BCIT

LAB 4 – ADC INTERFACE

Nicholas Scott AKA “White Cheddar”
A01255181 | FEB. 10TH, 2024

Contents

Table of Figures

“Ninety-five percent of people who pass my test bench get it working the first time.”

- *Sweet Bobby T*

‘adcinterface.sv’

After much hardship, I settled on the following code for my ADC interface module:

```
// ELEX 7660 Lab 4
// Nicholas Scott AKA "White Cheddar"
// A01255181
// Feb. 4th, 2024
// Instructor: Sweet Bobby T

module adcinterface(
    input logic clk, reset_n,   // clock and reset
    input logic [2:0] chan,     // ADC channel to sample
    output logic [11:0] result, // ADC result

    // ltc2308 signals
    output logic ADC_CONVST, ADC_SCK, ADC_SDI,
    input logic ADC_SDO
);

    logic [5:0] word; // word which tells ADC which channel we want
    logic [3:0] count; // a count bit that counts
    logic [11:0] tempResult; // a temp variable to store result

    always_comb begin

        // activate ADC clock for 12 clock cycles
        ADC_SCK = ((count >= 2) && (count <= 13)) ? clk : 1'b0;

        // Set config word based on chan input
        case (chan)
            0 : word = 6'b100010;
            1 : word = 6'b110010;
            2 : word = 6'b100110;
            3 : word = 6'b110110;
            4 : word = 6'b101010;
            5 : word = 6'b111010;
            6 : word = 6'b101110;
```

```
            7 : word = 6'b111110;
        endcase
    end

    always_ff @( negedge clk or negedge reset_n ) begin
        if (~reset_n)
            count <= 0;
        else
            count <= count + 1;  // automatically rolls over at 16

    end

    always_ff @( negedge clk or negedge reset_n ) begin
        if (~reset_n) begin
            ADC_CONVST <= 1;
            ADC_SDI <= 0;
        end
        else begin
            case (count)
                0 : ADC_CONVST <= 0; // conversion start signal
                // send config word one bit at a time
                1 : ADC_SDI <= word[5];
                2 : ADC_SDI <= word[4];
                3 : ADC_SDI <= word[3];
                4 : ADC_SDI <= word[2];
                5 : ADC_SDI <= word[1];
                6 : ADC_SDI <= word[0];
                15 : ADC_CONVST <= 1; // conversion end
                default : ADC_SDI <= 0;
            endcase
        end
    end

    always_ff @(posedge clk or negedge reset_n)
        if(~reset_n)
            result <= 0;
        else
            case (count)
                // receive ADC result one bit at a time
                2 : tempResult[11] <= ADC_SDO;
                3 : tempResult[10] <= ADC_SDO;
                4 : tempResult[9] <= ADC_SDO;
                5 : tempResult[8] <= ADC_SDO;
                6 : tempResult[7] <= ADC_SDO;
                7 : tempResult[6] <= ADC_SDO;
```

```
            8 : tempResult[5] <= ADC_SDO;
            9 : tempResult[4] <= ADC_SDO;
            10 : tempResult[3] <= ADC_SDO;
            11 : tempResult[2] <= ADC_SDO;
            12 : tempResult[1] <= ADC_SDO;
            13 : tempResult[0] <= ADC_SDO;
            14 : result <= tempResult; // assign result
            default: result <= result;
        endcase

endmodule
```

This code yielded the following simulation waveforms and transcript:

