Lab 4: Microprocessor Memory Caching 2: The Single-Line Cache

CME433-01

Addi Amaya

Caa746

11255790

Nov 5th , 2021

Contents

Single-Line Cache 2
Microprocessor Layout 2
My Program Sequencer 2
Questions 4
Question 1 4
Question 2 4
Question 3 4
Question 4 4
Cache Memory Benefits 6
My Program Sequencer 6
Questions 11
Question 1 11
Question 2 11
Question 3 12

Single-Line Cache

Microprocessor Layout

This section will outline how the overall microprocessor is laid out.

[program sequencer] → [program memory] → [cache] → [Instruction Decoder]

There is also additional information that is sent from the program sequencer to the cache such as wroffset, rdoffset, and wren.

My Program Sequencer

This section is here to confirm that all the steps from the lab manual were met before showing the outcome

```
//---------------------------Sync_reset_1 Logic----------------------------
always@(posedge clk)
        sync_reset_1 <= sync_reset;

//---------------------------reset_1shot Logic----------------------------
always@(posedge clk)
        if (sync_reset && !sync_reset_1)
                reset_1shot <= 1'b1;
        else
                reset_1shot <= 1'b0;

//-----------------------------Cache Logic---------------------------------
always@(posedge clk)
        begin
         cache_wroffset <= hold_count;
         cache_rdoffset <= pm_addr[4:0];
         cache_wren <= hold;
        end

//---------------------------Start_Hold Logic-----------------------------
always@(posedge clk)
        if (reset_1shot)
                start_hold <= 1'b1;
        else if (pc[7:5] != pm_addr[7:5])
                start_hold <= 1'b1;
        else
                start_hold <= 1'b0;


//-----------------------------Hold Logic---------------------------------
always@(posedge clk)
        if (end_hold)
                hold <= 0;
        else if (start_hold)
                hold <= 1;
        else
                hold <= hold;
```

```
//---------------------------Hold_count Logic----------------------------
always@(posedge clk)
        begin
                if (reset_1shot)
                        hold_count <= 5'd0;
                else if(hold)
                        hold_count <= hold_count + 5'd1;
                else
                        hold_count <= hold_count;
        end

//-----------------------------end_hold Logic----------------------------
always@(posedge clk)
        begin
                if((hold_count == 5'd31) && (hold == 1'b1))
                        end_hold <= 1'b1;
                else
                        end_hold <= 1'b0;
        end

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

//---------------------------ROM Address Logic--------------------------------
always@(*)
        if(reset_1shot)
                rom_address = 8'd0;
        else if (start_hold)
                rom_address = {pm_addr[7:5], 5'd0};
        else if (sync_reset)
                rom_address = {3'd0, hold_count + 5'd1};
        else
                rom_address = {pc[7:5], hold_count+5'd1};

//---------------------------PM Address Logic---------------------------------
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;
```

Questions

This section will be answering the questions from the lab manual of the first section

Question 1

The advantage of using a cache is that its faster than using the main memory. The disadvantage is that it requires a more complex implementations of hardware.

Question 2

The disadvantage of this implementation is that is seems to be running slightly slower at 56.21Mhz compared to the original which was 59.77MHz. Additionally, while the microprocessor is suspended it takes the cache 32842917ps (3.2*-10^5 s)

