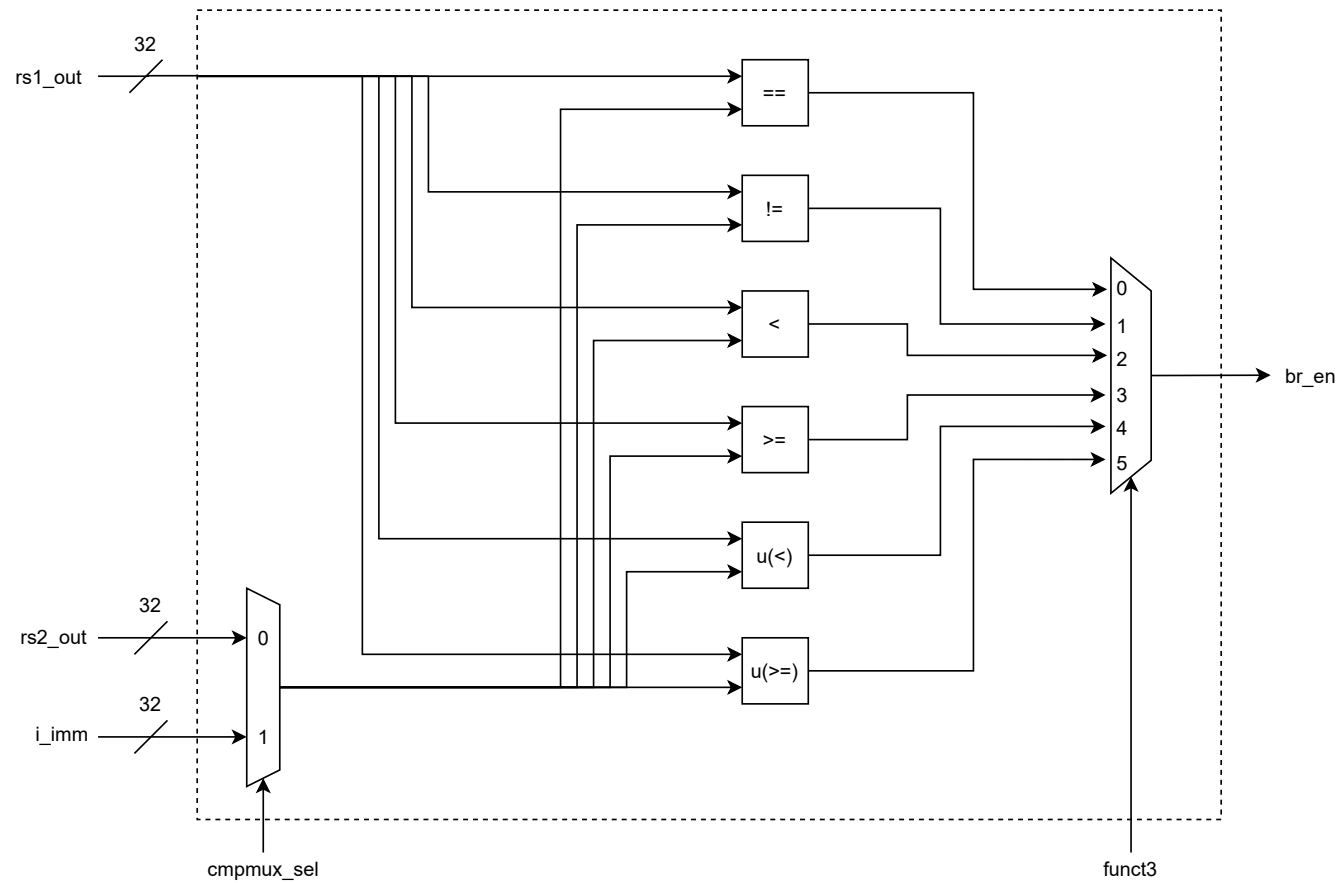


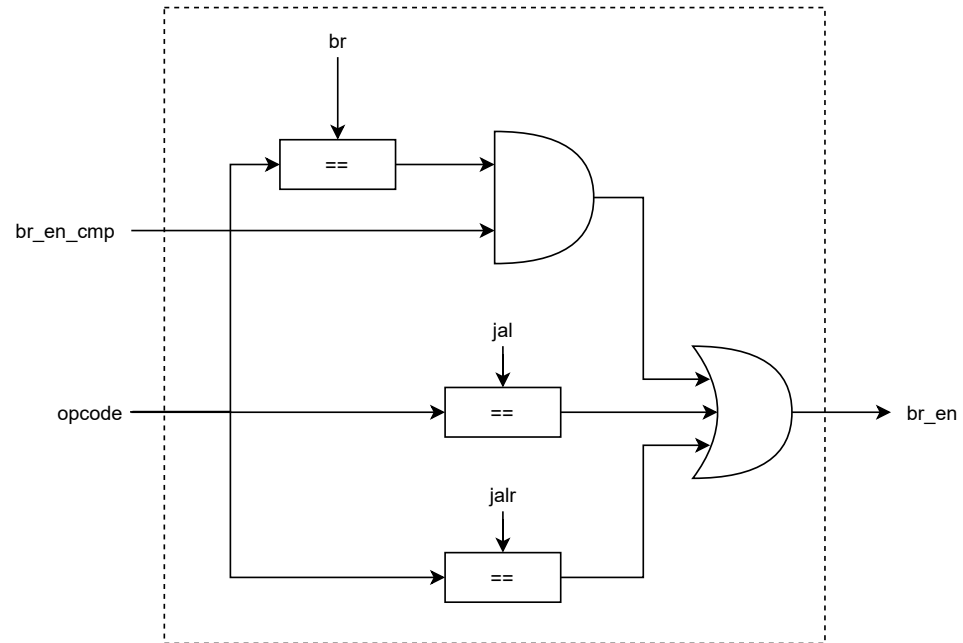
# CMP

## 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\_id

## Outputs:

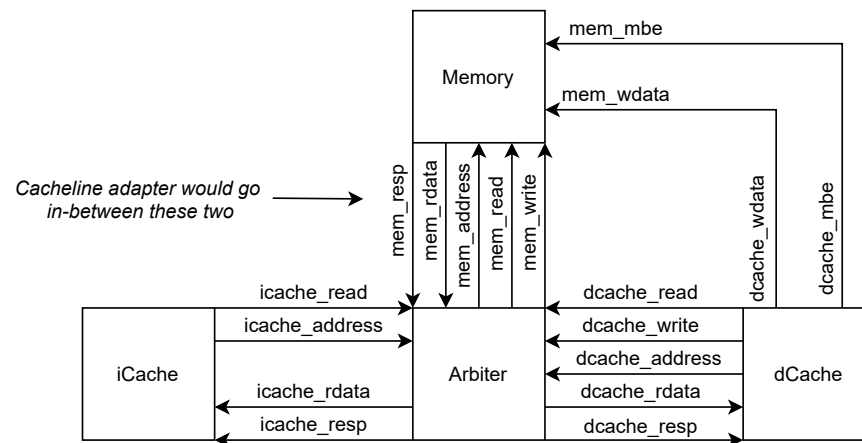
memforw\_mux\_sel

## Defaults:

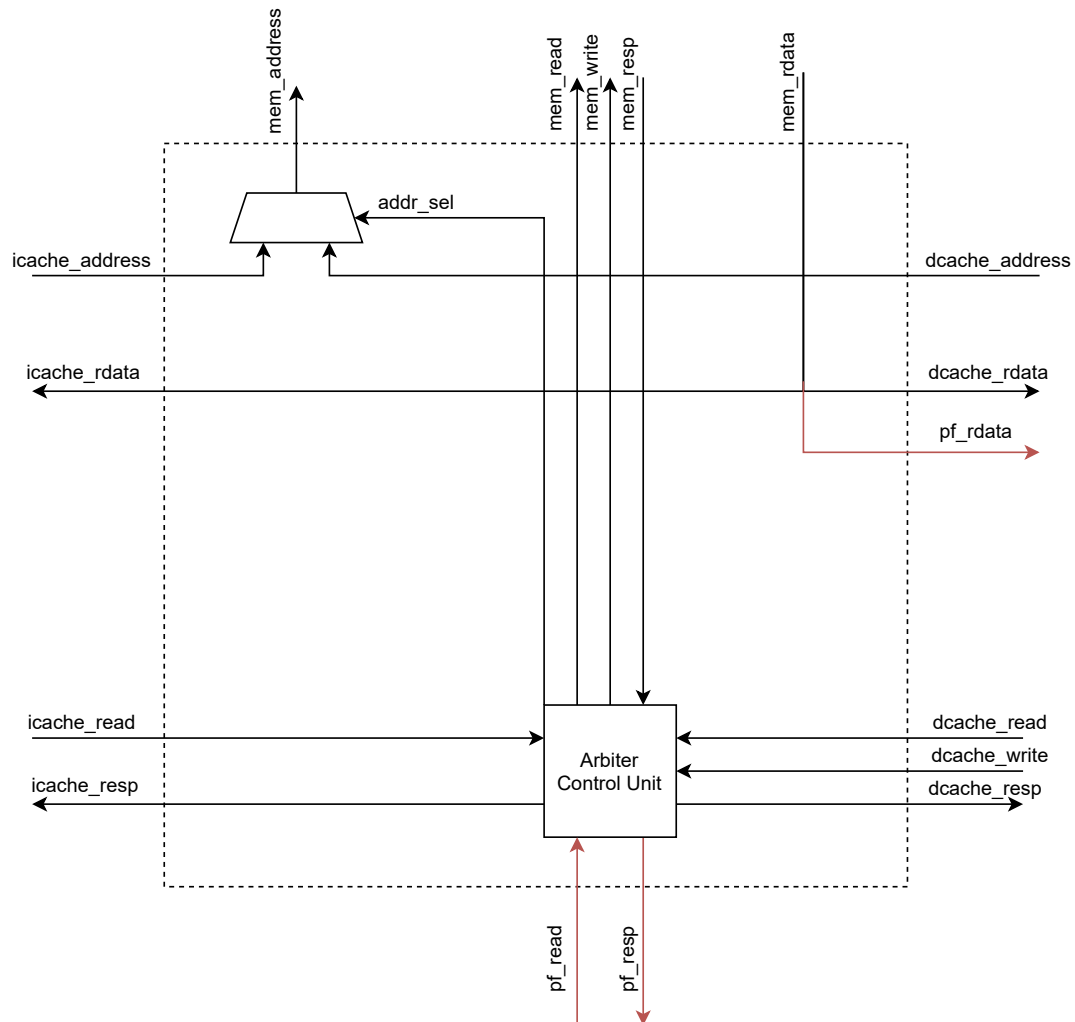
memforw\_mux\_sel = memforw\_mux::exmem\_rs2reg\_out

```
if (exmem_ireg_out.opcode == op_store
