EE 371 Autumn 2016 - Lab 1

William Li, Jun Park, Dawn Liang

November 2, 2016

We certify that the work in this report is our own, and that any work that is not ours is cited.

William Li   Dawn Liang   Jun Park

_____   _____   _____
Signature   *Signature*   *Signature*

_____   _____   _____
Date   *Date*   *Date*

Contents

1 ABSTRACT

This lab focuses on introducing us to the tools and methods of digital design. We were introduced to the four levels of abstraction in modeling and implementation with Verilog on the Altera DE1-SoC board. Then, we familiarised ourselves with the tools involved in designing and testing hardware applications with Quartus. Additionally, we were introduced to the general design process for hardware and software and applications.

2 INTRODUCTION

This lab focuses on designing and building VHDL (Verilog Hardware Description Language) programs. First, we built four different types of counters using different modeling techniques: a 4-bit ripple up counter using gate modeling, a 4-bit synchronous up counter using both dataflow model and schematic entry, and a 4-bit synchronous Johnson up counter using the behavioural model. In the process of building and testing these counters, we were introduced to Icarus Verilog (iVerilog) and GTKWave software, for compiling our designs and simulating waveforms. We then loaded our designs onto an Altera Cyclone V FPGA, where we verified their functionality on hardware. Then we utilised Signal Tap II, a logic analyzer for probing designs in hardware. Finally, we were briefly introduced to the C programming language. We learned the basics of a C program in the CodeBlocks IDE by compiling a provided C project, and then we built a simple C car price calculator program that asks for relevant input data and outputs an approximate list price for a brand new vehicle.

3 DISCUSSION

3.1 Design

3.1.1 Design Specification

We built four different types of counters: a ripple-up counter, two synchronous up counters, and a Johnson counter, all of which counted on every positive clock edge. All the counters we built had active low reset. The first three counters were directly written in SystemVerilog, the fourth counter was designed using Quartus schematic entry feature. The ripple-up and synchronous counters counted up in binary, while the Johnson counter counted up by the most significant bit. The listPrice calculator was to take appropriate input data and calculate the estimated price of a new car. The program prompts the user for the manufacturers cost, the estimated dealers markup, the pretax discount, and the sales tax, and then outputs the estimated list price.

The list price calculator was to take appropriate input data and calculate the estimated price of a new car. The program prompts the user for the manufacturer's cost, the estimated dealer's markup, the pre-tax discount, and the sales tax, and then calculates and outputs the estimated list price.

3.1.2 Design Procedure

In Verilog HDL, we designed and implemented four counters using three different levels of modeling abstraction: structural/gate-level, dataflow, and behavioural. The first three counters were implemented using the D-flipflop provided in the lab spec; the fourth counter was implemented via Quartus' schematic entry feature.

```verilog
// provided D-flipflop code
module DFlipFlop(q, qBar, D, clk, rst);
  input D, clk, rst;
  output q, qBar;
  reg q;

  not n1 (qBar, q);
  always@ (negedge rst or posedge clk) begin
    if(!rst)
      q = 0;
```

```
    else
        q = D;
    end
endmodule
```

The ripple-up counter was implemented at the structural level. The connections between the D-flipflops and outputs were explicitly assigned based on the gate-level diagram. Then, a testbench was written to verify its behaviour using iverilog and gtkwave simulations. The design was then loaded onto the DE1-SoC board through Quartus and probed using Signal Tap II Logic Analyser, to see if its performance on hardware matched expected ideal behaviour in simulation.

The first synchronous up counter was implemented at the dataflow level. We diagrammed the states and generated boolean expressions using truth tables and K-maps, depicted on the right. We then drew the gate-level diagram from those expressions, and wrote Verilog code from that. The testing procedure for this counter was the same as previously described.

The johnson counter was implemented at the behavioural level. All behaviour was described within the verilog code. As the counter was essentially a cycling of inputs, the gate-level diagram was fairly straightforward to draw as well. The testing procedure for this counter was the same as previously described.

Figure 1: Truth table & K-map logic reduction for synchronous counter

The second synchronous up counter was built via schematic entry, a feature of Quartus II. The verilog code was then generated from the schematic, using Quartus' generate HDL feature. We wrote the testbench for this in Quartus, and the rest of the testing procedure was the same as previously described.

3.1.3   System Description

The four counters take a clock and an active-low reset as the input, which we wired to the internal 50MHz clock and SW[0] respectively on the DE1-SoC board. For the synchronous counter built with Quartus' schematic entry, we included an active-high 'enable' signal input, which acts as an pause/resume switch, wired to SW[1] on the physical board. All four counters output 4-bits, counting up, outputting to LED's [3:0] on the board

The list price calculator takes the manufacturer's cost ($), the estimated markup (%), the pre-tax discount (%), and sales tax (%) as inputs. It then calculates the estimated list price of a vehicle, calculated from those values, and displays it to the user as output.

3.1.4   Software Implementation

Each counter module is written in Verilog, and utilises a clock divider and D-FlipFlop submodule. The connections between each submodule were determined by the logic reductions and gate-level designs.

The listPrice program was written in C, to be executed in the console as a script.

3.1.5   Hardware Implementation

The inputs/outputs of the counter modules were directly assigned to pins on the FPGA, using Quartus' pin planner and documentation for our specific device.
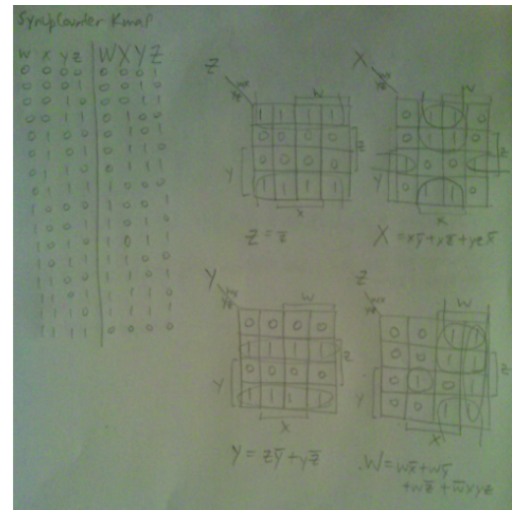
3.2 Test

3.2.1 Test Plan

The counters were both simulated on our computers using iverilog and gtkwave, to check the software implementation, and then tested on the physical DE1-SoC board, to check the hardware implementation. In simulation, the active-low reset and counting sequence were checked. In the case of the schematic entry counter, the enable signal was also checked. In hardware, the LEDs had to display the four outputs bits and count in the correct sequence; the reset signal should reset to all 0's and restart counting when untriggered. In the case of the schematic entry synchronous counter, the enable signal should pause counting when triggered and resume when untriggered.

The listPrice calculator was tested with different types of inputs, both expected and invalid inputs. Its output must be verified against external calculations to ensure it is mathematically correct as well.

3.2.2 Test Specification

The counters need to be tested such that the reset signal properly resets the output to all 0's, and that the reset signal is active-low. They also need to count in the correct 4-bit sequence, counting each positive clock edge. They should cycle back to their initial state after reaching the maximum state. For the schematic entry synchronous counter, the enable signal should alse pause/resume the counting sequence.

The listPrice calculator's inputs need to be tested against possible edge cases, such as negative values or invalid inputs like strings. Additionally, expected inputs need to be tested to check that they calculate the correct list price.

3.2.3 Test Cases

In simulation, the counters were connected to a clock and reset signal (in the case of the fourth counter, an additional enable signal connected), and the four-bit output bus was monitored. For each counter, the reset was triggered, untriggered, and then the clock ran until the reset was triggered again. The 4-bit outputs were measured to ensure appropriate counting behaviour, counting up at each clock edge and resetting to 0 when triggered.

On the board, the counters were connected to a switch for reset and four LEDs for outputs (for the fourth counter, an additional switch for enable). The reset was triggered, untriggered, then the clock ran until the reset was triggered again. The 4-bit outputs were monitored on the LEDs, and their behaviour checked to ensure they were counting up at each clock edge and resetting when triggered. In the case of the fourth counter, the enable signal was also checked to see if it paused/resumed counting.

The listPrice calculator was tested by running the program and checking edge cases. We tested regular, expected inputs, as well as negative value and string inputs, to check handling of unexpected inputs.

4 RESULTS

In simulation and on the DE1-SoC board, our four counters worked as expected. The waveforms showed that they incremented by 1 bit at each clock edge, cycling back to 0 once the maximum had been reached. They reset to 0 when signaled, showing no signs of metastable behaviour. For the fourth counter, the enable signal properly paused/resumed the counter.

