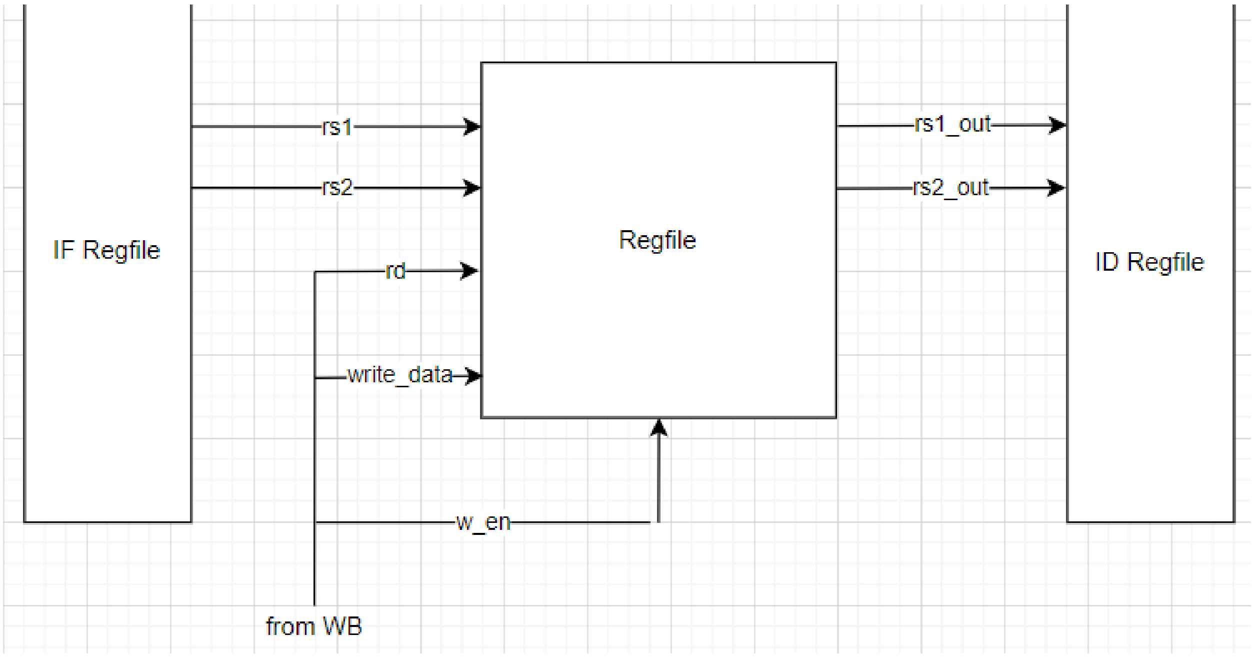
Signal	Description
ic.mem_rdata	data returned from the instruction cache
exe.pc_addr	location to branch to on branch enable
pcmux_sel	control signal to decide if we simply increment PC or jump to a new address. Set by the value of exe.br_en
ic.mem_address	Address in memory instruction cache should read from

Control word Psuedocode

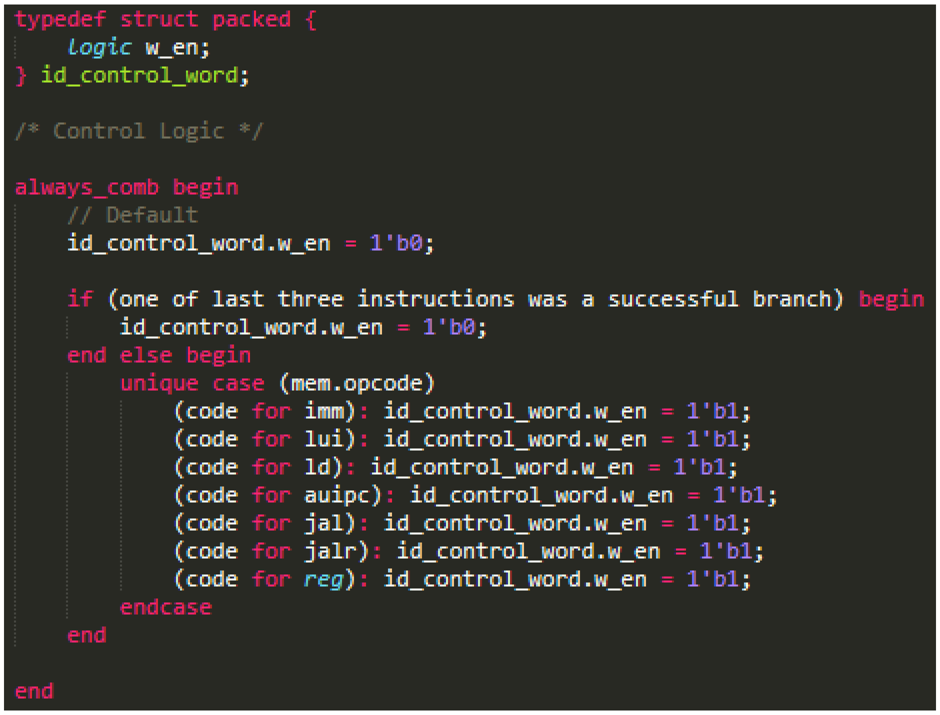


```
typedef struct packed {
    logic pc_mux_select;
    logic load_pc;
    logic mem_read;
    logic mem_write;
    logic mem_byte_enable;
} if_control_word;

/* Control Logic */

assign if_control_word.mem_write = 1'b0; // Instruction should be read only
assign if_control_word.mem_byte_enable = 4'hF; // Always read full word

always_comb begin

    unique case(is_bubble) // Signal that determines whether the previous pipe should bubble
        1'b0: begin
            if_control_word.mem_read = 1'b1; // Don't read if bubbling
            if_control_word.load_pc = 1'b1;
        end
        default: begin
            if_control_word.mem_read = 1'b0; // Read if not bubbling
            if_control_word.load_pc = 1'b0;
        end
    endcase

    if (exe.br_en) begin
        if (exe.opcode == (code for BR) | exe.opcode == (code for JAL) | exe.opcode == (code for JALR))
            pc_mux_select = 1'b1; // Choose the branching address
    end else
        pc_mux_select = 1'b0; // Choose the typical increment address.

end
```