    && memwb_ctrlreg_out.regfile_id == 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



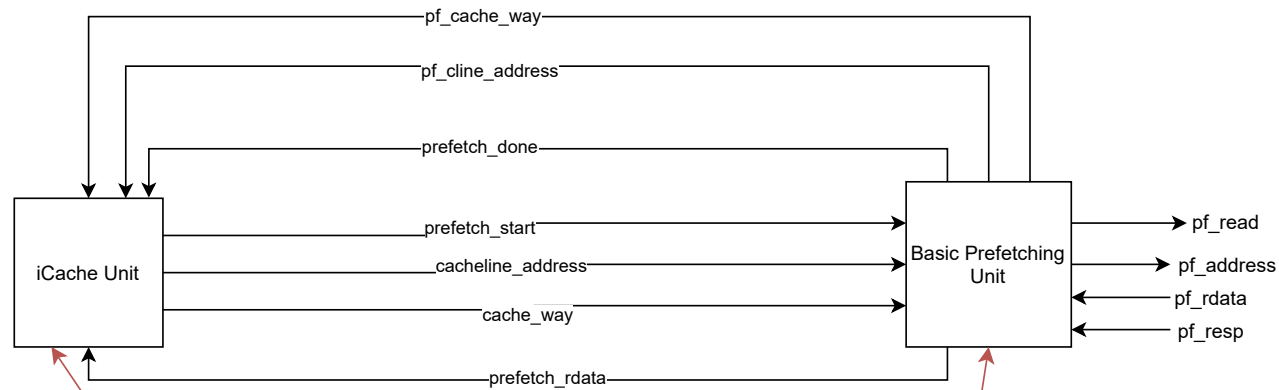
# Arbiter



# 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)

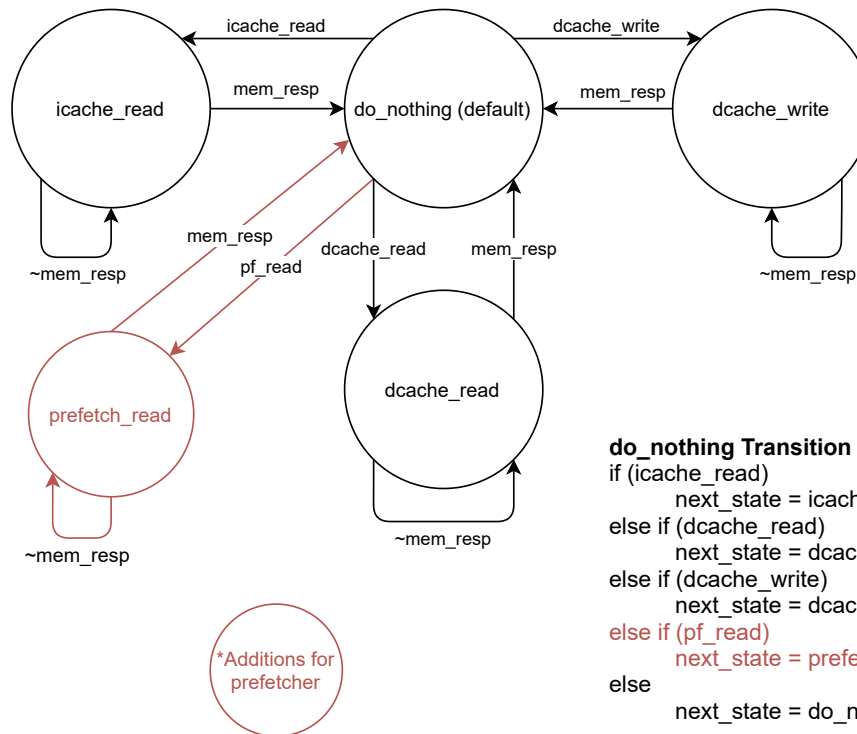
We would also need to make sure that our way selection logic takes in `pf_cache_way`

