EE 371 Autumn 2016 - Lab 1

William Li, Jun Park, Dawn Liang

October 17, 2016

Contents

1 ABSTRACT

This lab focuses on introducing us to the tools and methods of digital design. We were introduced to the various levels of abstraction in modeling and implementation, the tools involved in designing and testing hardware applications, as well as the overall design process for hardware and software and applications.

2 INTRODUCTION

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

Design Specification We were to build four different counters: a ripple-up counter, two synchronous up counters, and a Johnson counter, all of which counted every clock edge. Each counter used an active-low reset. The first three were directly written in Verilog, the fourth had to be built using Quartus' schematic entry feature. The ripple-up and synchronous counters count up in binary, while the Johnson counter counts up by the most significant bit.

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.

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, which included preset D-flipflops.

```
// 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.

The first synchronous up counter was implemented at the data flow level. We diagrammed the states and generated boolean expressions using truth tables and K-maps, depicted below. We then drew the gate-level diagram from those expressions.

The johnson counter was implemented at the behavioural level. All behaviour was described within the verilog code. To draw our gate-level diagram, we diagrammed the states and generated boolean statements using truth tables and K-maps.

The second synchronous up counter was built via schematic entry, a feature of Quartus II. The verilog code was then generated from the schematic.

System Description The four counters take a clock and reset as the input. For one of the synchronous counters, built with Quartus' schematic entry, we included an 'enable' signal input, which acts as an pause/resume switch. All four counters output 4-bits, counting up, to LEDs 3-0 of the DE1-SoC 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.

3.2 Test

Test Plan The counters need to be tested to see that they count each clock edge, and reset to 0 when the active-low reset was triggered. The asynchronicity of the clock and reset signal creates the risk of metastable behaviour, which must be checked for each counter.

The listPrice calculator must be tested to see how it handle various types of inputs. Its output must be verified against external calculations to ensure it follows the correct calculations.

Test Specification The counters were both simulated on our computers and tested on the physical board. Each possible state was simulated, in regards to the reset, the 4 bits, and in the case of the schematic synchronous counter, the enable signal.

The listPrice calculator's inputs were tested against possible edge cases, such as negative values or invalid inputs like strings. Additionally, expected inputs were also tested, to verify the mathematical validity of the program.

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). 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. 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. All evidence shows that our counter designs worked.

Between the three different
Compare and contrast the structural, dataflow, and behavioural level Quartus synthesized gate level implementations of the three counters with their gate level equivalents that you designed. Are there differences and if so, what are they and are they important? Why or why not?
1. In the Quartus II IDE, setup and configure the Signal Tap II logic analyzer to display the inputs and outputs of each of the four counters you designed in part 1 as they are running on the DE1-SoC board. Configure the Signal Tap II to trigger on the trailing (rising) edge of the input reset signal. 2. Configure the Signal Tap II to trigger on and capture and display data following the third state in each of your counters. 3. How do the displayed signal outputs compare with the ideal signals?

Our listPrice calculator design was also successful. In the case of expected inputs, the program behaves as it should. 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

We performed failure modes and effects analysis for three counters.

Ripple-up counter

Synchronous up counter

Johnson counter

We encountered many issues when trying to get Signal Tap II working.

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 Verilog code

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

```
/*
   4-bit ripple-up counter, implemented using D-flipflops

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

module rippleUpCounter(out, clk, rst);
   // declare inputs/outputs
   output logic [3:0] out;
   input logic clk, rst;
   logic Q0, Q1, Q2, Q3, clkTemp1, clkTemp2, clkTemp3, clkTemp4;

   // connect flip flops; one ff per bit
   DFlipFlop d0(.q(Q0), .qBar(clkTemp1), .D(clkTemp1), .clk(clk), .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
```

```
/*
   Ripple-up counter tester

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

`include "rippleUpCounter.v"
module rippleUpCounter_testbench;
   // declare variables
   logic [3:0] out;
   logic clk, rst;

   // module declaration
   rippleUpCounter dut(.out, .clk, .rst);

   // test module
   parameter PERIOD = 100; // period = length of clock
   initial begin
      clk <= 0;
      forever #(PERIOD/2) clk = ~clk;
   end

   initial begin
      rst=0; @(posedge clk);
      rst=1; @(posedge clk);
   end
```

```
endmodule
```

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

```
/*
   4-bit synchronous up counter, implemented using D-flipflops at
   dataflow level

