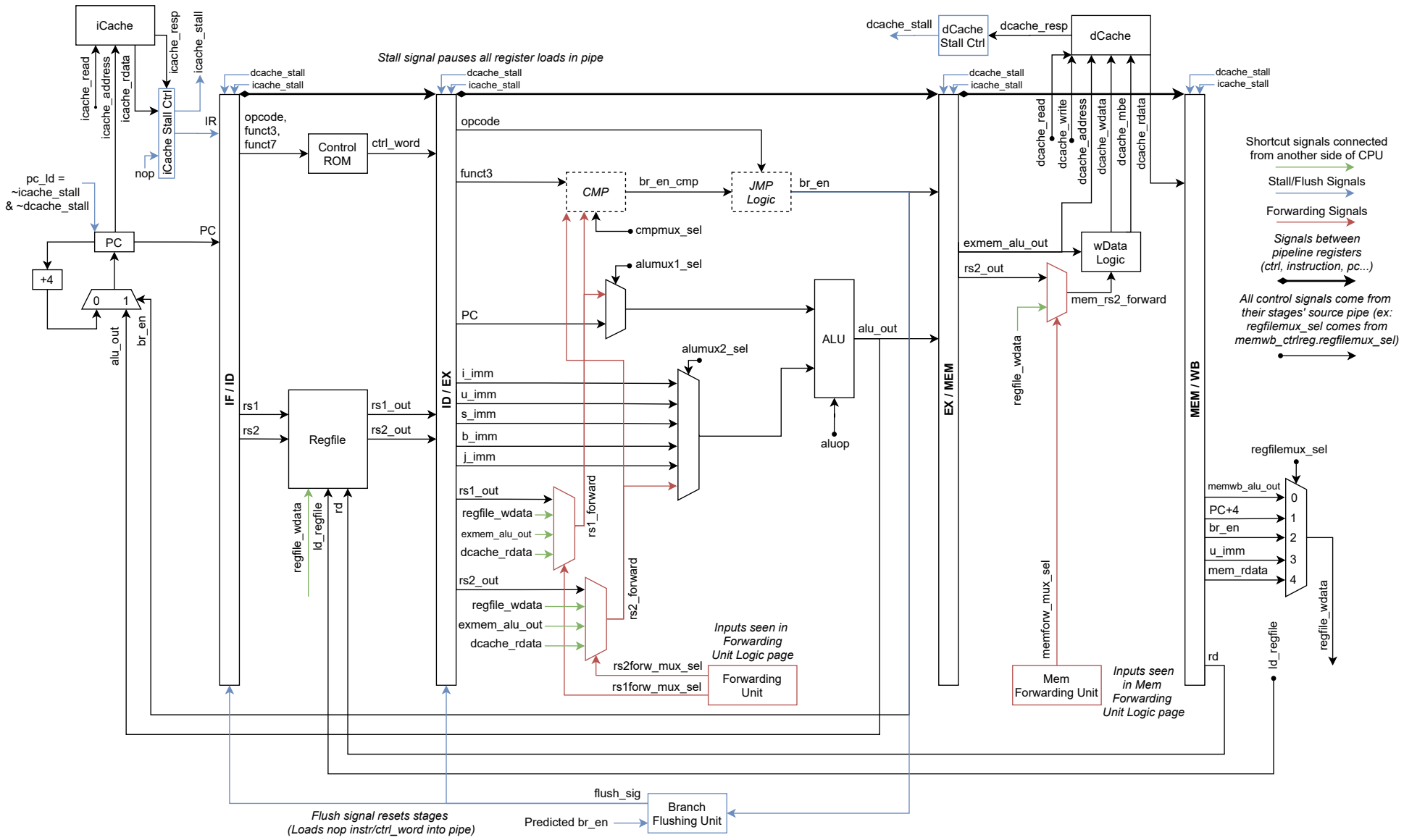
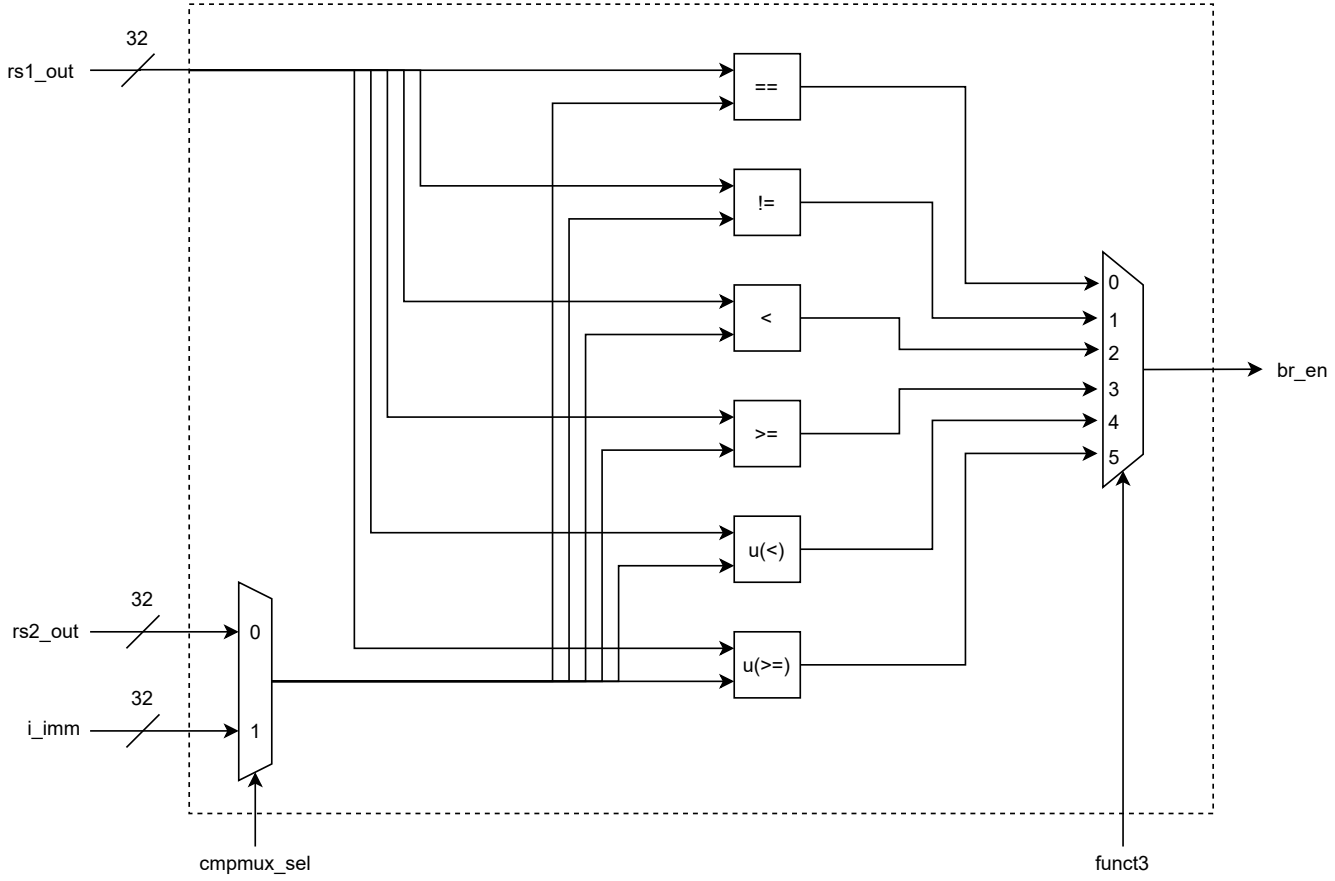
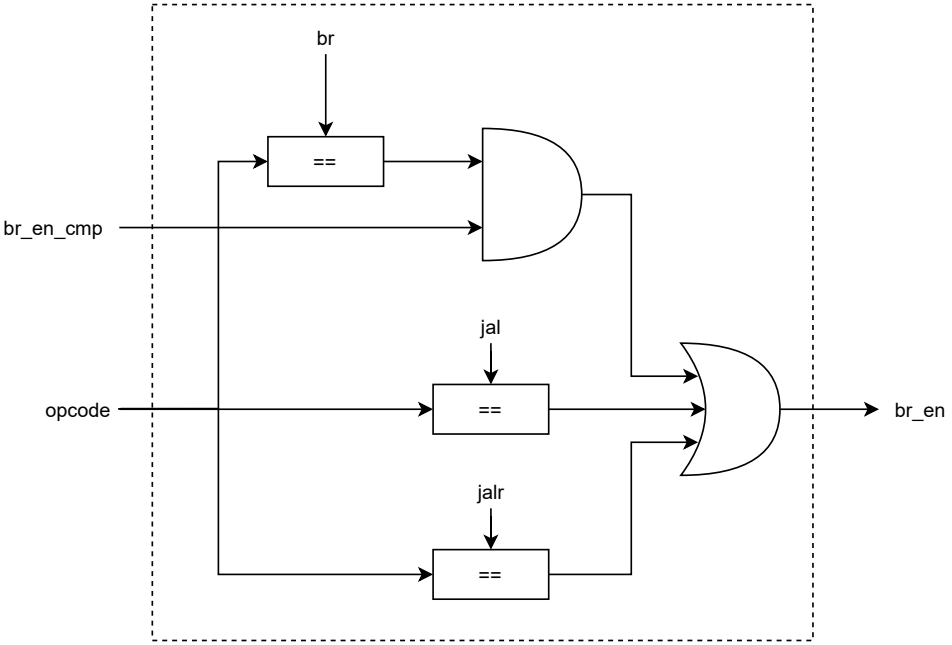

