Microprocessor Memory Caching 3: The Direct-Mapped & Set-Associative Cache

Objectives

The objective of this lab is to implement a multi-line direct-mapped cache and a set-associative cache for microprocessor developed in CME341. By the end of this lab, students should be able to:

- Understand and implement the operation of a 4-line, 8-words per line direct-mapped cache for the program memory.
- Understand and implement the operation of a 2-way set associative cache for the program memory.
- Understand the advantages of each of the 2 aforementioned cache architectures over the single-line cache.

Preliminaries

In this lab, you will be using concepts learned from the previous labs, named "Microprocessor Pipeline and Memory Caching 1: The Basics" and "Microprocessor Memory Caching 2: The Single-Line Cache". If you have not completed these labs, please do so before proceeding.

Before you begin this lab, be sure to make a copy of all the contents of the microprocessor developed in the previous labs, so you can have a working copy of it to roll back to in case you make major mistakes in the course of the lab.

Procedure

Multi-Line Direct-Mapped Cache

In this section, you will design and implement a multi-line direct-mapped cache according to the description below. Taking a careful look at the single-line cache implemented in the last lab, you will notice that its architecture is also a direct-mapped cache. In this lab, the direct-mapped cache will have 4 lines with 8 words each. Instead of having to load 32 words at a time when there is a cache miss, you will only need to load 8. In addition, using 4 lines instead of 1 increases the chance of a cache hit. The

tagID is the 3 MSBs of the PC. The next 2 bits are the line numbers and the 3 LSBs are the offset index. The following are some hints to assist you in completing the design.

- Copy all the project files of the microprocessor in “single_cache” from the previous lab into a new folder called “multi_cache”.
- The modified cache module has been provided to you in “cache_multi.v”. The cache now has additional inputs for line IDs. Using the MegaWizard function, modify the RAM used for the cache to have 4 words, and 8 x 8 = 64-bits per word.
- Make the following changes in the program_sequencer module.
- Modify the “cache_wroffset” and “cache_rdoffset” to 3 bits instead of 5.
- Add 2 more outputs, “cache_wrline” and “cache_rdline”. They are used for the line number of the cache.
- Modify the logic to detect for “start_hold”, which detects when a cache miss occurs. A cache miss occurs when tagID[pm_address[4:3]] != pm_address[7:5] or when valid[pm_address[4:3]] == 1’b0 and hold == 1’b0. Also assert “start_hold” when reset_1shot is high.
- Modify “hold_count” and “end_hold” to reduce the suspension of the microprocessor from 32 clock cycles to 8 clock cycles.
- Modify “rom_address” to output {pm_address[7;3], 3’d0} when “start_hold” is set, {5’d0, hold_count + 3’d1} when “sync_reset” is set, and {tagID[pc[4:3]], pc[4:3], hold_count + 3’d1} otherwise.
- Connect pc[4:3] to “cache_wrline”, connect pm_address[4:3] to “cache_rdline” and pm_address[2:0] to “cache_rdoffset”. “cache_wroffset” stays connected to “hold_count”.
- Add a set of 3-bit tagID registers to remember the current tag in the cache for each line (so there are 4 tagID registers). The tagID is updated when a cache miss occurs and the cache is filled. So these registers are reset to 0 when “reset_1shot” is high, set to pm_address[7:5] when “start_hold” is high and pm_address[4:3] equals the line that the register represents and remains unchanged otherwise.
- Add a set of 1-bit valid registers to indicate whether or not each line in the cache is currently valid (so there are 4 valid registers). The valid registers are reset to 0 when “reset_1shot” is high, set high when “end_hold” is high and pm_address[4:3] equals the line that the register represents and remains unchanged otherwise.
- Modify the top-level microprocessor to use the given “cache_multi.v” file and connect the appropriate ports to the program_sequencer.
- Compile and simulate the microprocessor using the provided testbench and .hex program memory file.

Questions to Consider

1. Draw a block diagram of the provided “cache_multi.v” module.
2. Explain the operation of the direct-mapped cache. Include examples on when a cache hit and cache miss occurs.
3. Compare the hardware resource utilization between the “single_cache_init” and “multi_cache”. What observations can you make? What is the memory bits utilization of the two implementations?
4. Observe the execution at the end of the program for “single_cache_init” and “multi_cache”, what do you notice? (You need to rerun the simulation for “single_cache_init” with the provided lab 5 .hex file)

2-Way Set Associative Cache

In this section, you will design and implement a 2-way set associative cache according to the description below. In this section, the cache has 2 sets. Each set has 2 entries and each entry has 8 words. The resultant cache has a total of 32 words, which is the same as in the previous labs. The 4 MSBs of the PC indicate the tag ID, the next bit is the set number and the 3 LSBs are the offset index. The selection criteria to write to the 2 entries in a set for cache miss is least recently used (LRU). The following are some hints to assist you in completing the design.

