Cover Page
CPE 324-02: Intro to Embedded Computer System

Lab 4

Extended Arithmetic Logic Unit

Submitted by: Nolan Anderson

Date of Experiment: 03/07/2021

Report Deadline: 03/08/2021

1. Introduction:

1.1 - What is to be studied, what is the purpose, and how is this purpose accomplished?

The student performing this laboratory report will expand their knowledge of synchronous design, adding synchronous resets, elements of clock synthesis with phase locked loops (PLLs), pseudo-random number generation with linear feedback registers (LFSRs), using internal logic analyzer tools, and techniques for overcoming timing failures. To attain this knowledge, we will work through the different phases of the project. The first phase will describe how to setup the PLL, see section 2.1.1 for more information on PLLs. Phase 2 will simply modify the output values to show to correct results, so I will not write any theory on that section. Phase 3 will implement the LFSR and will be covered in section 2.1.2. Phase 4 will describe how to setup signal tap and will be briefly described in section 2.1.3. Finally, phase 5 describes how to fix the timing issues with the multiplier and will be covered in section 2.1.4. This lab is a continuation from lab 3 with the modifications consisting of 8 bits – 16 bits and modifying the system clock to run at 200MHz.

2. Experiment Description

2.1 Theory, analysis, and purpose:

2.1.1 Phase Locked Loops (PLLs)

A typical PLL comprises of a voltage-controlled oscillator, a phase detector, and a low-pass filters, and operates in a closed loop. We will use this PLL to generate the desired system clock frequency of 200 MHz. We need a higher frequency to account for the larger 16bit values for this lab. Figure 1 displays a typical PLL.

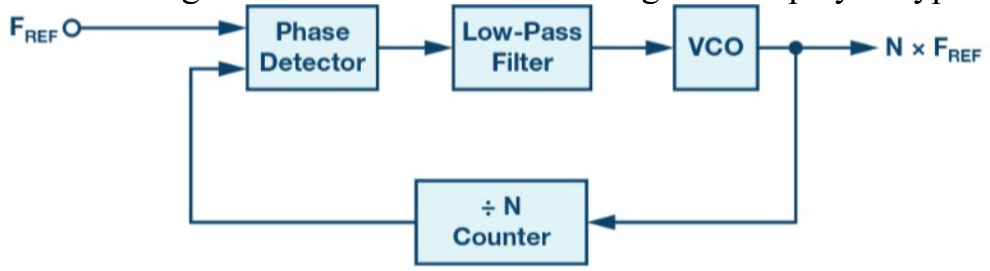


Figure 1: Phase Locked Loop

The frequency at which the ALU can operate depends upon the amount of combinational logic of the most complex OPCODE, plus any multiplexing logic for selecting that particular result.

2.1.2 Linear Feedback Registers (LFSRs) and PRBS.

The lab implements a pair of 16-bit LFSRs with the polynomial being x16+x13+x12+x11+1. Figure 2 shows the LFSR circuit. We will use this in combination with a random number generator, pseudo-random binary sequence. We need this because if we reach a value of 16'h0000 we will be stuck at this value forever. Our LFSR allows us to interface with switches [7:0] and the PRBS will allow us to always modify our values without error. Section 2.2.1 describes how the PRBS works. Lastly, we modify our LFSR value to change on positive edge of the clock. If reset is high, then we go to all FFFF. If Our value is 0000, then we will also go to FFFF. If we have a value inserted in (our randomly generated value) then we will XOR our polynomial and shift the values over by one, leaving the last bit a 0 for the reset of our operation.

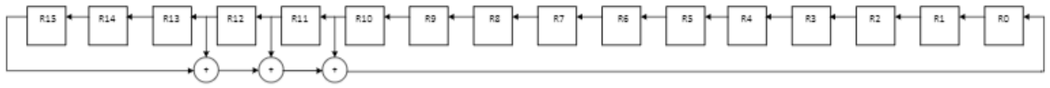


Figure 2: LFSR Circuit

2.1.3 Signal Tap

This section will be brief due to the nature of the implementation. We will setup the signal tap analyzer to display all values that are happening in the circuit including outSeq A and B, our OpCode, and 32 bit result. Signal Tap allows us to look at the values before and after the opcode changes and gives a good summary of the waveforms on the board. Figure 3 shows a simple and random example.

