Memory & Caches

Disclaimer: These slides may not represent the “best way” to design your memory and caches, but they are strong recommendations from former 470 GSIs.
Need something else like this for the D-Cache
Memory

- System memory is **non-synthesizable** (in 470)
- Instantiated in mem.v in ‘testbenches’ folder
- **Can not change system memory in 470 !!!**
- *mem.v changes at neg-edge of the clock:*

```verbatim
always @(negedge clk) begin
    mem2proc_response <= `SD next_mem2proc_response;
    mem2proc_data    <= `SD next_mem2proc_data;
    mem2proc_tag     <= `SD next_mem2proc_tag;
end
```
module mem (  
    input clk, // Memory clock  
    input [63:0] proc2mem_addr, // address for current command  
    input [63:0] proc2mem_data, // address for current command  
    input [1:0] proc2mem_command, // `BUS_NONE `BUS_LOAD or `BUS_STORE  

    output logic [3:0] mem2proc_response, // 0 = can't accept, other=tag of transaction  
    output logic [63:0] mem2proc_data, // data resulting from a load  
    output logic [3:0] mem2proc_tag // 0 = no value, other=tag of transaction  
  );

logic [63:0] next_mem2proc_data;  
logic [3:0] next_mem2proc_response, next_mem2proc_tag;

logic [63:0] unified_memory [`MEM_64BIT_LINES - 1:0];  
logic [63:0] loaded_data [`NUM_MEM_TAGS:1];

logic [`NUM_MEM_TAGS:1] [15:0] cycles_left;  
logic [`NUM_MEM_TAGS:1] waiting_for_bus;

logic acquire_tag;  
logic bus_filled;

wire valid_address = (proc2mem_addr[2:0]!==3'b0) &&
                     (proc2mem_addr<`MEM_SIZE_IN_BYTES);
Memory Related Macros

- `MEM_LATENCY_IN_CYCLES`
  - Memory latency is fixed to 100ns for every group
  - That means this macro will have a different value for each group

- `NUM_MEM_TAGS`
  - No. of slots in the memory/ No. of outstanding requests that the memory can handle
  - We will be testing your processor with the value set to 15
Memory Output

• Response (mem2proc_response)
  – Slot no. in which the memory has accommodated the request
  – Can be between 0 and 15 (inclusive)
  – ‘0’ is a special case and means that request has been rejected
    • Issued max amount of outstanding requests
    • Invalid address
    • No request at all
Memory Output

• Tag (mem2proc_tag)
  – Appears on the bus with the data for a load request
  – Slot no. in which the request had been accommodated
  – Can be between 0 and 15
  – ‘0’ means the data on the bus is invalid (X’s)
  – Non-zero means the data is valid
What is the tag for?

- Memory latency is non-zero
- Memory arbiter
  - Up to three things may be contending for the memory
    - Icache, Dcache and Prefetcher
- A non-blocking cache can be implemented to hide this memory latency
always @(negedge clk) begin
    next_mem2proc_tag = 4'b0;
    next_mem2proc_response = 4'b0;
    next_mem2proc_data = 64'bx;
    bus_filled = 1'b0;
    acquire_tag = ((proc2mem_command==`BUS_LOAD) ||
                    (proc2mem_command==`BUS_STORE)) && valid_address;

    for(int i=1;i<=`NUM_MEM_TAGS;i=i+1) begin
        if(cycles_left[i]>16'd0) begin // If cycles are left
            cycles_left[i] = cycles_left[i]-16'd1; // Decrement cycles left
        end
        else if(acquire_tag && !waiting_for_bus[i] && (cycles_left[i]==0)) begin // Incoming
            next_mem2proc_response = i;
            acquire_tag = 1'b0;
            cycles_left[i] = `MEM_LATENCY_IN_CYCLES;
            if(proc2mem_command==`BUS_LOAD) begin // Load
                waiting_for_bus[i] = 1'b1;
                loaded_data[i] = unified_memory[proc2mem_addr[63:3]];
            end else begin // Store
                unified_memory[proc2mem_addr[63:3]]=proc2mem_data;
            end
        end
    end

    if((cycles_left[i]==16'd0) && waiting_for_bus[i] && !bus_filled) begin // Outgoing
        bus_filled = 1'b1;
        next_mem2proc_tag = i;
        next_mem2proc_data = loaded_data[i];
        waiting_for_bus[i] = 1'b0;
    end
end
Important Memory Tidbits