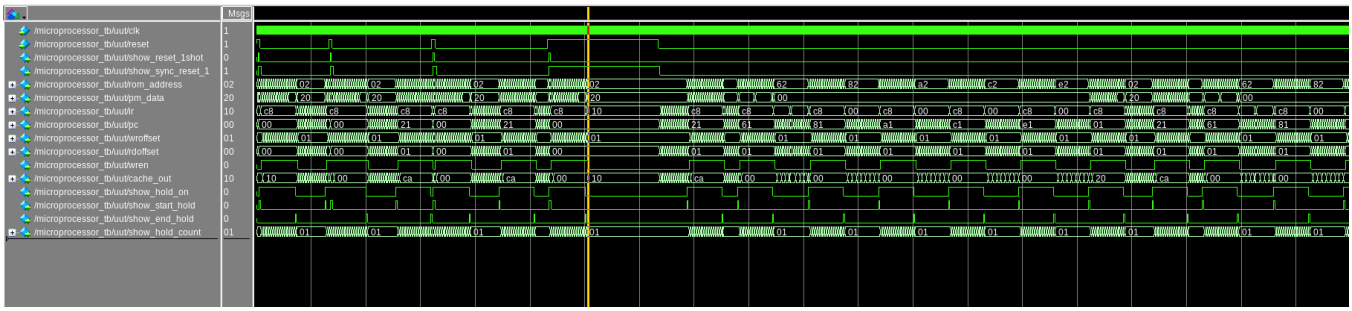
Question 3

A way to overcome the disadvantages of the cache it to pipeline the data into the cache instead of waiting for it to complete.

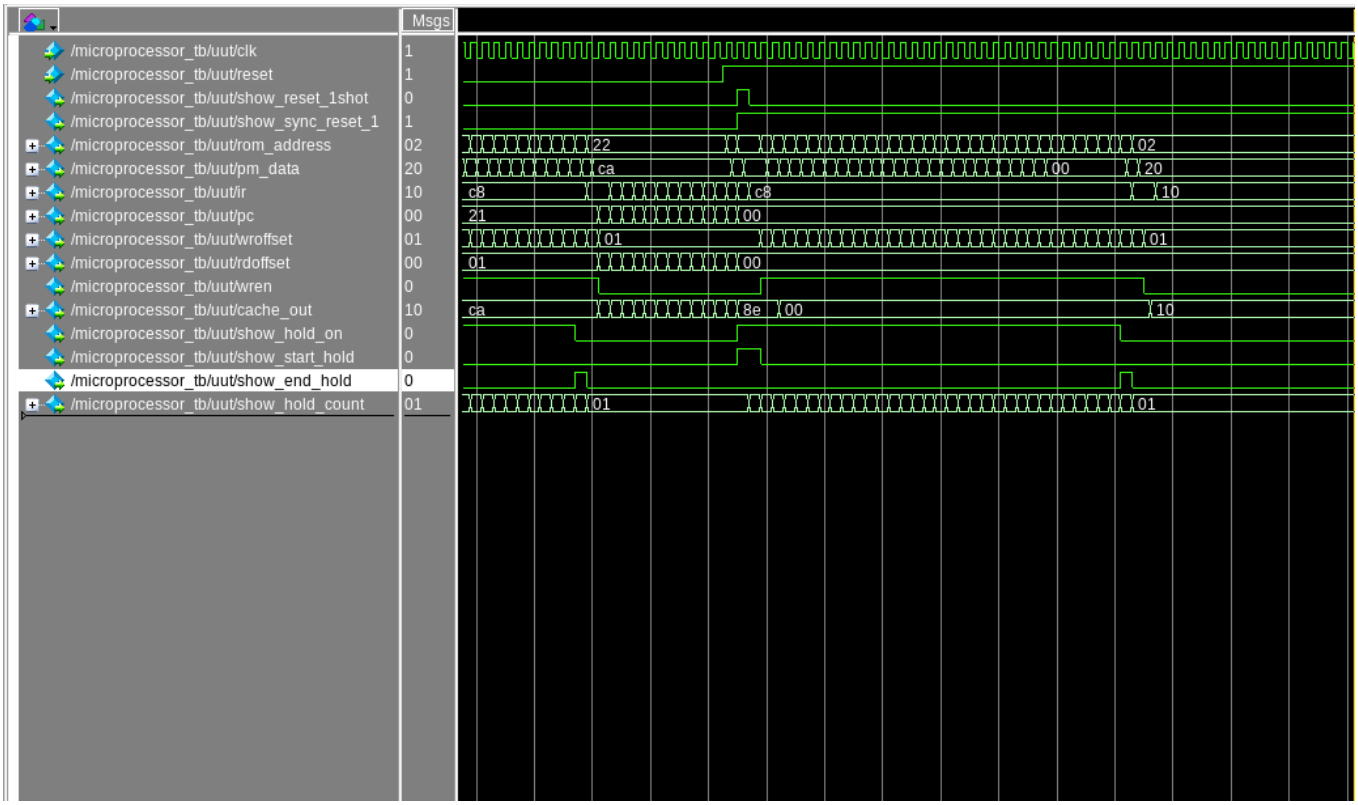
Question 4

This section will outline a series of screenshots all related to the outcome for part 1