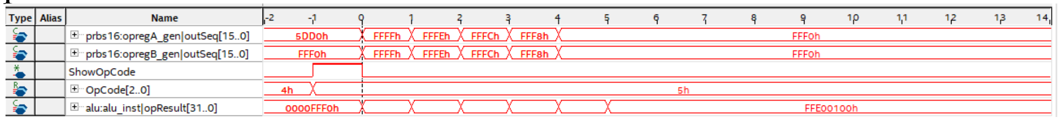


Figure 3: Signal Tap example

2.1.4 Cut-set retiming

Phase 5 of this lab seeks to solve timing issues with the provided ALU code and uses cut-set retiming to solve it. Section 2.2.2 displays the modifications I made to the code, so I will just stick to theory here. Cut-set retiming moves registers around to cut out the critical path. We essentially will store all of the opcode results into an array of registers and then select our desired output after this has been performed. This emphasizes front end calculation so that the timing of our result will not cause any timing issues. Essentially we get the result after all of the values have been calculated. Figure 4 shows the block diagram of this happening.

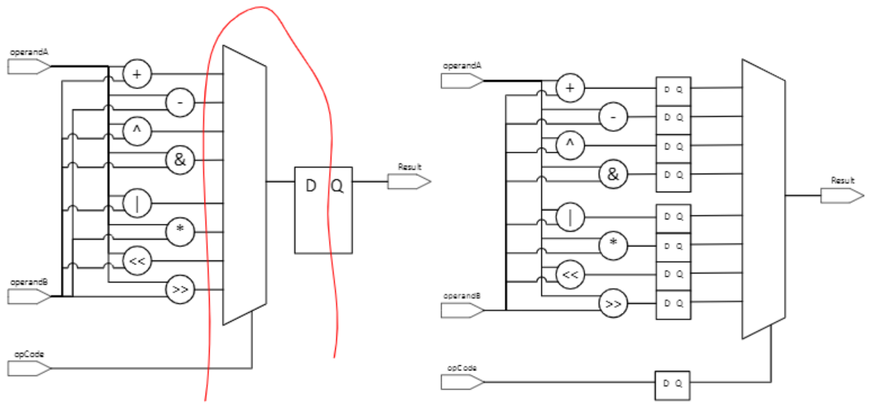
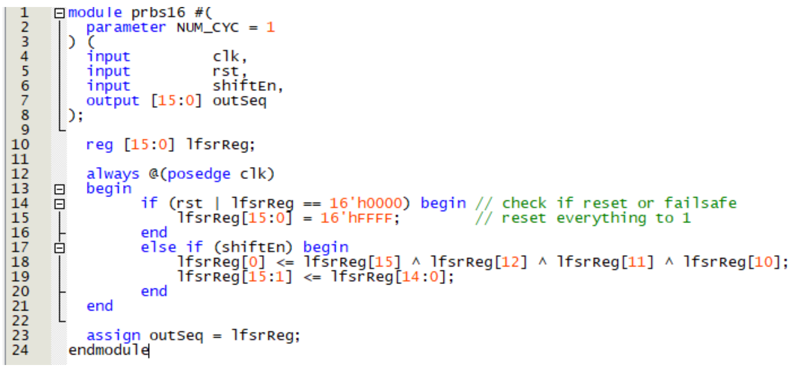


Figure 4: Cut-set retiming modifications to ALU.

2.2 Design and implementation procedure:

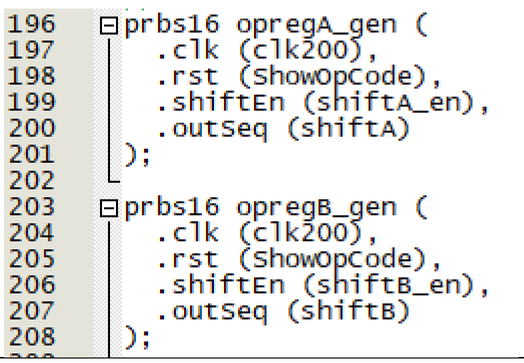
2.2.1 PRBS sequence generator

Section 2.1.2 describes the theory behind PRBS, this section will simply show the code that implements it.



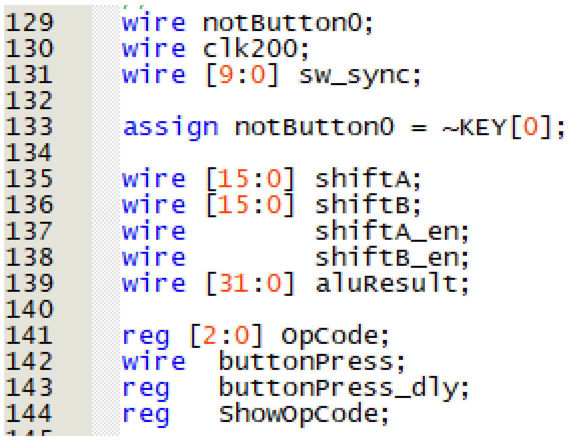
```
module prbs16 #(
  parameter NUM_CYC = 1
) (
  input         clk,
  input         rst,
  input         shiftEn,
  output [15:0] outSeq
);

  reg [15:0] lfsrReg;

  always @(posedge clk)
  begin
        if (rst | lfsrReg == 16'h0000) begin // check if reset or failsafe
           lfsrReg[15:0] = 16'hFFFF;         // reset everything to 1
        end
        else if (shiftEn) begin
           lfsrReg[0] <= lfsrReg[15] ^ lfsrReg[12] ^ lfsrReg[11] ^ lfsrReg[10];
           lfsrReg[15:1] <= lfsrReg[14:0];
        end
  end

  assign outSeq = lfsrReg;
endmodule
```