• Data bus == x’s except during BUS_LOAD cycles

• You are not allowed to modify memory/its interface
Icache Controller
(icache.v)
module icache(
    input  clock,
    input  reset,
    input  [3:0] Imem2proc_response,
    input  [63:0] Imem2proc_data,
    input  [3:0] Imem2proc_tag, // this is a ticket, not a real block tag
    input  [63:0] proc2lcache_addr,
    input  [63:0] cachemem_data,
    input  cachemem_valid,

    output logic  [1:0] proc2Imem_command,
    output logic  [63:0] proc2Imem_addr,

    output logic  [63:0] Icache_data_out, // value is memory[proc2lcache_addr]
    output logic  Icache_valid_out, // when this is high

    output logic  [4:0] current_index,
    output logic  [7:0] current_tag,
    output logic  [4:0] last_index,
    output logic  [7:0] last_tag,
    output logic data_write_enable
);
ICache Controller

- We will go thru each line in detail in the next few slides

assign {current_tag, current_index} = proc2Icache_addr[31:3];

wire changed_addr = (current_index!=last_index) || (current_tag!=last_tag);
wire send_request = miss_outstanding && !changed_addr;

assign lcache_data_out = cachemem_data;
assign lcache_valid_out = cachemem_valid;

assign proc2Imem_addr = {proc2Icache_addr[63:3],3'b0};
assign proc2Imem_command = (miss_outstanding && !changed_addr) ? `BUS_LOAD : `BUS_NONE;

assign data_write_enable = (current_mem_tag==lmem2proc_tag) && (current_mem_tag!=0);

wire update_mem_tag = changed_addr | miss_outstanding | data_write_enable;
wire unanswered_miss = changed_addr ? !lcache_valid_out :
    miss_outstanding && (lmem2proc_response==0);
ICache Controller (cont.)

always_ff @(posedge clock) begin
    if(reset) begin
        last_index  <= `SD -1;  // These are -1 to get ball rolling when
        last_tag   <= `SD -1;  // reset goes low because addr "changes"
        current_mem_tag  <= `SD 0;
        miss_outstanding <= `SD 0;
    end else begin
        last_index  <= `SD current_index;
        last_tag   <= `SD current_tag;
        miss_outstanding <= `SD unanswered_miss;
    end

    if(update_mem_tag)
        current_mem_tag <= `SD lmem2proc_response;

end

end
ICache Controller Piece by Piece

assign \{current\_tag, current\_index\} = proc2Icache\_addr[31:3];
output logic [4:0] last\_index,
output logic [7:0] last\_tag,

- The instruction cache is **direct mapped** with **32 lines**
- Memory consists of 8192 lines
- The **index** is therefore **5 bits** and the **block offset** is **3 bits**
- Every cycle last\_index/tag <= current\_index/tag
  - “current” signals come from IF
  - “last” registers used as write index/tag for ICache
ICache Controller Piece by Piece

```markdown
wire changed_addr = (current_index!=last_index) || (current_tag!=last_tag);
```

- Anytime the address changed in fetch, changed_addr will go high
  - Cycle 12 here, so memory request issued in cycle 13

<table>
<thead>
<tr>
<th>Cycle</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11:</td>
<td>4:lda</td>
<td>4:-</td>
<td>4:-</td>
<td>8:-</td>
<td>4:-</td>
</tr>
<tr>
<td>12:</td>
<td>8:-</td>
<td>4:lda</td>
<td>4:-</td>
<td>8:-</td>
<td>4:-</td>
</tr>
</tbody>
</table>
| 13:   | 8:-   | 8:-| 4:lda| 4:- | 4:-| BUS_LOAD ...
| 14:   | 8:-   | 8:-| 8:-| 4:lda | 4:-|
| 15:   | 8:-   | 8:-| 8:-| 8:- | 4:lda | r3=4096 |
ICache Controller Piece by Piece

```
assign Icache_data_out = cachemem_data;
assign Icache_valid_out = cachemem_valid;
```

