**ECE 411**

Spring 2020

Machine Problem 3 checkpoint #1

**RISC-V ISA**

**Basic Pipelined CPU**

Adit Umakanth, Zhifeng Ou, Hyun Do Jung

It Will Work Processor

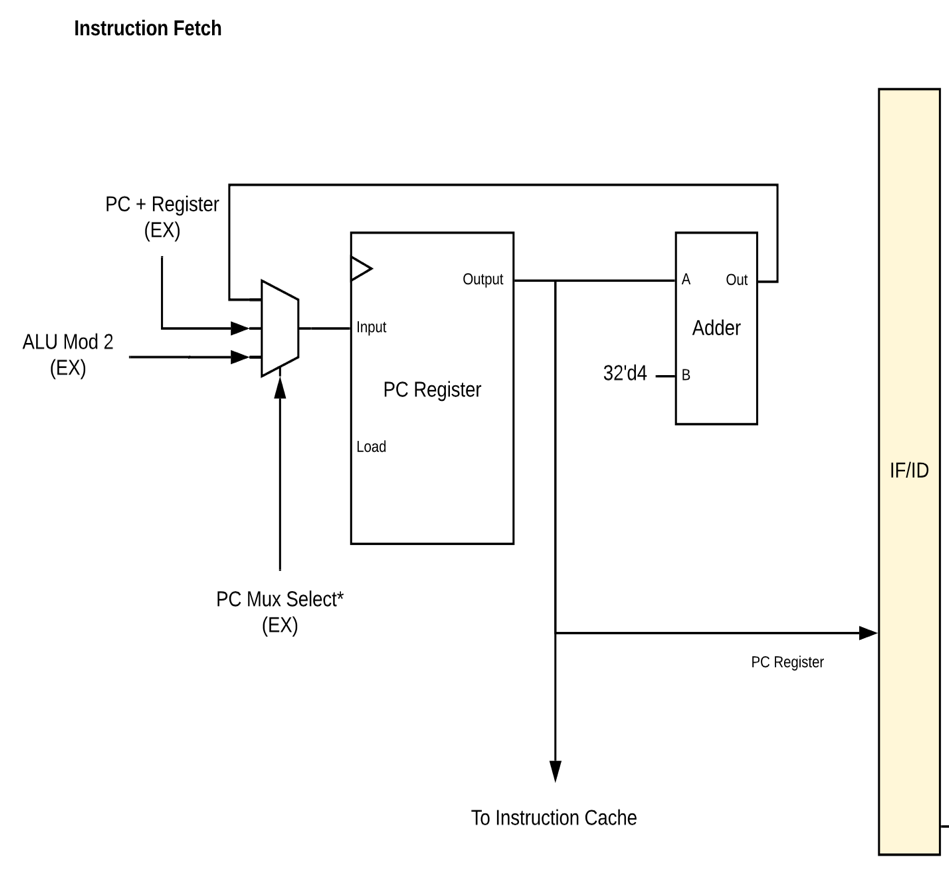
Sean T Ngo

**I. INTRODUCTION:**

In this checkpoint, we had to implement and modify the CPU, which is composed of control and datapath into 5 stages-pipelined datapath which should handle all the RV32I instructions (with a few exception of FENCE, ECALL, EBREAK, and CSRR instructions). In addition, an instruction cache, data cache, and an arbiter should be designed so that they would be implemented and integrated into the pipelined datapath in upcoming checkpoint.

