mp_cache

Lab 1

Announcements

- How are things?
    - How are recent adjustments helping?
- Checkpoints no longer exist for mp_cache
    - This session will still focus on what would have been CP1 (cache reads)
- Final (and only) deadline for mp_cache is 3/8/2024
- mp_bp cancelled

Overview

- Cache Overview
- Cache State Machine
- Cache Datapath
- Cache Addressing
- PLRU
- Verification

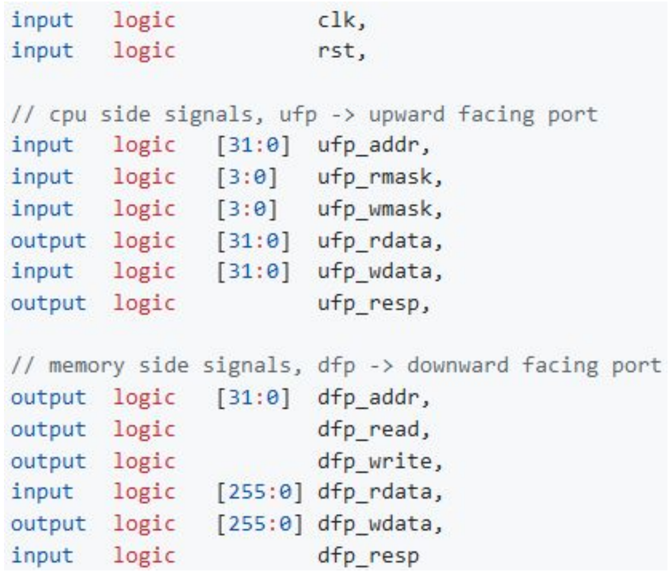
Cache Overview

Cache Specs & Interface

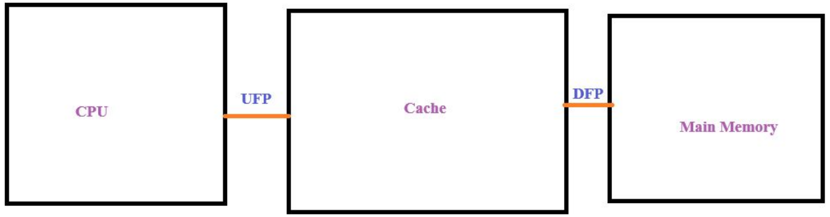
- Specs -> **MUST BE FOLLOWED**!
  - 16 sets
  - 4 ways
  - 32 byte (256-bit) cachelines
  - Write-back policy
  - Write allocate policy
  - PLRU replacement
- Ports
  - UFP: CPU <-> Cache
  - DFP: Cache <-> Main Memory

Cache Specs & Interface

- Specs -> **MUST BE FOLLOWED**!
    - 16 sets
    - 4 ways
    - 32 byte (256-bit) cachelines
    - Write-back policy
    - Write allocate policy
    - PLRU replacement
- Ports
    - UFP: CPU <-> Cache
    - DFP: Cache <-> Main Memory

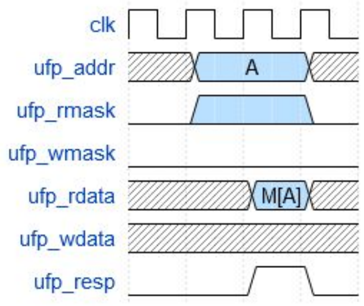


```
input   logic           clk,
input   logic           rst,

// cpu side signals, ufp -> upward facing port
input   logic   [31:0]  ufp_addr,
input   logic   [3:0]   ufp_rmask,
input   logic   [3:0]   ufp_wmask,
output  logic   [31:0]  ufp_rdata,
input   logic   [31:0]  ufp_wdata,
output  logic           ufp_resp,

// memory side signals, dfp -> downward facing port
output  logic   [31:0]  dfp_addr,
output  logic           dfp_read,
output  logic           dfp_write,
input   logic   [255:0] dfp_rdata,
output  logic   [255:0] dfp_wdata,
input   logic           dfp_resp
```

