Bruno E. Gracia Villalobos & Bobby Reyes
EE 5193
Professor Eratne
16-Bit ALU on Nexys 4 DDR FPGA
July 22, 2019

PROBLEM

Design a 16-Bit ALU that takes inputs from switches on the Nexys-4 DDR board to assert 4 arithmetic operations and a reset. The 16-bit ALU must implement addition, subtraction, multiplication, and shift right logical operations. The ALU will use two operands stored in memory--both 16-bit in length--and perform the appropriate operation as selected by the switches on the FPGA. Results must be displayed as a 32 bit number in hex on the eight 7-segment displays onboard the Nexys-4 DDR FPGA and the current operation should be reflected by the LEDS.

ASM CHART

To start, an *Algorithmic State Machine Chart* (ASM) was used to describe the desired behavior of the ALU and the outputs at each state of operation. This chart assists in breaking down the design of the verilog modules to realize the design.

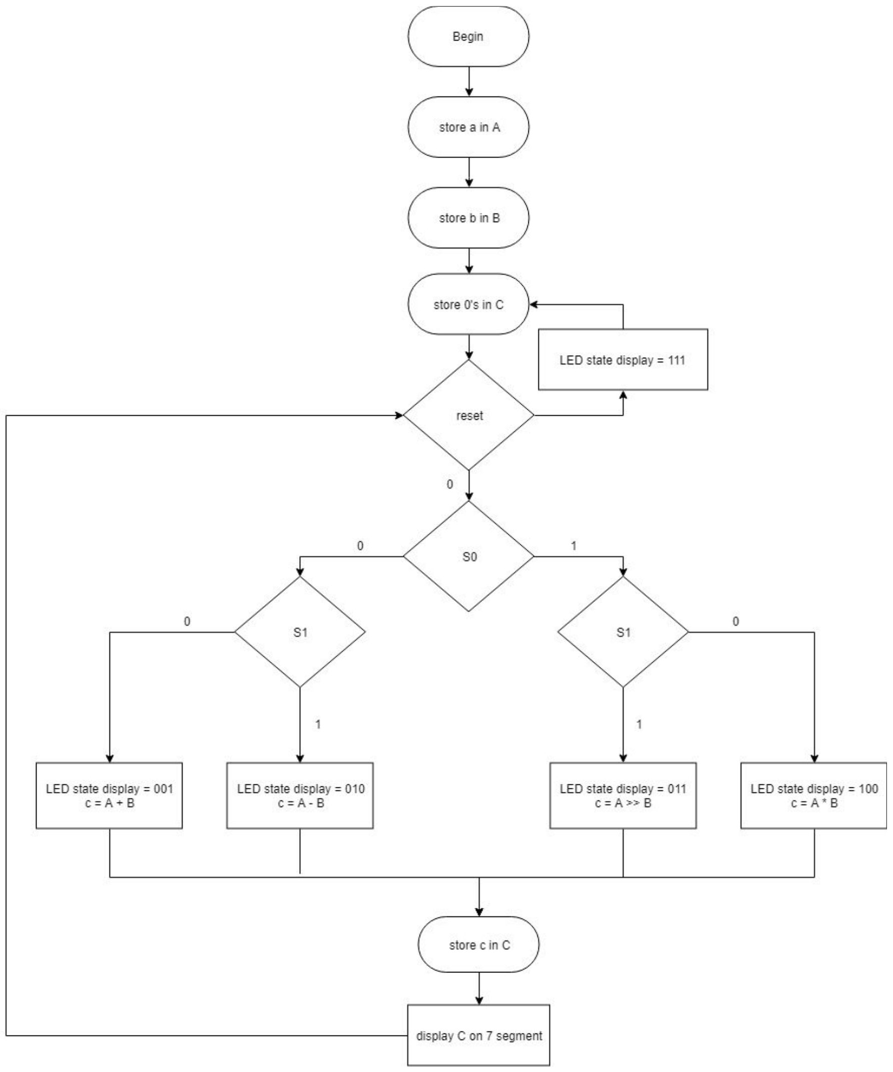
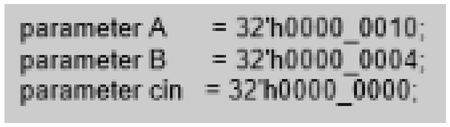

Begin
store a in A
store b in B
store 0's in C
LED state display = 111
reset
0
S0
0
1
S1
0
1
S1
0
1
LED state display = 001 c = A + B
LED state display = 010 c = A - B
LED state display = 011 c = A >> B
LED state display = 100 c = A * B
store c in C
display C on 7 segment

Figure 1 ASM Chart for 16-bit ALU

SOLUTION

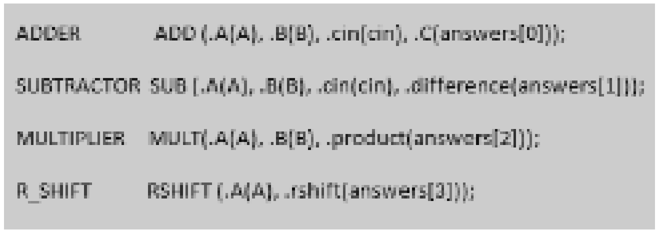
1st Step - Design a 16Bit ALU with Behavioral Model

In Verilog we designed the ALU with a top down approach. The ALU being the top module would perform using behavioral modeling to identify which operation the user wants to perform. A procedural block using case statements that depended on SW1 and SW2 would then select the output of each instantiated submodule and reset the output if SW4 was asserted. In the top module 2 16-bit variables A, and B were made as parameters to be used as the operands.



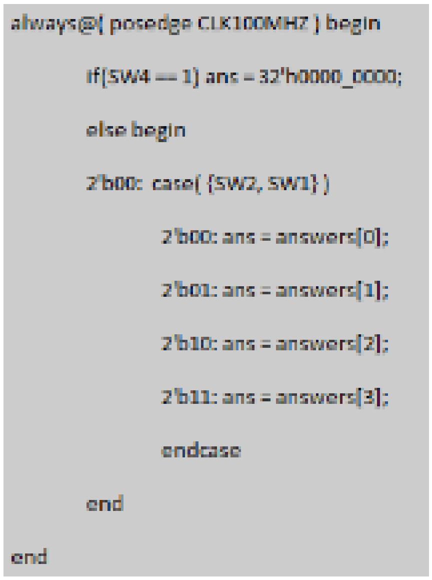
```
parameter A     = 32'h0000_0010;
parameter B     = 32'h0000_0004;
parameter cin   = 32'h0000_0000;
```

Figure 2 constant variables used as operands



```
ADDER          ADD (.A(A), .B(B), .cin(cin), .C(answers[0]));

SUBTRACTOR SUB (.A(A), .B(B), .cin(cin), .difference(answers[1]));

MULTIPLIER   MULT(.A(A), .B(B), .product(answers[2]));

R_SHIFT        RSHIFT (.A(A), .rshift(answers[3]));
```

Figure 3 Submodules for ALU operations instantiated in ALU16 top module



```
always@( posedge CLK100MHZ ) begin

        if(SW4 == 1) ans = 32'h0000_0000;

        else begin

        2'b00:  case( {SW2, SW1} )

                2'b00: ans = answers[0];

                2'b01: ans = answers[1];

                2'b10: ans = answers[2];

                2'b11: ans = answers[3];

                endcase

        end

end
```

Figure 4 Procedural Block in ALU16 top module for selecting operation

2nd Step - Modularize ALU Operations

Four different modules were written to represent the addition, subtraction, multiplication, and right shift operations of the ALU, and each of these modules was made using the native verilog operators that are available (+,-,*, >>). The operators were used to simplify the design of the submodules. Each of these submodules accepted two 16-bit operands A,B, and outputted a 32-bit result.