iCache
icache_read
icache_address
icache_rdata
icache_resp
icache_stall
iCache Stall Ctrl
nop
IR
pc_ld = ~icache_stall & ~dcache_stall
PC
+4
0
1
alu_out
br_en
IF / ID
opcode, funct3, funct7
Control ROM
ctrl_word
rs1
rs2
Regfile
rs1_out
rs2_out
regfile_wdata
ld_regfile
rd
Stall signal pauses all register loads in pipe
dcache_stall
icache_stall
ID / EX
opcode
funct3
CMP
br_en_cmp
JMP Logic
br_en
cmpmux_sel
alumux1_sel
PC
alumux2_sel
i_imm
u_imm
s_imm
b_imm
j_imm
ALU
alu_out
aluop
rs1_forward
rs2_forward
exmem_alu_out
dcache_rdata
rs2forw_mux_sel
rs1forw_mux_sel
Forwarding Unit
Inputs seen in Forwarding Unit Logic page
dcache_stall
dCache Stall Ctrl
dcache_resp
dCache
dcache_read
dcache_write
dcache_address
dcache_wdata
dcache_mbe
dcache_rdata
EX / MEM
wData Logic
mem_rs2_forward
memforw_mux_sel
Mem Forwarding Unit
Inputs seen in Mem Forwarding Unit Logic page
MEM / WB
regfilemux_sel
memwb_alu_out
PC+4
br_en
u_imm
mem_rdata
0
1
2
3
4
rd
Shortcut signals connected from another side of CPU
Stall/Flush Signals
Forwarding Signals
Signals between pipeline registers (ctrl, instruction, pc...)
All control signals come from their stages' source pipe (ex: regfilemux_sel comes from memwb_ctrlreg.regfilemux_sel)
flush_sig
Predicted br_en
Branch Flushing Unit
Flush signal resets stages (Loads nop instr/ctrl_word into pipe)

CMP

Key:
u(<) : Unsigned less than
u(>=) : Unsigned greater than equal to



JMP Logic



Forwarding Unit Logic

Inputs:
idex_ireg_out.rs1
idex_ireg_out.rs2
exmem_ireg_out.rd
exmem_ireg_out.opcode
memwb_ireg_out.rd
exmem_ctrlreg_out.regfile_ld
memwb_ctrlreg_out.regfile_ld

Outputs:
rs1forw_mux_sel
rs2forw_mux_sel

Defaults:
rs1forw_mux_sel = forw_mux::idex_rs1reg_out
rs2forw_mux_sel = forw_mux::idex_rs1reg_out

