



CPE 324 Advanced Logic Design Laboratory

Laboratory Assignment #4

Extended Arithmetic Logic Unit

(12% of Final Grade)

Purpose

The purpose of this laboratory is to give each student the opportunity to expand his/her knowledge of synchronous circuit design, adding synchronous resets, elements of clock synthesis with phase locked loops (PLLs), pseudo-random number generation with linear feedback shift registers (LFSRs), using internal logic analyzer tools, and techniques for overcoming timing failures. This laboratory experience also provides students with the opportunity to continue building experience using hardware description language design entry paradigms. It also requires that students be able to integrate their own design elements into a larger system that is composed of additional intellectual property modules.

Design Problem

The specific problem is to extend the synchronous ALU design from lab 3 by expanding the operands to 16-bits each, with a system clock running at 200 MHz. This experiment will be carried out on the DE2-115 or DE10-Lite development board (according to the student's preference), and will feature the use of the switches (SW[9:0]), pushbuttons (KEY[1:0]), and 7-segment LED displays (HEX[6:0]). This time, the number of available switches presents an even greater limitation, so the lab will make use of a pair of pseudo-random number generators to create the operand values. The switches will be used to key in the OPCODE for this lab and also to effectively halt the sequence of pseudo-random numbers when their bits match the value presented on the switches. The pushbuttons are used as a global synchronous reset as well as to restart the operand generators.

The following table shows the required opcodes for the ALU, where now the:

Table 1: ALU Opcode Table

OPCODE	Operation
0: ADD	RESULT = REGA[15:0] + REGB[15:0]
1: SUBTRACT	RESULT = REGA[15:0] - REGB[15:0]
2: XOR	RESULT = REGA[15:0] ^ REGB[15:0]
3: AND	RESULT = REGA[15:0] & REGB[15:0]
4: OR	RESULT = REGA[15:0] REGB[15:0]
5: MULTIPLY	RESULT = REGA[15:0] * REGB[15:0]
6: SHIFT LEFT	RESULT = REGA[15:0] << REGB[3:0]
7: SHIFT RIGHT	RESULT = REGA[15:0] >> REGB[3:0]

Background

The Arithmetic Logic Unit (ALU) we generated in the previous lab performed operations on 8-bit operands. Here we are maintaining the same operations, but expanding them to cover 16-bit operands. In this lab, we will take an example implementation for the 8-bit case from Lab 3, and create a new parameter that defines how wide the operands are. In Lab 3 we identified the 8 x 8 multiplier as the most complex operation that this ALU performs. When expanding to 16 x 16, the operation quadruples in complexity compared to 8 x 8, and the impact of the additional complexity is that it will be harder to meet timing constraints. 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.

This lab will make use of a PLL to generate the desired system clock frequency of 200 MHz. A typical PLL contains a voltage-controlled oscillator (VCO), a phase detector, and a low-pass filter, and operates in a closed loop. The VCO output is divided down (see $\div N$ Counter block) and compared to a reference clock (which also may be divided down to a different frequency). Let's call the rising edge of the divided-down clocks to be 0 degree phase (thus the falling edge would be at 180 degrees). The phase detector compares which input reaches 0 degrees phase first and adjusts the VCO control voltage accordingly; if the VCO output changed first then the control signal voltage decreases to slow down the VCO, but otherwise the control voltage increases to speed up the VCO. The low-pass filter averages out the control voltage to prevent the VCO from experiencing wild swings. A figure of a PLL is shown below:

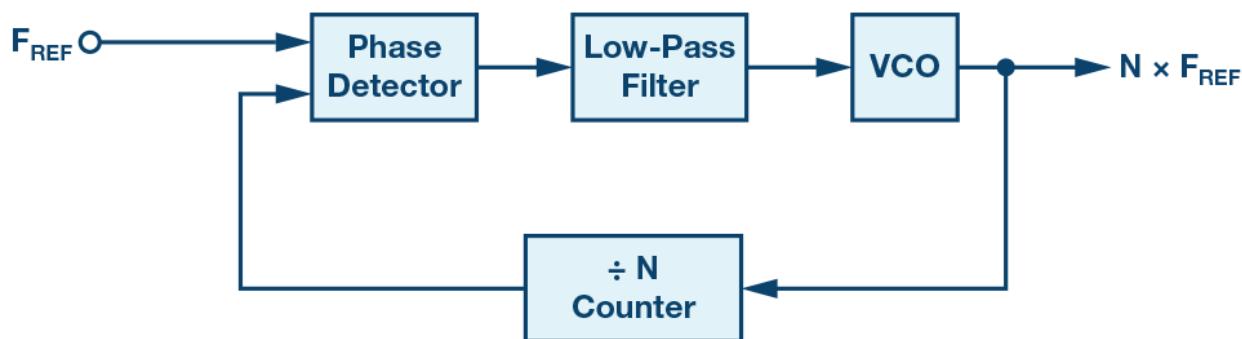


Figure 1: Phase Locked Loop

To generate a clean, low-jitter clock above ~ 1 MHz, the VCO should be an analog circuit. Fortunately, modern FPGAs, CPUs, and other ASICs typically integrate one or more PLLs to enable clock synthesis of other frequencies besides the ones available on the board. The DE10 and DE2 boards both are equipped with 50 MHz crystal oscillators, but for example if a 125 MHz clock is needed for a Gigabit Ethernet interface, a PLL could be used (by dividing the ref clock by 2 and dividing the VCO by 5). The KEY[0] pushbutton will be used to reset the PLL component, and LEDR[0] will display the “locked” output status from the PLL.

In this lab, again the student is hard-pressed to generate two 16-bit operands for stimulating the ALU. To overcome this challenge, the circuit will employ a pair of 16-bit LFSRs. With the correct feedback configuration, an LFSR spanning N bits of flip-flop registers can generate a repeating pattern of $2^N - 1$ bits in duration. The polynomial used for this lab is as follows:

$$x^{16} + x^{13} + x^{12} + x^{11} + 1$$

And it can be implemented by the following LFSR circuit:

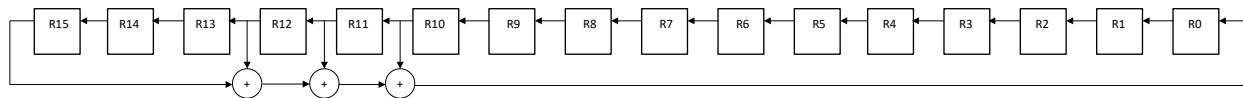


Figure 2: 16-bit LFSR for Lab 4

Note in the above circuit that each box labeled R is a different flip-flop, and the circles labeled “+” are 1-bit finite field adders, which are logically XOR gates. The whole arena of finite field arithmetic is extremely interesting for synchronous digital circuits, with many applications in error control coding schemes. For now, we are simply concerned with the above circuit’s ability to generate a repeating sequence of 65,535 bits. This sequence of numbers is also known as a pseudo-random binary sequence (PRBS). Because the number 65,535 is mutually prime with 16 (the number of registers), this circuit generates a sequence of all 65,535 non-zero valued 16-bit numbers in 16 clock cycles. Note that if the value 16’h0000 ever is reached in this circuit, it will be stuck forever at all-zeros.