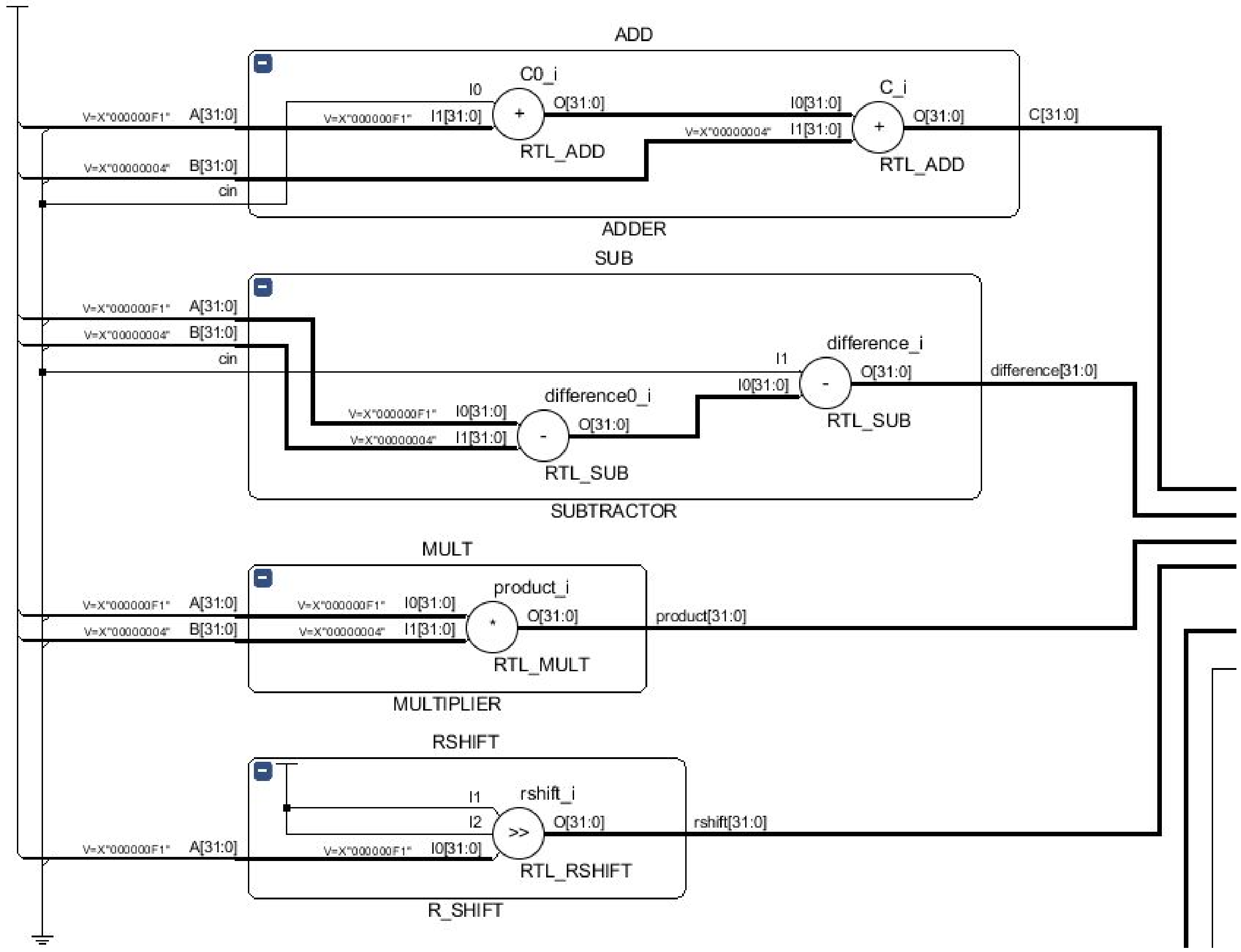


Figure 5 Synthesized block diagram of the 4 submodules used for ALU operations

3rd Step - ALU Simulation

To test the function of the 4 ALU submodules, a test bench was created to simulate the input from switches SW4, SW2, SW1 and verify that the correct ALU operation was being performed. Each switch was given the values respective to the different ALU operations. Performing these simulations verified that the ALU was performing as intended.

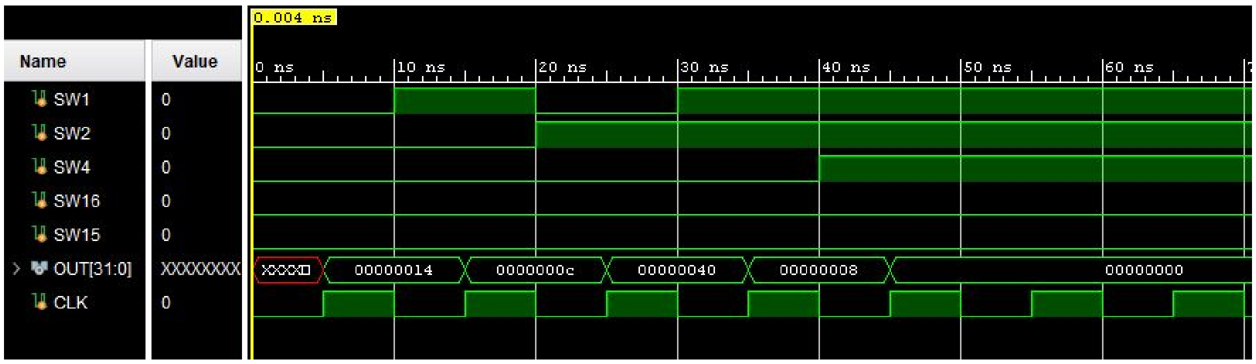


Figure 6 Shows the simulation waveform results from the ALU testbench

Time (ns)	**{SW4, SW2, SW1}**	**32 Bit Output**
5	000 (Addition)	0x0014
15	001 (Subtraction)	0x000C
25	010 (Multiplication)	0x0040
35	011 (Shift Right Logical)	0x0008
45	111 (Reset)	0x0000

Table 1 Shows the output from the simulation at their respective timestamps

4th Step - LED state display

The LEDs on the Nexys-4 DDR board needed to display in a way as to represent which operation is currently being performed by the ALU. The simplest way to accomplish this was to use a continuous assignment in the *ALU16* top module to assign LED[2:0] based on the state of the switches. A simple truth table was created with inputs (SW4, SW2, SW1) and outputs (LED[2], LED[1], LED[0]) to determine a boolean expression that would drive the LED's to behave in this way. The LED's would light based on the binary representation of each operation of the ALU (addition -1, subtraction -2, multiplication -3, shift -4), and all LEDs would light when SW 4 (reset) was asserted.

