This is the write-up for ECE574, Test #1 – Verilog Basics of Combinational Logic.

***Test 1: PART 1***

The Verilog code for this section can be found in the Pt 1 folder of the ZIP file:

*MooreSeqDetector.v*

*MooreSeqDetectorTester.v*

For testing this section, I created an *intest.dat* file comprised of **four** input sequences. In my testbench, more than four rows can be analyzed by adding them to the input file and by modifying the parameter *MAX* value to match the number of 16-bit rows provided.

The *intest.dat* filed was created with the following input sequences:

0100110011101100 🡺 1st sequence was provided in problem description.

1101101110110111

0011011101111101

1111110111010010

In running my test bench, the output file *outtest.dat* is created in my project directory, and yields the following results:

0000000000000110 🡺 1st output **matches** sequence provided in problem description.

0000010010011001

0000000100110000

0000000011001000

Here are the I/O sequences, stacked on top of each other, for easy analysis:

Seq1(I): 0100110011101100

Seq1(O): 0000000000000**11**0

Seq2(I): 1101101110110111

Seq2(O): 00000**1**00**1**00**11**00**1**

Seq3(I): 0011011101111101

Seq3(O): 0000000**1**00**11**0000

Seq4(I): 1111110111010010

Seq4(O): 00000000**11**00**1**000

**These results are correct given the Moore implementation output delay of 1 clock cycle.**

*Descriptions on how the input is read, processed bit-by-bit at each clock cycle, and written to the output,* ***can be found in detailed comments contained within both Verilog files****.*

This concludes the analysis for Test 1, Part 1.

***Test 1: PART 2***

**Part A**

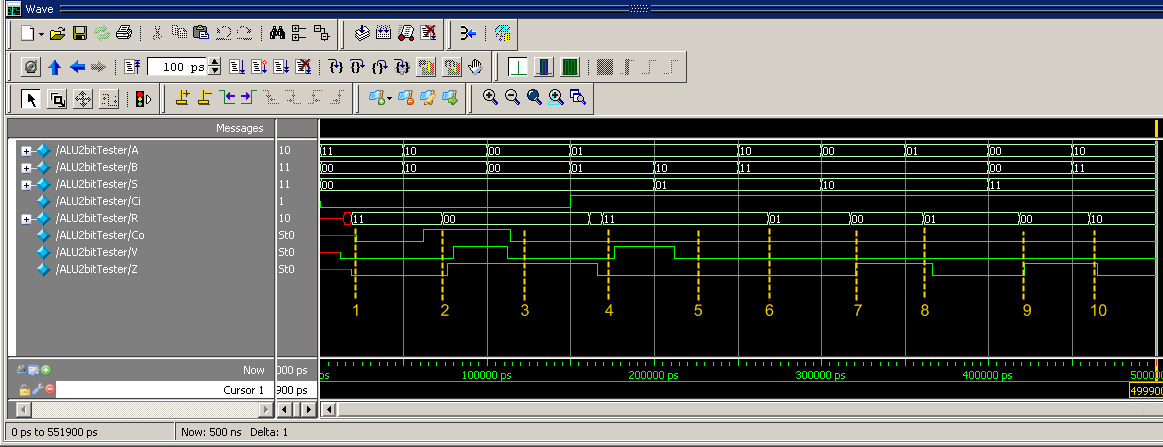
The Verilog code for this section can be found in the Pt 2, Pt A folder of the ZIP file:

*ALU2bit.v*

*ALU2bitTester.v*

In running the ALU2bitTester.v simulation, the following output is seen:

**Simulation Waveform Output of ALU2bitTester**

**

The image capture above serves to showcase the full functionality of the 2 bit ALU slice. Below are descriptions of each of the individual timestamps:

1. ALU is in ADD mode (S = 00), with A = 11, B = 00, Ci = 0
   * As expected: R = 11, Co = 0, V = 0, Z = 0
2. ALU is in ADD mode (S = 00), with A = 10, B = 10, Ci = 0
   * As expected: R = 00, Co = 1, V = 1, Z = 1
3. ALU is in ADD mode (S = 00), with A = 00, B = 00, Ci = 0
   * As expected: R = 00, Co = 0, V = 0, Z = 1
4. ALU is in ADD mode (S = 00), with A = 01, B = 01, Ci = 1
   * As expected: R = 11, Co = 0, V = 1, Z = 0
5. ALU is in XOR mode (S = 01), with A = 01, B = 10
   * As expected: R = 11, Z = 0
6. ALU is in XOR mode (S = 01), with A = 10, B = 11
   * As expected: R = 01, Z = 0
7. ALU is in AND mode (S = 10), with A = 00, B = 10
   * As expected: R = 00, Z = 1
8. ALU is in AND mode (S = 10), with A = 01, B = 11
   * As expected: R = 01, Z = 0
9. ALU is in Transparent mode (S = 11), with A = 00, B = 00
   * As expected: R = 00, Z = 1
10. ALU is in Transparent mode (S = 11), with A = 10, B = 11
    * As expected: R = 10, Z = 0