**II. DOCUMENTATION:**

1. **IF.sv**

****

**figure 1. Fetch Module Block Diagram**

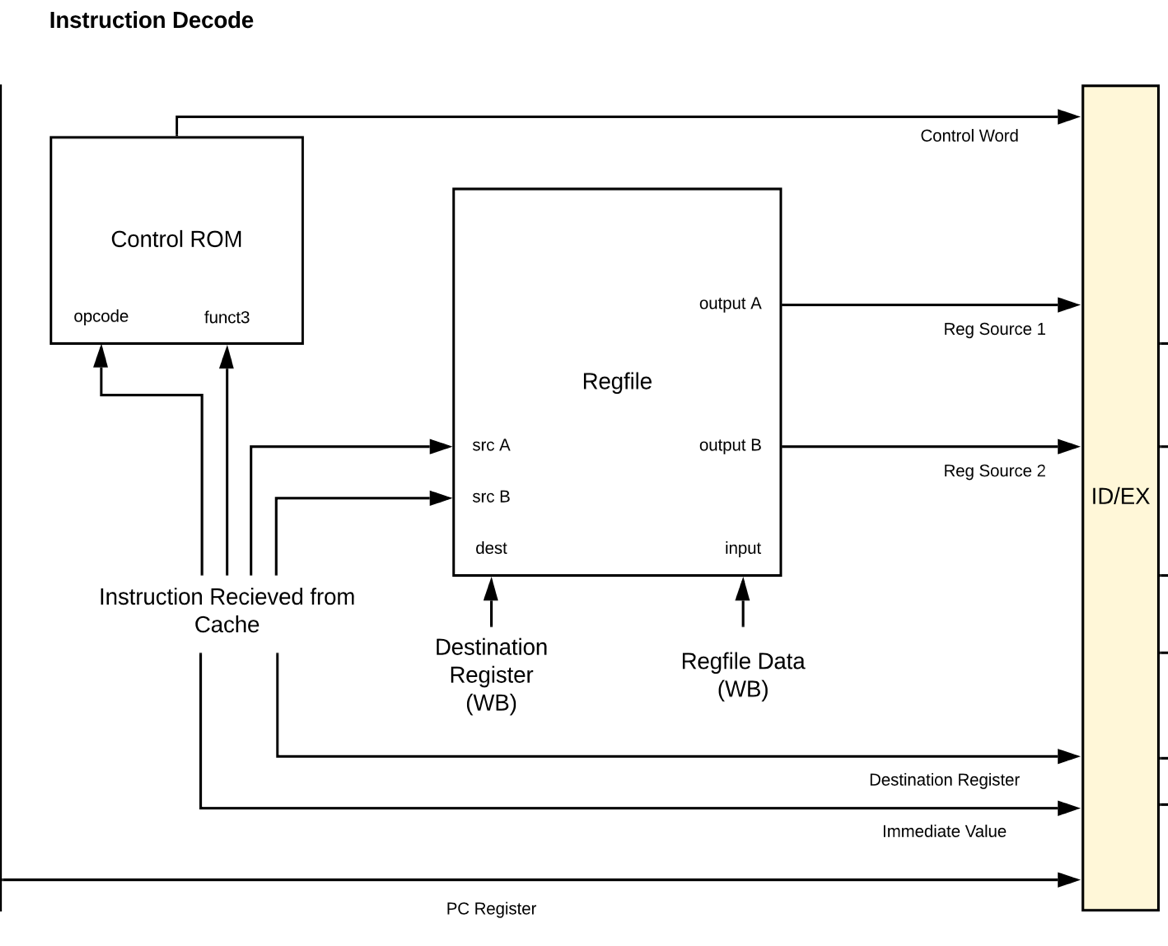
**Module**: IF

**Inputs**: clk, rst, load\_pc, [31:0] pc\_plus\_reg, pcmux\_sel, [31:0] alu\_out

**Outputs**: rv32i\_word PC\_reg\_out

**Description:** it uses pcmux toassign separate data into relevant values to update PC accordingly and store the result output in state register IF/ID for the future usage. This is basically the module that does very first three stages which are fetch 1, 2, and 3 of regular CPU control and datapath that we have done and studied in previous MPs. Here though, IR is replaced and taken out.