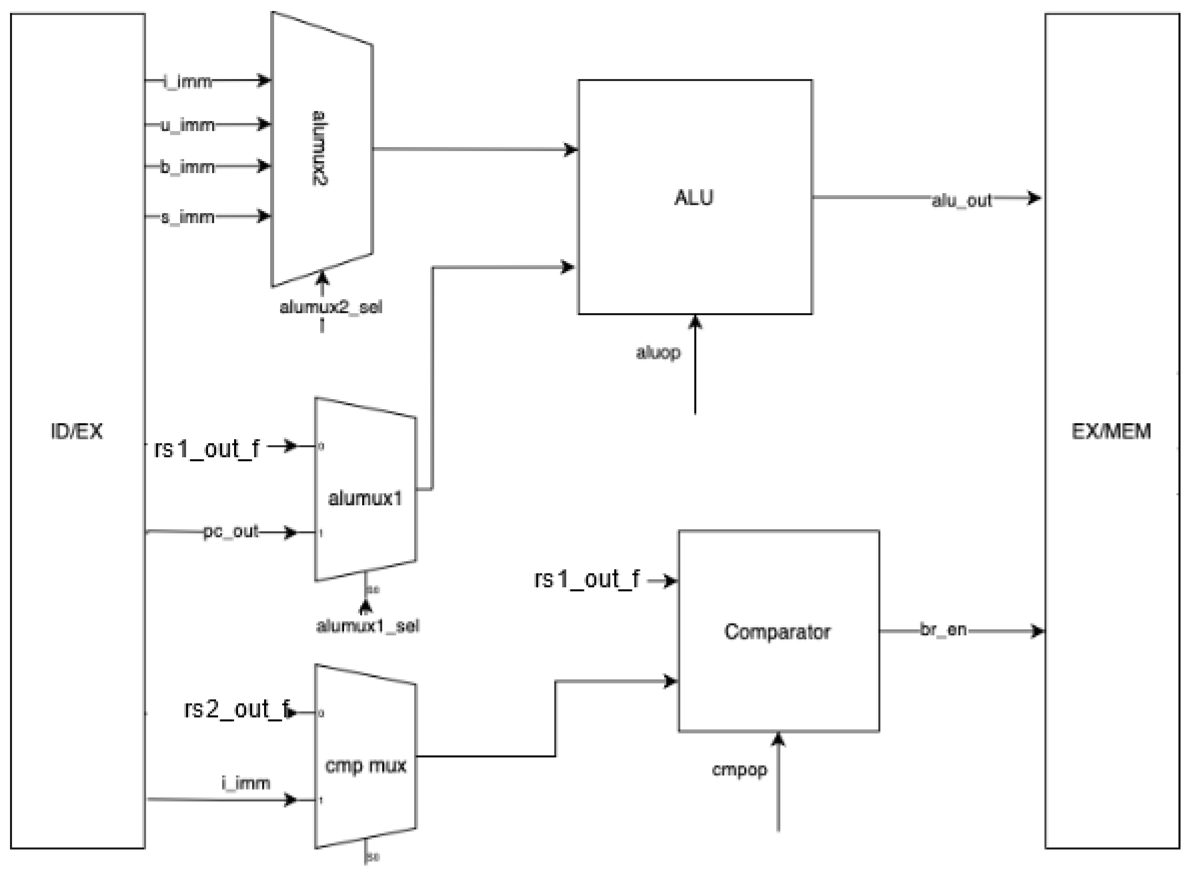
ID- Instruction Decode



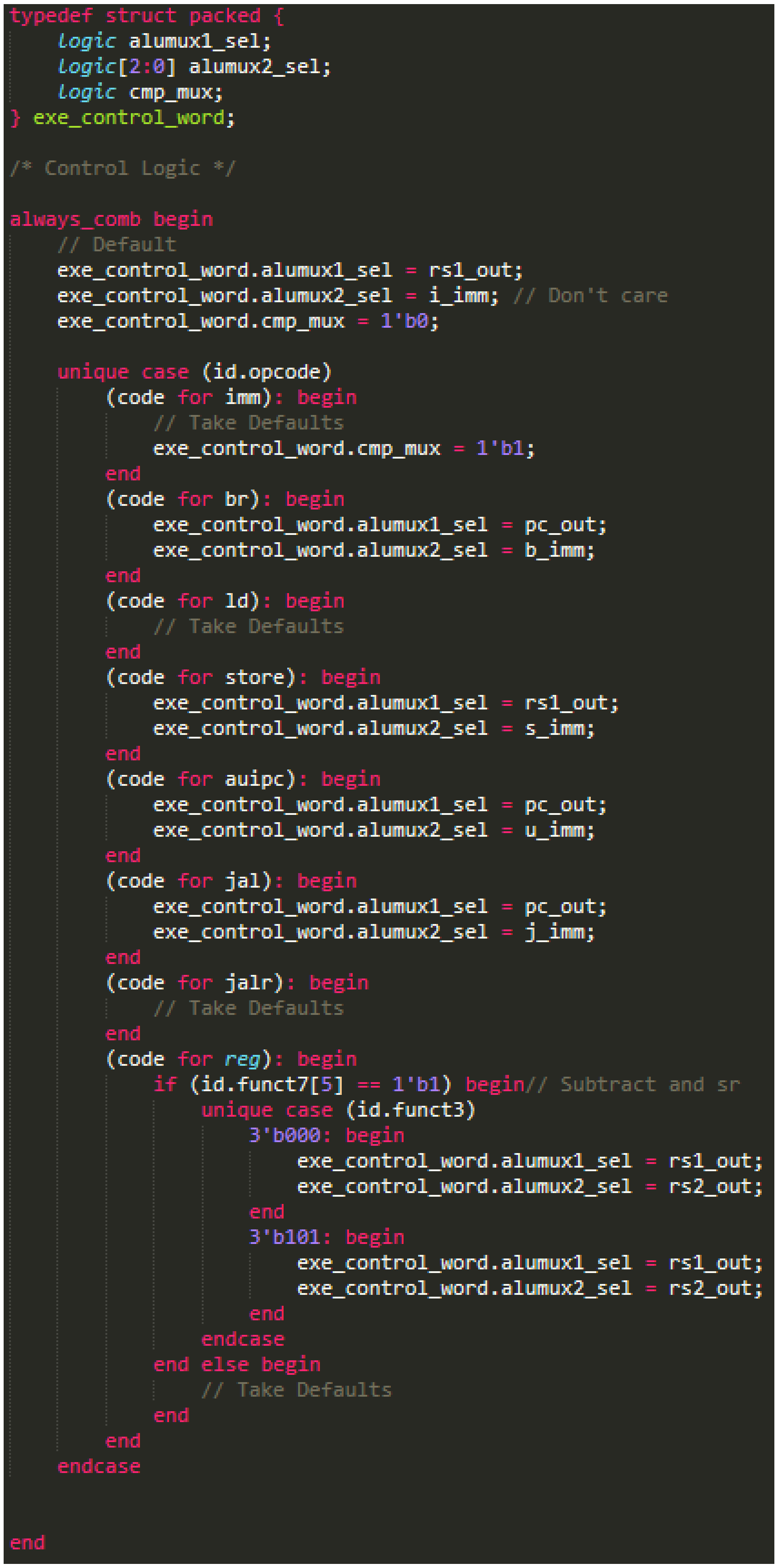
Control Word Pseudocode:


```
typedef struct packed {
    logic w_en;
} id_control_word;

/* Control Logic */

always_comb begin
    // Default
    id_control_word.w_en = 1'b0;

    if (one of last three instructions was a successful branch) begin
        id_control_word.w_en = 1'b0;
    end else begin
        unique case (mem.opcode)
            (code for imm): id_control_word.w_en = 1'b1;
            (code for lui): id_control_word.w_en = 1'b1;
            (code for ld): id_control_word.w_en = 1'b1;
            (code for auipc): id_control_word.w_en = 1'b1;
            (code for jal): id_control_word.w_en = 1'b1;
            (code for jalr): id_control_word.w_en = 1'b1;
            (code for reg): id_control_word.w_en = 1'b1;
        endcase
    end

end
```

