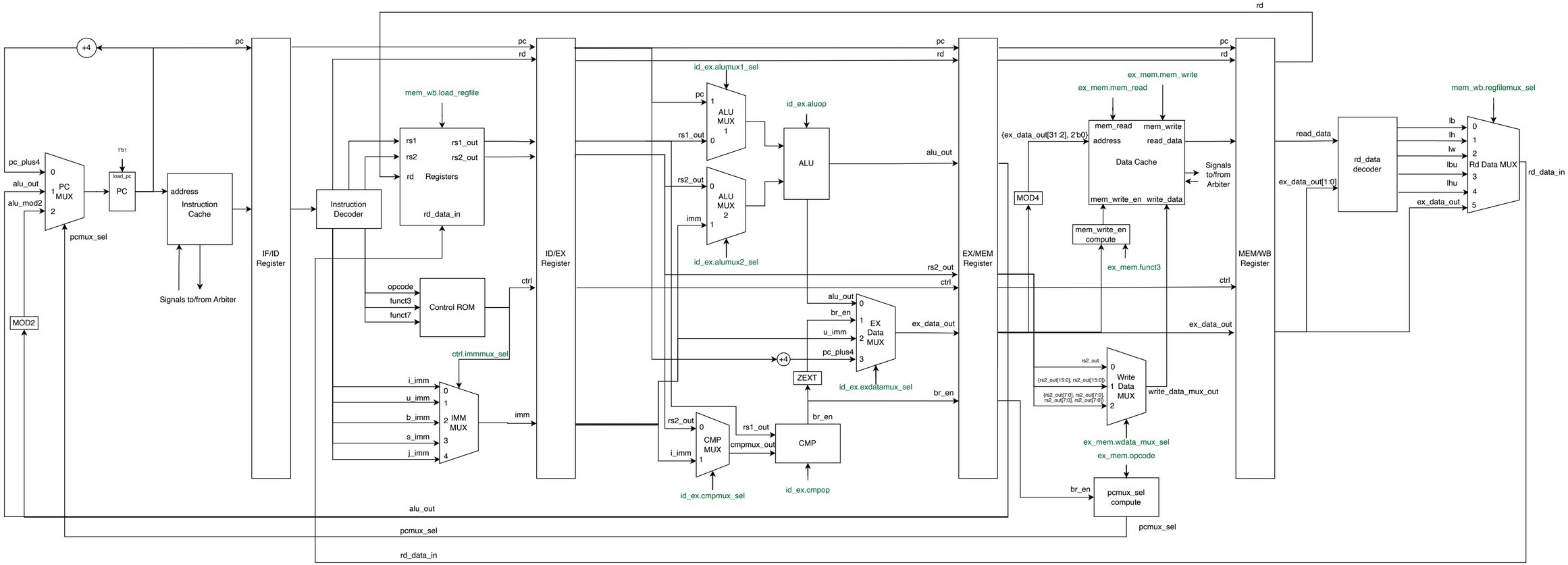
ECE 411 MP4
Final Report

Neha Agarwal
Yu Li
Mitchell Bifeld
TA: Yian Wang

May 10th, 2023


+4
pc
pc_plus4
alu_out
alu_mod2
0
1
2
PC MUX
pcmux_sel
1'b1
load_pc
PC
address
Instruction Cache
Signals to/from Arbiter
MOD2
IF/ID Register
pc
rd
Instruction Decoder
mem_wb.load_regfile
rs1
rs2
rd
Registers
rs1_out
rs2_out
rd_data_in
opcode
funct3
funct7
Control ROM
ctrl
ctrl.immmux_sel
i_imm
u_imm
b_imm
s_imm
j_imm
IMM MUX
imm
ID/EX Register
id_ex.alumux1_sel
ALU MUX 1
id_ex.aluop
ALU
alu_out
id_ex.alumux2_sel
ALU MUX 2
rs2_out
br_en
u_imm
pc_plus4
EX Data MUX
ex_data_out
id_ex.exdatamux_sel
ZEXT
CMP MUX
cmpmux_out
rs1_out
CMP
id_ex.cmpop
id_ex.cmpmux_sel
EX/MEM Register
MOD4
{ex_data_out[31:2], 2'b0}
ex_mem.mem_read
ex_mem.mem_write
mem_read
mem_write
address
read_data
Data Cache
mem_write_en
write_data
Signals to/from Arbiter
mem_write_en compute
ex_mem.funct3
Write Data MUX
write_data_mux_out
{rs2_out[15:0], rs2_out[15:0]}
{rs2_out[7:0], rs2_out[7:0], rs2_out[7:0], rs2_out[7:0]}
ex_mem.wdata_mux_sel
ex_mem.opcode
pcmux_sel compute
MEM/WB Register
read_data
ex_data_out[1:0]
rd_data decoder
mem_wb.regfilemux_sel
lb
lh
lw
lbu
lhu
Rd Data MUX
rd_data_in
ex_data_out
alu_out
pcmux_sel
rd_data_in

Controller Datapath Addendum:

Following is pseudo code for the non-standard blocks in the datapath which were not expanded in the datapath for clarity.

pcmux_sel_compute:

```
if ((op_br & br_en) | op_jal)
      pcmux_sel = pcmux::alu_out;
else if (op_jalr)
      pcmux_sel = pcmux::aluout_mod2;
else
      pcmux_sel = pcmux::pc_plus4;
```

rd_data_encoder:

```
lw = read_data
lh = sext32(addr[1] ? read_data[31:16] : read_data[15:0]);
lhu = zext32(addr[1] ? read_data[31:16] : read_data[15:0]);
lb = sext32(addr[0] ? lh[15:8] : lh[7:0]);
lbu = zext32(addr[0] ? lh[15:8] : lh[7:0]);
```

mem_write_en_compute:

```
case(funct3)
      3'b000: mem_write_en = (4'b0001 << ex_data_out[1:0]);
      3'b001: mem_write_en = (4'b0011 << ex_data_out[1:0]);
      3'b010: mem_write_en = 4'b1111;
      defaults: mem_write_en = 4'b0000;
endcase
```

Controller Description:

There is no state machine for the pipeline controller as this is a single cycle design. However, the control unit does output various signals to control the pipeline. As the implementation of the controller is simply setting the control signals based on the opcode, funct3, and funct7 of the current instruction, the entire controller implementation is provided.

```
import rv32i_types::*;

module control_rom
(
    input rv32i_opcode opcode,
    input [2:0] funct3,
    input [6:0] funct7,
    output rv32i_control_word ctrl
);

always_comb
begin
    /* Default assignments */
    ctrl.opcode = opcode;
    crtl.funct3 = funct3;
    ctrl.immmux_sel = immmux::u_imm;
    ctrl.load_regfile = 1'b0;
    ctrl.aluop = alu_ops'(funct3);
    ctrl.cmpop = cmp_ops'(funct3);
    ctrl.pcmux_sel = pcmux::pc_plus4;
    ctrl.alumux1_sel = alumux1::rs1_out;
    ctrl.alumux2_sel = alumux2::rs2_out;
    ctrl.cmpmux_sel = cmpmux::rs2_out;
    ctrl.exdatamux_sel = exdatamux::alu_out;
    ctrl.regfilemux_sel = regfilemux::mem_data;
    ctrl.writedatamux_sel = writedatamux::word;
    ctrl.mem_read = 1'b0;
    ctrl.mem_write = 1'b0;

    /* Assign control signals based on opcode */
    case(opcode)
        op_auipc: begin
            ctrl.aluop = alu_add;
            ctrl.alumux1_sel = alumux1::pc_out;
            ctrl.alumux2_sel = alumux2::imm;//u_imm
            ctrl.load_regfile = 1'b1;
            ctrl.immmux_sel = immmux::u_imm;
            ctrl.regfilemux_sel = regfilemux::ex_data_out; // Main computation result of EX stage
            ctrl.exdatamux_sel = exdatamux::alu_out;
        end
        op_lui: begin
            ctrl.load_regfile = 1'b1;
            ctrl.regfilemux_sel = regfilemux::ex_data_out;
            ctrl.immmux_sel = immmux::u_imm;
            ctrl.exdatamux_sel = exdatamux::u_imm;
        end
        op_br: begin
            ctrl.cmpop = funct3;
            ctrl.alumux1_sel = alumux1::pc_out;
            ctrl.alumux2_sel = alumux2::imm;//b_imm
            ctrl.alu_ops = alu_add;
            ctrl.immmux_sel = immmux::b_imm;
            ctrl.cmpmux_sel = cmpmux::rs2_out;
        end
        op_load: begin
            ctrl.alu_ops = alu_add;
            ctrl.mem_read = 1'b1;
            ctrl.alumux1_sel = alumux1::rs1_out;
            ctrl.alumux2_sel = alumux2::imm;
            ctrl.exdatamux_sel = exdatamux::alu_out;
            ctrl.load_regfile = 1'b1;
            ctrl.immmux_sel = immmux::i_imm;
            unique case(funct3)
                lb:  ctrl.regfilemux_sel = regfilemux::lb;
                lh:  ctrl.regfilemux_sel = regfilemux::lh;
                lbu: ctrl.regfilemux_sel = regfilemux::lbu;
                lhu: ctrl.regfilemux_sel = regfilemux::lhu;
                lw:  ctrl.regfilemux_sel = regfilemux::lw;
                default: ctrl.regfilemux_sel = regfilemux::ex_data_out;
            endcase
        end
        op_store: begin
            ctrl.mem_write = 1'b1;
            ctrl.alumux1_sel = alumux1::rs1_out;
            ctrl.alumux2_sel = alumux2::imm;
```

```
            ctrl.aluop = alu_add;
            ctrl.exdatamux_sel = exdatamux::alu_out;
            ctrl.immmux_sel = immmux::s_imm;
            case(funct3)
                sb: ctrl.writedatamux_sel = writedatamux::byte;
                sh: ctrl.writedatamux_sel = writedatamux::half;
                sw: ctrl.writedatamux_sel = writedatamux::word;
                default: ctrl.writedatamux_sel = writedatamux::byte;
            endcase
        end
        op_imm: begin
            ctrl.load_regfile = 1'b1;
            ctrl.regfilemux_sel = regfilemux::ex_data_out;
            ctrl.immmux_sel = immmux::i_imm;
            ctrl.alumux1_sel = alumux1::rs1_out;
            ctrl.alumux2_sel = alumux2::imm;
            if(funct3 == slt) begin
                ctrl.cmpmux_sel = cmpmux::i_imm;
                ctrl.cmpop = lt;
                ctrl.exdatamux_sel = exdatamux::br_en;
            end
            else if (funct3 == sltu) begin
                ctrl.cmpmux_sel = cmpmux::i_imm;
                ctrl.cmpop = ltu;
                ctrl.exdatamux_sel = exdatamux::br_en;
            end
            else if (funct3 == sr) begin
                ctrl.exdatamux_sel = exdatamux::alu_out;
                if(funct7[5]) ctrl.aluop = alu_sra;
                else          ctrl.aluop = alu_srl;
            end
            else begin
                ctrl.exdatamux_sel = exdatamux::alu_out;
                ctrl.aluop = alu_ops'(funct3);
            end
        end
        op_reg: begin
            ctrl.load_regfile = 1'b1;
            ctrl.regfilemux_sel = regfilemux::ex_data_out;
            ctrl.alumux1_sel  = alumux1::rs1_out;
            ctrl.alumux2_sel  = alumux2::rs2_out;
            if(funct3 == slt)begin
                ctrl.cmpmux_sel = cmpmux::rs2_out;
                ctrl.cmpop = lt;
                ctrl.exdatamux_sel = exdatamux::br_en;
            end
            else if (funct3 == sltu)begin
                ctrl.cmpmux_sel = cmpmux::rs2_out;
                ctrl.cmpop = ltu;
                ctrl.exdatamux_sel = exdatamux::br_en;
            end
            else if (funct3 == sr)begin
                ctrl.exdatamux_sel = exdatamux::alu_out;
                if(funct7[5]) ctrl.aluop = alu_sra;
                else          ctrl.aluop = alu_srl;
            end
            else if (funct3 == add) begin
                ctrl.exdatamux_sel = exdatamux::alu_out;
                if(funct7[5]) ctrl.aluop = alu_sub;
                else          ctrl.aluop = alu_add;
            end
            else begin
                ctrl.exdatamux_sel = exdatamux::alu_out;
                ctrl.aluop = alu_ops'(funct3);
            end
        end
        op_jal: begin
            ctrl.load_regfile = 1'b1;
            ctrl.regfilemux_sel = regfilemux::ex_data_out;
            ctrl.exdatamux_sel = exdatamux::pc_plus4;
            ctrl.alumux1_sel = alumux1::pc_out;
            ctrl.alumux2_sel = alumux2::imm;
            ctrl.aluop = alu_add;
            ctrl.immmux_sel = immmux::j_imm;
        end
```

```
        op_jalr: begin
            ctrl.load_regfile = 1'b1;
            ctrl.regfilemux_sel = regfilemux::ex_data_out;
            ctrl.exdatamux_sel = exdatamux::pc_plus4;
            ctrl.alumux1_sel = alumux1::rs1_out;
            ctrl.alumux2_sel = alumux2::imm;
            ctrl.aluop = alu_add;
            ctrl.immmux_sel = immmux::i_imm;
        end

        default: begin
            ctrl = 0;   /* Unknown opcode, set control word to zero */
        end
    endcase
end
endmodule : control_rom
```