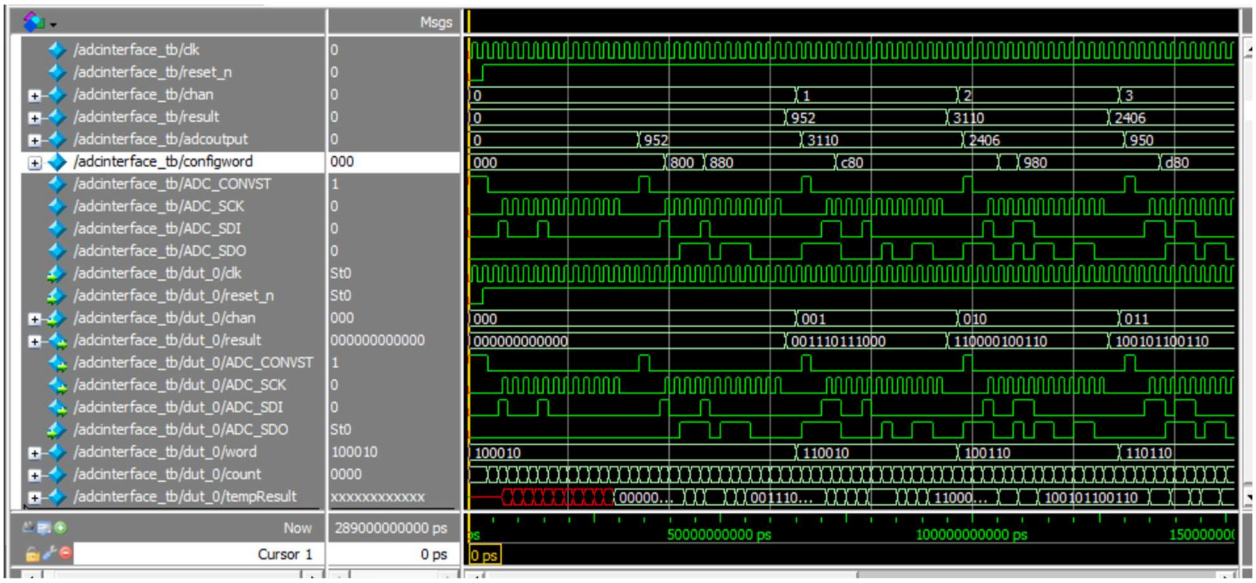

Figure 1 - 'adcinterface.sv' Simulation Waveforms

```
add wave -r sim:/adcinterface_tb/*
VSIM 3> run -all
# ADC output check -         PASS: expected ADC ouput 3b8, received 3b8
# Config word Check -         PASS: channel 0, expected config word 22, received 22
# ADC output check -         PASS: expected ADC ouput c26, received c26
# Config word Check -         PASS: channel 1, expected config word 32, received 32
# ADC output check -         PASS: expected ADC ouput 966, received 966
# Config word Check -         PASS: channel 2, expected config word 26, received 26
# ADC output check -         PASS: expected ADC ouput 3b6, received 3b6
# Config word Check -         PASS: channel 3, expected config word 36, received 36
# ADC output check -         PASS: expected ADC ouput 8c6, received 8c6
# Config word Check -         PASS: channel 4, expected config word 2a, received 2a
# ADC output check -         PASS: expected ADC ouput e46, received e46
# Config word Check -         PASS: channel 5, expected config word 3a, received 3a
# ADC output check -         PASS: expected ADC ouput f72, received f72
# Config word Check -         PASS: channel 6, expected config word 2e, received 2e
# ADC output check -         PASS: expected ADC ouput 32e, received 32e
# Config word Check -         PASS: channel 7, expected config word 3e, received 3e
# ** Note: $stop    : C:/Users/nicws/OneDrive/Desktop/2nd year Beng/Term 6/ELEX 7660/Labs/x7660/Lab4/adcinterface_tb.sv(63)
#    Time: 289 ms  Iteration: 1  Instance: /adcinterface_tb
# Break in Module adcinterface_tb at C:/Users/nicws/OneDrive/Desktop/2nd year Beng/Term 6/ELEX 7660/Labs/x7660/Lab4/adcinterface_tb.sv line 63
```

Figure 2 - 'adcinterface.sv' Simulation Transcript

‘enc2chan’