• This is just the data and valid cache line bit from the cache, it is ready every cycle and never needs to wait.
• These outputs go to IF, data to IF does not come from ICache directly!
ICache Controller Piece by Piece

```verilog
wire unanswered_miss = changed_addr ? !icache_valid_out :
                  miss_outstanding & (Imem2proc_response==0);
```

- Unanswered miss is saying that I checked the cache and the value came back invalid so I will have to go to memory to get the data, or I sent a request to memory and it hasn’t been accepted yet.
- In this case its using changed_addr to judge when it reads the cache.
- miss_outstanding is just the flopped value of unanswered miss (so I missed in the cache last cycle, or memory didn’t accept request last cycle).

<table>
<thead>
<tr>
<th>Cycle:</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11:</td>
<td>4:lda</td>
<td>4:-</td>
<td>4:-</td>
<td>8:-</td>
<td>4:-</td>
</tr>
<tr>
<td>12:</td>
<td>8:-</td>
<td>4:lda</td>
<td>4:-</td>
<td>8:-</td>
<td>4:-</td>
</tr>
<tr>
<td>13:</td>
<td>8:-</td>
<td>4:lda</td>
<td>4:-</td>
<td>4:-</td>
<td>4:-</td>
</tr>
<tr>
<td>14:</td>
<td>8:-</td>
<td>8:-</td>
<td>8:-</td>
<td>4:lda</td>
<td>4:-</td>
</tr>
<tr>
<td>15:</td>
<td>8:-</td>
<td>8:-</td>
<td>8:-</td>
<td>8:-</td>
<td>4:lda</td>
</tr>
</tbody>
</table>
ICache Controller Piece by Piece

assign proc2Imem_addr = {proc2Icache_addr[63:3], 3'b0};
assign proc2Imem_command = (miss_outstanding && !changed_addr) ? `BUS_LOAD : `BUS_NONE;

- proc2Imem_addr is pretty straightforward, cuts off the block offset bits.
- proc2Imem_command will issue a Bus Load every time you missed in the cache last cycle or a previous request wasn’t accepted.
  - If request is accepted, miss_outstanding will be cleared.
  - Looks at “!changed_addr” because this indicates fetch PC changed, so last request likely no good now.
ICache Controller Piece by Piece

\[
\text{wire} \quad \text{update\_mem\_tag} = \quad \text{changed\_addr} \mid \text{miss\_outstanding} \mid \\
\text{data\_write\_enable};
\]

- This controls the flip flop that will hold the ID number for your transaction (current\_mem\_tag)
- Once you send a `BUS_LOAD the memory will respond with a ID number on the negative edge
- When miss\_outstanding is high, you want to grab the ID number so that you can look for it when the memory broadcasts the value
- When data\_write\_enable is high, you want to clear the ID number so you don’t grab any value again from the bus with the same ID number
- When changed\_addr is high, you want to clear the ID number usually because a branch occurred and you don’t care about the access anymore (not the case when you have a non-blocking cache)
ICache Controller Piece by Piece

47: 28:bne | 28:- | 28:- | 28:- | 28:-
48: 32:- | 28:bne | 28:- | 28:- | 28:-
49: 32:- | 32:- | 28:bne | 28:- | 28:- | BUS_LOAD MEM[32]..
50: 32:- | 32:- | 32:- | 28:bne | 28:-
51: 8:- | 32:- | 32:- | 32:- | 28:bne
52: 8:blbs | 8:- | 32:- | 32:- | 32:-

- Whenever this needs to clear the ID number it just enables the flip flop because during the times that it does Imem2proc_response is 0
  - changed_addr would assert on cycle 51, so ID for request to MEM[32] cleared.
ICache Controller Piece by Piece

//TICKET COMES BACK

assign data_write_enable = (current_mem_tag==Imem2proc_tag) &&
                          (current_mem_tag!=0);