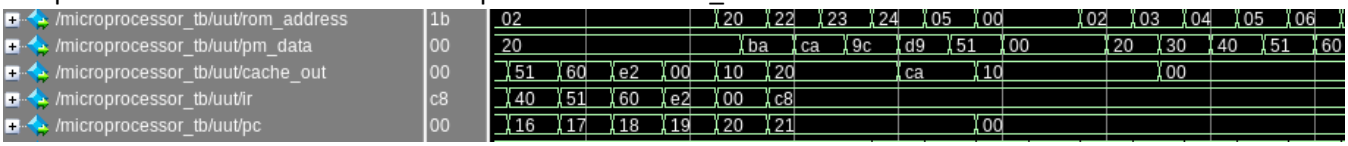
The picture below shows the entire wave form that was produced from the project



The photo below shoes the outcome of the project zoomed in

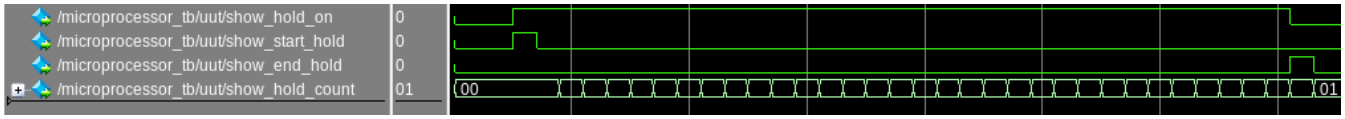


The picture below shows the relationship between the rom_address to the next instruction

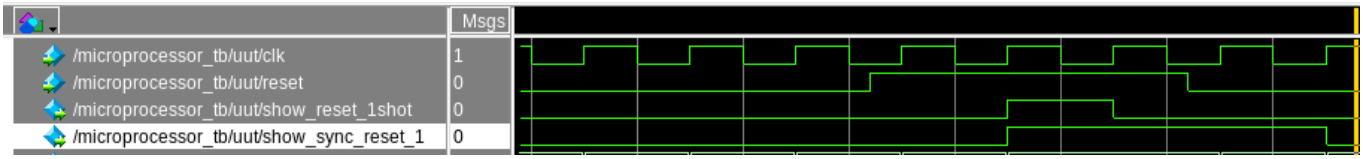


The picture below shows the relationship between the wroffset, rdoffset and wren

The picture below shoes the relationship between all the holds



This picture below shows the relationship between all the resets



Cache Memory Benefits

My Program Sequencer

This section is here to confirm that I did do the lab instructions correctly

The below screenshot shows the run and pc_count logic

```
    output reg [7:0] rom_address,
    output reg [1:0] pc_count,
    output reg run,

    //for debugging
    output reg start_hold,
    output reg end_hold,
    output reg hold,
    output reg [6:0] hold_count,
    output reg sync_reset_1,
    output reg reset_1shot
);

reg [7:0] pm_addr;

//------------------------------Run Logic------------------------------
always@(posedge clk)
    if (pc_count == 2'd3)
        run <= 1'b1;
    else
        run <= 1'b0;

//---------------------------pc_count Logic--------------------------
always@(posedge clk)
    if (reset_1shot)
        pc_count <= 2'd0;
    else if (hold)
        pc_count <= pc_count + 2'd1;
    else
        pc_count <= pc_count;
```

The below screenshot shows the pm_address logic

```
//---------------------PM Address Logic-----------------------------
always @ (*)
    if (sync_reset)
        pm_addr = 8'H00;
    else if (pc_count == 2'd3) //normal operation
        begin
            if ((jmp == 1'b1) || ((jmp_nz == 1'b1) && (dont_jmp == 1'b0)))
                pm_addr = {jmp_addr, 4'H0};
            else
                pm_addr = pc + 8'H01;
        end
    else
        pm_addr = pc; //when pc_count is NOT 3 then pm address is equal to pc


//-------------------------PC Logic-----------------------------------
always @ (posedge clk)
    pc <= pm_addr;
```