SW4	SW2	SW1	LED[2]	LED[1]	LED[0]
0	0	0	0	0	1
0	0	1	0	1	0
0	1	0	0	1	1
0	1	1	1	0	0
1	0	0	1	1	1
1	0	1	1	1	1
1	1	0	1	1	1
1	1	1	1	1	1

Table 2 truth table for desired LED behaviour

From the truth table three Kmaps were used to derive the minimum SOP representation for each LED output. From the Kmaps for the LED outputs we were able to make a continuous assignment for the LEDs in the *ALU* top module as shown in Figure 5. The expressions were continuously assigned using concatenation for simplicity.

LED[2]				
SW1/SW4,SW2	00	01	11	10
0	0	0	1	1
1	0	1	1	1

LED[1]				
SW1/SW4,SW2	00	01	11	10
0	0	1	1	1
1	1	0	1	1

LED[0]				
SW1/SW4,SW2	00	01	11	10
0	1	1	1	1
1	0	0	1	1

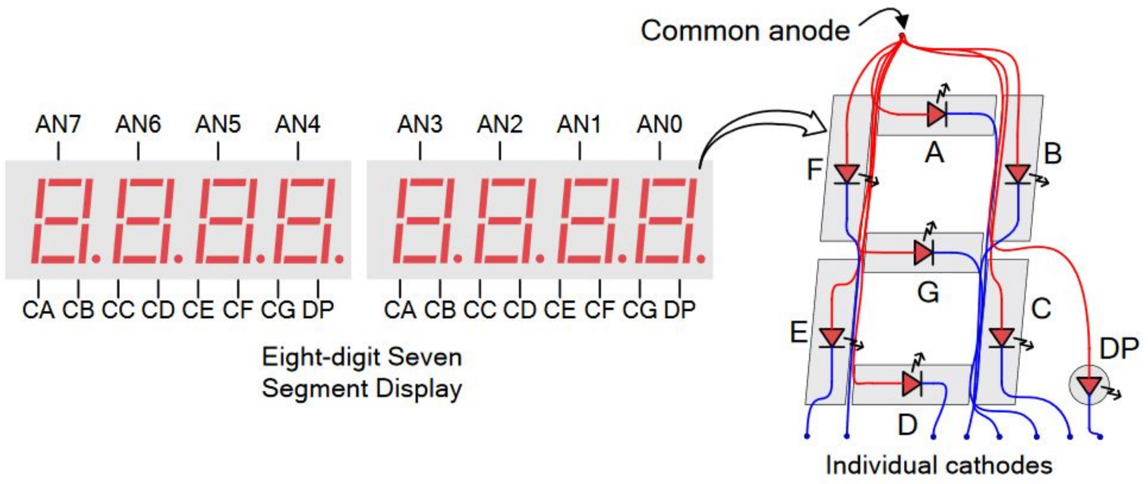
Table 3 Kmap Expressions for LEDs

```
assign LED[2:0] = {(SW4 | SW1 & SW2), (SW4 |SW4 & ~SW1 | SW1 & ~SW2) , (SW4 | ~SW1)};
```

Figure 7 Continuous assignment for LED[2:0] output

5th Step - Display ALU results to 7 segment Displays

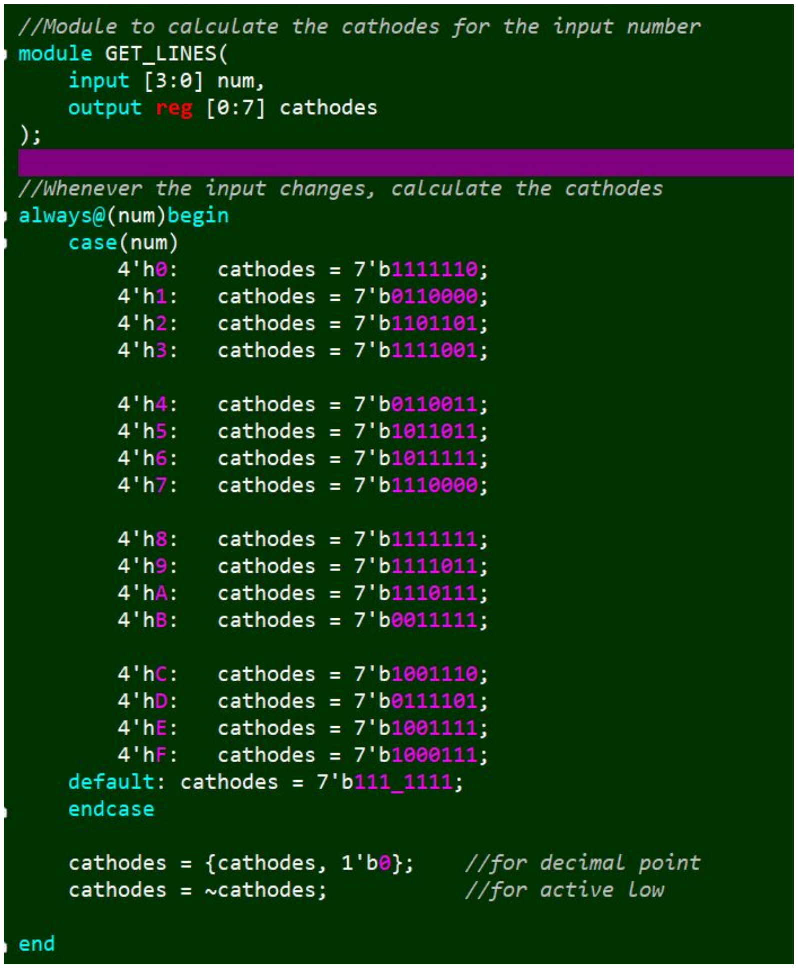
To display a 32 bit number in hex, we needed to take advantage of the eight 7-segment displays on the board. Two modules were written to handle the multiplexing SLOW_CLK and GET_LINES.



Figures 8 & 9 Eight-digit Seven Segment Displays (TOP), GET_LINES (BOTTOM)