EXE- Execute

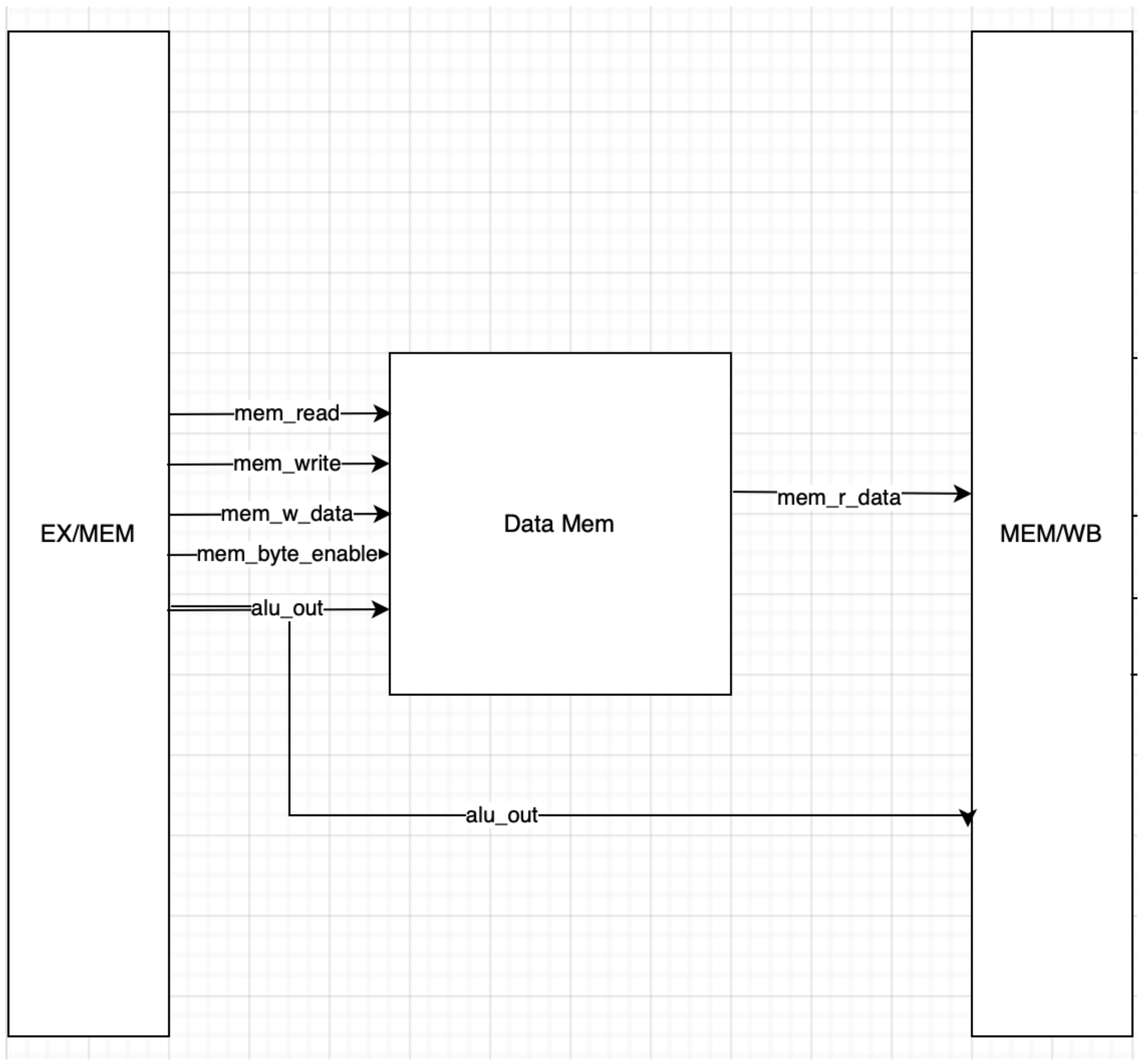


Control Word Pseudocode:

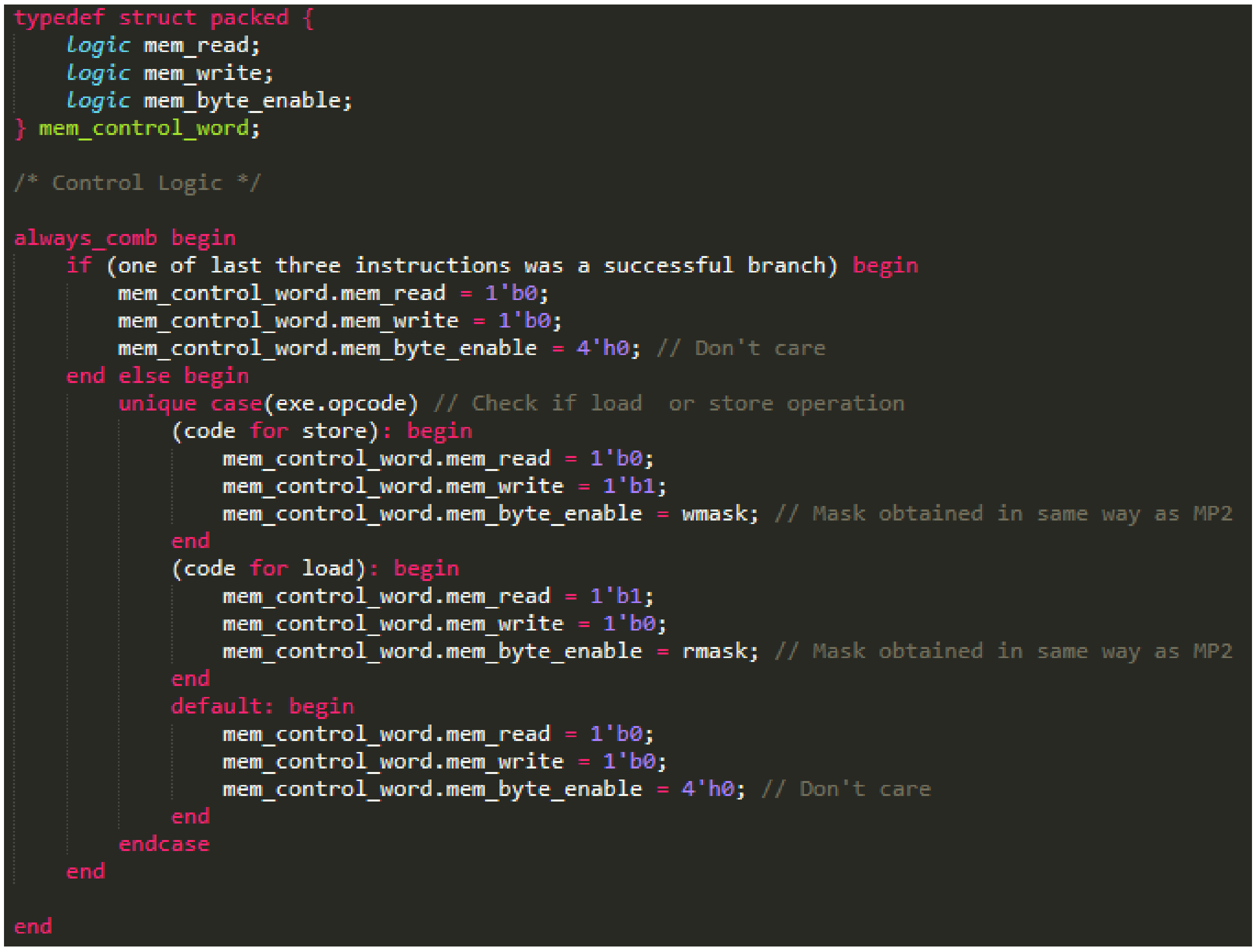


```
typedef struct packed {
    logic alumux1_sel;
    logic[2:0] alumux2_sel;
    logic cmp_mux;
} exe_control_word;

/* Control Logic */

always_comb begin
    // Default
    exe_control_word.alumux1_sel = rs1_out;
    exe_control_word.alumux2_sel = i_imm; // Don't care
    exe_control_word.cmp_mux = 1'b0;

    unique case (id.opcode)
        (code for imm): begin
            // Take Defaults
            exe_control_word.cmp_mux = 1'b1;
        end
        (code for br): begin
            exe_control_word.alumux1_sel = pc_out;
            exe_control_word.alumux2_sel = b_imm;
        end
        (code for ld): begin
            // Take Defaults
        end
        (code for store): begin
            exe_control_word.alumux1_sel = rs1_out;
            exe_control_word.alumux2_sel = s_imm;
        end
        (code for auipc): begin
            exe_control_word.alumux1_sel = pc_out;
            exe_control_word.alumux2_sel = u_imm;
        end
        (code for jal): begin
            exe_control_word.alumux1_sel = pc_out;
            exe_control_word.alumux2_sel = j_imm;
        end
        (code for jalr): begin
            // Take Defaults
        end
        (code for reg): begin
            if (id.funct7[5] == 1'b1) begin// Subtract and sr
                unique case (id.funct3)
                    3'b000: begin
                        exe_control_word.alumux1_sel = rs1_out;
                        exe_control_word.alumux2_sel = rs2_out;
                    end
                    3'b101: begin
                        exe_control_word.alumux1_sel = rs1_out;
                        exe_control_word.alumux2_sel = rs2_out;
                    end
                endcase
            end else begin
                // Take Defaults
            end
        end
    endcase


end
```