Code 1: PRBS implemented with LFSR checks



```
prbs16 opregA_gen (
   .clk (clk200),
   .rst (ShowOpCode),
   .shiftEn (shiftA_en),
   .outSeq (shiftA)
);

prbs16 opregB_gen (
   .clk (clk200),
   .rst (ShowOpCode),
   .shiftEn (shiftB_en),
   .outSeq (shiftB)
);
```

Code 2: Values inserted into Code 1.

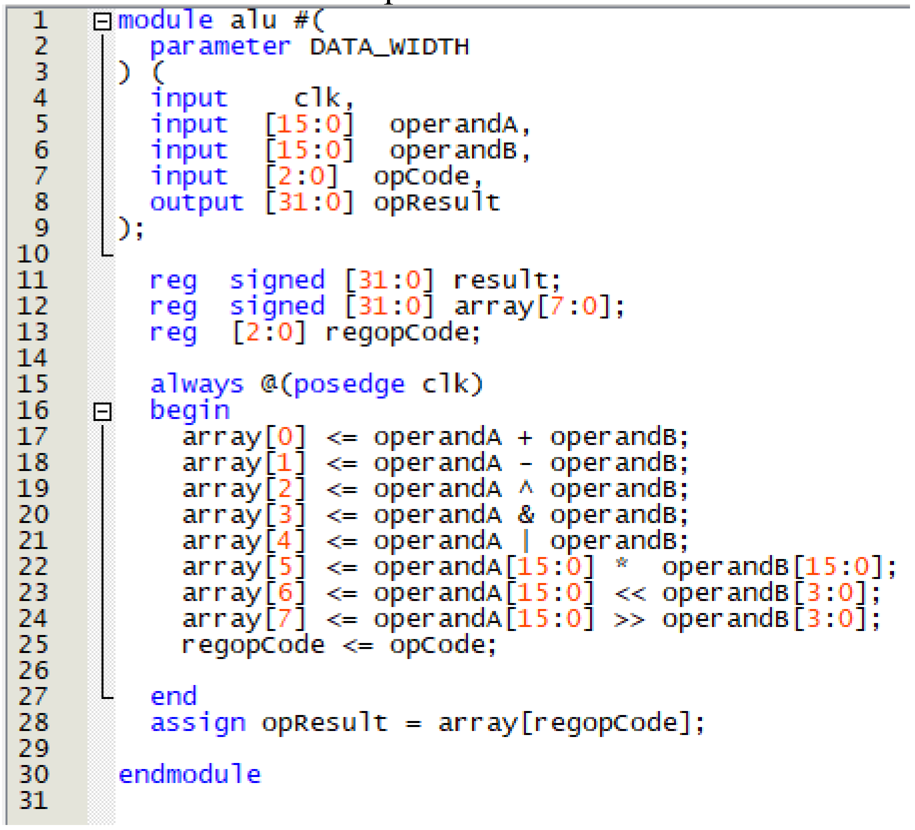


```
wire notButton0;
wire clk200;
wire [9:0] sw_sync;

assign notButton0 = ~KEY[0];

wire [15:0] shiftA;
wire [15:0] shiftB;
wire        shiftA_en;
wire        shiftB_en;
wire [31:0] aluResult;

reg [2:0] opCode;
wire  buttonPress;
reg   buttonPress_dly;
reg   showOpCode;
```

Code 3: Values defined for Code 2.