This code is basically the same as my ‘enc2freq.sv’ code, with the frequencies replaced with channels instead.

```
// ELEX 7660 Lab 4
// Nicholas Scott AKA "White Cheddar"
// A01255181
// Feb. 6th, 2024
// Instructor: Sweet Bobby T

module enc2chan
   ( input logic cw, ccw,       // outputs from lab 2 encoder module
     output logic [2:0] chan,  // desired channel
     input logic reset_n, clk); // reset and clock

    logic [7:0] countup = 0;
    logic [7:0] countdown = 0;

    always_ff @(posedge clk) begin
        if (reset_n) begin  // if active-low reset  is not pressed
            if (cw)
                countup <= countup + 1; // Count up for cw movement
            else if (ccw)
                countdown <= countdown + 1; // Count "down" for ccw movement

            if (countup >= 4) begin // after 4 cw pulses
                case (chan) // increment to next channel
                    0 : chan <= 1;
                    1 : chan <= 2;
                    2 : chan <= 3;
                    3 : chan <= 4;
                    4 : chan <= 5;
                    5 : chan <= 6;
                    6 : chan <= 7;
                    7 : chan <= 0; // cw rollover
                    default : chan <= 0;
                endcase
                countup <= 0;
            end
            else if (countdown >= 4) begin // after 4 ccw pulses
                case (chan) // decrement to last channel
                    0 : chan <= 7;    // ccw rollover
                    7 : chan <= 6;
                    6 : chan <= 5;
                    5 : chan <= 4;
```

```
                    4 : chan <= 3;
                    3 : chan <= 2;
                    2 : chan <= 1;
                    1 : chan <= 0;
                    default : chan <= 0;
                endcase
                countdown <= 0;
            end
        end
        else // set chan to zero if reset is pressed
            chan <= 0;
    end
endmodule
```

You didn't ask us to write a test bench for this. Besides, I know it will work because 'enc2freq.sv' worked.

'lab4.sv'

This module looks a lot like 'lab3.sv':

```
// ELEX 7660 Lab 4
// Nicholas Scott AKA "White Cheddar"
// A01255181
// Feb. 6th, 2024
// Instructor: Sweet Bobby T

module lab4 ( input logic ADC_SDO,        // ADC input
              input logic CLOCK_50,       // 50 MHz clock
              (* altera_attribute = "-name WEAK_PULL_UP_RESISTOR ON" *)
              input logic enc1_a, enc1_b, // Encoder 1 pins
              (* altera_attribute = "-name WEAK_PULL_UP_RESISTOR ON" *)
              input logic enc2_a, enc2_b, // Encoder 2 pins
              input logic s1, s2,         // reset and onOff pushbuttons
              output logic [7:0] leds,    // 7-seg LED enables
              output logic [3:0] ct,      // digit cathodes
              output logic ADC_CONVST, ADC_SCK, ADC_SDI // ADC outputs
              );

   logic [1:0] digit;  // select digit to display
   logic [3:0] disp_digit;  // current digit of count to display
   logic [15:0] clk_div_count; // count used to divide clock
   logic [2:0] chan;  // channel
   logic [11:0] result; // ADC result
```

```
   logic [7:0] enc1_count, enc2_count; // count used to track encoder movement
and to display
   logic enc1_cw, enc1_ccw, enc2_cw, enc2_ccw;  // encoder module outputs

   // instantiate modules to implement design
   decode2 decode2_0 (.digit,.ct) ;
   decode7 decode7_0 (.num(disp_digit),.leds) ;
   encoder encoder_1 (.clk(CLOCK_50), .a(enc1_a), .b(enc1_b), .cw(enc1_cw),
.ccw(enc1_ccw));
   enc2chan enc2chan_1 (.clk(CLOCK_50), .cw(enc1_cw), .ccw(enc1_ccw),
.chan(chan), .reset_n(s1));
   adcinterface adcinterface_1 (.clk(clk_div_count[15]), .reset_n(s1),
.chan(chan), .result, .ADC_CONVST, .ADC_SCK, .ADC_SDI, .ADC_SDO);

   // use count to divide clock and generate a 2 bit digit counter to determine
which digit to display
   always_ff @(posedge CLOCK_50)
     clk_div_count <= clk_div_count + 1'b1 ;

  // assign the top two bits of count to select digit to display
  assign digit = clk_div_count[15:14];

  // Select digit to display (disp_digit)
  // the right three digits display the ADC output in hexadecimal
  // the leftmost digit displays the channel
  always_comb begin

    case (digit)
      0 : disp_digit = result[3:0]; // result LSN (Least Significant Nibble)
      1 : disp_digit = result[7:4]; // the middle nibble
      2 : disp_digit = result[11:8]; // result MSN
      3 : disp_digit = chan; // Set digit 3 channel
    endcase

  end

endmodule
```

