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.

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.

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

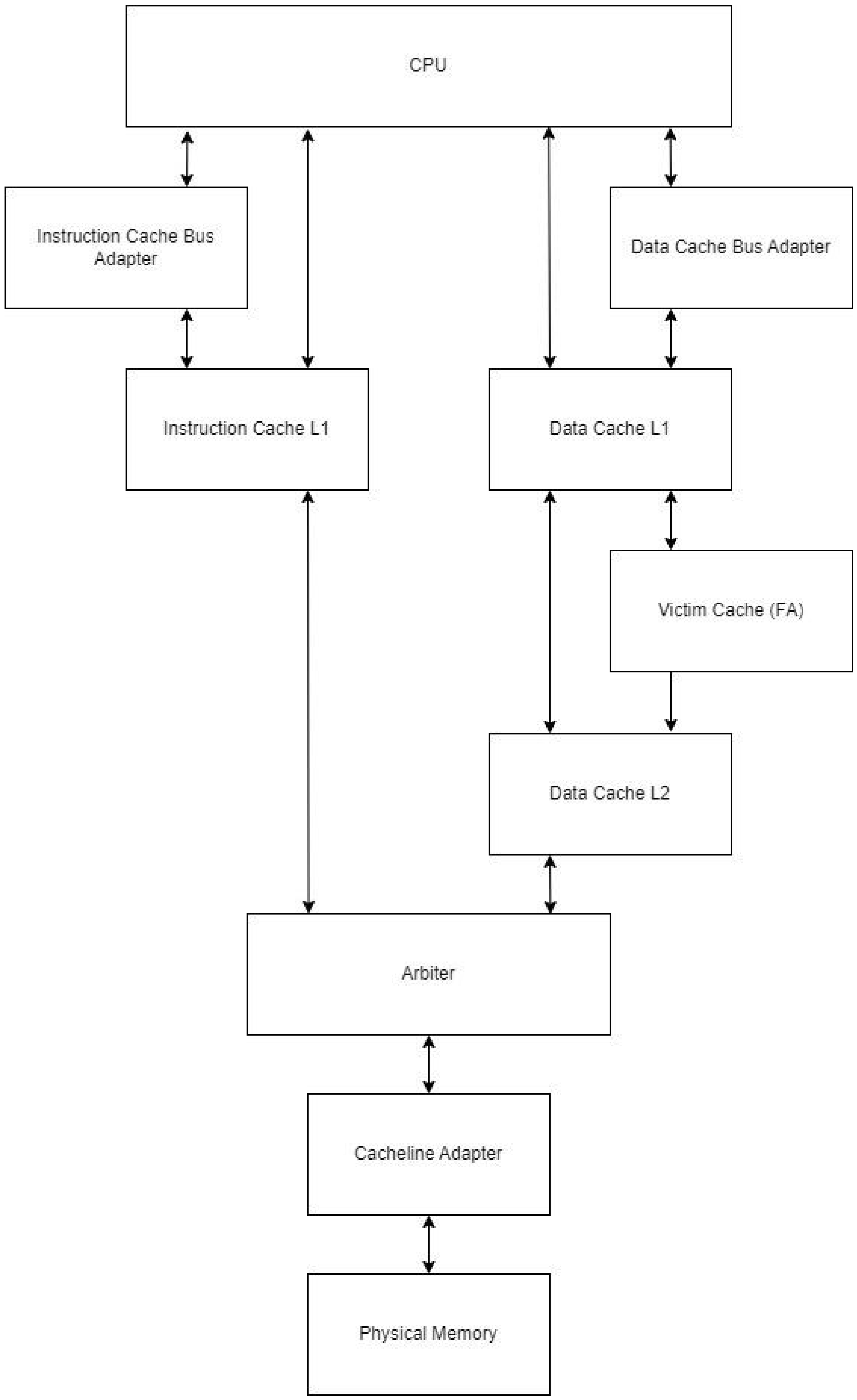
data_in_mux_sel = 1’b0

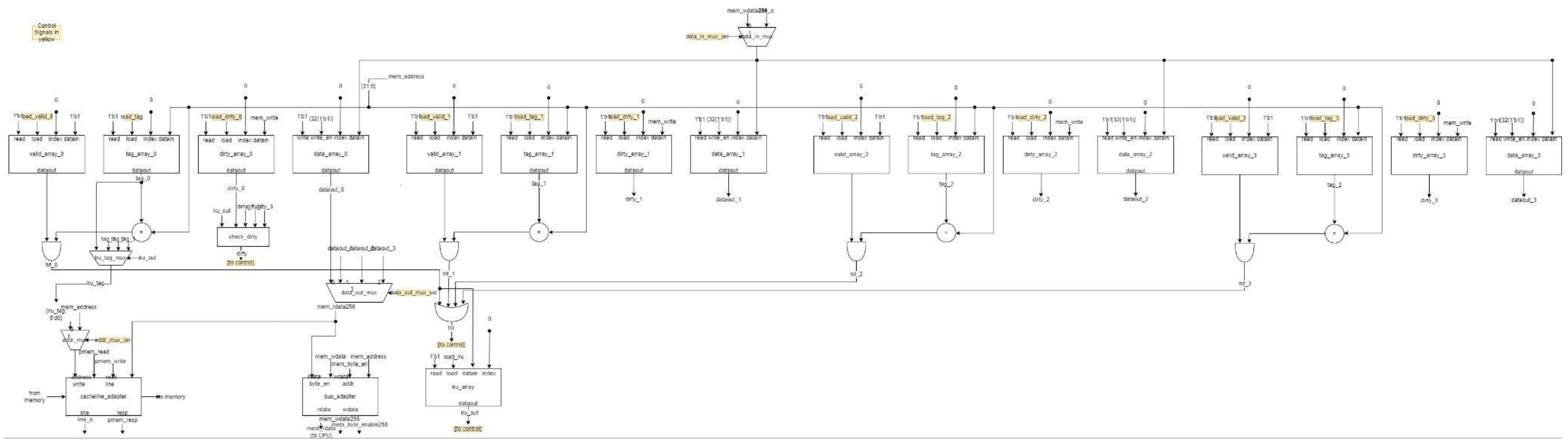
State Name	**Description**	**Transition Conditions**	**Outputs**

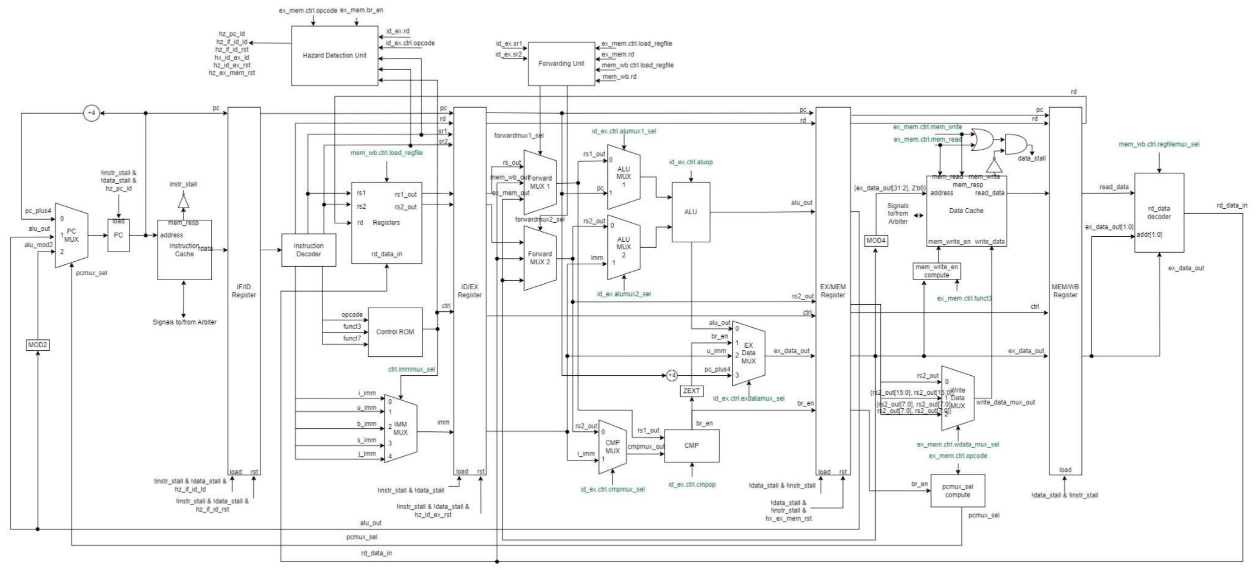
check_cache	Check for cache hit. Update LRU, and if necessary, update data, dirty, and tag arrays. This state is also an idle state.	if ((mem_read ^ mem_write) && !hit && dirty) -> write_mem;
		else if ((mem_read ^ mem_write) && !hit && !dirty) -> read_mem;
		else -> check_cache
			mem_resp = 1’b1
			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