2.2.2 Cut-set retiming modifications

This section will note the changes made to ALU for the Cut-Set retiming modifications. Code 4 shows the implementation.



```
module alu #(
    parameter DATA_WIDTH
) (
    input    clk,
    input   [15:0]  operandA,
    input   [15:0]  operandB,
    input   [2:0]  opCode,
    output  [31:0] opResult
);

    reg  signed [31:0] result;
    reg  signed [31:0] array[7:0];
    reg  [2:0] regopCode;

    always @(posedge clk)
    begin
      array[0] <= operandA + operandB;
      array[1] <= operandA - operandB;
      array[2] <= operandA ^ operandB;
      array[3] <= operandA & operandB;
      array[4] <= operandA | operandB;
      array[5] <= operandA[15:0] *  operandB[15:0];
      array[6] <= operandA[15:0] << operandB[3:0];
      array[7] <= operandA[15:0] >> operandB[3:0];
      regopCode <= opCode;

    end
    assign opResult = array[regopCode];

endmodule
```

Code 4: Cut set retiming implemented

3. Demonstration

3.1 Implementation method:

Section 2.2 covers the major TODO modifications in the code including the PRBS and Cut-set retiming modifications. I will not cover any other implementation here as those modifications suffice for the overall understanding of this lab.

3.2 Video Link:

Posted to the google drive provided.

4. Experimental Results

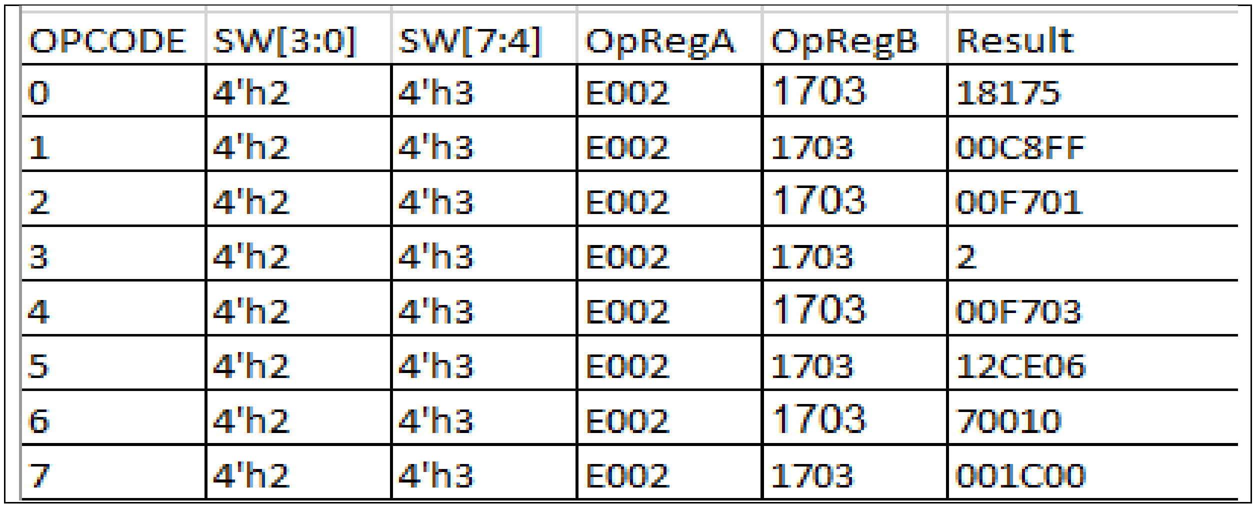
4.1 Observations:

Comparing to lab 3, this lab is a bit more confusing with the random number generator, but in general it makes sense why we implement it this way. Trying to see through the random number mess for your actual value can be a bit confusing at times but overall it’s not too bad.

4.2 Questions:

4.2.1 Phase 4 Questions:

-Settings for SW[7:4] and SW[3:0] & LFSR 16-bit values corresponding to these two inputs



OPCODE	SW[3:0]	SW[7:4]	OpRegA	OpRegB	Result
0	4'h2	4'h3	E002	1703	18175
1	4'h2	4'h3	E002	1703	00C8FF
2	4'h2	4'h3	E002	1703	00F701
3	4'h2	4'h3	E002	1703	2
4	4'h2	4'h3	E002	1703	00F703
5	4'h2	4'h3	E002	1703	12CE06
6	4'h2	4'h3	E002	1703	70010
7	4'h2	4'h3	E002	1703	001C00

-Do these computations all work out?

Yes. See section 4.4.5 to see more calculations.

-With OpCode=5 (multiply), did the result take an extra cycle to settle after the final value of outSeq settled? If so, that's the timing failure showing. If not, your device isn't as bad as a "Slow" device

Before I implemented the Cut-set retiming, it did take an extra cycle. Afterwards, however, everything worked out.

