Project 4
PUF Implementation
Hardware Security

Ivan Jimenez

November 22, 2012

1 Introduction

The purpose of this project is to implement a Physical Unclonnable Function (PUF) device in an FPGA and test its performance. The implementation of the PUF logic is done using combinatorial logic.

My goal was to use the PUF as an authentication device. For this, I need to create a Challenge-Response table. Originally, I was going to use USB communications to interface the PUF with a computer that would provide the challenges and receive the responses from the PUF in order to perform authentication. However, due to dificulties with the FPGA software, I could not get to complete this implementation. Leaving me with a simple implementation of the PUF.

2 The Design

My design target was to create a 8-bit to 8-bit PUF. By this I mean a PUF that takes an 8-bit challenge and produces an 8-bit response. This PUF was implemented in a Digilent Basys2 board, using a Xilinx Spartan-3E FPGA. This FPGA uses look-up tables to implement logic gates and flip flops for the implementation.

Since the PUF was to be implemented on an FPGA, I used a Ring Oscilator-based (RO) PUF. This design can be somewhat simple given the structure of the ROs of several NOT gates connected serially. In this implementation, I used the challenge bits to select two of 16 ROs available in the circuit. To select the oscillators, I split the challenge in two 4-bit parts, each used as the controller signals for two 16:1 MUXes. Both MUXes are connected to the same ROs but in different order. This allows for some randomness in the implementation in order to diminish the ability to characterize the device. The oscillating outputs of the MUXes are fed to two 12-bit counters as their clock signals. As soon as one of the counters overflows, the counting is stopped and one bit of the output of the PUF is calculated. For this calculation. some use a comparator. However, since I am using the counter overflows, I chose to compare which counter caused the overflow. In my implementation, the PUF will produce an output of 1 if the counter to overflow first corresponds to the RO chosen by the lower four bits of the challenge. Figure 1 shows the schematic of this 8-to-1 bit PUF implementation.

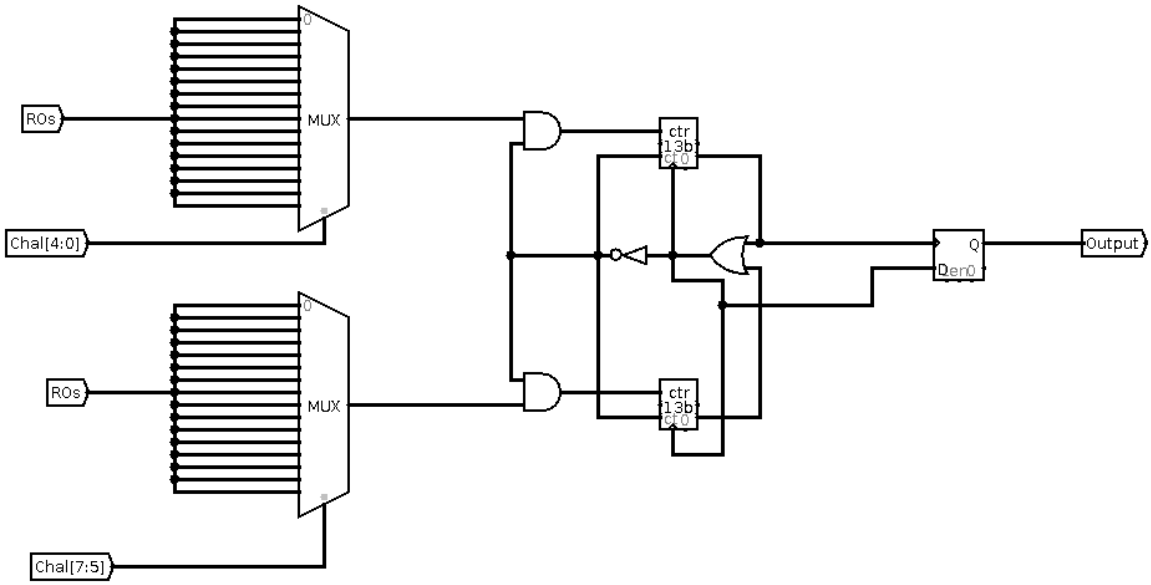

Figure 1: 8-bit to 1-bit RO-based PUF

This implementation, though simple, is insufficient for my purposes because it creates only one bit of the ouput I want. In order to create the other seven bits, I could create several instantiations of this system, or use several challenges to construct them. Because this simple implementation seemed to occupy most of the space in the FPGA, I chose a variant of the later option.