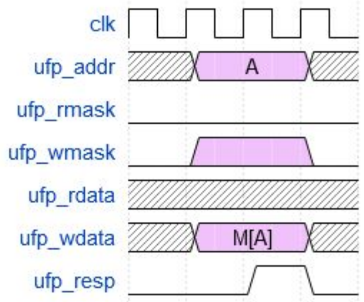


Cache Timings

Hit Timings

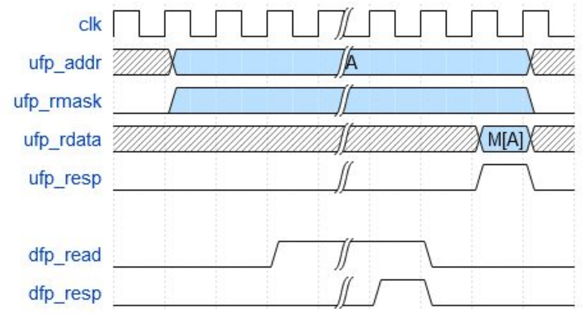


Read hit timing diagram

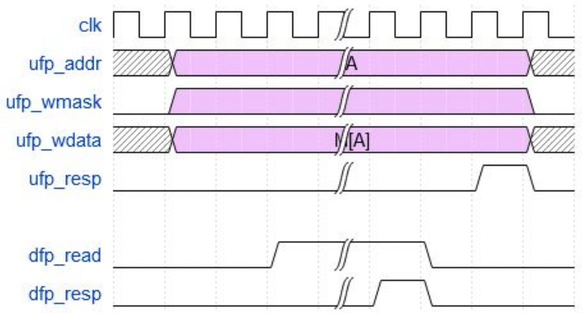


Write hit timing diagram

Clean Miss Timings

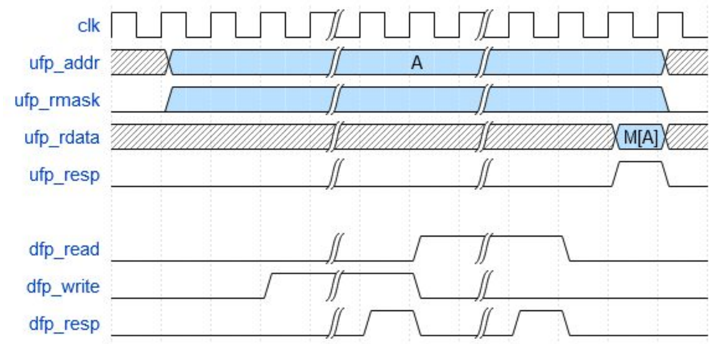


Read clean miss timing diagram

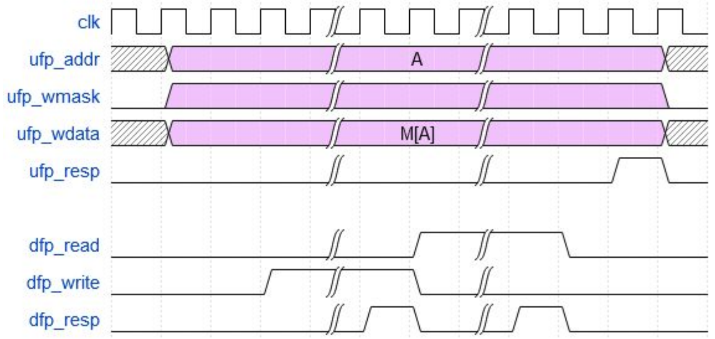


Write clean miss timing diagram

Dirty Miss Timings



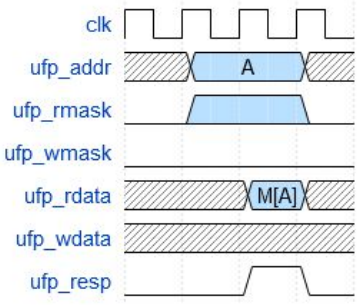
Read dirty miss timing diagram



Write dirty miss timing diagram

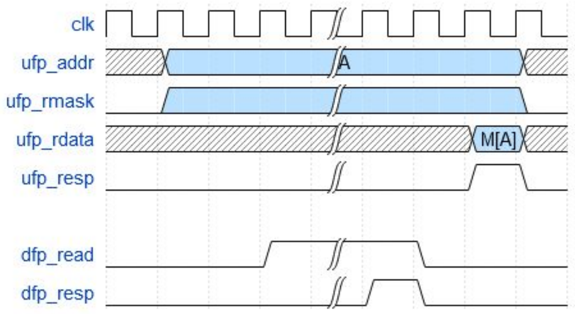
Cache Timings

Hit Timings



Read hit timing diagram

Clean Miss Timings



Read clean miss timing diagram

TIMINGS MUST BE FOLLOWED!!

