The goal of the assignment is to develop a Mealy Machine Sequence Detector which looks for the sequence **11010**, taking an input sequence ***x*** and having ***w*** as the output. The logic from this sequence detector will then be modeled in C++, using external Input/Output (IO) file constructs and gates.

**State Machine Logic Design** (Part A)

Start by designing the State Diagram for the 11010 Detector:

**STATE DIAGRAM – 11010 SEQUENCE DETECTOR**

**0/0**

**1/0**

**1/0**

**0/0**

**0/1**

**0/0**

Utilizing this diagram as a basis, develop the Stable Table of the 11010 Detector:

**STATE TABLE OF THE 1101 SEQUENCE DETECTOR**



Now create the state assignment matrix:

**STATE ASSIGNMENT TABLE**



With the state assignments, create the Transition Table for the 11010 Detector:

**TRANSITION TABLE FOR THE 11010 DETECTOR**



With the Flip-Flop Excitation Table, develop the Karnaugh Maps for each of the flip-flops and the output. Then, use the Karnaugh Maps to determine logic needed to model the flip-flops and outputs in terms of the state variables:

**FLIP-FLOP KARNAUGH MAPS**



**D2 = x y1 y0**



**D1 = x y1 y0 + y1 y0 + x y2**



**D0 = x y2 y1 y0 + x y1 y0**



**w = (Y2Y1Y0 == 100) && x y2**

With the following equations setup for the Mealy Machine Sequence Detector of **11010**, it is appropriate to begin the modeling in C++.

**C++ Programming and Testbench Design** (Part A & B)

The first file that was programmed was *characterPrimitives.h*, which lists the function templates for the character functions utilized in this homework. Note that only the scalar forms of the functions have been used, and the D Flip-Flop template was added from the in-class lectures:

// Carlos Lazo

// ECE 579D

// Homework 02

char and (char a, char b);

char or (char a, char b);

char not (char a);

char tri (char a, char c);

char resolve (char a, char c);

void and (char a[], char b[], char w[]);

void or (char a[], char b[], char w[]);

void tri (char a[], char c, char w[]);

void resolve (char a[], char b[], char w[]);

char xor (char a, char b);

void fullAdder (char a, char b, char ci, char & co, char & sum);

// File modified to include the definition for a D Flip-Flop

void dff\_PAH (char D, char clk, char reset, char&Q);

The second file listed is *characterPrimitives.cpp*, which instantiates all functionality of the simulated logic gates from the header file listed above. Again, functionality for the D Flip-Flop was added in:

// Carlos Lazo

// ECE 579D

// Homework 02

char and (char a, char b)

{

if ((a=='0')||(b=='0')) return '0';

else if ((a=='1')&&(b=='1')) return '1';

else return 'X';

}

char or (char a, char b)

{

if ((a=='1')||(b=='1')) return '1';

else if ((a=='0')&&(b=='0')) return '0';

else return 'X';

}

char not (char a)

{

if (a=='1') return '0';

else if (a=='0') return '1';

else return 'X';

}

char tri (char a, char c)

{

if (c=='1') return a;

else return 'Z';

}

char resolve (char a, char b)

{

if (a=='Z' || a==b) return b;

else if (b=='Z') return a;

else return 'X';

}

void and (char a[], char b[], char w[])

{

int i=0;

while (a[i] != '\0') {

w[i] = and (a[i], b[i]);

i++;

};

w[i] = '\0';

}

void or (char a[], char b[], char w[])

{

int i=0;

while (a[i] != '\0') {

w[i] = or (a[i], b[i]);

i++;

};

w[i] = '\0';

}

void tri (char a[], char c, char w[])

{

int i=0;

while (a[i] != '\0') {

w[i] = tri (a[i], c);

i++;

};

w[i] = '\0';

}

void resolve (char a[], char b[], char w[])

{

int i=0;

while (a[i] != '\0') {

w[i] = resolve (a[i], b[i]);

i++;

};

w[i] = '\0';

}

char xor (char a, char b)

{

if ((a=='X')||(b=='X')||(a=='Z')||(b=='Z')) return 'X';

else if (a==b) return '0';

else return '1';

}

void fullAdder (char a, char b, char ci, char & co, char & sum)

{

char axb, ab, abc;

axb = xor (a, b);

ab = and (a, b);

abc = and (axb, ci);

co = or (ab, abc);

sum = xor (axb, ci);

}

// File modified to include the definition for a D Flip-Flop.