On the board, the LEDs corresponding to the 4-bits exhibited appropriate counting behaviour, also cycling back to 0 once the maximum had been reached. When the reset switch was triggered, the LEDs switched off, and resumed counting at 0 when reset was switched off. For the fourth counter, the enable switch properly paused and resumed counting.

The designs were probed on the board using Signal Tap II Logic Analyser, and the waveforms generated matched the expected ideal waveforms. There was significantly more noise in the signals immediately after triggers in the Signal Tap data, due to gate delays that occur in real time but not in simulations.

Comparison of gate-level design to synthesised RTL view   The ripple-up counter we designed is exactly the same as the one Quartus synthesised. The Johnson up counters are also the same, though the Quartus diagram stacks the D-ff's on top of each other for conciseness and simpicity. For the synchronous counter, the diagram we drew was significantly different from the one Quartus synthesized. This is because the logic reduction performed by Quartus is different from the logic reduction we used.

The listPrice calculator outputted the correct prices when presented with expected positive numerical inputs. When given invalid inputs, the program also behaves as expected. For negative inputs, the program outputs a mathematically valid but nonsensical result (a negative price). For string inputs, the program throws an error and crashes.

5 ANALYSIS OF ERRORS

5.1 General Issues

The majority of our issues had to do with difficulty finding documentation, and sometimes outdated documentation, since resources were spread across the class website and not centralised in one place. This was mostly an issue when trying to get Signal Tap II working - figuring out how to set up the environment with regards to the device and the pins. Eventually, with TA help, we found documentation and figured it out.

5.2 Failure Modes & Effects Analysis

We performed failure modes and effects analysis for three counters. Looking at each signal in the gate-level circuit, we considered the effects on the overall circuit if it were 1) stuck at 0 (SA0), or 2) stuck at 1 (SA1).

5.2.1 Ripple-up counter

output signal out

- SA0: This means that all the bits Q0 to Q3 are stuck at zero. Therefore even if the clock was operating properly, the counter will never count any positive edges of the clock.

- SA1: This means that all the bits Q0 to Q3 are stuck at one. Therefore even if the clock was operating properly, the counter will remain at the binary value 1111, and never wrap back around to zero to count back up.

input signal clk

- SA0: This means that there wont be any positive clock edges, therefore the counter will stop counting up.

- SA1: This means that there wont be any more positive clock edges beyond this point, therefore the counter will stop counting up.

input signal rst

- SA0: The output will be reset to zero, but it will remain indefinitely at zero.

- SA1: The output will be at an unknown state if the rst becomes stuck at 1 from the moment the counter is powered on. If it rst becomes stuck at 1 during operation, the counter will still count up properly, but properly resetting the counter wont be possible.

4

internal signal Q0

- SA0: The least significant bit of the output will never change from 0 and the rest of the counter will stop counting up as the cascading clock into the series flipflops get stuck at 1.

- SA1: The least significant bit of the output will never change from 1 and the rest of the counter will stop counting up as the cascading clock into the series flipflops get stuck at 0.

internal signal Q1

- SA0: The second least significant bit of the output will never change from 0 and the rest of the counter will stop counting up as the cascading clock into the series flipflops get stuck at 1.

- SA1:The second least significant bit of the output will never change from 1 and the rest of the counter will stop counting up as the cascading clock into the series flipflops get stuck at 0.

internal signal Q2

- SA0: The second most significant bit of the output will never change from 0 and the rest of the counter(Q3 bit) will stop counting up as the cascading clock into the series flipflop get stuck at 1.

- SA1: The second most significant bit of the output will never change from 1 and the rest of the counter (Q3 bit) will stop counting up as the cascading clock into the series flipflop get stuck at 0.

internal signal Q3

- SA0: The most significant bit of the output will never change from 0 but otherwise attemps to count up.

- SA1:The most significant bit of the output will never change from 1 but otherwise attemps to count up.

internal siganl clkTemp1

- SA0: After the least significant bit of the output, the counter will stop counting as the cascading clock into the series flipflops get stuck at 0.

- SA1: After the least significant bit of the output, the counter will stop counting as the cascading clock into the series flipflops get stuck at 1.

internal signal clkTemp2

- SA0: After the second least significant bit of the output, the counter will stop counting as the cascading clock into the series flipflops get stuck at 0.

- SA1: After the second least significant bit of the output, the counter will stop counting as the cascading clock into the series flipflops get stuck at 1.

internal signal clkTemp3

- SA0: After the third least significant bit of the output, the counter will stop counting as the cascading clock into the series flipflops get stuck at 0.

- SA1: After the third least significant bit of the output, the counter will stop counting as the cascading clock into the series flipflops get stuck at 1.

internal signal clkTemp4

- SA0: The most significant bit of the output will be stuck at 0 indefinitely, otherwise attempts to count up.

- SA1: The most significant bit of the output will be stuck at 0 indefinitely otherwise attempt to count up.

5.2.2 Synchronous up counter

input signal rst

- SA0: The system will be in constant reset mode, so the output will remain at 0 indefinitely

- SA1: The system will count indefinitely, and the reset button will not work

input signal clk

- SA0: The clock will not advance, so the counter will not change output

- SA1: The clock will not advance, so the counter will not change output

output signal Q0

- SA0: The boolean expression for D0 will reduce to equal rst, so the first D-flipflop will be stuck at 1. The stuck at 1 will propogate down the counter and the counter will be stuck at 4'b1111.

- SA1: The boolean expression for D0 will always be false, so the D-flipflop will be stuck at 0. It will propogate down the counter and the counter will be stuck at 4'b0000.

output signal Q0

- SA0: D0 will always be 1, since rst is always 1 and Q0 is SA0 (XOR to 1). The XOR gates for the next D-flip flop will be passed Q0, so it will hold its value, and this applies to all the flip flops. Thus, Q1 will be stuck at 1 while the other 3 bits will hold their value.

- SA1: D0 will always be 0, since rst is always 1 and Q0 is SA1 (XOR to 0). The rest of the bits shift down one level of significance, since the first flip-flop is essentially pulled out of commission by the XOR gate, and the counter continues operating as a 3-bit counter.

output signal Q1

- SA0: D1 will always be high, and the following counters will hold their values. However the least significant bit, Q0, will continue to operate normally since it does not depend on Q1

- SA1: D1 will always be low, and the rest of the bits act as a 3-bit counter. The flipflop is essentially pulled out of the circuit by the XOR gate

output signal Q2

- SA0: D2 will always be high, and the following counter will hold its value. Q0 and Q1 will continue outputting as normal, essentially a 2-bit counter.

- SA1: D2 will always be low, and the rest of the bits act as a 3-bit counter.

output signal Q3

- SA0: D3 will always be high, but there are no following counters to be affected. Q0, Q1, and Q2 will countinue outputting as normal, essentially a 3-bit counter.

- SA1: D3 will always be low, and the rest of the bits act as a 3-bit counter.

5.2.3 Johnson counter

input signal rst

- SA0: Circuit will operate normally, but is unable to reset.

- SA1: Circuit will be stuck at all 0s (output LEDs will be all off).

input signal clk

- SA0: Circuit will be paused; output of the counter will not change.

- SA1: Circuit will be paused; output of the counter will not change.

internal signal QA

- SA0: The counter operation is normal except QA is always stuck at low.

- SA1: The counter operation is normal except QA is always stuck at high.

internal signal QB

- SA0: Counter will eventually be stuck in a stage where QB, QC and QD are 0s, and QA is a 1.

- SA1: Counter will eventually be stuck in a stage where QB, QC and QD are 1s, and QA is a 0.

internal signal QC

- SA0: Counter will eventually be stuck in a stage where QC and QD will be 0s, and QA and QB will be 1s.

- SA1: Counter will eventually be stuck in a stage where QC and QD will be 1s, and QA and QB will be 0s.

internal signal QD

- SA0: Counter will end in a stage where the output is all 0s.

- SA1: Counter will end in a stage where the output is all 1s.

internal signal QBar

- SA0: Counter will eventually end in a stage where the output is all 0s.

- SA1: Counter will eventually end in a stage where the output is all 1s.

6 SUMMARY AND CONCLUSION

Summary   This lab involved designing and building four 4-bit counters at various levels of abstraction in Verilog, to compare the various RTL implementations. We then tested them both in simulation and on the physical board. We also grew acquainted with iverilog and gtkwave as tools for compiling and simulating our designs, and Signal Tap II for debugging our hardware. We were then introduced to the C language, and wrote a simple program involving a few numerical inputs, calculations, and output. We were also introduced to the basic design process, involving spec writing and detailed descriptions of our project deliverables.