My plan was to create seven more "pseudo-challenges" besides the requested challenge by swapping the bits. Using this approach requires the use of memory to hold the outputs of the PUF for every challenge but requires less logic gates than using more MUXes or using several copies of the first system. To implement it, I tried to use a system of flip flops so as to avoid using much logic and making the program to big for the FPGA. Figure 2 shows how the different pseudo-challenges are created and stored in flip flops, and Figure 3 shows the flip flop system I used to store the response bits. The overall system is controlled with logic that controls which pseudo-challenge is used to create the output bit.

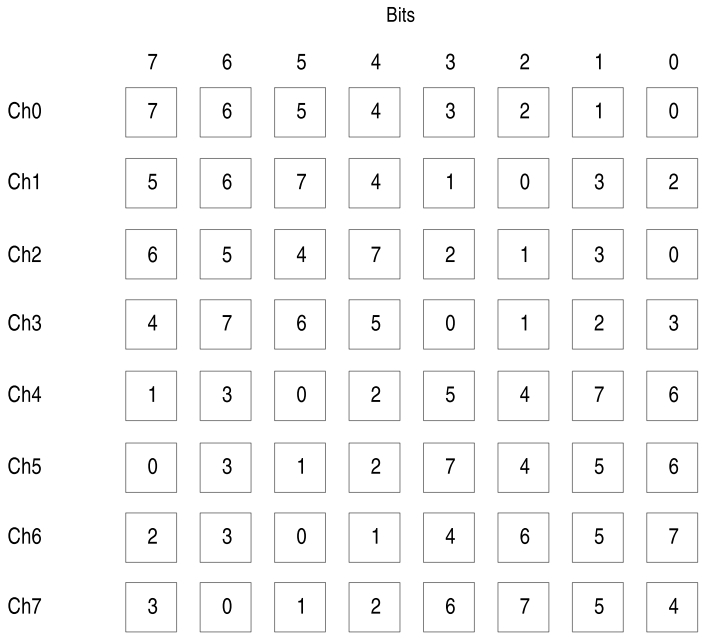

Figure 2: Diagram of how the pseudo-challenges are created

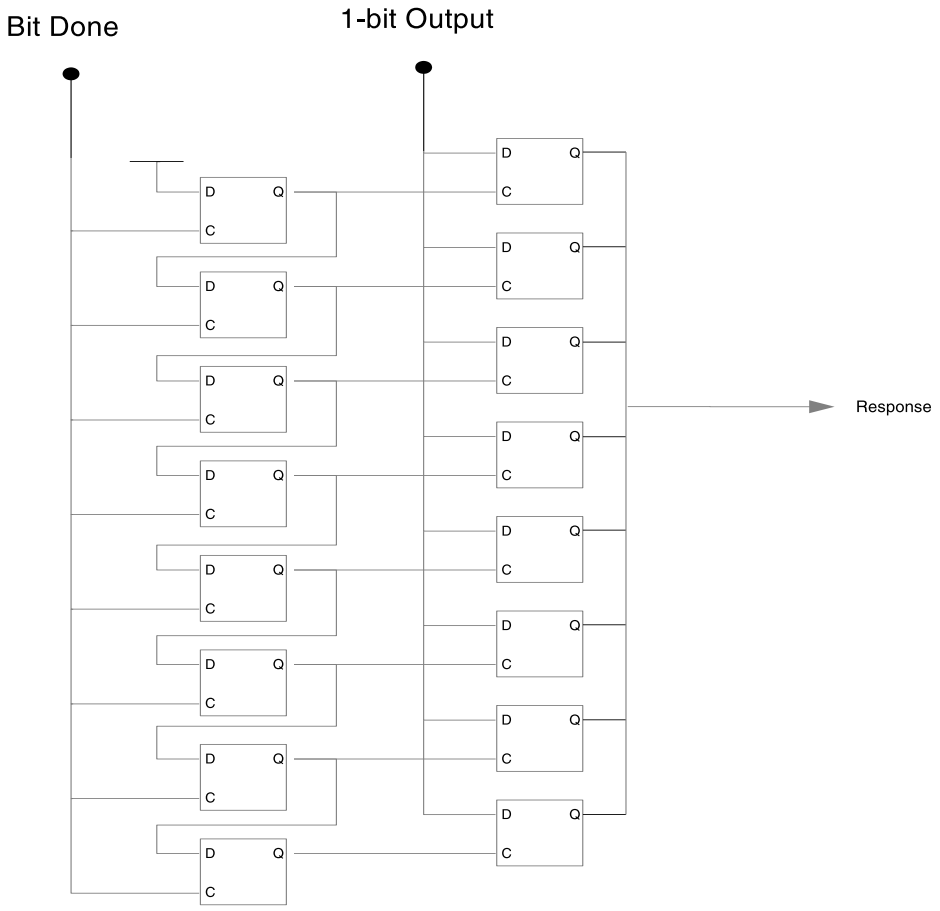

Figure 3: Flip Flop system to store the PUF output

3 The implementation

I used Verilog for the implementation of the PUF, which gave me some control over what elements could be optimized away. I say only "some" control because Xilinx would still try to get rid of any logic it deemed unnecessary. This little detail was the biggest hurdle of implementing this design. Here I show what I used to implement each element. In the code it can be seen how I rely on the delays caused by the gates and elements to create some of the control signals. For example, I use the signal coming out of the OR gate comparing the overflows coming out of the counters (as seen in Figure 1) to control the counter reset and the output register load. This delay is important because the output result might never reach the register if it is not long enough.