MUX Definitions:

```
package forwardmux;
typedef enum bit [1:0] {
    rs_out  = 2'b00,
    ex_mem_out  = 2'b01,
    mem_wb_out  = 2'b10
} forwardmux_sel_t;
endpackage
```

Forwarding Unit:
Below is the pseudo code for the forwarding unit shown in the datapath.

```
// source register 1 conflicts
forward1 = forwardmux::rs_out; // Output from register unit
if (id_ex.rs1 != 0)
      if (ex_mem.ctrl.load_regfile & (ex_mem.rd == id_ex.rs1)
            forward1 = forwardmux::ex_mem_out; // ex_mem.ex_data_out
      else if (mem_wb.ctrl.load_regfile) & (mem_wb.rd == id_ex.rs1)
            forward1 = forwardmux::mem_wb_out; // mem_wb.rd_data_in

// source register 2 conflicts  (same as above but for second mux)
forward2 = forwardmux::rs_out;
if (id_ex.rs2 != 0)
      if (ex_mem.ctrl.load_regfile) & (ex_mem.rd == id_ex.rs2)
            forward2 = forwardmux::ex_mem_out;
      else if (mem_wb.ctrl.load_regfile) & (mem_wb.rd == id_ex.rs2)
            forward2 = forwardmux::mem_wb_out;
```

Hazard Detection Unit:

```
hz_pc_ld = 1;
hz_if_id_ld = 1;
hz_if_id_rst = 0;
hz_id_ex_rst = 0;
hz_ex_mem_rst = 0;

// Data hazard
no_sr1 = [op_auipc, op_lui, op_jal] // Instr which don't read sr1
no_sr2 = [op_auipc, op_lui, op_jal, op_load, op_imm]
```

```
if (id_ex.opcode == op_load
& id_ex.rd != 0
& ((if_id.opcode not in no_sr1) & (id_ex.rd == if_id.sr1))
| (if_id.opcode not in no_sr2) & (id_ex.rd == if_id.sr2)))
      // Stall pipeline and insert nop in ex stage
      hz_pc_ld = 0    // no load pc
      hz_if_id_ld = 0 // no load if_id
      hz_id_ex_rst = 1 // Insert nop

// Control hazard
if ((ex_mem.opcode == op_br & br_en) | ex_mem.opcode == op_jal |
ex_mem.opcode == op_jalr)
      hz_if_id_rst = 1 // Reset if_id
      hz_id_ex_rst = 1
      hz_ex_mem_rst = 1
```

Arbiter Design

The interface for the arbiter is as follows:

```
module arbiter
(
    input clk,
    input rst,

    /* Instruction memory signals */
    input   logic [31:0]    imem_address,
    output  logic [255:0]   imem_rdata,
    input   logic [255:0]   imem_wdata,
    input   logic           imem_read,
    input   logic           imem_write,
    output  logic           imem_resp,

    /* Data memory signals */
    input   logic [31:0]    dmem_address,
    output  logic [255:0]   dmem_rdata,
    input   logic [255:0]   dmem_wdata,
    input   logic           dmem_read,
    input   logic           dmem_write,
    output  logic           dmem_resp,

    /* Physical memory signals */
    output  logic [31:0]    pmem_address,
    input   logic [255:0]   pmem_rdata,
    output  logic [255:0]   pmem_wdata,
    output  logic           pmem_read,
    output  logic           pmem_write,
    input   logic           pmem_resp
);
```

The arbiter itself can be described entirely by its state machine.

Controller Description:

Default signals:

pmem_address = dmem_address;

pmem_wdata = dmem_wdata;

pmem_read = 1'b0;

pmem_write = 1'b0;

dmem_resp = 1'b0;

dmem_rdata = pmem_rdata;

imem_resp = 1'b0;

imem_rdata = pmem_rdata;

State Name	**Description**	**Transition Conditions**	**Outputs**

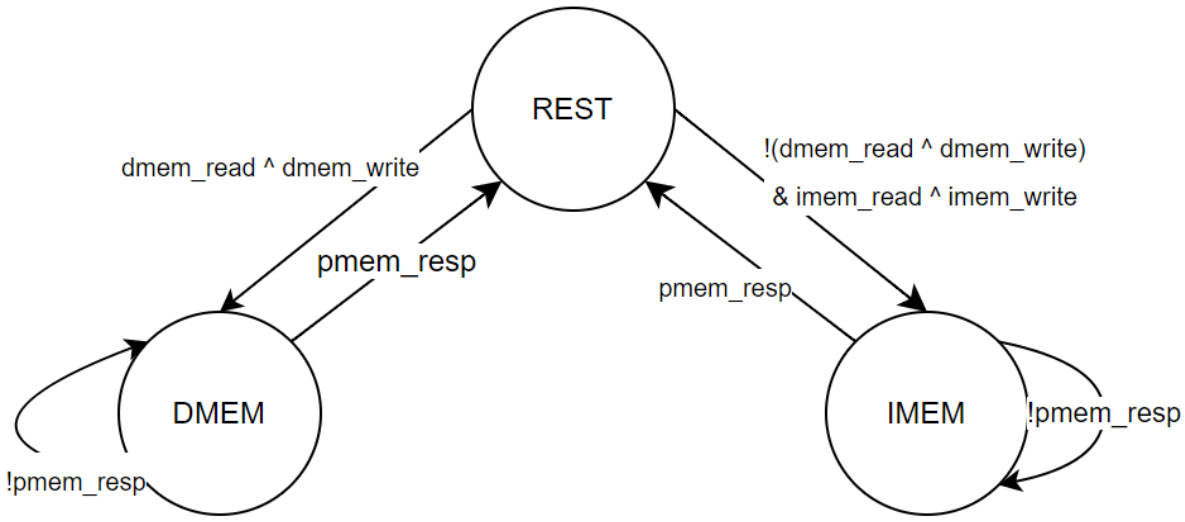
REST	Rest state. No physical memory read/writes in progress.	if (dmem_read ^ dmem_write)
		  DMEM
		else if (imem_read ^ imem_write)
		  IMEM
		else
		  REST
			if (dmem_read ^ dmem_write)
			  pmem_read = dmem_read;
			  pmem_write = dmem_write;
			  dmem_resp = pmem_resp;
			else if (imem_read ^ imem_write)
			  pmem_read = imem_read;
			  pmem_write = imem_write;
			  imem_resp = pmem_resp;
			  pmem_address = imem_address;
			  pmem_wdata = imem_wdata;

DMEM	Accessing physical memory for data.	if (pmem_resp)
		  REST
		else
		  DMEM
			pmem_read = dmem_read;
			pmem_write = dmem_write;
			dmem_resp = pmem_resp;

IMEM	Accessing physical memory for instruction.	if (pmem_resp) -> REST
		else -> IMEM
			pmem_read = imem_read;
			pmem_write = imem_write;
			imem_resp = pmem_resp;
			pmem_address = imem_address;
			pmem_wdata = imem_wdata;

Note that this design assumes that main memory accesses take at least 2 cycles to complete.