The below screenshot shows the implementation of the pm_data (called next_instr here)

```
//------------------instruction register--------
always @(posedge clk)
   if (run)
      ir <= next_instr;
   else
      ir <= 8'HCF;
```

The below screenshot shows the flow summary from steps 1-9

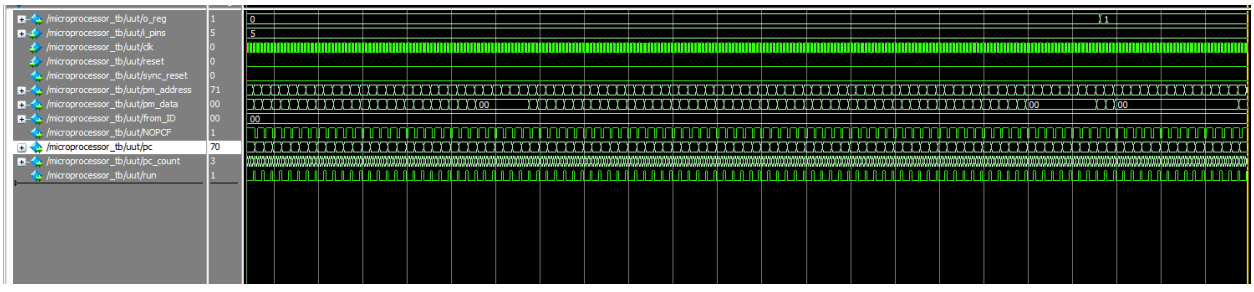
Flow Status	Successful - Wed Nov 03 19:54:00 2021
Quartus Prime Version	20.1.1 Build 720 11/11/2020 SJ Lite Edition
Revision Name	microprocessor
Top-level Entity Name	microprocessor
Family	Cyclone IV E
Device	EP4CE115F29C7
Timing Models	Final
Total logic elements	209 / 114,480 (< 1 %)
Total registers	53
Total pins	112 / 529 (21 %)
Total virtual pins	0
Total memory bits	2,112 / 3,981,312 (< 1 %)
Embedded Multiplier 9-bit elements	0 / 532 (0 %)
Total PLLs	0 / 4 (0 %)