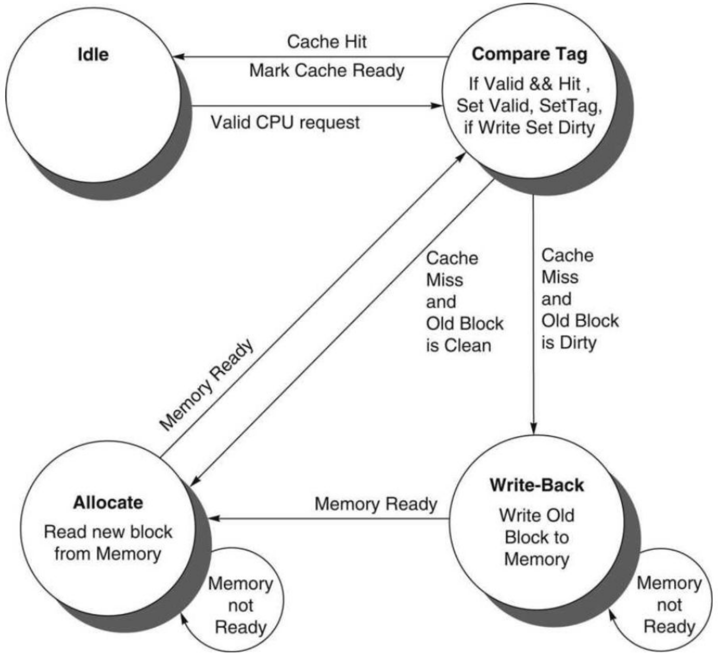
How Do We Translate a Spec into a Design?

- At a top level, what are the steps the cache must do to
    - Satisfy CPU requests
    - Do it within the appropriate time
    - Answer with a **state machine**
- What is the hardware I need to complete these steps?
    - Logic that determines transition of steps
        - -> control unit for FSM
    - Hardware to bookkeep all our data and its state
        - -> Arrays of data
    - These comprise the **datapath**
- Design is iterative & varies
    - Varies - Some prefer to start w/ hardware then think about how to govern
    - Iterative
        - Not just do all FSM then do all hardware
        - Sometimes as you do one, you realize you missed something from the other
            - So go back and add it and keep jumping between the two!

Cache State Machine

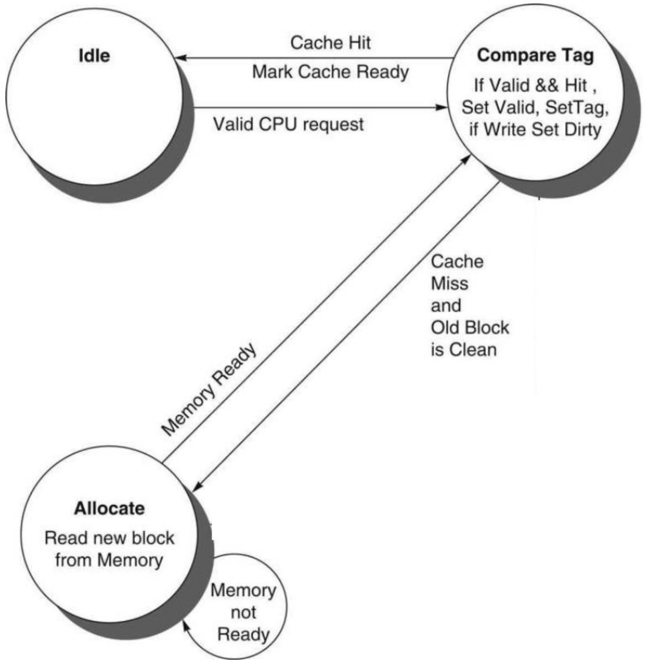
The FSM Answer Key

- Once again directly from the textbook a complete answer key to (part of) your MP!
  - Thank David Patterson & John Hennessy
    - Section 5.9 (pg 453)


Idle
Cache Hit
Mark Cache Ready
Valid CPU request
Compare Tag
If Valid && Hit , Set Valid, SetTag, if Write Set Dirty
Cache Miss and Old Block is Clean
Cache Miss and Old Block is Dirty
Memory Ready
Allocate
Read new block from Memory
Memory not Ready
Memory Ready
Write-Back
Write Old Block to Memory
Memory not Ready

The FSM Answer Key

- Once again directly from the textbook a complete answer key to (part of) your MP!
    - Thank David Patterson & John Hennessy
        - Section 5.9 (pg 453)

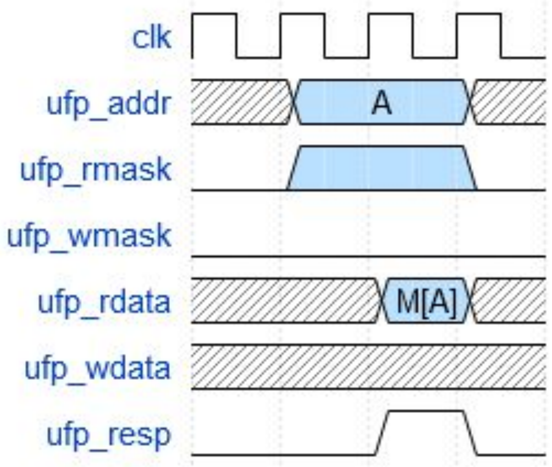

Idle
Cache Hit
Mark Cache Ready
Valid CPU request
Compare Tag
If Valid && Hit , Set Valid, SetTag, if Write Set Dirty
Cache Miss and Old Block is Clean
Memory Ready
Allocate
Read new block from Memory
Memory not Ready

Cache Timings

Hit Timings

Hits only need to observe UFP (CPU <-> Cache signals!)