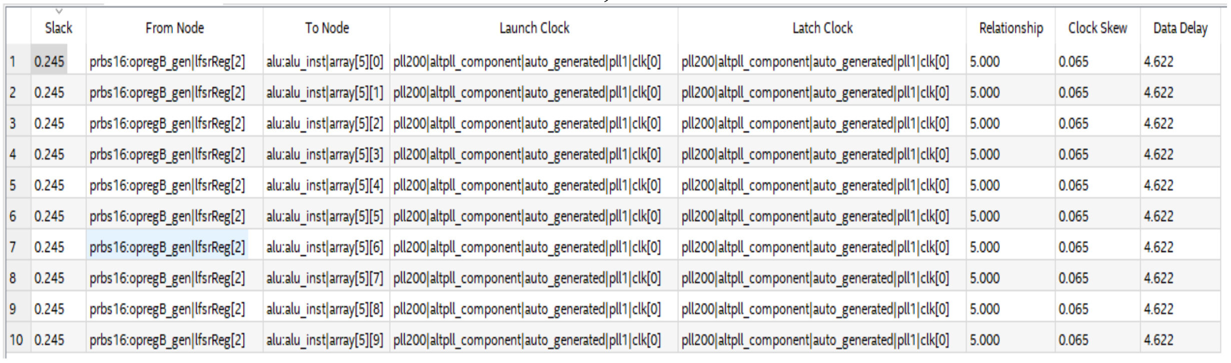
4.3 Pre-Laboratory results:

No pre-laboratory results are to be shown for this lab.

4.4 Post lab questions:

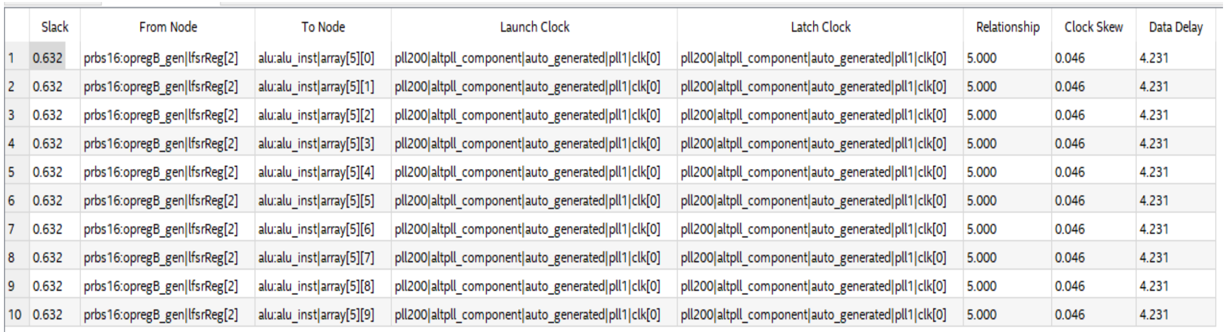
4.4.1 After the modification to the ALU, the circuit should have met timing. In other words, the worst-case path, including clock skew, met setup timing for a 5.0 nsec (200 MHz) system clock. By going into the TimeQuest timing analyzer, determine the worst-case slack of the final circuit. With that information –how fast of a clock could you have run this design at while still meeting timing at 85 degrees C?

The worst-case slack is for 85C is 0.245, however all the 10 worst are 0.245:



	Slack	From Node	To Node	Launch Clock	Latch Clock	Relationship	Clock Skew	Data Delay
1	0.245	prbs16:opregB_gen|lfsrReg[2]	alu:alu_inst|array[5][0]	pll200|altpll_component|auto_generated|pll1|clk[0]	pll200|altpll_component|auto_generated|pll1|clk[0]	5.000	0.065	4.622
2	0.245	prbs16:opregB_gen|lfsrReg[2]	alu:alu_inst|array[5][1]	pll200|altpll_component|auto_generated|pll1|clk[0]	pll200|altpll_component|auto_generated|pll1|clk[0]	5.000	0.065	4.622
3	0.245	prbs16:opregB_gen|lfsrReg[2]	alu:alu_inst|array[5][2]	pll200|altpll_component|auto_generated|pll1|clk[0]	pll200|altpll_component|auto_generated|pll1|clk[0]	5.000	0.065	4.622
4	0.245	prbs16:opregB_gen|lfsrReg[2]	alu:alu_inst|array[5][3]	pll200|altpll_component|auto_generated|pll1|clk[0]	pll200|altpll_component|auto_generated|pll1|clk[0]	5.000	0.065	4.622
5	0.245	prbs16:opregB_gen|lfsrReg[2]	alu:alu_inst|array[5][4]	pll200|altpll_component|auto_generated|pll1|clk[0]	pll200|altpll_component|auto_generated|pll1|clk[0]	5.000	0.065	4.622
6	0.245	prbs16:opregB_gen|lfsrReg[2]	alu:alu_inst|array[5][5]	pll200|altpll_component|auto_generated|pll1|clk[0]	pll200|altpll_component|auto_generated|pll1|clk[0]	5.000	0.065	4.622
7	0.245	prbs16:opregB_gen|lfsrReg[2]	alu:alu_inst|array[5][6]	pll200|altpll_component|auto_generated|pll1|clk[0]	pll200|altpll_component|auto_generated|pll1|clk[0]	5.000	0.065	4.622
8	0.245	prbs16:opregB_gen|lfsrReg[2]	alu:alu_inst|array[5][7]	pll200|altpll_component|auto_generated|pll1|clk[0]	pll200|altpll_component|auto_generated|pll1|clk[0]	5.000	0.065	4.622
9	0.245	prbs16:opregB_gen|lfsrReg[2]	alu:alu_inst|array[5][8]	pll200|altpll_component|auto_generated|pll1|clk[0]	pll200|altpll_component|auto_generated|pll1|clk[0]	5.000	0.065	4.622
10	0.245	prbs16:opregB_gen|lfsrReg[2]	alu:alu_inst|array[5][9]	pll200|altpll_component|auto_generated|pll1|clk[0]	pll200|altpll_component|auto_generated|pll1|clk[0]	5.000	0.065	4.622