MEM- Memory Access

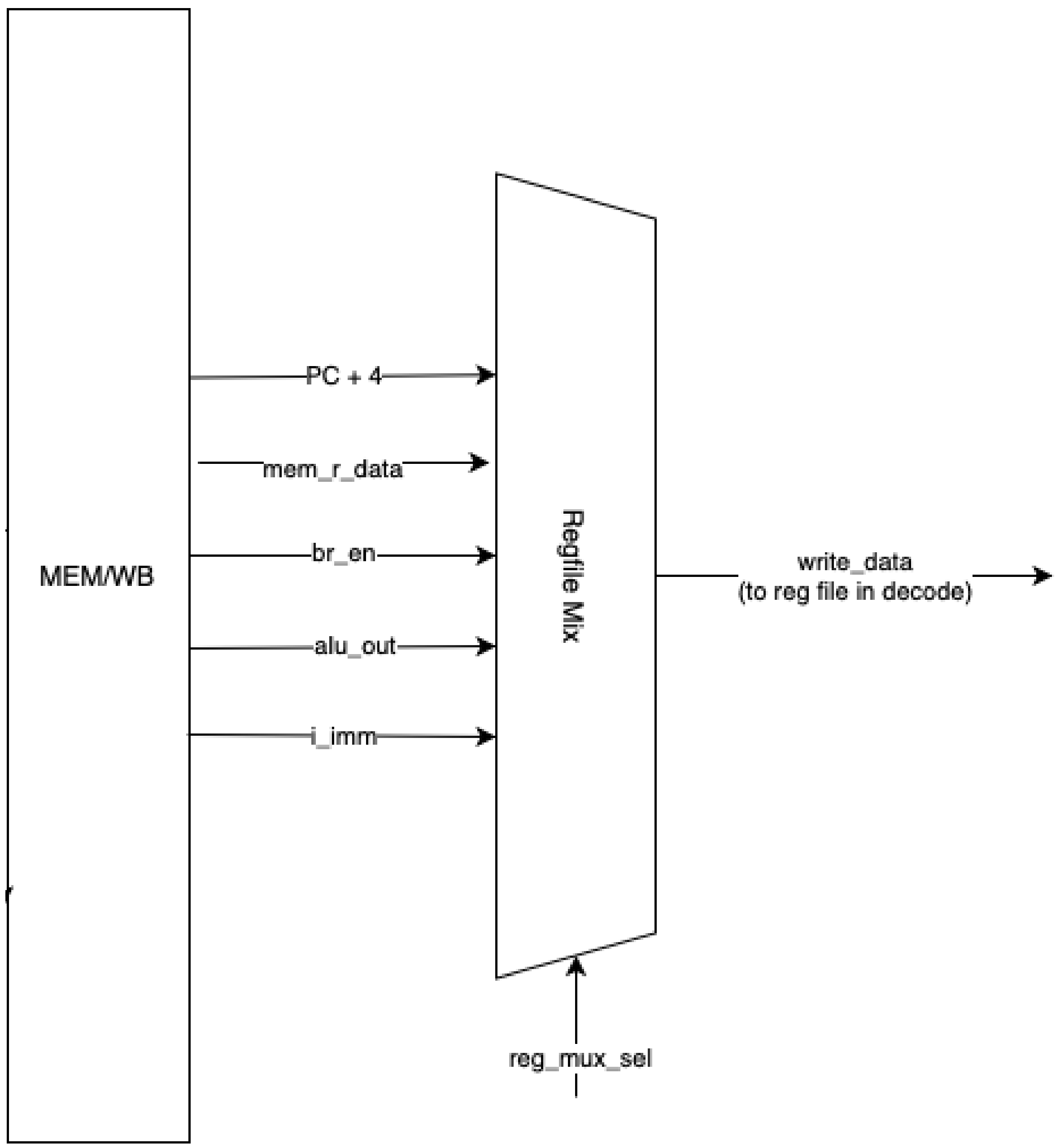


Control Word Pseudocode:



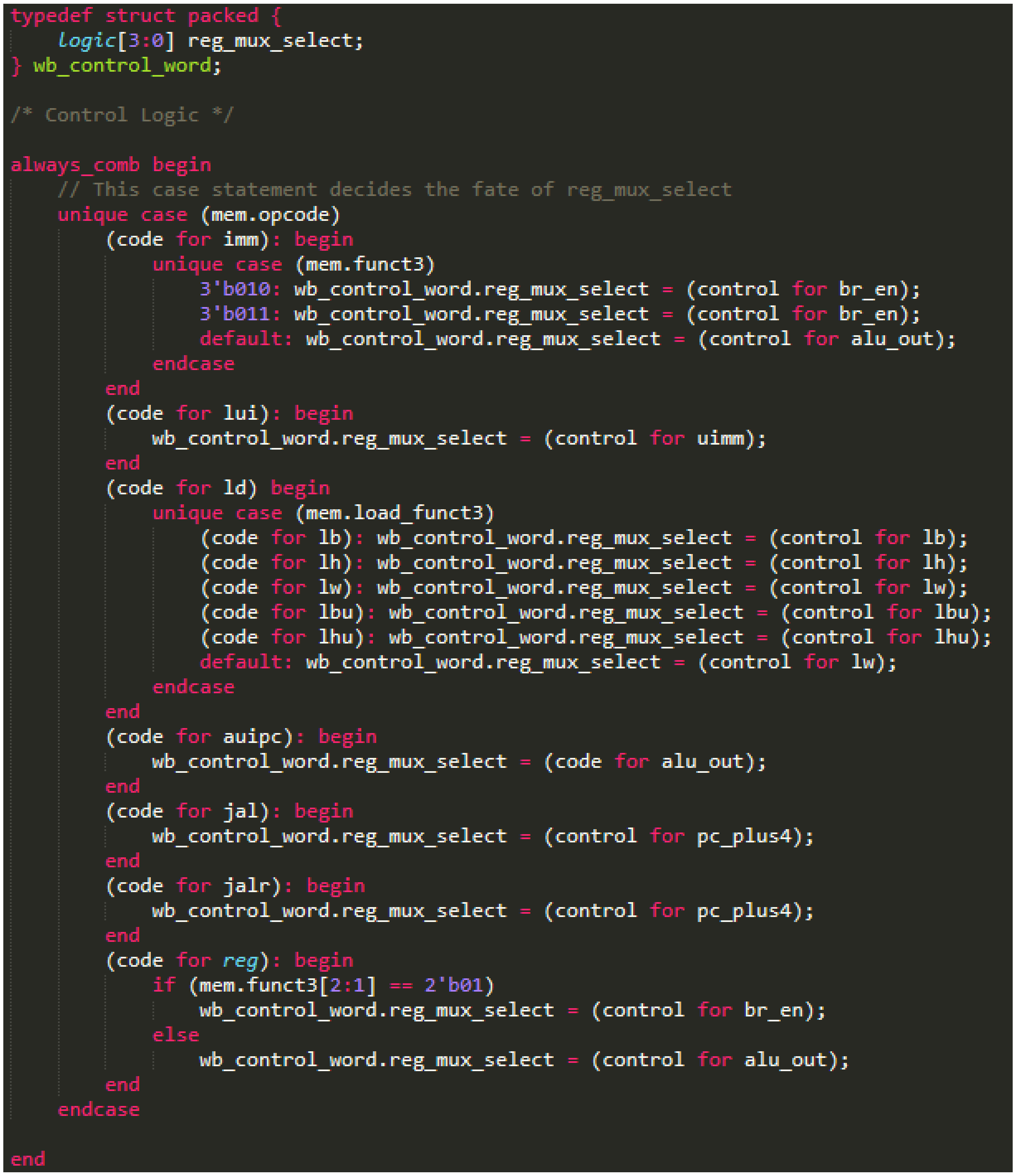
```
typedef struct packed {
    logic mem_read;
    logic mem_write;
    logic mem_byte_enable;
} mem_control_word;

/* Control Logic */

always_comb begin
    if (one of last three instructions was a successful branch) begin
        mem_control_word.mem_read = 1'b0;
        mem_control_word.mem_write = 1'b0;
        mem_control_word.mem_byte_enable = 4'h0; // Don't care
    end else begin
        unique case(exe.opcode) // Check if load  or store operation
            (code for store): begin
                mem_control_word.mem_read = 1'b0;
                mem_control_word.mem_write = 1'b1;
                mem_control_word.mem_byte_enable = wmask; // Mask obtained in same way as MP2
            end
            (code for load): begin
                mem_control_word.mem_read = 1'b1;
                mem_control_word.mem_write = 1'b0;
                mem_control_word.mem_byte_enable = rmask; // Mask obtained in same way as MP2
            end
            default: begin
                mem_control_word.mem_read = 1'b0;
                mem_control_word.mem_write = 1'b0;
                mem_control_word.mem_byte_enable = 4'h0; // Don't care
            end
        endcase
    end

end
```

WB- Writeback



PC+4 comes from pc_out in Instruction Fetch
Mem_r_data comes from Data Memory output in Memory Access
Br_en comes from output from Comparator in Execute
Alu_out comes from output of ALU in Execute
i_imm comes from the Instruction decoder in Instruction Fetch

Control Word Pseudocode



```
typedef struct packed {
    logic[3:0] reg_mux_select;
} wb_control_word;

/* Control Logic */

always_comb begin
    // This case statement decides the fate of reg_mux_select
    unique case (mem.opcode)
        (code for imm): begin
            unique case (mem.funct3)
                3'b010: wb_control_word.reg_mux_select = (control for br_en);
                3'b011: wb_control_word.reg_mux_select = (control for br_en);
                default: wb_control_word.reg_mux_select = (control for alu_out);
            endcase
        end
        (code for lui): begin
            wb_control_word.reg_mux_select = (control for uimm);
        end
        (code for ld) begin
            unique case (mem.load_funct3)
                (code for lb): wb_control_word.reg_mux_select = (control for lb);
                (code for lh): wb_control_word.reg_mux_select = (control for lh);
                (code for lw): wb_control_word.reg_mux_select = (control for lw);
                (code for lbu): wb_control_word.reg_mux_select = (control for lbu);
                (code for lhu): wb_control_word.reg_mux_select = (control for lhu);
                default: wb_control_word.reg_mux_select = (control for lw);
            endcase
        end
        (code for auipc): begin
            wb_control_word.reg_mux_select = (code for alu_out);
        end
        (code for jal): begin
            wb_control_word.reg_mux_select = (control for pc_plus4);
        end
        (code for jalr): begin
            wb_control_word.reg_mux_select = (control for pc_plus4);
        end
        (code for reg): begin
            if (mem.funct3[2:1] == 2'b01)
                wb_control_word.reg_mux_select = (control for br_en);
            else
                wb_control_word.reg_mux_select = (control for alu_out);
        end
    endcase

end
```

Registers Design

The register design that we decided to work with is a monolithic struct based approach. The registers will be in a struct using the descriptions below. Some signals will be passed through to the next stage, while others are used for the logic of that stage. We believe this will be the easiest to use design when it comes to implementation.

Below is a non-exhaustive list of the different registers contained at each stage.