- Copy all the project files of the microprocessor from the “multi_cache” folder created in “Microprocessor Memory Caching 3: The Multi-Line Cache” into a new folder called “set_assoc_cache”.
- The modified cache module has been provided to you in “cache_set_assoc.v”. The cache now has additional inputs for the entry being written and read. Using the MegaWizard function, modify the RAM used for the cache to have 2 words, and 8 x 8 = 64-bits per word. The “cache_set_assoc.v” module instantiates 2 such RAMs.
- Make the following changes to the program_sequencer.
- Modify the “cache_rdline” and “cache_wrline” outputs to a [0:0] bus.
- Add 2 1-bit outputs, “cache_rdentry” and “cache_wrentry”.
- Modify “start_hold” to be asserted when tagID[pm_address[3]][curentry] != pm_address[7:4] or valid[pm_addres[3]][curentry] == 1’b0 && hold == 1’b0 or when “reset_1shot” is high.
- Modify the default condition of “rom_address” to {tagID[pc[3]][~last_used[pc[3]], pc[3], hold_count + 3’d1}.
- Modify “cache_wrline” and “cache_rdline” to use [3:3] instead of [4:3].
- Assign “cache_rdentry” to currdentry and “cache_wrentry” to ~last_used[pm_address[3]].

- There are still 4 tag IDs and 4 valid registers, but they are now associated with 2 sets with 2 entries each. Multi-dimensional arrays are used to facilitate the implementation and indexing. Modify the code block for tagID and valid to the following:

```
reg [3:0] tagID [0:1][0:1];
always @ (posedge clk)
begin
    if(reset_1shot == 1'b1)
    begin
        tagID[0][0] <= 4'd0;
        tagID[0][1] <= 4'd0;
        tagID[1][0] <= 4'd0;
        tagID[1][1] <= 4'd0;
    end
    else if(start_hold == 1'b1)
        tagID[pm_address[3]][~lastused[pm_address[3]]] <=
pm_address[7:4];
    else
    begin
        tagID[0][0] <= tagID[0][0];
        tagID[0][1] <= tagID[0][1];
        tagID[1][0] <= tagID[1][0];
        tagID[1][1] <= tagID[1][1];
    end
end

// only really used for initialization
reg valid [0:1][0:1];
always @ (posedge clk)
begin
    if(reset_1shot == 1'b1)
    begin
        valid[0][0] <= 1'b0;
        valid[0][1] <= 1'b0;
        valid[1][0] <= 1'b0;
        valid[1][1] <= 1'b0;
    end
    else if(end_hold == 1'b1)
        valid[pm_address[3]][~lastused[pm_address[3]]] <= 1'b1;
    else
    begin
        valid[0][0] <= valid[0][0];
        valid[0][1] <= valid[0][1];
        valid[1][0] <= valid[1][0];
        valid[1][1] <= valid[1][1];
    end
end
```

- Add the following code to implement the least recently used (LRU) algorithm to determine the entry to write to when a cache miss occurs.

```
// search for current entry
(* keep *)reg currdentry;
always @ *
```

```
begin
    if(pm_address[3] == 1'b0)
    begin
        if(tagID[0][0] == pm_address[7:4])
            currdentry <= 1'b0;
        else
            currdentry <= 1'b1;
    end
    else
    begin
        if(tagID[1][0] == pm_address[7:4])
            currdentry <= 1'b0;
        else
            currdentry <= 1'b1;
    end
end

// last used
(* noprune *)reg lastused[0:1];
always @ (posedge clk)
begin
    if(reset_1shot == 1'b1)
        lastused[0] <= 1'b1;
    else if(pm_address[3] == 1'b0 && hold == 1'b0 && start_hold == 1'b0)
        lastused[0] <= currdentry;
    else if(pm_address[3] == 1'b0 && end_hold == 1'b1)
        lastused[0] <= ~lastused[0];
    else
        lastused[0] <= lastused[0];
end

always @ (posedge clk)
begin
    if(reset_1shot == 1'b1)
        lastused[1] <= 1'b1;
    else if(pm_address[3] == 1'b1 && hold == 1'b0 && start_hold == 1'b0)
        lastused[1] <= currdentry;
    else if(pm_address[3] == 1'b1 && end_hold == 1'b1)
        lastused[1] <= ~lastused[1];
    else
        lastused[1] <= lastused[1];
end
```

- Modify the top-level module accordingly to connect “cache_rdentry” and “cache_wrentry” properly.
- Compile and simulate the design using the provided testbench and .hex file to verify its functionality.

Questions to Consider

1. Draw a block diagram of the provided “cache.v” module.
2. Describe the operation of the set-associative cache. Include examples of cache hit and cache miss.

3. Compare the hardware resource utilization among the “single_cache_init”, “multi_cache” and “set_assoc_cache”. What observations can you make? What is the memory bits utilization of the three implementations?
4. Observe the execution at the end of the program for “single_cache_init”, “multi_cache” and “set_assoc_cache”, what do you notice? Comment specifically on the cache hits and misses at the end of the program, when the address locations 0x70, 0x80, 0x90 and 0xA0 are accessed.
5. Explain what “tagID”, “valid”, “currdentry” and “lastused” represent.

Deliverables

Please submit a lab report write-up that includes the answers to the questions in each section. Submit screenshots verifying that your implementation is working. Record the durations of the simulation of your original design, and of your modified designs for each part (the simulation end is when it reaches the last instruction). In a zip file, include your source code (just verilog and system verilog files if applicable).

-