Microprocessor Memory Caching 2: The Single-Line Cache

Objectives

The objective of this lab is to implement a single-line cache for microprocessor developed in CME341. By the end of this lab, students should be able to:

- Understand and implement the operation of a single-line cache for the program memory.
- Understand the benefits of using cache memory in a microprocessor.

Preliminaries

In this lab, you will be making modifications to the microprocessor that you have developed in the previous lab, named "Microprocessor Pipeline and Memory Caching 1: The Basics". If you have not completed that lab, 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 lab, 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

Single-Line Cache

In this section, you will implement an instruction cache for the program memory of the microprocessor using some of the knowledge obtained from doing the previous lab. A single-line instruction cache will be implemented. The program memory has 256 words and single cache line size is 32 words. Thus, there are 8 unique tags and the tag number is given by the 3 most significant bits (MSBs) of the program counter (pc). If the next instruction is outside the current tag number that is in the cache, a new line needs to be loaded into the cache memory. In the meantime, the microprocessor is suspended to wait for the program memory to write the new line into the cache. In this portion of the lab, you will be able to understand how an instruction cache operates. You will also need to initialize the cache memory when the reset signal is asserted. If the reset signal is asserted for less time than it takes to reload the cache, the microprocessor will not resume operation until the cache is completely reloaded. If the reset signal is held high for more time than it takes to reload the cache, then the microprocessor will not

resume operation until the reset signal is cleared. But the “hold” signal should only be asserted for 32 clock cycles to write the cache. Use the (* keep *) and (* noprune *) directives to force signals to not be synthesizes away, so you can use them in the simulator for debugging.

1. Copy all the project files of the microprocessor from the “Suspending the Microprocessor” section of the previous lab into a new folder called “single_cache”.
2. Add the “cache.v” module created in the previous lab to the project and instantiate it in the top-level microprocessor module. The port connections to this module will be described later.
3. Make the following modifications in the program sequencer.
4. Create 3 outputs called “cache_wroffset”, “cache_rdoffset”, and “cache_wren” that will be connected to the “wroffset”, “rdoffset” and “wren” ports of the cache module.
5. Connect “cache_wroffset” to “hold_count”.
6. Connect “cache_rdoffset” to “pm_address[4:0]”.
7. Connect “cache_wren” to “hold”.
8. Create a register called “sync_reset_1” that is a delayed version of “sync_reset”.
9. Create a signal called “reset_1shot” that is asserted when “sync_reset” is high and “sync_reset_1” is low.
10. Modify the logic of “start_hold” to also assert when “reset_1shot” is high.
11. Modify “hold_count” to reset on “reset_1shot” instead of “sync_reset”.
12. Modify “hold” register logic to not clear on “sync_reset”. Leave the rest of the logic intact.
13. Remove “pm_address” as an output of the module.
14. Create a new output called “rom_address” that will be used to connect to the address bus of the program memory.
15. “rom_address” is the 8-bit output of a combinational logic that outputs 0 when “reset_1shot” is high, is set to {pm_address[7:5], 5’d0} when “start_hold” is high, is set to {3’d0, hold_count + 5’d1} when “sync_reset” is high, otherwise it is set to {pc[7:5], hold_count + 5’d1}.
16. Make the following modifications to the top level microprocessor module.
17. Connect “rom_address”, “cache_wroffset”, “cache_rdoffset” and “cache_wren” to the appropriate ports as described above.
18. Connect the data output of the program memory to the “data” port of the dual-port RAM.
19. Connect the “q” output of the cache to the mux before the instruction decoder instead of “pm_data”.
20. Compile and simulate the design to verify the functionality of the single-page cache using the “microprocessor_tb.v” file and provided computational_test.hex file.

Questions to consider

1. In general, what are the advantages and disadvantages of using a cache?
2. What disadvantages do you see with the implementation as described above, in terms of the benefits you are supposed to gain by using a cache? (Hint: consider the time the microprocessor is suspended to reload the cache).
3. How can you overcome these disadvantages?
4. Provide snapshots of the simulation to demonstrate that your microprocessor has been suspended appropriately, show new contents are written into the cache, and provide a waveform of the entire simulation.