// RS1

if exmem_ctrlreg_out.regfile_ld & (exmem_ireg_out.rd != 0)
& (exmem_ireg_out.rd = idex_ireg_out.rs1))
    if (exmem_ireg_out.opcode == op_load)
        rs1forw_mux_sel = forw_mux::dcache_rdata
    else
        rs1forw_mux_sel = forw_mux::exmem_alureg_out
else if (memwb_ctrlreg_out.regfile_ld & (memwb_ireg_out.rd != 0)
& !(exmem_ctrlreg_out.regfile_ld & (exmem_ireg_out.rd != 0)
& (exmem_ireg_out.rd == idex_ireg_out.rs1))
& (memwb_ireg_out.rd = idex_ireg_out.rs1))
    rs1forw_mux_sel = forw_mux::regfile_wdata

// RS2

if exmem_ctrlreg_out.regfile_ld & (exmem_ireg_out.rd != 0)
& (exmem_ireg_out.rd = idex_ireg_out.rs2))
    if (exmem_ireg_out.opcode == op_load)
        rs2forw_mux_sel = forw_mux::dcache_rdata
    else
        rs2forw_mux_sel = forw_mux::exmem_alureg_out
else if (memwb_ctrlreg_out.regfile_ld & (memwb_ireg_out.rd != 0)
& !(exmem_ctrlreg_out.regfile_ld & (exmem_ireg_out.rd != 0)
& (exmem_ireg_out.rd == idex_ireg_out.rs1))
& (memwb_ireg_out.rd = idex_ireg_out.rs1))
    rs1forw_mux_sel = forw_mux::regfile_wdata

Mem Forwarding Unit Logic

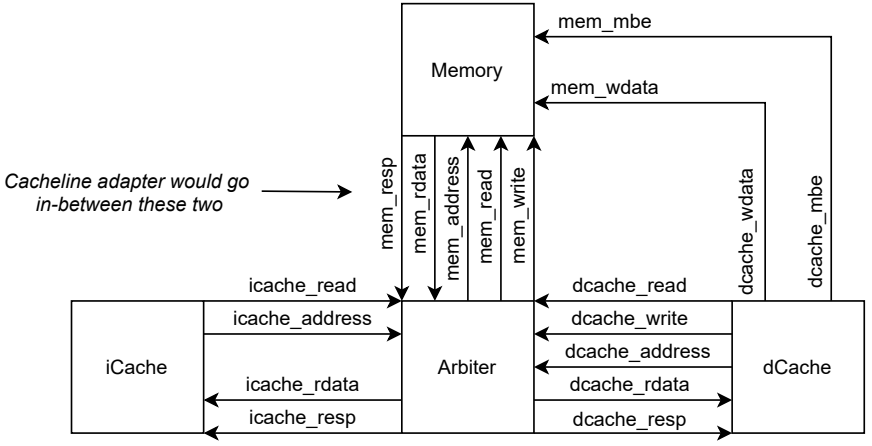
Inputs:
idex_ireg_out.opcode
idex_ireg_out.rs2
memwb_ireg_out.rd
memwb_ctrlreg_out.regfile_ld

Outputs:
memforw_mux_sel

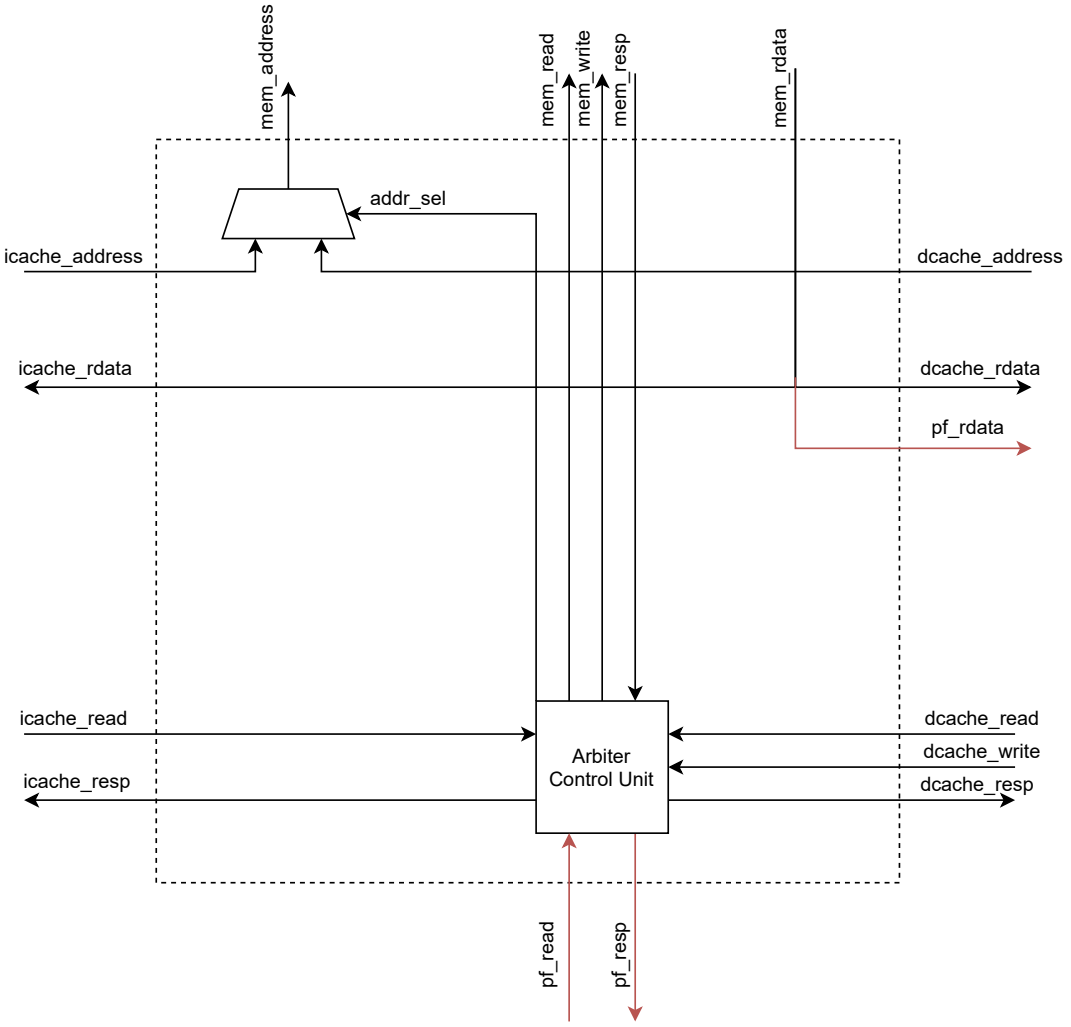
Defaults:
memforw_mux_sel = memforw_mux::exmem_rs2reg_out

```
if (exmem_ireg_out.opcode == op_store
&& memwb_ctrlreg_out.regfile_ld == 1'b1
&& memwb_ireg_out.rd != 0
&& memwb_ireg_out.rd == exmem_ireg_out.rs2)
        memforwmux_sel = memforw_mux::regfilemux_out
else
        memforwmux_sel = memforw_mux::exmem_rs2reg_out
```