Conclusion   Overall, the project was good for getting oriented with the environments and tools we will be using throughout the rest of the quarter. We refreshed ourselves with Quartus and verilog, and we were introduced to new tools like iverilog, gtkwave, Signal Tap II, and CodeBlocks. However, it was very messy and fragmented, and many resources were difficult to find. The spec was vague and unclear at many points, and a lot of the project was left up to our discretion.

7 APPENDIX

7.1 Designing and Building VHDL applications - counters

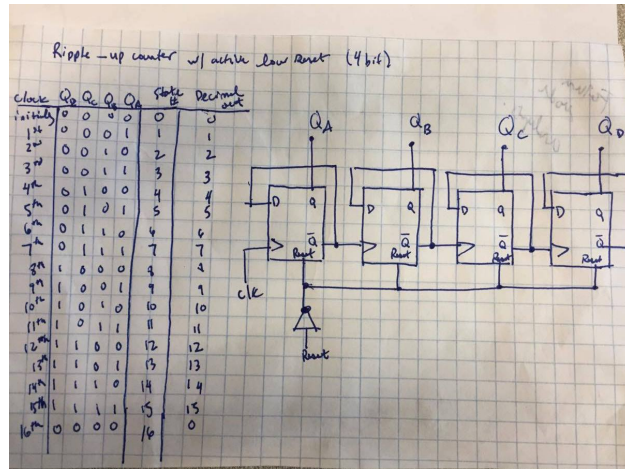### 7.1.1 Logic reduction & gate-level diagrams
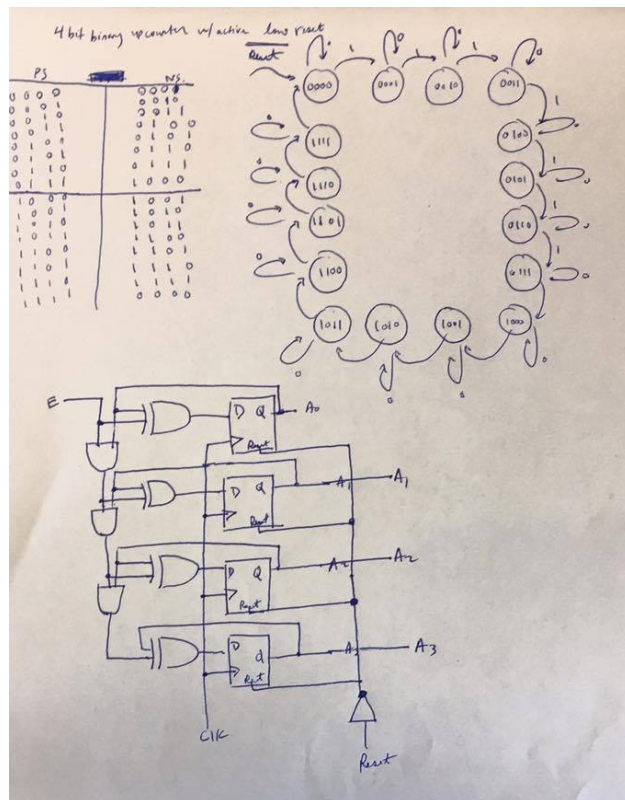

Figure 2: Ripple-up counter logic reduction

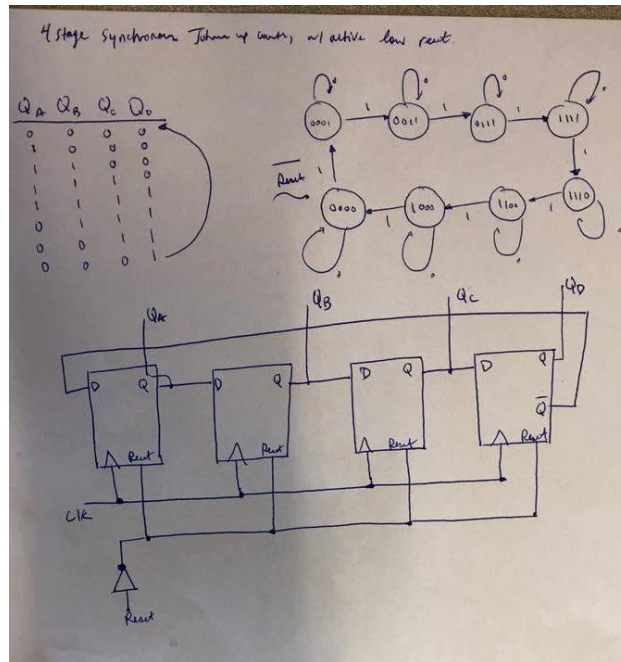Figure 3: Synchronous up counter logic reduction

Figure 4: Johnson counter logic reduction

7.1.2 Verilog code

Ripple-up counter   The ripple-up counter, implemented at the gate-level

```
/*
   Ripple-up counter module

   Authors: William Li, Dawn Liang, Jun Park
   Date: 16 Oct 2016
*/
module rippleUpCounter(out, clk, rst);
   // declare inputs/outputs, wires
   output logic [3:0] out;
   input logic clk, rst;
   logic Q0, Q1, Q2, Q3, clkTemp1, clkTemp2, clkTemp3, clkTemp4;

   // clock
   logic [31:0] clocks;
   parameter WHICH_CLOCK = 22; // choose clock speed
   clock_divider cdiv (clk, clocks);

   // connect Dffs
   DFlipFlop d0(.q(Q0), .qBar(clkTemp1), .D(clkTemp1), .clk(clocks[WHICH_CLOCK]),
       .rst(rst));
   DFlipFlop d1(.q(Q1), .qBar(clkTemp2), .D(clkTemp2), .clk(clkTemp1), .rst(rst));
   DFlipFlop d2(.q(Q2), .qBar(clkTemp3), .D(clkTemp3), .clk(clkTemp2), .rst(rst));
   DFlipFlop d3(.q(Q3), .qBar(clkTemp4), .D(clkTemp4), .clk(clkTemp3), .rst(rst));

   // output logic
   assign out = {Q3,Q2,Q1,Q0}; //not {Q0, Q1, Q2, Q3};
endmodule
```

```verilog
/*
   Testing module for ripple-up counter

   Authors: William Li, Dawn Liang, Jun Park
   Date: 16 Oct 2016
*/

`include "rippleUpCounter.v"

module rippleUpCounter_testbench;
  wire [3:0] out;
  reg clk, rst;

  rippleUpCounter dut(.out(out), .clk(clk), .rst(rst));

  //--------------------------------------------------------
  parameter delay = 4;

  initial begin
    clk = 0;
    rst = 0;
  end

  always begin
    #delay clk = ! clk;
  end

  //--------------------------------------------------------
  initial // Response
  begin
    $display("\t\t out\t clk\t reset\t Time ");
    $monitor("\t\t %b\t %b\t %b\t", out, clk, rst, $time);
  end



  initial // Stimulus
  begin
    rst = 0; #delay;
    rst = 1; #delay;
    #delay;
    #delay;
    #delay;
    #delay;
    #delay;
    #delay;
    #delay;
    $finish; // finish simulation
  end

  // file for gtkwave
  initial begin
    // these two files support gtkwave and are required
    $dumpfile("rup.vcd");
    $dumpvars;
  end
endmodule
```

