Jessica Diller – **Analog to Digital Converter Type Analysis** – 12/15/2015

I researched types of Analog to Digital Converters (ADCs) and their performance characteristics like timing and space. The two of interest I chose to focus on are the successive approximation ADC and the flash ADC. I found a chip that used each type of conversion and thus could examine how the types of conversions translated directly to the real digital application.

Successive Approximation Analog to Digital Converters:

Analog part:

The *Sample and Hold* circuit samples the input analog voltage, then holds and outputs that voltage until the next sample period

The *Comparator* compares the output of the sample and hold circuit to the output of the Digital to Analog Converter (DAC) and outputs this difference to the Successive Approximation Register (SAR).

The *Digital to Analog Converter* takes digital output of the successive approximation register and converts to analog – can be a resistor or transistor ladder

Digital part:

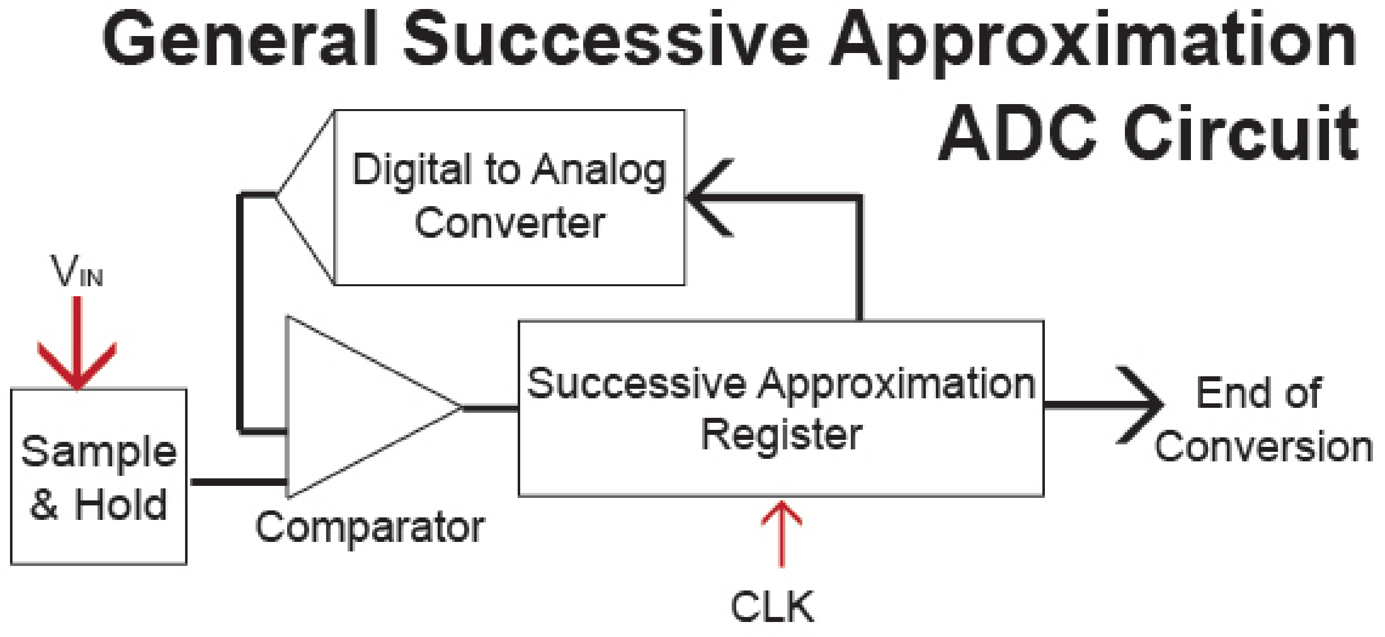
The *Successive Approximation Register* takes in output from comparator and uses it to decide what value is put in each register and outputs estimated digital bits to the DAC. When all the registers have been verified correct, it has reached the end of conversion and outputs them all to a separate port.

Function of a Successive Approximation Register:

The SAR functions in a loop. It begins by setting the Most Significant Bit (MSB) to 1 and all the other bits to 0. It outputs this set of register values to the DAC for analog comparison against the input signal. If the SAR's output analog voltage is higher than the DAC's, the MSB is left unchanged at 1. If the SAR's output analog voltage is lower than the DAC's, then the MSB stored in the register is changed to 0. The process then continues with the next most significant bit and so on down the bits. Once the least significant bit is reached, the input analog voltage and the analog-converted digital register values are determined as approximately equal. The ADC has reached its end of conversion and the SAR holds the approximate digital equivalent of the input analog voltage.

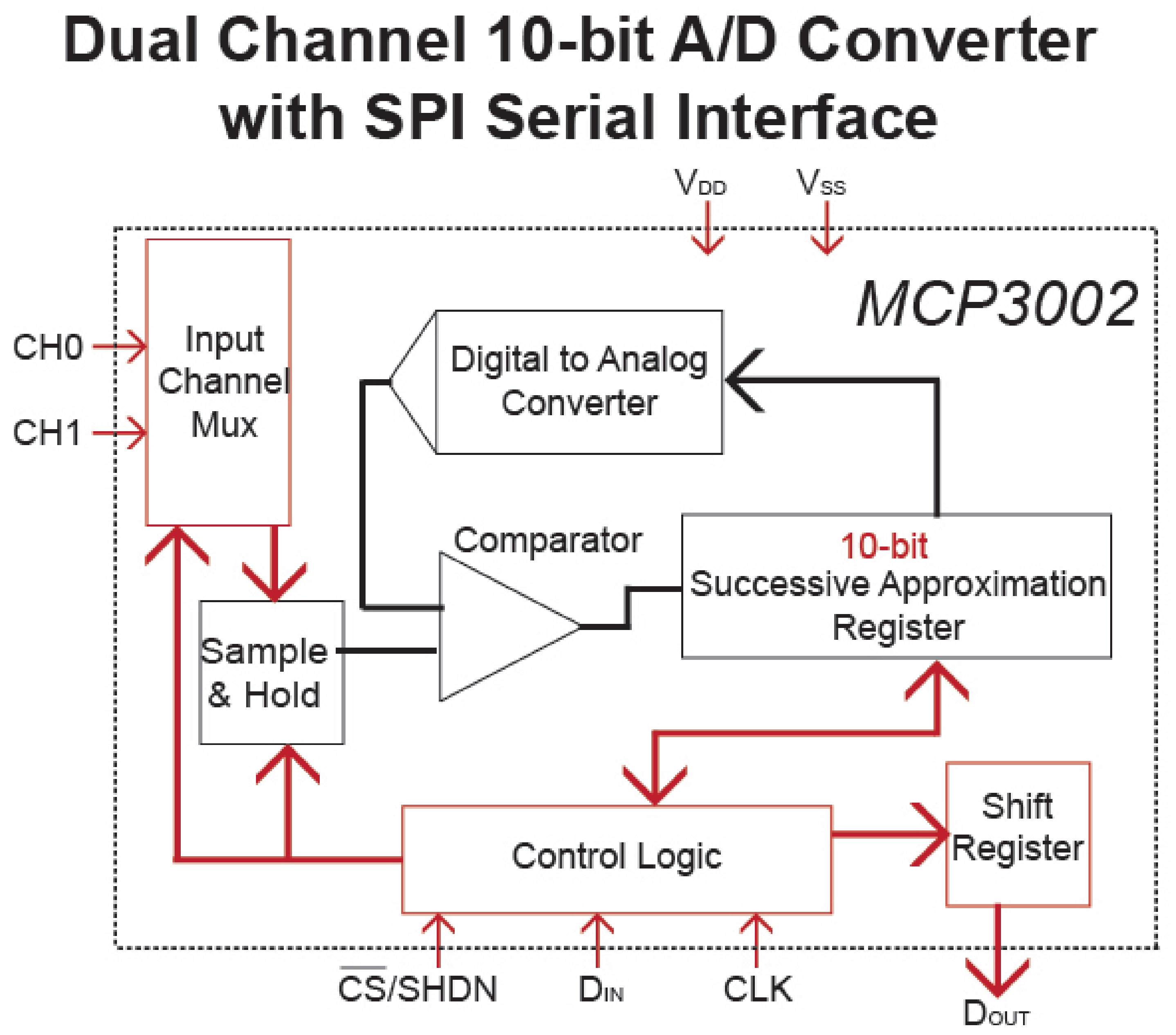
Characteristics Compared to Other ADCs:

The timing depends on the settling time of the Digital to Analog Converter within the large analog to digital converter. The type of DAC needs to be taken into consideration as well as it's functioning within the larger ADC system. Both that and the comparator are parts that if high quality, could contribute to making the ADC as a whole function much better, but if low quality could drastically slow down the entire chip's functioning. Those are the two limiting components to speed and accuracy. Successive Approximation ADCs require low levels of power compared to other ADCs and increased resolution of bits doesn't increase complexity or space requirements drastically either.


General Successive Approximation ADC Circuit
Digital to Analog Converter
V_{IN}
Sample & Hold
Comparator
Successive Approximation Register
End of Conversion
CLK

<u>The Successive Approximation Chip I Chose:</u>

I chose the MCP3002 form Microchip. It is a 2.7V Dual Channel 10-bit A/D Converter with SPI Serial Interface. Its operating voltages are between 2.7V and 5.5V. Its typical operating current is between 300-525 micro amps and has a maximum operating current of 650 micro amps. It is functional at any temperature from -40 to +85 degrees Celsius. See successive approximation chip data sheet [Appendix D].

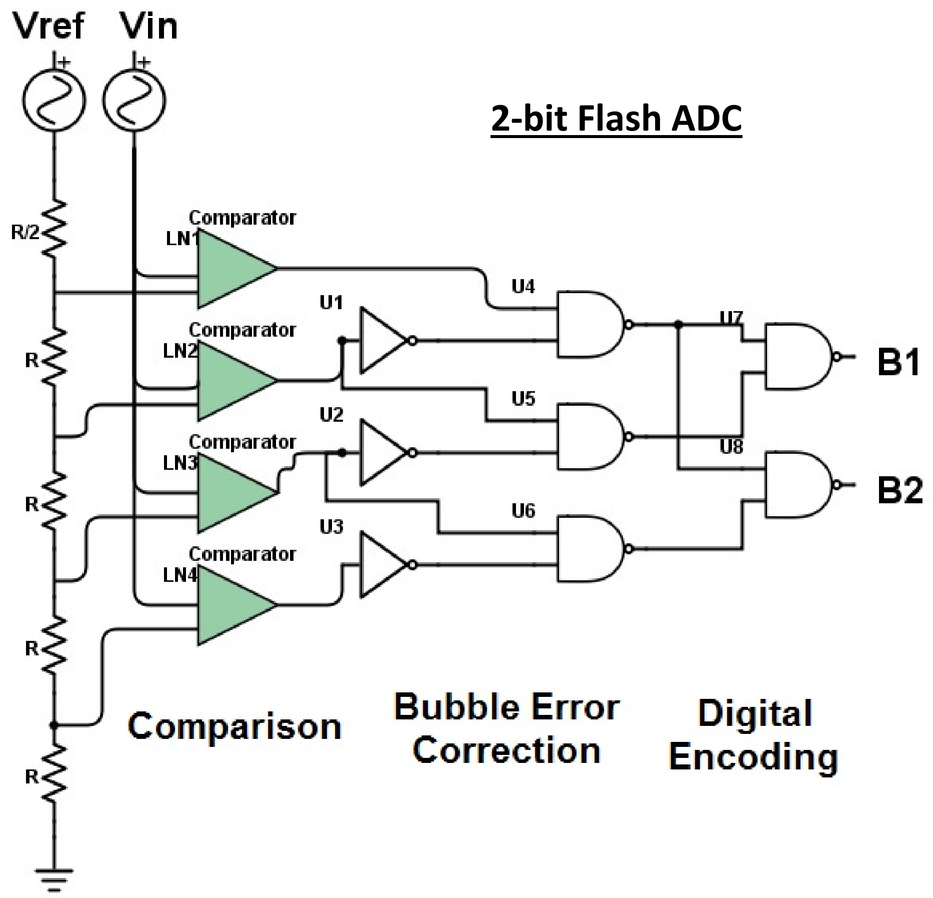

Dual Channel 10-bit A/D Converter with SPI Serial Interface
VDD
VSS
MCP3002
CH0
CH1
Input Channel Mux
Digital to Analog Converter
Comparator
10-bit Successive Approximation Register
Sample & Hold
Control Logic
Shift Register
CS/SHDN
DIN
CLK
DOUT

This is a black box diagram specific to the chip I chose. The red colored boxes and arrows are the areas specific to the chip while the black colored parts are the ones that are similar to all successive approximation ADCs (see previous diagram).

The input channel multiplexer picks whether CH0 or CH1 voltage is sampled and held. Control logic enables changes in the input channel mux selection, when sample and hold is occurring, which register bit is currently being tested in the successive approximation register (including any restarts needed), selects shift register bit and when to output it, and most importantly clocks all digital aspects of the system. [Appendix D]

Flash Analog to Digital Converters:

Each comparator generates a logical bit depending on how the values of Vin compare to the reference voltage ladder. If Vin is greater than the reference voltage for that comparator, it outputs a 1 and vice versa. You could get any degree of accuracy in numbers of bit depending on the amount of resistors and comparators that were part of the system. This is the simplest implementation of this technology, but generally error correction mechanisms are included as well. Amplifiers are sometimes added to amplify the voltage difference to suppress comparator offset and kickback noise of the comparator towards the reference ladder. To address the risk that a single comparator will report the incorrect value, logic gates are added after the comparators. If all the surrounding comparators are tripping high or low, then the not and and gates corrects the error if one randomly trips the wrong code. See diagram below for specifics on that circuit.