// It clocks on posedge, is asynchronous, and is active-low.

void dff\_PAH (char D, char clk, char reset, char&Q)

{

if (reset == '1')

Q = '0';

else if (clk == 'P')

Q = D;

}

The third file defined is *MealyDetector.h*, which simply lists all necessary C++ libraries and templates needed in order to make the Visual Studio project compile correctly:

// Carlos Lazo

// ECE 579D

// Homework 02

// Declare all necessary libraries:

#include <fstream>

#include <iostream>

#include <string>

using namespace std;

The fourth and final file represents the crux of the assignment – *MealyDetector.cpp*. This file contains all necessary logic based on the logic analysis performed at the beginning of this write-up. This file also instantiates a logic testbench, pulling in data from an external file called *indata.tst*, and outputting all data to a file called *outdata.tst*.

// Carlos Lazo

// ECE 579D

// Homework 02

#include "characterPrimitives.h"

#include "MealyDetector.h"

// Create main testbench for the 11010 Mealy Machine:

int main ()

{

string inVec;

string outVec = ",,,,,";

char xin('0'), reset, clock;

// Variables and intermediate logic chars declared in order to use

// all necessary primitive logic gates.

char Y2('X'), Y1('X'), Y0('X'), D2, D1, D11, D12, D13,

D0, D01, D02, D03, w, w1, w2;

ifstream finp ("indata.tst");

ofstream fout ("outdata.tst");

while (xin != '.')

{

finp >> inVec;

xin = inVec[0];

reset = inVec[1];

clock = inVec[2];

// Output current state and input being put into the system

fout << "Current State: " << Y2 << Y1 << Y0 <<

", Input: " << inVec << "\n";

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

/\* DEFINE D2 FlipFlop \*/

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

// D2 = x & Y1 & Y0

D2 = and ( and (xin, Y1), Y0);

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

/\* DEFINE D1 FlipFlop \*/

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

// D11 = x & ~Y1 & Y0

D11 = and ( and (xin, not(Y1)), Y0);

// D12 = Y1 & ~Y0

D12 = and (Y1, not(Y0));

// D13 = x & Y2

D13 = and (xin, Y2);

// D1 = D11 + D12 + D13

D1 = or ( or(D11,D12), D13);

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

/\* DEFINE D0 FlipFlop \*/

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

// D01 = x & ~Y2

D01 = and (xin, not(Y2));

// D02 = ~Y1 & ~Y0

D02 = and ( not(Y1), not(Y0));

// D03 = ~x & Y1 & ~Y0

D03 = and ( and (not(xin), Y1), not(Y0));

// D0 = (D01 & D02) + D03

D0 = or ( and(D01,D02), D03);

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

/\* DEFINE Output w \*/

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

// w1 = (Y2Y1Y0 == 100)

w1 = and ( Y2, and (not(Y1), not(Y0)));

// w2 = ~x & Y2

w2 = and ( not(xin), Y2);

// w = w1 & w2

w = and(w1,w2);

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

// D Flip-Flop Assignments, based on clock and reset values

dff\_PAH (D2, clock, reset, Y2);

dff\_PAH (D1, clock, reset, Y1);

dff\_PAH (D0, clock, reset, Y0);

// Apply the following values after application of previous inputs:

outVec[0] = w;

outVec[2] = Y2;

outVec[3] = Y1;

outVec[4] = Y0;

// For the Mealy Machine, output the next state and the respective output

// after the D flip-flops have been set with the current input values.

fout << "\tOutput, Next State: " << outVec + "\n\n";

}

fout << "END FILESTREAM";

return 0;