3.1 Ring Oscillators

The ring oscillator consists of 75 NOT gates and a NAND gate that implements the feedback and enable.

```
module oscillator(
input enable,
output wave
    );
```

```
nand g1(n0,enable,n74);
not b1(n1,n0);
not b2(n2,n1);
not b3(n3,n2);
not b4(n4,n3);
not b5(n5,n4);
not b6(n6,n5);
not b7(n7,n6);
not b8(n8,n7);
not b9(n9,n8);
not b10(n10,n9);
not b11(n11,n10);
......
not b74(n74,n73);
not bf(wave,n74);
end module
```

With this number of gates, the Post-Route simulation showed that the ROs were producing a square wave with a period of around 190ns.

3.2 Multiplexers

To create the 16-to-1 multiplexer, I used 4-to-1 multiplexers that I implemented combinatorially.

```
module mux4(
   input d0,
   input d1,
   input d2,
   input d3,
   input s0,
   input s1,
   output M
);
wire s1_n, s0_n, t0,t1,t2,t3;
not n0(s0_n,s0);
not n1(s1_n,s1);
and g0(t0,s0_n,s1_n,d0);
and
g1(t1,s0,s1_n,d1);
and g2(t2,s0_n,s1,d2);
and g3(t3,s0,s1,d3);
or g4(M,t0,t1,t2,t3);
endmodule
```

This module was instantiated in the module for the 16-to-1 MUX

```
module mux16( d0, d1, d2, d3, d4, d5, d6, d7, d8, d9, d10, d11, d12, d13, d14, d15, M,
s0, s1, s2, s3
    );
input d0, d1, d2, d3, d4, d5, d6, d7, d8, d9, d10, d11, d12, d13, d14, d15;
input s0, s1, s2, s3;
output M;
```

```
wire t0,t1,t2,t3;
mux4 m0(.d0(d0), .d1(d1), .d2(d2), .d3(d3), .s1(s1), .s0(s0), .M(t0));
mux4 m1(.d0(d4), .d1(d5), .d2(d6), .d3(d7), .s1(s1), .s0(s0), .M(t1));
mux4 m2(.d0(d8), .d1(d9), .d2(d10), .d3(d11), .s1(s1), .s0(s0), .M(t2));
mux4 m3(.d0(d12), .d1(d13), .d2(d14), .d3(d15), .s1(s1), .s0(s0), .M(t3));
mux4 m4(.d0(t0), .d1(t1), .d2(t2), .d3(t3), .s1(s3), .s0(s2), .M(M));
endmodule
```