To achieve a pair of 16-bit operands, the circuit will use the LFSRs in conjunction with the SW[7:0] switches. Any change to the OpCode register shall cause LFSRs to reset and count until their 4 least significant bits (LSBs) equal SW[3:0] for operandA and SW[7:4] for operandB. Selecting a new value for the OpCode[2:0] register is achieved by pressing KEY[1], upon which time the 6-digit hexadecimal display reads “Code X”, where X is on the range of 0 to 7.

As was the case in Lab 3, there are several signals that come into the FPGA that are totally asynchronous to the 200 MHz system clock. By asynchronous, it is meant that these signals will change values irrespective of the clock period and the corresponding setup and hold time of the flip-flops that the input signals feed into. As was shown in CPE 322 lectures, a synchronizing circuit is the preferred method for handling asynchronous signals entering into the destination clock domain. Although it wasn’t highlighted in Lab 3, the *meta.v* module creates a cascaded set of DATA_WIDTH wide by DEPTH deep flip-flops. Prior to their use on the clk200 clock domain, SW[7:0] and KEY[1] must pass through a 2-deep *meta* instance to eliminate the possibility of metastability in the circuit. The block diagram of the 2-deep *meta* block is shown in the following figure:

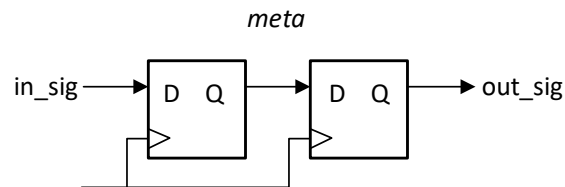
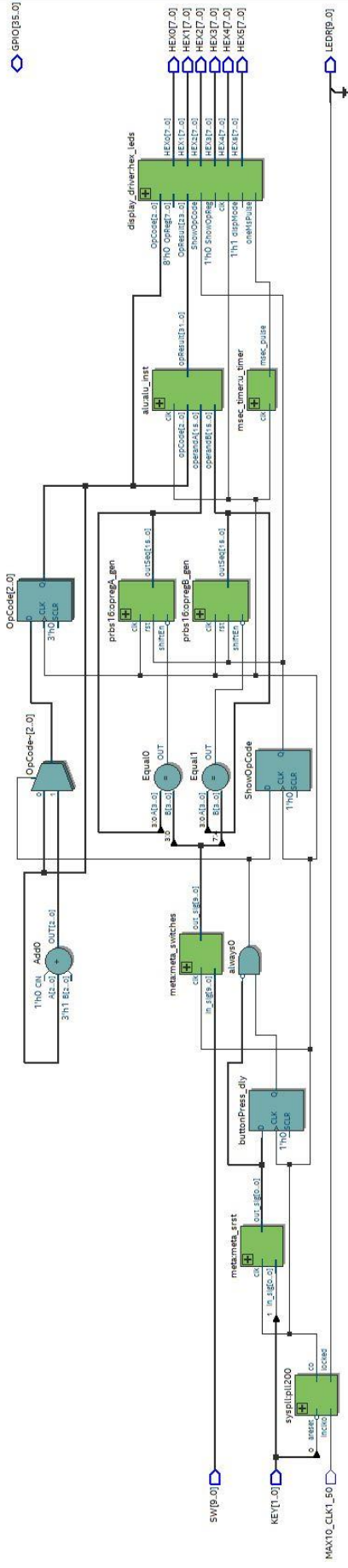


Figure 3: 2-flop Synchronizer Circuit

Finally, this lab will involve the use of the SignalTap logic analyzer for interactive viewing of the state of the signals as they are operating on the board. A logic analyzer is a classic piece of lab equipment used for sampling and displaying digital signals, as shown below:



Laboratory Assignment:

Quartus version 17.1 is used for this description. The hope is that later versions offer an equivalent design experience, but if not then please consider downloading 17.1 and changing to it.

This laboratory assignment is composed of five phases, with each phase culminating with the current state of the design being validated by prototyping it on the Terasic DE2-115 or DE10-Lite platform. To prevent the student demonstrations from exceeding the 5 minute suggested limit, please wait to demonstrate until after the final phase and into the assignment question section.

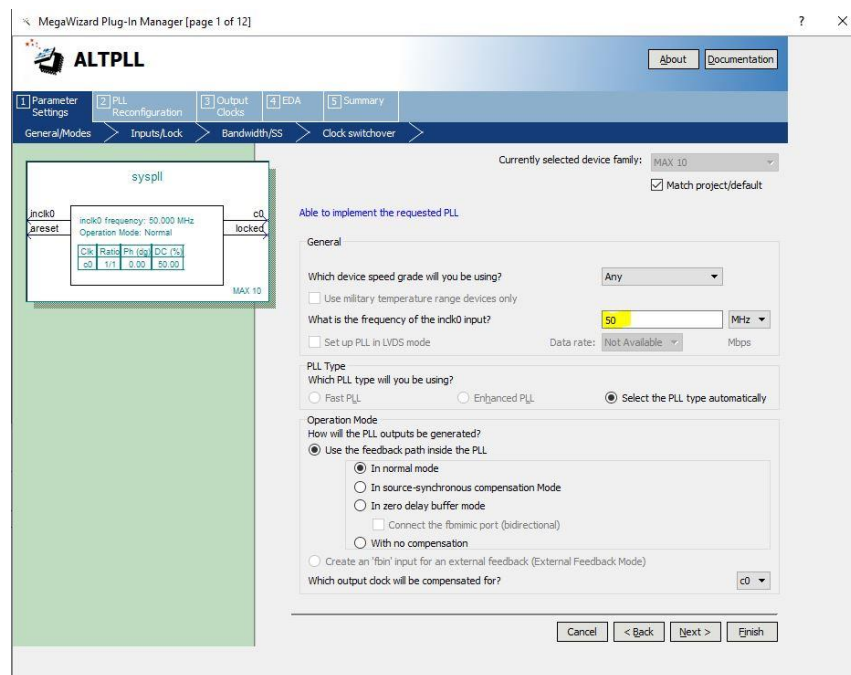
Preparation:

Use the lab4_de10_start.qar or lab4_de2_start.qar file according to the board with which you will perform the lab.