This type of analog to digital converter is one of the fastest out there. It can be clocked for power consumption, but generally isn't and thus the timing is only restrained by the time each logic gates takes. It does take up large amounts of space when you get to larger number of bits, as a consequence of it needing 2^n-1 comparators, so it's considered impractical for any clarity more than 8-bits. [Appendix D]

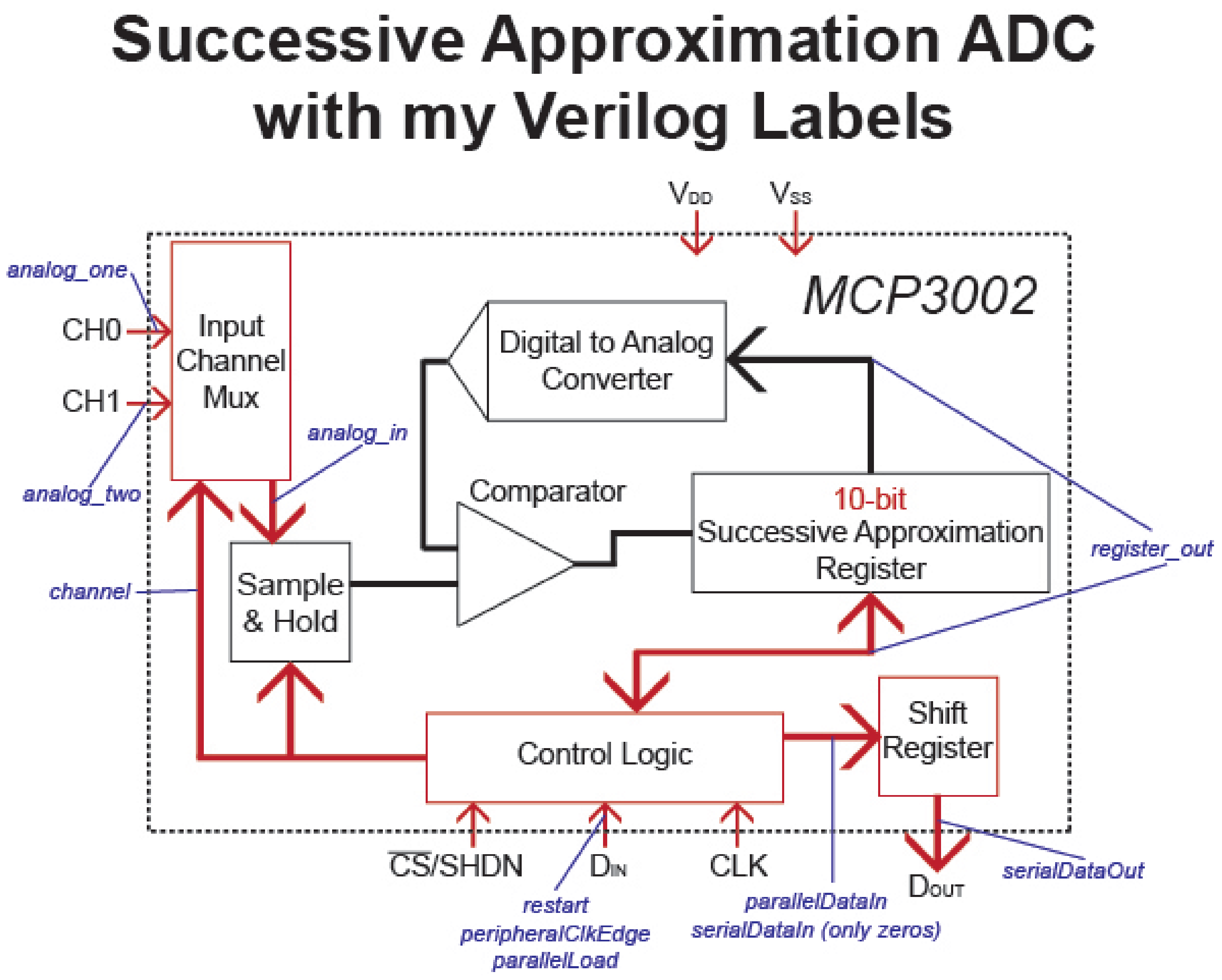
Why I Did It

I wanted to increase my knowledge of Analog to Digital Converters, increase my data sheet analysis skills as it relates to computer architecture chips and characteristics, and explore Verilog implementation of known parts (from data sheets).

How I Did It

Verilog Implementation:

The variables used in my code are explained on an edited version of the chip black box diagram below.



Test One – Analog Simulated Section: First, I created the section that would model the analog section of the device, which included the Sample and Hold circuit, the Comparator, and the Digital to Analog Converter. I researched ways other had implemented similar structures and any digital representations of analog signals. First, I attempted to use Verilog's "$bitstoreal" function which would allow a 64-bit double precision number to be treated as a real number. It was not very clearly documented online, I didn't previously have the basis of knowledge to understand this way of representing numbers in Verilog, and the smaller scope of this project led me to, after a couple hours of internet searching, to simplify analog signals to their equivalent in digital bits. I think that it's a very bad simulation of an analog circuit and that section of the system, but the digital computation and control were still able to be programmed. Since I knew I still would be able to simulate the digital, more important aspects of the system, I was okay with my choice and the ability it gave me to implement the whole system of the chip. As you can see in the Test 1 [Appendix A], *digital_result* from the comparator section is outputting appropriately based on different *analog_in* and *digital_in* bits.

Test Two – Successive Approximation Register: I then made the actual successive register part of the ADC. It took an input of *digital_result* and changed each bit of *register_out* accordingly. I also introduced the *restart* control to add a level of control complexity. *Restart* ended up being the edge to which the final system began operation, but it also allows the process of *register_out* loading to be interrupted and restarted for any reason. As you can see in Test 2 [Appendix A],, the *digital_result* value is loaded into a bit of *register_out* every clock cycle as expected.

Test Three – Successive Approximation Calculation: The next move was to combine the successive approximation register with the *analog_in* comparator calculation. As you can see in Test 3 [Appendix A], *analog_in* is bit by bit loaded into *register_out* through the same mechanisms as described in tests one and two.

Test Four – ADC with Input Channel Mux: Now that I had the basic ADC functionality working, it was on to implementing the more specific aspects of the chip I chose. The first of these aspects I chose to implement was the input channel multiplexer. The two input channels are *analog_one* and *analog_two*. When the input to the function *channel*=0, *analog_in* is set to *analog_one*. When *channel*=1, *analog_in* is set to *analog_two*. The digital logic picks which analog input is sampled and held to use as input to the comparator. As you can see in Test 4 [Appendix A], *channel* controls which of the two inputs is translated to *register_out*. In the first half of the simulation, *channel*=0 and *register_out* is digitally loaded to approximate *analog_one* while, in the second half of the simulation, *channel*=1 and *register_out* is digitally loaded to approximate *analog_two*.

Test Five – ADC with Shift Register: The second additional feature to add was a shift register controlling serial output. I used shift register code from previous labs and altered a few parts just to ensure functionality for this application. It was mostly just aligning the previous value with value that already existed inside my ADC module. As you can see in Test 5 [Appendix A], I got the expected results. The appropriate *analog_number* is loaded into *register_out* depending on *channel* value and *register_out* bit values are loaded into *serialDataOut* depending on when *parallelLoad* and *peripheralClkEdge* are enabled.