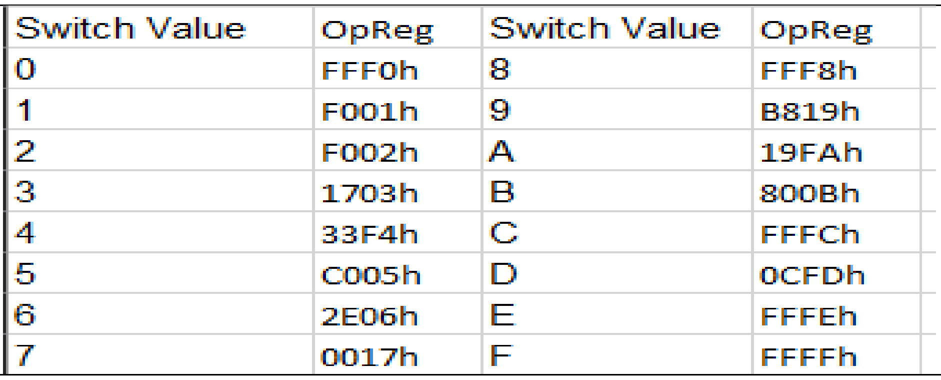
A similar behavior can be seen for the 0C model:



	Slack	From Node	To Node	Launch Clock	Latch Clock	Relationship	Clock Skew	Data Delay
1	0.632	prbs16:opregB_gen|lfsrReg[2]	alu:alu_inst|array[5][0]	pll200|altpll_component|auto_generated|pll1|clk[0]	pll200|altpll_component|auto_generated|pll1|clk[0]	5.000	0.046	4.231
2	0.632	prbs16:opregB_gen|lfsrReg[2]	alu:alu_inst|array[5][1]	pll200|altpll_component|auto_generated|pll1|clk[0]	pll200|altpll_component|auto_generated|pll1|clk[0]	5.000	0.046	4.231
3	0.632	prbs16:opregB_gen|lfsrReg[2]	alu:alu_inst|array[5][2]	pll200|altpll_component|auto_generated|pll1|clk[0]	pll200|altpll_component|auto_generated|pll1|clk[0]	5.000	0.046	4.231
4	0.632	prbs16:opregB_gen|lfsrReg[2]	alu:alu_inst|array[5][3]	pll200|altpll_component|auto_generated|pll1|clk[0]	pll200|altpll_component|auto_generated|pll1|clk[0]	5.000	0.046	4.231
5	0.632	prbs16:opregB_gen|lfsrReg[2]	alu:alu_inst|array[5][4]	pll200|altpll_component|auto_generated|pll1|clk[0]	pll200|altpll_component|auto_generated|pll1|clk[0]	5.000	0.046	4.231
6	0.632	prbs16:opregB_gen|lfsrReg[2]	alu:alu_inst|array[5][5]	pll200|altpll_component|auto_generated|pll1|clk[0]	pll200|altpll_component|auto_generated|pll1|clk[0]	5.000	0.046	4.231
7	0.632	prbs16:opregB_gen|lfsrReg[2]	alu:alu_inst|array[5][6]	pll200|altpll_component|auto_generated|pll1|clk[0]	pll200|altpll_component|auto_generated|pll1|clk[0]	5.000	0.046	4.231
8	0.632	prbs16:opregB_gen|lfsrReg[2]	alu:alu_inst|array[5][7]	pll200|altpll_component|auto_generated|pll1|clk[0]	pll200|altpll_component|auto_generated|pll1|clk[0]	5.000	0.046	4.231
9	0.632	prbs16:opregB_gen|lfsrReg[2]	alu:alu_inst|array[5][8]	pll200|altpll_component|auto_generated|pll1|clk[0]	pll200|altpll_component|auto_generated|pll1|clk[0]	5.000	0.046	4.231
10	0.632	prbs16:opregB_gen|lfsrReg[2]	alu:alu_inst|array[5][9]	pll200|altpll_component|auto_generated|pll1|clk[0]	pll200|altpll_component|auto_generated|pll1|clk[0]	5.000	0.046	4.231

Essentially, to run the code at 85C we would need to have run at 3300Mhz.

