This document contains the write-up for all three components of the Fall 2010 Midterm for ECE 57D.

**8-bit Magnitude Comparator** (Part A)

The first part of this midterm exam asks to 1st design a single-bit cascadable comparator which will d contain greater than (GT) and equality (EQ) functionality. The following signals will be treated as control inputs: ***ai***, ***bi***, ***eq***, and ***gt***. Outputs from the comparator will be **EQ** and **GT**. An important restriction is that only the use of **NAND** gates is allowed in the design of the circuit. Eventually, C++ will be used to model the circuit and to create a testbench in order to prove full functionality.

First and foremost, it is important to develop the schematic for the 1 bit comparator element:



The schematic above is being split into stages so that it will be much easier to translate this image into C++ code, as the gates in code are sequence dependant. As can be seen, the NAND gate is versatile in lieu of not being able to use other gates for this assignment. S1 is a NAND implementation of a NOT gate, stages S5 & S6 for EQ are an implementation of AND, and stage S5 and S6 for GT are an implementation of OR. The implementation above will be directly translated into C++ code.

We are then asked to design an 8-bit comparator. As this is the case, the design will cascade the above 1 bit comparator elements, as seen in the diagram below:



It is important to note that this schematic is to be read right from left, with the 1st 1-bit comparator on the right representing the MSB (most significant bit) of the 8-bit input stream. By default, so that the first 1 bit comparator is initialized correctly, **gt7** will = 0 and **eq7** = 1.

With all schematics in place, we can now begin to code this circuit in C++. The following five (5) C++ files were generated in order to implement this design:

*logicPrimitives.h, logicPrimitives.cpp*

*comparator.h, comparator.cpp*

*compTester.cpp*

The *logicPrimitives.h* file takes care of defining the *wire* and *nand* gate classes:

// Carlos Lazo

// ECE579D

// Midterm Exam - Part A

#include <fstream>

#include <iostream>

#include <string>

using namespace std;

#define MAX2(a,b)((a < b) ? b : a);

#define MAX3(a,b,c) (c>((a>b)?a:b))?c:((a>b)?a:b);

const int WIRE\_DELAY = 1;

// Define the wire class, which will hold logic value and timing.

// By default, wires will have a timing delay of 1

class wire {

char value;

int delay;

public:

wire () {value = 'X'; delay = WIRE\_DELAY;}

wire (char v, int d) {value = v; delay = d;}

void put (char v, int d) {value = v; delay = d;}

void get (char& v, int& d) {v = value; d = delay;}

int delayValue() {return this->delay;}

char getVal() {return this->value;}

};

// Define a 2-input NAND gate

class nand {

wire i1, i2, o1;

int gateDelay;

public:

nand () {gateDelay = 2;} // Gate Delay = 2, since this is a 2 input NAND gate.

int delayValue() {return this->gateDelay;}

void NAND (wire, wire, wire&);

};

The *logicPrimitives.cpp* fileprovides all definitions needed to instantiate the basic circuitry components:

// Carlos Lazo

// ECE579D

// Midterm Exam - Part A

#include "logicPrimitives.h"

// Define the NAND function

void nand::NAND(wire i1, wire i2, wire& o1)

{

char av, bv, wv;

int ad, bd, wd;

i1.get(av, ad);

i2.get(bv, bd);

// Apply all logic possibilities of NAND

if ((av == '1') && (bv == '1'))

wv = '0'; // Value = 0

else

wv = '1'; // Value = 1

// Setup gate delay based on largest total delay seen.

// Be sure to add in both the gate delay and wire delay

wd = this->delayValue() + WIRE\_DELAY + MAX2(ad, bd);

o1.put(wv, wd); // Place calculated value on output wire

}

The *comparator.h* fileprovides definitions for the *comp\_elem* and *comp\_8bit* classes, which represent both the single comparator element and the 8 bit cascaded comparator shown in the circuit diagrams at the beginning of this section:

// Carlos Lazo

// ECE579D

// Midterm Exam - Part A

#ifndef COMPARATOR\_H

#define COMPARATOR\_H

#include "logicPrimitives.h"

// Define the comp\_elem Class

class comp\_elem {

private:

wire ai, bi, eq, gt, EQ, GT;

public:

// Constructors:

comp\_elem ();

// Declare functions:

void proc (wire a, wire b, wire e, wire g, wire& E, wire& G);

};

// Define the comp\_8bit Class

class comp\_8bit {

private:

comp\_elem c\_array[8];

wire a[8], b[8], e[8], g[8];

string a\_s, b\_s;

public:

// Constructors:

comp\_8bit ();

void load\_in (string astr, string bstr);

void output();

};

#endif

The *comparator.cpp* fileprovides definitions for all the *comp\_elem* and *comp\_8bit* functions. As can be seen, the gate-level implementation of the *comp\_elem* matches exactly that of the intermediate NAND gates described above. Afterwards, the logic portion of the 8 bit comparator is built such that 2 8-bit strings are taken in and read into temporary memory. Using a simple for loop, the information is cascaded through the individual comparators until a solution is determined:

// Carlos Lazo

// ECE579D

// Midterm Exam - Part A

#include "comparator.h"

// Declare wires of Logic 0 and Logic 1 for later use

wire LOGIC\_0 ('0', WIRE\_DELAY);

wire LOGIC\_1 ('1', WIRE\_DELAY);

// Empty default constructor

comp\_elem::comp\_elem() {}

// Define function comp\_elem, which will perform NAND gate logic

void comp\_elem :: proc(wire a, wire b, wire e, wire g, wire& E, wire& G)

{

// Place all values from incoming wires into comparator memory -

ai.put(a.getVal(), a.delayValue()); bi.put(b.getVal(), b.delayValue());

eq.put(e.getVal(), e.delayValue()); gt.put(g.getVal(), g.delayValue());

// Stage 1 of Logic -

nand s1\_1 , s1\_2 ;

wire s1\_o1, s1\_o2;

s1\_1.NAND(ai, ai, s1\_o1);

s1\_2.NAND(bi, bi, s1\_o2);

// Stage 2 of Logic -

nand s2\_1 , s2\_2 ;

wire s2\_o1, s2\_o2;

s2\_1.NAND(bi, s1\_o1, s2\_o1);

s2\_2.NAND(ai, s1\_o2, s2\_o2);

// Stage 3 of Logic -

nand s3\_1 , s3\_2 ;

wire s3\_o1, s3\_o2;

s3\_1.NAND(s2\_o1, s2\_o2, s3\_o1);

s3\_2.NAND(s2\_o2, s2\_o2, s3\_o2);

// Stage 4 of Logic -

nand s4\_1;

wire s4\_o1;

s4\_1.NAND(s3\_o1, s3\_o1, s4\_o1);

// Stage 5 of Logic -

nand s5\_1 , s5\_2 , s5\_3 ;

wire s5\_o1, s5\_o2, s5\_o3;

s5\_1.NAND(eq, s4\_o1, s5\_o1);

s5\_2.NAND(gt, gt, s5\_o2);

s5\_3.NAND(s3\_o2, s3\_o2, s5\_o3);

// Stage 6 of Logic -

nand s6\_1 , s6\_2 ;

wire s6\_o1, s6\_o2;

s6\_1.NAND(s5\_o1, s5\_o1, E);

s6\_2.NAND(s5\_o2, s5\_o3, G);

// Set internal variables to final output as well:

EQ.put(E.getVal(), E.delayValue());

GT.put(G.getVal(), G.delayValue());

}

// Empty default constructor

comp\_8bit::comp\_8bit() {}

// This function will be used to initialize all ai and bi values of the comparator.