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

module synUpCounter(out, clk, rst);
   // declare inputs/outputs
   output logic [3:0] out;
   input logic clk, rst;

   // connecting wires
   logic Q0, Q1, Q2, Q3;
   logic D0, D1, D2, D3;

   // D-ff connections
   DFlipFlop dff0(.q(Q0), .qBar(), .D(D0), .clk(clk), .rst(rst));
   DFlipFlop dff1(.q(Q1), .qBar(), .D(D1), .clk(clk), .rst(rst));
   DFlipFlop dff2(.q(Q2), .qBar(), .D(D2), .clk(clk), .rst(rst));
   DFlipFlop dff3(.q(Q3), .qBar(), .D(D3), .clk(clk), .rst(rst));

   // output logic assignments
   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;

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

```
/*
   4-bit synchronous up counter tester

   Authors: William Li, Dawn Liang, Jun Park
   Date: 16 Oct 2016
*/
module synUpCounter_testbench;
   // declare variables
   logic [3:0] out;
   logic clk, rst;

   // declare module
   synUpCounter dut(.out, .clk, .rst);

   // test states
   parameter PERIOD = 100; // period = length of clock
   initial begin
      clk <= 0;
      forever #(PERIOD/2) clk = ~clk;
   end
```

```
   initial begin
      rst=0; @(posedge clk);
      rst=1; @(posedge clk);
             @(posedge clk);
             @(posedge clk);
             @(posedge clk);
             @(posedge clk);
             @(posedge clk);
             @(posedge clk);
             @(posedge clk);
             @(posedge clk);
             @(posedge clk);
             @(posedge clk);
             @(posedge clk);
             @(posedge clk);
             @(posedge clk);
             @(posedge clk);
             @(posedge clk);
             @(posedge clk);
             @(posedge clk);
      rst=0; @(posedge clk);
             @(posedge clk);
             @(posedge clk);
             @(posedge clk);
      $stop();
   end
endmodule
```

Johnson Counter The johnson counter, implemented at the behavioural level

```
/*
   4-bit Johnson up counter, implemeneted using D-flipflops at the
   behavioural level

   Authors: William Li, Dawn Liang, Jun Park
   Date: 16 Oct 2016
*/
module johnsonUpCounter(out, clk, rst);
   // declare inputs/outputs & variables
   output logic [3:0] out;
   input logic clk, rst;
   logic [3:0] temp;

   // counting every clock edge
   always_ff@(negedge rst or posedge clk) begin
      if (rst == 0)
         begin
            temp <= 4'b0000;
         end
      else if (clk == 1'b1)
         begin
            temp <= {~temp[0], temp[3:1]}; // right shift and negate most signf bit
         end
   end

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

```
/*
   Testing module for johnson up counter

   Author: Jun Park
   Date: 16 Oct 2016
*/
module johnsonUpCounter_testbench;
   logic [3:0] out;
   logic clk, rst;

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

   parameter PERIOD = 100; // period = length of clock
   initial begin
      clk <= 0;
      forever #(PERIOD/2) clk = ~clk;
   end

   initial begin
      rst=0; @(posedge clk);
      rst=1; @(posedge clk);
             @(posedge clk);
             @(posedge clk);
             @(posedge clk);
             @(posedge clk);
             @(posedge clk);
             @(posedge clk);
             @(posedge clk);
             @(posedge clk);
             @(posedge clk);
             @(posedge clk);
             @(posedge clk);
             @(posedge clk);
             @(posedge clk);
      rst=0; @(posedge clk);
             @(posedge clk);
             @(posedge clk);
      $stop();
   end
endmodule
```

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

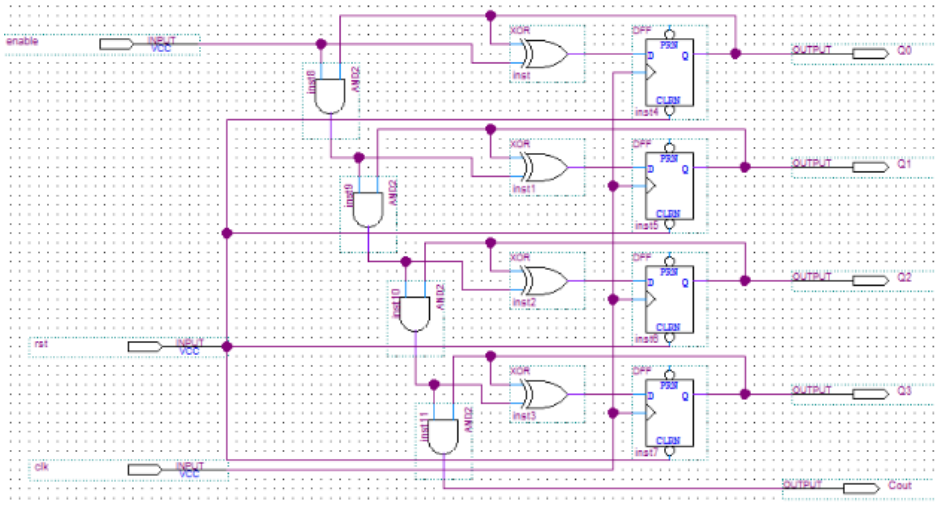

Figure 1: Synchronous counter schematic entry

The generated verilog code