Lessons I Learned:

If you have a functioning module or piece of the code, don't just change it because you think it might be a good idea. Modules are great and they're made to be placed into other code easily. It's better to find a way to keep the module more general and perhaps even have high amounts of inputs than hardcode something in. When I was adding the shift register into my system code, I decided to change where it had "width" as a parameter into pure hardcoding of the width of each bus I was using. It added a lot of complications as I tried to debug the connection between the shift register module and my wrapper modules because I would change one instance or all of them but one and get resulting errors. It added time to my project that didn't need to be added. I ended up switching it back to the width later anyways for ease of understanding the code and in case I would need to change it at an even later date.

Greater than does not mean greater than or equal to. I need to make sure my if statements cover all possible scenarios that could arise if I want to use the else statement.

A simple switch of number can make everything magically work.

Wires aren't numbers and need to be set according to their structure.

Regs, wires, initial, assign, always @ blocks, and other aspects of Verilog have interesting relationships and I should know them when designing in Verilog. I ran into an issue where I needed to set an initial value, but couldn't because the reg needed to be set in the always @block. That's what control inputs are for! I solved it by making an if statement for initial setting of variables I needed set.

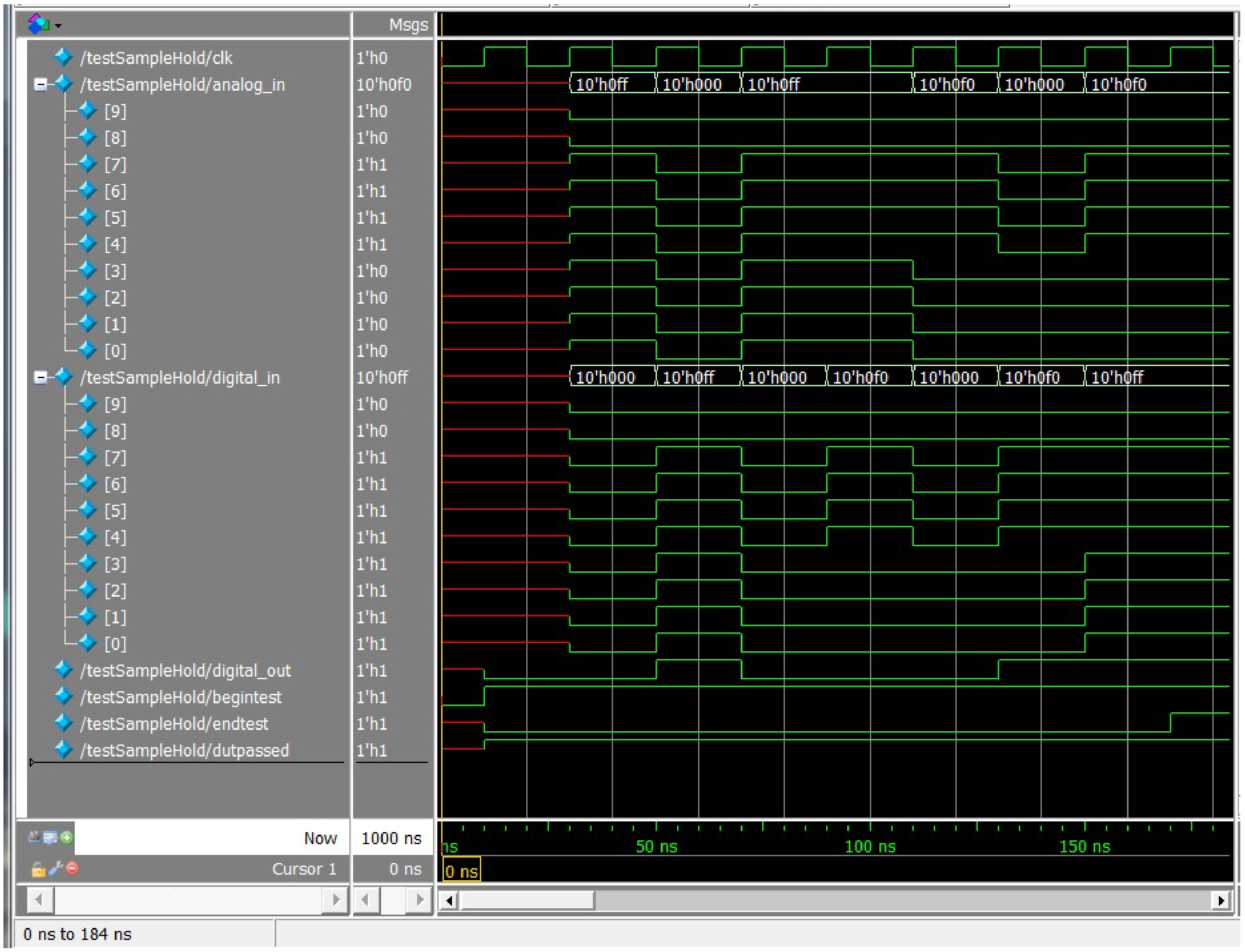
Possible Next Steps:

Someone could implement the analog input signal in 64-bit double precision number or using a shell program or hardware to model it more accurately. Also, more analog to digital converter types or more chips of the same types could be included in a more rigorous comparison analysis.

Appendix A: Test Bench Results

Test [1] – Sample & Hold, Comparator, and DAC

analog_in = analog input signal represented in digital bits
digital_in = analog signal from DAC represented in digital bits
digital_out = outputs one if analog_in is larger or the same as digital_in
clk = controls loading of new register values



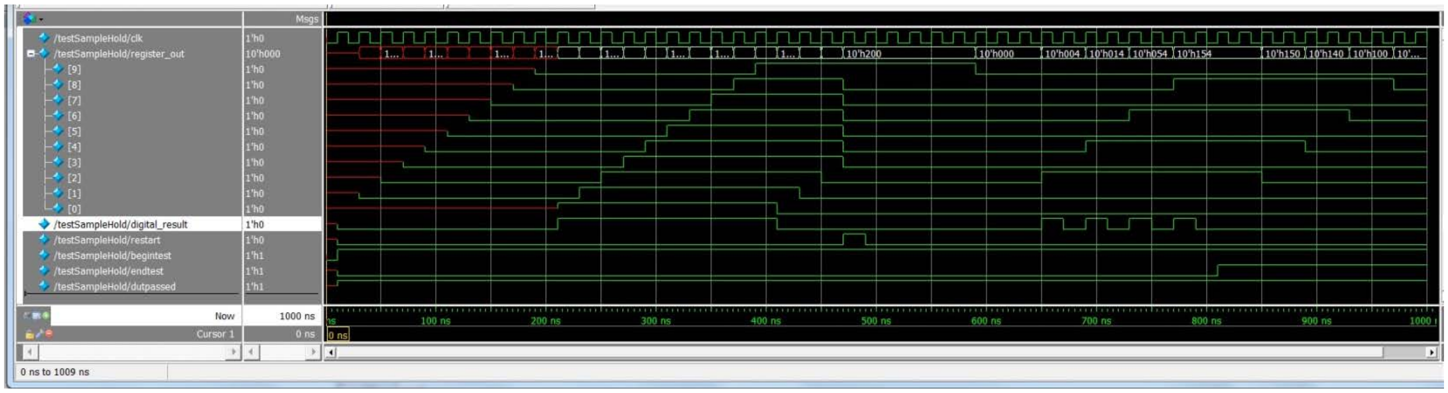
Msgs
/testSampleHold/clk 1'h0
/testSampleHold/analog_in 10'h0f0
[9] 1'h0
[8] 1'h0
[7] 1'h1
[6] 1'h1
[5] 1'h1
[4] 1'h1
[3] 1'h0
[2] 1'h0
[1] 1'h0
[0] 1'h0
/testSampleHold/digital_in 10'h0ff
[9] 1'h0
[8] 1'h0
[7] 1'h1
[6] 1'h1
[5] 1'h1
[4] 1'h1
[3] 1'h1
[2] 1'h1
[1] 1'h1
[0] 1'h1
/testSampleHold/digital_out 1'h1
/testSampleHold/begintest 1'h1
/testSampleHold/endtest 1'h1
/testSampleHold/dutpassed 1'h1
10'h0ff 10'h000 10'h0ff 10'h0f0 10'h000 10'h0f0
10'h000 10'h0ff 10'h000 10'h0f0 10'h000 10'h0f0 10'h0ff
Now 1000 ns
Cursor 1 0 ns
ns 50 ns 100 ns 150 ns
0 ns
0 ns to 184 ns