}

Each line of the *indata.tst* file contains 3 different pieces of information for each and every clock cycle of the Mealy Machine: input, reset, and clock. The D Flip-Flop sets the current D variables when the clock is set to a value of P. The following is the input stream used in the test bench – the **red** and **blue** strings represent the two sequences that will lead up to an output of 1, which will be represented in **green**. Comments are written to explain the intricacies of certain steps in deriving this testbench:

indata.tst

\* Reminder : x y z = input, reset, clock

000

000

000

100

10P

010 (Reset 🡺 all states = 0, which will *start* the machine)

**1**0P

**1**0P

**0**0P

**1**0P

**0**0P

10P

10P

10P (Machine should stay at State C here)

00P

**1**0P (Machine sees 1 instead of 0, so go back to State C)

**1**0P

**0**0P

**1**0P

**0**0P

00P

... (End transmission, output should be 1 a total of 2 times)

The following file represents the output of the testbench, which was edited from the example provided in class in order to more easily showcase the state transitions. Each set of two lines corresponds to one pass through the while loop, and maps 1-to-1 with the *indata.tst* file listed above. All input seen above is written in that respective block in the *outdata.tst* file. The following is the output created by the test bench – the **red** and **blue** strings represent the two sequences that should have led up to an output of 1, which will be represented in **green**. These should match those expected by the input stream.

Each block lists the current state, and the current input taken from *indata.tst*. The next line, which is started by a tab, lists the output seen and the next state to which the machine will go.

outdata.tst

Current State: XXX, Input: 000

Output, Next State: X,XXX

Current State: XXX, Input: 000

Output, Next State: X,XXX

Current State: XXX, Input: 000

Output, Next State: X,XXX

Current State: XXX, Input: 100

Output, Next State: 0,XXX

Current State: XXX, Input: 10P

Output, Next State: 0,XXX

Current State: XXX, Input: 010

Output, Next State: X,000

Current State: 000, Input: **1**0P

Output, Next State: 0,001

Current State: 001, Input: **1**0P

Output, Next State: 0,010

Current State: 010, Input: **0**0P

Output, Next State: 0,011

Current State: 011, Input: **1**0P

Output, Next State: 0,100

Current State: 100, Input: **0**0P

Output, Next State: **1**,000

Current State: 000, Input: 10P

Output, Next State: 0,001

Current State: 001, Input: 10P

Output, Next State: 0,010

Current State: 010, Input: 10P

Output, Next State: 0,010

Current State: 010, Input: 00P

Output, Next State: 0,011

Current State: 011, Input: **1**0P

Output, Next State: 0,100

Current State: 100, Input: **1**0P

Output, Next State: 0,010

Current State: 010, Input: **0**0P

Output, Next State: 0,011

Current State: 011, Input: **1**0P

Output, Next State: 0,100

Current State: 100, Input: **0**0P

Output, Next State: **1**,000

Current State: 000, Input: 00P

Output, Next State: 0,000

Current State: 000, Input: ...

Output, Next State: 0,000

END FILESTREAM

As can be seen above, the Mealy Machine correctly identifies all correct next-state and output pairings based on the given input stream. The *indata.tst* file maps directly to the *outdata.tst* file, and the C++ logic provided in *MealyDetector.cpp* correctly models the 11010 Mealy Sequence Detector.

**Test Methodology** (Part C)

As far as test bench sophistication, the following are different concepts that can constitute a comprehensive test scheme that can be implemented in this C++ model:

1. Simple logic testing can be performed by adding in different nodes (*wire* nodes as seen in Verilog) in order to ensure that all D and Y variables are being set correctly. Since all of the logic was derived in the first part of this analysis, knowing and understanding that all gates are propagating values correctly would be an easy way to quick error-checking.
2. Using the methodology from #1 to ensure the testbench produces the correct output, it would be possible to create a faulty model. In other words, a new Mealy Machine could be created to force-inject faults into the logic (e.g. setting D2 = 0 all the time, forcing x = 1 half of the time regardless of the logic). The later would be particularly destructive for a Mealy Machine, since the next state depends on the current state and the input.
3. With the realization of #2, the testbench could now be used to *detect faults* given an input sequence. Comparing the correct model from #1 to the incorrect model from #2, or Fault Detection, would be relatively easy, given all I/O paradigms available in C++.

This concludes the analysis for Homework 02.