Microprocessor Pipeline and Memory Caching 1: The Basics

Objectives

The objective of this lab is to introduce the students some basic concepts that will be used for implementing a memory caching architecture in the microprocessor developed in CME341. By the end of this lab, students should be able to:

- Understand and implement a pipelined microprocessor architecture
- Understand and implement the operation of suspending the microprocessor
- Understand the operations of a dual-port RAM

Preliminaries

In this lab, you will be making modifications to the microprocessor that you have developed in the CME341 class. If you are not familiar with the architecture of the microprocessor, please consult the instructor or visit the CME341 course website.

Before you begin this lab, be sure to make a copy of all the contents of the CME341 microprocessor, 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. Verify that your microprocessor is fully functional and mostly debugged before proceeding with the lab.

Procedure

2-Stage Pipelining

In this section of the lab, you will be adding two stages of pipelining to the microprocessor. In order to demonstrate the advantage of adding pipelining to the microprocessor, additional modules are added to the original microprocessor to increase the combinational delay between the program sequencer, program memory and instruction decoder, forcing that delay path to be the critical path of the microprocessor. This would be analogous to having a ROM that is much slower than built-in ROM. Subsequently, two pipelining registers will be placed one between the program memory and the instruction decoder and another one between the program sequencer and the program memory. Due to the pipelining, the microprocessor needs to be flushed when there is a jump operation, so NOP instructions need to be passed to

the instruction decoder instead of the instruction from the program memory. The pipelined architecture will not be used in the future labs.

1. Copy all the project files of the original microprocessor into a new folder called "microprocessor_slow_rom".
2. Use the "pipe_test.hex" file provided as the contents of the program memory.
3. Create a new verilog module called comb_logic.v. Create a circuit that generates a large series of combination logic (something that has a measurable delay). See Figure 1 for implementation.
4. Modify the top level of your microprocessor by creating two instances of the "comb_logic" module and connect them as shown in Figure 1.
5. Compile and simulate the microprocessor to ensure that it is still functioning properly. In addition, note down the number of logic elements, registers, memory bits utilized by the microprocessor, the maximum clock frequency and the critical path as shown by the Timing Analyzer.
6. Copy all the project files of the modified microprocessor into a new folder called "microprocessor_srom_pipe1".
7. Modify the top level of your microprocessor to put a pipeline register between the program memory and the instruction decoder. This register is to be clocked with the positive edge of the "clk". There is no need for a reset. Call this register "data_pipe". Place this register before the "comb_logic" module and modify the input to "comb_logic" accordingly.
8. Modify the top level of your microprocessor by building a combinational logic circuit and placing it between the "comb_logic" and the instruction decoder. The output of this logic circuit is to be 8'HC8 if a signal called "flush_pipeline" is high. Otherwise its output should be "pm_data_out". Modify the input of the instruction decoder accordingly.
9. Modify the top level of your microprocessor by building a combinational logic circuit that generates a signal called "flush_pipeline". The signal "flush_pipeline" is to be high while the instruction in the instruction register is either an unconditional jump or a successful conditional jump.
10. Compile and simulate the microprocessor to verify that the pipelining has occurred. The instruction register (ir) should get the values from the program memory one clock cycle later and the 8'HC8 instruction occurs when an unconditional jump or a successful conditional jump is executed. Once again, note down the number of logic elements, registers, memory bits utilized by the microprocessor, the maximum clock frequency and the critical path as shown by the Timing Analyzer.
11. Copy all the project files of the modified microprocessor into a new folder called "microprocessor_srom_pipe2".
12. Modify the top level of your microprocessor to put a second pipeline register between program sequencer and the program memory. Call this register "address_pipe" and clock it with the positive edge of the "clk". There is no need

for a reset. Place this register before the “comb_logic” module. Modify the input of the “comb_logic” module accordingly.

13. Modify the combinational logic circuit between “comb_logic” and the instruction decoder built in the first stage of pipelining to input a second 8’HC8 instruction into the instruction decoder when a successful jump or an unconditional jump occurs. The output of this logic circuit is to be the input to the instruction register.
14. Compile and simulate the microprocessor to verify that the pipelining has occurred. The instruction register (ir) should get the values from the program memory one extra clock cycle later and the 8’HC8 instruction occurs for 2 clock cycles after an unconditional jump and successful conditional jump. Once again, note down the number of logic elements, registers, memory bits utilized by the microprocessor, the maximum clock frequency and the critical path as shown by the Timing Analyzer.

Questions to consider

1. What are the hardware utilization results (logic elements, registers, memory bits), maximum frequency and critical path of each of the 3 versions of the microprocessor?
2. Comment on the trade-off of the hardware utilization and the maximum frequency.
3. What are the advantages of using pipelining registers? What are the disadvantages?
4. Include snapshots of the simulation waveforms to demonstrate that the pipelining stages did insert extra clock cycles between the program counter (pc inside the program sequencer) and the instruction register (ir inside the instruction decoder).