State Diagram:



Arbiter Integration:

In order to integrate the multicycle memory system with the processor, the processor must be able to be stalled. In this design, whenever there is a cache miss on either the instruction or data cache, the entire pipeline will be stalled by pausing loads on the pc and the four pipeline registers. Note that these load signals will override the reset signals generated by the hazard detection unit. This stalling logic is contained in the datapath but is replicated here for clarity.

```
instr_stall = !instruction_cache.mem_resp

data_stall = (ex_mem.ctrl.mem_write | ex_mem.ctrl.mem_read) &
!data_cache.mem_resp

pipeline_stall = instr_stall | data_stall

pc.load = !pipeline_stall & hz_pc_ld
```

```
if_id.load = !pipeline_stall & hz_if_id_ld

id_ex.load = !pipeline_stall

ex_mem.load = !pipeline_stall

mem_wb.load = !pipeline_stall

if_id.rst = !pipeline_stall & hz_if_id_rst // or main reset

id_ex.rst = !pipeline_stall & hz_id_ex_rst // or main reset

ex_mem.rst = !pipeline_stall & hz_ex_mem_rst // or main reset
```

Progress Report (3/10-3/29)

Over the last two weeks, we completed the requirements for Checkpoint 1 as well as parts of checkpoint 2.

For checkpoint 1 content, the basic division of labor was as follows:

- Implementation of basic RV32I pipelined datapath: Mitchell
- Arbiter, hazard detection, and forwarding design: Neha
- Arbiter, hazard detection, and forwarding design verification: Mitchell
- Pipeline implementation verification: Yu

The basic pipelined datapath was initially tested and debugged using the mp4-cp1.s test code provided by the TA's. Once this test successfully passed, the design was further tested by writing extra assembly code, which tested each instruction independently, and inspecting the waveform to check if the outcome matches the expectation.

For checkpoint 2 content that was implemented, the division of labor was as follows:

- Integration of caches, arbiter, hazard detection, forwarding, and static branch prediction: Neha
- RVFI + verification: Yu

First, hazard detection, forwarding, and static branch prediction were implemented and tested using the magic memory from checkpoint 1. Assembly test code was written to test each of the different forwarding, data hazard, and control hazard possibilities. To ensure correct behavior of this test code, Verdi was used to trace the timing for each of the instructions. Once this part of the design was verified, the arbiter was implemented and integrated with the given cache. From here, the same testing procedure was repeated, and we ensured that the mp4-cp2.s test code also passed. Finally, once the RVFI was correctly hooked up, the test code was rerun to ensure that our design matches the specifications.

The fmax and total power for our design as of now is given below. The full timing and power reports are at the end of this document.

slack = 5.15

fmax = 1/(10-slack) * 1000 = 206.2 MHz

total power = 1.78e+03 uW

A datapath for our design up to this point is given at the end of this document.

Roadmap (3/29-4/12)

As our next step, we plan to start working on designing the advanced features. The following is a list of advanced features we are planning to look into:

Cache organization and design

- L2+ cache system[2]
- 4-way set associative cache[2]
- Parameterized cache[points up to TA discretion]

Branch prediction

- Local branch history table[2]
- Global 2-level branch history table[3]
- Tournament branch predictor[5]
- Software branch predictor model [2]

Advanced cache

- Victim Cache[6]

Other than the advanced features, we are also looking for a more efficient branch stall design and adding the instruction for execution of coremark.

The work division for now is as follows:

- Cache organization and design options: Neha
- More efficient control hazard handling & victim cache: Mitchell
- Branch prediction & coremark: Yu

Each member is responsible for the design, implementation, and verification of their respective part. Our main goal for the next two weeks will be to do the design component.