3.3 Counters

To implement the counters, I used those available from the Xilinx library since they do not affect the delay path of the oscillations, and are better optimized by the Xilinx software. I used 12-bit counters by cascading a 4-bit and an 8-bit counter. The schematic for the counter is shown in Figure 4.

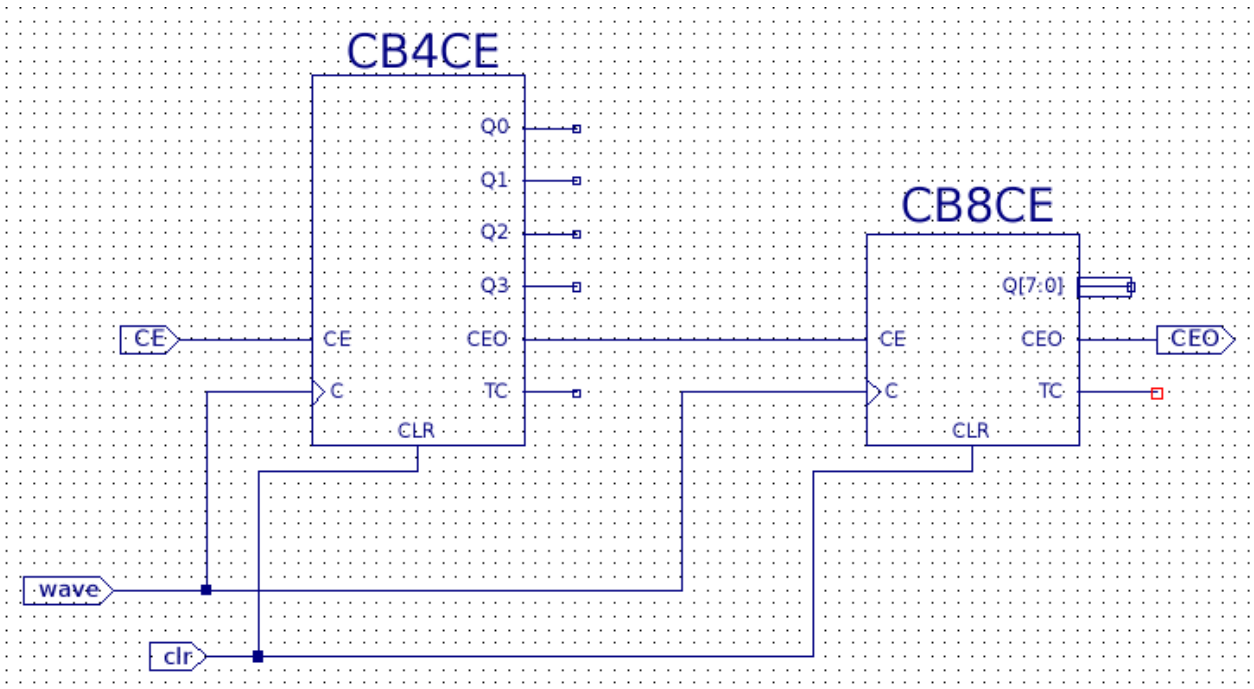

Figure 4: 12-bit counter

The signal CEO of the counter is used as the overflow signal to trigger the output logic.