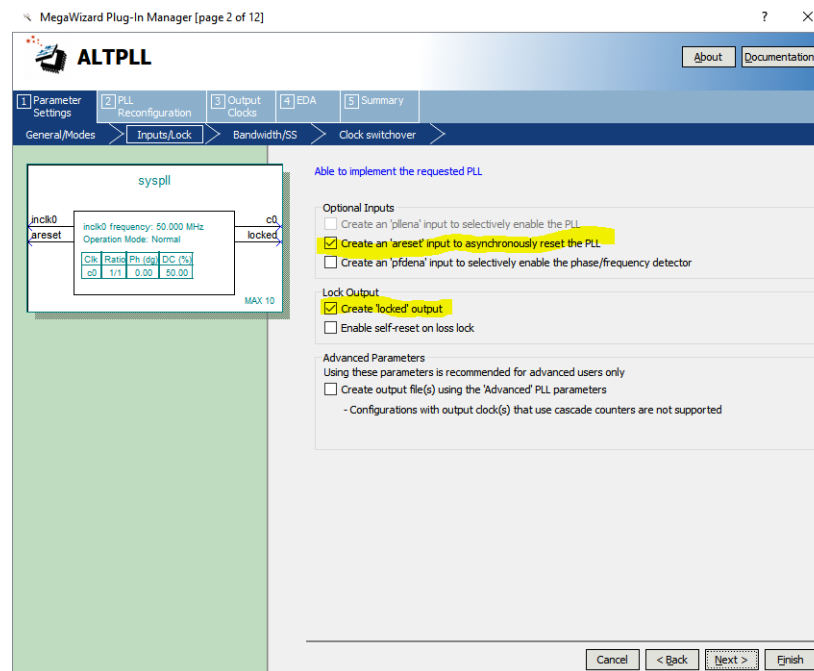
Phase I: Generate PLL for 200 MHz system clock. In this phase, students must create and instantiate a PLL. The Cyclone IV (DE2) and MAX10 (DE10) devices each contain 4 PLL components, and Quartus provides a convenient way to set these up and instantiate them. To do so, select “Tools > IP Catalog”, which brings up a new window pane in Quartus. In that pane, select “Installed IP > Library > Basic Functions > Clocks; PLLs and Resets > PLL > ALTPLL”. That should bring up a dialog box that asks the student to name the variation being generated: create the file name syspll.v (with IP variation file type set to Verilog).

In the “MegaWizard Plug-In Manager” interface that assists in the generation of the PLL, please use the following settings (defaults for other settings should be fine):

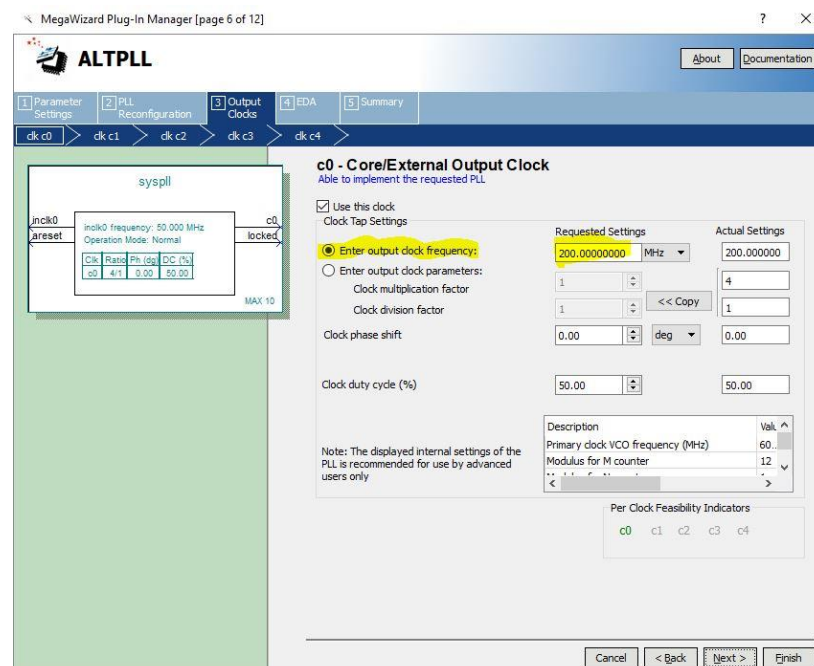
1. What is the frequency of the inclk0 input? **50.000 MHz**



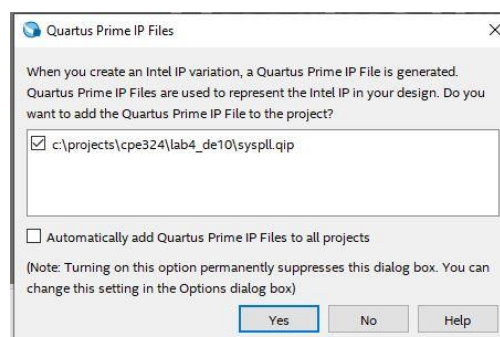
2. Create an 'areset' input to asynchronously reset the PLL: **CHECKED**
3. Create 'locked' output: **CHECKED**



4. c0 – Core/External Output Clock > Enter output clock frequency > Requested Settings:
200.000 MHz



5. After clicking Finish, choose Yes to add the .qip file to the project



Note that the PLLs in these FPGAs have multiple outputs that are driven from programmable divider ratios from the VCO. For the DE10 board, the GUI showed that the VCO frequency was 600.0 MHz, the M counter (for VCO divider) is 12, the N counter (for reference divider) is 1:

$$f_{ref}/N = f_{VCO}/M :: 50.0 \text{ MHz}/1 = 600.0 \text{ MHz}/12$$

To generate c0 output at 200 MHz, the VCO is divided down by a “post-scale counter” of 3, meaning that it counts from 0-2 repeatedly. The same VCO could produce 300 MHz, 150 MHz, 100 MHz, 85.714 MHz, 75 MHz, etc. simultaneously out of the c1-c4 outputs. Other possibilities are to generate multiple phases of the same clock (e.g. 90 degrees, 180 degrees out of phase between c0-c4 outputs), which can be helpful for chip interfaces, etc.

Lab 4 only requires a single 200 MHz clock on the c0 interface. Once that is finished, select the option in the dialog box to add this new component to the project (or add the source to the project as syspll.qip and/or syspll.v).

After adding this to the project, open up the lab4.sdc file (which should already be in the project), and adding the following lines (or ensuring each is in the file):

DE10-Lite:

```
create_clock -name {MAX10_CLK1_50} -period 20.0 -waveform {0.000 10.000}
[get_ports {MAX10_CLK1_50}]
```

```
derive_pll_clocks
```

DE2-115:

```
create_clock -name {CLOCK_50} -period 20.0 -waveform {0.000 10.000} [get_ports
{CLOCK_50}]
```

```
derive_pll_clocks
```

The create_clock constraint defines the clock period as a 50 MHz clock with the falling edge occurring 10 nsec after the rising edge. The derive_pll_clocks directive instructs Quartus to automatically propagate the input clock frequencies through all PLLs in the design. This is a quick alternative to the more formal create_generated_clocks constraint that is more industry-standard.