• Write enable to the ICache will go high when the tag that you’re looking for matches the tag that is on the memory bus and is not an invalid tag
• The write index/tag is the index you sent off to the memory, i.e. the current index/tag (last index/tag is also the same by now)
ICache Controller

• Don’t necessarily need to use the changed_addr line
  – Could have IF send “read_valid” signal
• Could use a wr_idx instead of last_idx
  – Gets set when you send off a `BUS_LOAD`
• This controller will wait one cycle between a miss in the cache and sending it to memory
  – Know that if you do it in one cycle that you would need to handle a miss to the cache in half a cycle
• Prefetching will drastically increase performance
• Make sure you can handle reads and writes in the same cycle if you prefetch instructions
**DCache Controller**

- I would have the DCache take priority over the ICache in every case
  - Stall the Fetch stage like P3 if this happens
  - Maybe change priority based on current ROB size
- Similar to the ICache controller except now the controller can store to the cache along with memory
  - Loads are handled the same as the ICache
  - Stores now store to the Cache and the Memory (unless WB D$)
    - If the response is non-zero, assume the store completes
    - But will still take up an ID for the entire memory access time
Non-blocking Cache

• A non-blocking cache has the ability to work on other requests while waiting for the memory to supply any misses
• Miss Status Handling Registers (MSHRs) help in tracking the misses
• Remember the response in order to map data coming from the memory to a load miss
• Increases complexity (and performance)
Memory Flow Example

- `NUM_MEM_TAGS = 3`
- Only loads are considered
- `MEM_LATENCY_IN_CYCLES = 5`
- Memory clocked on negedge
- MSHRs clocked on posedge
## Cycle 1: Ld A

<table>
<thead>
<tr>
<th>Response</th>
<th>Data</th>
<th>Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>0</td>
</tr>
</tbody>
</table>

### Main Memory

<table>
<thead>
<tr>
<th>Slot No.</th>
<th>Instruction</th>
<th>Cycles Left</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Free</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>Free</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>Free</td>
<td>0</td>
</tr>
</tbody>
</table>

### MSHRs

<table>
<thead>
<tr>
<th>Slot No.</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Free</td>
</tr>
<tr>
<td>2</td>
<td>Free</td>
</tr>
<tr>
<td>3</td>
<td>Free</td>
</tr>
</tbody>
</table>
## Cycle 2: Ld B

<table>
<thead>
<tr>
<th>Response</th>
<th>Data</th>
<th>Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>X</td>
<td>0</td>
</tr>
</tbody>
</table>

### Main Memory

<table>
<thead>
<tr>
<th>Slot No.</th>
<th>Instruction</th>
<th>Cycles Left</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Ld A</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>Free</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>Free</td>
<td>0</td>
</tr>
</tbody>
</table>

### MSHRs

<table>
<thead>
<tr>
<th>Slot No.</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Ld A</td>
</tr>
<tr>
<td>2</td>
<td>Free</td>
</tr>
<tr>
<td>3</td>
<td>Free</td>
</tr>
</tbody>
</table>
**Cycle 3: Ld C**

<table>
<thead>
<tr>
<th>Response</th>
<th>Data</th>
<th>Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>X</td>
<td>0</td>
</tr>
</tbody>
</table>

### Main Memory

<table>
<thead>
<tr>
<th>Slot No.</th>
<th>Instruction</th>
<th>Cycles Left</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Ld A</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>Ld B</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>Free</td>
<td>0</td>
</tr>
</tbody>
</table>

### MSHRs

<table>
<thead>
<tr>
<th>Slot No.</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Ld A</td>
</tr>
<tr>
<td>2</td>
<td>Ld B</td>
</tr>
<tr>
<td>3</td>
<td>Free</td>
</tr>
</tbody>
</table>
## Cycle 4: Ld D

<table>
<thead>
<tr>
<th>Response</th>
<th>Data</th>
<th>Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>X</td>
<td>0</td>
</tr>
</tbody>
</table>

### Main Memory

<table>
<thead>
<tr>
<th>Slot No.</th>
<th>Instruction</th>
<th>Cycles Left</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Ld A</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>Ld B</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>Ld C</td>
<td>5</td>
</tr>
</tbody>
</table>