The FMAX was 60.06 MHz

The setup for clk -7.821

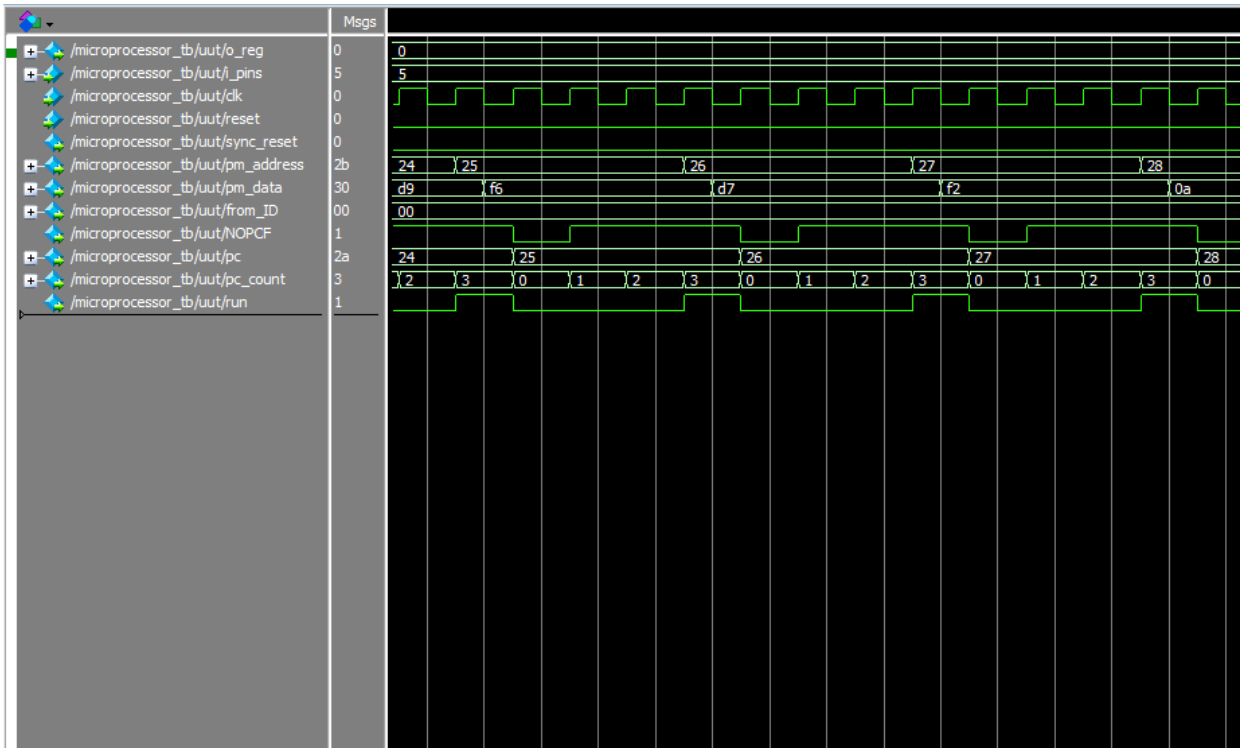
The hold for clk 0.354

The below screenshot shows the waveform from the negative edge of sync_reset to the start of pc when it gets to 8’H70



The time between these two points were 452018627 ps. This was done with two cursors on the waveform

The below screenshot just shows a zoomed in version of the waveform to see the operation



The screenshot below shows the run and pc_count logic for step 13/14

```
    output reg run,

    //for debugging
    output reg start_hold,
    output reg end_hold,
    output reg hold,
    output reg [6:0] hold_count,
    output reg sync_reset_1,
    output reg reset_1shot
);

reg [7:0] pm_addr;

//-----------------------------Run Logic-----------------------------------
always@(posedge clk)
    if (pc_count == 2'd3)
        run <= 1'b1;
    else
        run <= 1'b0;

//---------------------------pc_count Logic-------------------------------
always@(posedge clk)
    if (reset_1shot)
        pc_count <= 2'd0;
    else if (hold)
        pc_count <= pc_count + 2'd1;
    else
        pc_count <= pc_count;
```

The below screenshot shows the implementation for end_hold for step 15 and shows the adjustment to rom_address as told by step 16

```
//--------------------------------end_hold Logic-----------------------------
always@(posedge clk)
   begin
      if((hold_count == 7'd127) && (hold == 1'b1))
         end_hold <= 1'b1;
      else
         end_hold <= 1'b0;
   end

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

//-----------------------------ROM Address Logic--------------------------------
always@(*)
   if(reset_1shot)
      rom_address = 8'd0;
   else if (sync_reset)
      rom_address = {3'd0, hold_count[6:2]};
   else
      rom_address = {pc[7:5], hold_count[6:2]};
```

The below screenshot shows my code for the cache logic from step 17 and 18

```
//-----------------------------Cache Logic--------------------------------
always@(posedge clk)
   begin
      cache_wroffset <= hold_count[6:2];
      cache_rdoffset <= pm_addr[4:0];
      if (hold & run)
         cache_wren <= 1'b1;
      else
         cache_wren <= 1'b0;

   end
```