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

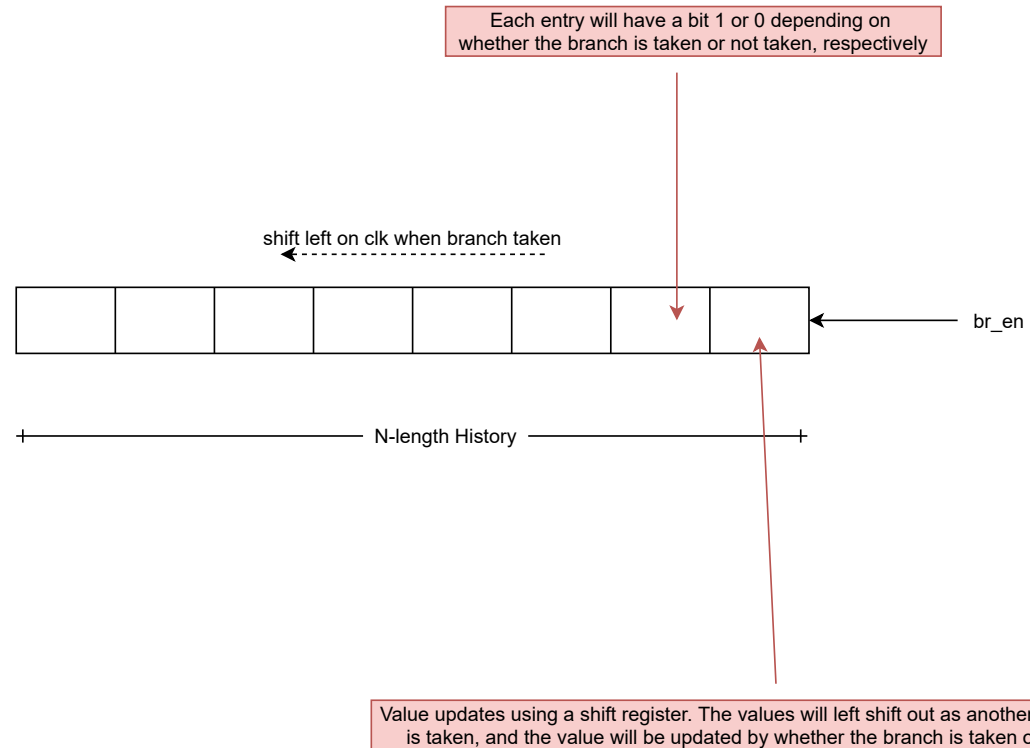
### dcache\_write

- mem\_read = 0
- mem\_write = 1
- addr\_sel = addr\_mux::dcache\_addr
- dcache\_resp = mem\_resp
- icache\_resp = 0
- pf\_resp = 0

### prefetch\_read

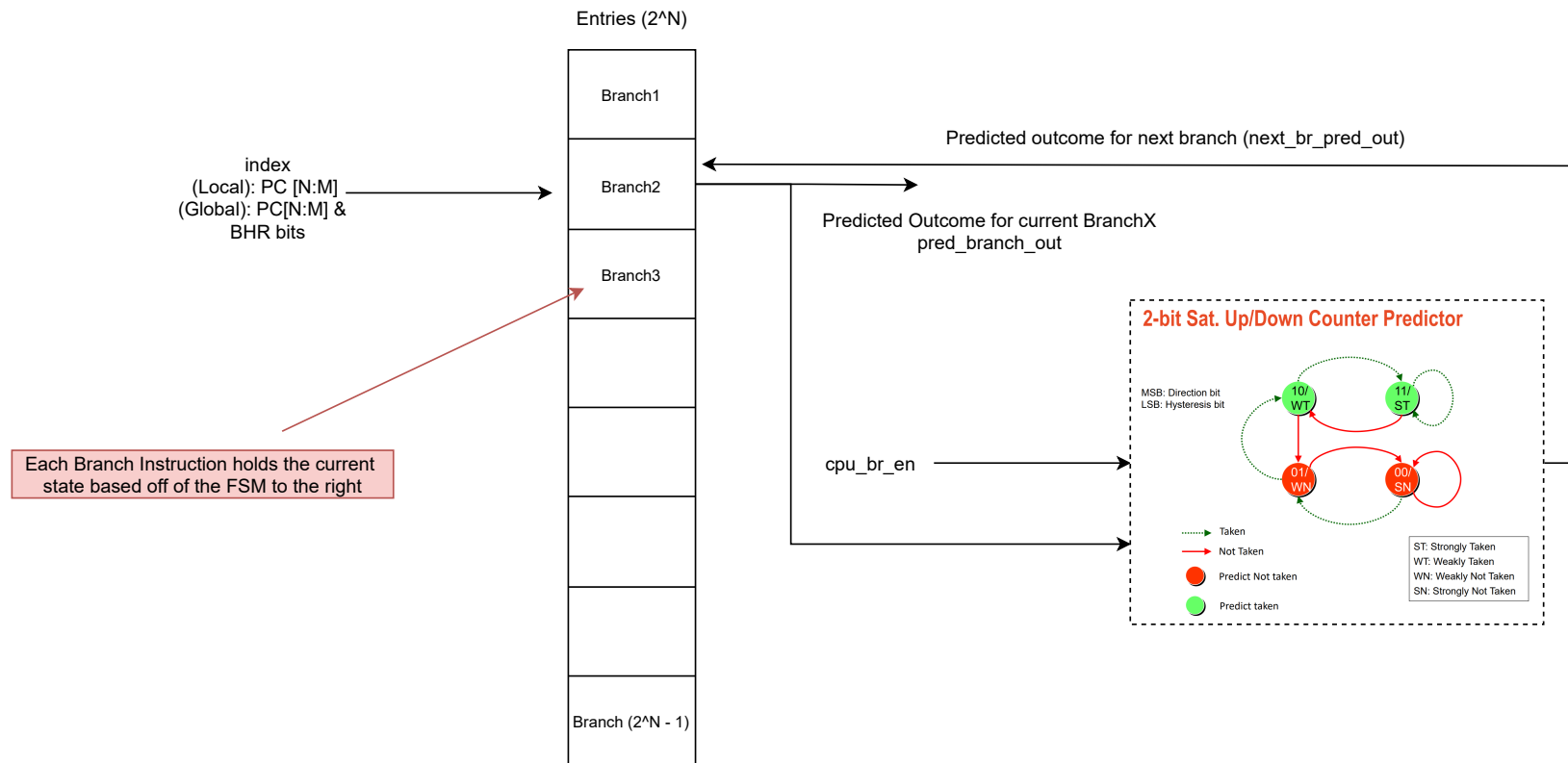
- mem\_read = 1
- mem\_write = 0
- addr\_sel = addr\_mux::pf\_addr