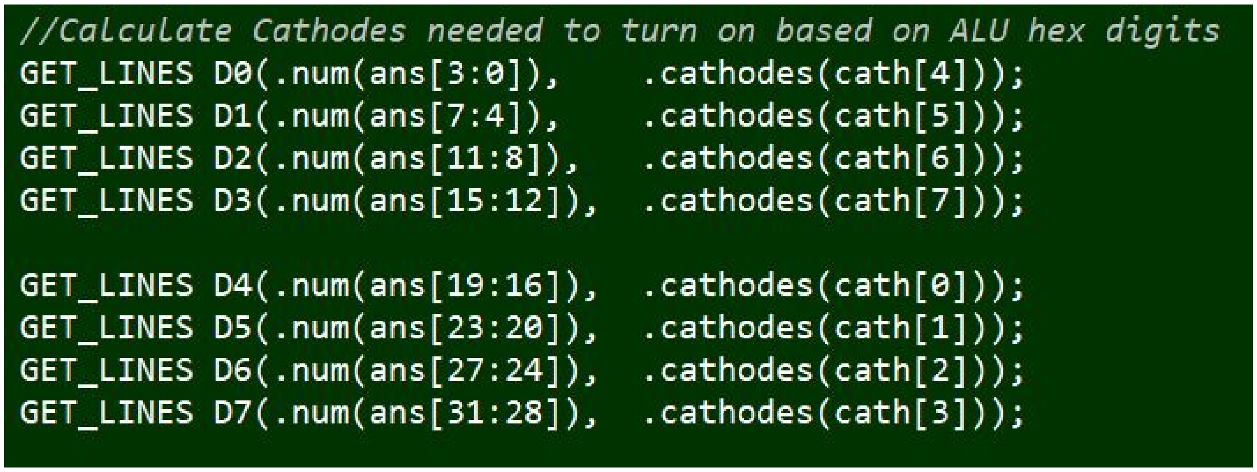
GET_LINES Module

To light up a single segment LED, the anode must be driven high and the cathode low. Figure 6 showcases the circuitry for the displays on the board. However, since four displays share the same anode for that particular segment, it is not possible to light four distinct digits on four displays at once. For this same reason, the displays must be "multiplexed", or rapidly switched on and off faster than the human eye can perceive to have the effect of being constantly on.



```
//Module to calculate the cathodes for the input number
module GET_LINES(
    input [3:0] num,
    output reg [0:7] cathodes
);

//Whenever the input changes, calculate the cathodes
always@(num)begin
    case(num)
        4'h0:   cathodes = 7'b1111110;
        4'h1:   cathodes = 7'b0110000;
        4'h2:   cathodes = 7'b1101101;
        4'h3:   cathodes = 7'b1111001;

        4'h4:   cathodes = 7'b0110011;
        4'h5:   cathodes = 7'b1011011;
        4'h6:   cathodes = 7'b1011111;
        4'h7:   cathodes = 7'b1110000;

        4'h8:   cathodes = 7'b1111111;
        4'h9:   cathodes = 7'b1111011;
        4'hA:   cathodes = 7'b1110111;
        4'hB:   cathodes = 7'b0011111;

        4'hC:   cathodes = 7'b1001110;
        4'hD:   cathodes = 7'b0111101;
        4'hE:   cathodes = 7'b1001111;
        4'hF:   cathodes = 7'b1000111;
    default: cathodes = 7'b111_1111;
    endcase

    cathodes = {cathodes, 1'b0};    //for decimal point
    cathodes = ~cathodes;           //for active low

end
```

Figure 7 showcases the GET_LINES module, which takes in a 4 bit number and outputs a bus of 8 bits representing the codes needed to light up the segments to form a hex digit. This module was instantiated 8 times (one for each 7-segment display) in the TOP ALU16 module to handle each of the 8 hex digits needed to represent the 32 bit output of the operation. These were stored in an array of 8 bit busses called "cath" for easier implementation as shown in Figure 10.



```
//Calculate Cathodes needed to turn on based on ALU hex digits
GET_LINES D0(.num(ans[3:0]),    .cathodes(cath[4]));
GET_LINES D1(.num(ans[7:4]),    .cathodes(cath[5]));
GET_LINES D2(.num(ans[11:8]),   .cathodes(cath[6]));
GET_LINES D3(.num(ans[15:12]),  .cathodes(cath[7]));

GET_LINES D4(.num(ans[19:16]),  .cathodes(cath[0]));
GET_LINES D5(.num(ans[23:20]),  .cathodes(cath[1]));
GET_LINES D6(.num(ans[27:24]),  .cathodes(cath[2]));
GET_LINES D7(.num(ans[31:28]),  .cathodes(cath[3]));
```

Figure 10 GET_LINES Instantiations in TOP Module

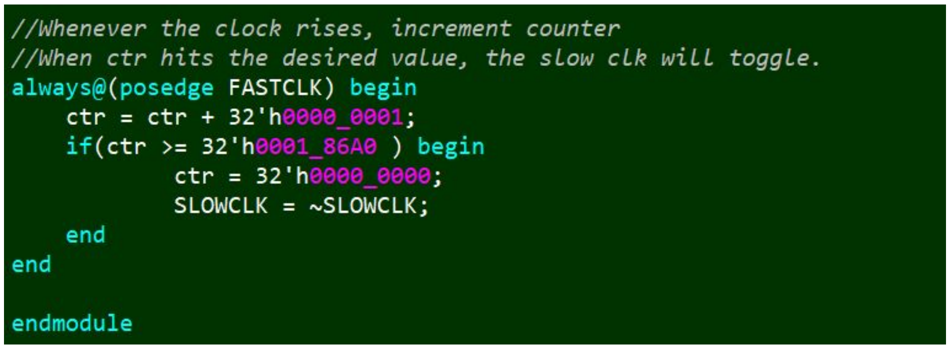
SLOW_CLK Module

With an onboard 100MHz clock, multiplexing at this speed would be too fast to see. A slower clock of about 1KHz is needed to rapidly switch on and off individual displays to have the "constantly on" effect. To achieve this, we wrote a module called SLOW_CLK, which takes in the fast 100MHz clock and divides it down to about 1KHz--the speed slow but fast enough clock speed needed to multiplex the displays.

A counter can be used to count "fast" clock ticks and only increment at a certain time to achieve the slower clock speed. For example, to get a 1KHz clock out of a 100KHz clock, we must count after certain "fast" clock ticks have passed called N. The formula below can be used N for a specified slower clock speed.

100MHz / Slower Clk = N

100MHz / 1KHz = 0x0001_86A0



```
//Whenever the clock rises, increment counter
//When ctr hits the desired value, the slow clk will toggle.
always@(posedge FASTCLK) begin
    ctr = ctr + 32'h0000_0001;
    if(ctr >= 32'h0001_86A0 ) begin
            ctr = 32'h0000_0000;
            SLOWCLK = ~SLOWCLK;
    end
end

endmodule
```

Figure 11 SLOW_CLK Procedural Block

In Figure 11, the module SLOW_CLK takes in the 100MHz Clock (FASTCLK) and a counter increments whenever the rising edge hits. When the counter hits N as formulated above(0x168A0), the SLOWCLK output register will toggle, therefore creating a slower clock.

With both SLOW_CLK and GET_LINES modules in place, the last step was to create a procedural block in the ALU TOP module to handle the multiplexing. First, the SLOW_CLK is instantiated to obtain the 1kHz. Next, temporary registers, regAN and regCATH are created to represent the displays needed to be turned on and the cathodes needed to be turned on respectively for the current hex digit display.

To multiplex the displays, a procedural block triggering on the posedge of the 1kHz clock assigns the turns on the corresponding cathodes and anode to the current hex digit using a circular shift-left method as shown in Figure 12. R0 is the first display on the board, and as the clock ticks, the turn on bit is shifted left, therefore displaying the next hex digit. This ensures only one 7-segment display is on at the same time, and therefore creating the illusion that all 8 displays are concurrently on.

