Lab 2 : Microprocessor Pipeline and Memory Caching 1: The Basics

CME433-01

Addi Amaya

Caa746

11255790

October 23rd, 2021

Contents

2-Stage Pipelining

This section will showcase the different phases that were implemented and will conclude with the questions mentioned in the lab manual.

Microprocessor_srom_pipe1

This section will outline the outcome from steps 1-6 from the lab manual.

The implementation in the microprocessor is as follows:

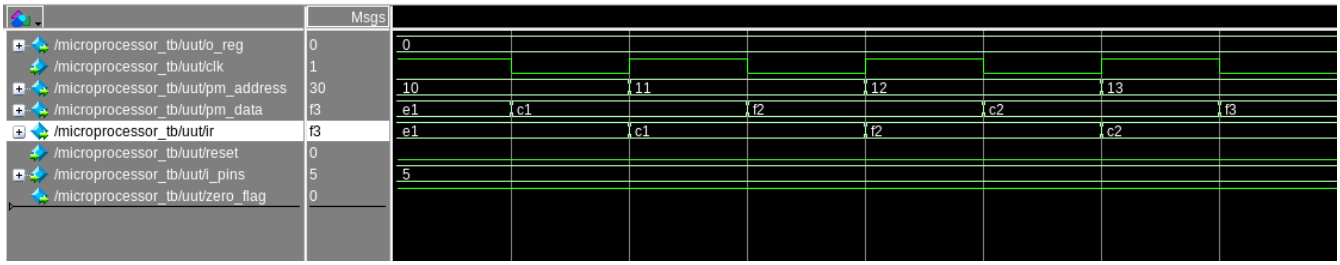
[Program Sequencer] → [comb_logic (cl2)] → [program memory]

[program memory] → [comb_logic(cl1)] → [instruction decoder]

The logic for the combinational logic is a simple loop

```
module comb_logic(input [7:0] info_In, output reg [7:0] info_Out);
    integer counter;

    always@(info_In)
        begin
        counter = 0;

        while (counter < 1000)
            begin
            counter = counter + 8'd1;
            end

        info_Out <= info_In;

        end

endmodule
```

The waveform produced from implementing this function is as follows



Mircoprocessor_srom_pipe2

This section will show the approach and the outcome for steps 7-11

The implementation of the microprocessor is as follows:

[program sequencer] → [comb_logic(cl2)] → [program memory]

[program memory] → [data_pipe (dp)] → [comb_logic(cl1)] → [comb_flush (cf)] →[instruction decoder]

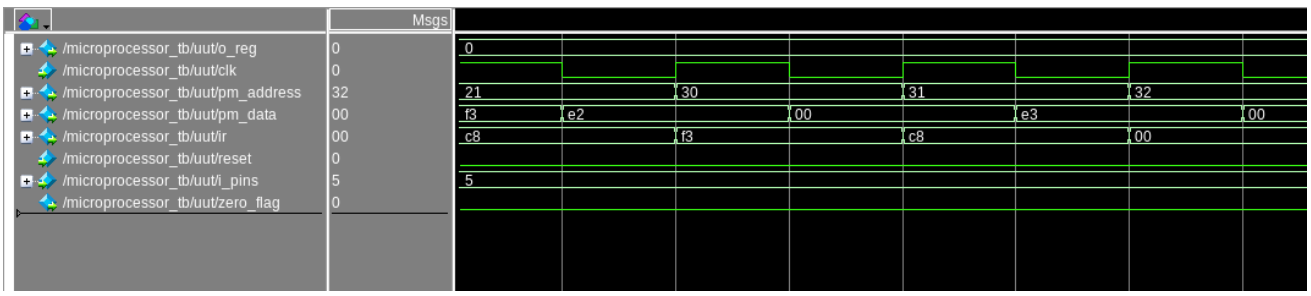
The code for the data_pipe is as follows

```
module data_pipe(
    input [7:0] in,
    output reg [7:0] out,
    input clk
);

    always@(posedge clk)
        out <= in;


endmodule
```

The code for the comb_flush is as follows:

```
module comb_flush(
    input flush,
    input [7:0] in,
    output reg [7:0] out
);

    always@(in)
        begin
            if(flush)
                out <= 8'HC8;
            else
                out <= in;
        end


endmodule
```