This concludes the analysis for Test 1, Part 2A.

**Part B**

The Verilog code for this section can be found in the Pt 2, Pt B folder of the ZIP file:

*ALU2bit.v*

*ALU16bit.v*

Detailed comments on the 16-bit ALU implementation can be found in the Verilog files.

This concludes the analysis for Test 1, Part 2B.

**Part C**

The Verilog code for this section can be found in the Pt 2, Pt C folder of the ZIP file:

*ALU2bit.v*

*ALU16bit.v*

ALU16bitTimeTest.v

The following is a hand analysis of the worst-case delay of the system, which occurs when:

A = 0000000011111111

B = 1111111100000000

Ci = 1

All expressions will be in terms of the gates and wires as denoted in the file *ALU2bit.v*. This was determined by backtracking from the output that was looking to be changed, and seeing where the timing resolutions occurred in the code.

* Delay to 1st bit resolution, assuming S = 00 (ADD):
  + + 3ns – o\_xor\_1 set to 1, so use tPLH of #3 (*ALU2bit.v*, Line 30)
  + + 5ns – R0\_S00 set to 1, so use tPHL of #5 (*ALU2bit.v*, Line 31)
  + + 6ns – R0\_out00 set to 0, so use tPHL of #6 (*ALU2bit.v*, Line 94)
  + + 7ns – R[0] set to 0, so use tPHL of #7 (*ALU2bit.v*, Line 100)
* Total Delay of +21ns
* Delay to 2nd bit resolution, assuming S = 00 (ADD):
  + + 3ns – o\_xor\_1 will be set to 1, so use tPLH of #3 (*ALU2bit.v*, Line 30)
  + + 5ns – o\_nand1\_1 set to 0, so use tPHL of #5 (*ALU2bit.v*, Line 34)
  + + 3ns – o\_cin set to 1, so use tPLH of #3 (*ALU2bit.v*, Line 36)
  + + 5ns – R1\_S00 set to 0, so use tPHL of #4 (*ALU2bit.v*, Line 46)
  + + 6ns – R1\_out00 set to 0, so use tPHL of #6 (*ALU2bit.v*, Line 107)
  + + 7ns – R[1] set to 0, so use tPHL of #7 (*ALU2bit.v*, Line 113)
* Total Delay of +29ns
* Note: The time it takes to resolve the 2nd bit in comparison to the 1st bit of the ALU2bit is:

29ns – 21ns = 8ns

* Note: As the 1st ALU bit is being resolved, processing on the next bits causes the 1st bit of the

next ALU2bit slice to be available at 41ns. This means that the difference between 1st

bit ALU resolution is 41ns – 21ns = 20+ ns.

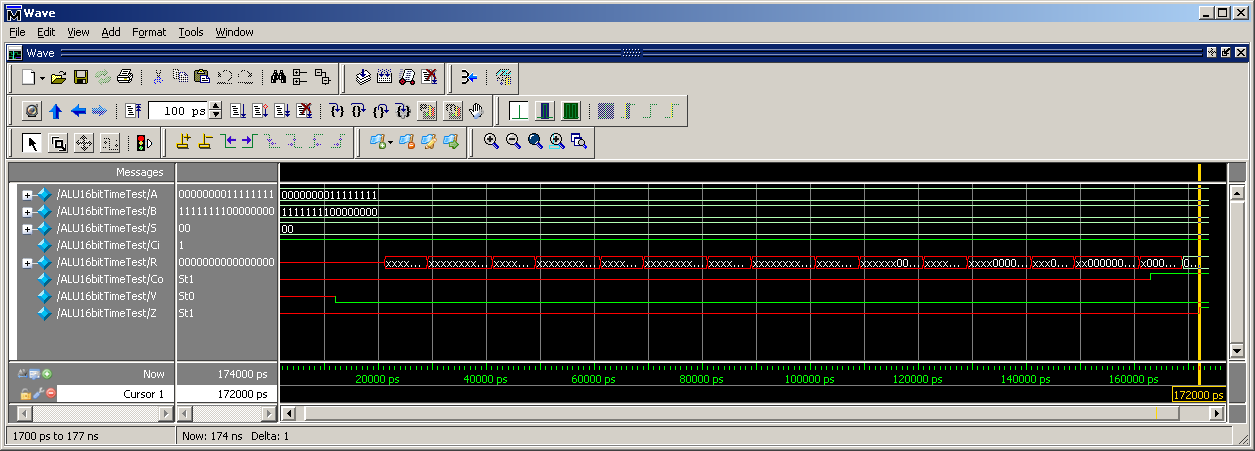
With this information, build the total estimate:

* Total = 29ns: time to resolve 1st and 2nd bits (see above)
* Total = 169ns: (+ 20ns \* 7 = 140ns), for remaining seven groups of 2bits
* Total = 172ns: +3ns since Z is set to 1 once R[15] is resolved, so use tPHL of #3

Therefore, the total time it \*should\* take to resolve the arithmetic of the circuit, with respect to all gate-primitive delays assigned in *ALU2bit.v* is: **172ns**

This hand-calculation is confirmed in the waveform output below, for the worst-case timing delay:

**Simulation Waveform Output of ALU16bitTimeTest**



As seen in the image above, the total time to compute the final output, the zero flag (Z), puts the circuit at a total time of **172ns**, which matches the hand calculation performed.

This concludes the analysis for Test 1, Part 2C.

**Part D**

The Verilog code for this section can be found in the Pt 2, Pt D folder of the ZIP file:

*ALU16bitAlways.v*

This implementation uses an always block that has a sensitivity list comprised of all the inputs:

*// Use an always statement to detect changes in either:*

*// Ain or Bin (inputs), S (mode of ALU), or Ci (carry-in)*

*always @(A, B, S, Ci) begin*…

Please see the code for comments on how this section was modeled.

This concludes the analysis for Test 1, Part 2D.

**Parts E & F**

The Verilog code for this 1st section can be found in the Pt 2, Pt E folder of the ZIP file:

*ALU16bitAlways.v*

The Verilog code for this 2nd section can be found in the Pt 2, Pt F folder of the ZIP file:

*ALU2bit.v*

*ALU16bit.v*

*ALU16bitAlways.v*

*ALU16bitModelComparison.v*

Using timing analysis taken from Pt2 PtB, the following timing was incorporated as output pin-to-pin delays in the Verilog module for **worst** **case** delay analysis:

*specify*

*(A,B,S,Ci \*> Co) = 163;*

*(A,B,S,Ci \*> R) = 169;*

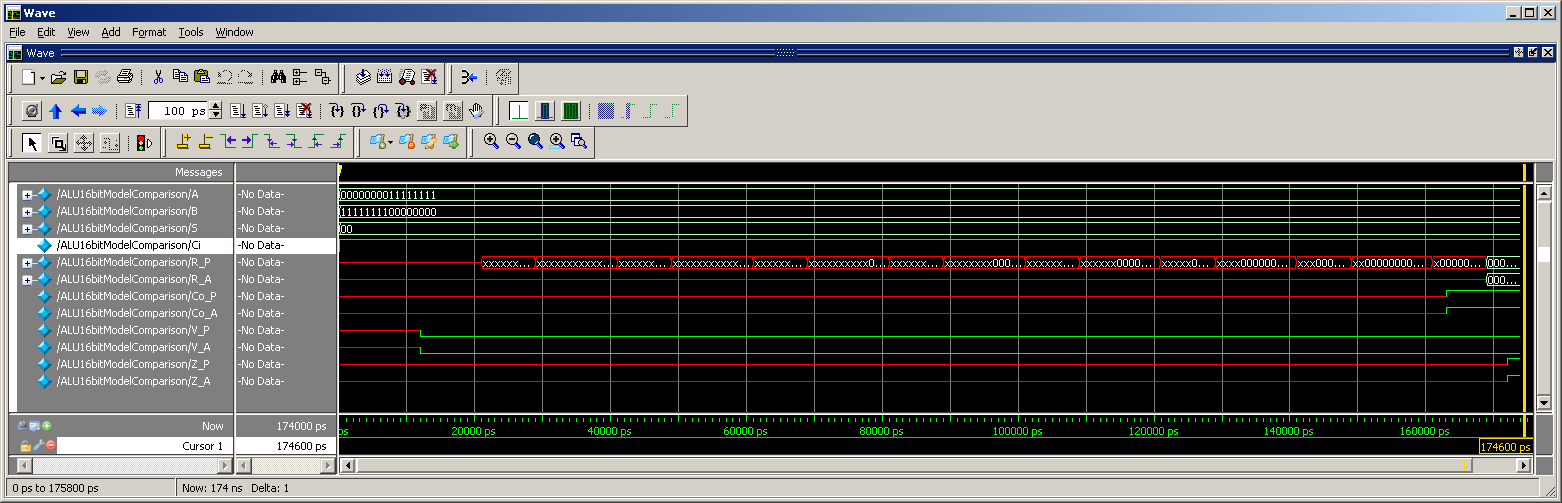
*(A,B,S,Ci \*> V) = 12;*

*(A,B,S,Ci \*> Z) = 172;*

*endspecify*

In defining both UUTs, the *Primitive* and *Always* versions, the following output graph was generated for the Worst Case timing scenario:

**Simulation Waveform Output of ALU16bitModelComparison.v – Scenario 1**



In running through the other different functions of the ALU, I’ve encapsulated all timing information from the waveform graphs into an Excel table in order to compute the **average** pin-to-pin delays for all four different functions. In order to be valid, I utilized the same data set for all four functions of the ALU in my testbench.

A <= 16'b1100011010111000;

B <= 16'b0001010110100100;

S <= 2'b??; 🡺 ?? changew depending on the ALU function

Ci <= 0;

Here is the table with the *final* *pin-to-pin* delays used in the *Always* 16bit ALU:

**Average ALU Output Pin-to-Pin Delays Based on ALU Averaging**



These were added into the *ALU16bitAlways.v* file, and is used when not running the worst case scenario.

*specify*

*(A,B,S,Ci \*> R) = 31;*

*(A,B,S,Ci \*> Co) = 20;*

*(A,B,S,Ci \*> V) = 28;*

*(A,B,S,Ci \*> Z) = 35;*

*endspecify*

This concludes the analysis for Test 1, Parts 2E & 2F.

***Test 1: PART 3***

The Verilog code for this section can be found in the Pt 3 folder of the ZIP file:

*VotingControl.v*

*\*All Project files associated with the Quartus II realization of the VotingMachine*

The *VotingControl.v* file contains all of the primary processing that’s responsible for the realization of this Verilog Voting Machine.

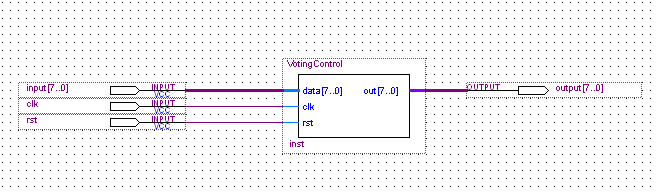
Given the problem statement, and given that this was a Voting Machine implementation, I chose to make the following assumption before programming this Verilog Module:

**The Voting Machine would only receive one bit as HIGH (= 1) during any input sample**.

This forced my waveform models and verification of the circuit to only deal with 8 unique states. I feel that this adequately models a Voting Machine, as with traditional voting, you are only allowed to vote for one particular candidate.

This circuit implementation was realized in Quartus II, with the following BDF schematic file:

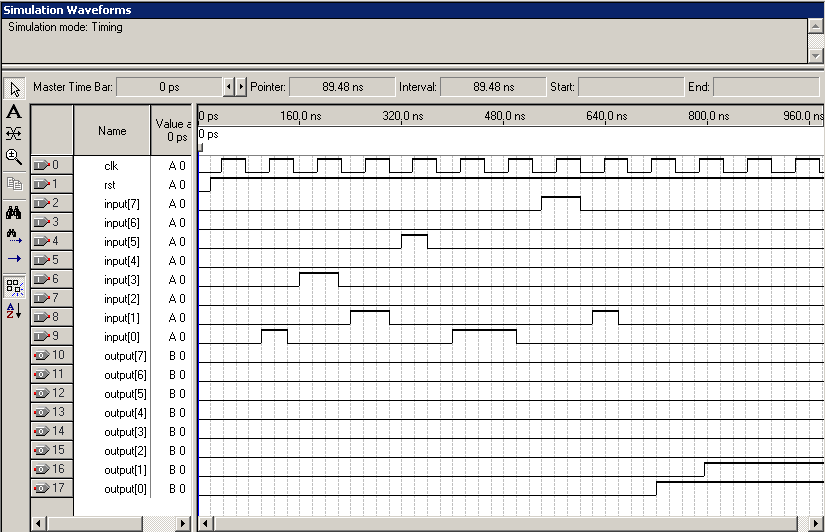
**VotingMacine.bdf Block Diagram / Schematic**