### MSHRs

<table>
<thead>
<tr>
<th>Slot No.</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Ld A</td>
</tr>
<tr>
<td>2</td>
<td>Ld B</td>
</tr>
<tr>
<td>3</td>
<td>Ld C</td>
</tr>
</tbody>
</table>
Cycle 5: Stall, Ld D

<table>
<thead>
<tr>
<th>Response</th>
<th>Data</th>
<th>Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Main Memory</th>
<th>MSHRs</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Slot No.</strong></td>
<td><strong>Instruction</strong></td>
</tr>
<tr>
<td>1</td>
<td>Ld A</td>
</tr>
<tr>
<td>2</td>
<td>Ld B</td>
</tr>
<tr>
<td>3</td>
<td>Ld C</td>
</tr>
</tbody>
</table>
# Cycle 6: Stall, Ld D

<table>
<thead>
<tr>
<th>Response</th>
<th>Data</th>
<th>Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>X</td>
<td>0</td>
</tr>
</tbody>
</table>

## Main Memory

<table>
<thead>
<tr>
<th>Slot No.</th>
<th>Instruction</th>
<th>Cycles Left</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Ld A</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Ld B</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>Ld C</td>
<td>3</td>
</tr>
</tbody>
</table>

## MSHRs

<table>
<thead>
<tr>
<th>Slot No.</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Ld A</td>
</tr>
<tr>
<td>2</td>
<td>Ld B</td>
</tr>
<tr>
<td>3</td>
<td>Ld C</td>
</tr>
</tbody>
</table>
## Cycle 7: Ld D

<table>
<thead>
<tr>
<th>Response</th>
<th>Data</th>
<th>Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Data(A)</td>
<td>1</td>
</tr>
</tbody>
</table>

### Main Memory

<table>
<thead>
<tr>
<th>Slot No.</th>
<th>Instruction</th>
<th>Cycles Left</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Free</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>Ld B</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>Ld C</td>
<td>2</td>
</tr>
</tbody>
</table>

### MSHRs

<table>
<thead>
<tr>
<th>Slot No.</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Free</td>
</tr>
<tr>
<td>2</td>
<td>Ld B</td>
</tr>
<tr>
<td>3</td>
<td>Ld C</td>
</tr>
</tbody>
</table>
## Cycle 8: Ld E

<table>
<thead>
<tr>
<th>Response</th>
<th>Data</th>
<th>Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Data(B)</td>
<td>2</td>
</tr>
</tbody>
</table>

### Main Memory

<table>
<thead>
<tr>
<th>Slot No.</th>
<th>Instruction</th>
<th>Cycles Left</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Ld D</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>Free</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>Ld C</td>
<td>1</td>
</tr>
</tbody>
</table>

### MSHRs

<table>
<thead>
<tr>
<th>Slot No.</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Ld D</td>
</tr>
<tr>
<td>2</td>
<td>Free</td>
</tr>
<tr>
<td>3</td>
<td>Ld C</td>
</tr>
</tbody>
</table>
## Cycle 9: No request

### Response

<table>
<thead>
<tr>
<th>Response</th>
<th>Data</th>
<th>Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Data(C)</td>
<td>3</td>
</tr>
</tbody>
</table>

### Main Memory

<table>
<thead>
<tr>
<th>Slot No.</th>
<th>Instruction</th>
<th>Cycles Left</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Ld D</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>Ld E</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>Free</td>
<td>0</td>
</tr>
</tbody>
</table>

### MSHRs

<table>
<thead>
<tr>
<th>Slot No.</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Ld D</td>
</tr>
<tr>
<td>2</td>
<td>Ld E</td>
</tr>
<tr>
<td>3</td>
<td>Free</td>
</tr>
</tbody>
</table>
# Cycle 10:

<table>
<thead>
<tr>
<th>Response</th>
<th>Data</th>
<th>Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Slot No.</th>
<th>Instruction</th>
<th>Cycles Left</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Ld D</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>Ld E</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>Free</td>
<td>Free</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Slot No.</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Ld D</td>
</tr>
<tr>
<td>2</td>
<td>Ld E</td>
</tr>
<tr>
<td>3</td>
<td>Free</td>
</tr>
</tbody>
</table>
Memory