Synchronous up counter   The synchronous up counter, implemented at the dataflow level

```
/*
   Synchronous up counter module

   Authors: William Li, Dawn Liang, Jun Park
   Date: 16 Oct 2016
*/
module synUpCounter(out, clk, rst);
   // declare inputs/outputs, wires
   output logic [3:0] out;
   input logic clk, rst;
   logic Q0, Q1, Q2, Q3;
   logic D0, D1, D2, D3;

   // clocks
   logic [31:0] clocks;
   parameter WHICH_CLOCK = 5;
   clock_divider cdiv (clk, clocks);

   // connect flip-flops
   DFlipFlop dff0(.q(Q0), .qBar(), .D(D0), .clk(clocks[WHICH_CLOCK]), .rst(rst));
   DFlipFlop dff1(.q(Q1), .qBar(), .D(D1), .clk(clocks[WHICH_CLOCK]), .rst(rst));
   DFlipFlop dff2(.q(Q2), .qBar(), .D(D2), .clk(clocks[WHICH_CLOCK]), .rst(rst));
   DFlipFlop dff3(.q(Q3), .qBar(), .D(D3), .clk(clocks[WHICH_CLOCK]), .rst(rst));

   // input boolean logic
   assign D0 = (~Q0)&rst;
   assign D1 = ((Q0&~Q1) | (Q1&~Q0))&rst;
   assign D2 = ((Q2&~Q1) | (Q2&~Q0) | (Q1&Q0&~Q2))&rst;
   assign D3 = ((Q3&~Q2) | (Q3&~Q1) | (Q3&~Q0) | (~Q3&Q2&Q1&Q0))&rst;

   // output logic
   assign out = {Q3, Q2, Q1, Q0};
endmodule
```

```
/*
   Testing module for synchronous counter

   Authors: William Li, Dawn Liang, Jun Park
   Date: 16 Oct 2016
*/
`include "synUpCounter.v"

module synUpCounter_testbench;
   wire [3:0] out;
   reg clk, rst;

   synUpCounter dut(.out(out), .clk(clk), .rst(rst));

   //------------------------------------------------------
   parameter delay = 4;

   initial begin
    clk = 0;
    rst = 0;
   end
```

```verilog
  always begin
    #delay clk = ! clk;
  end

  //--------------------------------------------------------
  initial // Response
  begin
    $display("\t\t out\t clk\t reset\t Time ");
    $monitor("\t\t %b\t %b\t %b\t", out, clk, rst, $time);
  end



  initial // Stimulus
  begin
    rst = 0; #delay;
    rst = 1; #delay;
    #delay;
    #delay;
    #delay;
    #delay;
    #delay;
    #delay;
    #delay;
    $finish; // finish simulation
  end

  // file for gtkwave
  initial begin
    // these two files support gtkwave and are required
    $dumpfile("syn.vcd");
    $dumpvars;
  end
endmodule
```

Johnson Counter   The johnson counter, implemented at the behavioural level

```verilog
/*
  Johnson counter module

  Authors: William Li, Dawn Liang, Jun Park
  Date: 16 Oct 2016
*/

module johnsonCounter(out, clk, rst);
  // declare inputs/outputs, wires
  output [3:0] out;
  input clk, rst;
  reg [3:0] temp;

  // clock
  logic [31:0] clocks;
  parameter WHICH_CLOCK = 5;
  clock_divider cdiv (clk, clocks);

  // count edges
  always_ff@(negedge rst or posedge clocks[WHICH_CLOCK]) begin
```

```verilog
      if (rst == 0) begin
        temp <= 4'b0000;
      end else if (clocks[WHICH_CLOCK] == 1'b1) begin
        temp <= {~temp[0], temp[3:1]}; // right shift and negating most signf bit
      end
   end

   // output logic
   assign out = temp;
endmodule
```

```verilog
/*
   Testing module for johnson up counter

   Author: William Li, Dawn Liang, Jun Park
   Date: 16 Oct 2016
*/
`include "johnsonUpCounter.v"

module johnsonUpCounter_testbench;
   wire [3:0] out;
   reg clk, rst;

   johnsonUpCounter dut(.out(out), .clk(clk), .rst(rst));

   //-------------------------------------------------------
   parameter delay = 4;

   initial begin
     clk = 0;
     rst = 0;
   end

   always begin
     #delay clk = ! clk;
   end

   //-------------------------------------------------------
   initial // Response
   begin
     $display("\t\t out\t clk\t reset\t Time ");
     $monitor("\t\t %b\t %b\t %b\t", out, clk, rst, $time);
   end


   initial // Stimulus
   begin
     rst = 0; #delay;
     rst = 1; #delay;
     #delay;
     #delay;
     #delay;
     #delay;
     #delay;
     #delay;
     #delay;
     #delay;
```

13

```
        #delay;
        #delay;
        #delay;
        #delay;
        #delay;
        #delay;
        #delay;
        $finish; // finish simulation
    end

    // file for gtkwave
    initial begin
        // these two files support gtkwave and are required
        $dumpfile("jhn.vcd");
        $dumpvars;
    end
endmodule
```

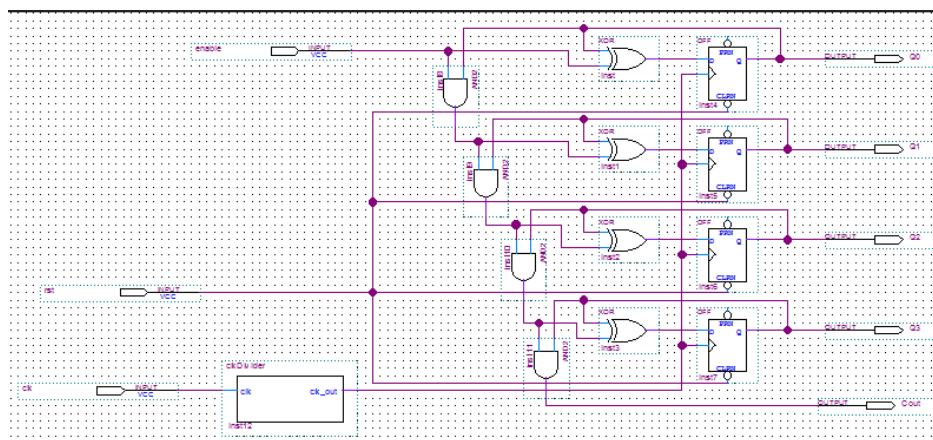**Synchronous up counter (schematic entry)** Schematic of the synchronous up counter, designed using Quartus' schematic entry

Figure 5: Synchronous counter schematic entry

The generated verilog code & tester

```
/*
  verilog code generated from Quartus' schematic entry
*/

// Copyright (C) 1991-2014 Altera Corporation. All rights reserved.
// Your use of Altera Corporation's design tools, logic functions
// and other software and tools, and its AMPP partner logic
// functions, and any output files from any of the foregoing
// (including device programming or simulation files), and any
// associated documentation or information are expressly subject
// to the terms and conditions of the Altera Program License
// Subscription Agreement, the Altera Quartus II License Agreement,
// the Altera MegaCore Function License Agreement, or other
// applicable license agreement, including, without limitation,
// that your use is for the sole purpose of programming logic
// devices manufactured by Altera and sold by Altera or its
```

14

```verilog
// authorized distributors. Please refer to the applicable
// agreement for further details.

// PROGRAM  "Quartus II 64-Bit"
// VERSION  "Version 14.0.0 Build 200 06/17/2014 SJ Web Edition"
// CREATED  "Tue Oct 18 12:20:52 2016"

module shematicEntrySynUpCounter(
   rst,
   clk,
   enable,
   Cout,
   Q0,
   Q1,
   Q2,
   Q3
);


input wire rst;
input wire clk;
input wire enable;
output wire Cout;
output wire Q0;
output wire Q1;
output wire Q2;
output wire Q3;

reg  SYNTHESIZED_WIRE_14;
reg  SYNTHESIZED_WIRE_15;
wire SYNTHESIZED_WIRE_16;
reg  SYNTHESIZED_WIRE_17;
wire SYNTHESIZED_WIRE_18;
reg  SYNTHESIZED_WIRE_19;
wire SYNTHESIZED_WIRE_20;
wire SYNTHESIZED_WIRE_21;
wire SYNTHESIZED_WIRE_6;
wire SYNTHESIZED_WIRE_8;
wire SYNTHESIZED_WIRE_10;
wire SYNTHESIZED_WIRE_12;

assign Q0 = SYNTHESIZED_WIRE_14;
assign Q1 = SYNTHESIZED_WIRE_15;
assign Q2 = SYNTHESIZED_WIRE_17;
assign Q3 = SYNTHESIZED_WIRE_19;



assign SYNTHESIZED_WIRE_6 = SYNTHESIZED_WIRE_14 ^ enable;

assign SYNTHESIZED_WIRE_8 = SYNTHESIZED_WIRE_15 ^ SYNTHESIZED_WIRE_16;

assign SYNTHESIZED_WIRE_20 = SYNTHESIZED_WIRE_17 & SYNTHESIZED_WIRE_18;

assign Cout = SYNTHESIZED_WIRE_19 & SYNTHESIZED_WIRE_20;


clkDivider b2v_inst12(
   .clk(clk),
```