// It is important to note that ai[0] is the MSB and ai[7] is the LSB.

void comp\_8bit::load\_in (string astr, string bstr)

{

for (int i = 0; i < 8; i++)

{

a[i].put(astr[i], WIRE\_DELAY);

b[i].put(bstr[i], WIRE\_DELAY);

}

a\_s = astr; b\_s = bstr;

}

// Create the logic for the 8bit comparator.

void comp\_8bit::output()

{

// Using a for-loop, cascade all 1-bit comparators with one-another.

// Note that for the 1st comparator, we use predetermined values for eq & gt.

int flag = 1;

int comp\_num = 8;

for (int i = 0; i < 8; i++)

{

if (i == 0)

c\_array[0].proc (a[0], b[0], LOGIC\_1, LOGIC\_0, e[0], g[0]);

else

c\_array[i].proc(a[i], b[i], e[i-1], g[i-1], e[i], g[i]);

// Extra information regarding in which comparator EQ and GT were "really" determined:

if ((e[i].getVal() == '0' || g[i].getVal() == '1') && (flag == 1))

{

comp\_num = i + 1;

flag = 0;

}

}

cout << "The 8-bit value for A = " << a\_s << endl;

cout << "The 8-bit value for B = " << b\_s << endl;

cout << "\tIt was determined that GT = " << g[7].getVal()

<< " : Delay Time = " << g[7].delayValue() << endl;

cout << "\tIt was determined that EQ = " << e[7].getVal()

<< " : Delay Time = " << e[7].delayValue() << endl;

cout << "\t\tEQ and GT were determined at 1 bit comparator #" << comp\_num << endl << endl;

}

Finally, a testbench is created in *compTester.cpp* in order to prove the functionality of the circuit. It is crucial to note one important assumption that plays into the overall propagation delay of the system and forces it constant – it is assumed that all ***ai***[i] and ***bi***[i] values are available at once. For one comparator, it takes a total of **19** ns for the EQ delay and **16** ns for the GT delay. The amounts of the overall system delay will NOT be 19\*8 and 16\*8, but instead, they will be much smaller. Since all input is available at once, all 8-bit comparators can begin performing their operations, and merely have to wait on resolution from ***eq*** and ***gt*** from the previous comparator and that associated timing delay. In addition, the testbench prints the comparator at which the determination of **EQ** and **GT** were made:

// Carlos Lazo

// ECE579D

// Midterm Exam - Part A

#include "comparator.h"

// Main Testbench for 8-bit Comparator -

int main()

{

// Initialize 8-bit Comparator

comp\_8bit uut;

// Test #1 (EQ)

string a = "00000000";

string b = "00000000";

uut.load\_in(a,b);

uut.output();

// Test #2 (GT)

a = "01100001";

b = "01011110";

uut.load\_in(a,b);

uut.output();

// Test #3 (EQ)

a = "11111111";

b = "11111111";

uut.load\_in(a,b);

uut.output();

// Test #4 (GT)

a = "11100100";

b = "11100000";

uut.load\_in(a,b);

uut.output();

// Test #5 (0)

a = "11100111";

b = "11110100";

uut.load\_in(a,b);

uut.output();

// Test #6 (0)

a = "10100101";

b = "11100111";

uut.load\_in(a,b);

uut.output();

// Test #7 (GT)

a = "11101110";

b = "11101100";

uut.load\_in(a,b);

uut.output();

// Test #8 (GT)

a = "01100000";

b = "11100000";

uut.load\_in(a,b);

uut.output();

// Test #9 (0)

a = "00110111";

b = "00111111";

uut.load\_in(a,b);

uut.output();

// Test #10 (EQ)

a = "10101001";

b = "10101001";

uut.load\_in(a,b);

uut.output();

return 0;

}

The output from the testbench is seen below, and verifies the full operation of the 8-bit comparator with proven resolution and timing at all comparators:

The 8-bit value for A = 00000000

The 8-bit value for B = 00000000

It was determined that GT = 0 : Delay Time = 58

It was determined that EQ = 1 : Delay Time = 61

EQ and GT were determined at 1 bit comparator #8

The 8-bit value for A = 01100001

The 8-bit value for B = 01011110

It was determined that GT = 1 : Delay Time = 58

It was determined that EQ = 0 : Delay Time = 61

EQ and GT were determined at 1 bit comparator #3

The 8-bit value for A = 11111111

The 8-bit value for B = 11111111

It was determined that GT = 0 : Delay Time = 58

It was determined that EQ = 1 : Delay Time = 61

EQ and GT were determined at 1 bit comparator #8

The 8-bit value for A = 11100100

The 8-bit value for B = 11100000

It was determined that GT = 1 : Delay Time = 58

It was determined that EQ = 0 : Delay Time = 61

EQ and GT were determined at 1 bit comparator #6

The 8-bit value for A = 11100000

The 8-bit value for B = 11110000

It was determined that GT = 0 : Delay Time = 58

It was determined that EQ = 0 : Delay Time = 61

EQ and GT were determined at 1 bit comparator #4

The 8-bit value for A = 10100101

The 8-bit value for B = 11100111

It was determined that GT = 0 : Delay Time = 58

It was determined that EQ = 0 : Delay Time = 61

EQ and GT were determined at 1 bit comparator #2

The 8-bit value for A = 11101110

The 8-bit value for B = 11101100

It was determined that GT = 1 : Delay Time = 58

It was determined that EQ = 0 : Delay Time = 61

EQ and GT were determined at 1 bit comparator #7

The 8-bit value for A = 01100000

The 8-bit value for B = 11100000

It was determined that GT = 0 : Delay Time = 58

It was determined that EQ = 0 : Delay Time = 61

EQ and GT were determined at 1 bit comparator #1

The 8-bit value for A = 00110111

The 8-bit value for B = 00111111

It was determined that GT = 0 : Delay Time = 58

It was determined that EQ = 0 : Delay Time = 61

EQ and GT were determined at 1 bit comparator #5

The 8-bit value for A = 10101001

The 8-bit value for B = 10101001

It was determined that GT = 0 : Delay Time = 58

It was determined that EQ = 1 : Delay Time = 61

EQ and GT were determined at 1 bit comparator #8

Press any key to continue . . .

This concludes the analysis for Midterm Exam – Part A.

