EECS 470 Lab 6

Final Project Memory & Caches

Department of Electrical Engineering and Computer Science
College of Engineering
University of Michigan

Thursday, February 22nd, 2024

Overview

Project time is here

Project

- Milestone 1 is due Tuesday, March 5th (2-3 school days!)
  - **At least one module written and debugged**
    - Should have at least one other partially written
  - Deliverables: 1-page progress report and your module plus testbench
    - Submission: Gradescope for report, autograder will be released soon
    - We'll grade manually by adding bugs to your module and running the testbench
    - Testbench should print "`@@@ Passed`" or "`@@@ Failed`".
- Milestone 2 due Thursday, March 28th
  - Run `mult_no_lsq.s` with an instruction cache.
  - Another 1-page progress report, with a top level architectural diagram
  - Past experience suggests it takes 7-10 days to wire your pipeline together and debug *after* writing all individual modules

Project Specifics/Rules

Cache Size Restriction

- 256 bytes (32 x 8 bytes) of data in the instruction cache
- 256 bytes of data in the data cache.
- One victim cache of four 8-byte blocks (32 bytes of data).
    - Does not include whatever metadata you need for each block
    - LRU bits, valid bits, tag bits, etc...
    - Levels the playing field for everyone, and avoids long synthesis times

Number of CDBs can be at most number of ways you are superscalar

- Why? Design Compiler doesn't punish you as much as it should
- You will need to schedule or stall functional units

Memory and Caches

Disclaimer

- What follows is recommendations from current and prior course staff
- Better performance with different choices may be possible
- The goal isn't to try to use everything...
- Instead, think about what is worthwhile to incorporate in your project

Memory

Memory

- System memory is non-synthesizable
- Instantiated in `mem.sv` in `test/` directory
- You cannot change memory in the final project
- Keep in mind that although the address space is 32 bits, we only have 64 KiB of memory (16 bits worth of address space)
- Memory responds at neg-edge of the clock

Memory

Wait, what is a memory "tag"?

- (Different from cache definition of "tag")
- Tag is a transaction number: like a valet service or shipping a package
- You order something online and get a tracking number
  - Tells you the order has been processed
  - Gives you a handle to sort through your mail
- Why not just use the address?
  - Many addresses could be reading one after the other
    i.e. Load 0x1000, Store 0x1000, Load 0x1000
  - Without tags the third instruction would get the first's data!

Memory Interface

Memory Interface

```
module mem (
   input clk, // Memory clock
   input ADDR      proc2mem_addr, // address of current comman
   input MEM_BLOCK proc2mem_data, // data of current command
   input [1:0]     proc2mem_command, // MEM_{NONE,LOAD,STORE}

    // Memory tag for current transaction (0 = can't accept)
   output MEM_TAG   mem2proc_transaction_tag,
    // data for a load
   output MEM_BLOCK mem2proc_data,
    // 0 = no value, other = tag of finished transaction
   output MEM_TAG   mem2proc_data_tag,
);
```

Memory Interface

Memory Internal Signals

```
// This format is needed for Verilog's $readmemh() function
logic [63:0] unified_memory [`MEM_64BIT_LINES-1:0];

MEM_BLOCK   next_mem2proc_data;
MEM_TAG     next_mem2proc_transaction_tag,
            next_mem2proc_data_tag;

wire [31:3] block_addr = proc2mem_addr[31:3];
wire [2:0]  byte_addr = proc2mem_addr[2:0];

logic [63:0] loaded_data     [`NUM_MEM_TAGS:1];
logic [15:0] cycles_left     [`NUM_MEM_TAGS:1];
logic        waiting_for_bus [`NUM_MEM_TAGS:1];
```

Memory Interface

Memory 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
  - We will test default value, but you should test other latencies
- `` `NUM_MEM_TAGS ``
  - No. of outstanding requests that the memory can handle
  - We will be testing your processor with the value set to 15

Memory Interface

Memory Output

- Response (`mem2proc_transaction_tag`)
  - Slot number 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 (command) was made
- Tag (`mem2proc_data_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

Memory Interface

Memory Output

- Why do we need a tag anyway?
  - Memory latency is non-zero
    - Want to pipeline more than one request at a time
    - This is called a non-blocking controller
    - Need to know when a particular request has been fulfilled
  - Memory arbiter
    - Up to three things may be contending for the memory
    - I-cache, D-cache and Prefetcher
    - Need to route requests to the right structure

Memory

Important Tidbits

- You can change what you do with memory
  - e.g. pipeline requests, prefetch addresses, novel caching techniques
- But not how the memory actually works
  - No modifying the memory module
  - No modifying the memory bus to handle more requests or wider requests
- Remember, mem data will be X's except after a `MEM_LOAD`

More data types???