15

```verilog
  .clk_out(SYNTHESIZED_WIRE_21));

assign SYNTHESIZED_WIRE_10 = SYNTHESIZED_WIRE_17 ^ SYNTHESIZED_WIRE_18;

assign SYNTHESIZED_WIRE_12 = SYNTHESIZED_WIRE_19 ^ SYNTHESIZED_WIRE_20;


always@(posedge SYNTHESIZED_WIRE_21 or negedge rst)
begin
if (!rst)
  begin
  SYNTHESIZED_WIRE_14 <= 0;
  end
else
  begin
  SYNTHESIZED_WIRE_14 <= SYNTHESIZED_WIRE_6;
  end
end


always@(posedge SYNTHESIZED_WIRE_21 or negedge rst)
begin
if (!rst)
  begin
  SYNTHESIZED_WIRE_15 <= 0;
  end
else
  begin
  SYNTHESIZED_WIRE_15 <= SYNTHESIZED_WIRE_8;
  end
end


always@(posedge SYNTHESIZED_WIRE_21 or negedge rst)
begin
if (!rst)
  begin
  SYNTHESIZED_WIRE_17 <= 0;
  end
else
  begin
  SYNTHESIZED_WIRE_17 <= SYNTHESIZED_WIRE_10;
  end
end


always@(posedge SYNTHESIZED_WIRE_21 or negedge rst)
begin
if (!rst)
  begin
  SYNTHESIZED_WIRE_19 <= 0;
  end
else
  begin
  SYNTHESIZED_WIRE_19 <= SYNTHESIZED_WIRE_12;
  end
end

assign SYNTHESIZED_WIRE_16 = SYNTHESIZED_WIRE_14 & enable;
```

```verilog
  assign SYNTHESIZED_WIRE_18 = SYNTHESIZED_WIRE_15 & SYNTHESIZED_WIRE_16;


endmodule
```

```verilog
/*
   Testing module for synchronous counter generated via schematic entry

   Authors: William Li, Dawn Liang, Jun Park
   Date: 16 Oct 2016
*/
`include "shematicEntrySynUpCounter.v"

module schmSynUpCounter_testbench;
  wire Cout, Q0, Q1, Q2, Q3;
  reg clk, rst, enable;

  shematicEntrySynUpCounter dut(.Cout, .Q0, .Q1, .Q2, .Q3, .rst, .clk, .enable);

  //-------------------------------------------------------
  parameter delay = 4;

  initial begin
    clk = 0;
    rst = 0;
  end

  always begin
    #delay clk = ! clk;
  end

  //-------------------------------------------------------
  initial // Response
  begin
    $display("\t\t Cout\t Q3\t Q2\t Q1\t Q0\t enable\t clk\t rst\t Time");
    $monitor("\t\t %b\t %b\t %b\t %b\t %b\t %b\t %b\t %b\t", Cout, Q3, Q2, Q1, Q0,
        enable, clk, rst, $time);
  end

  initial // Stimulus
  begin
    rst = 0; enable = 0; #delay;
    rst = 1; enable = 0; #delay;
          enable = 1; #delay;
          enable = 1; #delay;
          enable = 1; #delay;
          enable = 1; #delay;
          enable = 1; #delay;
          enable = 1; #delay;
          enable = 1; #delay;
          enable = 1; #delay;
          enable = 1; #delay;
          enable = 1; #delay;
          enable = 1; #delay;
          enable = 1; #delay;
          enable = 1; #delay;
          enable = 1; #delay;
```

```verilog
            enable = 0; #delay;
            enable = 0; #delay;
            enable = 0; #delay;
            enable = 1; #delay;
            enable = 1; #delay;
            enable = 1; #delay;
            enable = 1; #delay;
            enable = 1; #delay;
            enable = 1; #delay;
            enable = 1; #delay;
            enable = 1; #delay;
        $finish; // finish simulation
    end

    // file for gtkwave
    initial begin
        // these two files support gtkwave and are required
        $dumpfile("schm.vcd");
        $dumpvars;
    end
endmodule
```

7.1.3 RTL views
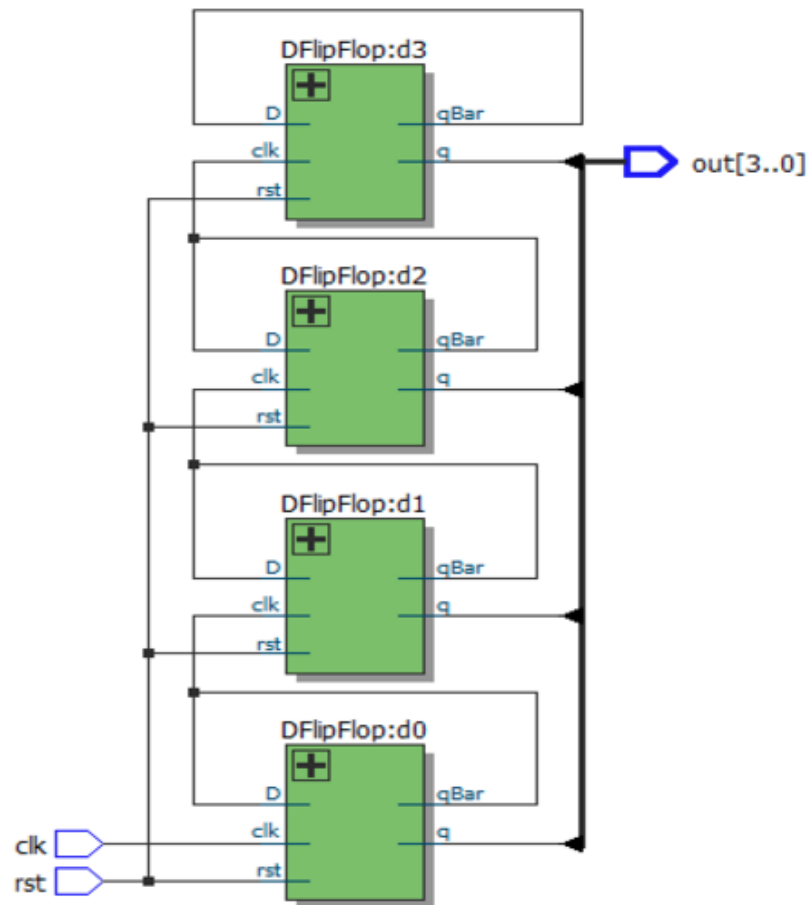
Figure 6: Ripple-up counter RTL view
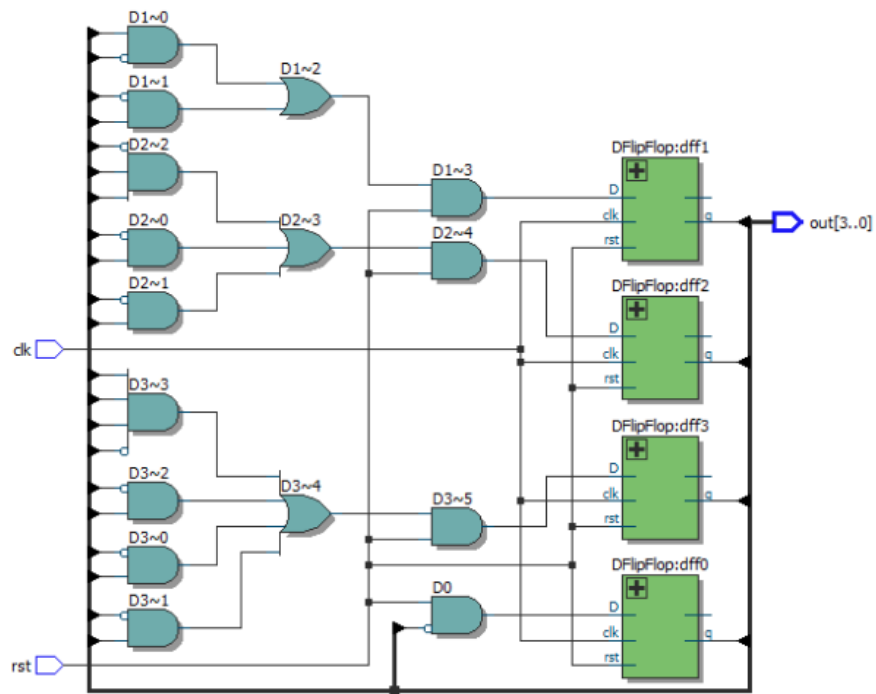
Figure 7: Synchronous up counter RTL view
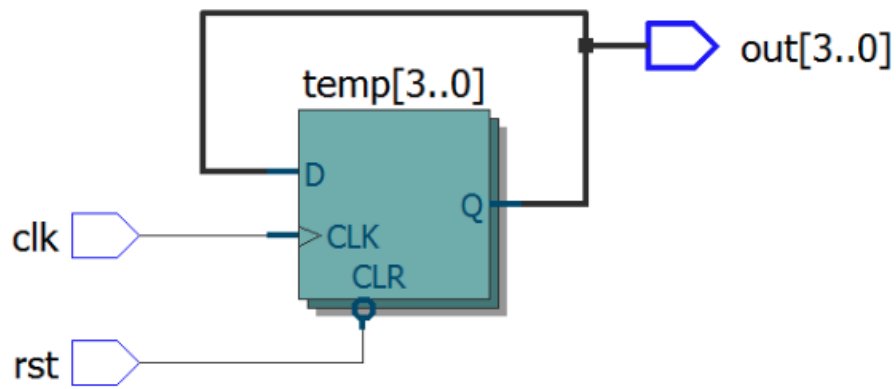
Figure 8: Johnson counter RTL view