```
module schematicEntrySynUpCounter(
   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_10;
reg SYNTHESIZED_WIRE_11;
wire SYNTHESIZED_WIRE_12;
reg SYNTHESIZED_WIRE_13;
wire SYNTHESIZED_WIRE_14;
reg SYNTHESIZED_WIRE_15;
wire SYNTHESIZED_WIRE_16;

wire SYNTHESIZED_WIRE_5;
wire SYNTHESIZED_WIRE_6;
wire SYNTHESIZED_WIRE_7;
wire SYNTHESIZED_WIRE_8;

assign Q0 = SYNTHESIZED_WIRE_10;
assign Q1 = SYNTHESIZED_WIRE_11;
assign Q2 = SYNTHESIZED_WIRE_13;
```

```
assign Q3 = SYNTHESIZED_WIRE_15;

assign SYNTHESIZED_WIRE_5 = SYNTHESIZED_WIRE_10 ^ enable;
assign SYNTHESIZED_WIRE_6 = SYNTHESIZED_WIRE_11 ^ SYNTHESIZED_WIRE_12;
assign SYNTHESIZED_WIRE_16 = SYNTHESIZED_WIRE_13 & SYNTHESIZED_WIRE_14;
assign Cout = SYNTHESIZED_WIRE_15 & SYNTHESIZED_WIRE_16;
assign SYNTHESIZED_WIRE_7 = SYNTHESIZED_WIRE_13 ^ SYNTHESIZED_WIRE_14;
assign SYNTHESIZED_WIRE_8 = SYNTHESIZED_WIRE_15 ^ SYNTHESIZED_WIRE_16;

always@(posedge clk or negedge rst)
begin
if (!rst)
   begin
   SYNTHESIZED_WIRE_10 <= 0;
   end
else
   begin
   SYNTHESIZED_WIRE_10 <= SYNTHESIZED_WIRE_5;
   end
end


always@(posedge clk or negedge rst)
begin
if (!rst)
   begin
   SYNTHESIZED_WIRE_11 <= 0;
   end
else
   begin
   SYNTHESIZED_WIRE_11 <= SYNTHESIZED_WIRE_6;
   end
end


always@(posedge clk or negedge rst)
begin
if (!rst)
   begin
   SYNTHESIZED_WIRE_13 <= 0;
   end
else
   begin
   SYNTHESIZED_WIRE_13 <= SYNTHESIZED_WIRE_7;
   end
end

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

```
assign SYNTHESIZED_WIRE_12 = SYNTHESIZED_WIRE_10 & enable;
assign SYNTHESIZED_WIRE_14 = SYNTHESIZED_WIRE_11 & SYNTHESIZED_WIRE_12;

endmodule
```