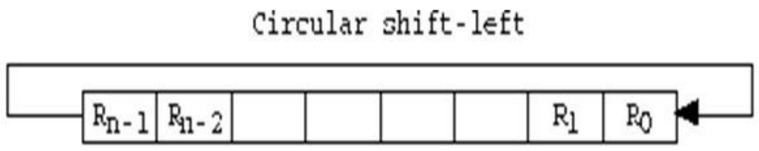
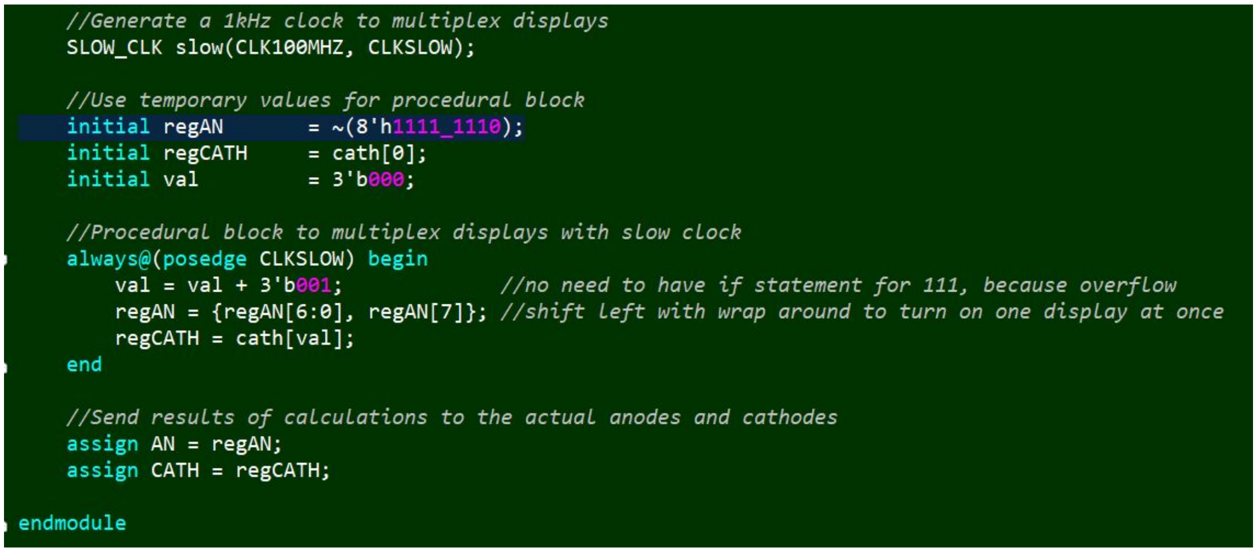


Figure 12   regAN 8 bit Bus implementation

In the procedural block, a 3 bit counter is used to select the hex digit to display from the 32 bit ALU output. For example, bits [3:0] of the ALU are stored in cath[0], and within the procedural block, the val counter is used to select the hex digit to display at the same time that the regAN on bit is being shifted left. In other words, as the current display is being multiplexed, the cathodes are too being selected to display the corresponding hex digit. Lastly, the AN, and CATH busses are assigned to with the regAN and regCATH register busses respectively to drive the current in the displays as shown in Figure 13.



```
    //Generate a 1kHz clock to multiplex displays
    SLOW_CLK slow(CLK100MHZ, CLKSLOW);

    //Use temporary values for procedural block
    initial regAN       = ~(8'h1111_1110);
    initial regCATH     = cath[0];
    initial val         = 3'b000;

    //Procedural block to multiplex displays with slow clock
    always@(posedge CLKSLOW) begin
        val = val + 3'b001;             //no need to have if statement for 111, because overflow
        regAN = {regAN[6:0], regAN[7]}; //shift left with wrap around to turn on one display at once
        regCATH = cath[val];
    end

    //Send results of calculations to the actual anodes and cathodes
    assign AN = regAN;
    assign CATH = regCATH;

endmodule
```

Figure 13 Eight digit 7 segment display logic

CONCLUSION

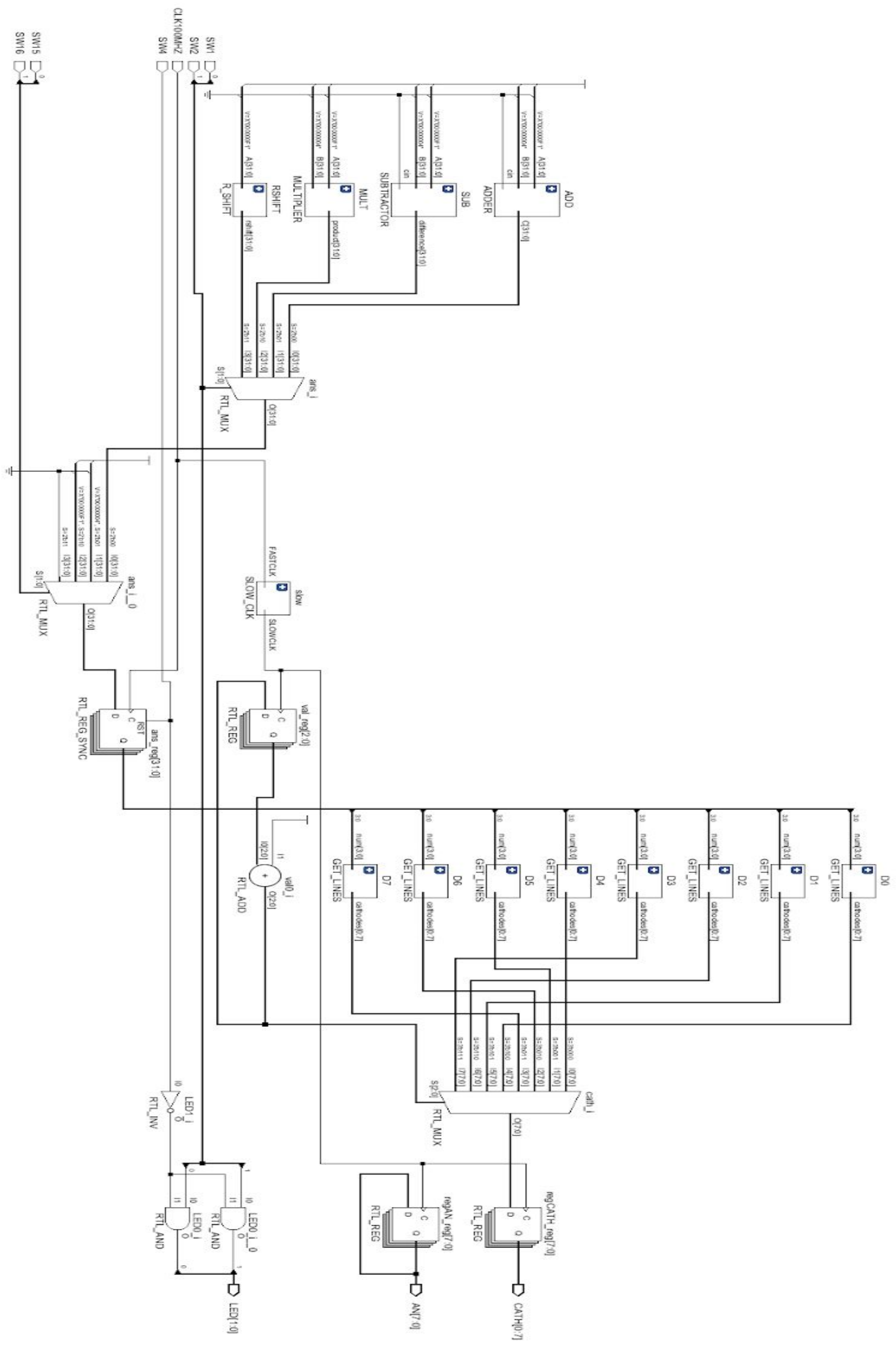
The 16-bit ALU implementation on the Nexys-4 DDR board strengthened our design, simulation, and synthesis skills. At times, there were errors that prevented the synthesis of code, and these provided valuable learning information. For example, one time the output of the ALU was driven by multiple registers and the implementation failed. The authors realized this was due to latched behavior for non-consistent case statements and solved the problem by having a single procedural block for asserting conditions.