Cache Memory Benefits

In this section of the lab, you will gain a deeper appreciation of the benefits of using an instruction cache. First you will deteriorate the performance of the program memory, to simulate the scenario that the program memory is slow and requires 4 clock cycles to retrieve every instruction. Subsequently, you will add an instruction cache, the same way you did in the first part of this lab in order to see the benefits of using a cache memory. Follow the following instructions to carry out this section of the lab:

1. Copy all the project files of the original microprocessor into a new folder called “rom_more_clk”.
2. Perform the following modifications to the program sequencer.
3. Add a 2-bit counter, called “pc_count” that is reset to 0, when “sync_reset” is asserted, otherwise it will count up by 1.
4. Add a new output to the module called “run” that is set high when the value of “pc_count” is 2, otherwise it is set low.
5. Modify the “pm_address” logic such that its value is set to 0 when “sync_reset” is high; its value will change according to the normal operation of the microprocessor when the value of “pc_count” is equal to 3; otherwise it is equal to “pc”.
6. Perform the following modifications to the top level microprocessor.
7. Create a wire for the “run” output implemented above.
8. Add a multiplexer before the instruction decoder that selects “pm_data” as the input of the instruction decoder when “run” is high; otherwise, input NOPCF (8’HCF) into the instruction decoder.
9. Compile and simulate the microprocessor with the provided computational_test.hex file and the “microprocessor_tb.v” file. Verify that the instructions in the program memory are only executed every 4 clock cycles. Note the time to completion of the program (this is measured from the falling edge of

"sync_reset" to after the microprocessor executes the instruction at address location "8'h70" the first time).

10. At this point, you have essentially deteriorated the performance of the microprocessor to simulate the situation where the program memory is slower and it takes 4 clock cycles to fetch an instruction. In the next steps, you will be modifying the single-line cache microprocessor you implemented in the first section of this lab, to slow down the program memory access to 4 clock cycles per instruction.
11. Copy all the project files of the microprocessor with single-line cache completed in the first part of this lab into a new folder called "rom_more_clk_cache".
12. Make the following modifications to the program sequencer.
13. Add a 2-bit counter called "pc_count" that is reset to 0 when "reset_1shot" is asserted, increments by 1 when "hold" is asserted and remains unchanged otherwise.
14. Add a wire called "run" that is set high when "pc_count" is equal to 3 and set low otherwise.
15. Modify the "hold_count" counter to be 7 bits long and modify "end_hold" to be asserted when "hold" is high and "hold_count" equals to 127 (this extends the hold time to 128 clock cycles from 32 to allow 32 instructions to be access and written to cache at 4 clock cycles per instruction).
16. Modify the "rom_address" output 0 when "reset_1shot" is set, output {3'd0, hold_count[6:2]} when "sync_reset" is set and {pc[7:5], hold_count[6:2]} otherwise.
17. Modify "cache_wroffset" to be assigned with "hold_count[6:2]".
18. Modify "cache_wren" to be asserted only when both "hold" and "run" are asserted.
19. Compile and simulate the microprocessor with the same .hex file and testbench as before. Verify that the microprocessor is suspended for 128 clock cycles when the cache is being reloaded. Verify that the microprocessor still functions as intended. Once again note the time to completion of the program (this is measured from the falling edge of "sync_reset" to after the microprocessor executes the instruction at address location "8'h70" the first time).

Questions to consider

1. What is the program's time to completion of the microprocessor that takes 4 clock cycles to access the program memory? What is the program's time to completion of the microprocessor after adding the instruction cache? Which one is better? Comment on the difference and the trade-off.

2. What type of program causes the single-line cache version of the microprocessor to perform better than the one without any cache (i.e. what type of structures: if-else, loops, long sequential, etc)? What type of program would cause the opposite to be true? Explain. (Hint: look at the assembly and listing file of the hex file given)
3. Provide snapshots of the simulation to demonstrate that the microprocessor does take 4 clock cycles to access the program memory for both versions of the microprocessor. Also, provide a snapshot of the end of the program showing the end time of the program in each version of the microprocessor.

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).