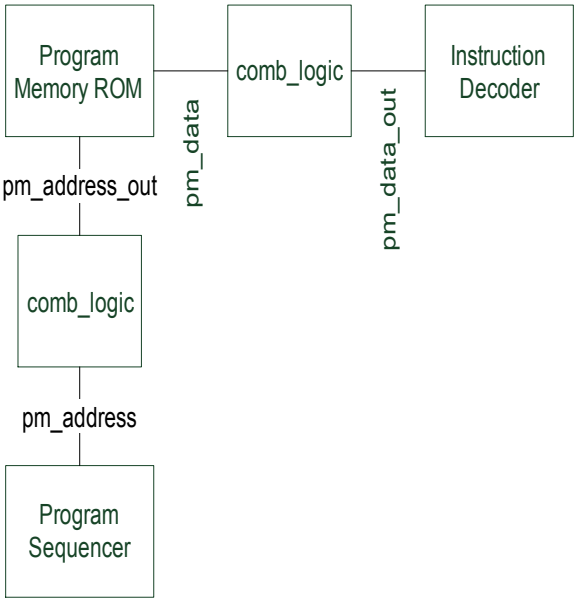

Figure 1: Modified microprocessor

Suspending the Microprocessor

This section implements the circuit needed to suspend the operation of a microprocessor for a certain number of clock cycles. When a certain condition occurs, the program sequencer asserts a hold signal that makes the microprocessor pass NOP instructions into the instruction decoder and the program counter (pc) remains unchanged. After a certain number of clock cycles, the microprocessor operation resumes. This feature will be used in subsequent labs to suspend the microprocessor when the memory cache is being updated.

1. Copy all the project files of the original microprocessor into a new folder called "suspend_micro".
2. Make the following modifications inside the program sequencer.
3. Generate a flag called "start_hold" that is asserted when the 3 most significant bits (MSBs) of the pc is different from the 3 MSBs of the output of the next address logic (pm_address).
4. Create a counter called "hold_count" that starts counting from 0_{10} when "start_hold" is asserted. It will continue counting as long as "hold" signal is asserted. The counter will count up to 31_{10}. Set the counter to 0_{10} on reset.
5. Create a flag called "end_hold" that is asserted when the counter is full and the "hold" register (described next) is high.
6. Create a "hold" register that is set when "start_hold" is high and clears when "end_hold" is high, otherwise it should remain unchanged. Clear the register on reset.

7. Add an output to the program sequencer called “hold_out” that is asserted when “start_hold” or “hold” is high but not when “end_hold” is high (this essentially creates a signal that is exactly like “hold” but 1 clock cycle earlier).
8. Modify the next address logic such that the pc will remain unchanged if “hold” is asserted.
9. Make the following modifications in the top level microprocessor.
10. When the “hold_out” signal from the program sequencer is high, push a NOP instruction into the instruction decoder instead of the next instruction from the program memory.
11. Use the “computational_test.hex” provided as the contents of your program member.

Questions to Consider

1. How do you know if your solution is correct? *Hint: look at the computational test and what it is doing.*
2. Provide snapshots of the simulated waveform that demonstrates the suspension of the microprocessor, and of the whole simulation.
3. The modifications in this section will be re-used in the next lab to create a single-page cache for the microcontroller. Based on the suspension of the microcontroller described above, what would be the size of the cache that you will be implementing in the next lab? Explain your answer.

Dual-Port RAM

(Not graded. This is needed for Lab 4.)

This section introduces the dual-port RAM and its usage. A dual-port RAM is a RAM that has one read port and one write port or two read/write ports. It is generally used when there is a need to read and/or write to the same RAM simultaneously. Intel’s Quartus Prime has a IP Core Function that implements a dual-port RAM. In this section, you will use the IP Core Function to create a dual-port RAM and implement some logic to create a single-line cache that will be used in subsequent labs.

1. Create a new project called “cache”.
2. Start the “IP Catalog” (under “Tools”) and create a new 2-port RAM.
3. Select “With one read port and one write port” and “As a number of words”.
4. Create an 256-bit 2-word RAM (only 1 word will be used in single-line cache).
5. Select “Create byte enable for port A”
6. Do NOT register the output.
7. Select “Don’t care” for “Mixed Port Read-During-Write for Single Input Clock RAM”.
8. Create a top level module called “cache.v” with the following input and output port:

   a. “clk”: 1-bit input clock signal
   b. “data”: 8-bit input data bus
   c. “rdoffset”: 5-bit input word offset for reading data
   d. “wroffset”: 5-bit input word offset for writing data
   e. “wren”: 1-bit input signal that enables the “data” to be written
   f. “q”: 8-bit output data bus
9. Implement the following logic functions and connections:
   a. Create a signal called “byteena_a” that encodes “wroffset” into a 32-bit 1-hot value.
   b. Instantiate the dual-port RAM and connect:
      i. byteena_a -> byteena_a
      ii. clock -> ~clk
      iii. data -> {32{data}}
      iv. rdaddress -> 1’b0
      v. wraddress -> 1’b0
      vi. q -> q_tmp
         - “q_tmp is a 256-bit wire
   c. Assign the output “q” with the respective value depending on “rdoffset”
      i. e.g. rdoffset == 32’d0, then q = q_tmp[7:0]
      ii. NB: use a case statement
10. Simulate the module to verify its functionality.

Questions to Consider

(Not Graded).

1. What cases did you do to test the functionality of the dual-port RAM?
2. How did you verify that the data you wrote into the RAM did go into the RAM?
3. How did you verify that both ports are accessing the same data?
4. What happens when you read and write to the same address location? Double click on the IP Core function again and find the “Documentation” button, select “On the Web” and “Cyclone IV Memory Blocks”. Read the section on “Read-During-Write Operation at the Same Address”. Briefly describe the difference between the two Mixed-Port Read-During-Write Modes (Old Date Mode and Don’t Care Mode).

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).