I instantiated my Verilog code as a block, Voting Control, in the schematic and used Quartus II pins in order to model possible inputs and show the corresponding outputs.

The following pages depict three (3) scenarios that were constructed in order to validate the construction of the circuit and its overall operation.

**VotingMachine-S1 : Simulation Report**



The following is the timetable of events:

@ CLK 1 > rst = 1, so enable circuit on next clk

@ CLK 2 > **data** = 0000000**1**

@ CLK 3 > data = 00001000

@ CLK 4 > data = 00000010

@ CLK 5 > data = 00100000

@ CLK 6 > **data** = 0000000**1**

@ CLK 7 > **data** = 0000000**1**

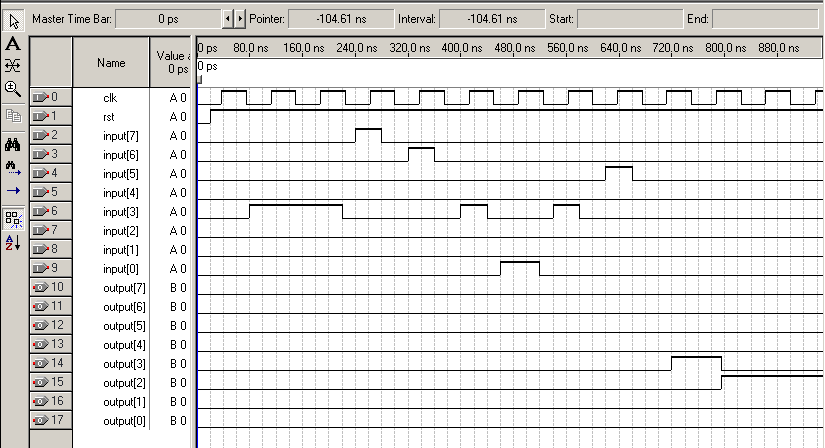
@ CLK 8 > data = 10000000

@ CLK 9 > data = 00000010

@ CLK 10 > output = 0000000**1** 🡺 **Correct**, since state 0000000**1** is the most repeated.

@ CLK 11 > output = 000000**11** 🡺 **Correct**, since state is the repeated 000000**11** (**3**) times.

**VotingMachine-S2 : Simulation Report**



The following is the timetable of events:

@ CLK 1 > rst = 1, so enable circuit on next clk

@ CLK 2 > **data** = 0000**1**000

@ CLK 3 > **data** = 0000**1**000

@ CLK 4 > data = 10000000

@ CLK 5 > data = 01000000

@ CLK 6 > **data** = 0000**1**000

@ CLK 7 > data = 00000001

@ CLK 8 > **data** = 0000**1**000

@ CLK 9 > data = 00100000

@ CLK 10 > output = 0000**1**000 🡺 **Correct**, since state 0000**1**000is the most repeated.

@ CLK 11 > output = 00000**1**00🡺 **Correct**, since state is the repeated 00000**1**00 (**4**) times.

**VotingMachine-S3 : Simulation Report**



The following is the timetable of events of what can be considered a **landslide victory**:

@ CLK 1 > rst = 1, so enable circuit on next clk

@ CLK 2 > **data** = 0**1**000000

@ CLK 3 > **data** = 0**1**000000

@ CLK 4 > **data** = 0**1**000000

@ CLK 5 > **data** = 0**1**000000

@ CLK 6 > data = 00010000

@ CLK 7 > **data** = 0**1**000000

@ CLK 8 > **data** = 0**1**000000

@ CLK 9 > **data** = 0**1**000000

@ CLK 10 > output = 0**1**000000 🡺 **Correct**, since state 0**1**000000 is the most repeated.

@ CLK 11 > output = 00000**111** 🡺 **Correct**, since state is the repeated 00000**111** (**7**) times.

This concludes the analysis for Test 1, Part 3.