Signal name	Description
if.rs1	register source #1 of fetched instruction (if applicable)
if.rs2	register source #2 of fetched instruction (if applicable)
if.rd	register destination of fetched instruction (if applicable)
if.opcode	opcode of fetched instruction
if.*_imm	instruction immediate values
if.pc_out	PC value at time of instruction fetch. Needed to calculate BR/JAL commands

Signal name	Description
id.rs1_out	register source #1 value of fetched instruction (if applicable)
id.rs2_out	register source #2 value of fetched instruction (if applicable)
id.rd	register destination of fetched instruction (if applicable)
id.opcode	opcode of fetched instruction
id.*_imm	instruction immediate values

Below 2 still need to be done

Signal name	Description
exe.alu_out	register source #1 value of fetched instruction (if applicable)
exe.rs2_out	register source #2 value of fetched instruction (if applicable)
exe.rd	register destination of fetched instruction (if applicable)
exe.opcode	opcode of fetched instruction
exe.br_addr	Value that pc should jump to on branch condition true

Signal name	Description
mem.pc_out	Used to store the address of the next instruction into a register
mem.mem_rdata	Data from Data Cache used for load operations

mem.br_en	Branch enable result stored in certain immediate arithmetic operations.
mem.alu_out	Output of the ALU
mem.u_imm	U immediate value written in load upper immediate
mem.rd	Register to write into
mem.w_en	Control value to determine whether we write to regfile. Taken from opcode.
mem.opcode	opcode of fetched instruction

Advanced Features:

Multiplier

A multiplier will be added into the pipeline. The design will either be a simple shift-add multiplier like seen in MP1, or a multiplier that is more advanced like a Wallace tree. Regardless of the multiplier used, there are a few components that will likely be constant throughout the operations.

First, the multiplier will sit in the EXE stage with the ALU. It will be part of the ALU but in a separate module, meaning it will share the inputs and outputs of the ALU itself. The multiplier will necessitate creating a new opcode, which will match the riscv convention from the design documents. A new opcode will require change to the control logic at each stage as well.

Furthermore, the multiplier may not execute in one cycle. If that is the case, then it will be necessary to stall the pipeline until the operation finishes, similar to a cache miss. In fact, a structural hazard occurs when a resource is busy, not just when the memory is busy. That means this would be a structural hazard and can use the same control logic.

Tournament Branch Prediction

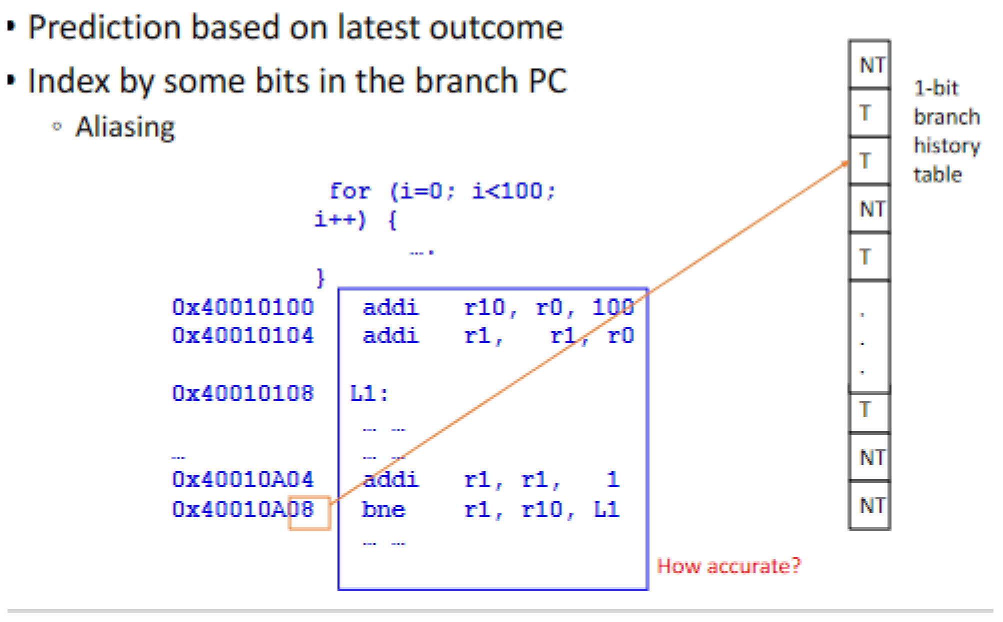
This feature will be implementing a tournament branch prediction modules in order to replace the static branch prediction. This module will fit in the same exact location as the static branch prediction is in right now.

The Tournament Branch Predictor has three main parts. One is the local branch prediction, which will be based on the simple branch prediction design presented in lecture. This predictor will use the same 2 bit FSM as well. The 2 bit FSM will give some measure of confidence to the predictor.

The second component is the global branch predictor, which will be based on the correlation prediction model presented in the slides. This will also use two bit FSM to provide the predictions.

The final component is the control to select the prediction output. This control will select the prediction with higher confidence, where 11 and 00 are more confident predictions than 10 and 01.

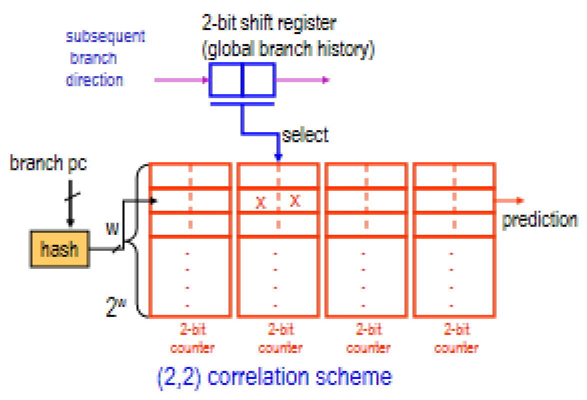
The way that we will handle branch taken predictions has not yet been decided on. It will likely involve some simple BTB implementation.



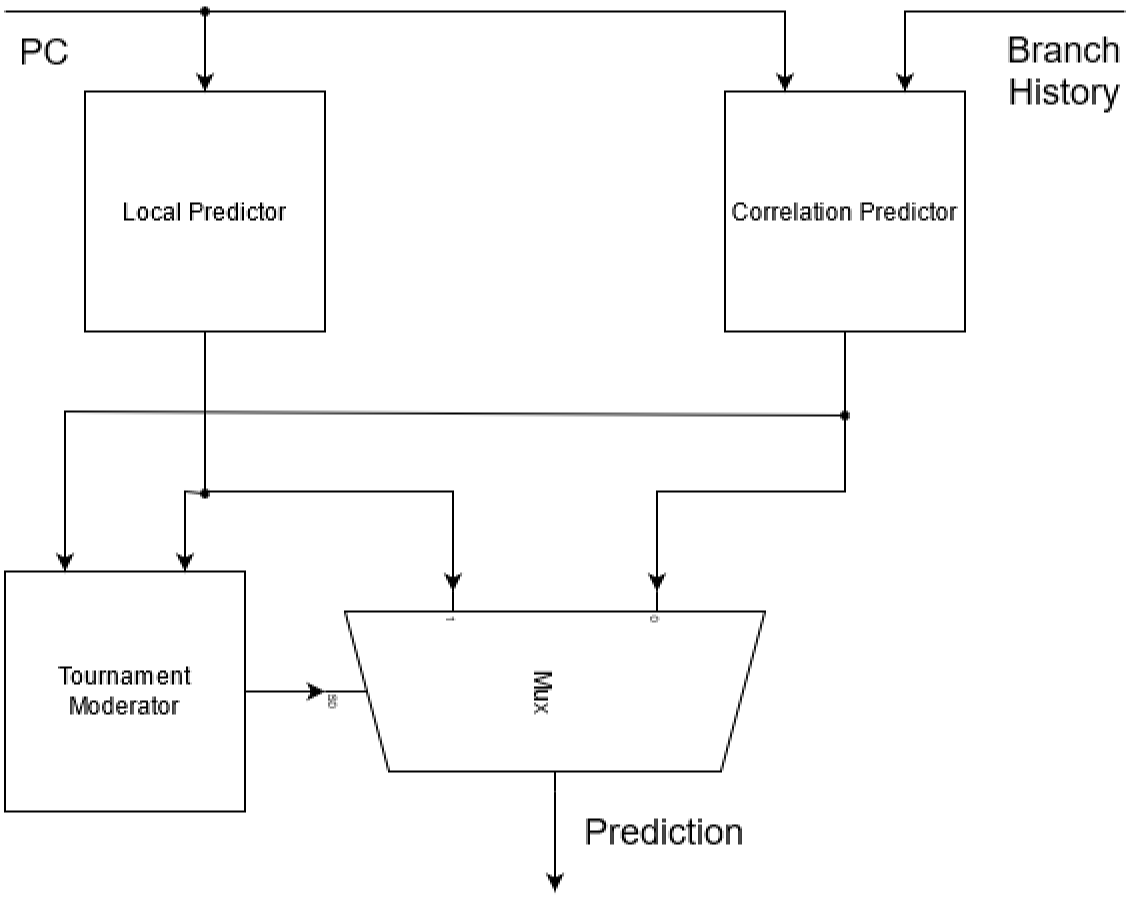
- Prediction based on latest outcome
- Index by some bits in the branch PC
  - Aliasing

```
                for (i=0; i<100;
              i++) {
                    ….
              }
0x40010100   addi   r10, r0, 100
0x40010104   addi   r1,    r1, r0

0x40010108  L1:
             … …
…            … …
0x40010A04   addi   r1, r1,   1
0x40010A08   bne    r1, r10, L1
             … …
```