During the simulation phase, we also encountered issues with the output of the ALU being either unknown (X) or hi-Z (Z). We learned to double-check the ports of the unit under test (UUT) to make sure they were all properly accounted for.

Another problem came up when constraints were not properly defined, and this prevented the implementation phase from completing successfully. We solved this by making sure each constraint had a corresponding port in the TOP module**.**

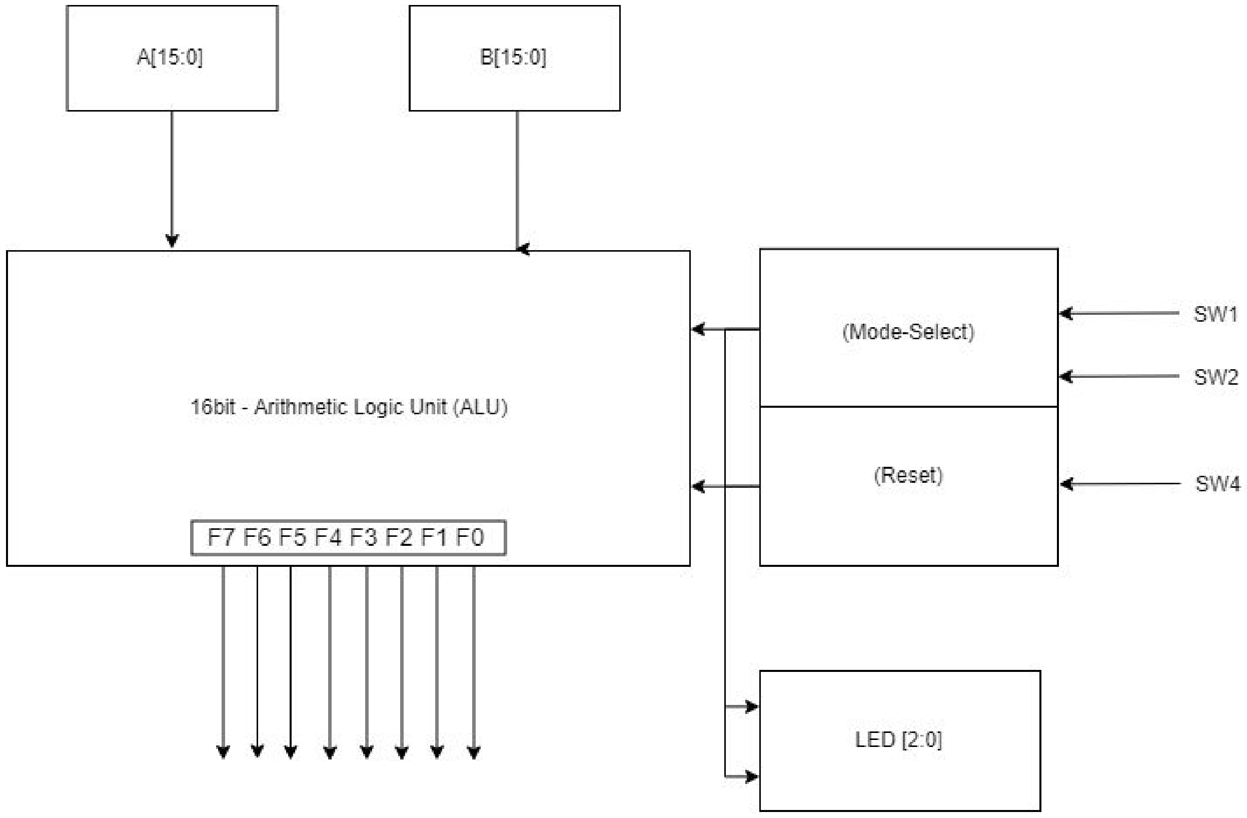
Designing the SLOW_CLK was also an issue. The first test had the upper 4 hex digits in the lower position, and the lower 4 hex digits in the upper position. The exact cause for this is unknown to the authors but we just rearranged the cath[0] to cath[3] in place of cath[4] to cath[7]. The second test had the displays blinking, and the digits were not constantly lit up. A higher slow clock speed (1kHz) was selected to solve this problem.

In future implementations, we would like to have an option for changing the predefined ALU operands. We could use the switches to input a new number and this would increase the usefulness of the project. In the end, we see this project can be a great way to design and test an ALU for a processor.