**3-Bit Up/Down Counter** (Part B)

The second part of this midterm exam asks to design a 3 bit Up/Down counter with a rising edge clock ***clk***, a synchronous reset ***rst***, and ***UpDown*** control. An important restriction is that only the use of **NAND** gates is allowed in the design of the circuit. Eventually, C++ will be used to model the circuit and to create a testbench in order to prove full functionality, using both an input and output external file.

We begin by designing all needed logic tables for the Up/Down counter, starting with state transitions:

It is important to note that this counter is being developed with a possibility of a direction change in counting. If the signal *UpDown* = 1, then the circuit will count as intended based on the figure above. If the signal *direction* = 0, then the circuit will count in reverse order based on the figure above. For the sake of visual simplicity, the variable *UpDown* has been renamed *d*. The following State Transition Table is then formed:

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Present State** | | |  | **Next State** | | | | | | |
|  | *d = 0* (DOWN) | | |  | *d = 1* (UP) | | |
| **Q2** | **Q1** | **Q0** |  | **Q2+** | **Q1+** | **Q0+** |  | **Q2+** | **Q1+** | **Q0+** |
| 0 | 0 | 0 |  | 1 | 1 | 1 |  | 0 | 0 | 1 |
| 0 | 0 | 1 |  | 0 | 0 | 0 |  | 0 | 1 | 0 |
| 0 | 1 | 0 |  | 0 | 0 | 1 |  | 0 | 1 | 1 |
| 0 | 1 | 1 |  | 0 | 1 | 0 |  | 1 | 0 | 0 |
| 1 | 0 | 0 |  | 0 | 1 | 1 |  | 1 | 0 | 1 |
| 1 | 0 | 1 |  | 1 | 0 | 0 |  | 1 | 1 | 0 |
| 1 | 1 | 0 |  | 1 | 0 | 1 |  | 1 | 1 | 1 |
| 1 | 1 | 1 |  | 1 | 1 | 0 |  | 0 | 0 | 0 |

In order to develop the Karnaugh maps for the individual D inputs, it is important to remember the Transition Table for the D Flip-Flop:



Utilizing both State Transition Tables, develop the Karnaugh maps for all D Flip-Flop inputs for the counting portion of the circuit:



From the Karnaugh maps above, the following expressions for the D Flip Flops are derived:

**D2 = ~d~ y2~y1~y0 + y2 y1~y0 + dy2~y1 + ~dy2y0 + d~y2 y1y0**

**D1 = ~dy1y0 + dy1~y0 + ~d~y1~y0 + d~y1y0**

**D0 = ~y1~y0  + y1~y0**

The following logic was drawn out by hand, with implementations using only NAND gates. It was determined that for **D2**, a total of 36 NAND gates will be used. It was determined that for **D1**, a total of 27 NAND gates will be used. It was determined that for **D0**, a total of 14 NAND gates will be used.

With all schematics in place, we can now begin to code this circuit in C++. The following five (5) C++ files were generated in order to implement this design:

*logicPrimitives.h, logicPrimitives.cpp*

*updownCounter.h, updownCounter.cpp*

*updownTester.cpp*

The *logicPrimitives.h* file takes care of defining the *wire*, *nand* gate, and *dff* classes:

// Carlos Lazo

// ECE579D

// Midterm Exam - Part B

#include <fstream>

#include <iostream>

#include <string>

using namespace std;

#define MAX2(a,b)((a < b) ? b : a);

#define MAX3(a,b,c) (c>((a>b)?a:b))?c:((a>b)?a:b);

const int WIRE\_DELAY = 1;

// Define the wire class, which will hold logic value and timing.

// By default, wires will have a timing delay of 1

class wire {

char value;

int delay;

public:

wire () {value = 'X'; delay = WIRE\_DELAY;}

wire (char v, int d) {value = v; delay = d;}

void put (char v, int d) {value = v; delay = d;}

void get (char& v, int& d) {v = value; d = delay;}

int delayValue() {return this->delay;}

char getVal() {return this->value;}

};

// Define a 2-input NAND gate

class nand {

wire i1, i2, o1;

int gateDelay;

public:

nand () {gateDelay = 2;} // Gate Delay = 2, since this is a 2 input NAND gate.

int delayValue() {return this->gateDelay;}

void NAND (wire, wire, wire&);

};

// Define the DFF class

class dff {

wire Q;

int clkQDelay;

public:

dff() {clkQDelay = 3;} // Set Clock Delay = 3

int delayValue() {return this->clkQDelay;}

wire get\_Q() {return this->Q;}

void DFF(wire, wire, wire);

};

The *logicPrimitives.cpp* fileprovides all definitions needed to instantiate the basic circuitry components:

// Carlos Lazo

// ECE579D

// Midterm Exam - Part B

#include "logicPrimitives.h"

// Define the NAND function

void nand::NAND(wire i1, wire i2, wire& o1)

{

char av, bv, wv;

int ad, bd, wd;

i1.get(av, ad);

i2.get(bv, bd);

// Apply all logic possibilities of NAND

if ((av == '1') && (bv == '1'))

wv = '0'; // Value = 0

else if ((av == 'X') || (bv == 'X'))

wv = 'X'; // Value = X

else

wv = '1'; // Value = 1

// Setup gate delay based on largest total delay seen.

// Be sure to add in both the gate delay and wire delay

wd = this->delayValue() + WIRE\_DELAY + MAX2(ad, bd);

o1.put(wv, wd); // Place calculated value on output wire

}

// Define the DFF function

void dff::DFF(wire D, wire C, wire R)

{

char Dv, Cv, Rv, Qv;

int Dd, Cd, Rd, Qd;

D.get(Dv, Dd);

C.get(Cv, Cd);

R.get(Rv, Rd);

if (Rv == '1') {

Qv = '0'; Qd = 0;

}

else if (Cv == 'P') {

if (Cd >= Dd) Qv = Dv;

else Qv = 'X';

Qd = this->clkQDelay;

}

// Catch-all statement, leave Q as it is (e.g. clk = 'N')

else

this->Q.get(Qv, Qd);

this->Q.put(Qv, Qd);

}

The *updownCounter.h* file takes care of defining the *updownCounter* class, which will be used to instantiate the 3 bit up-down counter. We have been asked to define the circuit element as a class, using input and output file structures for testbench application:

// Carlos Lazo

// ECE579D

// Midterm Exam - Part B

#include "logicPrimitives.h"

// Define the updownCounter Class

class updownCounter {

private:

// Declare all private member variables of the updownCounter:

char qout [3];

char c\_clk, c\_rst, c\_ud;

wire clk, rst, ud;

int t\_Delay;

string in\_str; // Stores input string for printing

string pr\_dff; // Stores previous flip flop values

dff reg [3]; // DFF's that make up updownCounter

wire dTemp [3]; // Define temp DFF wires for computation

public:

// Public functions that will be executed in order:

updownCounter(); // Default Constructor

void load\_input (string); // Parse input string into member variables.

void update\_reg(); // Based on ud mode, update counter accordingly.

void set\_output(); // Set all DFF's, based on clock and reset signals.

string output\_reg();// Formatted output generated for the updownCounter.