1. **ID.sv**

****

**figure 2. Decode Module Block Diagram**

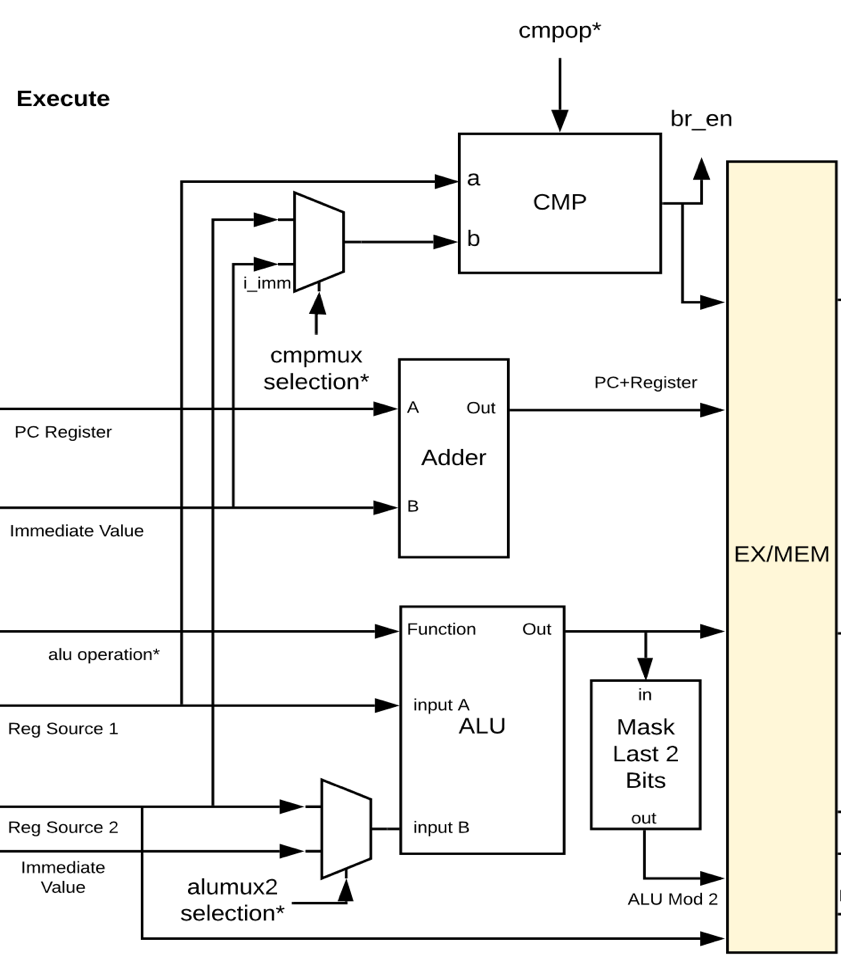
**Module**: ID

**Inputs**: clk, rst, stage\_reg\_load, [31:0] PC\_reg\_out, [4:0] regfile\_dest, [31:0] regfile\_in, regfile\_load, [31:0] instruction

**Outputs**: stage\_reg\_t stage\_reg\_out

**Description:** 2nd stage of pipelining, which is decode. This module is mainly operating with regfile register we have had in MP1 and MP2. Eventually what it does is translate each instruction into separate values we need that may be used in later stages for specific instruction operation. Control ROM mainly decodes and make these signals ready and packed within a struct along with a newly defined rv32i\_type.