CPU <-> Cache Signals



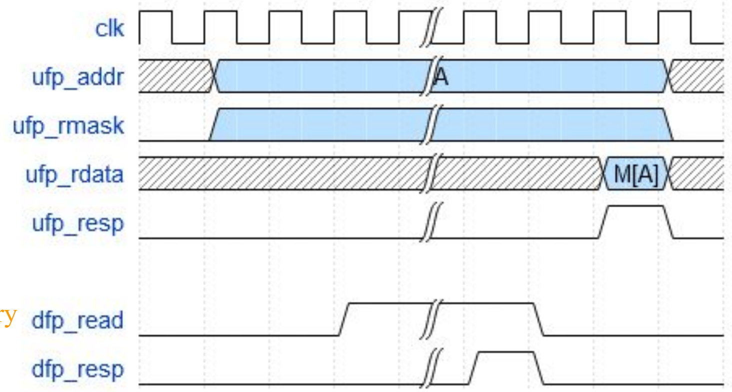
Read hit timing diagram

Clean Miss Timings

Misses need both UFP and DFP!

CPU <-> Cache Signals

CPU <-> Main Memory Signals



Read clean miss timing diagram

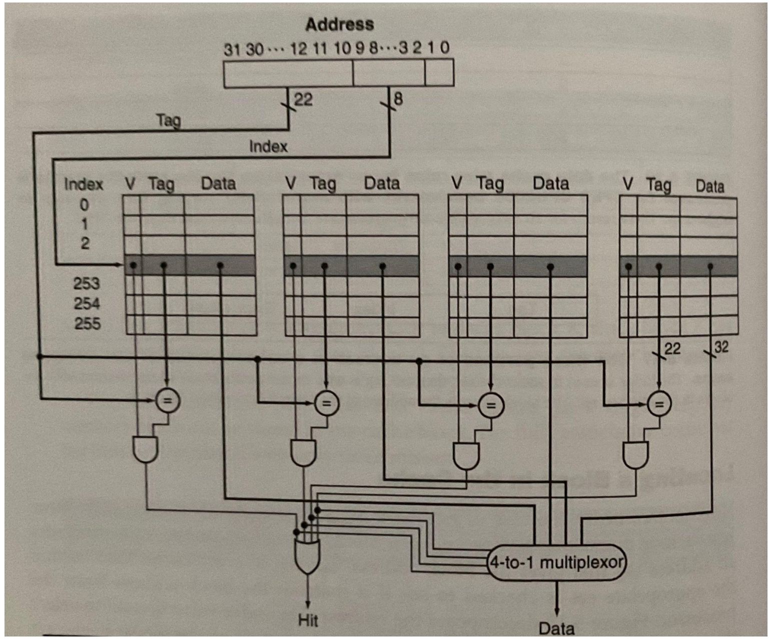
Cache Datapath

What Hardware Do We Need?

- Bookkeeping hardware -> Arrays
    - PLRU
    - Cacheline data
    - Cacheline tag
    - Dirty bit (in tag)(not needed for CP1/read only cache!)
    - Valid bit
- Hit detection & way select logic
    - Need to know if we have data in cache
    - If we have data, which way should supply it
- Communication / address translation logic
    - Addressing into cache from CPU address
    - Data size discrepancy between memory, cache, and CPU
        - Memory <-> Cache translation already “handled” in provided MP files for you (no work!)
        - CPU <-> Cache translation needs to be done by you

What Hardware Do We Need?

- Bookkeeping hardware -> Arrays
  - PLRU
  - Cacheline data
  - Cacheline tag
  - Dirty bit (in tag)(not needed for CP1/read only cache!)
  - Valid bit
- Hit detection & way select logic
  - Need to know if we have data in cache
  - If we have data, which way should supply it
- Communication / address translation logic
  - Addressing into cache from CPU address
  - Data size discrepancy between memory, cache, and CPU
    - Memory <-> Cache translation already "handled" in provided MP files for you (no work!)
    - CPU <-> Cache translation needs to be done by you