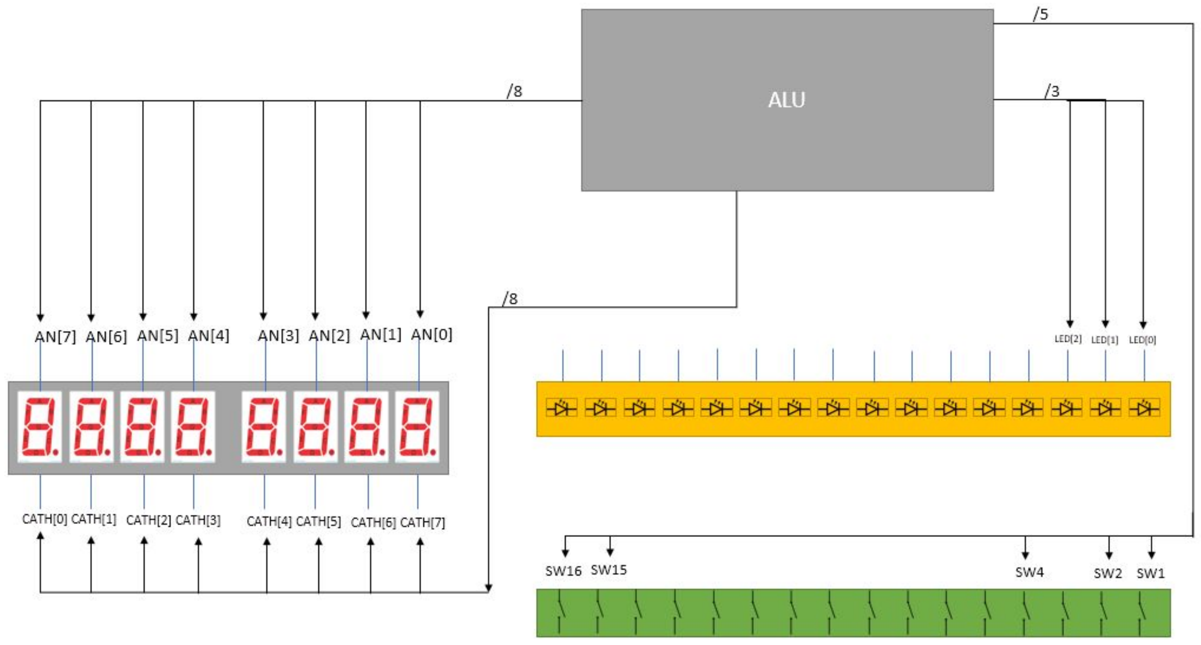

SW1
SW2
CLK100MHZ
SW4
SW15
SW16
ADD
ADDER
SUB
SUBTRACTOR
MULT
MULTIPLIER
RSHIFT
R_SHIFT
ans_i
RTL_MUX
ans_i_0
RTL_MUX
slow
FASTCLK
SLOW_CLK
SLOWCLK
ans_reg[31:0]
RTL_REG_SYNC
val_reg[2:0]
RTL_REG
val0_i
RTL_ADD
D0
D1
D2
D3
D4
D5
D6
D7
GET_LINES
cath_i
RTL_MUX
LED1_i
RTL_INV
LED0_i
LED0_i_0
RTL_AND
regAN_reg[7:0]
regCATH_reg[7:0]
RTL_REG
LED[1:0]
AN[7:0]
CATH[0:7]

BLOCK DIAGRAM

ALU Internals



ALU Externals


ALU
/5
/3
/8
/8
AN[7] AN[6] AN[5] AN[4] AN[3] AN[2] AN[1] AN[0]
CATH[0] CATH[1] CATH[2] CATH[3] CATH[4] CATH[5] CATH[6] CATH[7]
LED[2] LED[1] LED[0]
SW16 SW15 SW4 SW2 SW1

VERILOG CODE

```verilog
`timescale 1ns / 1ps

module ALU16
(
        input   SW1, SW2, SW4, SW15, SW16, //SW15 displays B operand, SW16 displays A operand
        input   CLK100MHZ,
        output  [1:0] LED,
        output  [0:7] CATH,
        output  [7:0] AN
        //output  ans
);
        // This is the current output of the ALU operation for 7SEG Displays
        reg [31:0] ans;

        // This is a memory register to store all ALU operations
        // answers[0] == adder output
        // answers[1] == sub output
        // answers[2] == mult output
        // answers[3] == rshift output
        wire [31:0] answers [3:0];

        // Predefined operands for ALU
        parameter A      = 32'h0000_0010;
        parameter B      = 32'h0000_0004;
        parameter cin   = 32'h0000_0000;

        // Showcase the ALU operation with LED's
        assign LED[1:0] = {SW2 & ~SW4, SW1 & ~SW4}; //check if reset isnt high

        //Instantiate ALU Modules
        ADDER              ADD(.A(A), .B(B), .cin(cin), .C(answers[0]));
        SUBTRACTOR         SUB(.A(A), .B(B), .cin(cin), .difference(answers[1]));
        MULTIPLIER         MULT(.A(A), .B(B), .product(answers[2]));
        R_SHIFT            RSHIFT(.A(A), .rshift(answers[3]));

        // Always block to check for ALU opcodes
        always@( posedge CLK100MHZ ) begin
        if(SW4 == 1) ans = 32'h0000_0000;
        else begin
        case( {SW16, SW15} )
        //if SW16 and 15 are low, then resume normal ALU operation
        2'b00:  case( {SW2, SW1} )
                2'b00: ans = answers[0];
                2'b01: ans = answers[1];
                2'b10: ans = answers[2];
                2'b11: ans = answers[3];
```

```
            endcase
//Display ALU operands for respective SW16 == A or SW15 == B
2'b01:  ans = B;
2'b10:  ans = A;
2'b11:  ans = 32'h0;
endcase
end
end

// Handle display of numbers
// [size] name [index]
wire [0:7] cath [0:7];
wire CLKSLOW;

//Temporary registers for multiplexing procedural block
reg [7:0] regAN;
reg [0:7] regCATH;
reg [2:0] val;

//Calculate Cathodes needed to turn on based on ALU hex digits
GET_LINES D0(.num(ans[3:0]), .cathodes(cath[4]));
GET_LINES D1(.num(ans[7:4]), .cathodes(cath[5]));
GET_LINES D2(.num(ans[11:8]),   .cathodes(cath[6]));
GET_LINES D3(.num(ans[15:12]),  .cathodes(cath[7]));

GET_LINES D4(.num(ans[19:16]),  .cathodes(cath[0]));
GET_LINES D5(.num(ans[23:20]),  .cathodes(cath[1]));
GET_LINES D6(.num(ans[27:24]),  .cathodes(cath[2]));
GET_LINES D7(.num(ans[31:28]),  .cathodes(cath[3]));

//Generate a 1kHz clock to multiplex displays
SLOW_CLK slow(CLK100MHZ, CLKSLOW);

//Use temporary values for procedural block
initial regAN      = ~(8'h1111_1110);
initial regCATH  = cath[0];
initial val           = 3'b000;

//Procedural block to multiplex displays with slow clock
always@(posedge CLKSLOW) begin
val = val + 3'b001;         //no need to have if statement for 111, because overflow
regAN = {regAN[6:0], regAN[7]}; //shift left with wrap around to turn on one display at once
regCATH = cath[val];
end

//Send results of calculations to the actual anodes and cathodes
assign AN = regAN;
assign CATH = regCATH;
```

```
endmodule

//Module to generate a slow clock around 1kHz to multiplex displays
module SLOW_CLK(
          input FASTCLK,
          output reg SLOWCLK
);
//  05F5_E100 is 100M
//  0001_86A0 is 1K
//  02FA_F080? is 50M
//  0015_CC5B count for 70 Hz
//  000F_4240 is 1M
//  ?0003_2DCD? is 480 Hz
//  000?3_0D40? is 500 Hz
// need to have 1ms clock, or 1000Hz for each 7seg.

//Use a 32 bit register for storing counter values
reg [31:0] ctr;
initial ctr = 32'h0000_0000;
initial SLOWCLK = 0;

//Whenever the clock rises, increment counter
//When ctr hits the desired value, the slow clk will toggle.
always@(posedge FASTCLK) begin
          ctr = ctr + 32'h0000_0001;
          if(ctr >= 32'h0001_86A0 ) begin
          ctr = 32'h0000_0000;
          SLOWCLK = ~SLOWCLK;
          end
end

endmodule


//Module to calculate the cathodes for the input number
module GET_LINES(
          input [3:0] num,
          output reg [0:7] cathodes
);

//Whenever the input changes, calculate the cathodes
always@(num)begin
          case(num)
          4'h0:   cathodes = 7'b1111110;
          4'h1:   cathodes = 7'b0110000;
          4'h2:   cathodes = 7'b1101101;
          4'h3:   cathodes = 7'b1111001;
```

```
		4'h4:   cathodes = 7'b0110011;
		4'h5:   cathodes = 7'b1011011;
		4'h6:   cathodes = 7'b1011111;
		4'h7:   cathodes = 7'b1110000;

		4'h8:   cathodes = 7'b1111111;
		4'h9:   cathodes = 7'b1111011;
		4'hA:   cathodes = 7'b1110111;
		4'hB:   cathodes = 7'b0011111;

		4'hC:   cathodes = 7'b1001110;
		4'hD:   cathodes = 7'b0111101;
		4'hE:   cathodes = 7'b1001111;
		4'hF:   cathodes = 7'b1000111;
		default: cathodes = 7'b111_1111;
		endcase

		cathodes = {cathodes, 1'b0};      //for decimal point
		cathodes = ~cathodes;             //for active low

end

endmodule

//Module to add two 16-bit numbers with carry in
module ADDER(
		input [31:0] A, B,
		input cin,
		output [31:0] C
);

assign C = cin + A + B;

endmodule

//Module to subtract two 16 bit numbers with carry in
module SUBTRACTOR(
		input [31:0] A, B,
		input cin,
		output [31:0] difference
);

assign difference = A - B - cin;

endmodule

//Module to multiply two 16-bit numbers
```

```
module MULTIPLIER(
	input [31:0] A, B,
	output [31:0] product
);

assign product = A * B;

endmodule

//Module for logical shift right one bit number A
module R_SHIFT(
	input [31:0] A,
	output [31:0] rshift
);

assign rshift = A >> 1;

endmodule
```