- pf\_resp = mem\_resp
- dcache\_resp = 0
- icache\_resp = 0

# Branch History Register



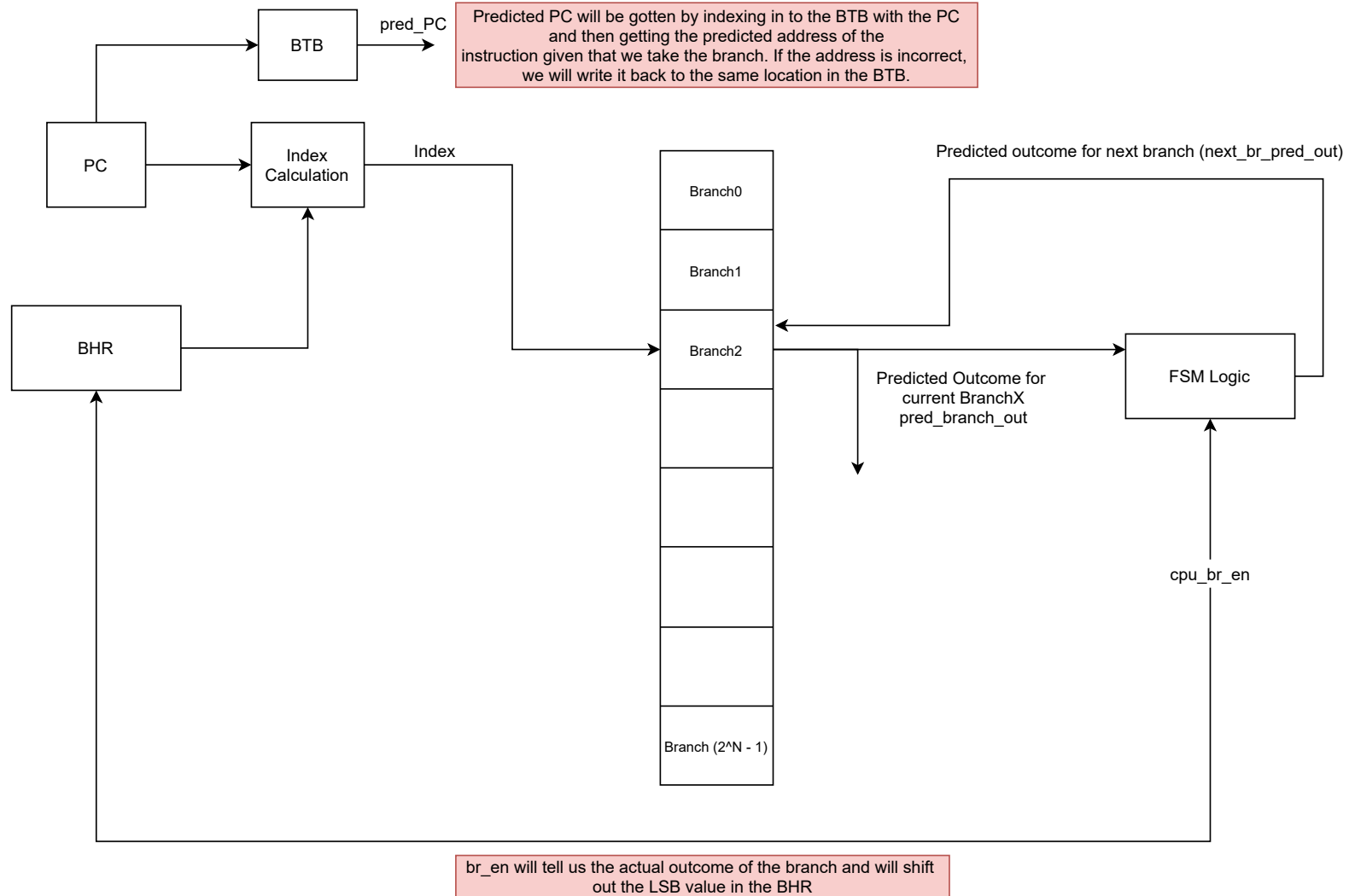
This will keep track of the last N branches, and the N-length history is used as part of the index for the PHT.

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

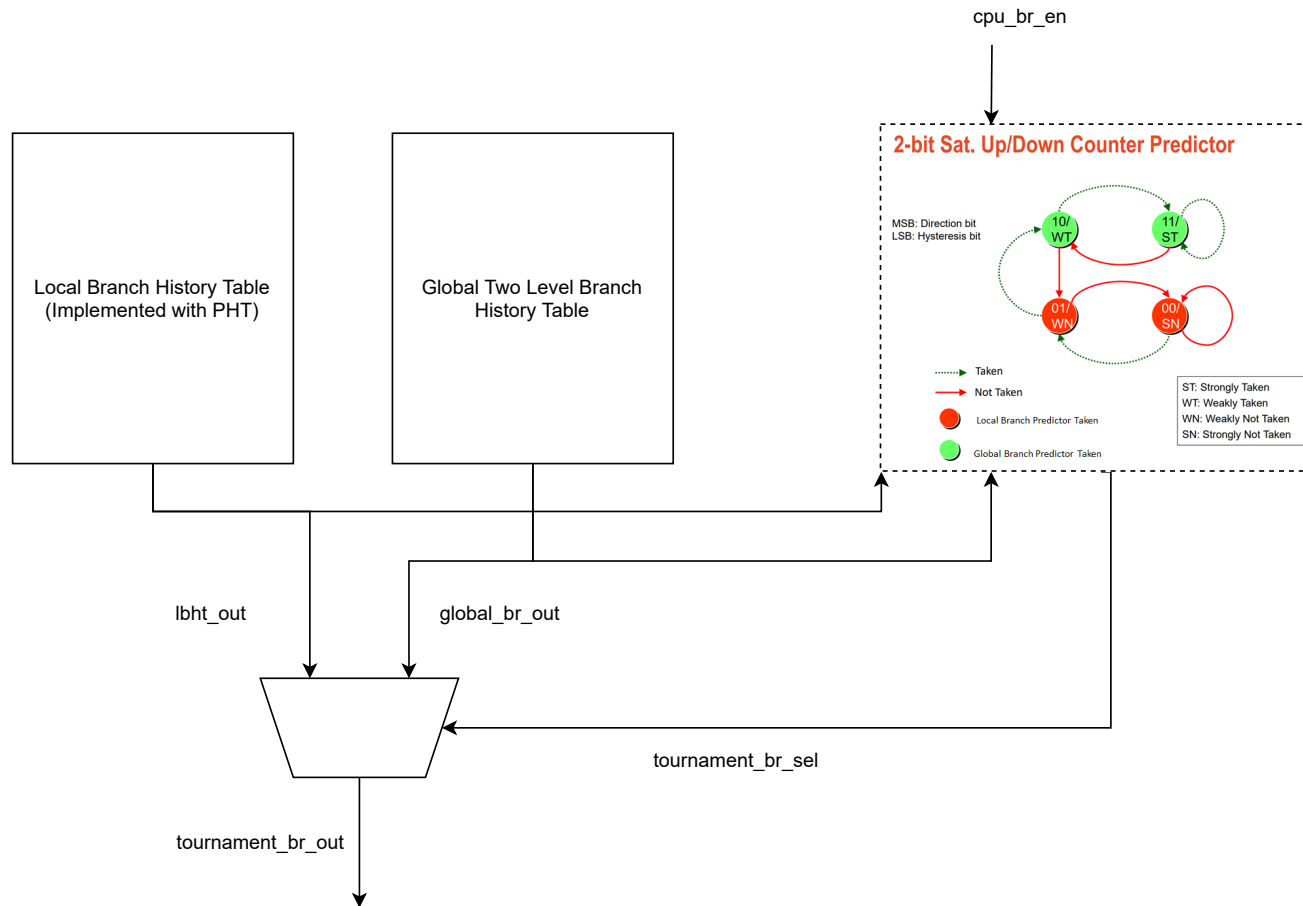