Test [2] – loading the new digital value to the register

register_out = the bit values that are stored in the register

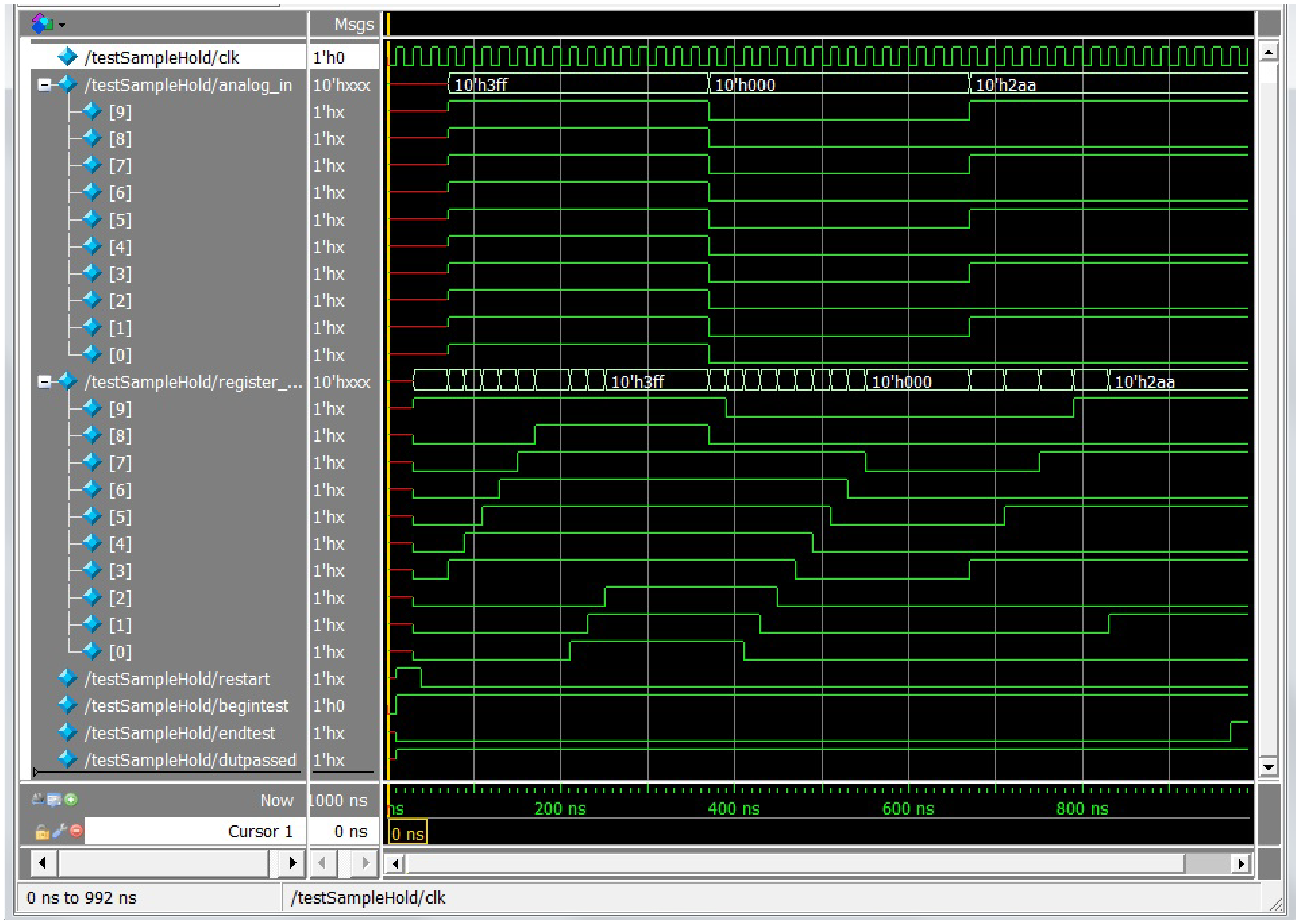
digital_result = the bit to put in the registers

restart = sets initial successive approximation register values (1 for the most significant bit and 0 for all others)