The screenshot below shows the flow summary for steps 11-19

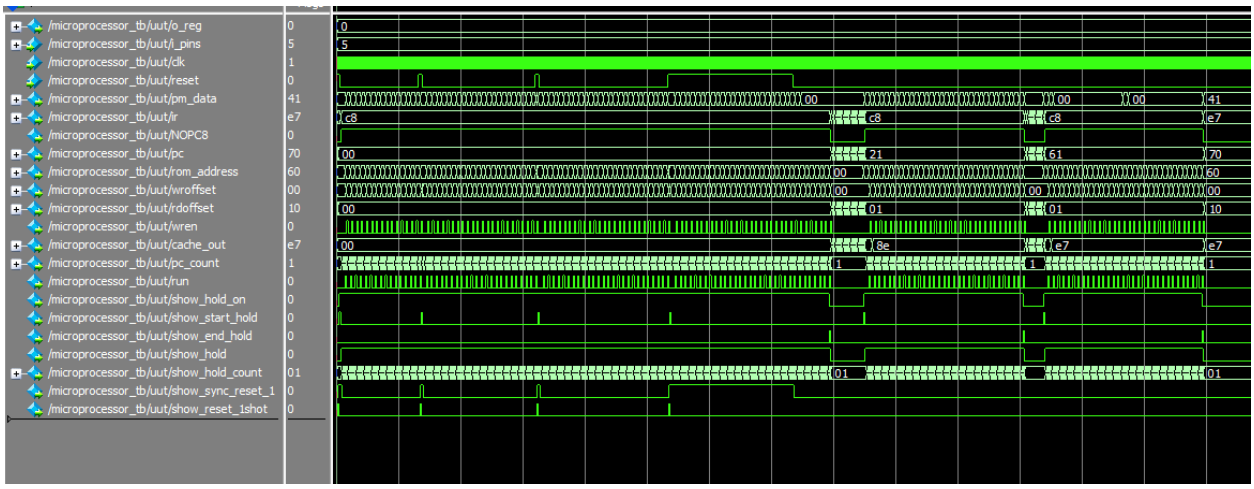
Flow Status	Successful - Wed Nov 03 20:15:43 2021
Quartus Prime Version	20.1.1 Build 720 11/11/2020 SJ Lite Edition
Revision Name	microprocessor
Top-level Entity Name	microprocessor
Family	Cyclone IV E
Device	EP4CE115F29C7
Timing Models	Final
Total logic elements	446 / 114,480 (< 1 %)
Total registers	71
Total pins	143 / 529 (27 %)
Total virtual pins	0
Total memory bits	2,624 / 3,981,312 (< 1 %)
Embedded Multiplier 9-bit elements	0 / 532 (0 %)
Total PLLs	0 / 4 (0 %)