ex_mem.ctrl.opcode
ex_mem.br_en
Hazard Detection Unit
hz_pc_ld
hz_if_id_ld
hz_if_id_rst
hx_id_ex_ld
hz_id_ex_rst
hz_ex_mem_rst
id_ex.rd
id_ex.ctrl.opcode
id_ex.sr1
id_ex.sr2
Forwarding Unit
ex_mem.ctrl.load_regfile
ex_mem.rd
mem_wb.ctrl.load_regfile
mem_wb.rd
rd
+4
pc
IF/ID Register
pc
rd
sr1
sr2
mem_wb.ctrl.load_regfile
!instr_stall & !data_stall & hz_pc_ld
instr_stall
pc_plus4
alu_out
alu_mod2
PC MUX
load
PC
mem_resp
address
Instruction Cache
rdata
pcmux_sel
Signals to/from Arbiter
MOD2
load
rst
!instr_stall & !data_stall & hz_if_id_ld
!instr_stall & !data_stall & hz_if_id_rst
Instruction Decoder
rs1
rs2
rd
Registers
rs1_out
rs2_out
rd_data_in
opcode
funct3
funct7
Control ROM
ctrl
ctrl.immmux_sel
i_imm
u_imm
b_imm
s_imm
j_imm
IMM MUX
imm
!instr_stall & !data_stall
!instr_stall & !data_stall & hz_id_ex_rst
ID/EX Register
forwardmux1_sel
forwardmux2_sel
rs_out
mem_wb_out
ex_mem_out
Forward MUX 1
Forward MUX 2
id_ex.ctrl.alumux1_sel
id_ex.ctrl.alumux2_sel
ALU MUX 1
ALU MUX 2
rs1_out
rs2_out
pc
imm
id_ex.ctrl.aluop
ALU
alu_out
br_en
u_imm
pc_plus4
EX Data MUX
ex_data_out
ZEXT
id_ex.ctrl.exdatamux_sel
rs2_out
rs1_out
CMP MUX
cmpmux_out
CMP
i_imm
id_ex.ctrl.cmpmux_sel
id_ex.ctrl.cmpop
load
rst
!data_stall & !instr_stall
!data_stall & !instr_stall & hx_ex_mem_rst
EX/MEM Register
pc
rd
alu_out
rs2_out
ctrl
br_en
ex_data_out
ex_mem.ctrl.mem_write
ex_mem.ctrl.mem_read
data_stall
{ex_data_out[31:2], 2'b0}
mem_read
mem_write
mem_resp
address
read_data
Data Cache
mem_write_en
write_data
Signals to/from Arbiter
MOD4
mem_write_en compute
ex_mem.ctrl.funct3
rs2_out
{rs2_out[15:0], rs2_out[15:0]}
{rs2_out[7:0], rs2_out[7:0], rs2_out[7:0], rs2_out[7:0]}
Write Data MUX
write_data_mux_out
ex_mem.ctrl.wdata_mux_sel
ex_mem.ctrl.opcode
br_en
pcmux_sel compute
pcmux_sel
load
!data_stall & !instr_stall
MEM/WB Register
pc
rd
ctrl
ex_data_out
mem_wb.ctrl.regfilemux_sel
read_data
ex_data_out[1:0]
rd_data decoder
addr[1:0]
rd_data_in
ex_data_out
alu_out
pcmux_sel
rd_data_in
Arbiter
signals to/from instruction cache
signals to/from data cache
signals to/from physical memory

```
Information: Updating design information... (UID-85)
Warning: There are 291 switching activity information conflicts. (PWR-19)
Information: Propagating switching activity (high effort zero delay
simulation). (PWR-6)
Warning: Design has unannotated sequential cell outputs. (PWR-415)

****************************************
Report : power
        -hier
        -analysis_effort high
Design : mp4
Version: R-2020.09-SP4
Date   : Mon Mar 27 19:37:58 2023
****************************************


Library(s) Used:

    NangateOpenCellLibrary (File: /class/ece411/freepdk-45nm/stdcells.db)


Operating Conditions: typical   Library: NangateOpenCellLibrary
Wire Load Model Mode: top

Design        Wire Load Model            Library
------------------------------------------------
mp4                   5K_hvratio_1_1    NangateOpenCellLibrary


Global Operating Voltage = 1.1
Power-specific unit information :
    Voltage Units = 1V
    Capacitance Units = 1.000000ff
    Time Units = 1ns
    Dynamic Power Units = 1uW    (derived from V,C,T units)
    Leakage Power Units = 1nW


--------------------------------------------------------------------------------
-------
                                        Switch   Int      Leak     Total
Hierarchy                               Power    Power    Power
Power    %
--------------------------------------------------------------------------------
-------
mp4                                      272.791  757.822 7.49e+05
1.78e+03 100.0
  cacheline_adapter (cacheline_adaptor)
                                          20.161   45.124 3.70e+04
102.238   5.7
  arbiter (arbiter)                        5.671    4.263 9.83e+03
19.767   1.1
  dcache (cache_0)                        64.613  186.456 2.43e+05
493.621  27.7
```

```
    bus (line_adapter_0)                  0.242    0.175 6.12e+03
6.533   0.4
    datapath (cache_datapath_0)          64.307  185.684 2.36e+05
486.095  27.3
      dirty (array_width1_0)              1.341    6.092 1.60e+03
9.036   0.5
      valid (array_width1_1)              0.121    0.696 1.53e+03
2.345   0.1
      tag (array_width24_0)               0.368    5.493 1.92e+04
25.065   1.4
      DM_cache (data_array_0)            61.554  172.762 1.98e+05
432.111  24.3
    control (cache_control_0)          6.33e-02    0.598  332.251
0.993   0.1
  icache (cache_1)                      76.643  202.565 2.48e+05
526.882  29.6
    bus (line_adapter_1)                  1.285    0.641 4.24e+03
6.161   0.3
    datapath (cache_datapath_1)          75.290  201.421 2.43e+05
520.000  29.2
      valid (array_width1_3)              0.139    0.822 1.59e+03
2.553   0.1
      tag (array_width24_1)               0.979    6.006 1.94e+04
26.418   1.5
      DM_cache (data_array_1)            72.738  193.807 2.14e+05
480.348  27.0
    control (cache_control_1)          6.90e-02    0.503  149.170
0.721   0.0
  cpu (cpu)                            105.293  319.292 2.10e+05
634.864  35.7
    datapath (datapath)                105.293  319.292 2.10e+05
634.864  35.7
      rd_data_decoder (rd_data_decoder)
                                          0.424    0.173 2.78e+03
3.378   0.2
      forwarding_unit (forwarding)        0.234    0.318 1.19e+03
1.746   0.1
      CMP (cmp)                           0.347    0.473 4.42e+03
5.234   0.3
      ALU (alu)                           0.891    0.938 2.26e+04
24.441   1.4
      hazard_detection (hazard_detection)
                                          0.141    0.223  995.991
1.360   0.1
      control_rom (control_rom)           0.518    0.247 2.41e+03
3.180   0.2
      regfile (regfile)                   6.089   29.810 1.12e+05
147.573   8.3
      instr_decoder (instr_decoder)       0.000    0.000    0.000
0.000   0.0
      PC (pc_register)                    1.476    3.919 3.24e+03
8.632   0.5
      mem_wb_reg (mem_wb_reg)            14.230   41.075 7.27e+03
62.572   3.5
```

```
      ex_mem_reg (ex_mem_reg)             27.057   80.468 1.18e+04
119.314   6.7
      id_ex_reg (id_ex_reg)               38.577  118.398 1.68e+04
173.765   9.8
      if_id_reg (if_id_reg)               14.090   42.505 6.39e+03
62.983   3.5
1
```

```
****************************************
Report : timing
        -path full
        -delay max
        -max_paths 1
Design : mp4
Version: R-2020.09-SP4
Date   : Mon Mar 27 19:36:14 2023
****************************************

Operating Conditions: typical   Library: NangateOpenCellLibrary
Wire Load Model Mode: top

  Startpoint: cpu/datapath/mem_wb_reg/data_reg[rd][0]
              (rising edge-triggered flip-flop clocked by my_clk)
  Endpoint: cpu/datapath/ex_mem_reg/data_reg[ex_data_out][31]
            (rising edge-triggered flip-flop clocked by my_clk)
  Path Group: my_clk
  Path Type: max

  Des/Clust/Port     Wire Load Model       Library
  ------------------------------------------------
  mp4                5K_hvratio_1_1        NangateOpenCellLibrary

  Point                                                         Incr       Path
  ------------------------------------------------------------------------------
---
  clock my_clk (rise edge)                                      0.00       0.00
  clock network delay (ideal)                                   0.00       0.00
  cpu/datapath/mem_wb_reg/data_reg[rd][0]/CK (DFF_X1)           0.00       0.00
r
  cpu/datapath/mem_wb_reg/data_reg[rd][0]/Q (DFF_X1)            0.10       0.10
f
  cpu/datapath/mem_wb_reg/out[rd][0] (mem_wb_reg)               0.00       0.10
f
  cpu/datapath/forwarding_unit/wb_rd[0] (forwarding)            0.00       0.10
f
  cpu/datapath/forwarding_unit/U20/ZN (INV_X1)                  0.06       0.15
r
  cpu/datapath/forwarding_unit/U21/ZN (OAI22_X1)                0.04       0.19
f
  cpu/datapath/forwarding_unit/U22/ZN (AOI221_X1)               0.09       0.28
r
  cpu/datapath/forwarding_unit/U24/ZN (NAND4_X1)                0.05       0.33
f
  cpu/datapath/forwarding_unit/U25/ZN (NOR3_X1)                 0.07       0.40
r
  cpu/datapath/forwarding_unit/forward2[1] (forwarding)
                                                                0.00       0.40
r
  cpu/datapath/U36/ZN (INV_X1)                                  0.03       0.42
f
```

```
  cpu/datapath/U40/ZN (NOR2_X1)                    0.16       0.58
r
  cpu/datapath/U41/Z (CLKBUF_X1)                   0.19       0.77
r
  cpu/datapath/U42/ZN (NOR2_X1)                    0.11       0.88
f
  cpu/datapath/U43/Z (CLKBUF_X1)                   0.17       1.05
f
  cpu/datapath/U224/ZN (AOI22_X1)                  0.10       1.15
r
  cpu/datapath/U225/ZN (OAI21_X1)                  0.04       1.19
f
  cpu/datapath/U227/ZN (INV_X1)                    0.04       1.23
r
  cpu/datapath/U229/ZN (AOI22_X1)                  0.03       1.26
f
  cpu/datapath/U230/Z (CLKBUF_X1)                  0.13       1.39
f
  cpu/datapath/ALU/b[2] (alu)                      0.00       1.39
f
  cpu/datapath/ALU/U37/ZN (INV_X1)                 0.15       1.54
r
  cpu/datapath/ALU/U25/ZN (INV_X1)                 0.10       1.64
f
  cpu/datapath/ALU/U113/Z (XOR2_X1)                0.11       1.75
f
  cpu/datapath/ALU/U186/CO (FA_X1)                 0.11       1.86
f
  cpu/datapath/ALU/U255/CO (FA_X1)                 0.09       1.95
f
  cpu/datapath/ALU/U360/CO (FA_X1)                 0.09       2.04
f
  cpu/datapath/ALU/U382/CO (FA_X1)                 0.09       2.13
f
  cpu/datapath/ALU/U400/CO (FA_X1)                 0.09       2.22
f
  cpu/datapath/ALU/U673/CO (FA_X1)                 0.09       2.31
f
  cpu/datapath/ALU/U415/CO (FA_X1)                 0.09       2.40
f
  cpu/datapath/ALU/U429/CO (FA_X1)                 0.09       2.49
f
  cpu/datapath/ALU/U450/CO (FA_X1)                 0.09       2.58
f
  cpu/datapath/ALU/U469/CO (FA_X1)                 0.09       2.67
f
  cpu/datapath/ALU/U483/CO (FA_X1)                 0.09       2.76
f
  cpu/datapath/ALU/U500/CO (FA_X1)                 0.09       2.86
f
  cpu/datapath/ALU/U519/CO (FA_X1)                 0.09       2.95
f
  cpu/datapath/ALU/U557/CO (FA_X1)                 0.09       3.04
f
```

```
  cpu/datapath/ALU/U687/CO (FA_X1)                          0.09       3.13
f
  cpu/datapath/ALU/U700/CO (FA_X1)                          0.09       3.22
f
  cpu/datapath/ALU/U713/CO (FA_X1)                          0.09       3.31
f
  cpu/datapath/ALU/U725/CO (FA_X1)                          0.09       3.40
f
  cpu/datapath/ALU/U736/CO (FA_X1)                          0.09       3.49
f
  cpu/datapath/ALU/U746/CO (FA_X1)                          0.09       3.58
f
  cpu/datapath/ALU/U763/CO (FA_X1)                          0.09       3.67
f
  cpu/datapath/ALU/U775/CO (FA_X1)                          0.09       3.76
f
  cpu/datapath/ALU/U576/CO (FA_X1)                          0.09       3.85
f
  cpu/datapath/ALU/U600/CO (FA_X1)                          0.09       3.94
f
  cpu/datapath/ALU/U786/CO (FA_X1)                          0.09       4.03
f
  cpu/datapath/ALU/U798/CO (FA_X1)                          0.09       4.12
f
  cpu/datapath/ALU/U617/CO (FA_X1)                          0.09       4.22
f
  cpu/datapath/ALU/U639/CO (FA_X1)                          0.09       4.31
f
  cpu/datapath/ALU/U660/CO (FA_X1)                          0.09       4.40
f
  cpu/datapath/ALU/U663/Z (XOR2_X1)                         0.07       4.46
f
  cpu/datapath/ALU/U664/ZN (AND2_X1)                        0.04       4.50
f
  cpu/datapath/ALU/U665/ZN (OR2_X1)                         0.06       4.56
f
  cpu/datapath/ALU/f[31] (alu)                              0.00       4.56
f
  cpu/datapath/U652/ZN (AOI22_X1)                           0.06       4.62
r
  cpu/datapath/U653/ZN (OAI21_X1)                           0.03       4.65
f
  cpu/datapath/ex_mem_reg/in[ex_data_out][31] (ex_mem_reg)
                                                            0.00       4.65
f
  cpu/datapath/ex_mem_reg/U102/ZN (AND2_X1)                 0.05       4.70
f
  cpu/datapath/ex_mem_reg/data_reg[ex_data_out][31]/D (DFF_X1)
                                                            0.01       4.71
f
  data arrival time                                                    4.71

  clock my_clk (rise edge)                                 10.00      10.00
  clock network delay (ideal)                               0.00      10.00
```

```
  clock uncertainty                                       -0.10       9.90
  cpu/datapath/ex_mem_reg/data_reg[ex_data_out][31]/CK (DFF_X1)
                                                           0.00       9.90
r
  library setup time                                      -0.04       9.86
  data required time                                                  9.86
  ------------------------------------------------------------------------
---
  data required time                                                  9.86
  data arrival time                                                  -4.71
  ------------------------------------------------------------------------
---
  slack (MET)                                                         5.15


1
```

Cache Advanced Features

Before the advanced features for the cache can be implemented, our cache design from MP3 must be updated to respond in 1 cycle. In the MP3 implementation, the array register output values are only updated by the end of the clock cycle. This means, for example, that the 'hit' signal takes the whole cycle to generate, making a 1-hit design impossible. In order to adjust this, the arrays' dataout value is generated in a combinational block. The extra state in the state machine can be removed.

L2 Cache

Currently, the cache module contains the cache implementation, datapath and controller, plus the line adapter. In order to support a 2 level cache, we will first create a cache_unit module which will just contain just the cache datapath and controller. It will have the following interface:

```
module cache_unit (
    input clk,
    input rst,

    /* Line adapter/CPU memory signals */
    input logic             mem_read,
    input logic             mem_write,
    input logic [31:0]      mem_byte_enable,
    input logic [31:0]      mem_address,
    input logic [255:0]     mem_wdata,
    output logic            mem_resp,
    output logic [255:0]    mem_rdata,

    /* Physical memory signals */
    output  logic [31:0]    pmem_address,
    input   logic [255:0]   pmem_rdata,
    output  logic [255:0]   pmem_wdata,
    output  logic           pmem_read,
    output  logic           pmem_write,
    input   logic           pmem_resp
);
```

Then, a new top level cache_l2.sv module will instantiate two cache_unit modules as well as the line adapter. The line adapter will connect to the L1 cache and the CPU as in the current implementation, and the L2 cache will connect to the L1 cache and physical memory. The L2 byte enable input will be hard-coded to '1.