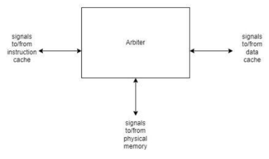
write_memory	Write dirty value from cache into main memory.	if (pmem_resp) -> read_memory
		else -> write_memory
			pmem_write = 1’b1
			data_out_mux_sel = lru_idx
			load_dirty = 1’b1
			load_dirty_idx = lru_idx

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;


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
mem_wdata256_o
data_in_mux_sel
data_in_mux
mem_address
[31:5]
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
load_valid_0
load_tag
load_dirty_0
load_valid_1
load_tag_1
load_dirty_1
load_valid_2
load_tag_2
load_dirty_2
load_valid_3
load_tag_3
load_dirty_3
read
load
index
datain
dataout
write_en
mem_write
1'b1
{32{1'b1}}
dirty_0
dirty_1
dirty_2
dirty_3
dataout_0
dataout_1
dataout_2
dataout_3
tag_0
tag_1
tag_2
lru_out
check_dirty
dirty
[to control]
hit_0
hit_1
hit_2
hit_3
hit
lru_tag_mux
lru_tag
data_out_mux
data_out_mux_sel
mem_rdata256
{lru_tag, 5'b0}
addr_mux
addr_mux_sel
pmem_read
pmem_write
address
write
read
line
cacheline_adapter
from memory
to memory
line_o
resp
pmem_resp
mem_wdata
mem_address
mem_byte_en
bus_adapter
byte_en
wdata
addr
rdata
mem_wdata256
mem_byte_enable256
mem_rdata
[to CPU]
load_lru
lru_array
lru_out
[to control]


ex_mem.ctrl.opcode
ex_mem.br_en
Hazard Detection Unit
id_ex.rd
id_ex.ctrl.opcode
hz_pc_ld
hz_if_id_ld
hz_if_id_rst
hz_id_ex_ld
hz_id_ex_rst
hz_ex_mem_rst
id_ex.sr1
id_ex.sr2
Forwarding Unit
ex_mem.ctrl.load_regfile
ex_mem.rd
mem_wb.ctrl.load_regfile
mem_wb.rd
+4
pc
rd
sr1
sr2
pc_plus4
alu_out
alu_mod2
PC MUX
pcmux_sel
MOD2
!instr_stall & !data_stall & hz_pc_ld
load
PC
instr_stall
mem_resp
address
Instruction Cache
rdata
Signals to/from Arbiter
IF/ID Register
Instruction Decoder
mem_wb.ctrl.load_regfile
rs1
rs2
rd
rs1_out
rs2_out
Registers
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
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
rs1_out
rs2_out
ALU MUX 1
ALU MUX 2
id_ex.ctrl.aluop
ALU
alu_out
br_en
u_imm
pc_plus4
EX Data MUX
id_ex.ctrl.exdatamux_sel
ZEXT
CMP MUX
cmpmux_out
CMP
id_ex.ctrl.cmpmux_sel
id_ex.ctrl.cmpop
ex_data_out
EX/MEM Register
!data_stall & !instr_stall
!data_stall & !instr_stall & hx_ex_mem_rst
ex_mem.ctrl.mem_write
ex_mem.ctrl.mem_read
data_stall
{ex_data_out[31:2], 2'b0}
Signals to/from Arbiter
MOD4
mem_read
mem_write
mem_resp
read_data
Data Cache
mem_write_en
write_data
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
pcmux_sel compute
MEM/WB Register
!data_stall & !instr_stall
mem_wb.ctrl.regfilemux_sel
read_data
ex_data_out[1:0]
rd_data decoder
addr[1:0]
ex_data_out
rd_data_in


signals to/from instruction cache
Arbiter
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.