The FMAX 58.3MHz

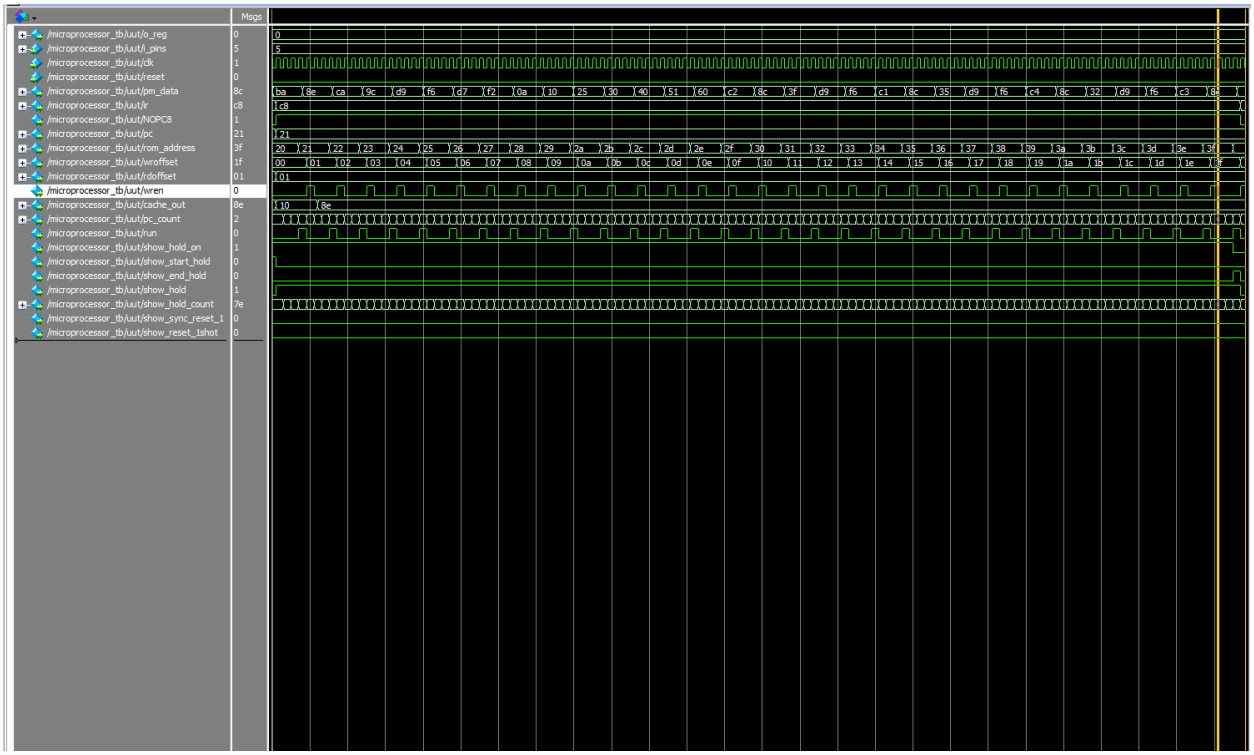
The setup for clk -8.077

The hold for clk 0.246

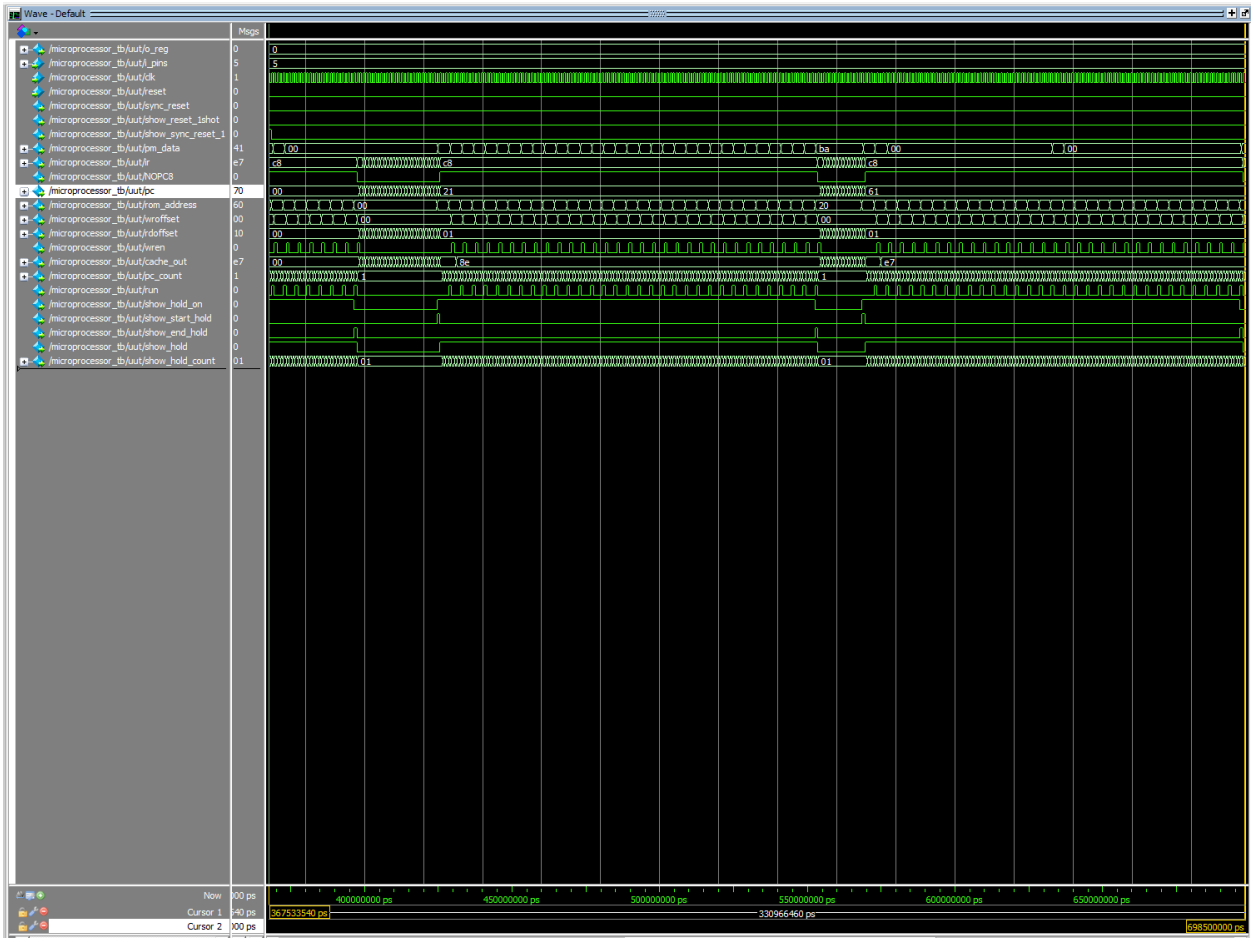
The screenshow below shows the entire waveform produced from steps 11-19