The waveform produced from this section of the lab is as follows



Microprocessor_slow_rom

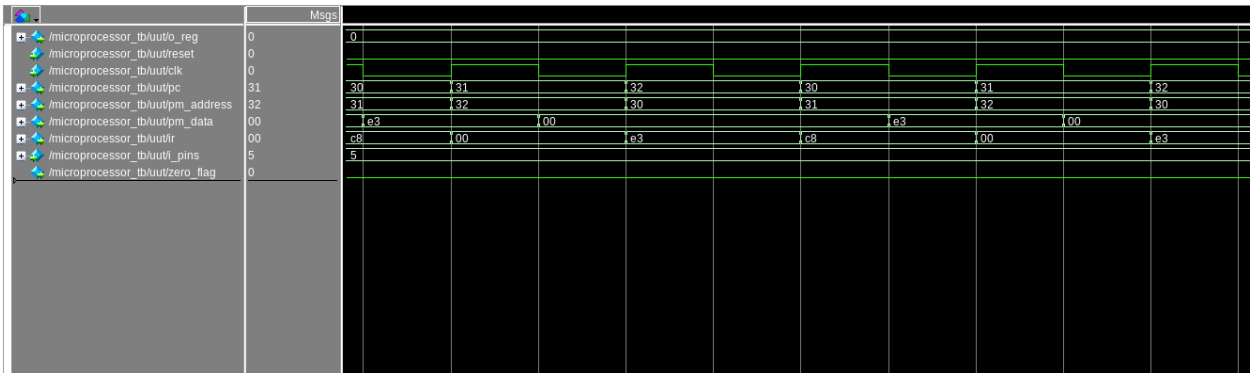
This section will explain the structure and the outcome of the microprocessor for steps 12-14 of the lab manual.

The layout of the microprocessor is as follows:

[program sequencer] → [data_pipe (address_pipe)] → [comb_logic(cl2)] → [program memory]

[program memory] → [data_pipe (dp)] → [comb_logic(cl1)] → [comb_flush (cf)] →[instruction decoder]

The code for the data_pipe (address_pipe) for part of the lab is the exact some as the previously mentioned pipeline module. The wave outcome from this section of the lab was as follows:



Questions

Question 1

For the first version of the microprocessor, it returned

Number of Logic elements: 196

Number of registers: 50

Memory bits: 2112

Max Clock frequency 0C: 55.77MHz

critical path (setup): -8.465

critical path (hold): 0.335

For the second version of the microprocessor, it returned

Number of Logic Elements: 206

Number of registers: 58

Memory bits: 2112

Max Clock Frequency 0C: 59.77Mhz

Critical (Setup): -7.904

Critical (hold): 0.252

For the third version of the microprocessor, it returned

Number of logic elements: 203

Number of registers: 58

memory bits: 2112

Max clock frequency 0C: 60.55Mhz

Crticial (setup): -7.757

Critical (Hold): 0.273

Question 2

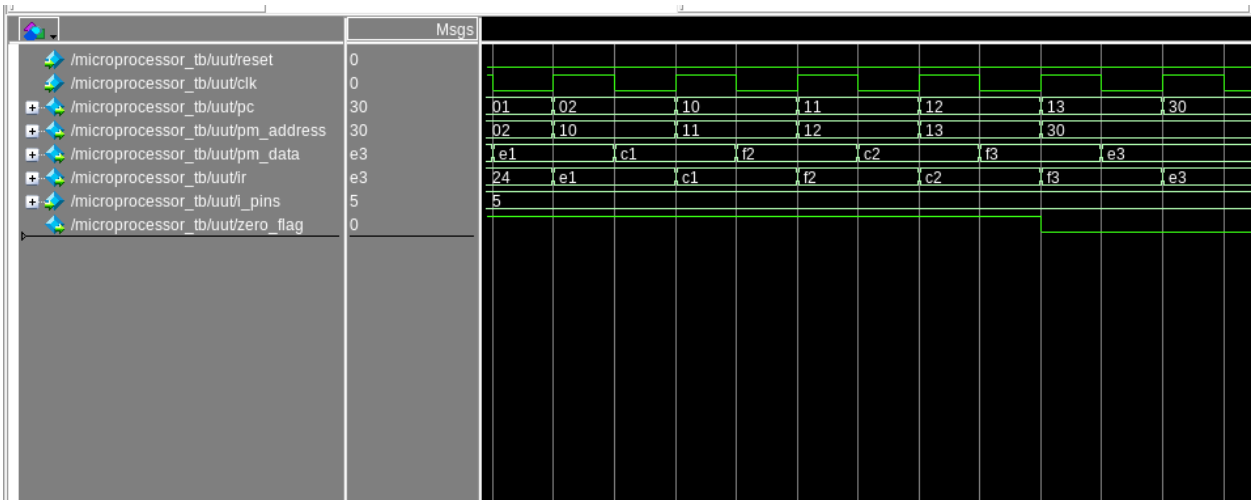
Regarding the maximum frequency, as we progressed in the lab, and we implemented more and more hardware you can see that the speed of increased. Also, it can be seen that the number of registered increased. The number of logic elements increase but went back down on the third version. And, the critical path decreased with every version