Parameterized Cache

We will parameterize the number of sets and the number of ways in our implementation.

Sets

In order to parameterize the number of sets, the number of bits used for indexing, and therefore the number of bits in the tag, must be adjustable. To do this, the main cache module has a parameter s_index, which represents the number of bits used to represent the index for a given address. There are therefore 2^s_index sets. This is supported by parameterizing the cache_datapath, the data_array, and array modules. The mem_address must be indexed correctly in the datapath to send the different arrays the correct read and write indexes, and the array modules need to instantiate internal data of the correct size.

Ways

In order to parameterize the number of ways, we will create a module pseudo_lru. This module will have the following interface.

```
module pseudo_lru #(
      parameter way_bits = 1,
      parameter s_index = 3,
      parameter num_ways = 2**way_bits,
      parameter num_sets = 2**s_index
)
(
      input clk,
      input rst,
      input read,
      input load,
      input [s_index-1:0] rindex,
      input [s_index-1:0] windex,
      input [way_bits-1:0] datain,
      output [way_bits-1:0] dataout
);
```

We will keep track of the LRU state with the following structure:

```
logic [num_ways-2:0] data [num_sets-1:0]
```

Then, the logic for operating on the LRU state is as follows.

Getting LRU:

```
dataout = 0;
int lru_idx = 0;
```

```
for (int i = way_bits - 1; i >=0; i--)
    dataout[i] = ~data[rindex][lru_idx];
    lru_idx = lru_idx +  lru_idx + 1 + ~data[rindex][lru_idx];
```

Setting most recently used:

```
if (load)
    int lru_idx = 0;
    for (int i = way_bits - 1; i >=0; i--)
        data[windex][lru_idx] = datain[i];
        lru_idx = lru_idx + lru_idx + 1 + datain[i];
```

In addition to the new least-recently-used logic, it is necessary to instantiate the correct number of data, dirty, valid, and tag arrays and adjust the controller logic to set load signals for these arrays appropriately. To do this, the arrays can be instantiated in a loop in a generate block.

Controller Description:

Following is a list of all the control signals and their default outputs.

load_tag = 1’b0

load_valid = 1’b0

load_dirty = 1’b0

load_data = 1’b0

load_lru = 1’b0

load_data_idx = 0

load_dirty_idx = 0

write_en_mux_sel = 1’b0

mem_resp = 1’b0

addr_mux_sel = 1’b0

pmem_read = 1’b0

pmem_write = 1’b0

data_out_mux_sel = 1’b0

data_in_mux_sel = 1’b0

State Name	Description	Transition Conditions	Outputs
check_cache	Check for cache hit. Update LRU, and if necessary, update data, dirty, and tag arrays. This state is also an idle state.	if ((mem_read ^ mem_write) && !hit && dirty) -> write_mem; else if ((mem_read ^ mem_write) && !hit && !dirty) -> read_mem; else -> check_cache	mem_resp = 1’b1 load_lru = 1’b1 data_out_mux_sel = hit_idx if (mem_write) load_dirty = 1’b1 load_dirty_idx = hit_idx load_data = 1’b1 load_data_idx = hit_idx
write_memory	Write dirty value from cache into main memory.	if (pmem_resp) -> read_memory else -> write_memory	pmem_write = 1’b1 data_out_mux_sel = lru_idx load_dirty = 1’b1 load_dirty_idx = lru_idx
read_memory	Read from main memory and write value to cache.	if (!pmem_resp) -> read_memory else -> update_cache	pmem_read = 1’b1; addr_mux_sel = 1’b1; load_tag = 1’b1; load_valid = 1’b1; load_data = 1’b1; load_data_idx = lru_idx data_in_mux_sel = data_in_mux::rdata; write_en_mux_sel = 1’b1;

4-Way/8-Way Set Associative Cache

Since the number of ways is already parameterized, creating a 4 or 8-way set associative cache just requires setting the parameter accordingly.

Victim Cache

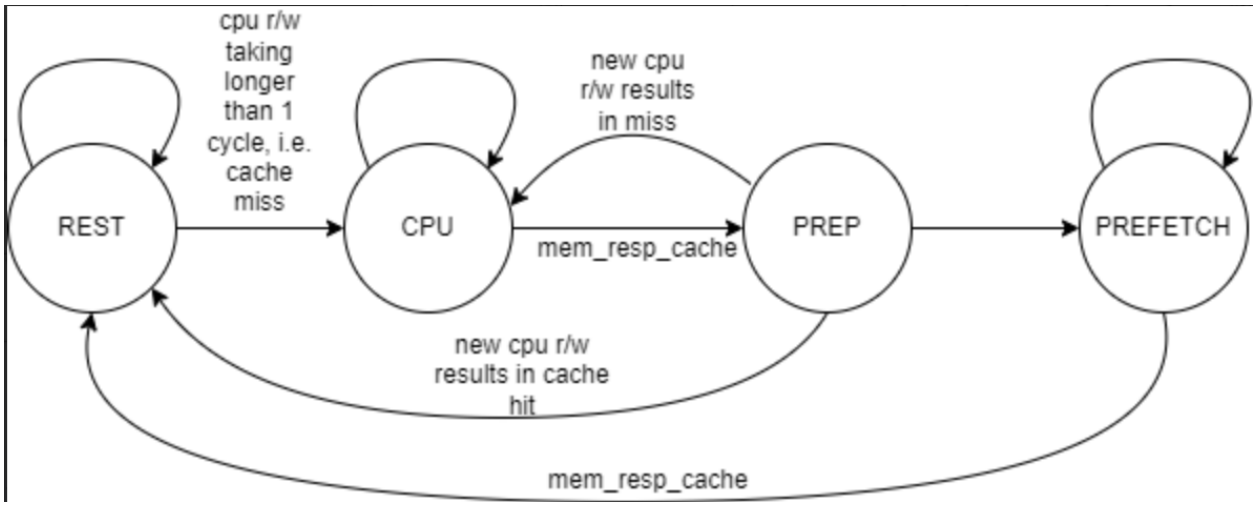
The victim cache (VC) will sit in between the L1 and L2 cache. It will be a fully associative cache with 4 ways (each with 1 set). The number of ways might be incremented to 8 or 16 depending on the performance/power results. The parameterized ways/sets of advanced features will be used to accomplish this.

Initially, the data in the L1 cache will be checked. If there is an L1 miss, then the VC will be checked. Both checks happen in the same clock cycle. If there is a VC hit, then the data is moved into L1 and returned to the CPU. The evicted value from the L1 cache (if applicable) is written into the victim cache. If there is a miss in L1 and VC, then the data is read from the L2 cache. The read data is placed in the L1 cache, and the L1 evicted data (if applicable) is written into the VC cache. If an eviction from the VC occurs, it will be written to the L2 cache.

Basic Hardware Prefetching:

The prefetching hardware will sit between the CPU and the L1 cache. CPU read/writes will route through the prefetcher, and when a memory access results in a cache miss, determined by monitoring the number of clock cycles it takes the cache to respond, and there are no other immediate memory accesses immediately initiated by the CPU, the prefetcher will initiate a prefetch on the next cache line. The controller description and a simplified state machine is shown below.

State Machine:



Controller Description:

Defaults:

mem_address_cache = mem_address_cpu;
mem_read_cache = mem_read_cpu;
mem_write_cache = mem_write_cpu;
mem_wdata_cache = mem_wdata_cpu;
mem_byte_enable_cache = mem_byte_enable_cpu;
mem_resp_cpu = mem_resp_cache;
mem_rdata_cpu = mem_rdata_cache;

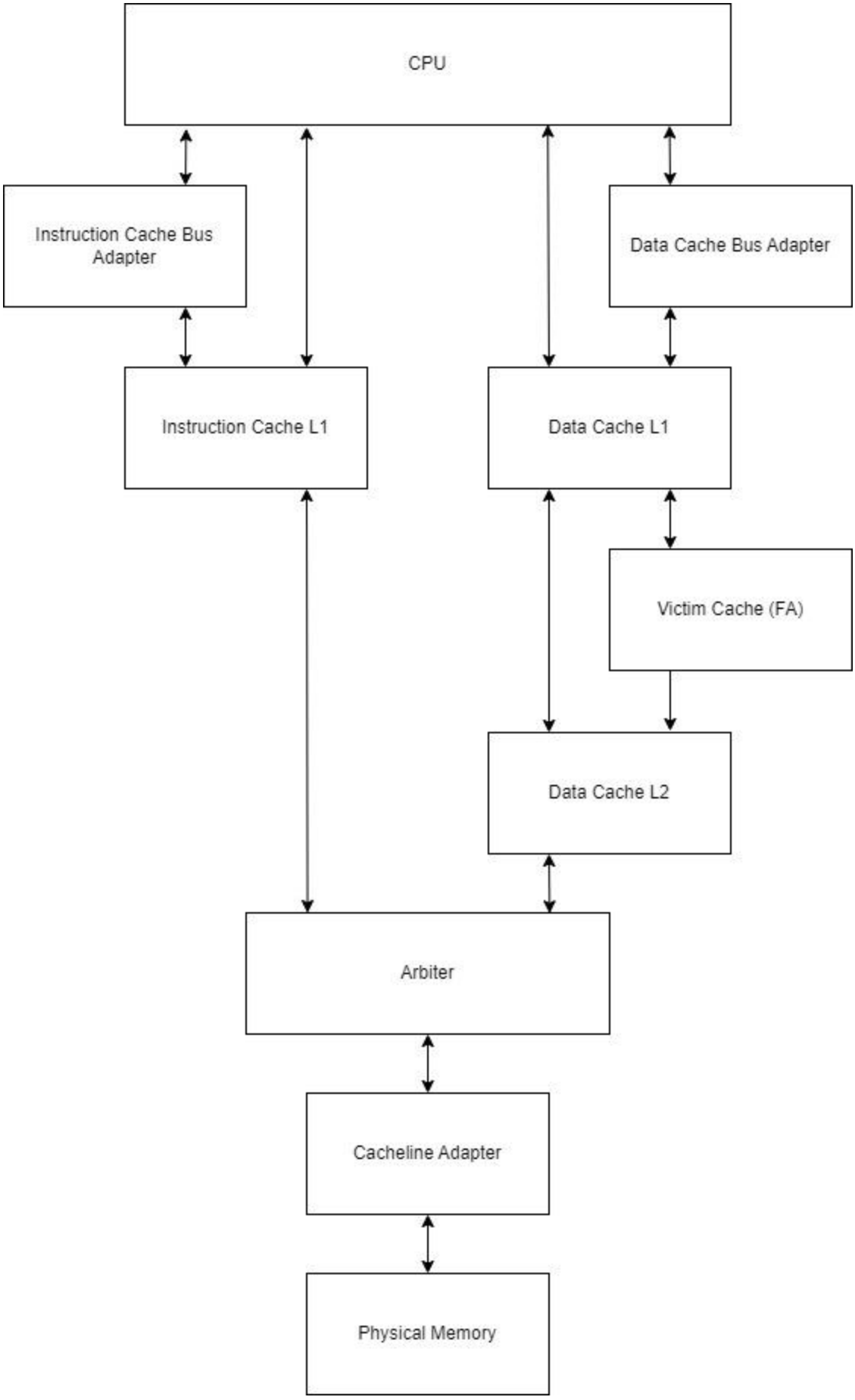
State Name	Description	Transition Conditions	Outputs