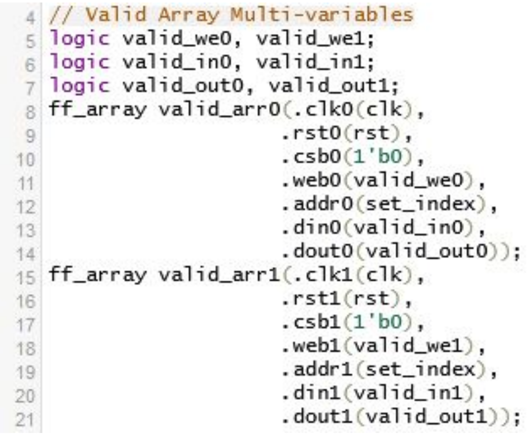
Pg 400 in textbook

Generate Statement

- Lot of duplicated arrays between the ways
  - Can
    - copy
    - paste
    - copy
    - paste . . .
  - Or can use generate statement
    - Instantiate one module multiple times
    - Connect the different instantiations to difference indices of logic variable arrays

Generate Statement

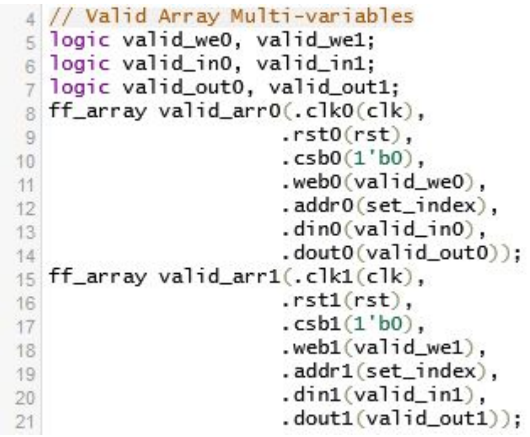
- Lot of duplicated arrays between the ways
  - Can copy, paste, copy, paste . . .
  - Or can use generate statement
    - Instantiate one module multiple times
    - Connect the different instantiations to difference indices of logic variable arrays



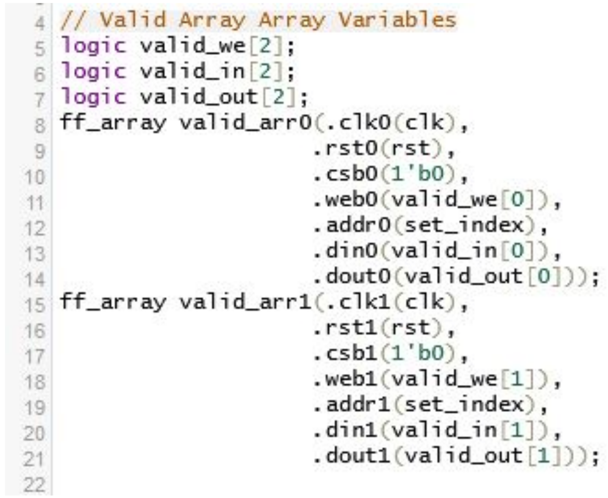
```
// Valid Array Multi-variables
logic valid_we0, valid_we1;
logic valid_in0, valid_in1;
logic valid_out0, valid_out1;
ff_array valid_arr0(.clk0(clk),
                    .rst0(rst),
                    .csb0(1'b0),
                    .web0(valid_we0),
                    .addr0(set_index),
                    .din0(valid_in0),
                    .dout0(valid_out0));
ff_array valid_arr1(.clk1(clk),
                    .rst1(rst),
                    .csb1(1'b0),
                    .web1(valid_we1),
                    .addr1(set_index),
                    .din1(valid_in1),
                    .dout1(valid_out1));
```

Generate Statement

- Lot of duplicated arrays between the ways
  - Can copy, paste, copy, paste . . .
  - Or can use generate statement
    - Instantiate one module multiple times
    - Connect the different instantiations to difference indices of logic variable arrays



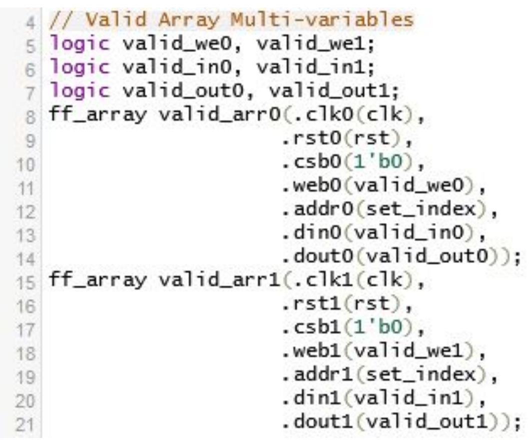
```
// Valid Array Multi-variables
logic valid_we0, valid_we1;
logic valid_in0, valid_in1;
logic valid_out0, valid_out1;
ff_array valid_arr0(.clk0(clk),
                    .rst0(rst),
                    .csb0(1'b0),
                    .web0(valid_we0),
                    .addr0(set_index),
                    .din0(valid_in0),
                    .dout0(valid_out0));
ff_array valid_arr1(.clk1(clk),
                    .rst1(rst),
                    .csb1(1'b0),
                    .web1(valid_we1),
                    .addr1(set_index),
                    .din1(valid_in1),
                    .dout1(valid_out1));
```