- So we covered structs before and you should be using them already
- There is a "dual" of that - union
- Just like its origin in C, a SystemVerilog union allows a single piece of storage to be represented different ways using different named member types
- "In type theory, a struct is the product type of all its members, whereas a union is the sum type" - my buddy Pranav

Union example

In a simple example, we have a representation of a 64 bit cache block

```
// A memory or cache block
typedef union packed {
    logic [7:0][7:0]  byte_level;
    logic [3:0][15:0] half_level;
    logic [1:0][31:0] word_level;
    logic      [63:0] dbbl_level;
} MEM_BLOCK;
MEM_BLOCK block;
always_comb begin
    block.dbbl_level = 64'hfacefacefaceface; // the full block
    block.word_level[1] = 32'd420; //write to the upper half
    block.byte_level[2] = 8'd42;  //write only one byte
end
```

Another example

Now let's say you want to break down addressing for different caches

```
typedef struct packed {
    logic [17:0] tag;
    logic [10:0] block_num;
    logic [2:0]  block_offset;
} DMAP_ADDR; // breakdown for a direct-mapped cache
typedef struct packed {
    logic [19:0] tag;
    logic [7:0]  set_index;
    logic [2:0]  block_offset;
} SASS_ADDR; // breakdown for a set associative cache
typedef union packed {
    DMAP_ADDR d; // for direct mapped
    SASS_ADDR s; // for set associative
} CACHE_ADDR; // now we can use a common data type!
```

I-Cache Controller Piece by Piece

I-Cache Controller Interface

```
assign {current_tag, current_index} = proc2Icache_addr[15:3];
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 Fetch
  - “last” registers used as write index/tag for I-Cache

I-Cache Controller Piece by Piece

Fetch Memory Load

```
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

```
Cycle: |    IF   |   ID    |   EX    |   MEM   |   WB
10:    |  -:-    |  -:-    |  -:-    |  -:-    |  -:-
11:    |  4:or   |  -:-    |  -:-    |  -:-    |  -:-
12:    |  8:add  |  4:or   |  -:-    |  -:-    |  -:-     LOAD[4]
13:    |  -:-    |  8:add  |  4:or   |  -:-    |  -:-     LOAD[8]
14:    |  -:-    |  -:-    |  8:add  |  4:or   |  -:-
```

Note: this means that the icache takes one more cycle than the basic p3 fetch stage when it starts up

I-Cache Controller Piece by Piece

Hit in cache

```
assign Icache_data_out  = icache_data[current_index].data;
assign Icache_valid_out = icache_data[current_index].valid &&
            (icache_data[current_index].tags == current_tag);
```

- 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 Fetch
- Data to Fetch does not come from memory directly!

I-Cache Controller Piece by Piece

Unanswered miss

```
wire unanswered_miss = changed_addr ? !Icache_valid_out :
        miss_outstanding & (Imem2proc_transaction_tag==0);
```

- Checked the cache and the value came back invalid
  - Now 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
- miss_outstanding is just the stored value of unanswered miss
  - Either I missed in the cache last cycle, or memory didn't accept request

```
Cycle:  |    IF   |   ID    |   EX    |   MEM   |   WB
11:     |  4:or   |  -:-    |  -:-    |  -:-    |  -:-
12:     |  8:add  |  4:or   |  -:-    |  -:-    |  -:-
13:     |  -:-    |  8:add  |  4:or   |  -:-    |  -:-    LOAD
14:     |  -:-    |  -:-    |  8:add  |  4:or   |  -:-
```

I-Cache Controller Piece by Piece

Unanswered miss

```
assign proc2Imem_command = (miss_outstanding && !changed_addr)
                            ? MEM_LOAD : MEM_NONE;
assign proc2Imem_addr    = {proc2Icache_addr[31:3],3'b0};
```

- `proc2Imem_addr` just cuts off the block offset bits
- `proc2Imem_command` will issue a Mem Load
  - If 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
    - If this happened, need to work on new request instead

I-Cache Controller Piece by Piece

Tracking Tags

```
wire update_mem_tag = changed_addr || miss_outstanding
                                    || got_mem_data;
```

- Once you send a `MEM_LOAD` the memory will respond with a ID number on the negative edge
- Need to hold onto this ID for your transaction (`current_mem_tag`)
- When `miss_outstanding` is high, grab the ID number
  - So that you can look for it when the memory broadcasts the value
- When `got_mem_data` is high, you want to clear the ID number
  - So you don't grab a new value with the same ID number
- When `changed_addr` is high, clear the ID number
  - You don't care about the access anymore
  - Usually because a branch occurred