Insert an instance of syspll in the top-level design, connecting the 50 MHz input pin (MAX10_CLK1_50 for DE10 or CLOCK_50 in DE2) to the inclk0 input, connecting an inverted KEY[0] to the areset input, a new wire signal clk200 to the c0 output, and finally LEDR[0] to the locked output. Since the areset input is active-high, and pressing the button causes the KEY[0] input signal to go low, be sure to invert KEY[0] on its way into areset. To complete this phase, build the project. Assuming the build completes correctly, verify that pressing KEY[0] makes LEDR[0] go dark, and releasing it causes LEDR[0] to illuminate.

Phase II: Modifications to display_driver and alu modules

The display_driver module was originally designed to display a 5-digit decimal number of 4-digit hex number. Now that an 8-digit hex number is possible, the decimal display is no longer needed,

and the student is required to expand the OpResult input to a 24-bit value. Likewise, the OpReg display will not be needed. The student is required to find the TODO lines to change the contents of the display. The student must instantiate the display_driver module in the top-level, tying the dispMode, OpReg, and showOpReg inputs to values that avoids their use.

The alu module was originally designed to perform 8 opcodes on two 8-bit input operands, producing a 16-bit output. In the previous lab it was up to the student to generate the file, but it is provided for the student in this lab as a starting point. Now it must be changed to accommodate a 16-bit variation, but the requirement here is to avoid backwards incompatibility. Therefore, the student must add a new parameter to the alu module, named DATA_WIDTH, and use it to declare input, output, and internal signal widths.

Upon completion of this phase, simply run the Compile Design > Analysis & Synthesis step to ensure proper syntax and connections.

Phase III: Create the PRBS sequence generator

The prbs16 module must implement the LFSR that is illustrated in Figure 2, remembering that the + operator is simply an XOR and that each register is a 1-bit DQ flip-flop register. There are a few additional behavioral constraints required for this step:

1. The clk input is connected to clk200 in the top-level and shifts the LFSRs by one
2. The rst input is a synchronous, active-high reset that sets the value of all LFSR registers to 1 (i.e. the set of 16 registers is set to 16'hFFFF)
3. A failsafe mode must be added that detects if the LFSR register values altogether reach the value 16'h0000, they are all set to 1's (16'hFFFF)
4. An input signal named shiftEn controls whether the 16 registers are updated (Q output takes the value of D input during the clock edge) or not; when 1, the 16-bit reg is assigned a new value within the always @(posedge clk) block, and when 0, it is not assigned a new value (or it is assigned the Q output value). DO NOT GATE THE CLOCK SIGNAL DIRECTLY - in other words, DO NOT do something like the following:

```
wire clock_gated = shiftEn & clk;
```

```
always @(posedge clock_gated)...
```

5. The register Q outputs shall be connected to the output signal outSeq

Upon completion of this module, a pair of instances must be enabled in the top-level design by simply removing the comment slashes `//`. The student will note that the shiftEn input is activated whenever the LFSR output's 4 LSBs are not equal to the state of the SW[3:0] inputs for shiftA or SW[7:4] for shiftB.

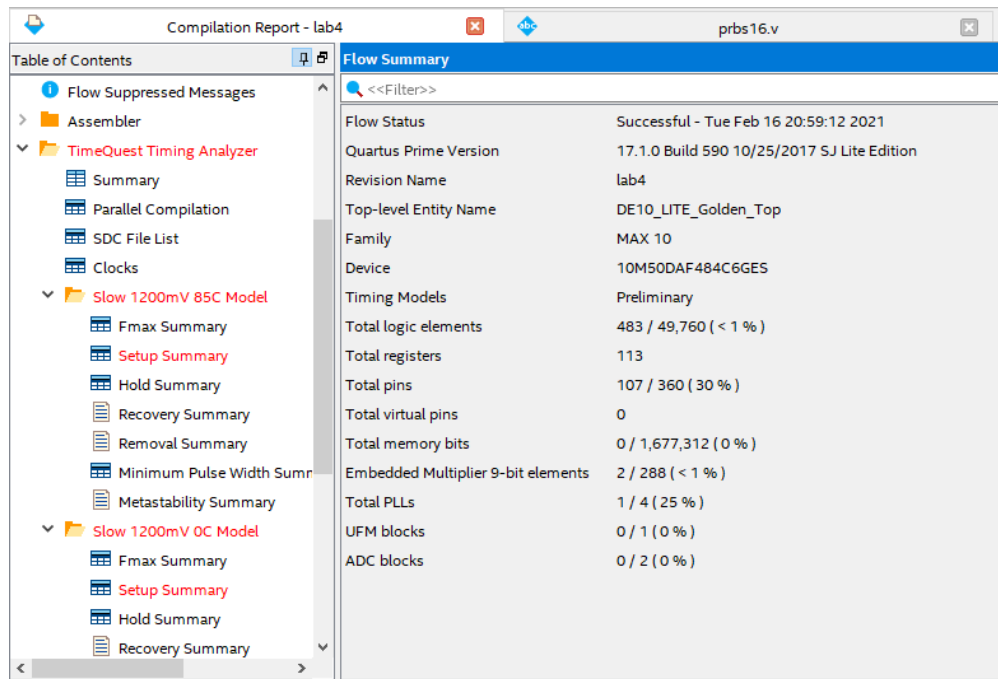
At this point, the student must compile the design, which at this point should be nearing a level of completion that allows functionality to be demonstrated.

Prior to moving on to Phase IV, please set SW[7:0] to all zeros on your board (switch position

closest to card edge). Program the board with the .sof file in the Programmer tool. Verify that 6 zero digits show up on the display and that pressing KEY[1] button results in the familiar CodE X display that increments upon each button press.

Phase IV: Enable SignalTap

Prior to moving on, please notice one thing in the Compilation Report > Table of Contents window pane. The TimeQuest Timing Analyzer item is red, but expanding it reveals that Setup timing has failed for devices characterized as Slow for both the 0 degrees C and 85 degrees C operating temperatures:

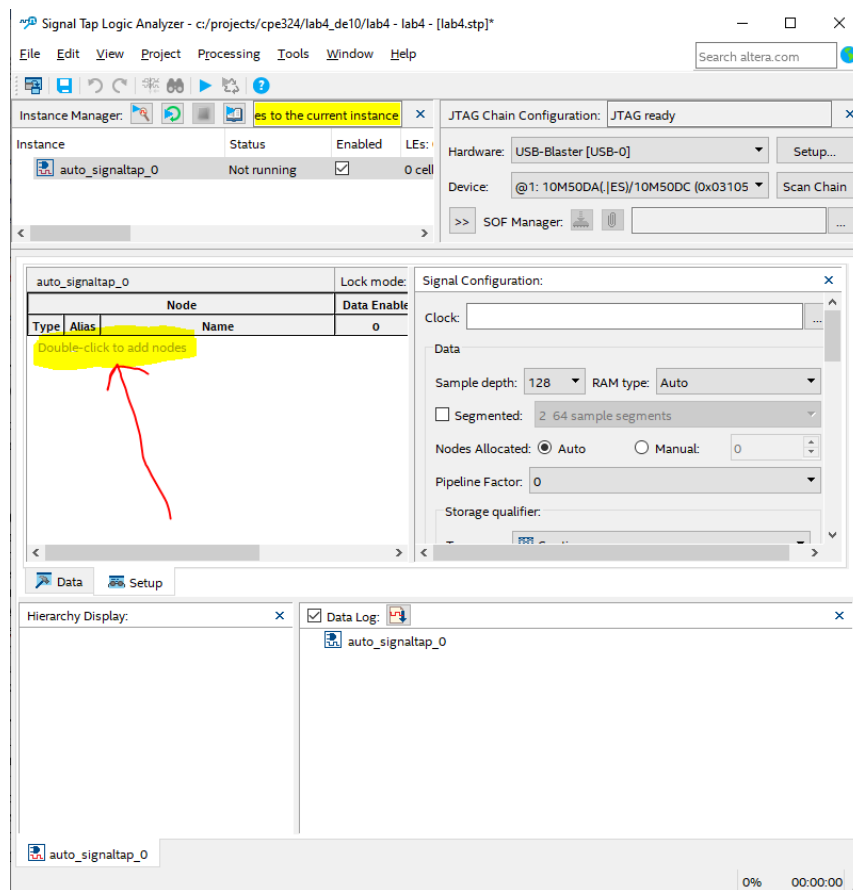


The screenshot shows the 'Compilation Report - lab4' window with the 'prbs16.v' file selected. The 'Table of Contents' pane on the left shows the 'TimeQuest Timing Analyzer' folder expanded, with 'Setup Summary' highlighted in red. The main pane displays the 'Flow Summary' table.

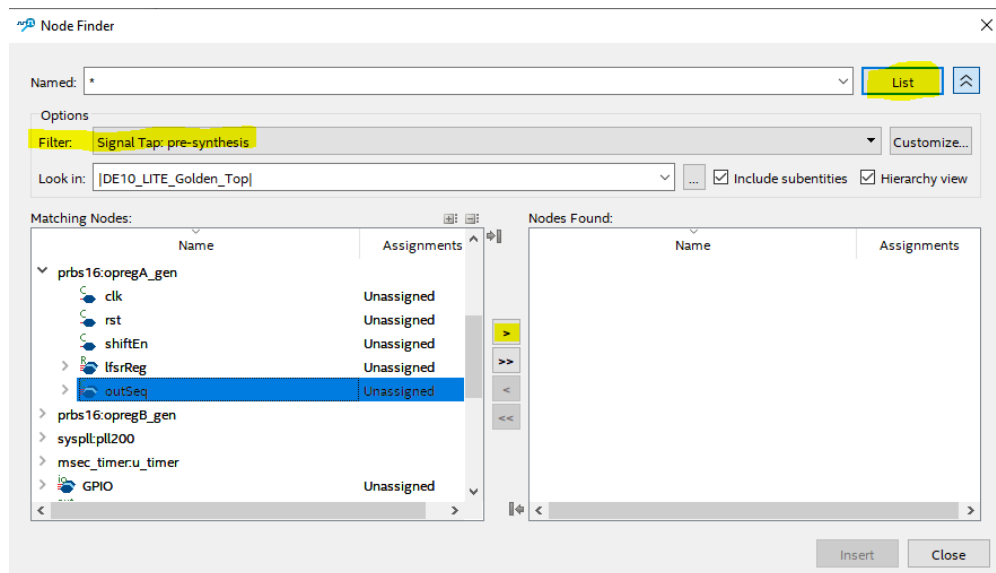
Flow Summary	
<<Filter>>	
Flow Status	Successful - Tue Feb 16 20:59:12 2021
Quartus Prime Version	17.1.0 Build 590 10/25/2017 SJ Lite Edition
Revision Name	lab4
Top-level Entity Name	DE10_LITE_Golden_Top
Family	MAX 10
Device	10M50DAF484C6GES
Timing Models	Preliminary
Total logic elements	483 / 49,760 (< 1 %)
Total registers	113
Total pins	107 / 360 (30 %)
Total virtual pins	0
Total memory bits	0 / 1,677,312 (0 %)
Embedded Multiplier 9-bit elements	2 / 288 (< 1 %)
Total PLLs	1 / 4 (25 %)
UFM blocks	0 / 1 (0 %)
ADC blocks	0 / 2 (0 %)

This timing failure must be addressed, but the next step to be performed is to investigate the behavior of the LFSR (prbs16) modules in the unit.

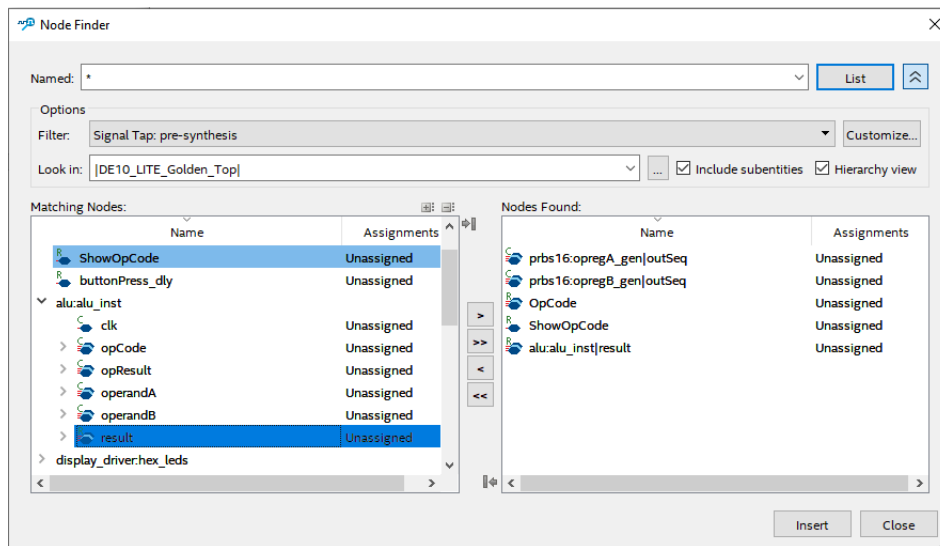
Select Tools > Signal Tap Logic Analyzer to launch its GUI, and also close the Programmer tool window prior to proceeding. In the left hand pane where it prompts, double-click the window to bring up a signal selection dialog:



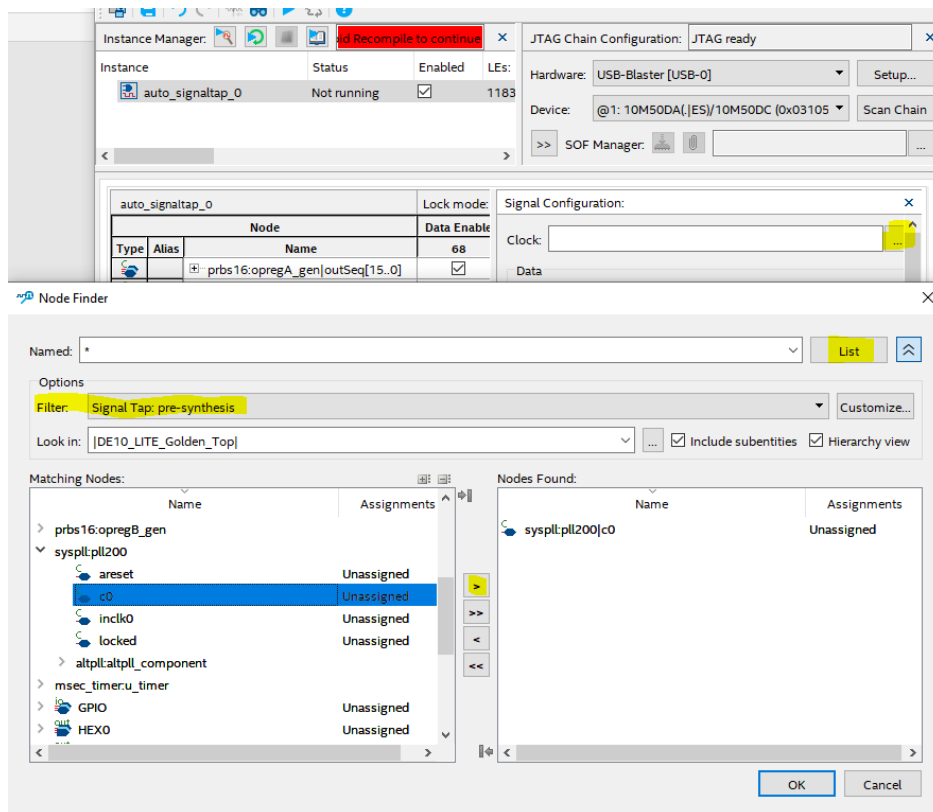
In the Node Finder window, select “Signal Tap: pre-synthesis” as the Filter option, then click the List button:



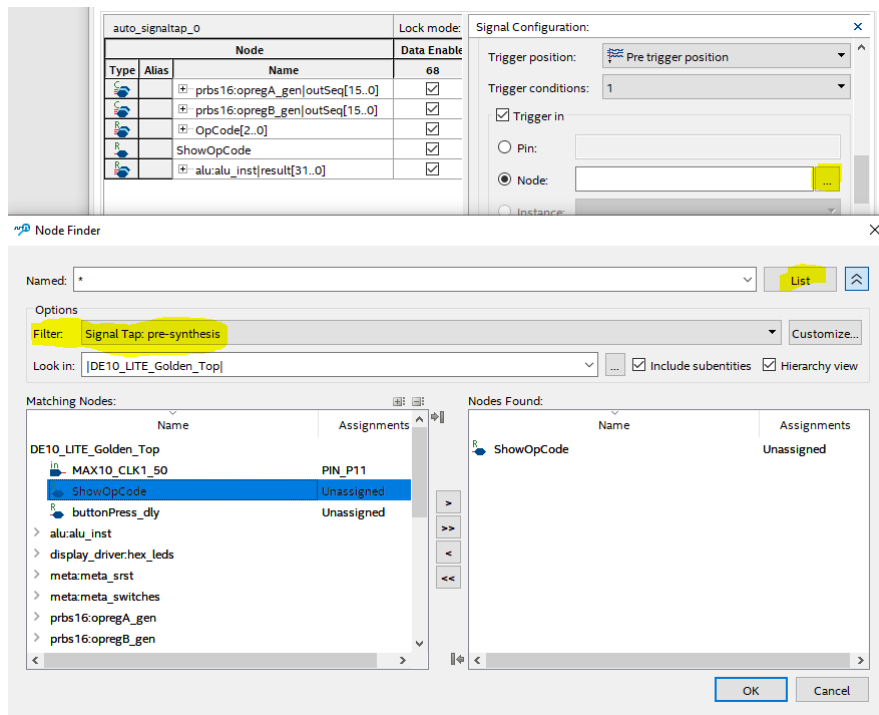
Start by selecting the outSeq signal from each of the prbs16 modules, and hitting the “>” button to put them into the list on the right. Scroll down and add OpCode to the list, and scroll back up to add ShowOpCode and finally the result output from the alu module:





Hit the Insert button and Close. On the right side of the Signal Tap GUI window, notice the “Signal Configuration” pane and click the “...” button next to the blank Clock: box.




Repeat the prior steps to select the c0 output from the syspll module as the sample clock for the logic analyzer. Finally, scroll down in the Signal Configuration pane to reach the Trigger configuration, checking Trigger in, and selecting Node, then using the “...” button:

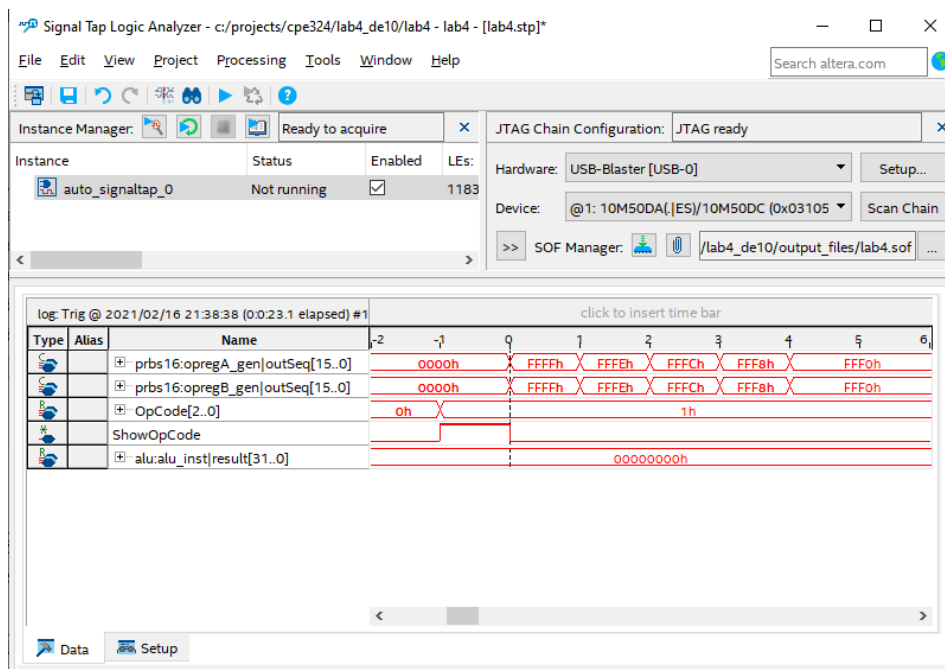


Choose ShowOpCode (at the top-level) as the trigger. Finally, scroll down further in the Signal Configuration pane and set the pattern to Rising Edge. Save the Signal Tap file (or Save-As), and enable it for the current project when prompted. Click on the  button to start compilation.