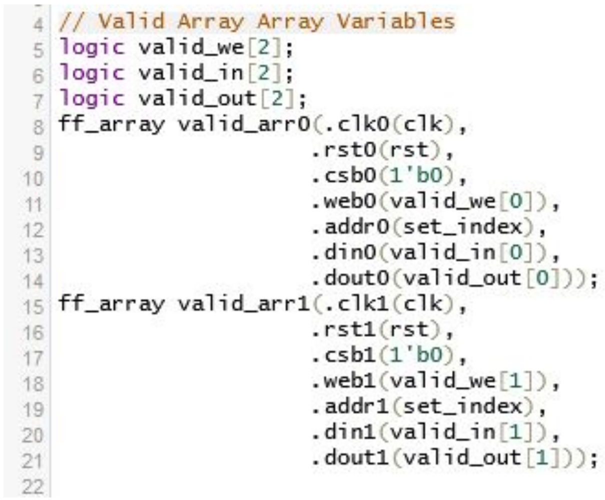
```
// Valid Array Array Variables
logic valid_we[2];
logic valid_in[2];
logic valid_out[2];
ff_array valid_arr0(.clk0(clk),
                    .rst0(rst),
                    .csb0(1'b0),
                    .web0(valid_we[0]),
                    .addr0(set_index),
                    .din0(valid_in[0]),
                    .dout0(valid_out[0]));
ff_array valid_arr1(.clk1(clk),
                    .rst1(rst),
                    .csb1(1'b0),
                    .web1(valid_we[1]),
                    .addr1(set_index),
                    .din1(valid_in[1]),
                    .dout1(valid_out[1]));
```

Generate Statement

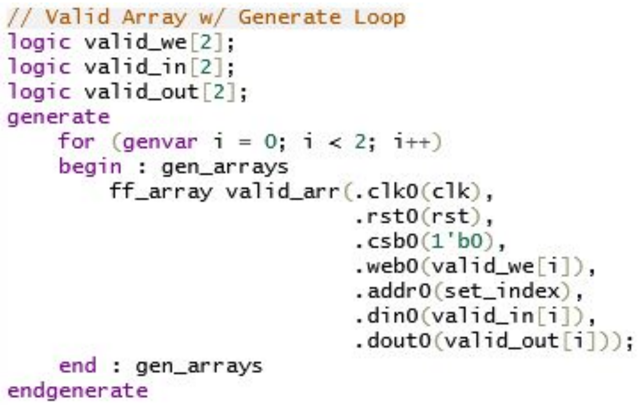
- Lot of duplicated arrays between the ways
  - Can copy, paste, copy, paste . . .
  - Or can use generate statement
    - Instantiate one module multiple times
    - Connect the different instantiations to difference indices of logic variable arrays



```
// Valid Array Multi-variables
logic valid_we0, valid_we1;
logic valid_in0, valid_in1;
logic valid_out0, valid_out1;
ff_array valid_arr0(.clk0(clk),
                    .rst0(rst),
                    .csb0(1'b0),
                    .web0(valid_we0),
                    .addr0(set_index),
                    .din0(valid_in0),
                    .dout0(valid_out0));
ff_array valid_arr1(.clk1(clk),
                    .rst1(rst),
                    .csb1(1'b0),
                    .web1(valid_we1),
                    .addr1(set_index),
                    .din1(valid_in1),
                    .dout1(valid_out1));
```



```
// Valid Array Array Variables
logic valid_we[2];
logic valid_in[2];
logic valid_out[2];
ff_array valid_arr0(.clk0(clk),
                    .rst0(rst),
                    .csb0(1'b0),
                    .web0(valid_we[0]),
                    .addr0(set_index),
                    .din0(valid_in[0]),
                    .dout0(valid_out[0]));
ff_array valid_arr1(.clk1(clk),
                    .rst1(rst),
                    .csb1(1'b0),
                    .web1(valid_we[1]),
                    .addr1(set_index),
                    .din1(valid_in[1]),
                    .dout1(valid_out[1]));
```



```
// Valid Array w/ Generate Loop
logic valid_we[2];
logic valid_in[2];
logic valid_out[2];
generate
    for (genvar i = 0; i < 2; i++)
    begin : gen_arrays
        ff_array valid_arr(.clk0(clk),
                           .rst0(rst),
                           .csb0(1'b0),
                           .web0(valid_we[i]),
                           .addr0(set_index),
                           .din0(valid_in[i]),
                           .dout0(valid_out[i]));

    end : gen_arrays
endgenerate
```