• What happens with stores?
  – The stores are registered in the memory in the same way.
  – They also need the same no. of cycles as loads
  – If the response is 0, then it means that the store has not been registered (same as load)

  – Do we need to track them in MSHRs?
Prefetching
Prefetching

• On a miss, grab more than just the current block
  – Maybe make some sort of state machine
• **The farther you prefetch the more likely you will interfere with the DCache**
• The logic will get **more complicated** the more you prefetch
• Suppose you prefetch two lines ahead and the original miss was a taken branch
  – The two lines you prefetched are no longer needed
• Watch out for the interleaving of prefetched data and DCache data
• Suppose your access misses but the data that you would prefetch hits in the cache
Prefetching – High Level

• Most of this is targeted toward the Icache
• On a miss, grab more than just the current block
  – Hope that the instruction/data will be requested in the near future, in which case it will already be in the cache (unless evicted)
• Issues
  – Need to track multiple outstanding requests to memory
  – Don’t want to issue requests for lines that are already valid in the cache
  – What to do when IF requests something else in the middle of waiting for the previous miss to come back?
  – May run out of memory bandwidth
  – May not get access to memory (if Dcache requesting)
Prefetching – High Level Cont’d

• Main algorithm (after miss observed)
  – Issue request for missed line, store address and memory response, start prefetch FSM.
  – For as many cycles as we want to prefetch
    • Increment prefetch address to next line, see if that line is valid in the cache
    • If not, store address somewhere to be requested later
  – Many policies on when to stop/change prefetch addr
    • If you hit a valid line, IF requests something else (branch mispredicted), etc...
  – Easier if you have a 2\textsuperscript{nd} cache read port for this purpose. Can get away with prefetcher taking over IF read port.
Prefetching – Tracking Requests

- Keep buffer of requests in cache controller (MSHRs)
  - Allocate entry on cache miss and we wish to prefetch
    - Store address (so we know where to write into cache)
    - Mark entry as wanting to send request
  - Look for entries wanting to send request (pick one)
    - Send request to memory with entry’s stored address
    - Store mem2proc_response back in entry
    - Mark entry as having sent a request
  - When data comes back from memory
    - Compare mem2proc_tag with stored responses from all valid buffer entries
    - Get {tag,index} from stored address for writing into the cache
    - De-allocate entry
Prefetching ideas

• Grab next block on miss
• March through Memory
  – May evict useful instructions with useless ones
• Move prefetch pointer on branch
Non-blocking Caches
Non-blocking Caches – High Level

• If we have multiple independent memory operations, want to be able to service another if one misses in cache.
  – Really only considering the Dcache here

• Different types
  – Hit under miss
  – Miss under miss

• Can re-use some of the ideas from prefetching
  – Buffers for outstanding requests
Non-blocking Caches - Operation

• When you get a miss, allocate a request buffer
  – Don’t need to worry about buffering stores, since a store miss doesn’t really matter (except for wide cache lines)
  – If we run out of buffers, will have to indicate to LSQ
  – Controller will look for valid buffers waiting to send request
    • Send request, store response back in buffer, mark as sent
  – When data comes back from mem
    • Look for matching request buffer, grab address and store data into cache
    • Send necessary info back to LSQ so that it knows op was serviced.
• Can have buffer tagged with some LSQ info, or just send address back to LSQ so it can figure out which op finished
EXTRA: Questionable Prefetching FSM

• This Prefetching FSM is provided because it was used in a previous semester, but it is incomplete (and possibly also incorrect).

• Use at your own risk

• Not covered in the lab
(Partial) Prefetch FSM (1 prefetch)
Prefetch State Machine Notes

• I didn’t add the arrows from each state to **Cache Miss** driven by Icache_rd & ~cache_hit, but know they exist
• Also know which states you can’t move from if the DCache is using the memory bus
• Look at the arrow moving from **Cache Miss** to **Wait for Miss** if prefetch data was in cache
  – Note the potential problem of moving to **Wait for Prefetch** after the miss is processed