Memory Hierarchy

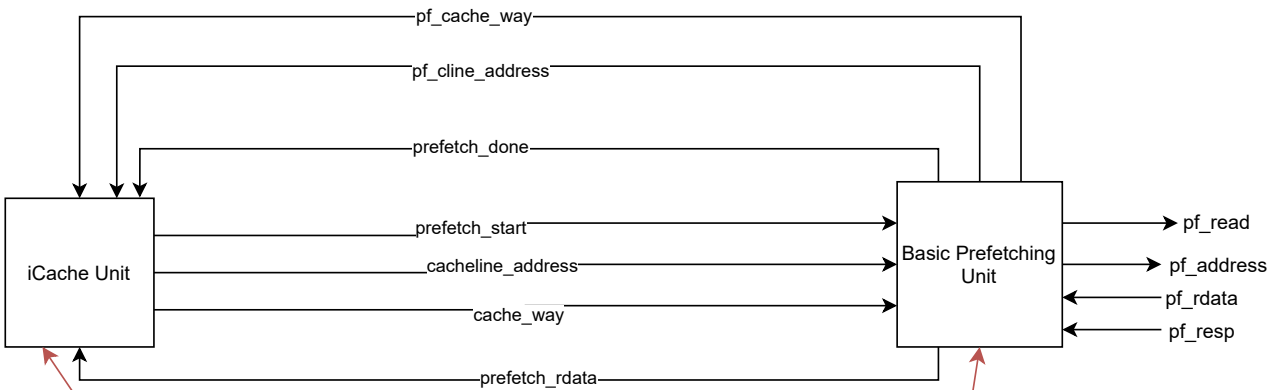


Arbiter


mem_address
mem_read
mem_write
mem_resp
mem_rdata
addr_sel
icache_address
dcache_address
icache_rdata
dcache_rdata
pf_rdata
icache_read
dcache_read
dcache_write
icache_resp
dcache_resp
Arbiter
Control Unit
pf_read
pf_resp

Advanced Features

Prefetching Unit



The iCACHE would now have a new state called prefetch_handle that would load the data from prefetch_rdata into the right index, then go back to processing normal cache requests

We would thus need extra logic to differentiate between a prefetch load and a normal load (extra muxes to our cache indices)

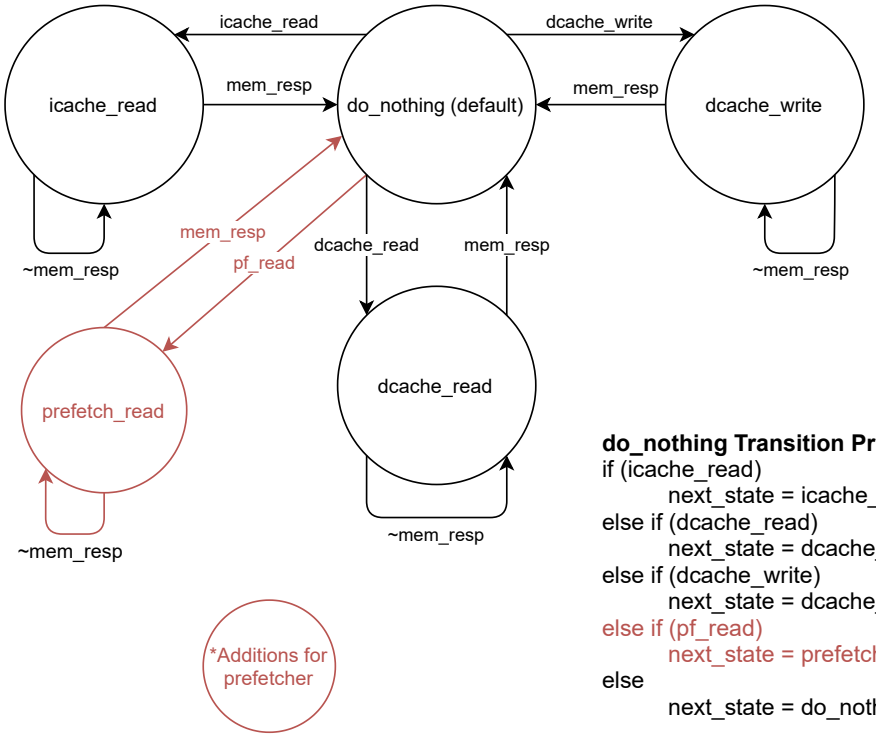
We would also need to make sure that our way selection logic takes in pf_cache_way

Prefetch interacts with Arbiter with this input

Arbiter now has a new state to handle prefetch requests

However, new state would be at lowest priority (after instruction and data read requests)

Arbiter Control Unit State Machine



do_nothing Transition Priority:

```
if (icache_read)
        next_state = icache_read
else if (dcache_read)
        next_state = dcache_read
else if (dcache_write)
        next_state = dcache_write
else if (pf_read)
        next_state = prefetch_read
else
        next_state = do_nothing
```