Question 3

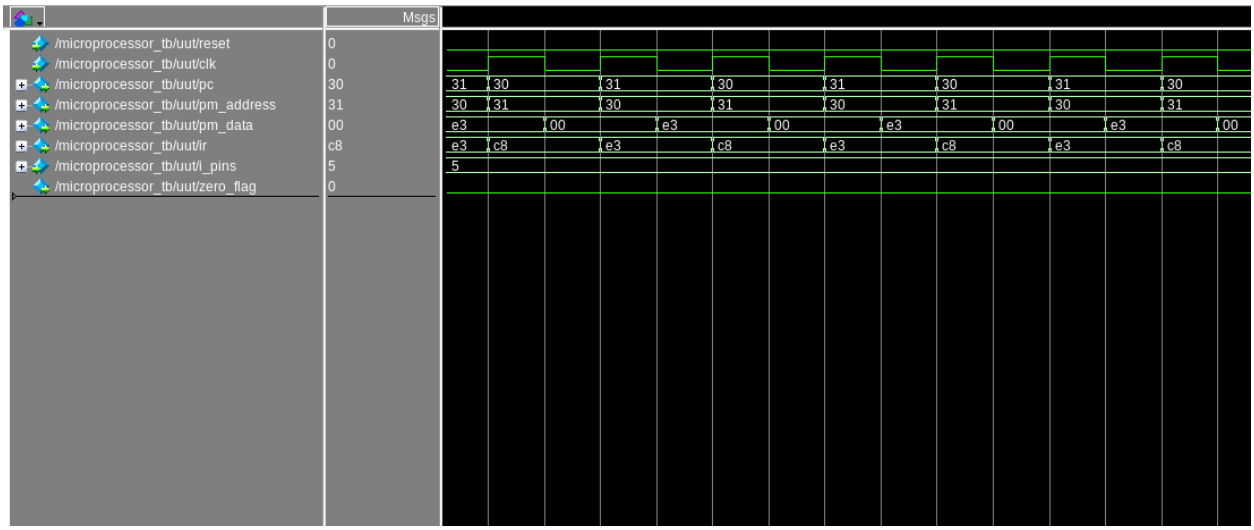
The advantages of the pipelining method is that it will increase the maximum frequency of your system at the cost of number of registers and a bit more complexity in the design.

Question 4

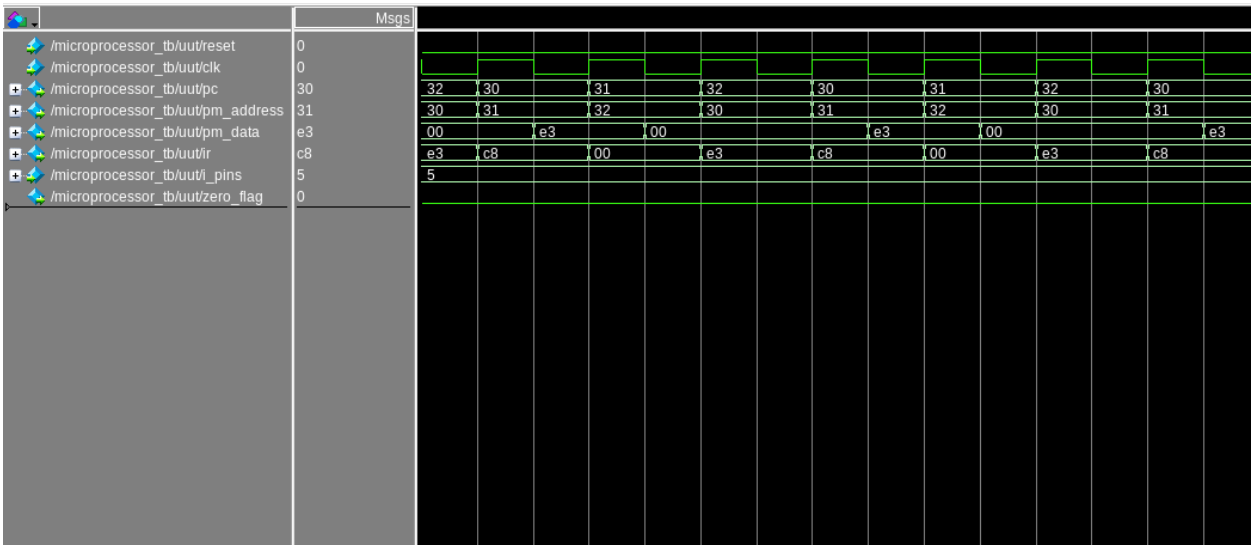
First Version:



Second Version:



Third version:



Suspending the Microprocessor

This section will show the outcome for the adjusted program sequencer and instruction decoder.

In the program sequencer

Start_logic is used when to start the hold

The hold logic is in its own register

The hold_count is in its own register and it clocked every clock cycle creating a 32 clock cycle delay that will be shown later

```
//----------------------------Start_Hold Logic-----------------------------
always@(pc || pm_addr)
   if ((pc[7:5] == 3'b000) && (pm_addr[7:5] == 3'b000)) //when they are the same but all zeros so it would fail the logic check
      start_hold <= 0;
   else if ((pc[7:5] & pm_addr[7:5])) //when they are the same, set low
      start_hold <= 0;
   else
      begin
      start_hold <= 1; //they must be different
      end

//-----------------------------Hold Logic----------------------------------
always@(*)
   if (sync_reset | end_hold)
      hold <= 0;
   else if (start_hold)
      hold <= 1;
   else
      hold <= hold;


//----------------------------Hold_count Logic----------------------------
always@(posedge clk)
   begin
      if (sync_reset)
         hold_count <= 0;
      else if(hold)
         hold_count <= hold_count + 1;
      else
         hold_count <= hold_count;
   end
```

The end_hold logic is in its own register with the appropriate logic

The hold_out logic is an assignment operator its always being written too

In the case that it is hold, the next instruction will always be the previous one, creating that loop until the hold is done.

```
//------------------------------end_hold Logic---------------------------
always@(*)
   begin
      if(hold && (hold_count > 31))
         begin
         end_hold <= 1'b1;
         end
      else
         end_hold <= 1'b0;
   end

//----------------------------Hold_out Logic-----------------------------------
assign hold_out = ((start_hold | hold) & !(end_hold));

assign from_PS = 8'H00;
//assign from_PS = pc;



always @ (*)
   if (sync_reset == 1'b1)
      pm_addr = 8'H00;
   else if (hold) //--------------------Holding
      pm_addr <= pm_addr;
   else if ((jmp == 1'b1) || ((jmp_nz == 1'b1) && (dont_jmp == 1'b0)))
      pm_addr = {jmp_addr, 4'H0};
   else
      pm_addr = pc + 8'H01;
```

In the instruction decoder, there is a section that will load the instruction register to the NOPC8 for as long as the hold_out flag is high

```
//-----------------instruction register-----------------------------------------------
always @(posedge clk)
   begin
      if(hold_out)
         ir <= 8'hC8;
      else
         ir <= next_instr;
   end
```