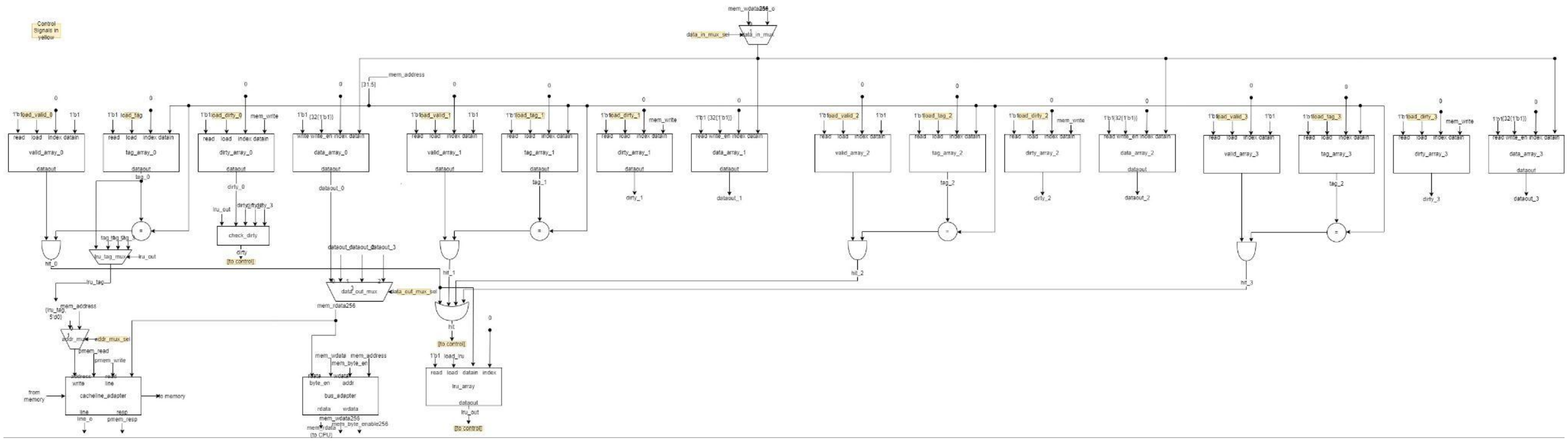
REST	Rest or do 1 cycle of CPU memory access.	if ((mem_read_cpu ^ mem_write_cpu) && !mem_resp_cache))
		=> next_state = CPU
		else
		=> next_state = REST	

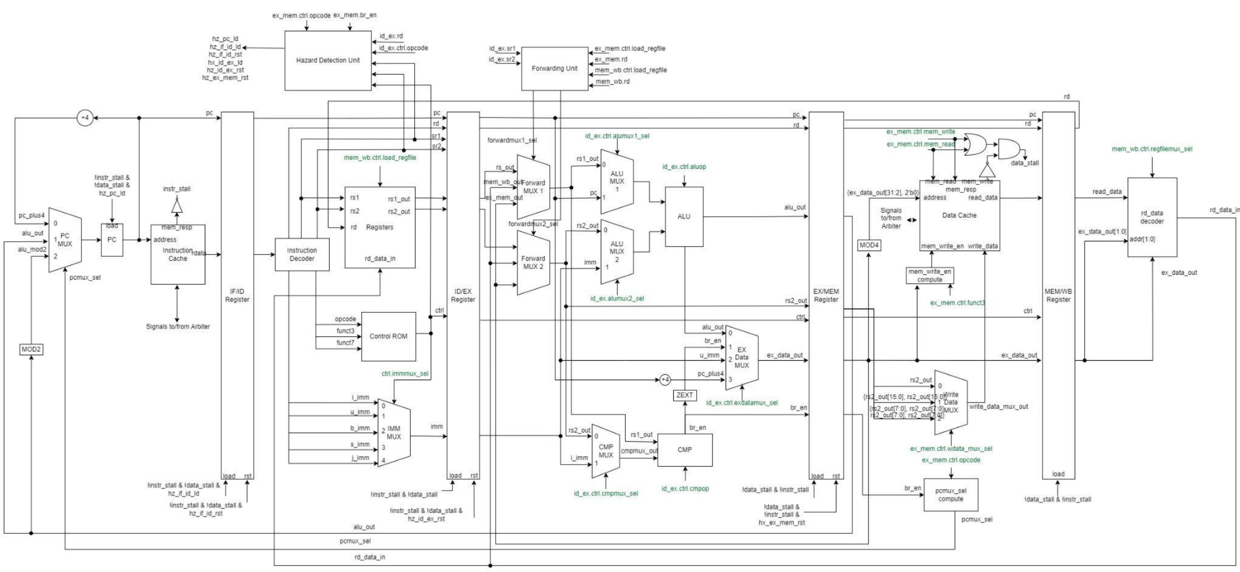
CPU	Complete memory access for CPU.	if (mem_resp_cache)
		=> next_state = PREP
		else
		=> next_state = CPU	

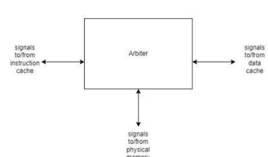
PREP	Optimization state. Do memory access for CPU if available or move on to prefetch.	if ((mem_read_cpu ^ mem_write_cpu) && !mem_resp_cache))
		=> next_state = CPU
		else if (mem_resp_cache))
		=> next_state = REST
		else
		=> next_state = PREFETCH
			if (!(mem_read_cpu ^ mem_write_cpu))
			mem_address_cache = last_address + 32'h00000020;
			mem_read_cache = 1'b1;
			mem_write_cache = 1'b0;
			mem_resp_cpu = 1'b0;

PREFETCH	Set memory control signals to initiate prefetch	if (mem_resp_cache)
		=> next_state = REST
		else
		=> next_state = CPU
			mem_address_cache = last_address + 32'h00000020;
			mem_read_cache = 1'b1;
			mem_write_cache = 1'b0;
			mem_resp_cpu = 1'b0;


CPU
Instruction Cache Bus Adapter
Data Cache Bus Adapter
Instruction Cache L1
Data Cache L1
Victim Cache (FA)
Data Cache L2
Arbiter
Cacheline Adapter
Physical Memory


Control Signals in yellow
valid_array_0
tag_array_0
dirty_array_0
data_array_0
valid_array_1
tag_array_1
dirty_array_1
data_array_1
valid_array_2
tag_array_2
dirty_array_2
data_array_2
valid_array_3
tag_array_3
dirty_array_3
data_array_3
mem_address
data_in_mux
check_dirty
lru_tag_mux
data_out_mux
addr_mux
cacheline_adapter
bus_adapter
lru_array
from memory
to memory


Hazard Detection Unit
Forwarding Unit
PC MUX
PC
Instruction Cache
IF/ID Register
Instruction Decoder
Registers
Control ROM
IMM MUX
ID/EX Register
Forward MUX 1
Forward MUX 2
ALU MUX 1
ALU MUX 2
ALU
EX Data MUX
CMP MUX
CMP
ZEXT
EX/MEM Register
Data Cache
mem_write_en compute
Write Data MUX
pcmux_sel compute
MEM/WB Register
rd_data decoder
MOD2
MOD4
Signals to/from Arbiter


Arbiter
signals to/from instruction cache
signals to/from data cache
signals to/from physical memory

Branch Advanced Features:

Local Branch History Table:

The local branch history table consists of a 2-bit predictor indexed by PC values. The table contains 512 entries, with each entry containing a 2-bit predictor:

00: Strongly not taken; 01: Weakly not taken; 10: Weakly taken; 11: Strongly taken

Since we have 512 entries and all the PC addresses are 4-byte aligned, we use PC[10:2] as the table's index. Whenever there's a branch instruction, we use the PC value in the fetch stage indexing into the table to predict whether the branch is taken.

An FSM is used to update the predictor once there's a mismatch between the predicted direction and the actual direction. The actual direction and actual target address are available at MEM stage and the actual direction is the input to the FSM.

A Branch target buffer stores the target address for each branch instruction. The BTB acts like a fully-associative cache. Since the predictor table has 512 entries, the BTB also has 512 entries. In each BTB entry, we have a valid bit, a tag, and a target address. The tag is the branch PC address[31:2] because they are 4-byte aligned.