The Signal Tap GUI allows a user to program the FPGA directly (avoiding the Programmer GUI that was used in Labs 2 & 3). Once the FPGA compilation is complete, find the JTAG Chain Configuration window pane in the top-right of the Signal Tap window, and click the “...” to the right of the empty box. Find the lab4.sof file in the output_files folder, and choose it. Finally click the Program Device button .

Now the Signal Tap logic analyzer is set up to enable the user to capture trigger events based on the ShowOpCode signal, and will show the selected signals for 128 clock cycles. More clock cycles can be enabled for sample storage, but in this lab 128 is sufficient.

Begin by clicking the Run Analysis button . After clicking it, press the KEY[1] button on the board. This should trigger Signal Tap to capture the data collected on the board.



Right-clicking the waveform will zoom in and left-clicking will zoom out. Note that changes to the SW[7:0] switches will cause the outSeq values from the LFSRs to change and correspondingly the ALU's result output will change. However, clicking the KEY[0] pushbutton is required to update the waveform view.

At this point, please set the input SW[7:0] to any pair of different numbers, and go through a full range of OpCodes from 0 to 7. Note in your lab report the following:

- Settings for SW[7:4] and SW[3:0]
- LFSR 16-bit values corresponding to these two inputs
- 32-bit ALU result outputs for each OpCode value from 0 to 7
 - Do these computations all work out?
 - 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

Phase V: Fix Timing for the Multiplier

The timing failure is a concern because it limits the ability to propagate this design across different devices and boards. As a rule-of-thumb during debug, if a signal fails timing by less than 10%, it can still be trusted to run at room temperature in a lab environment and operate properly. And true, one board may be fast enough to run this design properly, but there's a point where timing failures definitely disrupt proper functionality.

Prior to attempting to fix timing, the student must first open the TimeQuest Timing Analyzer to verify the worst-case path that is failing timing. To do so, select Tools > TimeQuest Timing Analyzer. In that window, find the Tasks pane and double click Reports > Slack > Report Setup Summary. This shows a list of clocks and their slack (red/negative slack indicates failed timing). To report timing slack details, right-click the clock name associated with the 200 MHz syspll output,

then select Report Timing.

The Report Timing dialog box allows several options, the most important one is the number of paths to be reported (starting from worst case setup slack, then diminishing towards positive slack). Here simply select the default of 10, and click the Report Timing button. Selecting the worst-case path, look at the Data Path shown, which reveals multiple nsec of delay it takes to propagate through the multiplier:

Slow 1200mV 85C Model

Command Info		Summary of Paths										
	Slack	From Node		To Node		Launch Clock		Latch Clock		Relationship	Clock Skew	Data Delay
1	-1.482	prbs16.opregB_gen lfsrReg[7]		alu.alu_inst result[2]		pll200 altpll_component auto_generated pll1 clk[0]		pll200 altpll_component auto_generated pll1 clk[0]		5.000	-0.054	6.440
2	-1.480	prbs16.opregB_gen lfsrReg[4]		alu.alu_inst result[2]		pll200 altpll_component auto_generated pll1 clk[0]		pll200 altpll_component auto_generated pll1 clk[0]		5.000	-0.056	6.436
3	-1.466	prbs16.opregA_gen lfsrReg[1]		alu.alu_inst result[2]		pll200 altpll_component auto_generated pll1 clk[0]		pll200 altpll_component auto_generated pll1 clk[0]		5.000	-0.054	6.424
4	-1.446	prbs16.opregB_gen lfsrReg[13]		alu.alu_inst result[2]		pll200 altpll_component auto_generated pll1 clk[0]		pll200 altpll_component auto_generated pll1 clk[0]		5.000	-0.055	6.403
5	-1.446	prbs16.opregB_gen lfsrReg[14]		alu.alu_inst result[2]		pll200 altpll_component auto_generated pll1 clk[0]		pll200 altpll_component auto_generated pll1 clk[0]		5.000	-0.054	6.404
6	-1.443	prbs16.opregA_gen lfsrReg[2]		alu.alu_inst result[2]		pll200 altpll_component auto_generated pll1 clk[0]		pll200 altpll_component auto_generated pll1 clk[0]		5.000	-0.054	6.401
7	-1.441	prbs16.opregA_gen lfsrReg[10]		alu.alu_inst result[2]		pll200 altpll_component auto_generated pll1 clk[0]		pll200 altpll_component auto_generated pll1 clk[0]		5.000	-0.055	6.398

Path #1: Setup slack is -1.482 (VIOLATED)

Path Summary		Statistics	Data Path	Waveform			
Data Arrival Path							
	Total	Incr	RF	Type	Fanout	Location	Element
3	6.941	6.440					data path
1	0.695	0.194		uTco	1	FF_X51_Y32_N31	prbs16.opregB_gen lfsrReg[7]
2	0.695	0.000	FF	CELL	12	FF_X51_Y32_N31	opregB_gen lfsrReg[7]q
3	1.466	0.771	FF	IC	1	DSPMULT_X48_Y32_N0	alu_inst Mult0 auto_generated mac_mult1 data0[9]
4	5.073	3.607	FR	CELL	32	DSPMULT_X48_Y32_N0	alu_inst Mult0 auto_generated mac_mult1 dataout[35]
5	5.073	0.000	RR	IC	1	DSPOUT_X48_Y32_N2	alu_inst Mult0 auto_generated mac_out2 data0[6]
6	5.196	0.123	RF	CELL	1	DSPOUT_X48_Y32_N2	alu_inst Mult0 auto_generated mac_out2 dataout[6]
7	6.101	0.905	FF	IC	1	LCCOMB_X49_Y34_N10	alu_inst Mux29-3 data0
8	6.200	0.099	FF	CELL	1	LCCOMB_X49_Y34_N10	alu_inst Mux29-3 combout
9	6.423	0.223	FF	IC	1	LCCOMB_X49_Y34_N24	alu_inst Mux29-4 data0

Path #1: Setup slack is -1.482 (VIOLATED)

Path Summary		Statistics	Data Path	Waveform
Property		Value		
1	From Node	prbs16.opregB_gen lfsrReg[7]		
2	To Node	alu.alu_inst result[2]		
3	Launch Clock	pll200 altpll_component auto_generated pll1 clk[0]		
4	Latch Clock	pll200 altpll_component auto_generated pll1 clk[0]		
5	Data Arrival Time	6.941		
6	Data Required Time	5.459		
7	Slack	-1.482 (VIOLATED)		

Slow Required Path

	Total	Incr	RF	Type	Fanout	Location	Element
1	5.000	5.000					latch edge time
2	5.447	0.447					clock path
1	5.000	0.000					source latency
2	5.000	0.000			1	PIN_P11	MAX10_CLK1_50