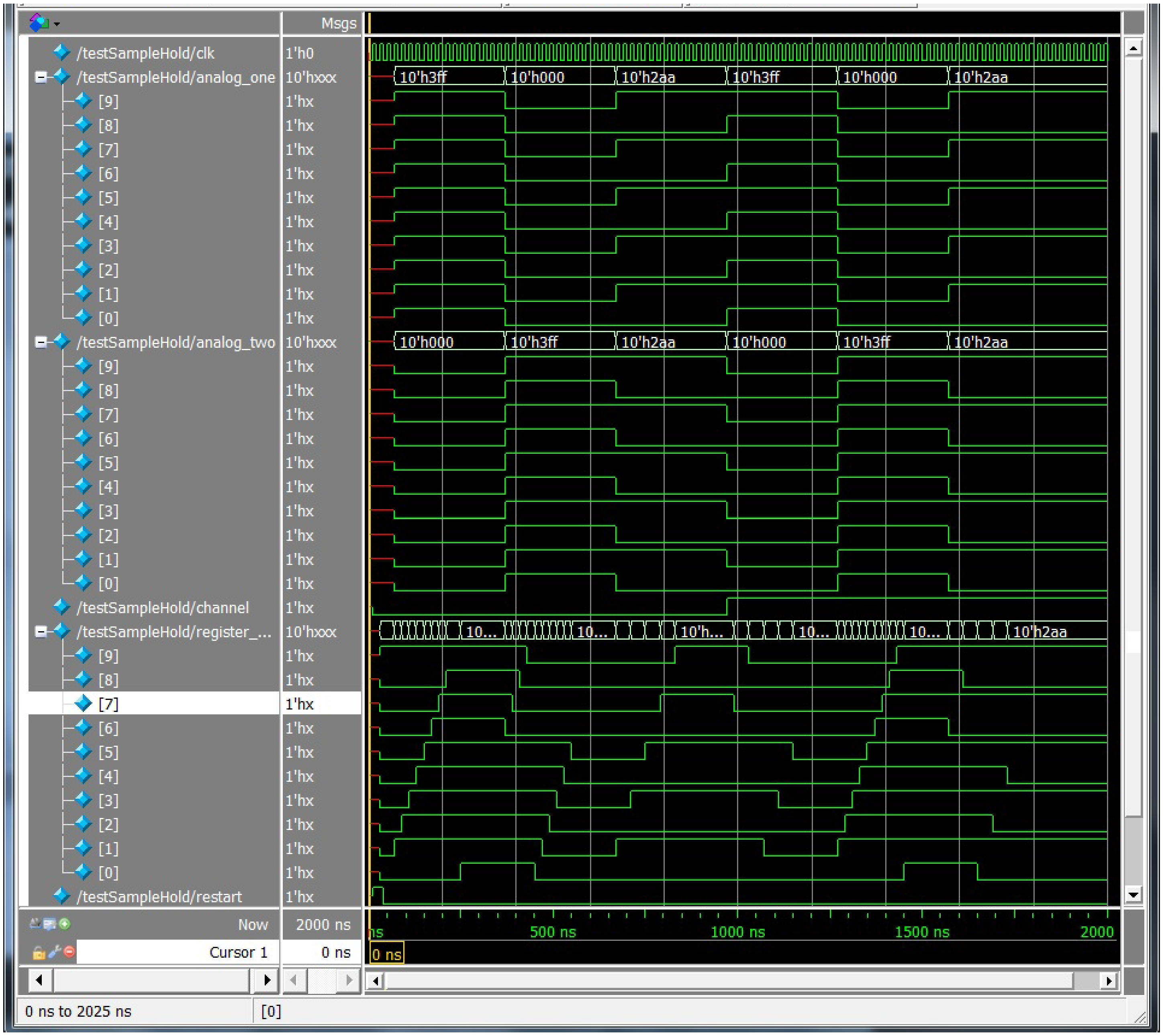
```
                                      Msgs
/testSampleHold/clk                   1'h0
/testSampleHold/register_out          10'h000
  [9]                                 1'h0
  [8]                                 1'h0
  [7]                                 1'h0
  [6]                                 1'h0
  [5]                                 1'h0
  [4]                                 1'h0
  [3]                                 1'h0
  [2]                                 1'h0
  [1]                                 1'h0
  [0]                                 1'h0
/testSampleHold/digital_result        1'h0
/testSampleHold/restart               1'h0
/testSampleHold/begintest             1'h1
/testSampleHold/endtest               1'h1
/testSampleHold/dutpassed             1'h1
                              Now     1000 ns
                         Cursor 1     0 ns
0 ns to 1009 ns
```

Test [3] – analog in controls register_out



Test [4] – two input channels to choose from

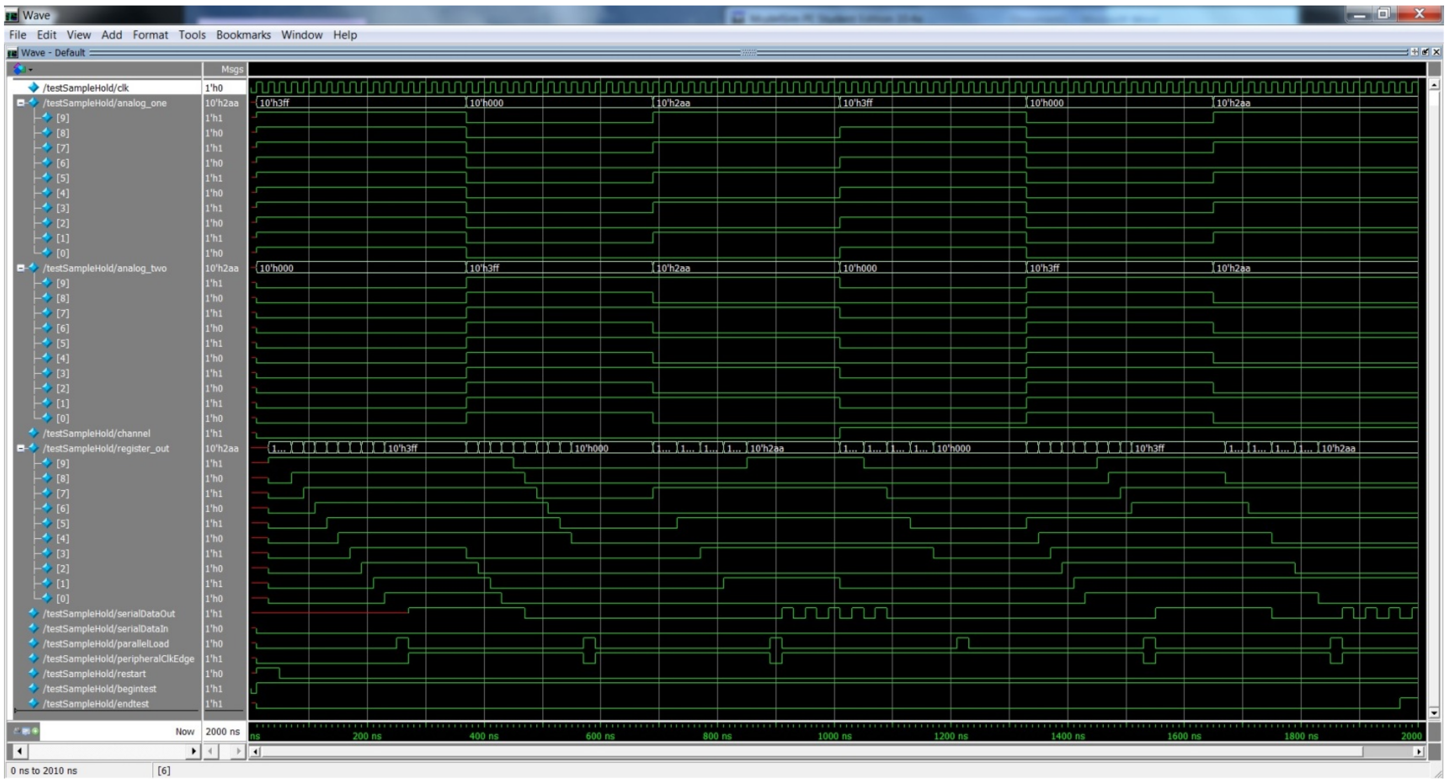
channel = controls whether analog_one or analog_two sets analog_in



```
                              Msgs
/testSampleHold/clk           1'h0
/testSampleHold/analog_one    10'hxxx
  [9]                         1'hx
  [8]                         1'hx
  [7]                         1'hx
  [6]                         1'hx
  [5]                         1'hx
  [4]                         1'hx
  [3]                         1'hx
  [2]                         1'hx
  [1]                         1'hx
  [0]                         1'hx
/testSampleHold/analog_two    10'hxxx
  [9]                         1'hx
  [8]                         1'hx
  [7]                         1'hx
  [6]                         1'hx
  [5]                         1'hx
  [4]                         1'hx
  [3]                         1'hx
  [2]                         1'hx
  [1]                         1'hx
  [0]                         1'hx
/testSampleHold/channel       1'hx
/testSampleHold/register_...  10'hxxx
  [9]                         1'hx
  [8]                         1'hx
  [7]                         1'hx
  [6]                         1'hx
  [5]                         1'hx
  [4]                         1'hx
  [3]                         1'hx
  [2]                         1'hx
  [1]                         1'hx
  [0]                         1'hx
/testSampleHold/restart       1'hx
Now                           2000 ns
Cursor 1                      0 ns
0 ns to 2025 ns               [0]
```