The waveform below shows that my code does hold for 128 clock cycles as you can see hold_count counts up to 127 (or 7F in the waveform) confirming the suspension for 128 cycles because hold_count increments on every clock cycle.



The screenshot below shows from the negative edge of sync_reset to when pc is equal to 8’H70



The outcome for this shows that the time elapsed was 330966460 ps

Questions

Question 1

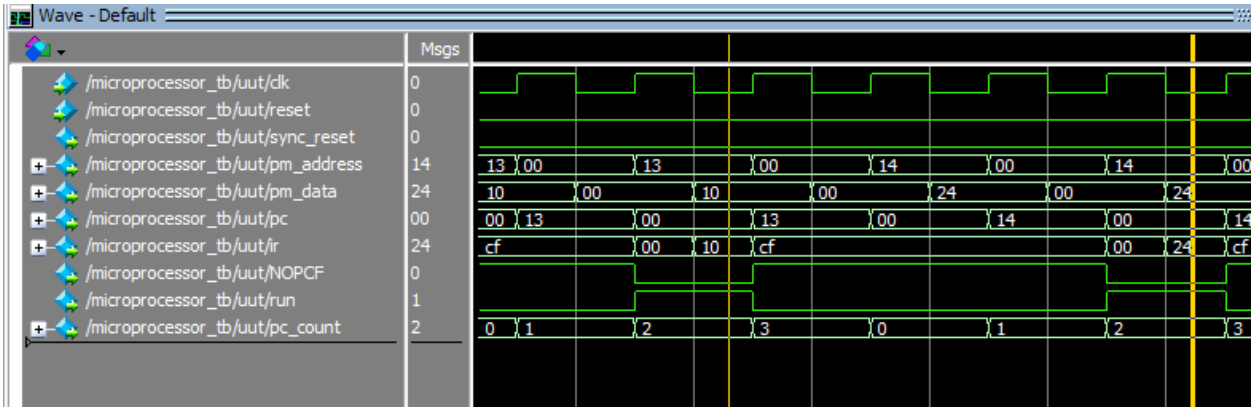
The program’s time to complete that takes 4 clock cycles to access the program memory was 452018627 ps. The program’s time to complete after adding the instruction cache was 330966460 ps. Because the cache implementation is shorter, you can say the cache version was better. The cache implementation did take more hardware, as seen in the flow summary the cache took 446 whereas the 4 clock cycle took 209. It also took more resisters. Also, the cache implementation seem to run slowers (58Mhz compared to the 60Mhz)

Question 2

Programs with short instructions but more of them would work better in a single-line cache than a cache but the moment you introduce some complexing such a function call within a for-loop the single-line cache will struggle and would require better programming or a more complex cache. The reason being is that the single-line cache can process individual numbers quickly whereas a real cache can hold more variety of different data types

Question 3

The below screenshot shows that it takes 4 clock cycles for the pm_addr to get to the ir (next_intr) for the single-cache implementation



The below screenshot shows that it takes 4 clock cycles for the pc to reach the rom_address for the instruction cache implementation