};

Naturally, the *updownCounter.cpp* contains definitions for all functions in the respective header file. Input from the testbench is loaded via the *load\_input* command, and *update\_reg()* performs all the gate level NAND logic derived from the colored expressions from the Karnaugh maps. The DFF’s are set in the *set\_output()* function (depending on *clk* & *rst*), and the results are printed with nice formatting and all pertinent information using the *output\_reg()* function.

// Carlos Lazo

// ECE579D

// Midterm Exam - Part B

#include "updownCounter.h"

// Define the updownCounter constructor

updownCounter::updownCounter()

{

for (int i = 0; i < 3; i++)

{

dTemp[i].put('X',WIRE\_DELAY);

qout[i] = 'X';

}

c\_clk = 'X'; c\_rst = 'X'; c\_ud = 'X';

t\_Delay = 0;

}

// Define the updownCounter load\_input function

void updownCounter::load\_input(string indata)

{

in\_str = indata;

// The input stream from the text file is as follows:

// updown | rst | clk ==> Information

// 0 1 2 ==> Char Location

c\_ud = indata[0]; c\_rst = indata[1]; c\_clk = indata[2];

// Set all input wires accordingly:

ud.put(c\_ud, WIRE\_DELAY);

rst.put(c\_rst, WIRE\_DELAY);

clk.put(c\_clk, 200); // 200ns pulse defined for clk

pr\_dff = "";

// Store current flip-flop values:

for (int i = 0; i < 3; i++)

pr\_dff += qout[i];

t\_Delay = 0;

}

// GATE LEVEL Implementation of updownCounter

void updownCounter::update\_reg()

{

// Begin 1st by defining y signals and their complements:

wire y2(qout[0], WIRE\_DELAY);

wire y1(qout[1], WIRE\_DELAY);

wire y0(qout[2], WIRE\_DELAY);

nand y2\_ng, y1\_ng, y0\_ng, ud\_ng;

wire y2\_n , y1\_n , y0\_n , ud\_n ;

y2\_ng.NAND(y2,y2,y2\_n); y1\_ng.NAND(y1,y1,y1\_n);

y0\_ng.NAND(y0,y0,y0\_n); ud\_ng.NAND(ud,ud,ud\_n);

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// \* Peform computation for D2 input - use color code from writeup for NAND gates: \*

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Blue Expression

nand b2\_1, b2\_2, b2\_3, b2\_4, b2\_5, b2\_6;

wire b2\_o1, b2\_o2, b2\_o3, b2\_o4, b2\_o5, d2\_b;

b2\_1.NAND(ud\_n ,y2\_n ,b2\_o1); b2\_2.NAND(b2\_o1,b2\_o1,b2\_o2);

b2\_3.NAND( y1\_n,y0\_n ,b2\_o3); b2\_4.NAND(b2\_o3,b2\_o3,b2\_o4);

b2\_5.NAND(b2\_o2,b2\_o4,b2\_o5); b2\_6.NAND(b2\_o5,b2\_o5, d2\_b);

// Green Expression

nand g2\_1, g2\_2, g2\_3, g2\_4;

wire g2\_o1, g2\_o2, g2\_o3, d2\_g;

g2\_1.NAND(y2 ,y1,g2\_o1); g2\_2.NAND(g2\_o1,g2\_o1,g2\_o2);

g2\_1.NAND(g2\_o2,y0\_n,g2\_o3); g2\_4.NAND(g2\_o3,g2\_o3, d2\_g);

// Orange Expression

nand o2\_1, o2\_2, o2\_3, o2\_4;

wire o2\_o1, o2\_o2, o2\_o3, d2\_o;

o2\_1.NAND( ud, y2,o2\_o1); o2\_2.NAND(o2\_o1,o2\_o1,o2\_o2);

o2\_3.NAND(o2\_o2,y1\_n,o2\_o3); o2\_4.NAND(o2\_o3,o2\_o3, d2\_o);

// Red Expression

nand r2\_1, r2\_2, r2\_3, r2\_4;

wire r2\_o1, r2\_o2, r2\_o3, d2\_r;

r2\_1.NAND(ud\_n ,y2,r2\_o1); r2\_2.NAND(r2\_o1,r2\_o1,r2\_o2);

r2\_3.NAND(r2\_o2,y0,r2\_o3); r2\_4.NAND(r2\_o3,r2\_o3,d2\_r);

// Purple Expression

nand p2\_1, p2\_2, p2\_3, p2\_4, p2\_5, p2\_6;

wire p2\_o1, p2\_o2, p2\_o3, p2\_o4, p2\_o5, d2\_p;

p2\_1.NAND( ud, y2\_n,p2\_o1); p2\_2.NAND(p2\_o1,p2\_o1,p2\_o2);

p2\_3.NAND( y1, y0,p2\_o3); p2\_4.NAND(p2\_o3,p2\_o3,p2\_o4);

p2\_5.NAND(p2\_o2,p2\_o4,p2\_o5); p2\_6.NAND(p2\_o5,p2\_o5, d2\_p);

// Combine Blue + Green Expressions

nand bg2\_1, bg2\_2, bg2\_3;

wire bg2\_o1, bg2\_o2, d2\_bg;

bg2\_1.NAND( d2\_b, d2\_b,bg2\_o1); bg2\_2.NAND(d2\_g,d2\_g,bg2\_o2);

bg2\_3.NAND(bg2\_o1,bg2\_o2, d2\_bg);

// Combine Orange + Red Expressions

nand or2\_1, or2\_2, or2\_3;

wire or2\_o1, or2\_o2, d2\_or;

or2\_1.NAND( d2\_o, d2\_o,or2\_o1); or2\_2.NAND(d2\_r,d2\_r,or2\_o2);

or2\_3.NAND(or2\_o1,or2\_o2, d2\_or);

// Combine (Blue + Green) + (Orange + Red) Expressions

nand bgor2\_1, bgor2\_2, bgor2\_3;

wire bgor2\_o1, bgor2\_o2, d2\_bgor;

bgor2\_1.NAND(d2\_bg ,d2\_bg ,bgor2\_o1); bgor2\_2.NAND(d2\_or,d2\_or,bgor2\_o2);

bgor2\_3.NAND(bgor2\_o1,bgor2\_o2,d2\_bgor );

// Combine (Blue + Green + Orange + Red) with Purple Expression = D2

nand bgorp2\_1, bgorp2\_2, bgorp2\_3;

wire bgorp2\_o1, bgorp2\_o2;

bgorp2\_1.NAND(d2\_bgor,d2\_bgor,bgorp2\_o1); bgorp2\_2.NAND(d2\_p,d2\_p,bgorp2\_o2);

bgorp2\_3.NAND(bgorp2\_o1,bgorp2\_o2,dTemp[0]);

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// \* Peform computation for D1 input - use color code from writeup for NAND gates: \*