Test [5] – serial output

serialDataOut = the serial version of register_out


Wave
File Edit View Add Format Tools Bookmarks Window Help
Wave - Default
Msgs
/testSampleHold/clk 1'h0
/testSampleHold/analog_one 10'h2aa
/testSampleHold/analog_two 10'h2aa
/testSampleHold/channel 1'h1
/testSampleHold/register_out 10'h2aa
/testSampleHold/serialDataOut 1'h1
/testSampleHold/serialDataIn 1'h0
/testSampleHold/parallelLoad 1'h0
/testSampleHold/peripheralClkEdge 1'h1
/testSampleHold/restart 1'h0
/testSampleHold/begintest 1'h1
/testSampleHold/endtest 1'h1
10'h3ff
10'h000
10'h2aa
Now 2000 ns
200 ns
400 ns
600 ns
800 ns
1000 ns
1200 ns
1400 ns
1600 ns
1800 ns
0 ns to 2010 ns

Appendix B: Final Verilog Code

Analog to Digital Converter

```
 module ADC
(
  input                  clk,          // clock
  input                  restart,                //when brought high the cycles restart
  input [9:0]                          analog_one,
  input [9:0]                          analog_two,
  input                  channel,
  output reg [9:0]       register_out    // 10 bits stored in register
);
reg [9:0] analog_in;
parameter counterwidth=10; // Counter size
reg[counterwidth-1:0] counter=0; //?? bit bus
always @(posedge clk) begin
          //input mux
          if (channel==0) begin
                    analog_in = analog_one;
          end
          if (channel==1) begin
                    analog_in = analog_two;
          end
          //restart option
          if (restart==1 )begin
                    register_out[9] <= 1;
                    register_out[8:0] <= 0;
                    counter <= 9;
          end
          //Successive Approximation Register and Logic
          else begin
                    if (counter==0 ) begin
                              counter <= 9;
                              if (register_out[counter] > analog_in[counter]) begin
                                        register_out[counter] <= 0;
                              end
                              if (register_out[counter] < analog_in[counter]) begin
                                        register_out[counter] <= 1;
                              end
                    end
                    else begin
                              counter <= counter-1;
                              if (register_out[counter] > analog_in[counter]) begin
                                        register_out[counter] <= 0;
                              end
                              if (register_out[counter] < analog_in[counter]) begin
                                        register_out[counter] <= 1;
                              end
                    end
          end
end
endmodule
```

Shift Register

```
module shiftregister
#(parameter width = 10)
(
input            clk,            // FPGA Clock
input            peripheralClkEdge,  // Edge indicator - when this goes high, load whatever serial data is
input            parallelLoad,      // 1 = Load shift reg with parallelDataIn
input  [width-1:0]  parallelDataIn,    // Load shift reg in parallel
input            serialDataIn,      // Load shift reg serially
output reg [width-1:0]  parallelDataOut,    // Shift reg data contents
output reg           serialDataOut      // Positive edge synchronized
);
  reg [width-1:0]      shiftregistermem;
  always @(posedge clk) begin
          if (peripheralClkEdge == 1) begin //when the peripheralClkEdge goes high
```

```
                shiftregistermem <= {shiftregistermem[width-2:0], serialDataIn};
                end
                if (parallelLoad == 1) begin
                shiftregistermem <= parallelDataIn;
                end
                parallelDataOut <= shiftregistermem[7:0];
                serialDataOut <= shiftregistermem[width-1];
   end
endmodule
```