Cache Addressing

Address Translation

- An address is used for indexing into giant arrays
  - Software has a simple view of this
    - uint64_t main_memory [SIZE];
  - If a similar view was kept for hardware -> implies one giant monolithic array w/ a single index
    - Bad because of high capacitance and resistance
      - Very very very slow
- In reality, there are many different smaller arrays indexing by subsets of the full address
  - Faster indexing, less power (only power actively indexed parts)
  - Requires some transformation function to address (may be as simple as bit slicing)

DRAM’s Indexing Scheme

- DRAM’s array consists of
  - Channels
  - Ranks
  - Banks
  - Rows
  - Columns
- Address translation applied
  - Subsets taken
  - XOR for randomness on banks -> bank level parallelism

The “Sub-Arrays” of the Cache

- Like main memory, caches are not one giant monolithic array
  - 4 ways -> Accessed in parallel, so 0 bits (from our address) to “index” our way
    - Indexing happens in hit logic which is separate from address
  - 16 sets -> Access only 1 set, so log_2(16) = 4 bits to choose a set
  - 32 bytes in a 256 bit cacheline -> Access 1 byte, log_2(32) = 5 bits to choose a byte in cachline
- Recall “fine grain” and “coarse grain” access signals of the CPU address and mask
- Two types of arrays
  - SRAM
    - Less transistors
      - Less area
      - Less power
    - Does not have reset
  - Flip flop
    - Has reset
  - What arrays should be what?
    - Main consideration is what needs to be reset?
      - Why is valid in its own FF array and why is dirty shared w/ tag?

Cache Address Scheme

- Where do we start to understand the breakdown of the address translation in caches?
  - Questions to consider
    - Is there a discrepancy between the data sizes of the cache and CPU?
      - Cache: 256 bits
      - CPU: 32 bits, 16 bits, 8 bits
        - Since cache is bigger, this indicates that we need to have our address somehow index into the large 256-bit data from the cache to choose our word, half-word, or byte of interest!
    - Is there a discrepancy between the address size from the CPU and the sub-arrays of the cache?
      - Cache: 16 sets in a single array -> 4-bit address
      - CPU: 32-bit address
        - CPU address is bigger, need a subset of the CPU address to select our set

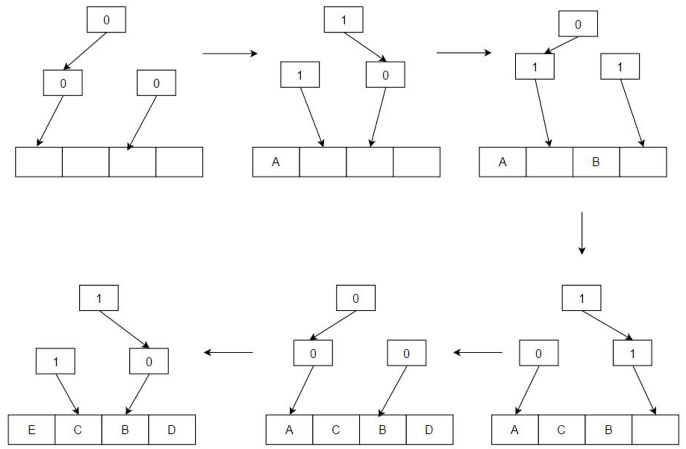
PLRU

Pseudo Least Recently Used

- Want to track what way to replace in event of eviction
  - What makes a good replacement policy?
- Replacing data that hasn’t been recently used may be good
  - Embracing idea of temporal locality in caches
- Don’t deviate from strict PLRU in this MP
  - If PLRU tells you to replace way n, you replace way n
  - Do not consider valid bit in replacement
- There is a great explanation for PLRU on Wikipedia
  - With step by step images too!
  - So I don’t need to explain it here . . .
    - JK