Output at each state:

do_nothing
- mem_read = 0
- mem_write = 0
- addr_sel = addr_mux::icache_addr
- icache_resp = 0
- dcache_resp = 0
- pf_resp = 0

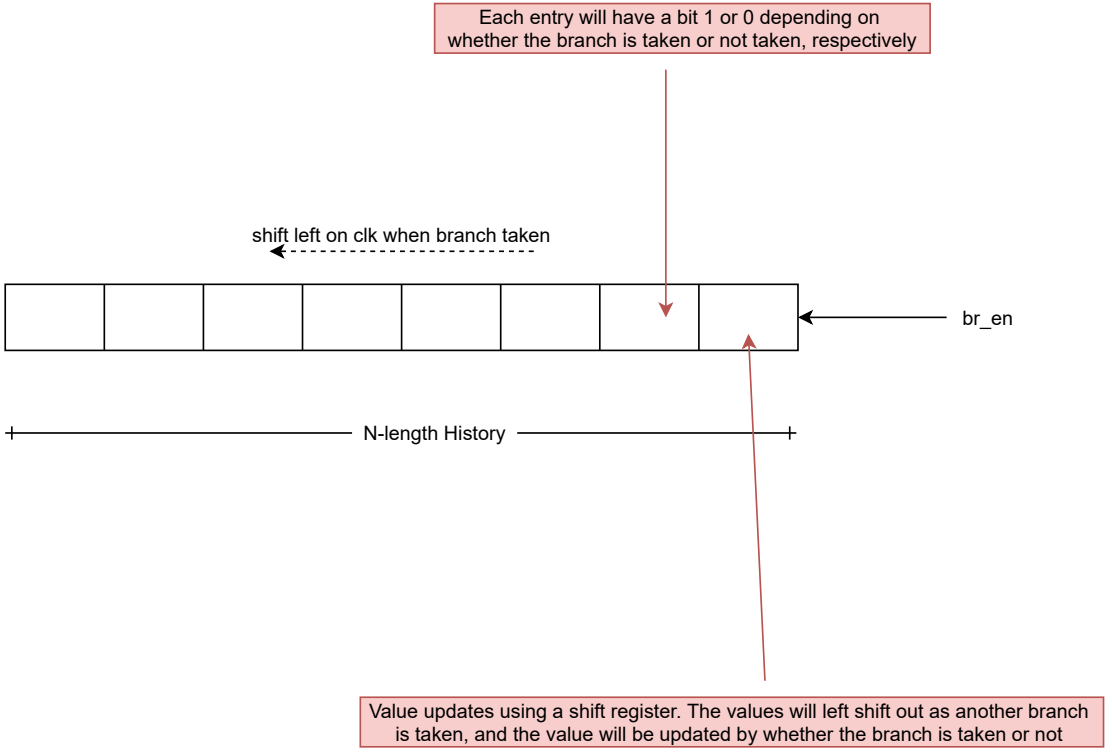
icache_read
- mem_read = 1
- mem_write = 0
- addr_sel = addr_mux::icache_addr
- icache_resp = mem_resp
- dcache_resp = 0
- pf_resp = 0

dcache_read
- mem_read = 1
- mem_write = 0
- addr_sel = addr_mux::dcache_addr
- dcache_resp = mem_resp
- icache_resp = 0
- pf_resp = 0

dcache_write
- mem_read = 0
- mem_write = 1
- addr_sel = addr_mux::dcache_addr
- dcache_resp = mem_resp
- icache_resp = 0
- pf_resp = 0

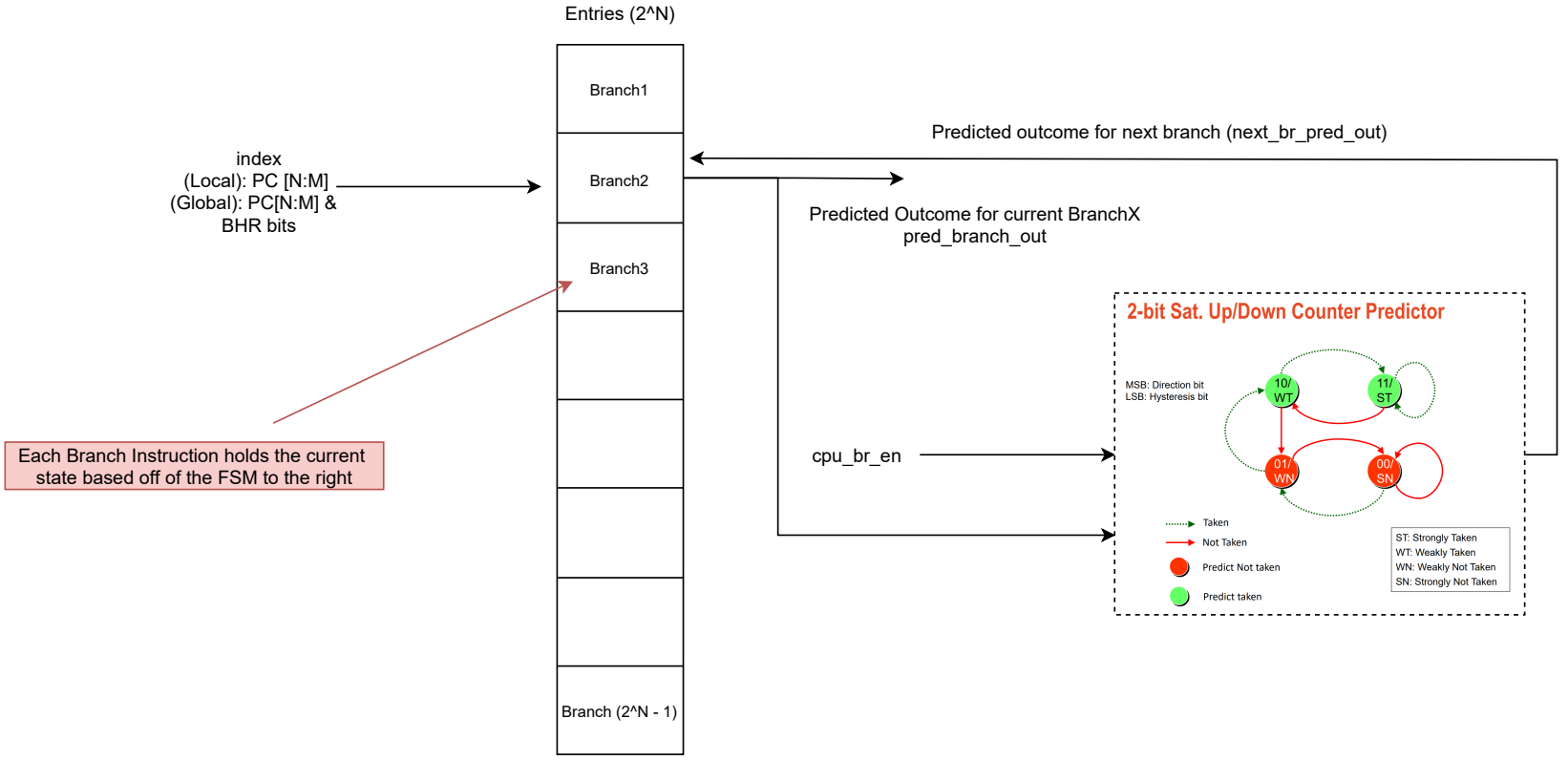
prefetch_read
- mem_read = 1
- mem_write = 0
- addr_sel = addr_mux::pf_addr
- pf_resp = mem_resp
- dcache_resp = 0
- icache_resp = 0