Test Wrapper Code

```
module testSampleHold();
   wire clk;
   wire [9:0] analog_one;
   wire [9:0] analog_two;
   wire [9:0] register_out;
   wire channel;
   wire peripheralClkEdge;
   wire parallelLoad;
   wire [9:0] parallelDataIn;
   wire [9:0] parallelDataOut;
   wire serialDataOut;
   wire digital_result;
   wire restart;
   reg begintest;
   wire endtest;
   wire dutpassed;
   ADC digital (.clk(clk),
                                .restart(restart),
                                .analog_one(analog_one),
                                .analog_two(analog_two),
                                .channel(channel),
                                .register_out(register_out)
                                );
   shiftregister spi_out (.clk(clk),
                                .peripheralClkEdge(peripheralClkEdge),
                                .parallelLoad(parallelLoad),
                                .parallelDataIn(register_out),
                                .serialDataIn(serialDataIn),
                                .parallelDataOut(parallelDataOut),
                                .serialDataOut(serialDataOut)
                                );
   SampleHoldTestBench tester(
                .begintest(begintest),
                .endtest(endtest),
                .dutpassed(dutpassed),
                .clk(clk),
                .analog_one(analog_one),
                .analog_two(analog_two),
                .channel(channel),
                .restart(restart),
                .register_out(register_out),
                .peripheralClkEdge(peripheralClkEdge),
                .parallelLoad(parallelLoad),
                .serialDataIn(serialDataIn),
                .serialDataOut(serialDataOut)
                );
initial begin
                begintest = 0;
                #10;
                begintest = 1;
                #1000;
end
always @(posedge endtest) begin
                $display("DUT passed?: %b", dutpassed);
end
endmodule
```

```
module SampleHoldTestBench(
        input begintest,
        output reg endtest,
        output reg dutpassed,
        output reg clk,
        output reg [9:0] analog_one,
        output reg [9:0] analog_two,
        output reg channel,
        output reg restart,
        input [9:0] register_out,
        output reg peripheralClkEdge,
        output reg parallelLoad,
        output reg serialDataIn,
                parallelDataOut,
                input serialDataOut
        );
  // Generate clock (50MHz)
initial clk=0;
always #10 clk=!clk;    // 50MHz Clock
  always @(posedge begintest) begin
        //testing values
        endtest = 0;
        dutpassed =1;
                peripheralClkEdge = 0;
        serialDataIn = 0;
        parallelLoad = 0;
        channel = 0;
        restart = 1;
        analog_one[9] = 1;
        analog_one[8] = 1;
        analog_one[7] = 1;
        analog_one[6] = 1;
        analog_one[5] = 1;
        analog_one[4] = 1;
        analog_one[3] = 1;
        analog_one[2] = 1;
        analog_one[1] = 1;
        analog_one[0] = 1;
        analog_two[9] = 0;
        analog_two[8] = 0;
        analog_two[7] = 0;
        analog_two[6] = 0;
        analog_two[5] = 0;
        analog_two[4] = 0;
        analog_two[3] = 0;
        analog_two[2] = 0;
        analog_two[1] = 0;
        analog_two[0] = 0;
        #40
        restart = 0;
        #200
        parallelLoad = 1;
        #20
        parallelLoad = 0;
        peripheralClkEdge = 1;
        #100
        analog_one[9] = 0;
        analog_one[8] = 0;
        analog_one[7] = 0;
        analog_one[6] = 0;
        analog_one[5] = 0;
        analog_one[4] = 0;
        analog_one[3] = 0;
        analog_one[2] = 0;
        analog_one[1] = 0;
        analog_one[0] = 0;
        analog_two[9] = 1;
        analog_two[8] = 1;
        analog_two[7] = 1;
        analog_two[6] = 1;
        analog_two[5] = 1;
        analog_two[4] = 1;
        analog_two[3] = 1;
        analog_two[2] = 1;
        analog_two[1] = 1;
        analog_two[0] = 1;
        #200
        peripheralClkEdge = 0;
        parallelLoad = 1;
        #20
        parallelLoad = 0;
        peripheralClkEdge = 1;
        #100
        analog_one[9] = 1;
        analog_one[8] = 0;
        analog_one[7] = 1;
        analog_one[6] = 0;
        analog_one[5] = 1;
        analog_one[4] = 0;
        analog_one[3] = 1;
        analog_one[2] = 0;
        analog_one[1] = 1;
        analog_one[0] = 0;
        analog_two[9] = 1;
        analog_two[8] = 0;
        analog_two[7] = 1;
        analog_two[6] = 0;
        analog_two[5] = 1;
        analog_two[4] = 0;
        analog_two[3] = 1;
        analog_two[2] = 0;
        analog_two[1] = 1;
        analog_two[0] = 0;
        #200
        parallelLoad = 1;
        peripheralClkEdge = 0;
        #20
        parallelLoad = 0;
        peripheralClkEdge = 1;
        #100
        channel = 1;
        analog_one[9] = 1;
        analog_one[8] = 1;
        analog_one[7] = 1;
        analog_one[6] = 1;
        analog_one[5] = 1;
        analog_one[4] = 1;
        analog_one[3] = 1;
        analog_one[2] = 1;
        analog_one[1] = 1;
        analog_one[0] = 1;
        analog_two[9] = 0;
        analog_two[8] = 0;
        analog_two[7] = 0;
        analog_two[6] = 0;
        analog_two[5] = 0;
        analog_two[4] = 0;
        analog_two[3] = 0;
        analog_two[2] = 0;
        analog_two[1] = 0;
        analog_two[0] = 0;
        #200
        parallelLoad = 1;
        #20
        parallelLoad = 0;
        #100
        analog_one[9] = 0;
        analog_one[8] = 0;
        analog_one[7] = 0;
        analog_one[6] = 0;
        analog_one[5] = 0;
        analog_one[4] = 0;


        analog_one[3] = 0;
        analog_one[2] = 0;
        analog_one[1] = 0;
        analog_one[0] = 0;
        analog_two[9] = 1;
        analog_two[8] = 1;
        analog_two[7] = 1;
        analog_two[6] = 1;
        analog_two[5] = 1;
        analog_two[4] = 1;
        analog_two[3] = 1;
        analog_two[2] = 1;
        analog_two[1] = 1;
        analog_two[0] = 1;
        #200
        parallelLoad = 1;
        peripheralClkEdge = 0;
        #20
        parallelLoad = 0;
        peripheralClkEdge = 1;
        #100
        analog_one[9] = 1;
        analog_one[8] = 0;
        analog_one[7] = 1;
        analog_one[6] = 0;
        analog_one[5] = 1;
        analog_one[4] = 0;
        analog_one[3] = 1;
        analog_one[2] = 0;
        analog_one[1] = 1;
        analog_one[0] = 0;
        analog_two[9] = 1;
        analog_two[8] = 0;
        analog_two[7] = 1;
        analog_two[6] = 0;
        analog_two[5] = 1;
        analog_two[4] = 0;
        analog_two[3] = 1;
        analog_two[2] = 0;
        analog_two[1] = 1;
        analog_two[0] = 0;
        #200
        parallelLoad = 1;
        peripheralClkEdge = 0;
        #20
        parallelLoad = 0;
        peripheralClkEdge = 1;
        #100
        endtest = 1;
    end
endmodule
```

Appendix C: Work Plan

Task	Time Estimate	Date Finished Estimate	Time Actual	Date Finished Actual
Basic Understanding				
Successive Approx.	1 hour	December 7th	1 hour	December 7
Flash	1 hour	December 7th	1 hour	December 7
~~Pipeline~~	~~1 hour~~	~~December 7th~~		
Comparison				
Think about how to quantify and qualify characteristics for comparison	2 hours	December 7th	15 min	December 14
Find & clarify comparison data				
Successive Approx.	2 hours	December 7th	30 min	December 15
Flash	2 hours	December 10th	30 min	December 15
~~Pipeline~~	~~2 hours~~	~~stretch~~		
Verilog implementation				
Black box diagrams-inputs/outputs				
Successive Approximation	15 min	December 7th	2 hours	December 7
Flash	15 min	December 10th	1 hour	December 11
~~Pipeline~~	~~15 min~~	~~December 10th~~		
Successive Approximation				
S.A. Register	1 hour	December 10th	2 hour	December 12
DAC	1 hour	December 10th	3 hours	December 12
Sample and Hold Circuit	1 hour	December 10th	30 min	December 13
Comparator	1 hour	December 10th	30 min	December 13
~~x10?~~	~~1 hour~~	~~December 14th~~		
Integration	NEW	NEW	3 hours	December 14
Mux	NEW	NEW	30 min	December 14
Shift Register	NEW	NEW	30 min	December 14
Test documentation	NEW	NEW	3 hours	December 15
~~Flash~~				
~~Analog Ref~~	~~15 min~~			
~~PreAmp~~	~~1 hour~~			
~~Comparator~~	~~-done-~~			
~~Decoder (Errors)~~	~~1 hour~~			
~~De-Multiplexer~~	~~1 hour~~			
~~x8?~~	~~3 hours~~			
Write Up				
Write up what I did	NEW	NEW	1 hour	December 15
Write up why I did	NEW	NEW	1 hour	December 15
Write up Verilog method	NEW	NEW	1 hour	December 15
Poster and Presentation Prep	NEW	NEW		December 15

Appendix D: Resources

On ADCs in general: https://en.wikipedia.org/wiki/Analog-to-digital_converter
Datasheets:

- Successive Approximation: http://ww1.microchip.com/downloads/en/DeviceDoc/21294C.pdf
- Flash: http://pdf.datasheetcatalog.com/datasheet/maxim/MAX1150.pdf
- ~~Pipeline Options:~~
  - ~~http://www.analog.com/media/en/technical-documentation/data-sheets/AD9254.pdf~~
  - ~~https://datasheets.maximintegrated.com/en/ds/MAX1205.pdf~~
  - ~~http://www.analog.com/media/en/technical-documentation/data-sheets/AD9461.pdf~~

Info on conversion types:

- Successive Approximation ADC: https://en.wikipedia.org/wiki/Successive_approximation_ADC
- Flash ADC: https://en.wikipedia.org/wiki/Flash_ADC
- ~~Pipelined ADC: https://www.maximintegrated.com/en/app-notes/index.mvp/id/1023~~