Figure 9: Schematic entry synchronous counter RTL view

7.1.4 Waveforms
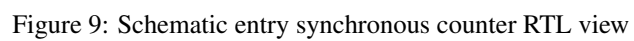
We simulated our designs using iverilog & gtkwave

Figure 10: Ripple-up counter waveform in gtkwave
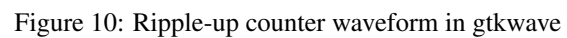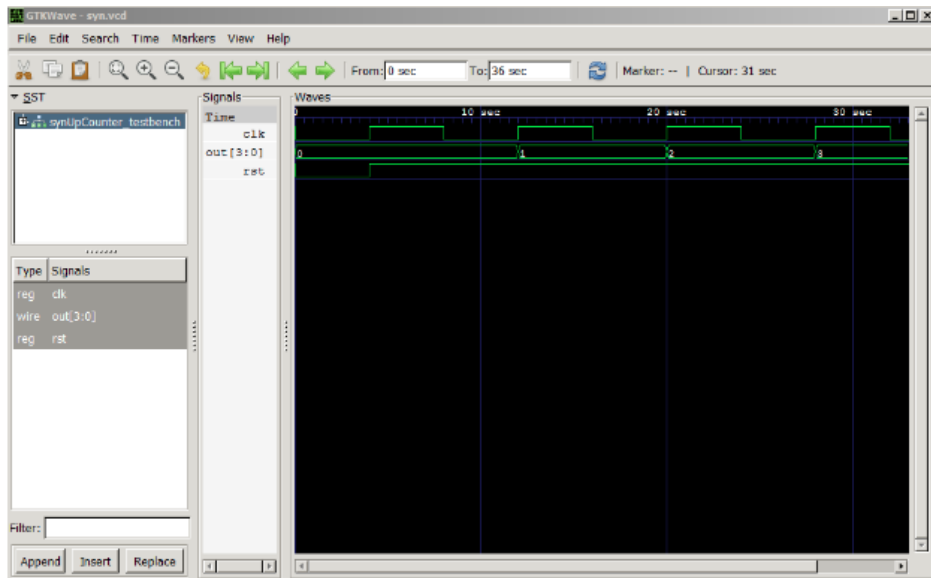
Figure 11: Synchronous up counter waveform
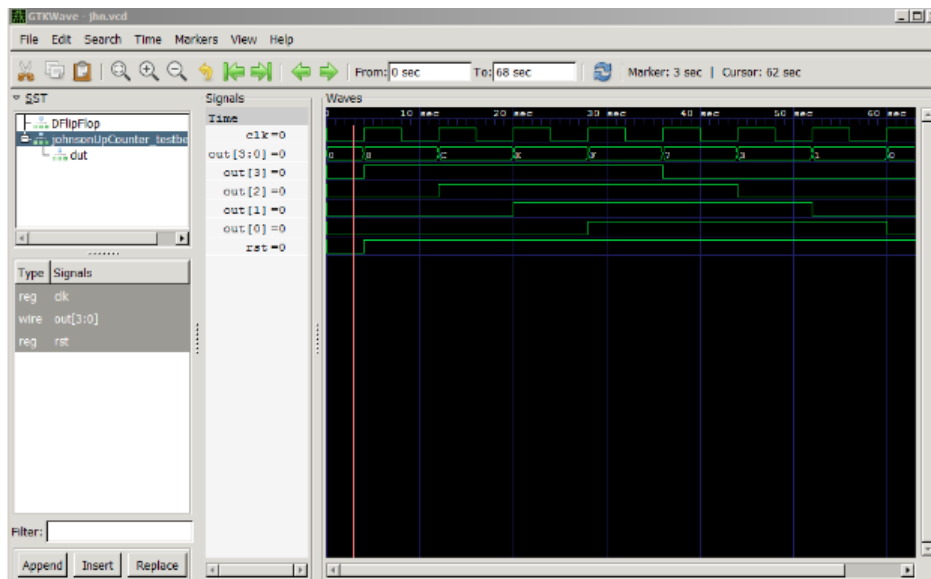
Figure 12: Johnson counter waveform in gtkwave
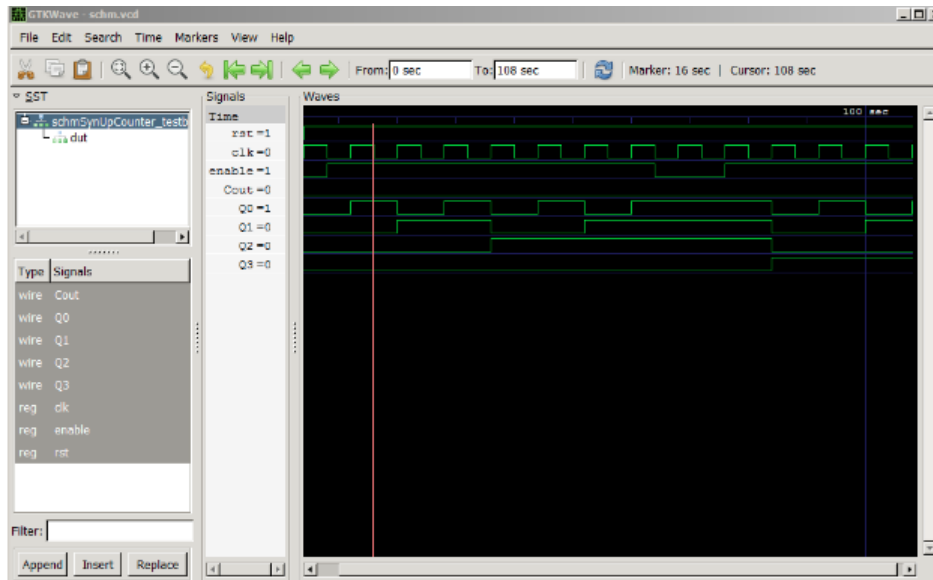
Figure 13: Synchronous up counter (schematic entry) waveform in gtkwave

7.2 Signal Tap II

We checked our designs in real-time using Signal Tap II
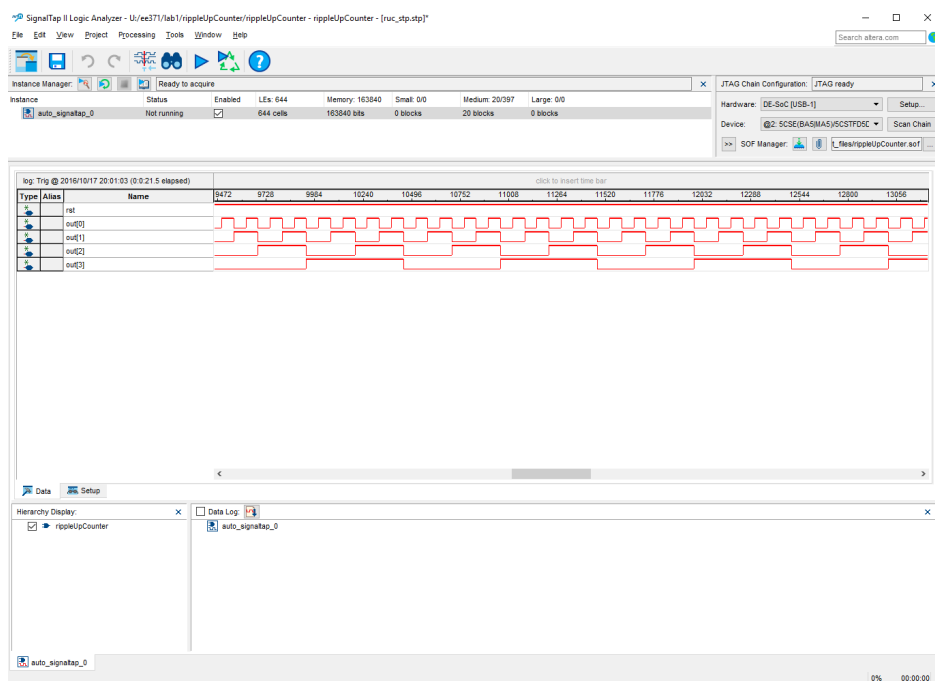
Figure 14: Signal Tap II on ripple-up counter