The FSM shown to the right decides what state the branch entry will be in. This is based off of whether a specific branch is taken or not taken. The state inside the table will be updated depending on the FSM.

# Global Predictor

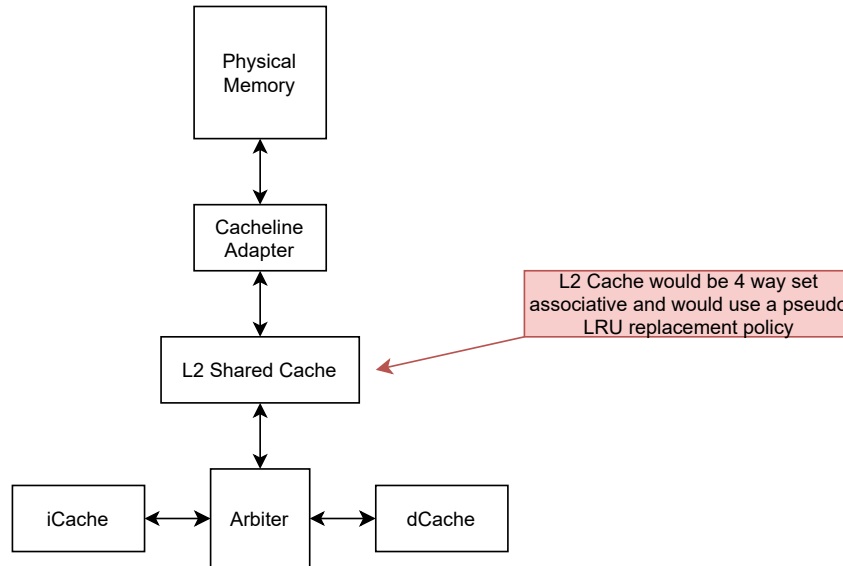


# Tournament Predictor



# Advanced Cache

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