I-Cache Controller Piece by Piece

Tracking Tags

```
Cycle:|   IF    |  ID     |  EX     |  MEM    |  WB
47:   | 28:bne  | 28:-    | 28:-    | 28:-    | 28:-
48:   | 32:     | 28:bne  | 28:-    | 28:-    | 28:-
49:   | 32:-    | 32:-    | 28:bne  | 28:-    | 28:-    LOAD[32]
50:   | 32:-    | 32:-    | 32:-    | 28:bne  | 28:-
51:   | 8:-     | 32:-    | 32:-    | 32:     | 28:bne
52:   | 8:blt   | 8:-     | 32:-    | 32:-    | 32:
```

- Clear ID number when `changed_addr` is high
- It's safe to clear on that cycle because the old request isn't needed
- A new memory request doesn't launch until next cycle
    - `changed_addr` would assert on cycle 51, so ID for request gets cleared

I-Cache Controller Piece by Piece

Tag Comes Back

```
assign got_mem_data = (current_mem_tag==Imem2proc_data_tag)
                        && (current_mem_tag!=0);
```

- `got_mem_data` enables writing to the I-Cache when the tag that is on the memory bus matches the current outstanding request tag
- The write index/tag is the index you sent off to the memory
  - Stored as `current_tag`

I-Cache Controller Piece by Piece

Design Choices

- Don't necessarily need to use `changed_addr`
  - Could have IF send "`read_valid`" signal
- Could use a `wr_idx` instead of `last_idx`
  - Gets set when you send off a `MEM_LOAD`
- Controller waits one cycle after cache miss to send to memory
  - Can probably be done in one cycle
  - But you have to handle the cache lookup in half a cycle
- Prefetching will drastically increase performance
  - Make sure you can handle reads and writes in the same cycle

D-Cache Controller

D-Cache Controller

- Have the D-Cache take priority over the I-Cache in every case
  - Stall the Fetch stage like P3 if this happens
  - Maybe change priority based on current ROB size
- Similar to the I-Cache controller except now the controller can store to the cache along with memory
  - Loads are handled the same as the I-Cache
  - 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

D-Cache Controller

Non-blocking Cache

- Can work on other requests while waiting for memory to supply misses
- Miss Status Handling Registers (MSHRs) help in tracking the misses
  - Basically a table of tag, address, and data values that are waiting
  - A lot in common with a reservation station
- Need to match tag of incoming data to the proper load in the table
  - May be able to simplify since `mem.sv` services requests in-order...
- Increases complexity (but also performance!)

Non-blocking Caches

Non-blocking Caches

- For the D-Cache: have multiple independent memory operations
  - Want to be able to service another if one misses in cache
  - Will likely evict useful instructions for useless ones
- Basic idea: Use MSHRs to keep track of requests
- Hard part is the implementation...
  - Figuring out when a request can go
    - Depends on forwarding/speculative logic from lecture
  - Updating and ordering requests
  - Once you launch a store, it's gone

Stores

Wait, what about stores?

- Stores are registered in the memory in the same way
- Need the same number of cycles as loads
- If the response is 0, it means the store has not launched
- Memory requests are never reordered

  Take a minute to convince yourself this is the case...
- Do we need to track stores in MSHRs?

Prefetching

Prefetching

- Idea: on a miss, grab more than just the current block
- Probably best to stick with prefetching for just I-Cache, not D-Cache
- More complicated issues the more you prefetch...
  - Suppose you prefetch two lines ahead of a taken branch
    - Best case: The two lines you prefetched are no longer needed
    - Worst case: you evict instructions you need from your I-Cache
  - Need to track multiple outstanding requests to memory
  - Don't want to issue requests for lines that are already valid in the cache
  - Watch out for the interleaving of prefetched data and D-Cache data
    - Don't want to slow down the D-Cache
  - May run out of memory bandwidth
  - What to do when Fetch requests something else in the middle of waiting for the previous miss to come back?

Prefetching

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
  - When should you stop?
    - If you hit a valid line?
    - Fetch requests something else? (branch mispredicted)
    - D-Cache needs access to bus?
- Recommend having a second read port on I-Cache for prefetcher to use

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
    - Send request to memory with entry's stored address
    - Store mem2proc_transaction_tag back in entry
    - Mark entry as having sent a request
  - When data comes back from memory
    - Compare mem2proc_data_tag with stored responses from all valid buffer entries
    - Get {tag,index} from stored address for writing into the cache
    - De-allocate entry

Prefetching

Prefetching Ideas

- Conservative strategy: Grab next block on miss
  - Helps quite a bit: half of all instructions are prefetched
- Greedy strategy: march through memory
  - Will likely evict useful instructions for useless ones
- Move prefetch pointer on branch
  - Predict taken? Or not taken? Or both?
  - Branch predictor information could be helpful to decide