Image taken from the lecture notes. The simple predictor will follow this model. The size of the history table, as well as the hashing function have not yet been decided.



The above figure is taken from the notes on correlation prediction schemes.



Short diagram showing the basic implementation of a tournament predictor. The final prediction is chosen from the two predictor results.

L2 Cache + Eviction Write Buffer

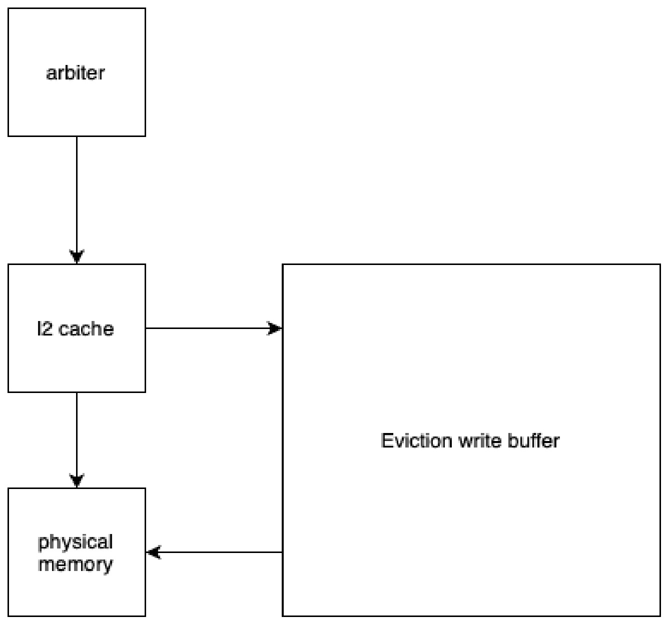
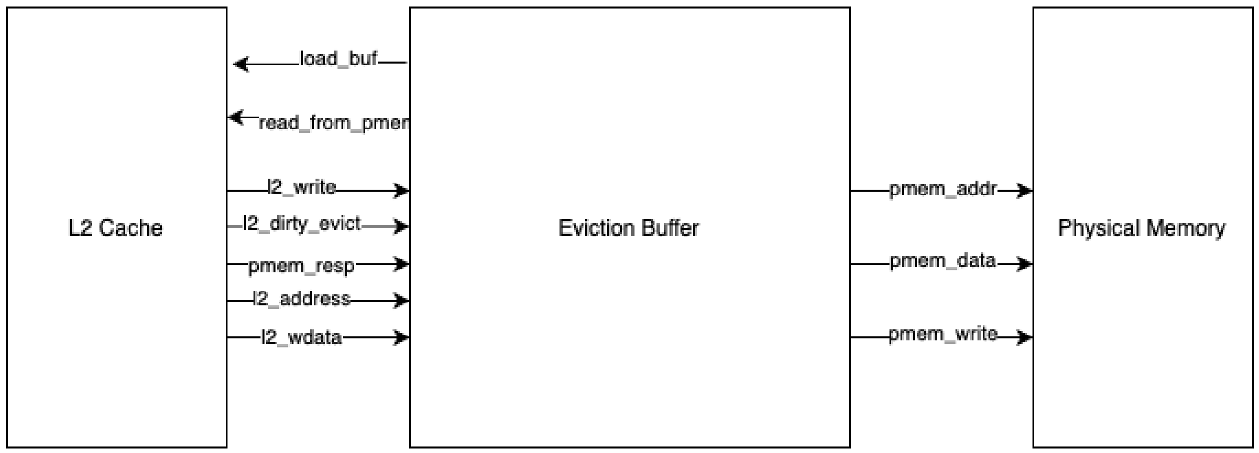
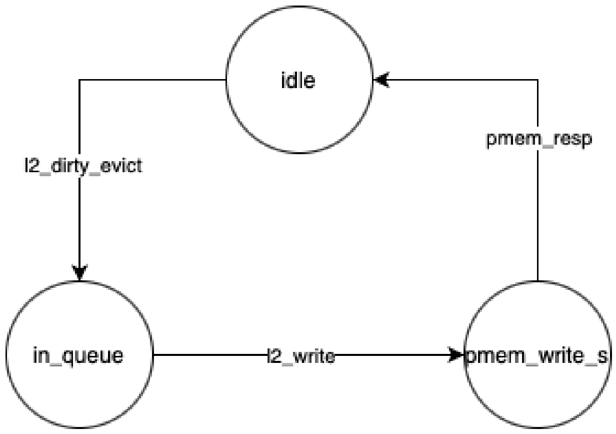


Figure (above): High level representation of the memory system with the L2 Cache and eviction write buffer. Original I_cache and D_cache are connected via the arbiter (not shown)

With the eviction write buffer, it takes an input of the l2 cache’s address and data, and outputs an address and data to the physical memory. Three signals drive the state diagram from 3 states.