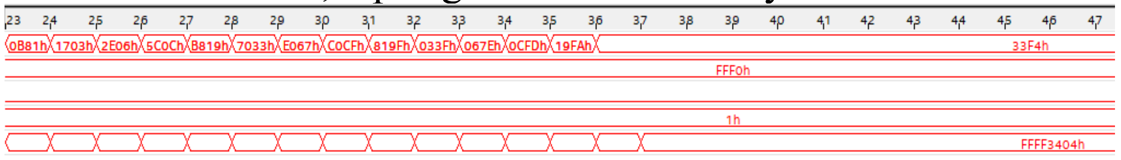
4.4.2 The LFSR generates a pseudo-random binary sequence, which is the same for both instances of the LFSR. Both instances will stop counting once their least-significant bits match the value on the 4 switches each is tied to. Using the SignalTap logic analyzer, determine the final state for all 16 possible values of SW[7:4] or SW[3:0] (they should be the same as each other), writing them in your lab notebook.



Switch Value	OpReg	Switch Value	OpReg
0	FFF0h	8	FFF8h
1	F001h	9	B819h
2	F002h	A	19FAh
3	1703h	B	800Bh
4	33F4h	C	FFFCh
5	C005h	D	0CFDh
6	2E06h	E	FFFEh
7	0017h	F	FFFFh

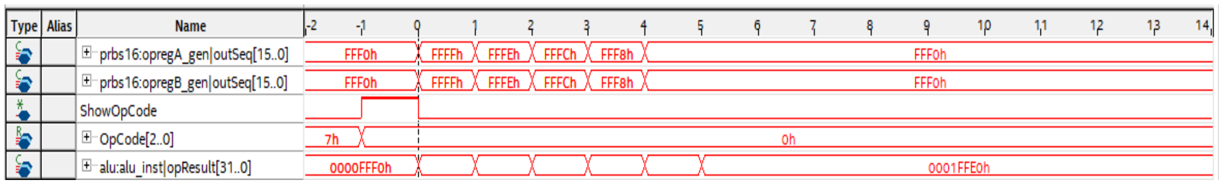
4.4.3 Again, using SignalTap, write out the sequence (in Verilog 16’hxxxx hexadecimal notation) of output numbers as long as possible given that it will stop once the sequence has reached the SW[3:0] or SW[7:4] value. How many clock cycles is the longest sequence?

The switch value of 4, OpReg 33F4h takes 36 cycles to settle.



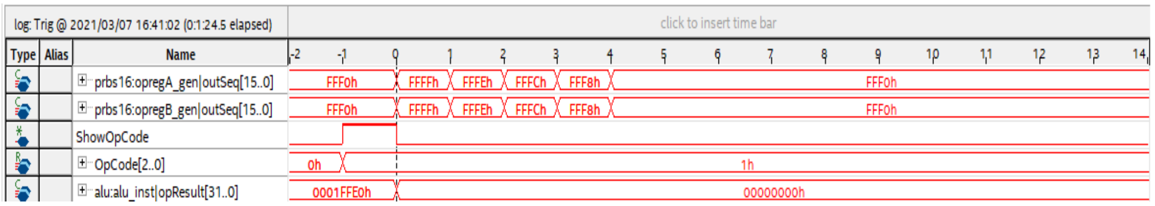
4.4.4 Set both SW[7:4] and SW[3:0] to the value 4’h0. Select OpReg=3’d0 and observe the ALU’s result signal in SignalTap. Report on its behavior, what values it puts out at which cycles following the trigger (again,triggering on the ShowOpCode signal). Then, keeping all the switches set to 0 still, select OpReg=3’d1. Observe and report on the behavior of the ALU’s result signal for this opcode.

The first photo shown describes the output for the OpCode 3'd0. The first 4 cycles are trying to figure out where they want to be, by cycle 4 we have figured it out, however. Opresult takes one cycle longer to settle than the outsequence.