Ripple-up counter Signal Tap II waveform
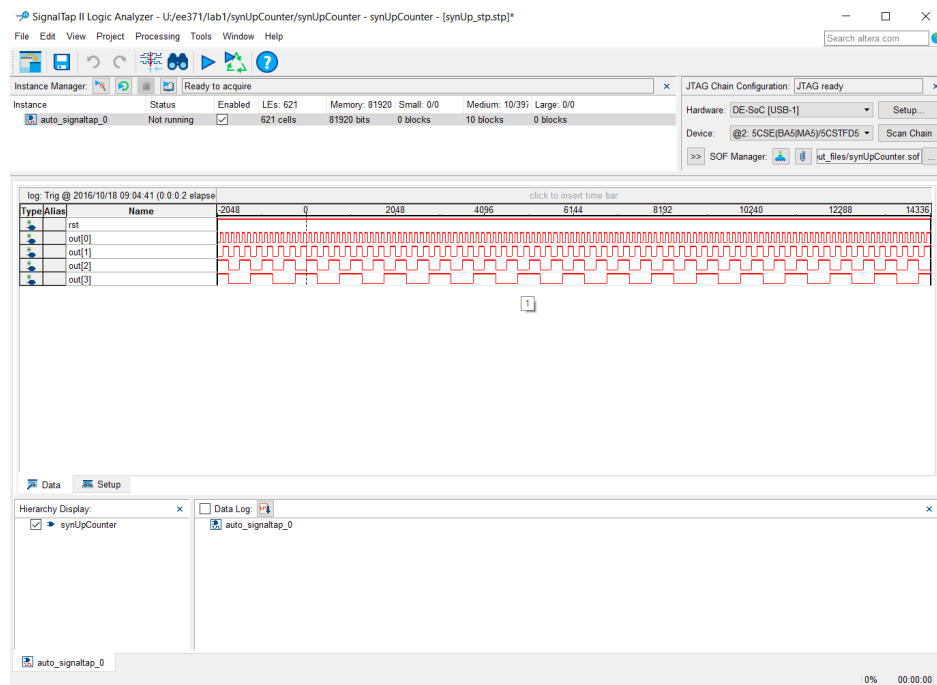
Figure 15: Signal Tap II on synchronous up counter

Synchronous up counter Signal Tap II waveform
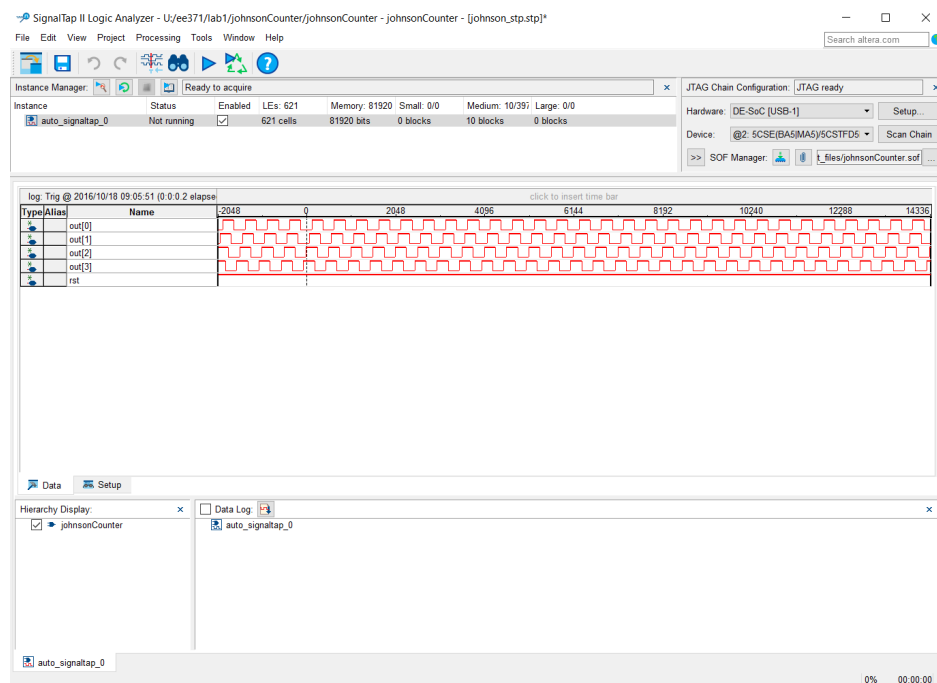
Figure 16: Signal Tap II on johnson counter

Johnson counter Signal Tap II waveform
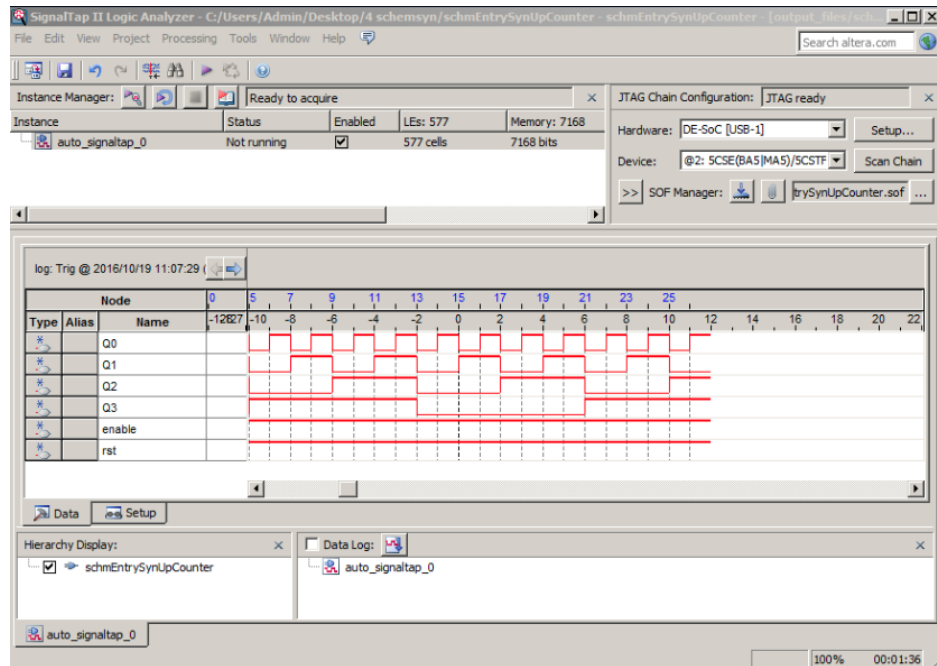
Figure 17: Signal Tap II on schematic entry synchronous up counter

Schematic entry synchronous up counter Signal Tap II waveform

7.3 iverilog & gtkwave

We were provided verilog code to test, as an introduction to iverilog and gtkwave.

```verilog
// andOr0.v
// Compute the logical AND and OR of inputs A and B.
module AndOr(AandB, AorB, A, B);
  output [1:0] AandB, AorB;
  input [1:0] A, B;

  and myAnd [1:0] (AandB[1:0], A[1:0], B[1:0]);
  or myOr [1:0] (AorB[1:0], A[1:0], B[1:0]);
endmodule
```

```verilog
// andorTop0.v
`include "andOr0.v"

module testBench;
  // connect the two modules
  wire [1:0] X, Y;
  wire [1:0] XandY, XorY;

  // declare an instance of the AND module
  AndOr myAndOr (XandY[1:0], XorY[1:0], X[1:0], Y[1:0]);

  // declare an instance of the testIt module
  Tester aTester (X[1:0], Y[1:0], XandY[1:0], XorY[1:0]);

  // file for gtkwave
  initial
```