1. **EX.sv**

****

**figure 3. Execute Module Block Diagram**

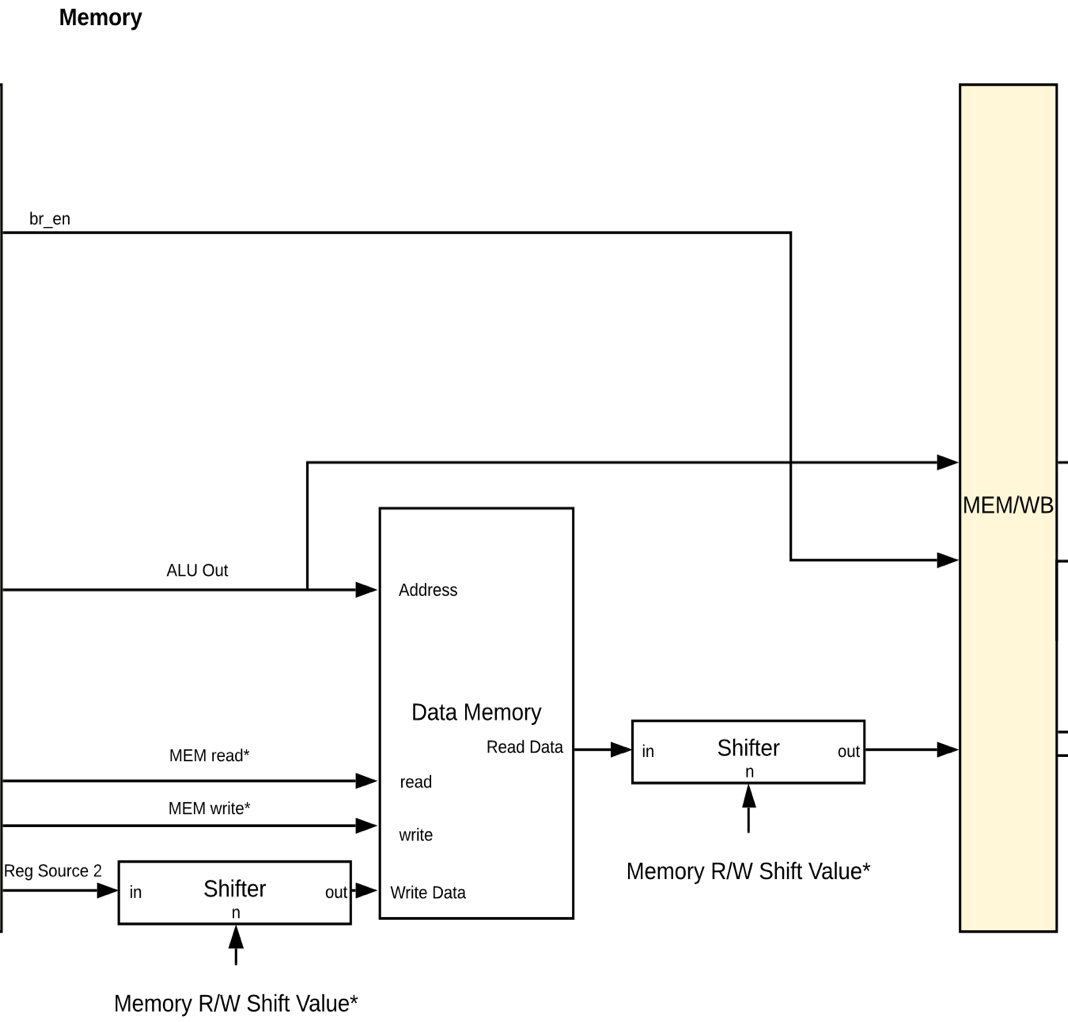
**Module**: EX

**Inputs**: clk, rst, stage\_reg\_load, stage\_reg\_t stage\_reg\_prev

**Outputs**: stage\_reg\_t stage\_reg\_out

**Description:** This module mainly deals with the ALU operation by taking all data assigned by fetch and decode module. It not only splits each instruction into corresponding data such as opcode, funct3, funct7 and so on, but also, deals with making a signal logic called br\_en which is just a signal indicator for branch enable.