For a branch instruction, the upper 30 bits pass to the BTB and compare with each tag value to see if there's a hit. If there's a hit and the local branch history table says taken, the target address is passed to the PCmux as our next PC. If there's a hit but the local branch history table says not taken, the next PC would be PC+4. If there's a miss in BTB, the next PC would be PC+4. When the BTB encounters a miss and it's a branch instruction, we want to update the BTB with the PC address and the calculated target address. If there's an entry with a valid bit 0, fill the entry with the new data. If the buffer is fully occupied, use pseudo LRU to replace an entry with the new data.

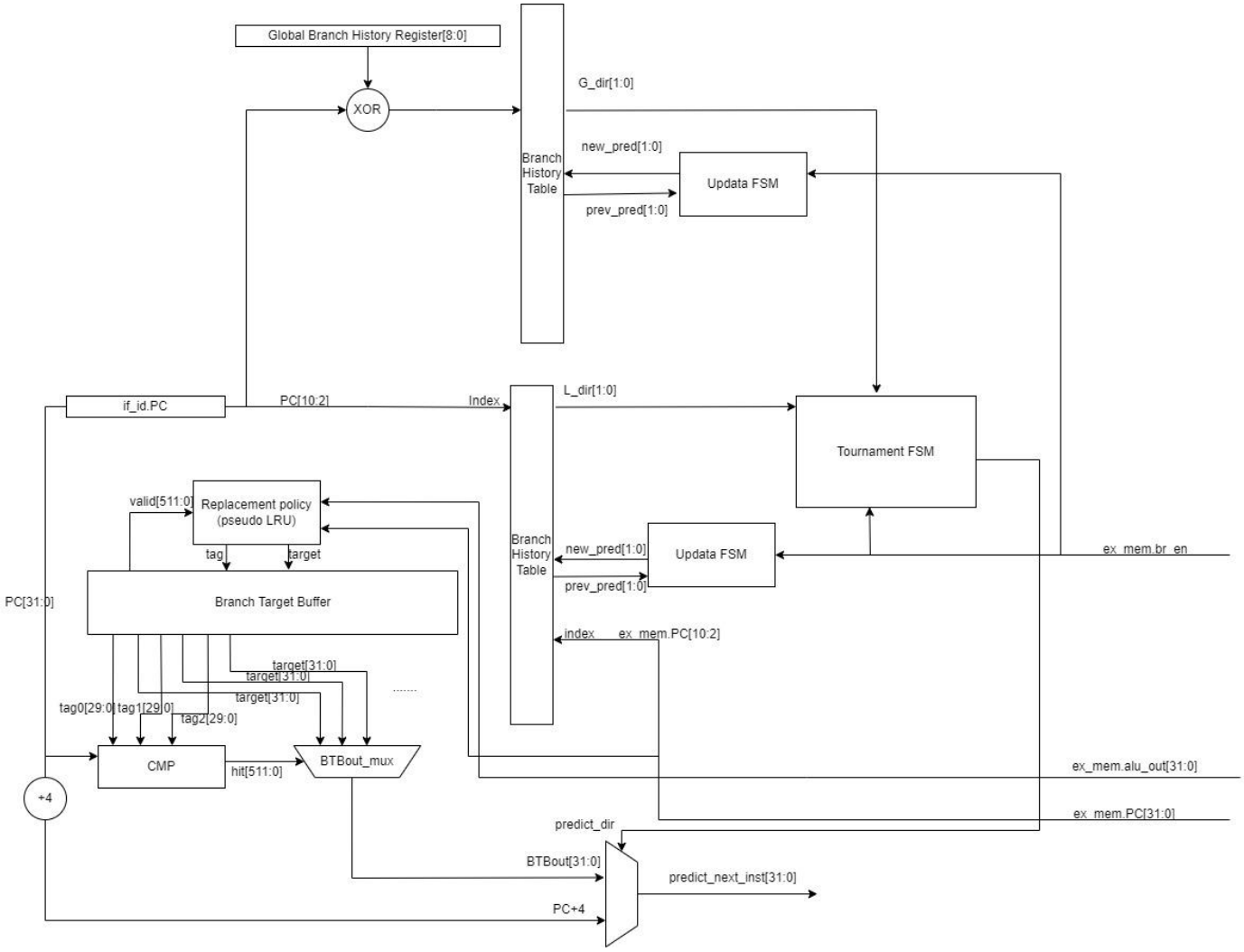
Global 2-level Branch History Table:

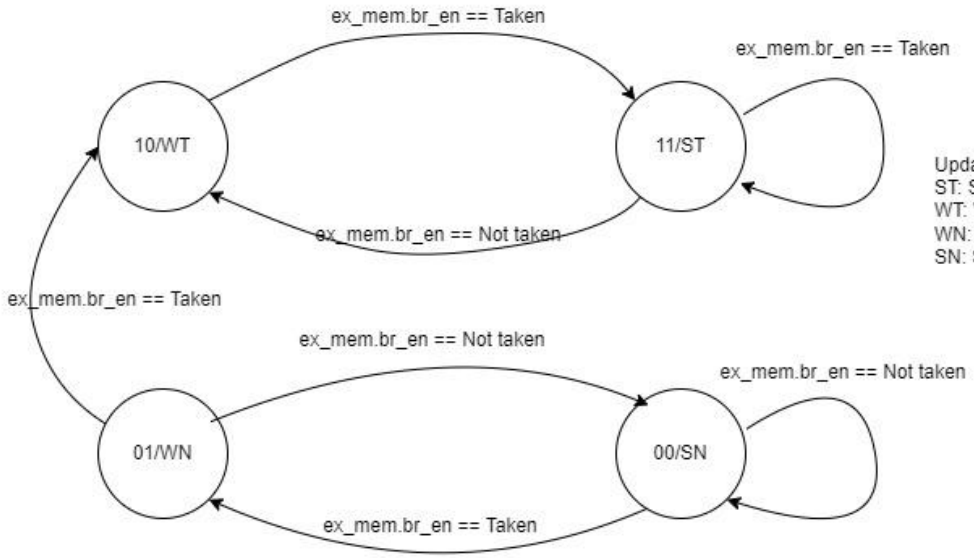
The global 2-level branch history table is a shift register that records the direction of the previous branch. It's a 9-bit register, so it records 9 previous branch directions. It also has a table of the 2-bit predictor. The value in the shift register XOR with the PC [10:2] to form the index into the table, and the rest is the same as the local branch history table.

Tournament Branch Predictor:

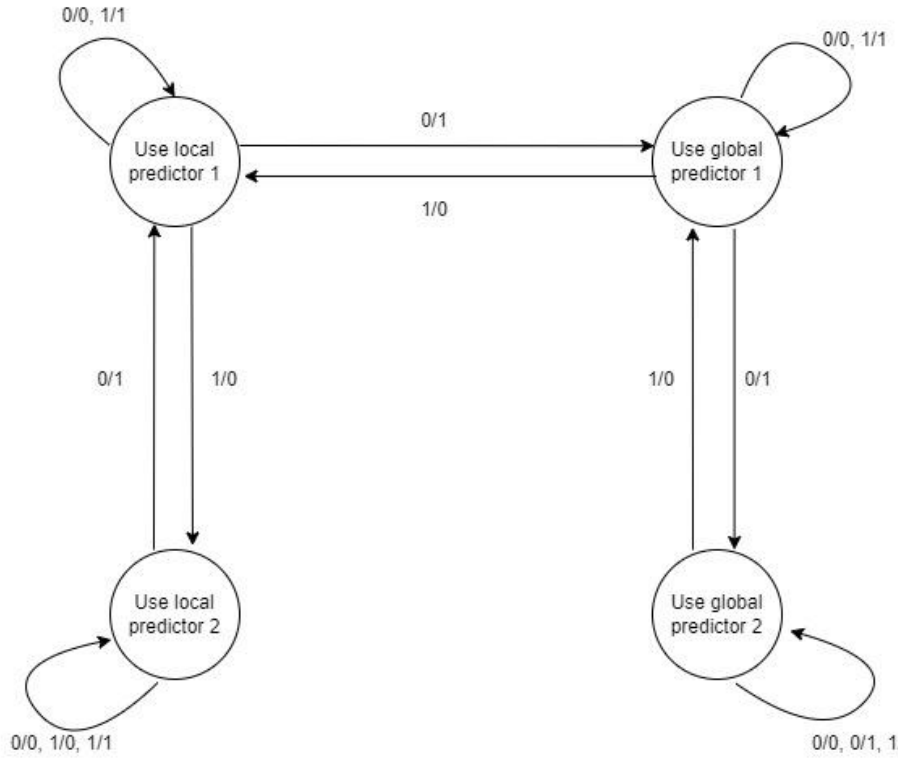
The tournament branch predictor selects between the local branch direction output and the global branch direction output. A 2-bit FSM is used to select between the two directions. Based on the accuracy of previous local and global predictions, the FSM will update its selection bias.

00: Use_local_predictor_1; 01: Use_local_predictor_2; 10: Use_global_predictor_1; 11: Use_global_predictor_2





Update FSM:
ST: Strongly Taken
WT: Wealy Taken
WN: Weakly Not Taken
SN: Strongly Not Taken



Tournament Predictor:
local predictor/global predictor
1: correct, 0: not correct

Progress Report (3/29-4/12)

Over the last two weeks, the main addition to our code was to integrate our MP3 cache into our design and add support for coremark. As of now, CMP 1-3 and Coremark all successfully run. The fmax and total power for our design as of now is given below. The current datapath for our design is given at the end of this document.

slack = 5.14

fmax = 1/(10-slack) * 1000 = 205.76 MHz

Clock cycle = 1/(205.76 MHZ) * 1000 = 4.86

total power = 4.65e+03 uW

In addition, we worked on the designs for the advanced features. The division of work matches the implementation details described below. Mitchell also researched an improved control hazard detection implementation but came to the conclusion that the power increase of doing so will be substantial. As a result, this feature was put on hold.