7.1.2 Waveforms

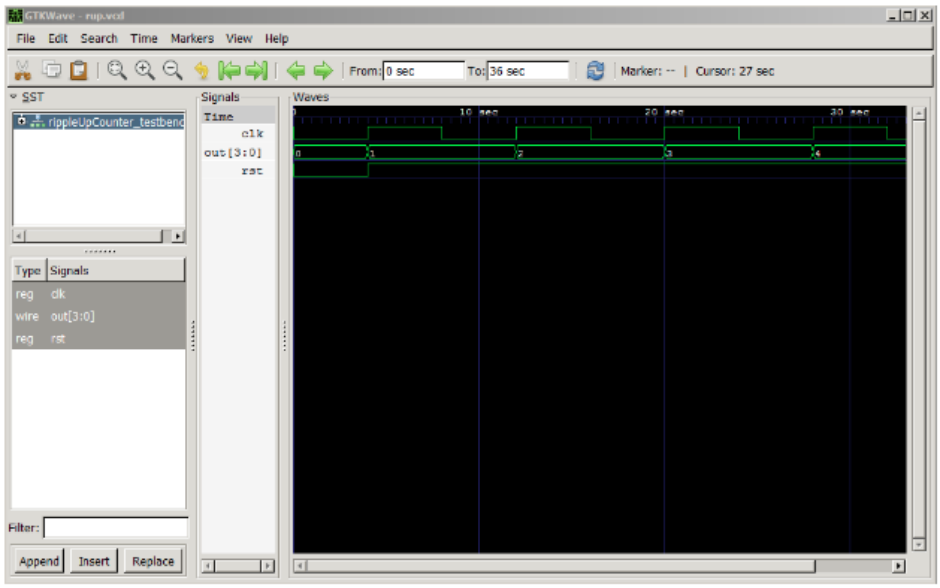

Figure 2: Ripple-up counter waveform in gtkwave

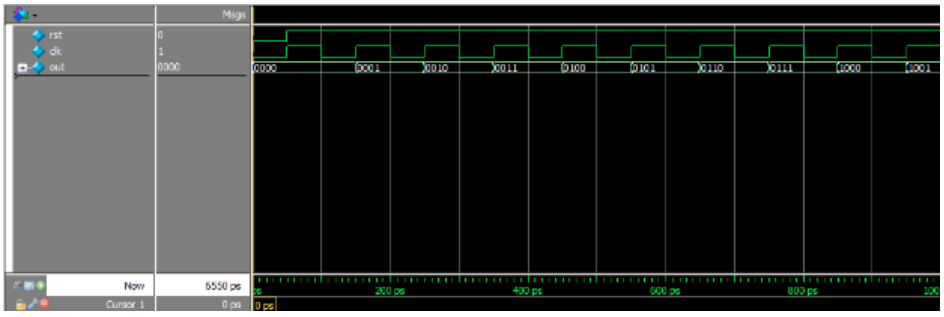

Figure 3: Synchronous up counter waveform

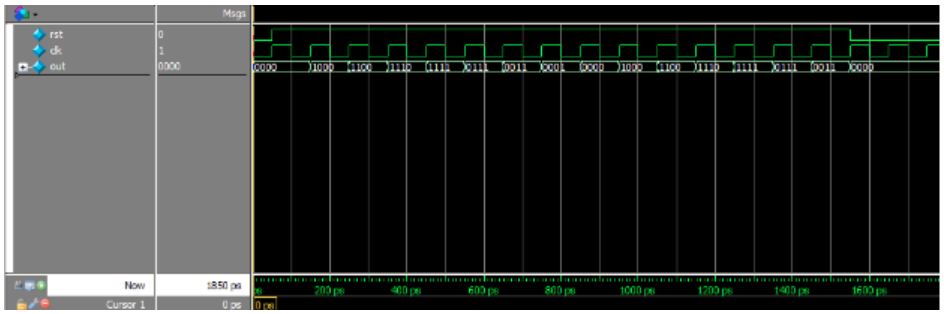

Figure 4: Johnson counter waveform in gtkwave

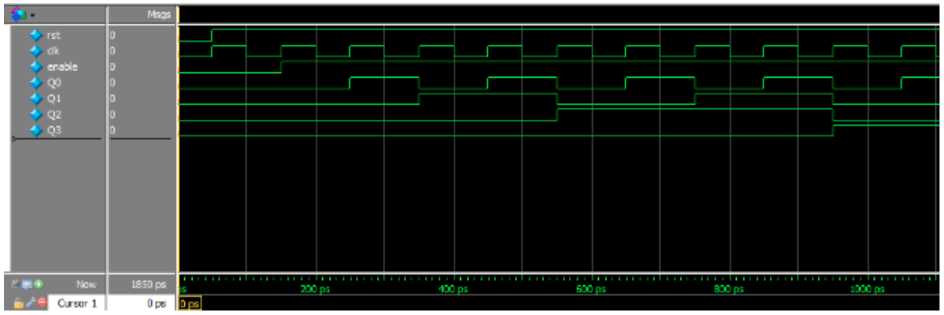

Figure 5: Synchronous up counter (schematic entry) waveform in gtkwave

7.1.3 RTL views

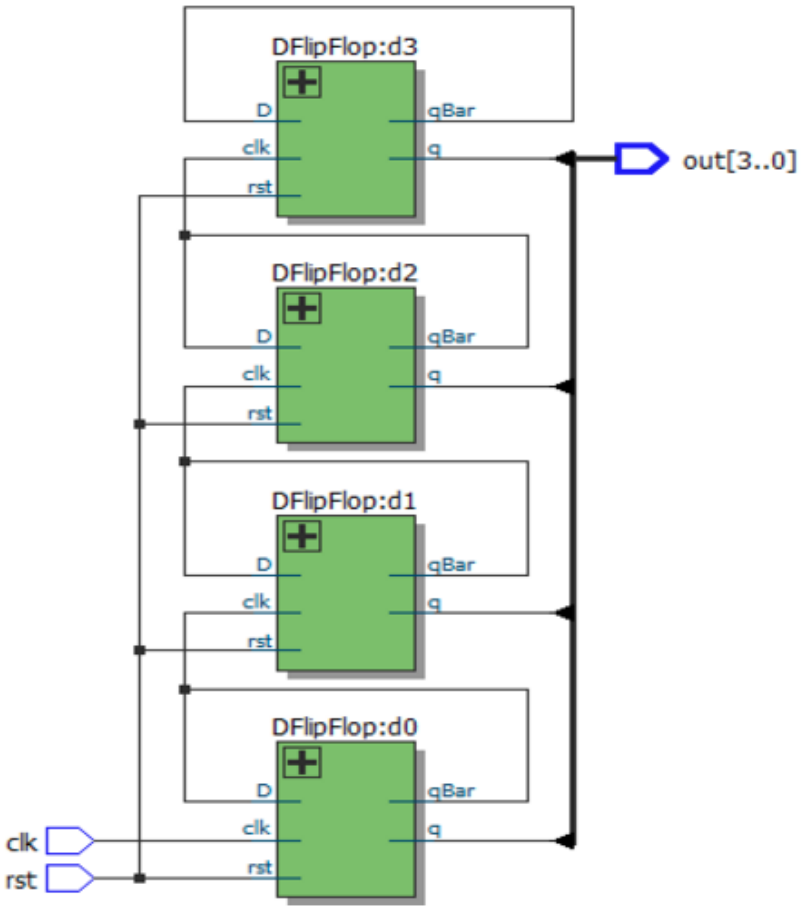