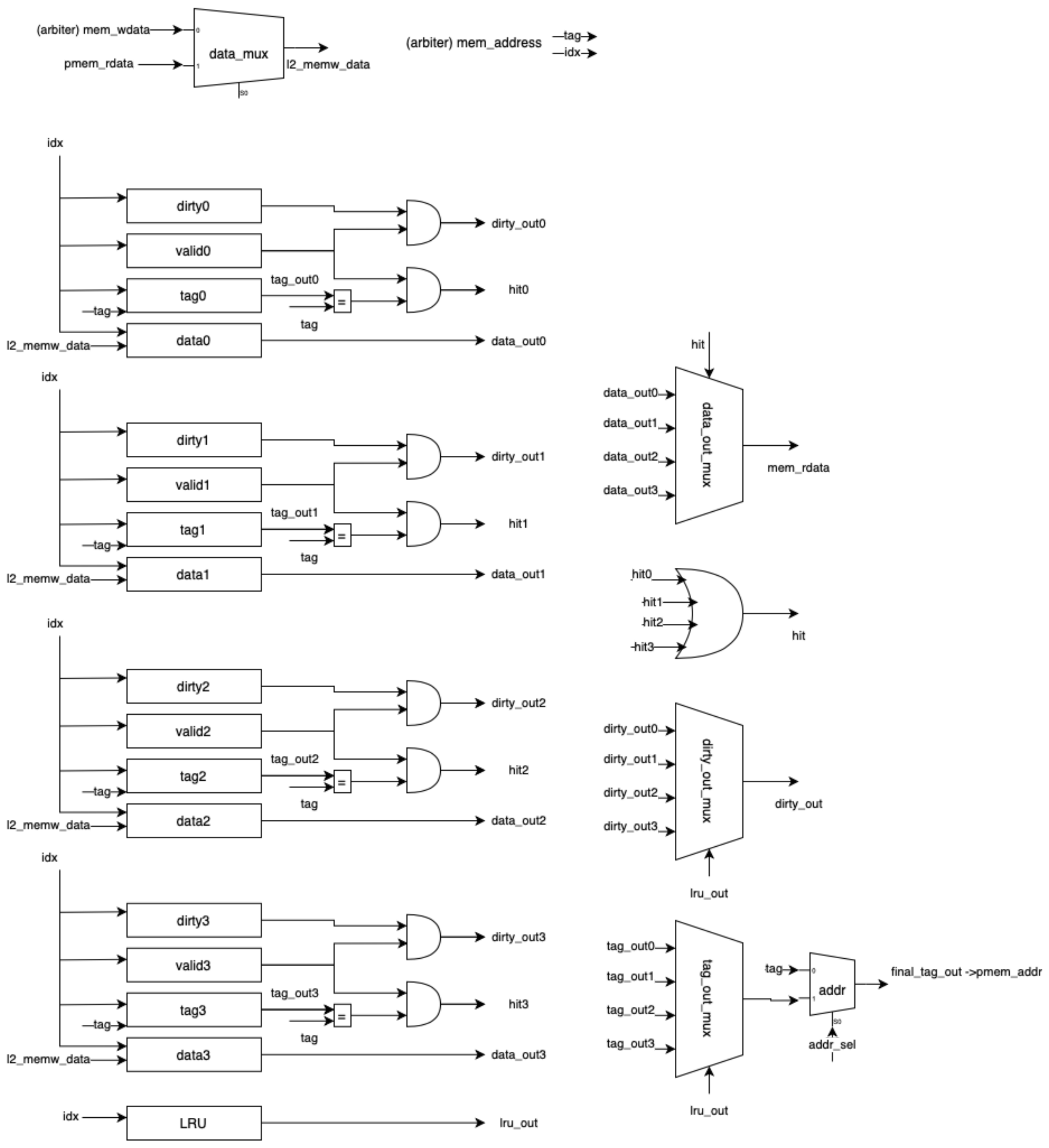
Three signals involved with eviction buffer:

Load_buf is set low on default, high when in the idle state

Read_from_pmem is set low on default high when in the in_queue state

Pmem_write is set to high when in the pmem_write_s state

Below we have the 4 way set associative L2 Cache. Note, mem_rdata goes back to the arbiter. We may expand to a parameterized cache if time allows, but the above diagram describes the 4 way L2 Cache.


(arbiter) mem_wdata
pmem_rdata
data_mux
l2_memw_data
(arbiter) mem_address
tag
idx
dirty0
valid0
tag0
data0
tag_out0
dirty_out0
hit0
data_out0
dirty1
valid1
tag1
data1
tag_out1
dirty_out1
hit1
data_out1
dirty2
valid2
tag2
data2
tag_out2
dirty_out2
hit2
data_out2
dirty3
valid3
tag3
data3
tag_out3
dirty_out3
hit3
data_out3
LRU
lru_out
hit
data_out_mux
mem_rdata
dirty_out_mux
dirty_out
tag_out_mux
addr
addr_sel
final_tag_out ->pmem_addr

Progress Report:

CP0 and CP1 work.

Functionality	Testing Strategy	Who worked on this component?
Basic RV32I pipelined datapath design	NA	All
Basic Pipeline Stages- Datapath and Control	Tested using test codes, similar to previous labs.	Andrew, Tim
Basic Pipeline- Setting up Magic Memory and Testing	^	Matthew, Andrew
Arbiter, hazard detection & forwarding design	NA	All

Roadmap:

CP2 Requirements:

- Integration of L1 caches & Arbiter (Tim)
- Hazard detection & forwarding (Matthew)
- Static branch predictor (Andrew)

The three requirements here are all closely related, so it is likely that each member will be contributing to each requirement. As an example, the hazard detection & forwarding topic relies on memory transactions to determine if the pipeline should stall (ex. on a cache miss).

Integration of L1 caches & Arbiter involves setting up the memory system detailed above, as well as setting up the testing environment to work with RVFI monitor and shadow memory.

Hazard detection involves addressing structural, data and control hazards using forwarding and prediction when possible, and stalling when we can't.

The static branch predictor will be one of the simplest branch prediction techniques, guessing that the branch is never taken. This will simplify the logic for incrementing pc after fetching the branch.

Progress Report:

CP0, CP1 work. CP2 is waiting on memory

Functionality	Testing Strategy	Who worked on this component?
Static Branch Prediction	Removing NOPs from cp1 test code near the branch statements	Andrew
Data Forwarding	Tested using test codes, similar to previous labs.	Matthew
Memory System	Shadow memory with test code	Tim, Andrew
Advanced Features Design	NA	All

Roadmap:

CP3 Requirements:

- L2+ Cache (Tim)
- RISC-V M Extension (Matthew)
- Advanced branch predictor (Andrew)
- 4+ way set associative Cache (Parameterized Cache?) (All)
- Eviction Write Buffer (All)

The 4+ way set associative Cache will likely work similarly to the 2 way set associative Cache created in MP3. This will be implemented in the L2 cache, which will interact with the arbiter and physical memory as a secondary cache. Issues that may come up include the combination of the L2 Cache with the arbiter, as well as the parameterization, so it is important for us to be completely sure the L2 Cache is fully functioning before expanding the cache to include the other advanced features.

The Eviction Write Buffer will be used in between the L2 cache and the physical memory due to the slower nature of the physical memory. We plan on implementing the L2 cache as well as make it 4 way set associative first before implementing the eviction write buffer. Most issues with the eviction write buffer will come from connections, as the state logic diagram is fairly simple, but the L2 cache has

The advanced branch predictor implementation will likely be a tournament predictor as explained above. It will use a simple local predictor and a correlation predictor as the two contestants.

This will likely also include a simple BTB with a similar design as presented in lecture.

For now, I plan on implementing an add-shift multiplier for the M extension, similar to what we did for part of MP1. However I looked into other possibilities such as Wallace tree and Dadda multipliers, and if the simple implementation proves to be easy I will likely implement a more advanced multiplier.