1. **MEM.sv**

****

**figure 4. Memory Module Block Diagram**

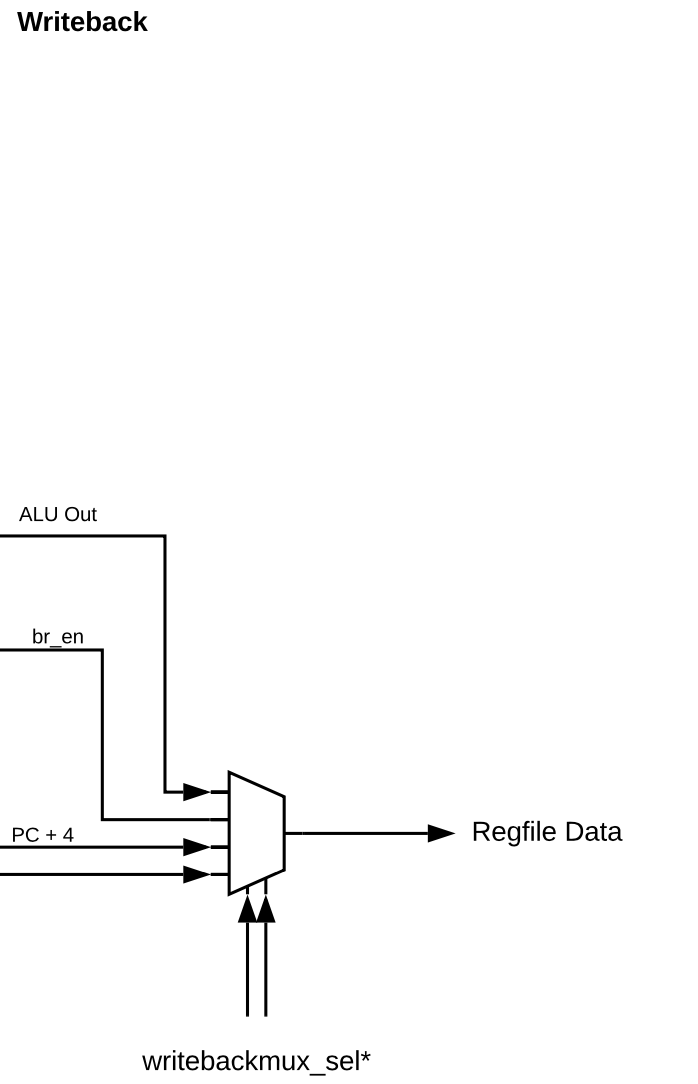
**Module**: MEM

**Inputs**: clk, rst, stage\_reg\_load, stage\_reg\_t stage\_reg\_in, rv32i\_word mem\_rdata

**Outputs**: stage\_reg\_t stage\_reg\_out, mem\_read, mem\_write, rv32i\_word mem\_wdata, mem\_addr, [3:0] mem\_byte\_enable

**Description:** This is the module that is basically dealing with all memory signals and generates physical memory signals for actual physical memory. For simplicity purpose, we use magic dual port yet while later checkpoints we are going to use different memory style such that mem\_resp is always set to be high so that we don’t have to handle any cache misses or memory stall issues. The module communicates with the actual memory. Using wmask and rmask is very similar to what we have done in previous MPs that handles where memory byte we are accessing for load and store instructions