CONSTRAINTS

```
set_property -dict { PACKAGE_PIN E3   IOSTANDARD LVCMOS33 } [get_ports { CLK100MHZ }];
#IO_L12P_T1_MRCC_35 Sch=clk100mhz
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports {CLK100MHZ}];

set_property -dict { PACKAGE_PIN J15   IOSTANDARD LVCMOS33 } [get_ports { SW1 }];
#IO_L24N_T3_RS0_15 Sch=sw[0]
set_property -dict { PACKAGE_PIN L16   IOSTANDARD LVCMOS33 } [get_ports { SW2 }];
#IO_L3N_T0_DQS_EMCCLK_14 Sch=sw[1]
set_property -dict { PACKAGE_PIN R15   IOSTANDARD LVCMOS33 } [get_ports { SW4 }];
#IO_L13N_T2_MRCC_14 Sch=sw[3]

set_property -dict { PACKAGE_PIN U11   IOSTANDARD LVCMOS33 } [get_ports { SW15 }];
#IO_L19N_T3_A09_D25_VREF_14 Sch=sw[14]
set_property -dict { PACKAGE_PIN V10   IOSTANDARD LVCMOS33 } [get_ports { SW16 }];
#IO_L21P_T3_DQS_14 Sch=sw[15]

set_property -dict { PACKAGE_PIN H17   IOSTANDARD LVCMOS33 } [get_ports { LED[0] }];
#IO_L18P_T2_A24_15 Sch=led[0]
set_property -dict { PACKAGE_PIN K15   IOSTANDARD LVCMOS33 } [get_ports { LED[1] }];
#IO_L24P_T3_RS1_15 Sch=led[1]

set_property -dict { PACKAGE_PIN T10   IOSTANDARD LVCMOS33 } [get_ports { CATH[0] }];
#IO_L24N_T3_A00_D16_14 Sch=ca
set_property -dict { PACKAGE_PIN R10   IOSTANDARD LVCMOS33 } [get_ports { CATH[1] }];
#IO_25_14 Sch=cb
set_property -dict { PACKAGE_PIN K16   IOSTANDARD LVCMOS33 } [get_ports { CATH[2] }];
#IO_25_15 Sch=cc
set_property -dict { PACKAGE_PIN K13   IOSTANDARD LVCMOS33 } [get_ports { CATH[3] }];
#IO_L17P_T2_A26_15 Sch=cd
set_property -dict { PACKAGE_PIN P15   IOSTANDARD LVCMOS33 } [get_ports { CATH[4] }];
#IO_L13P_T2_MRCC_14 Sch=ce
set_property -dict { PACKAGE_PIN T11   IOSTANDARD LVCMOS33 } [get_ports { CATH[5] }];
#IO_L19P_T3_A10_D26_14 Sch=cf
set_property -dict { PACKAGE_PIN L18   IOSTANDARD LVCMOS33 } [get_ports { CATH[6] }];
#IO_L4P_T0_D04_14 Sch=cg

set_property -dict { PACKAGE_PIN H15   IOSTANDARD LVCMOS33 } [get_ports { CATH[7] }];
#IO_L19N_T3_A21_VREF_15 Sch=dp

set_property -dict { PACKAGE_PIN J17   IOSTANDARD LVCMOS33 } [get_ports { AN[0] }];
#IO_L23P_T3_FOE_B_15 Sch=an[0]
set_property -dict { PACKAGE_PIN J18   IOSTANDARD LVCMOS33 } [get_ports { AN[1] }];
#IO_L23N_T3_FWE_B_15 Sch=an[1]
set_property -dict { PACKAGE_PIN T9   IOSTANDARD LVCMOS33 } [get_ports { AN[2] }];
#IO_L24P_T3_A01_D17_14 Sch=an[2]
set_property -dict { PACKAGE_PIN J14   IOSTANDARD LVCMOS33 } [get_ports { AN[3] }];
#IO_L19P_T3_A22_15 Sch=an[3]
```

```
set_property -dict { PACKAGE_PIN P14   IOSTANDARD LVCMOS33 } [get_ports { AN[4] }];
#IO_L8N_T1_D12_14 Sch=an[4]
set_property -dict { PACKAGE_PIN T14   IOSTANDARD LVCMOS33 } [get_ports { AN[5] }];
#IO_L14P_T2_SRCC_14 Sch=an[5]
set_property -dict { PACKAGE_PIN K2   IOSTANDARD LVCMOS33 } [get_ports { AN[6] }];
#IO_L23P_T3_35 Sch=an[6]
set_property -dict { PACKAGE_PIN U13   IOSTANDARD LVCMOS33 } [get_ports { AN[7] }];
#IO_L23N_T3_A02_D18_14 Sch=an[7]
```

TESTBENCH

```
`timescale 1ns / 1ps

module TB();
reg SW1, SW2, SW4, SW16, SW15;
wire [31:0] OUT;
reg CLK;          //for simulating 100MHz clock

initial CLK = 0;
always #5 CLK = ~ CLK; //100MHZ CLK

//SLOW_CLK clk(CLK, SCLK);
ALU16 UUT(.CLK100MHZ(CLK), .SW1(SW1), .SW2(SW2), .SW4(SW4),
.SW16(SW16), .SW15(SW15), .ans(OUT));

initial begin
        {SW16, SW15, SW4}    = 3'b000;

        {SW2, SW1}     = 2'b00; //ADD
        #10 {SW2, SW1}  = 2'b01; //SUBTRACT
        #10 {SW2, SW1}  = 2'b10; //MULT
        #10 {SW2, SW1}  = 2'b11; //RSHIFT
        #10 SW4         = 1'b1;  //RESET

end
```
Endmodule

TERMINATING SHEET

Bobby Work	**Bruno Work**
ALU Modules (ADDITION, SUBTRACTION, MULTIPLICATION, R_SHIFT)	7 Segment Displays (GET_LINES, SLOW_CLK) Modules
Testbench simulation	3rd Step ALU simulation
LED State Display	4th Step Report and Conclusion
ASM Chart	ALU Externals Block Diagram
ALU Internals Block Diagram	ALU TOP Module
ALU TOP Module	ALU Schematic
1st, 2nd, 3rd steps of solution section - Report	1st, 2nd, 3rd steps of solution section - Report