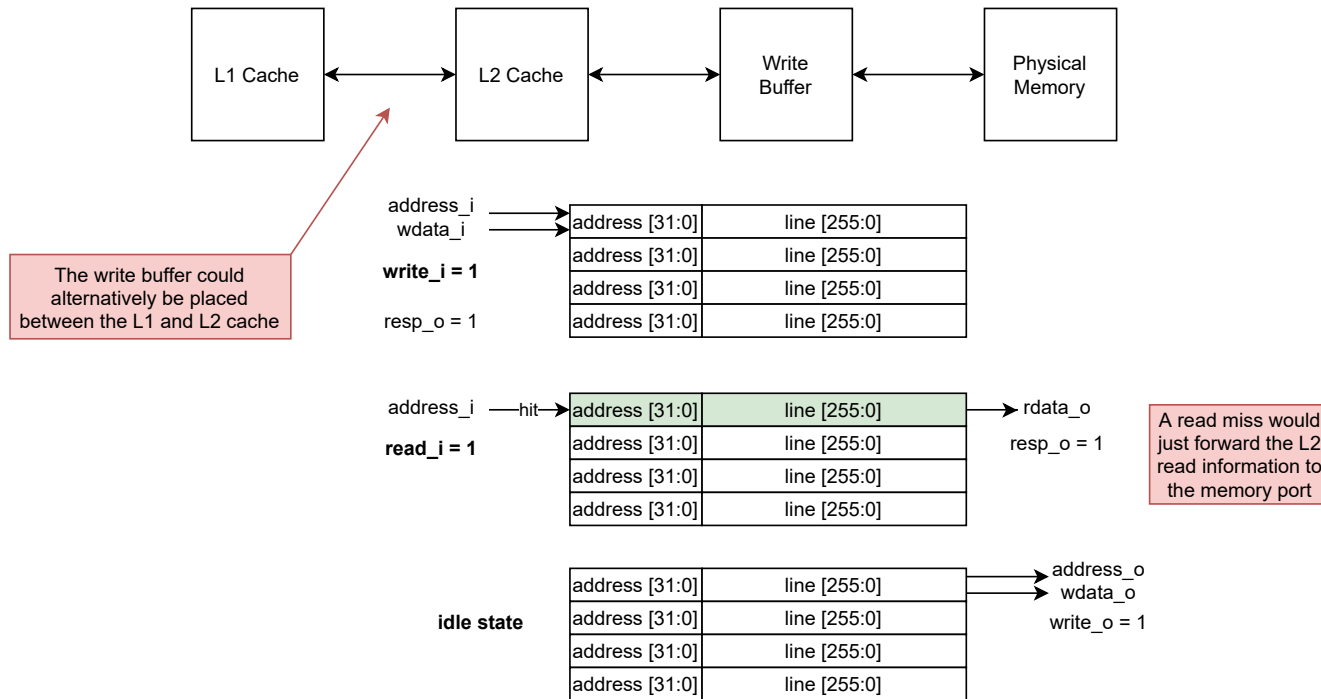


The 4-way cache would use a pseudo LRU to approximate replacement instead of a complete one because it is more efficient. The hardware needed to have a true replacement policy would need to keep an order of previous line accesses which has  $4! = 24$  different scenarios. This would be expensive and 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