Roadmap (4/12-4/26)

As our next step, we plan to start building and verifying the advanced features. The division of work is as follows:

Neha: L2+ cache system, 4-way set associative cache, parameterized cache

Mitchell: Victim cache

Yu: Local branch history table, global 2-level branch history table, tournament branch predictor software branch predictor model

Each member is responsible for the implementation and verification of their respective part. For verification, we intend to create a DUT for each of the added components in order to separate testing. Additionally, we will add performance counters where necessary.

Progress Report (4/12-5/1)

This week, we worked on completing our advanced features. The advanced features that were implemented, as well as the division of labor is as follows:

- Parameterized Cache (sets and ways): Neha (6)
- L2 Cache: Neha (implementation+testing), Mitchell (performance analysis) (2)
- OBL Prefetching: Neha (4)
- Victim Cache: Mitchell (6)
- Local Branch History: Yu (2)
- Global Branch History: Yu (3)
- Tournament Branch Predictor: Yu (1)
- BTB Jump support: Yu (1)
- 4-way set associative or higher BTB: Yu (3)
- Software Branch Prediction: Yu (2)

Since some of these features are mutually exclusive, there is no one processor performance number, but the performance of each of these features independently is available as part of the performance analysis.

For testing, we created a comprehensive test bench to test each of the advanced features as a DUT. In addition, for some of the features, we added additional assembly tests to test the added feature alongside the rest of the processor. We also verified that all the provided test code runs correctly with the additional features.

Roadmap:

Our next steps are to use our analysis to determine which of the advanced features best improved the performance of our processor and integrate those for the final competition runs. In addition, we will need to prepare our presentation and lab report.

Final Summary

For this project, we implemented a 5-stage in-order, pipelined RISC-V CPU. Additionally, we implemented the following advanced features:

- Parameterized Cache (sets and ways)
- 4/8-way Cache
- L2 Cache
- Hardware Prefetching (OBL)
- Victim Cache
- Local Branch History
- Global Branch History
- Tournament Branch Predictor
- BTB Jump support
- Set associative BTB
- Software Branch Prediction

Of the advanced features we implemented, we used the following in our final design:

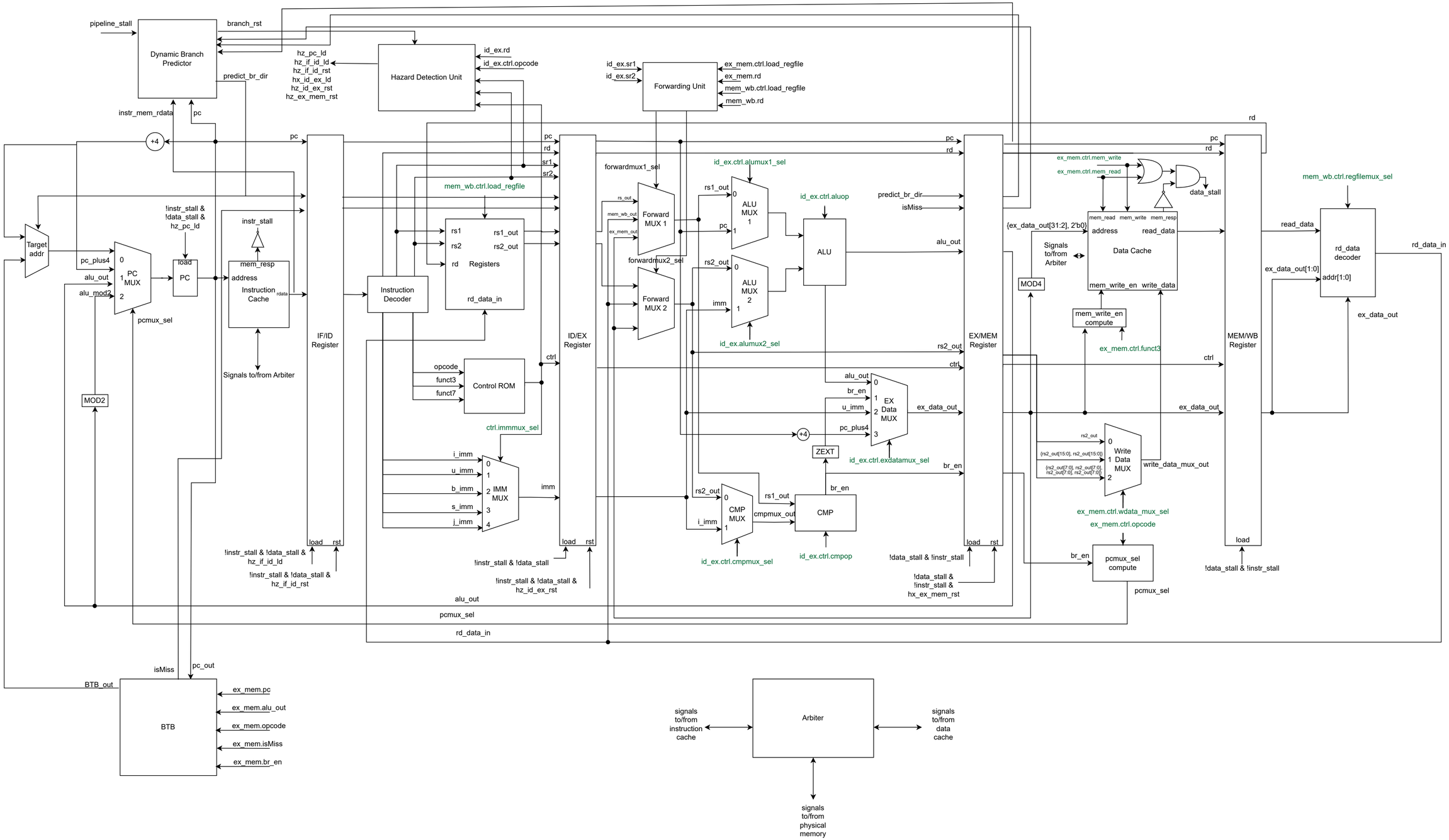
- Instruction Cache: 2-way SA L1 cache w/ 16 sets
- Data Cache: 2-way SA L1 cache w/ 4 sets
- Branch Prediction: Local branch predictor w/ 32 entries

The final results with these features are summarized below. Note that in the previous advanced feature results, power is scaled down by 10^3. However, the results below use the units as given in the power report. The datapath for the final design can be found at the end of this report.

Freq	Test Code	Power	Delay	Score	Total Score	DCache Miss Rate	ICache Miss Rate	Branch Miss Pred Rate
333	**Comp 1**	10500	107403000	1.09228E+19		0.131	0.002	0.148
	Comp 2	10900	228089000	5.11384E+19		0.067	0.0003	0.208
	Comp 3	9800	109821000	1.06588E+19	1.81244E+19	0.09	0.001	0.131
	Core mark	9290	7112465000	4.23806E+22	4.23806E+22	0.114	0.001	0.222

Our team worked well together. We started on all the checkpoints early, which gave us plenty of time to complete all of our features. The main thing that we would have done differently is that since the test codes are small, we would have not focused so much on cache advanced features. Instead, we might have looked into features such as the RISC-V multiplication

extension and the return address stack, which likely would have given us a better performance increase.


pipeline_stall
Dynamic Branch Predictor
branch_rst
predict_br_dir
hz_pc_ld
hz_if_id_ld
hz_if_id_rst
hx_id_ex_ld
hz_id_ex_rst
hz_ex_mem_rst
Hazard Detection Unit
id_ex.rd
id_ex.ctrl.opcode
id_ex.sr1
id_ex.sr2
Forwarding Unit
ex_mem.ctrl.load_regfile
ex_mem.rd
mem_wb.ctrl.load_regfile
mem_wb.rd
instr_mem_rdata
pc
+4
Target addr
pc_plus4
alu_out
alu_mod2
PC MUX
pcmux_sel
!instr_stall & !data_stall & hz_pc_ld
load
PC
instr_stall
mem_resp
address
Instruction Cache
rdata
Signals to/from Arbiter
MOD2
IF/ID Register
Instruction Decoder
rs1
rs2
rd
Registers
rs1_out
rs2_out
rd_data_in
opcode
funct3
funct7
Control ROM
ctrl
ctrl.immmux_sel
i_imm
u_imm
b_imm
s_imm
j_imm
IMM MUX
imm
!instr_stall & !data_stall & hz_if_id_ld
!instr_stall & !data_stall & hz_if_id_rst
!instr_stall & !data_stall
!instr_stall & !data_stall & hz_id_ex_rst
load
rst
ID/EX Register
forwardmux1_sel
forwardmux2_sel
Forward MUX 1
Forward MUX 2
rs_out
mem_wb_out
ex_mem_out
id_ex.ctrl.alumux1_sel
id_ex.ctrl.alumux2_sel
ALU MUX 1
ALU MUX 2
id_ex.ctrl.aluop
ALU
predict_br_dir
isMiss
br_en
CMP
CMP MUX
cmpmux_out
id_ex.ctrl.cmpmux_sel
id_ex.ctrl.cmpop
ZEXT
id_ex.ctrl.exdatamux_sel
EX Data MUX
ex_data_out
EX/MEM Register
!data_stall & !instr_stall
!data_stall & !instr_stall & hx_ex_mem_rst
MOD4
ex_mem.ctrl.mem_write
ex_mem.ctrl.mem_read
data_stall
{ex_data_out[31:2], 2'b0}
Signals to/from Arbiter
mem_read
mem_write
mem_resp
address
read_data
Data Cache
mem_write_en
write_data
mem_write_en compute
ex_mem.ctrl.funct3
Write Data MUX
write_data_mux_out
ex_mem.ctrl.wdata_mux_sel
ex_mem.ctrl.opcode
pcmux_sel compute
MEM/WB Register
mem_wb.ctrl.regfilemux_sel
read_data
rd_data decoder
rd_data_in
ex_data_out[1:0]
addr[1:0]
ex_data_out
!data_stall & !instr_stall
alu_out
pcmux_sel
rd_data_in
isMiss
pc_out
BTB_out
BTB
ex_mem.pc
ex_mem.alu_out
ex_mem.opcode
ex_mem.isMiss
ex_mem.br_en
signals to/from instruction cache
Arbiter
signals to/from data cache
signals to/from physical memory