	Slack	From Node	To Node	Launch Clock
1	-1.482	prbs16:opregB_gen lfsrReg[7]	alu:alu_inst[result[2]	pll200 altpll_component auto_gen
2	-1.480	prbs16:opregB_gen lfsrReg[4]	alu:alu_inst[result[2]	pll200 altpll_component auto_gen
3	-1.466	prbs16:opregA_gen lfsrReg[1]	alu:alu_inst[result[2]	pll200 altpll_component auto_gen
4	-1.446	prbs16:opregB_gen lfsrReg[13]	alu:alu_inst[result[2]	pll200 altpll_component auto_gen
5	-1.446	prbs16:opregB_gen lfsrReg[14]	alu:alu_inst[result[2]	pll200 altpll_component auto_gen
6	-1.443	prbs16:opregA_gen lfsrReg[2]	alu:alu_inst[result[2]	pll200 altpll_component auto_gen
7	-1.441	prbs16:opregA_gen lfsrReg[10]	alu:alu_inst[result[2]	pll200 altpll_component auto_gen

Path #1: Setup slack is -1.482 (VIOLATED)

Path Summary	Statistics	Data Path	Waveform			
Property	Value	Count	Total Delay	% of Total	Min	Max
1 Setup Relationship	5.000					
2 Clock Skew	-0.054					
3 Data Delay	6.440					
4 Number of Logic Levels	5					
5 Physical Delays						
1 Arrival Path						
1 Clock						
1 IC	4	7.261	86	0.000	3.221	
2 Cell	4	1.162	14	0.000	0.678	
3 PLL Compensation	1	-7.922	0	-7.922	-7.922	
2 Data						
1 IC	6	2.088	32	0.000	0.905	
2 Cell	7	4.158	65	0.000	3.607	
3 uTco	1	0.194	3	0.194	0.194	
2 Required Path						

The ALU Verilog code provided in this lab placed the operations and the multiplexer for the OpCode all prior to the final **result** register. While this helps minimize flip-flop usage, this adds an 8:1 multiplexer on the 32-bit output of the multiplier, exacerbating the worst-case path in the design. The following block diagram shows the way the ALU was originally coded:

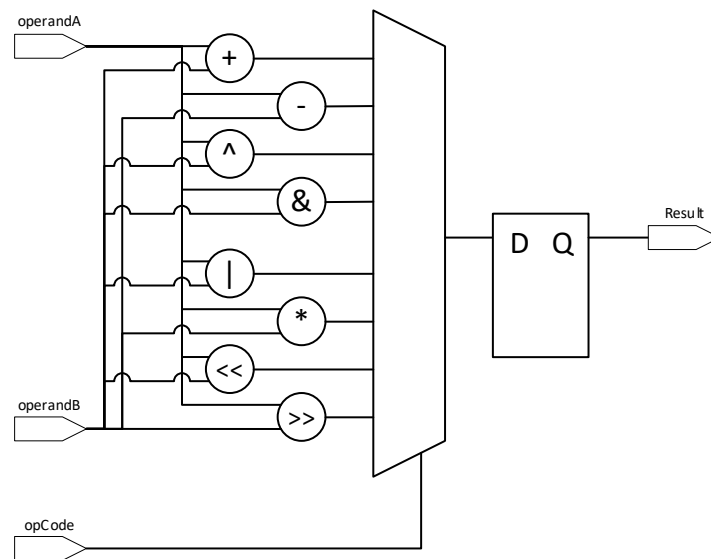


Figure 5: ALU, Output Registered

One method of fixing timing, called “cut-set retiming” is to move registers around to cut the critical path. Generally speaking, if one can draw a line through a feed-forward path, one can move registers from one side of the line to the other. Feed-forward paths exclude any that feed back from any register output back through combinational logic to the same register input. In this case, we draw the following line (Figure 6, left) through the output register, and around the mux to the input. Removing the flip-flops on the result output and placing them on the inputs to the multiplexer eliminates the 8:1 mux from the critical path. Each operation requires its own set of flip-flops, as does the OpCode input, but it retains the exact same behavior. Other methods to fix timing add

latency (thus changing behavior), or duplicate parts of the circuit and run each at a lower frequency.

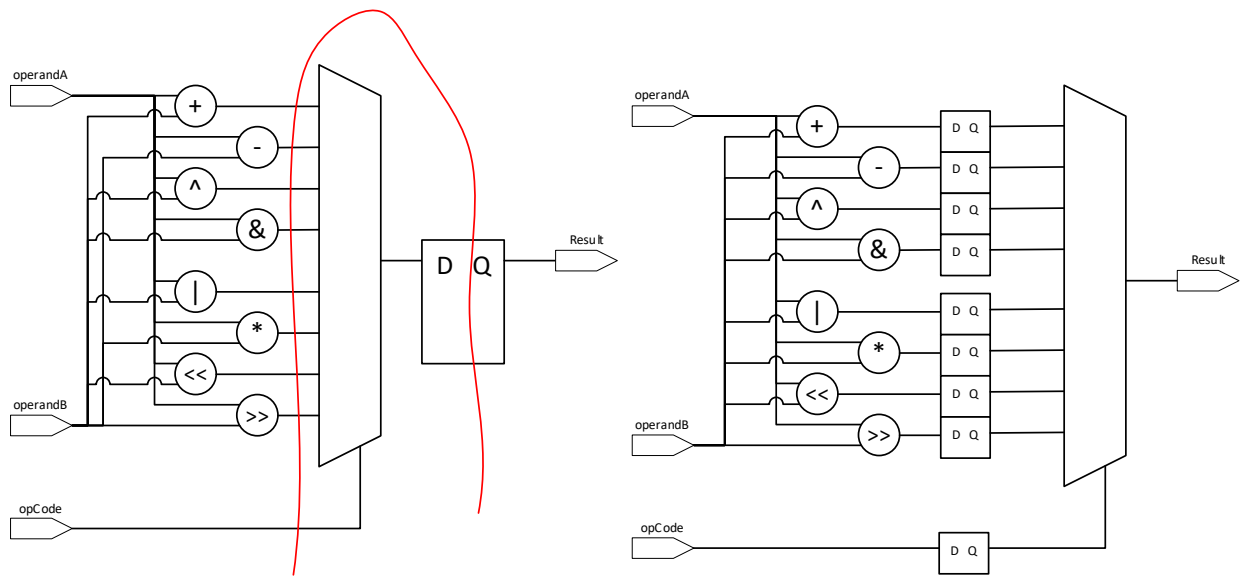


Figure 6: Cutset marked (Left), then Retimed by Moving Registers (Right)

Additional methods will be discussed in future lectures and labs.

Phase V requires you to re-code the alu.v file to move those registers to the path between each of the 8 operation assignments, plus the opCode input, and remove it from the output. After performing this modification, verify that the timing report is met.

Post-Lab Questions:

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?
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.
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?
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.

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				
1	4'hB	4'hD				
2	4'h5	4'hF				
3	4'h8	4'h1				
4	4'h2	4'h0				
5	4'hA	4'h9				
6	4'h6	4'h4				
7	4'hE	4'h3				
5	4'h6	4'h7				
1	4'hF	4'h0				