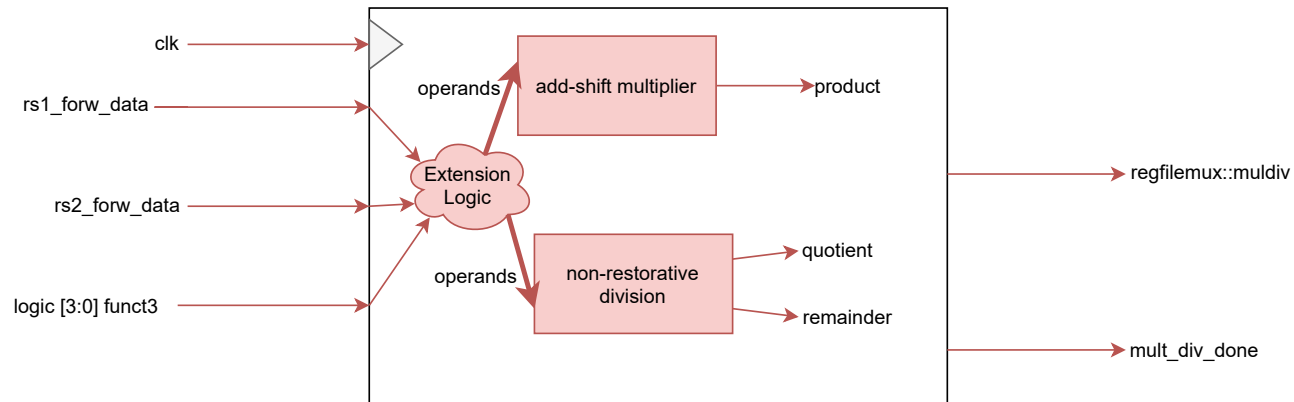
The write buffer has to sit in between levels of memory to intercept incoming writes. This lets lower level caches not waste as much time waiting for writebacks and can instead move on to getting read responses.

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 get the correct operation, by sign-extending them accordingly

Our control word would most likely have a load multiplier signal to know when to load the multiplier. Then, the EX-MEM would be stalled until the multiplication or division operation is done (checked by mult\_div\_done signal)

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

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