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Blue Expression

nand b1\_1, b1\_2, b1\_3, b1\_4;

wire b1\_o1, b1\_o2, b1\_o3, d1\_b;

b1\_1.NAND( y1, y0,b1\_o1); b1\_2.NAND(b1\_o1,b1\_o1,b1\_o2);

b1\_3.NAND(b1\_o2,ud\_n,b1\_o3); b1\_4.NAND(b1\_o3,b1\_o3, d1\_b);

// Green Expression

nand g1\_1, g1\_2, g1\_3, g1\_4;

wire g1\_o1, g1\_o2, g1\_o3, d1\_g;

g1\_1.NAND( ud, y1,g1\_o1); g1\_2.NAND(g1\_o1,g1\_o1,g1\_o2);

g1\_3.NAND(g1\_o2,y0\_n,g1\_o3); g1\_4.NAND(g1\_o3,g1\_o3, d1\_g);

// Orange Expression

nand o1\_1, o1\_2, o1\_3, o1\_4;

wire o1\_o1, o1\_o2, o1\_o3, d1\_o;

o1\_1.NAND( ud\_n,y1\_n,o1\_o1); o1\_2.NAND(o1\_o1,o1\_o1,o1\_o2);

o1\_3.NAND(o1\_o2,y0\_n,o1\_o3); o1\_4.NAND(o1\_o3,o1\_o3, d1\_o);

// Red Expression

nand r1\_1, r1\_2, r1\_3, r1\_4;

wire r1\_o1, r1\_o2, r1\_o3, d1\_r;

r1\_1.NAND( ud,y1\_n,r1\_o1); r1\_2.NAND(r1\_o1,r1\_o1,r1\_o2);

r1\_3.NAND(r1\_o2, y0,r1\_o3); r1\_4.NAND(r1\_o3,r1\_o3, d1\_r);

// Combine Blue + Green Expressions

nand bg1\_1, bg1\_2, bg1\_3;

wire bg1\_o1, bg1\_o2, d1\_bg;

bg1\_1.NAND( d1\_b, d1\_b,bg1\_o1); bg1\_2.NAND(d1\_g,d1\_g,bg1\_o2);

bg1\_3.NAND(bg1\_o1,bg1\_o2, d1\_bg);

// Combine Orange + Red Expressions

nand or1\_1, or1\_2, or1\_3;

wire or1\_o1, or1\_o2, d1\_or;

or1\_1.NAND( d1\_o, d1\_o,or1\_o1); or1\_2.NAND(d1\_r,d1\_r,or1\_o2);

or1\_3.NAND(or1\_o1,or1\_o2, d1\_or);

// Combine (Blue + Green) + (Orange + Red) = D1

nand bgor1\_1, bgor1\_2, bgor1\_3;

wire bgor1\_o1, bgor1\_o2;

bgor1\_1.NAND( d1\_bg, d1\_bg,bgor1\_o1); bgor1\_2.NAND(d1\_or,d1\_or,bgor1\_o2);

bgor1\_3.NAND(bgor1\_o1,bgor1\_o2,dTemp[1]);

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// \* Peform computation for D0 input - use color code from writeup for NAND gates: \*

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Blue Expression

nand b0\_1, b0\_2;

wire b0\_o1, d0\_b;

b0\_1.NAND(y1\_n,y0\_n,b0\_o1); b0\_2.NAND(b0\_o1,b0\_o1,d0\_b);

// Green Expression

nand g0\_1, g0\_2;

wire g0\_o1, d0\_g;

g0\_1.NAND(y1,y0\_n,g0\_o1); g0\_2.NAND(g0\_o1,g0\_o1,d0\_g);

// Combine Blue + Green Expressions = D0

nand bg0\_1, bg0\_2, bg0\_3;

wire bg0\_o1, bg0\_o2;

bg0\_1.NAND( d0\_b, d0\_b,bg0\_o1); bg0\_2.NAND(d0\_g,d0\_g,bg0\_o2);

bg0\_3.NAND(bg0\_o1,bg0\_o2, dTemp[2]);

// !!!!!!!!!!!!!!!!!!!!!!!!!!!

// !!! END NAND GATE LOGIC !!!

// !!!!!!!!!!!!!!!!!!!!!!!!!!!

// Now that all flip flop logic is performed, compute maximum time delay:

t\_Delay = MAX3(dTemp[0].delayValue(),dTemp[1].delayValue(),dTemp[2].delayValue());

}

void updownCounter::set\_output()

{

// Set all DFF - this is dependant on clock and reset values.

// Utilize values determined in the update\_reg() function.

for (int i = 0; i < 3; i++)

{

reg[i].DFF(dTemp[i],clk,rst);

// Also, set qout values equal to values stored in D.

qout[i] = reg[i].get\_Q().getVal();

}

}

string updownCounter::output\_reg()

{

string pretty\_print = "";

pretty\_print += "The current state of the UpDown Counter is:\t";

for (int i = 0; i < 3; i++)

pretty\_print += pr\_dff[i];

pretty\_print += "\n";

pretty\_print += "Given >>> clk = ";

pretty\_print += c\_clk;

pretty\_print += ", rst = ";

pretty\_print += c\_rst;

pretty\_print += ", UpDown =";

pretty\_print += c\_ud;

pretty\_print += "\n";

pretty\_print += "The next state of the UpDown Counter will be:\t";

for (int i = 0; i < 3; i++)

pretty\_print += qout[i];

char bufD[32];

pretty\_print += "\nTotal time delay to achieve this result was: ";

pretty\_print += \_itoa(t\_Delay,bufD,10);

pretty\_print += "\n\n";

return pretty\_print;

}

With all code now developed for the functionality, we build a test bench that will apply data to the *updownCounter* from external file – *updownTester.cpp* contains the code:

// Carlos Lazo

// ECE579D

// Midterm Exam - Part B

#include "updownCounter.h"

// Create main testbench for the updownCounter:

int main ()

{

string inVec;

updownCounter UUT;

ifstream finp ("indata.tst");

ofstream fout ("outdata.tst");

char stop\_in ('0');

finp >> inVec; // Read in first line of testbench data

while (stop\_in != '.')

{

UUT.load\_input(inVec); // Load inputs into ShiftRegister

UUT.update\_reg(); // Compute operation based on mode

UUT.set\_output(); // Set DFFs based on computed values

fout << UUT.output\_reg(); // Output data to external file

// Gather next data input:

finp >> inVec;

stop\_in = inVec[0];

}

fout << "END FILESTREAM";

return 0;

}

Information read in from *indata.tst* has the following format: <UpDown> <rst> <clk>

The file stream will terminate once a ‘.’ is seen in the input file. The following is *indata.tst*:

**indata.tst**

00P

01P

10P

10P

10P

10P

10P

10P

00P

00P

00P

10P

10P

01P

00N

00P

00P

00P

00P

00N

00P

...