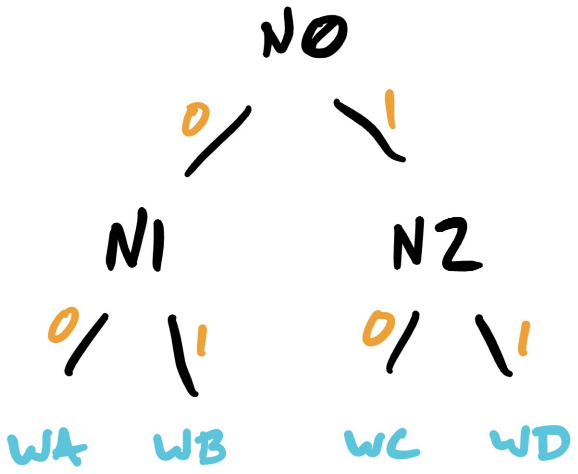
Pseudo Least Recently Used

- Want to track what way to replace in event of eviction
  - What makes a good replacement policy?
- Replacing data that hasn't been recently used may be good
  - Embracing idea of temporal locality in caches
- Don't deviate from strict PLRU in this MP
  - If PLRU tells you to replace way n, you replace way n
  - Do not consider valid bit in replacement
- There is a great explanation for PLRU on Wikipedia
  - With step by step images too!
  - So I don't need to explain it here . . .
    - JK



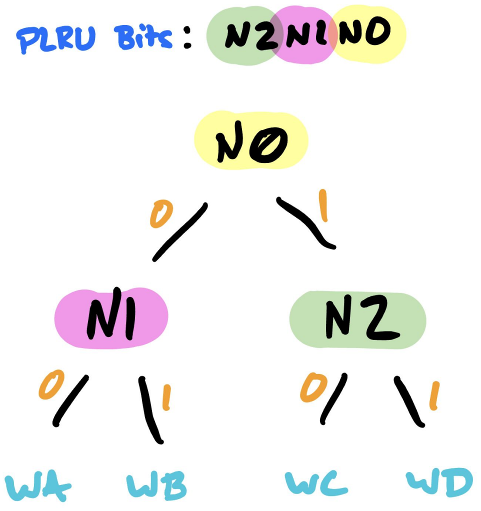
PLRU Walkthrough

- Update PLRU tree bits to point to the most recently used way
  - Track the steps/turns we took to hit that way
-



PLRU Walkthrough

- Update PLRU tree bits to point to the most recently used way
  - Track the steps/turns we took to hit that way
- Eviction selection done by stepping/turning in opposite direction
  - Invert output of PLRU
    - Turn the opposite direction for every node visited
- Create a table for
  - The update applied to the bits when hitting a certain way
  - The target way for replacement when the PLRU bits are a certain value
- Implement that table logic in HDL



Chiral PLRU Trees

- There is a small difference between PLRU approaches
  - Approach here: we remember the most recently used way
  - Wikipedia approach: we remember what way to replace
- Both apply inversions at different steps
  - Approach here: Invert array’s output
  - Wikipedia approach: Invert array’s input
- Both are PLRU!
  - The cache storage will just be mirror images of each other

Verification

What to Verify?

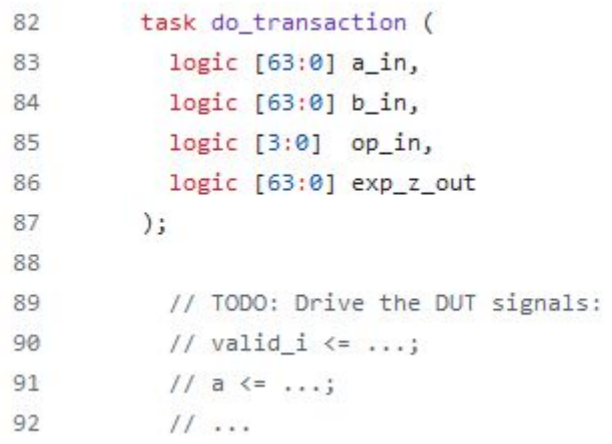
- Data correctness
  - Does the cache correctly return the data requested by the CPU
  - Does the cache correctly modify data stored by the CPU
- Cache timing
  - Do cache hits respond in the correct time
- Cache replacement policy
  - Is the correct data evicted following PLRU
    - This may be intertwined w/ cache timing verification

Additional Verification Suggestions

- Some coverpoints to consider
    - Are all sets touched?
    - Are all ways touched?
    - Are all state visited and do number of visits make sense?
- Create a behavioral model of your cache in HVL
    - Use SystemVerilog's built in array types
- Use SystemVerilog tasks to emulate CPU reads and writes
    - Refer to mp_verif for inspiration

Additional Verification Suggestions

- Some coverpoints to consider
  - Are all sets touched?
  - Are all ways touched?
  - Are all state visited and do number of visits make sense?
- Create a behavioral model of your cache in HVL
  - Use SystemVerilog's built in array types
- Use SystemVerilog tasks to emulate CPU reads and writes
  - Refer to mp_verif for inspiration



```
task do_transaction (
  logic [63:0] a_in,
  logic [63:0] b_in,
  logic [3:0]  op_in,
  logic [63:0] exp_z_out
);

  // TODO: Drive the DUT signals:
  // valid_i <= ...;
  // a <= ...;
  // ...
```

Good Luck!