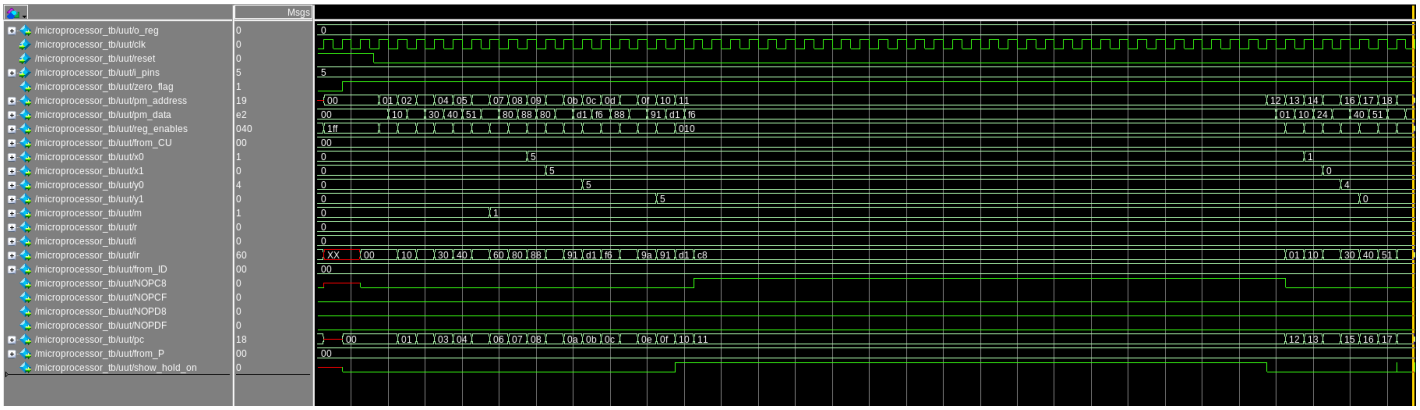
Questions

This section will answer the questions provided by the lab

Question 1

I know that my solution is correct because looking at the wave form it is clear that the program is not doing anything for the time that the hold is on (I made a custom flag “show_hold_on” to see when the flag goes high and low. Another way that I can confirm that my code works is that the program counter doesn’t increment, the pm_data doesn’t change and the pm_address doesn’t change

Question 2



Question 3

Base on the suspension that lasts 32 clock cycles I would guess that the cache would be 5 bits wide because 2^5 gives us 32. Maybe holding for the time it takes to load a bit of information into the ram.

Dual Port RAM

This section will describe the implementation of the ram module created by the IP Catalog.

Encoding the byteena_a with wroffset into a a 32-bit 1-hot value was approached like this. 1-hot means that only the value specified will be written as one and everything else will be written as zeros. My logic was to use a bitwise mask so that every value besides the one mentioned by wroffset would be zero. To do this almost did a type cast of the value within wroffset and masked the byteena_a with that value

```
integer temp;

always @(wroffset)
    temp <= wroffset;


assign byteena_a = byteena_a & (temp);
```

For step 9c, it was assumed that the pattern of the offset continued for the entirety of the 256 bit q_tmp. As such, there was a case statement made to reflect the offset.

```
always@(*)
   begin
      case(rdoffset)
         5'd0:    q = q_tmp[7:0];
         5'd1:    q = q_tmp[15:8];
         5'd2:    q = q_tmp[23:16];
         5'd3:    q = q_tmp[31:24];
         5'd4:    q = q_tmp[39:32];
         5'd5:    q = q_tmp[47:40];
         5'd6:    q = q_tmp[55:48];
         5'd7:    q = q_tmp[63:56];
         5'd8:    q = q_tmp[71:64];
         5'd9:    q = q_tmp[79:72];
         5'd10:   q = q_tmp[87:80];
         5'd11:   q = q_tmp[95:88];
         5'd12:   q = q_tmp[103:96];
         5'd13:   q = q_tmp[111:104];
         5'd14:   q = q_tmp[119:112];
         5'd15:   q = q_tmp[127:120];
         5'd16:   q <= q_tmp[135:128];
         5'd17:   q <= q_tmp[143:136];
         5'd18:   q <= q_tmp[151:144];
         5'd19:   q <= q_tmp[159:152];
         5'd20:   q <= q_tmp[167:160];
         5'd21:   q <= q_tmp[175:168];
         5'd22:   q <= q_tmp[183:176];
         5'd23:   q <= q_tmp[191:184];
         5'd24:   q <= q_tmp[199:192];
         5'd25:   q <= q_tmp[207:200];
         5'd26:   q <= q_tmp[215:208];
         5'd27:   q <= q_tmp[223:216];
         5'd28:   q <= q_tmp[231:224];
         5'd29:   q <= q_tmp[239:232];
         5'd30:   q <= q_tmp[247:240];
         5'd31:   q <= q_tmp[255:248];
         default: q = 8'd0;
      endcase
   end
```

The testbench for the program was a simple increment of data and increment of the wroffset and rdoffset at the same rate. A picture can be seen below.

```
//------------------Initialization------------------
initial #1000 $stop;

initial begin
    clk = 1'b0;
    wren = 1'b1;
    data = 8'd0;
    rdoffset = 5'd0;
    wroffset = 5'd0;
end

//---------------Clock definition-------------------
always #0.5 clk = ~clk;

//---------------Wren definition-------------------
always #900 wren = ~wren;

//----------------data definition-----------------
always@(posedge clk)
    data = data + 8'd1;

//------------------rdoffset-----------------------
always@(posedge clk)
    begin
        rdoffset = rdoffset + 5'd1;
        wroffset = wroffset + 5'd1;
    end
```