Branch History Register



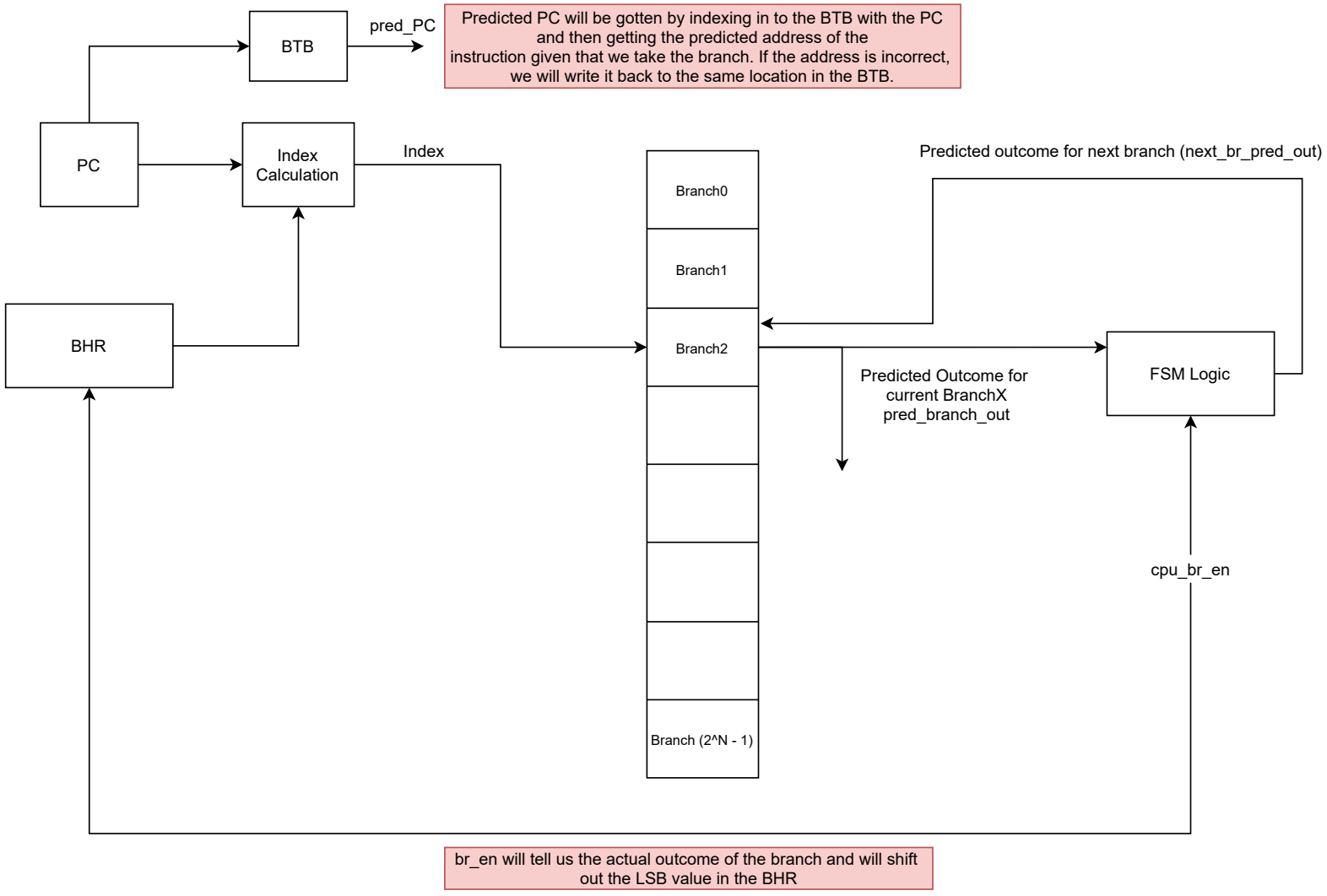
This will keep track of the last N branches, and the N-length history is used as part of the index for the PHT.