This code yielded the following compilation report:

Flow Summary

<<Filter>>

Flow Status	Successful - Sat Feb 10 15:41:04 2024
Quartus Prime Version	18.1.0 Build 625 09/12/2018 SJ Lite Edition
Revision Name	lab4
Top-level Entity Name	lab4
Family	Cyclone V
Device	5CSEMA4U23C6
Timing Models	Final
Logic utilization (in ALMs)	48 / 15,880 (< 1 %)
Total registers	75
Total pins	23 / 314 (7 %)
Total virtual pins	0
Total block memory bits	0 / 2,764,800 (0 %)
Total DSP Blocks	0 / 84 (0 %)
Total HSSI RX PCSs	0
Total HSSI PMA RX Deserializers	0
Total HSSI TX PCSs	0
Total HSSI PMA TX Serializers	0
Total PLLs	0 / 5 (0 %)
Total DLLs	0 / 4 (0 %)

Figure 3 - 'lab4.sv' Compilation Report

My Issue

The reason I couldn't demo on the first week is because I was defining my 'word' variable incorrectly. For some reason, ModelSim did not complain about it, but Quartus didn't like it. I found out that:

```
case (chan)
    0 : word = '{1, 0, 0, 0, 1, 0};
    1 : word = '{1, 1, 0, 0, 1, 0};
    2 : word = '{1, 0, 0, 1, 1, 0};
    3 : word = '{1, 1, 0, 1, 1, 0};
    4 : word = '{1, 0, 1, 0, 1, 0};
    5 : word = '{1, 1, 1, 0, 1, 0};
    6 : word = '{1, 0, 1, 1, 1, 0};
    7 : word = '{1, 1, 1, 1, 1, 0};
endcase
```

≠

```
case (chan)
    0 : word = 6'b100010;
    1 : word = 6'b110010;
    2 : word = 6'b100110;
    3 : word = 6'b110110;
    4 : word = 6'b101010;
    5 : word = 6'b111010;
    6 : word = 6'b101110;
    7 : word = 6'b111110;
endcase
```

The code on the left worked in ModelSim, but not Quartus. I suspect that ModelSim was smart enough to realize that I was concatenating single-bit logic, but Quartus assumed a width of 32 bits for each of the values, and thus the leading 0's caused 'word[5:0]' to be all 0's, and hence I was permanently stuck on channel 0 and 1 differential mode. Very silly of me.

RTL Netlists

This whole design yielded the following RTL Netlists:

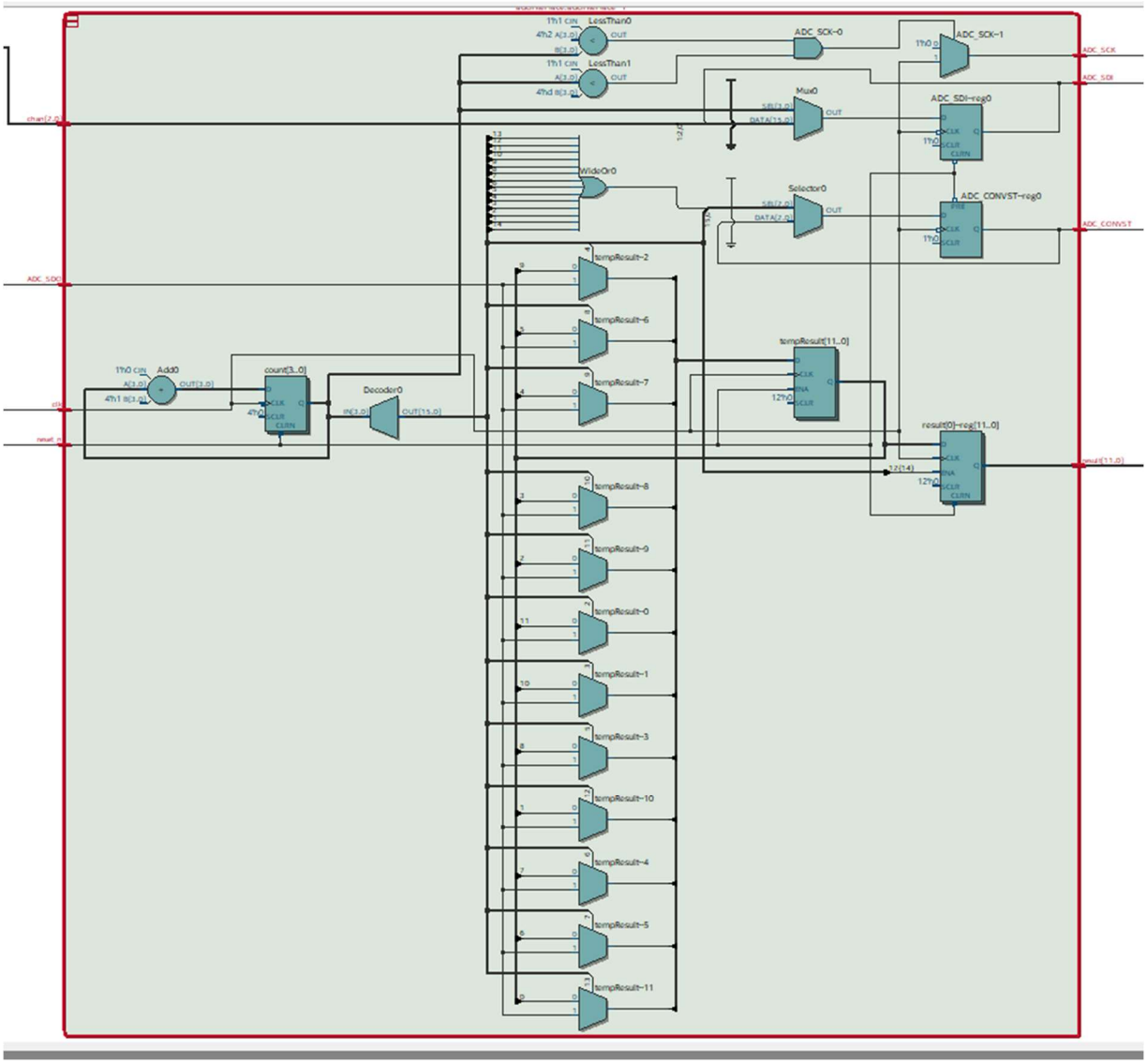

Figure 4 - 'adcinterface.sv' RTL Netlist

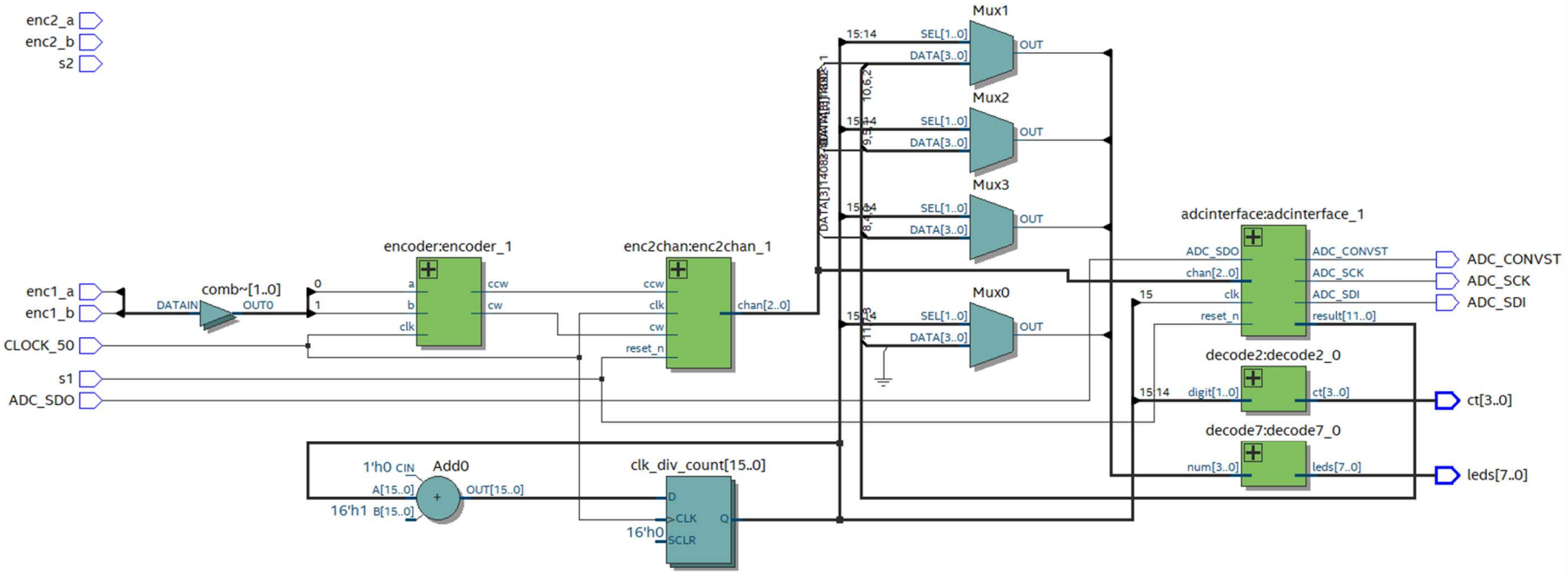