The following data, based on the formatting seen in the *output\_reg()* function, is what is placed into *outdata.tst* upon compiling and running the program. Please note that I did not explicitly delineate at each step what the Up/Down Counter is supposed to be doing. In looking at each line of output, one sees the current value of the counter, the data applied to it, and then the values after the DFF values have been clocked in. Different items that have been shown here are synchronous resetting, a few clk values of N, which imply that the DFF’s should not change in value, and up/down counting across the series and rolling over from 111 🡪 000 and vice versa:

**outdata.tst**

The current state of the UpDown Counter is: XXX

Given >>> clk = P, rst = 0, UpDown =0

The next state of the UpDown Counter will be: XXX

Total time delay to achieve this result was: 34

The current state of the UpDown Counter is: XXX

Given >>> clk = P, rst = 1, UpDown =0

The next state of the UpDown Counter will be: 000

Total time delay to achieve this result was: 34

The current state of the UpDown Counter is: 000

Given >>> clk = P, rst = 0, UpDown =1

The next state of the UpDown Counter will be: 001

Total time delay to achieve this result was: 34

The current state of the UpDown Counter is: 001

Given >>> clk = P, rst = 0, UpDown =1

The next state of the UpDown Counter will be: 010

Total time delay to achieve this result was: 34

The current state of the UpDown Counter is: 010

Given >>> clk = P, rst = 0, UpDown =1

The next state of the UpDown Counter will be: 011

Total time delay to achieve this result was: 34

The current state of the UpDown Counter is: 011

Given >>> clk = P, rst = 0, UpDown =1

The next state of the UpDown Counter will be: 100

Total time delay to achieve this result was: 34

The current state of the UpDown Counter is: 100

Given >>> clk = P, rst = 0, UpDown =1

The next state of the UpDown Counter will be: 101

Total time delay to achieve this result was: 34

The current state of the UpDown Counter is: 101

Given >>> clk = P, rst = 0, UpDown =1

The next state of the UpDown Counter will be: 110

Total time delay to achieve this result was: 34

The current state of the UpDown Counter is: 110

Given >>> clk = P, rst = 0, UpDown =0

The next state of the UpDown Counter will be: 101

Total time delay to achieve this result was: 34

The current state of the UpDown Counter is: 101

Given >>> clk = P, rst = 0, UpDown =0

The next state of the UpDown Counter will be: 100

Total time delay to achieve this result was: 34

The current state of the UpDown Counter is: 100

Given >>> clk = P, rst = 0, UpDown =0

The next state of the UpDown Counter will be: 011

Total time delay to achieve this result was: 34

The current state of the UpDown Counter is: 011

Given >>> clk = P, rst = 0, UpDown =1

The next state of the UpDown Counter will be: 100

Total time delay to achieve this result was: 34

The current state of the UpDown Counter is: 100

Given >>> clk = P, rst = 0, UpDown =1

The next state of the UpDown Counter will be: 101

Total time delay to achieve this result was: 34

The current state of the UpDown Counter is: 101

Given >>> clk = P, rst = 1, UpDown =0

The next state of the UpDown Counter will be: 000

Total time delay to achieve this result was: 34

The current state of the UpDown Counter is: 000

Given >>> clk = N, rst = 0, UpDown =0

The next state of the UpDown Counter will be: 000

Total time delay to achieve this result was: 34

The current state of the UpDown Counter is: 000

Given >>> clk = P, rst = 0, UpDown =0

The next state of the UpDown Counter will be: 111

Total time delay to achieve this result was: 34

The current state of the UpDown Counter is: 111

Given >>> clk = P, rst = 0, UpDown =0

The next state of the UpDown Counter will be: 110

Total time delay to achieve this result was: 34

The current state of the UpDown Counter is: 110

Given >>> clk = P, rst = 0, UpDown =0

The next state of the UpDown Counter will be: 101

Total time delay to achieve this result was: 34

The current state of the UpDown Counter is: 101

Given >>> clk = P, rst = 0, UpDown =0

The next state of the UpDown Counter will be: 100

Total time delay to achieve this result was: 34

The current state of the UpDown Counter is: 100

Given >>> clk = N, rst = 0, UpDown =0

The next state of the UpDown Counter will be: 100

Total time delay to achieve this result was: 34

The current state of the UpDown Counter is: 100

Given >>> clk = P, rst = 0, UpDown =0

The next state of the UpDown Counter will be: 011

Total time delay to achieve this result was: 34

END FILESTREAM

This concludes the analysis for Midterm Exam – Part B.

**8-Bit Serial Adder with SystemC** (Part C)

The goal of this final part of the midterm is to develop an 8-bit Serial Adder using SystemC. The serial adder waits for a pulse on the *start* signal, at which point, it begins receiving serial data for 8 *clk* pulses on *ain* and *bin* serial inputs. At this point, the *ready* output of the circuit is deasserted, and bit by bit calculations are made ready on the *result* output. The datapath can be shown below, taken directly from the Midterm Examination:



As will be seen in the code, the Huffman Model was used, splitting the diagram into a sequential portion and a combinational portion. The first two code files for this problem showcase such a paradigm:

The file *serialAdder.h* defines the *serialAdder* class and all of its internal and external signals. It is noted that BOTH an SC\_METHOD and SC\_THREAD were used – one for the combinational portion, and one of the sequential portion, respectively:

// Carlos Lazo

// ECE579D

// Midterm Exam - Part C

#include <iostream>

#include <string>

#include "systemc.h"

using namespace std;

SC\_MODULE(serialAdder)

{

// Define all fixed module variables:

sc\_in <sc\_logic> ain; // Serial input for a

sc\_in <sc\_logic> bin; // Serial input for b

sc\_in <sc\_logic> start; // Start pulses received here

sc\_in <sc\_logic> clk; // System clock

sc\_in <sc\_logic> rst; // System reset

sc\_out<sc\_logic> ready; // Indicates that system is ready to add

sc\_out<sc\_lv<8> > result; // Outputs result of serial addition

// Declare internal variables to serialAdder:

sc\_uint<8> counter; // Counter for FSM

sc\_logic sum; // Sum for current computation

sc\_logic c\_in; // Carry in for current computation

sc\_logic c\_out; // Carry out for next computation

enum states {idle,idle\_t,add,add\_t,output}; // States for FSM

sc\_signal <states> p\_state, n\_state; // Signals for FSM

// Define functions for serialAdder

void comb\_func();

void seq\_func();

// Define constructor for serialAdder:

SC\_CTOR (serialAdder) : counter(0)

{

SC\_METHOD(comb\_func);

sensitive << p\_state << ain << bin;

SC\_THREAD(seq\_func);

sensitive << clk.pos();

}

};

The file *serialAdder.cpp* defines all the functionality of the *serialAdder* class. Note that there are two toggle states defined for the *idle* and the *add* states of the Finite State Machine (FSM). Since the sequential portion is looking for a change in *p\_state* to active or “awaken” that method, it was necessary to do so. The code for this portion can be seen below:

// Carlos Lazo

// ECE579D

// Midterm Exam - Part C

#include "serialAdder.h"

#include "systemc.h"

using namespace std;

using namespace sc\_dt;

#define HI SC\_LOGIC\_1

#define LO SC\_LOGIC\_0

// Combinational logic:

// Define the combinational logic for the serialAdder as a FSM

void serialAdder::comb\_func()

{

// By default, set the next state to idle:

n\_state = idle;

// Delineate transitional logic of serial adder:

switch (p\_state)

{

case (idle):

{

if (start.read() == SC\_LOGIC\_1)

n\_state = add;

else

n\_state = idle\_t;

break;

}

case (idle\_t):

{

if (start.read() == SC\_LOGIC\_1)

n\_state = add;

else

n\_state = idle;

break;

}

case (add):

{

if (counter < 7)

n\_state = add\_t;

else

n\_state = output;

break;

}

case (add\_t):

{

if (counter < 7)

n\_state = add;

else

n\_state = output;

break;

}

case (output):

{

n\_state = idle;

break;

}

default : n\_state = idle;

}

// Perform logic operations based on p\_state:

if (p\_state == idle || p\_state == idle\_t)

{

cout << "I'm in the idle state." << endl;

ready.write(SC\_LOGIC\_1);

counter = 0;

}

else if (p\_state == add || p\_state == add\_t)

{

cout << "I'm in the add state, and counter = " << counter << "." << endl;

if (counter == 0)

{

c\_in = SC\_LOGIC\_0;

sum = SC\_LOGIC\_0;

}

// If adding, perform logic on values and store to result.

sc\_lv<8> temp = result.read();

cout << "Current result is: ";

for (int i = 0; i < 8; i++)

cout << temp[i];

cout << endl;

//cout << "DEBUG 2" << endl;

sum = (ain.read() ^ bin.read()) ^ c\_in;

c\_out = (ain.read() & bin.read()) | (c\_in & (ain.read() ^ bin.read()));

// Perform a right shift since LSB's are coming in first,

// and set the current cout to the next cin.

temp[7 - counter] = sum;

c\_in = c\_out;

cout << "The new result is: ";

for (int i = 0; i < 8; i++)

cout << temp[i];

cout << endl;

// Write current sum to result register:

result.write(temp);

counter = counter + 1;

}

else if (p\_state == output)

{

counter = 0;

ready.write(SC\_LOGIC\_1);

}

}

// Declare the sequential part of the serialAdder,

// which is controlled by the posedge clk transition.

void serialAdder::seq\_func()

{

while(1){

if (rst.read() == SC\_LOGIC\_1)

{

cout << "I was RESET!" << endl;

p\_state = idle;

}

else

p\_state = n\_state;

wait();

}

}

// Quick Testbench to prove functionality:

int sc\_main (int argc, char\* argv[])

{

sc\_report\_handler::set\_actions("/IEEE\_Std\_1666/deprecated", SC\_DO\_NOTHING);

sc\_signal <sc\_logic> clk;

sc\_signal <sc\_logic> rst;

sc\_signal <sc\_logic> ain;

sc\_signal <sc\_logic> bin;

sc\_signal <sc\_logic> str;

clk = SC\_LOGIC\_0;

rst = SC\_LOGIC\_0;

ain = SC\_LOGIC\_0;

bin = SC\_LOGIC\_0;

str = SC\_LOGIC\_0;

sc\_signal <sc\_logic> ready;

sc\_signal <sc\_lv<8> > res;

serialAdder UUT("UUT");

UUT.rst(rst);

UUT.ain(ain);

UUT.bin(bin);

UUT.clk(clk);

UUT.start(str);

UUT.ready(ready);

UUT.result(res);

sc\_trace\_file \*out\_file;

out\_file = sc\_create\_vcd\_trace\_file ("outfile");

sc\_trace(out\_file,ain,"ain");

sc\_trace(out\_file,bin,"bin");

sc\_trace(out\_file,rst,"rst");

sc\_trace(out\_file,rst,"clk");

sc\_trace(out\_file,str,"start");

sc\_trace(out\_file,ready,"ready");

sc\_trace(out\_file,res,"result");

sc\_start(5);

clk = SC\_LOGIC\_1;

rst = SC\_LOGIC\_1;

sc\_start(5);

clk = SC\_LOGIC\_0;

rst = SC\_LOGIC\_0;

sc\_start(5);

ain = SC\_LOGIC\_1;

clk = SC\_LOGIC\_1;

sc\_start(5);

clk = SC\_LOGIC\_0;

sc\_start(5);

str = SC\_LOGIC\_1;

clk = SC\_LOGIC\_1;

sc\_start(5);

str = SC\_LOGIC\_0;

clk = SC\_LOGIC\_0;

sc\_start(5);

clk = SC\_LOGIC\_1;

sc\_start(5);

clk = SC\_LOGIC\_0;

for (int i = 0; i < 10; i++)

{

sc\_start(5);

clk = SC\_LOGIC\_1;

sc\_start(5);

clk = SC\_LOGIC\_0;

}

sc\_close\_vcd\_trace\_file (out\_file);

cout << endl << endl;

return 0;

}

Note that the very bottom of this file contains the Testbench for the serial adder. This was primarily done in order to showcase full functionality of the Serial Adder in full operation across the 8 block pulses. Output is displayed here both from the command line, and is then verified by a VCD output (text only). A few simple scenarios are described below – the code above describes these output statements. The VCD files are all saved in the project directory. Since the testbench was modified and some of the timings were erased, it is acceptable to look at the VCD to see when variables were changed and which ones were modified in real time.

Please note that the output from CMD OUTPUT and will be **backwards** in comparison to each other. This is due to the nature of the way the code updates the output register, and regarding the way the VCD file updates byte changes in signals.

**Scenario #1**: *ain* = 1, *bin* = 0 🡪 outfile1.vcd

**CMD OUTPUT**

SystemC 2.2.0 --- Oct 18 2010 19:53:18

Copyright (c) 1996-2006 by all Contributors

ALL RIGHTS RESERVED

I'm in the idle state.

WARNING: Default time step is used for VCD tracing.

I was RESET!

I'm in the idle state.

I'm in the idle state.

I'm in the idle state.

I'm in the add state, and counter = 0.

Current result is: XXXXXXXX

The new result is: XXXXXXX1

I'm in the add state, and counter = 1.

Current result is: XXXXXXX1

The new result is: XXXXXX11

I'm in the add state, and counter = 2.

Current result is: XXXXXX11

The new result is: XXXXX111

I'm in the add state, and counter = 3.

Current result is: XXXXX111

The new result is: XXXX1111

I'm in the add state, and counter = 4.

Current result is: XXXX1111

The new result is: XXX11111

I'm in the add state, and counter = 5.

Current result is: XXX11111

The new result is: XX111111

I'm in the add state, and counter = 6.

Current result is: XX111111

The new result is: X1111111

I'm in the add state, and counter = 7.

Current result is: X1111111

The new result is: 11111111

I'm in the idle state.

I'm in the idle state.