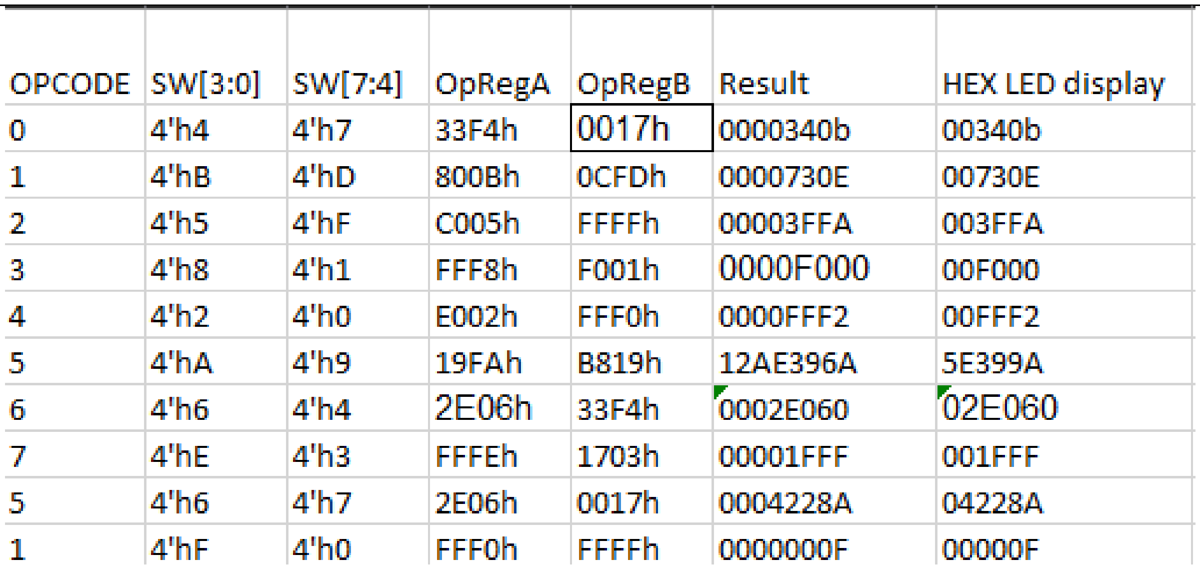
Type
Alias
Name
prbs16:opregA_gen|outSeq[15..0]
prbs16:opregB_gen|outSeq[15..0]
ShowOpCode
OpCode[2..0]
alu:alu_inst|opResult[31..0]
FFF0h
FFFFh
FFFEh
FFFCh
FFF8h
7h
0h
0000FFF0h
0001FFE0h

The second photo shown has the same input values but with opcode 3'd1, or subtraction. As you can see, an operation that goes to 000…h is much quicker than one that is generated by the PBSR. The first photo has PBSR generate an initial value, however for subtraction we do not do this as we cannot go lower than 0.



log: Trig @ 2021/03/07 16:41:02 (0:1:24.5 elapsed)
click to insert time bar
Type
Alias
Name
prbs16:opregA_gen|outSeq[15..0]
prbs16:opregB_gen|outSeq[15..0]
ShowOpCode
OpCode[2..0]
alu:alu_inst|opResult[31..0]
FFF0h
FFFFh
FFFEh
FFFCh
FFF8h
0h
1h
0001FFE0h
00000000h

4.4.5 By observing the values in SignalTap, fill out the following table:



OPCODE	SW[3:0]	SW[7:4]	OpRegA	OpRegB	Result	HEX LED display
0	4'h4	4'h7	33F4h	0017h	0000340b	00340b
1	4'hB	4'hD	800Bh	0CFDh	0000730E	00730E
2	4'h5	4'hF	C005h	FFFFh	00003FFA	003FFA
3	4'h8	4'h1	FFF8h	F001h	0000F000	00F000
4	4'h2	4'h0	E002h	FFF0h	0000FFF2	00FFF2
5	4'hA	4'h9	19FAh	B819h	12AE396A	5E399A
6	4'h6	4'h4	2E06h	33F4h	0002E060	02E060
7	4'hE	4'h3	FFFEh	1703h	00001FFF	001FFF
5	4'h6	4'h7	2E06h	0017h	0004228A	04228A
1	4'hF	4'h0	FFF0h	FFFFh	0000000F	00000F

5. Conclusions

5.1 - Results and lessons learned:

This lab was very informative. I learned more about the tools in Quartus (specifically the Signal Tap Logic Analyzer). This lab also introduced different timing ideas and how to fix them. We can see this fix having real effects on the performance of the code. Lastly, understanding how the PBSR and LFSR works helps to better understand the functions we usually use in our everyday code. Being able to generate random numbers to fix a bug is very interesting to me.