25

```verilog
  begin
    // these two files support gtkwave and are required
    $dumpfile("andor0.vcd");
    $dumpvars(1,myAndOr);
  end
endmodule

module Tester (xOut, yOut, XandYin, XorYin);
  input [1:0] XandYin, XorYin;
  output [1:0] xOut, yOut;
  reg [1:0] xOut, yOut;

  parameter stimDelay = 20;

  initial // Response
  begin
    $display("\t\t xOut yOut \t XandYin XorYin \t Time");
    $monitor("\t\t %b\t %b \t %b \t %b", xOut, yOut, XandYin, XorYin, $time);
  end

  initial // Stimulus
  begin
    xOut = 'b00; yOut = 'b10;
    #stimDelay xOut = 'b10;
    #stimDelay yOut = 'b01;
    #stimDelay xOut = 'b11;

    #(2*stimDelay);
    $finish;
  end
endmodule
```

```verilog
module testBench;
  // connect the two modules
  wire inputs;
  wire outputs;

  // declare an instance of the MyDesign module MyDesign
  myDesign(outputs, inputs);

  // declare an instance of the Tester module Tester
  myTester (outputs, inputs);

  // file specifications for gtkwave
  initial
  begin
    // dump file is for dumping all the variables in a simulation
    $dumpfile("gfxFile.vcd");

    // dumps all the variables in module myDesign and below
    // but not modules instantiated in myDesign into the dump file.
    $dumpvars(1,myDesign);
  end
endmodule
```
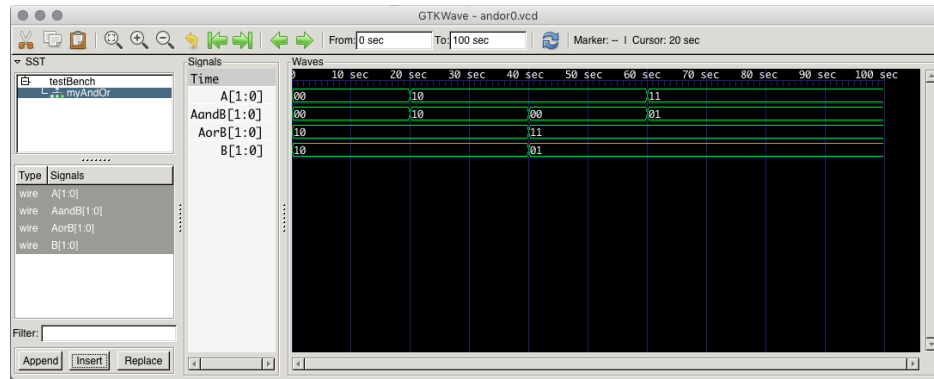
Figure 18: Results of iverilog & gtkwave

7.4 Learning the C language

The car listPrice C program we wrote, and the console output results

```c
/*
  EE 371 Lab 1
  Car Calculator Program

  The listPrice class estimates the user's desired car price
  given manufacturing costs, markup price, sales tax, and
  pre-tax discount price.

  @author(s) William Li, Jun Park, Dawn Liang
*/

#include <stdio.h>

int main(void) {
  // Variables to hold the required values to be calculated
  double in_price;
  double markup_price;
  double tax_percentage;
  double discount_price;
  double result;

  // User prompt and data entry
  printf("Please enter the manufacture's cost of your desired car ($): ");
  scanf("%lf", &in_price);
  printf("Please enter an estimated dealer's markup (%%): ");
  scanf("%lf", &markup_price);
  printf("Please enter the sales tax that you will need to pay (%%): ");
  scanf("%lf", &tax_percentage);
  printf("Please estimate the pre-tax discount you will receive (%%): ");
  scanf("%lf", &discount_price);

  // calculate list price & print
  result = in_price * (1 + markup_price / 100) *(1 - discount_price / 100) * (1 +
      tax_percentage / 100);
  printf("Your estimated price of your vehicle is %2.2f\n", result);
  getchar();
  return 0;
}
```
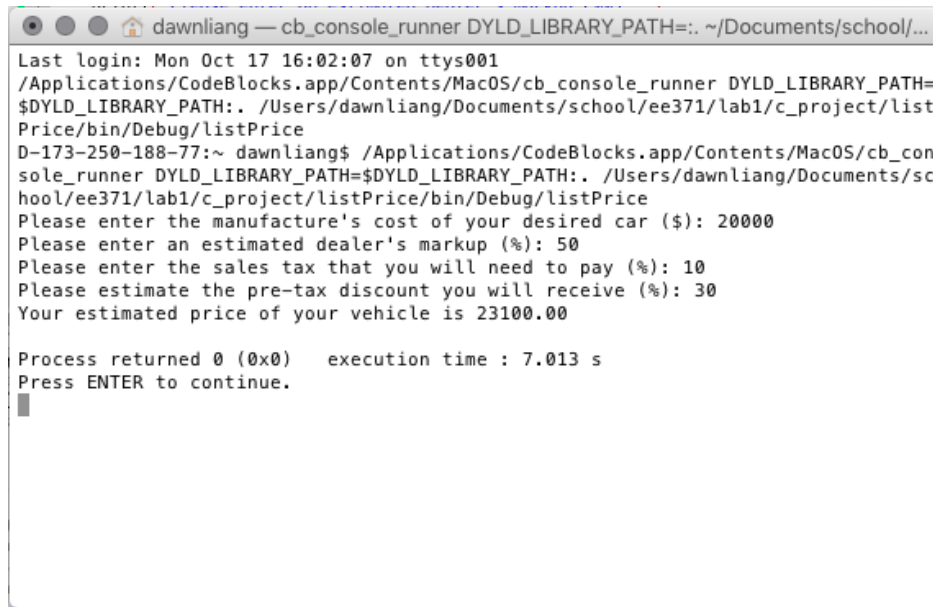
27

Figure 19: listPrice program results

We were also required to write a detailed specification document for our car list price calculator.

7.4.1 Requirements Documentation

Abstract   The calculator program is a project that calculates the users desired car price given four estimated criteria: manufactures cost, dealer markup price, sales tax, and pretax discount. The estimated information is then taken in, and processed using a simple equation. Our program results had matched up to our expectations, outputting the correct value given the four estimated costs as input.

Introduction   This project gave us an overview to the basics of C programming and the CodeBlocks IDE. The result of this project was a program that allows the user to be able to estimate his/her car price based on a few estimated values.

Inputs

1. The manufactures cost of the vehicle chosen by the user. This input is used to help give the initial base price to what the user is expected to pay.

2. The estimated dealers markup price chosen by the user. This input is used to help give the user an idea of how much more he/she needs to pay in addition to the base price.

3. The estimated sales tax chosen by the user. This input is used to figure out the additional costs associated from tax levied by the government on the vehicle.

4. The dealer discount chosen by the user. This input is used to help reduce the total price (before tax) of the car the user will pay.

Outputs   The program will output the total price of the vehicle after the user inputs the manufacturing costs, estimated dealer markup, the sales tax, and the dealer discount.

Major Functions   This program gives the user an estimation of the total price of the vehicle based on four estimated input cost values. The estimated input values are manufacturing costs, estimated dealer markup, sales tax and the dealer discount.

28

7.4.2 Design Specification

Abstract   The listPrice program is a project that calculates the users desired car price given four estimated criteria: manufactures cost, dealer markup price, sales tax, and pretax discount. The estimated information was then taken in, and processed using a simple equation. Written in C and the CodeBlocks IDE, listPrice is our first program that provides taught us the fundamental basics of how to program in C. Our program results have matched up to our expectations, outputting the correct value given the four estimated costs as input.

Introduction   This project gave us an overview to the basics of C programming and the CodeBlocks IDE. The result of this project was a program that allows the user to be able to estimate his/her car price based on a few estimated values.

Inputs   All values entered into listPrice must be entered as doubles. If any other value is entered (for instance, a string or a character), then the program will fail and not output the correct calculated answer.

1. Manufactures cost of the vehicle: must be a positive number, and cannot be 0. If it is less than 0, then the output of the total car price will be negative.

2. Estimated dealer markup: must be greater than or equal to 0, and entered as a percentage between 0 and 100 (for instance 5% markup is entered as a 5). If the markup is less than 0, then the dealer will be losing money.

3. Estimated Sales Tax: must be greater than or equal to 0, and entered as a percentage that is greater than 0 (for instance 9.7% sales tax is entered in as 9.7). If the tax is less than 0, then the user will receive an unintentional discount from the government.

4. Dealer Discount: must be greater or equal to 0, and entered as a percentage between 0 and 100 (for instance a 2% discount is entered as a 2). If the discount is less than 0, then the final price of the vehicle will be greater than if no discount had been applied. If the dealer discount is greater than 100, then the resulting vehicle price will be negative.

Outputs   Outputs the total calculated cost of the vehicle as a double. The output is expected to be greater than 0. If the output is negative, then one or more of the inputs were incorrectly inputted into the program.

Major Functions   The listPrice program calculates the total cost of the vehicle given four estimated price inputs. Each input is taken in as a double, and then computed using the formula:

$$list\_price = in\_price \times (1 + \frac{markup\_price}{100}) \times (1 - \frac{discount\_price}{100}) \times (1 + \frac{tax\_percentage}{100})$$

The output of the vehicle cost is the total cost that the user should expect to the pay if he/she were to purchase that particular vehicle.