Figure 5 - 'lab4.sv' RTL Netlist

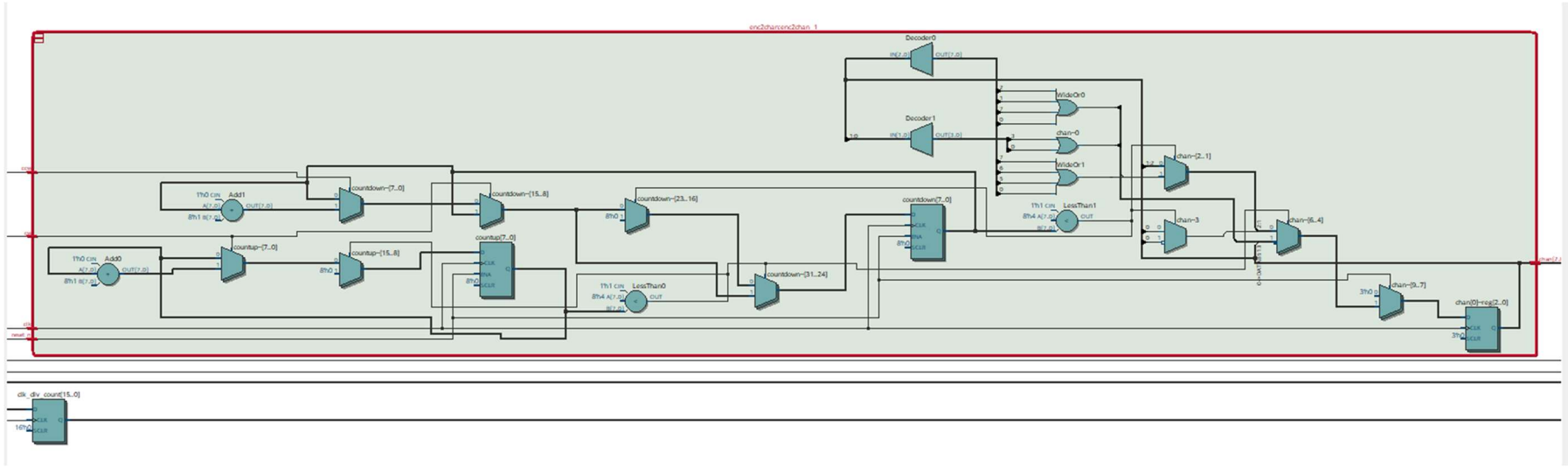

Figure 6 - 'enc2chan.sv' RTL Netlist