Pattern History Table (Local History Table & 2nd Level of Global Predictor)

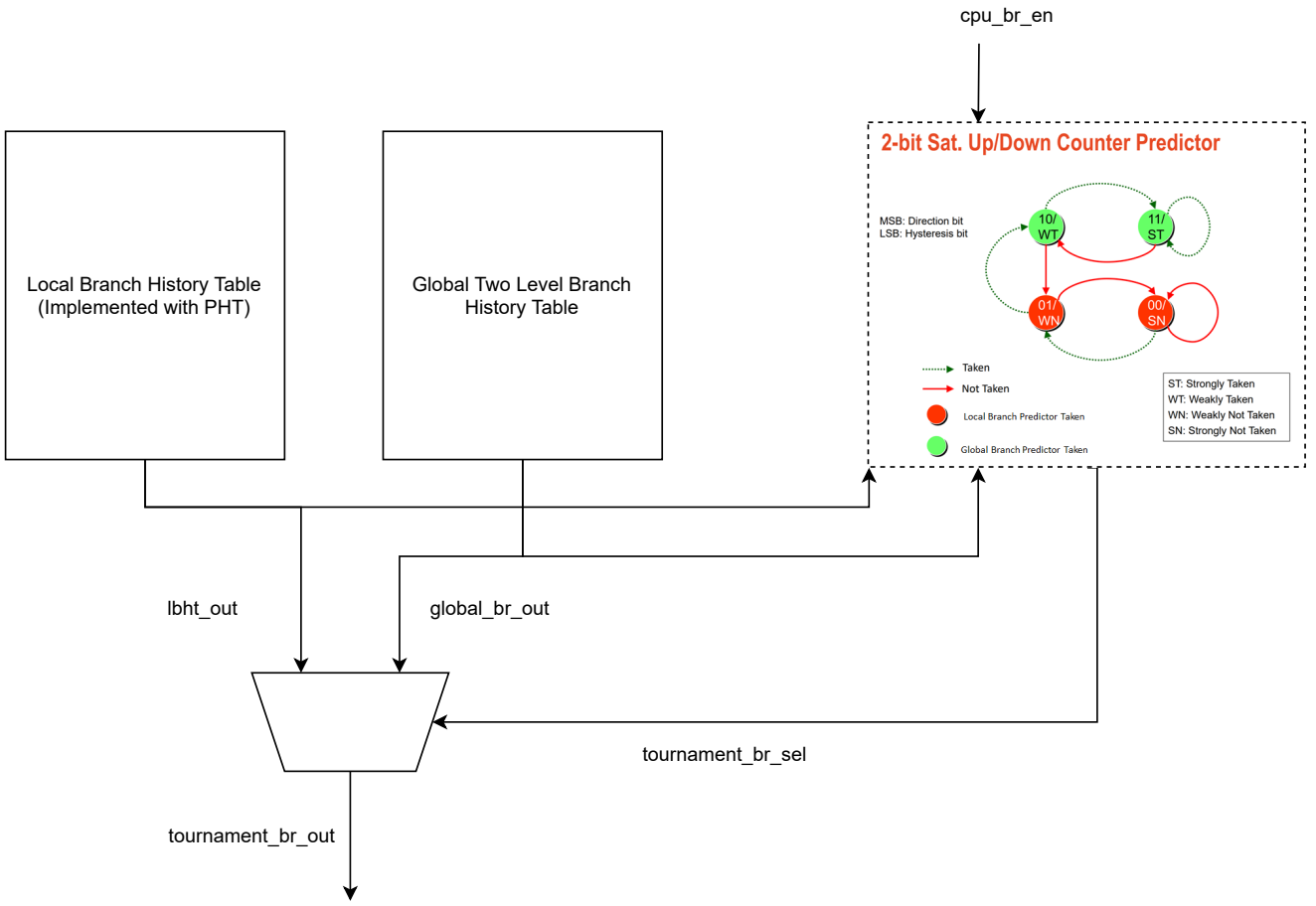


The FSM shown to the right decides what state the branch entry will be in. This is based off of whether a specific branch is taken or not taken. The state inside the table will be updated depending on the FSM.