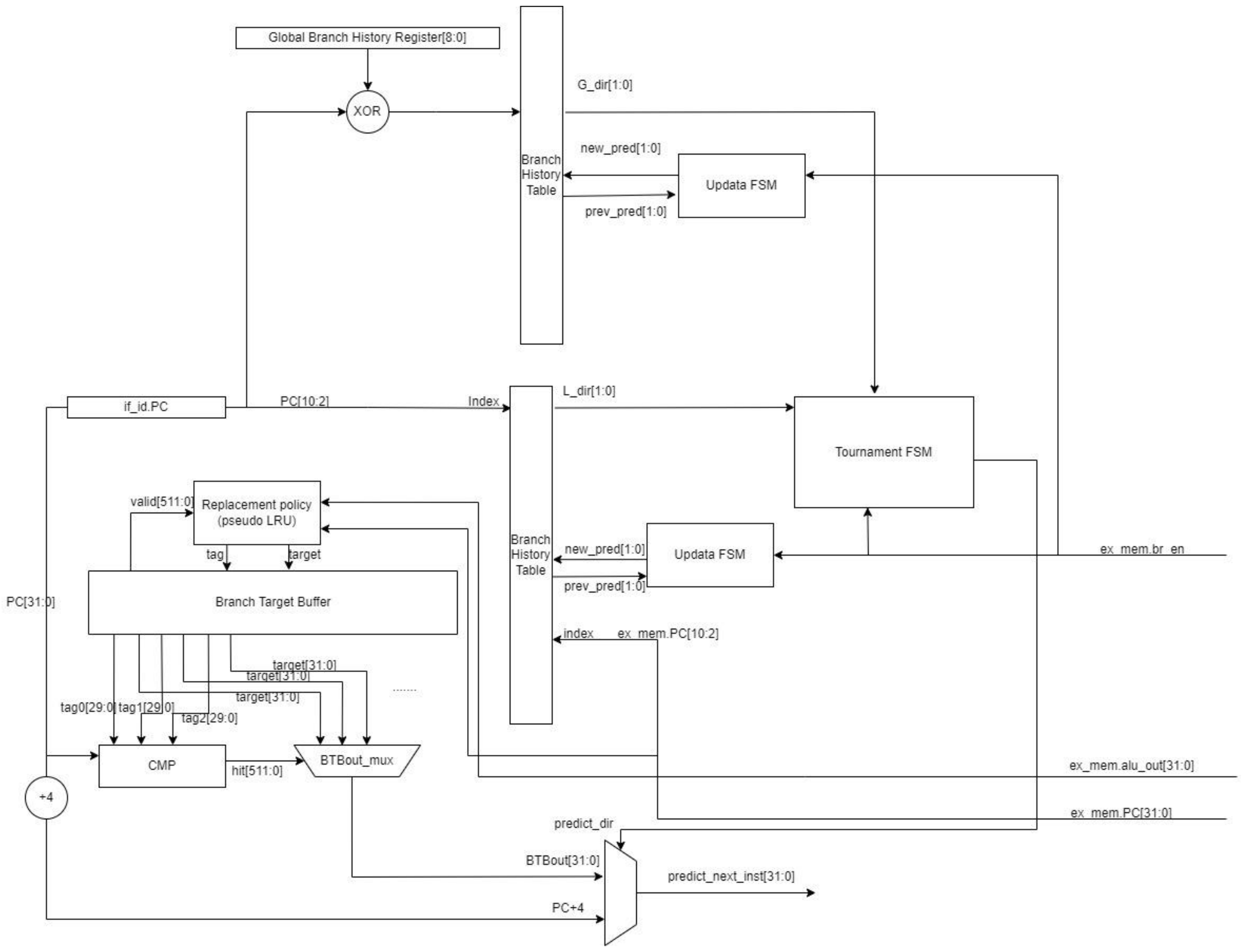
Global 2-level Branch History Table:

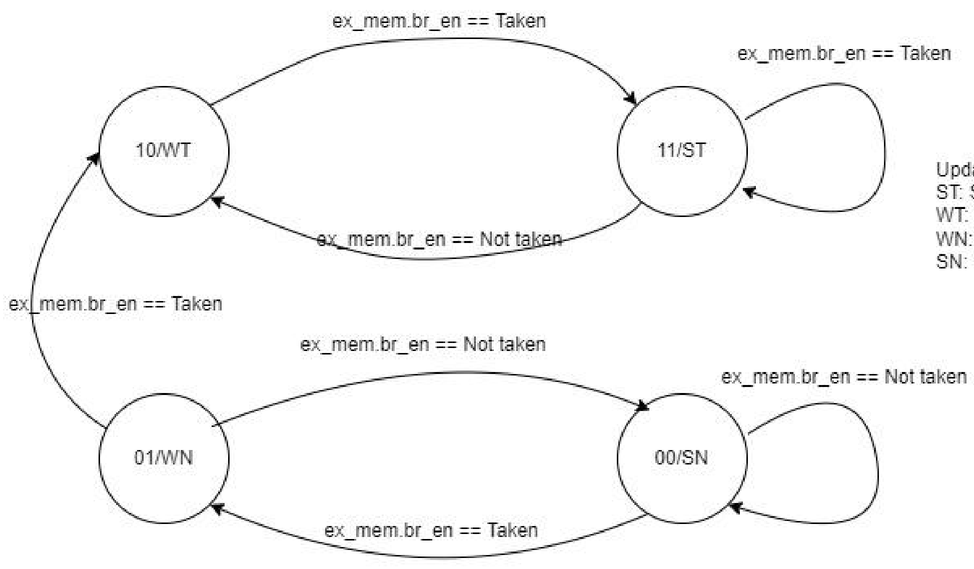
The global 2-level branch history table is a shift register that records the direction of the previous branch. It's a 9-bit register, so it records 9 previous branch directions. It also has a table of the 2-bit predictor. The value in the shift register XOR with the PC [10:2] to form the index into the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

Tournament Branch Predictor:

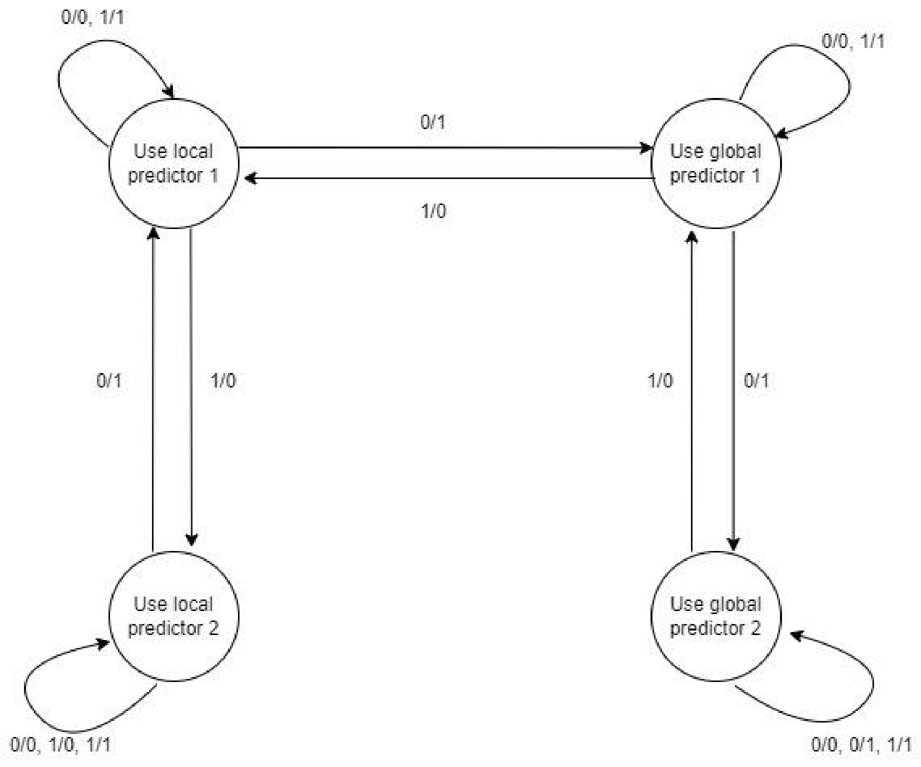
The tournament branch predictor selects between the local branch direction output and the global branch direction output. A 2-bit FSM is used to select between the two directions. Based on the accuracy of previous local and global predictions, the FSM will update its selection bias.

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