Figure 6: Ripple-up counter RTL view

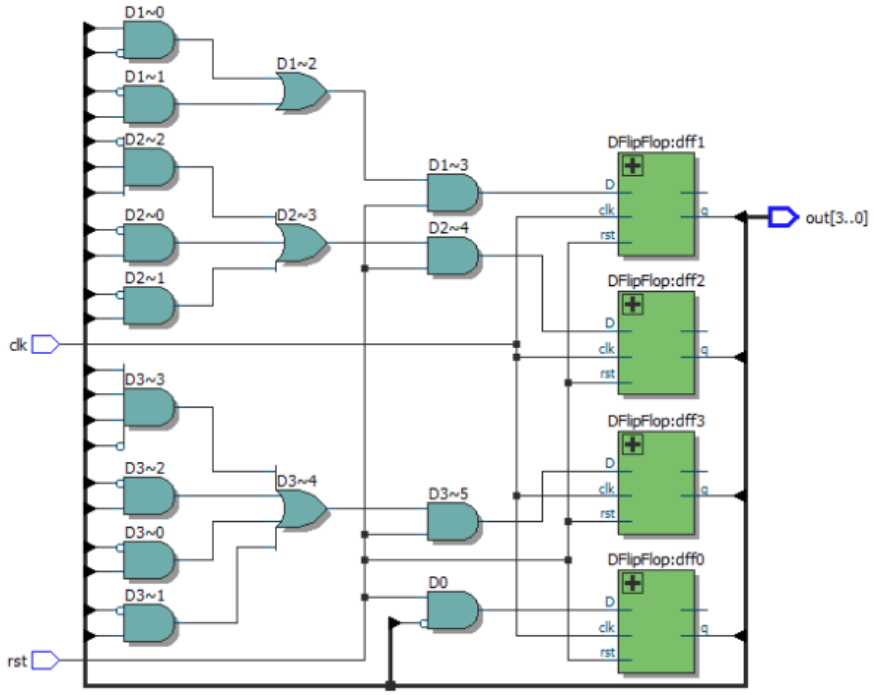

Figure 7: Ripple-up counter RTL view

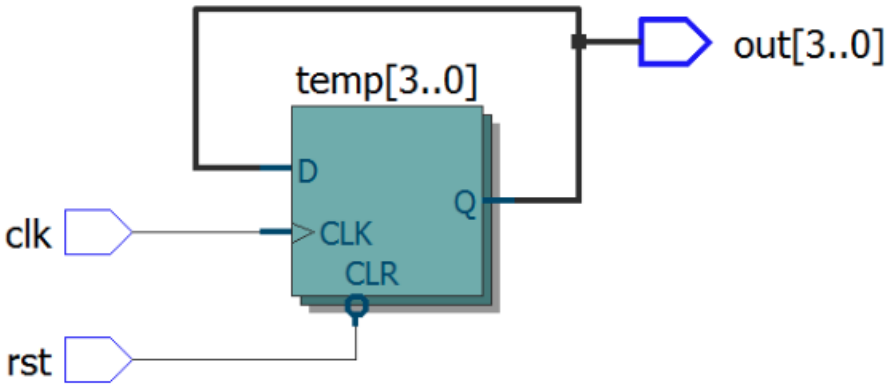

Figure 8: Ripple-up counter RTL view

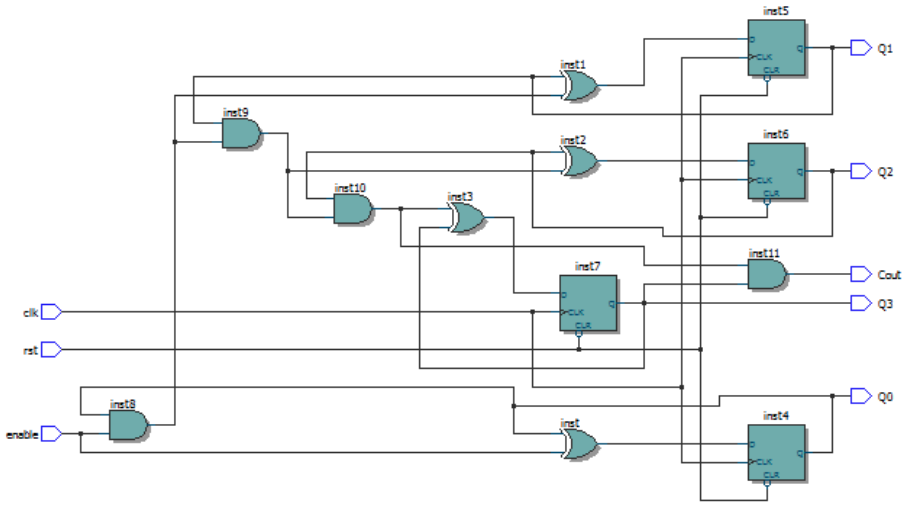

Figure 9: Ripple-up counter RTL view

7.2 iverilog & gtkwave

We were provided verilog code to test, in order to familiarise ourselves with iverilog and gtkwave.

```