Global Predictor

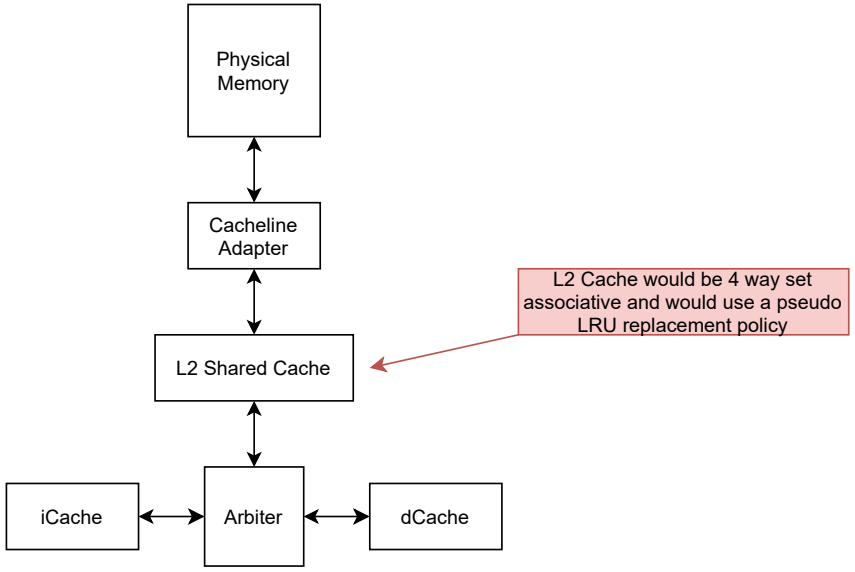


Tournament Predictor



Advanced Cache

L2 Cache, 4-way set associative cache, parameterized cache

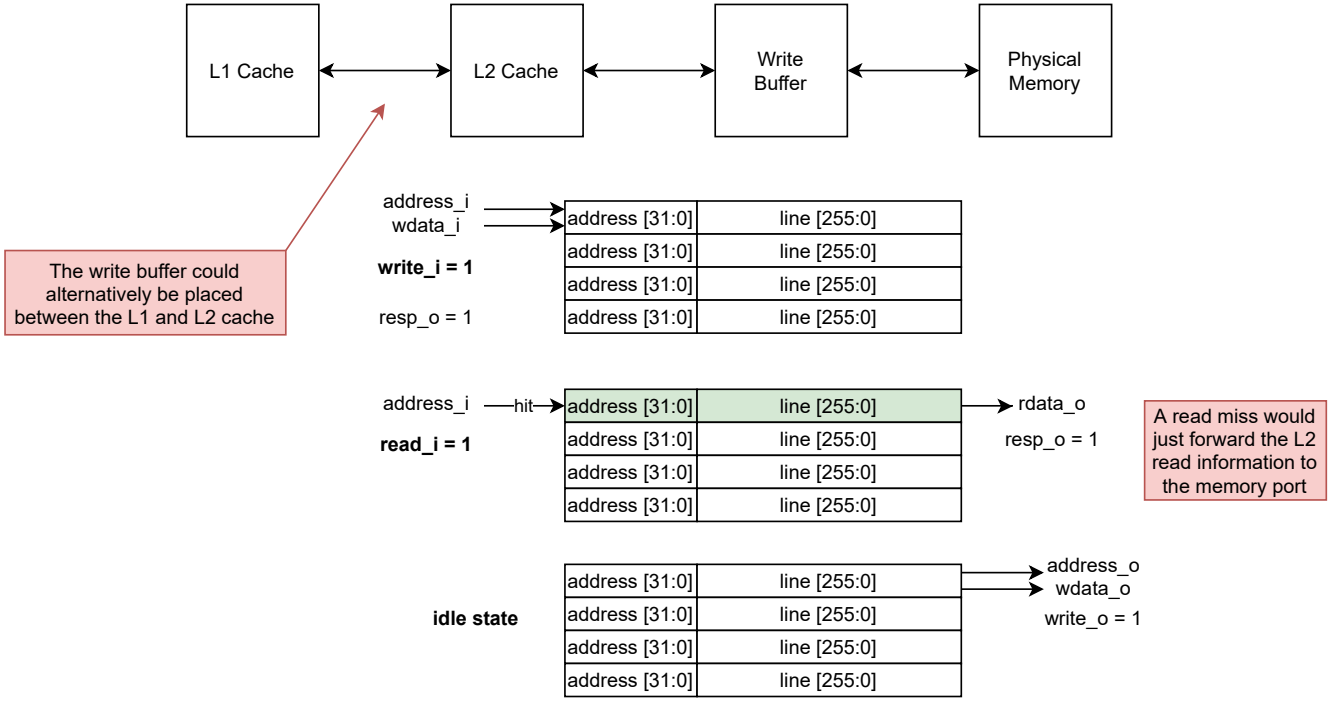


The 4-way cache would use a pseudo LRU to approximate replacement instead of a complete one because it is more efficient. The hardware needed to have a true replacement policy would need to keep an order of previous line accesses which has 4!=24 different scenarios. This would be expensive and not worth. A binary tree instead would only need 3 bits to approximate.

Since the L2 cache does not have to respond to hits in one cycle like the other caches do, it can respond to hits in 2 cycles like in mp3. However, we are not sure yet what the final hit response time will be for our L2 cache.

We also can parameterize the number of sets in our cache. This gives us an option in the future to tune our memory system by adding more sets. This makes the cache footprint bigger but it can allow more hits. The tradeoffs for this will be evaluated in more detail further down the road when we try to maximize performance.

Write Buffer

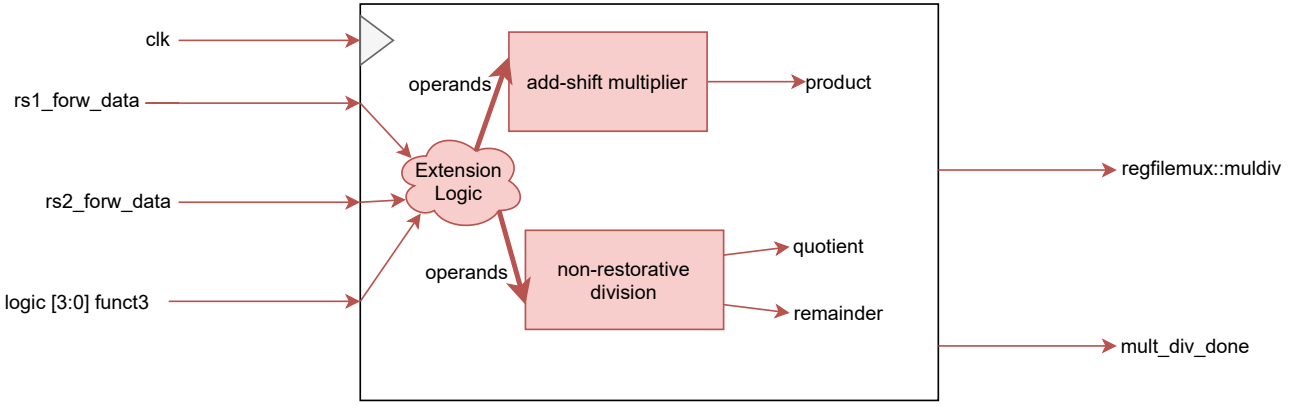


The write buffer has to sit in between levels of memory to intercept incoming writes. This lets lower level caches not waste as much time waiting for writebacks and can instead move on to getting read responses.

The buffer itself is similar to a cache in that it would need to have multiple lines to store information and logic to check if there was a hit on of them.

For incoming writes, if there is space, it stores the data and the address in its internal buffer. If there is no space, it is forced to write it back. For incoming reads if the address hits, the buffer is outputted because it has the most up to date copy. If the read misses, the read request is forwarded to the next level. When the buffer is in a idle state, it can choose a buffer to write back.

Multiplier/Division Unit



We will 'prep' the operands in order to perform the correct operation, by extending the operands according to the opcode

Our control word would most likely have a load_muldiv signal to know when to load the multiplier/divider unit and start the operation. Then, the EX-MEM would be stalled until the requested operation is done (checked by mult_div_done signal)

Our multiplier/divider unit takes in either the original or forwarded data for the registers and then also takes in the operand for the type of operation. This would then choose what unit to activate and also whether to use the product, quotient or remainder buffer

Algorithm for multiplication:
Add-Shift Multiplier from ECE 385

Algorithm for division and remainder:
https://ieeexplore.ieee.org/document/146763
"A hardware algorithm for integer division"