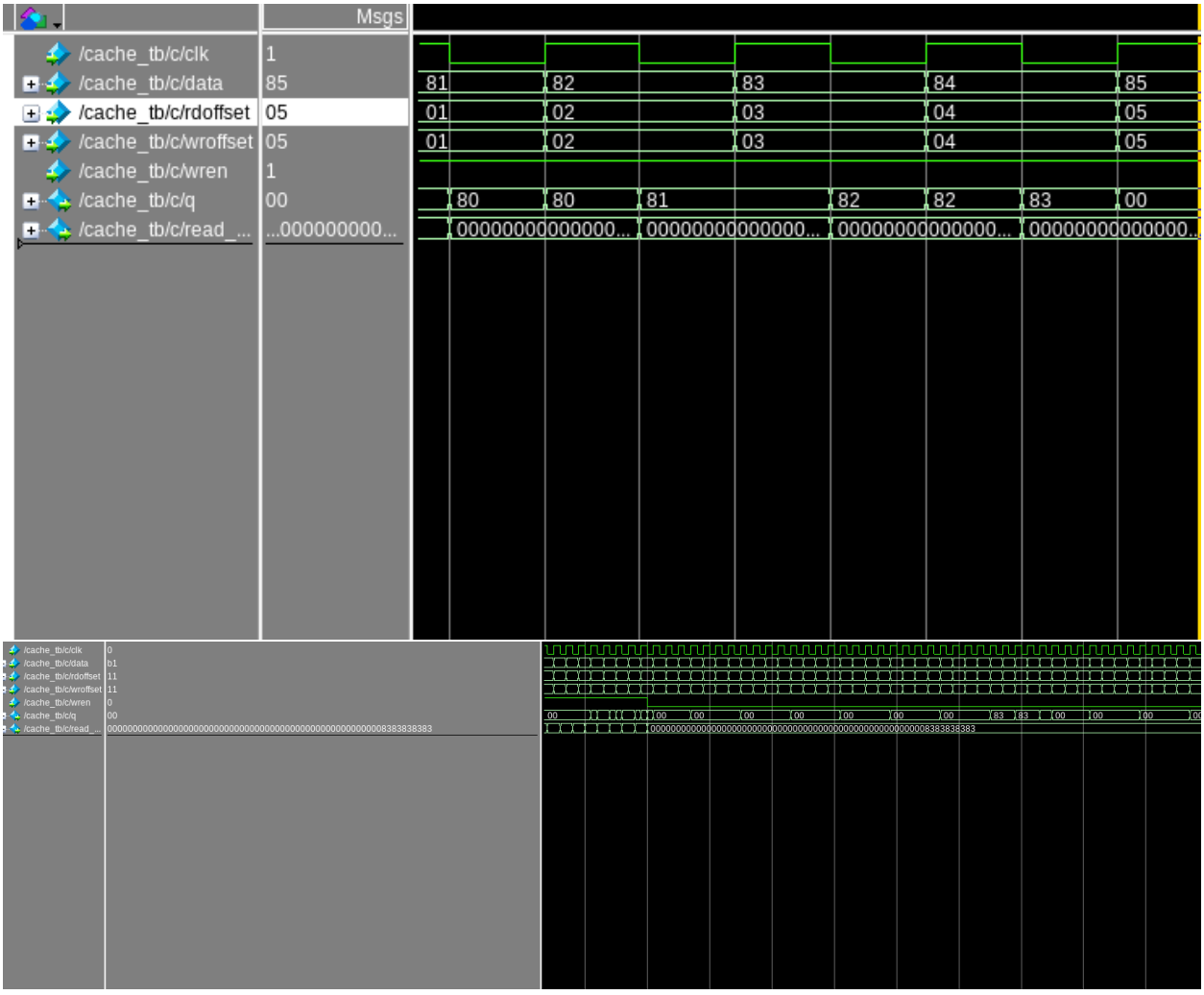
Questions

Question 1

I tested the cases to check to see if data was being read or not and that wren worked with the ram

Question 2

The waveform that was generated can confirm that there was data being written to the ram because it weren't, the the q value would always be zero. Also, it can be seen in the waveform that during the time wren was low that there were no changes to the q_tmp variable (I made a value called "read_q_tmp" to see what was happening in the program.



Question 3

Verified that the port were being accessed by just looking at the waveform and reading the same values for all the ports

Question 4

Old memory: The ram output reflect the old data at the address before the write operation proceeds

Don't Care: The ram output don't care or unknown values for read-during-write operation without analyzing the timing path.