// 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
```

```
// 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
   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
```

```
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
```

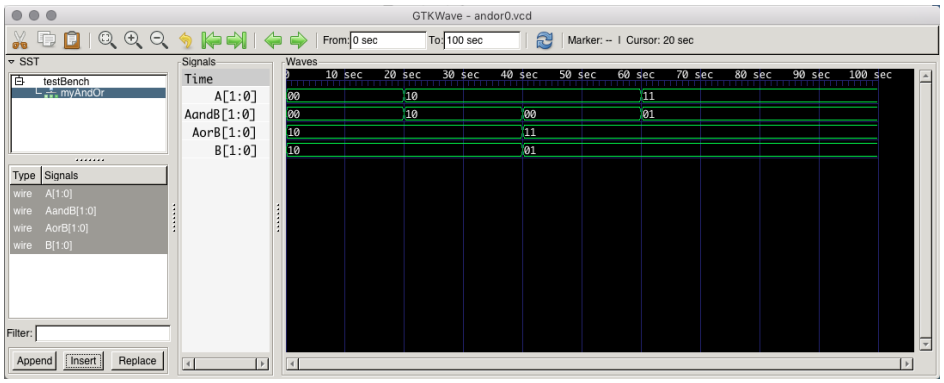

Figure 10: Results of iverilog & gtkwave

7.3 Learning the C language

The car listPrice C program we wrote

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