1. **WB.sv**

****

**figure 5. Writeback Module Block Diagram**

**Module**: WB

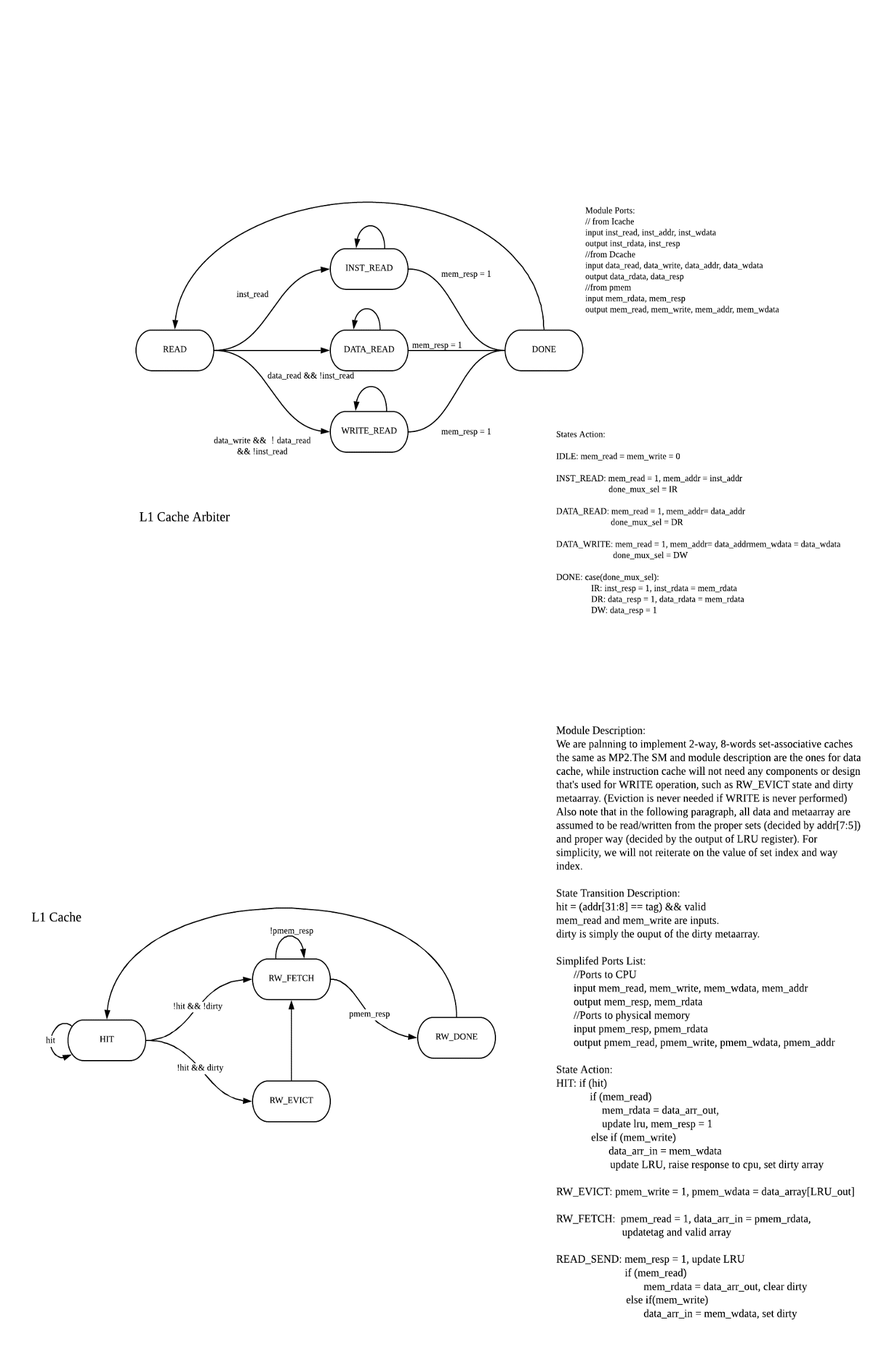
**Inputs**: clk, rst, stage\_reg\_load, stage\_reg\_t stage\_reg\_in

**Outputs**: stage\_reg\_t stage\_reg\_out, rv32i\_word regfile\_data, [4:0] rd

**Description:** The last stage of pipelining. we take ALU output, branch enable signal, PC+4, pc\_plus\_reg and mem\_rdata so that each of these can be selected inside the writeback mux accordingly and outputs regfile\_data that will go back to previous stages and update.

**IV. CACHE HIERARCHY & ROADMAP :**

1. **Cache hierarchy**

****

**figure 6. Cache Hierarchy Block Diagram**

The above diagram is our caches and an arbiter design for the upcoming checkpoint that we intend to implement two caches of similar design with a little difference within each cache because I-cache only deals with reading not writing. Arbiter logic will be located between this I-cache and D-cache to determine such edge cases we may meet for example, instruction trying to reach both I-cache and D-cache and which one should be dealt first while not losing capability of doing rest of jobs on another cache too before continuing to the next instruction we want to handle.

1. **Roadmap for next checkpoint**

Refer the following page for the figure.

**V. CONTRIBUTION:**

Putting together the separate stages as a team went smoothly after discussing some basics for how we would work together. The nature of this pipelined CPU allowed us to efficiently distribute work on different stages and aspects of the project as follows:

* **Adit**: Instruction Decode stage, Execute stage, general debugging
* **Hyun**: Instruction Fetch stage, Instruction Decode stage, report creation
* **Zhifeng**: Memory stage, Writeback stage, general debugging, cache design

스크린샷이(가) 표시된 사진

자동 생성된 설명

**figure 7. Roadmap for Checkpoint 2**