**VCD OUTPUT**

$date

Nov 07, 2010 22:54:54

$end

$version

SystemC 2.2.0 --- Oct 18 2010 19:53:18

$end

$timescale

1 ps

$end

$scope module SystemC $end

$var wire 1 aaa ain $end

$var wire 1 aab bin $end

$var wire 1 aac rst $end

$var wire 1 aad clk $end

$var wire 1 aae start $end

$var wire 1 aaf ready $end

$var wire 8 aag result [7:0] $end

$upscope $end

$enddefinitions $end

$comment

All initial values are dumped below at time 0 sec = 0 timescale units.

$end

$dumpvars

0aaa

0aab

0aac

0aad

0aae

1aaf

bXXXXXXXX aag

$end

#5000

1aac

1aad

#10000

0aac

0aad

#15000

1aaa

#25000

1aae

#30000

0aae

#35000

b1XXXXXXX aag

#45000

b11XXXXXX aag

#55000

b111XXXXX aag

#65000

b1111XXXX aag

#75000

b11111XXX aag

#85000

b111111XX aag

#95000

b1111111X aag

#105000

b11111111 aag

It is noted that the results for both cases **MATCH** the expected results.

**Scenario #2**: *ain* = 1 (initially), *bin* = 1 (later), then *ain* = 0 (later) 🡪 outfile2.vcd

**CMD OUTPUT**

SystemC 2.2.0 --- Oct 18 2010 19:53:18

Copyright (c) 1996-2006 by all Contributors

ALL RIGHTS RESERVED

I'm in the idle state.

WARNING: Default time step is used for VCD tracing.

I was RESET!

I'm in the idle state.

I'm in the idle state.

I'm in the idle state.

I'm in the add state, and counter = 0.

Current result is: XXXXXXXX

The new result is: XXXXXXX1

I'm in the add state, and counter = 1.

Current result is: XXXXXXX1

The new result is: XXXXXX01

I'm in the add state, and counter = 2.

Current result is: XXXXXX01

The new result is: XXXXX101

I'm in the add state, and counter = 3.

Current result is: XXXXX101

The new result is: XXXX1101

I'm in the add state, and counter = 4.

Current result is: XXXX1101

The new result is: XXX11101

I'm in the add state, and counter = 5.

Current result is: XXX11101

The new result is: XX111101

I'm in the add state, and counter = 6.

Current result is: XX111101

The new result is: X0111101

I'm in the add state, and counter = 7.

Current result is: X0111101

The new result is: 00111101

I'm in the idle state.

I'm in the idle state.

I'm in the idle state.

I'm in the idle state.

I'm in the idle state.

I'm in the idle state.

I'm in the idle state.

I'm in the idle state.

I'm in the idle state.

Press any key to continue . . .

**VCD OUTPUT**

$date

Nov 07, 2010 23:04:15

$end

$version

SystemC 2.2.0 --- Oct 18 2010 19:53:18

$end

$timescale

1 ps

$end

$scope module SystemC $end

$var wire 1 aaa ain $end

$var wire 1 aab bin $end

$var wire 1 aac rst $end

$var wire 1 aad clk $end

$var wire 1 aae start $end

$var wire 1 aaf ready $end

$var wire 8 aag result [7:0] $end

$upscope $end

$enddefinitions $end

$comment

All initial values are dumped below at time 0 sec = 0 timescale units.

$end

$dumpvars

0aaa

0aab

0aac

0aad

0aae

1aaf

bXXXXXXXX aag

$end

#5000

1aac

1aad

#10000

0aac

0aad

#15000

1aaa

#25000

1aae

#30000

0aae

#35000

b1XXXXXXX aag

#40000

1aab

b10XXXXXX aag

#45000

b101XXXXX aag

#55000

b1011XXXX aag

#65000

b10111XXX aag

#70000

0aaa

0aab

b101111XX aag

#75000

b1011110X aag

#85000

b10111100 aag

It is noted that the results for both cases **MATCH** the expected results.

Integrating the above *serialAdder* using external data files for *ain* and *bin* can be done with the following files below, which instantiate a testbench as a class and apply data from external files:

Here is *tb\_serialAdder.h*, which defines the overall testbench and port mappings to the *serialAdder*:

// Carlos Lazo

// ECE579D

// Midterm Exam - Part C

#include "systemc.h"

#include "serialAdder.h"

SC\_MODULE (tb\_serialAdder) {

sc\_out <sc\_logic> ain; // Serial input for a

sc\_out <sc\_logic> bin; // Serial input for b

sc\_out <sc\_logic> start; // Start pulses received here

sc\_out <sc\_logic> clk; // System clock

sc\_out <sc\_logic> rst; // System reset

serialAdder UUT; // Instantiated serial adder

// Declare all functions which will read in data:

void data\_ain();

void data\_bin();

void data\_start();

void data\_clk();

void data\_rst();

// Declare tb\_serialAdder Constructor

SC\_CTOR(tb\_serialAdder) : UUT("UUT")

{

SC\_THREAD(data\_ain);

SC\_THREAD(data\_bin);

SC\_THREAD(data\_start);

SC\_THREAD(data\_clk);

SC\_THREAD(data\_rst);

sensitive << clk << ain << bin;

UUT.ain(ain);

UUT.bin(bin);

UUT.start(start);

UUT.rst(rst);

UUT.clk(clk);

}

};

Here is *tb\_serialAdder.cpp*, which defines input paradigms for the testbench. Note that the *clk*, *rst*, and *start* signals are force set here at random intervals:

// Carlos Lazo

// ECE579D

// Midterm Exam - Part C

#include "systemc.h"

#include "tb\_serialAdder.h"

using namespace sc\_dt;

// Generate data for ain value:

void tb\_serialAdder::data\_ain()

{

int data;

FILE \*ifp;

ifp = fopen("ainfile.dat","r");

for (int i = 0; i < 100; i = i + 5)

{

if (feof(ifp)) ain = SC\_LOGIC\_0;

else ain = SC\_LOGIC\_1;

wait (5,SC\_NS);

}

}

// Generate data for bin value from file:

void tb\_serialAdder::data\_bin()

{

int data;

FILE \*ifp;

ifp = fopen("binfile.dat","r");

for (int i = 0; i < 100; i = i + 10)

{

if (feof(ifp)) bin = SC\_LOGIC\_0;

else bin = SC\_LOGIC\_1;

wait (10,SC\_NS);

}

}

// Set timing data for start here in the testbench:

void tb\_serialAdder::data\_start()

{

wait (20,SC\_NS);

start = SC\_LOGIC\_1;

wait (10,SC\_NS);

start = SC\_LOGIC\_0;

}

// Set timing data for reset here in the testbench:

void tb\_serialAdder::data\_rst()

{

wait (5,SC\_NS);

rst = SC\_LOGIC\_1;

wait (10,SC\_NS);

rst = SC\_LOGIC\_0;

}

// Set timing data for clk here in the testbench:

void tb\_serialAdder::data\_clk()

{

for (int i = 0; i < 50; i++)

{

clk = SC\_LOGIC\_1;

wait (5,SC\_NS);

clk = SC\_LOGIC\_0;

wait (5,SC\_NS);

}

}

This concludes the analysis for Midterm Exam – Part C, and for the entire Midterm Exam.