3.4 Pseudo-challenges

The pseudo-challenges are just flip flops grabbing each specific bit.

```
// Load the challenge
always @(posedge enable)
begin
Ch0 = challenge;
Ch1[7] = challenge[5];
Ch1[6] = challenge[6];
Ch1[5] = challenge[7];
```

```
Ch1[4] = challenge[4];
Ch1[3] = challenge[1];
Ch1[2] = challenge[0];
Ch1[1] = challenge[3];
Ch1[0] = challenge[2];
    Ch2[7] = challenge[6];
Ch2[6] = challenge[5];
Ch2[5] = challenge[4];
Ch2[4] = challenge[7];
Ch2[3] = challenge[2];
Ch2[2] = challenge[1];
Ch2[1] = challenge[3];
Ch2[0] = challenge[0];

    .....
end
```

Xilinx seemed to get rid of some of the flip flops during the optimization, but it all seemed to work during simulations.

3.5 Response construction

I implemented this with a chain of flip flops, as designed, and it seemed to work during simulation. However, the transition between states had some random numbers for small amounts of time.

```
// Direct the result to the output
(* KEEP *) FDC cont0(.D(1), .C(done), .CLR(~enable), .Q(resClk[0]));
(* KEEP *) FDC cont1(.D(resClk[0]), .C(done), .CLR(~enable), .Q(resClk[1]));
(* KEEP *) FDC cont2(.D(resClk[1]), .C(done), .CLR(~enable), .Q(resClk[2]));
(* KEEP *) FDC cont3(.D(resClk[2]), .C(done), .CLR(~enable), .Q(resClk[3]));
(* KEEP *) FDC cont4(.D(resClk[3]), .C(done), .CLR(~enable), .Q(resClk[4]));
(* KEEP *) FDC cont5(.D(resClk[4]), .C(done), .CLR(~enable), .Q(resClk[5]));
(* KEEP *) FDC cont6(.D(resClk[5]), .C(done), .CLR(~enable), .Q(resClk[6]));
(* KEEP *) FDC cont7(.D(resClk[6]), .C(done), .CLR(~enable), .Q(resClk[7]));
// Flip Flops to store the outputs
(* KEEP *) FDC res0(.D(res), .C(resClk[0]), .CLR(~enable), .Q(resTemp[0]));
(* KEEP *) FDC res1(.D(res), .C(resClk[1]), .CLR(~enable), .Q(resTemp[1]));
(* KEEP *) FDC res2(.D(res), .C(resClk[2]), .CLR(~enable), .Q(resTemp[2]));
(* KEEP *) FDC res3(.D(res), .C(resClk[3]), .CLR(~enable), .Q(resTemp[3]));
(* KEEP *) FDC res4(.D(res), .C(resClk[4]), .CLR(~enable), .Q(resTemp[4]));
(* KEEP *) FDC res5(.D(res), .C(resClk[5]), .CLR(~enable), .Q(resTemp[5]));
(* KEEP *) FDC res6(.D(res), .C(resClk[6]), .CLR(~enable), .Q(resTemp[6]));
(* KEEP *) FDC res7(.D(res), .C(resClk[7]), .CLR(~enable), .Q(resTemp[7]));
```

The *contN* provides the enabling signal *resClk* to the storage flip flops *resN*, which load the 1-bit output of the N round of the PUF.

3.6 Control

The control logic was implemented using behavioral code so as to allow Xilinx to process it and optimize accordingly since it was becoming a bit of a challenge for me to implement it withouth breaking the PUF implementation.

```
// Control
always @(resClk) // Use this signal to change the states
begin
case (resClk)
8'b00000000: begin
select = Ch0;
// next = 1;
// #5
// next = 0;
end
8'b00000001: begin
select = Ch1;
// next = 1;
// #5
// next = 0;
end
8'b00000011: begin
select = Ch2;
// next = 1;
// #5
// next = 0;
end
8'b00000111: begin
select = Ch3;
// next = 1;
// #5
// next = 0;
end
8'b00001111: begin
select = Ch4;
// next = 1;
// #5
// next = 0;
end
8'b00011111: begin
select = Ch5;
// next = 1;
// #5
// next = 0;
end
8'b00111111: begin
select = Ch6;
// next = 1;
// #5
// next = 0;
end
8'b01111111: begin
select = Ch7;
// next = 1;
// #5
// next = 0;
end
endcase
```

In this code we can see an attempt to use the signal *next* to wait during the clearing of some of the flip flops, but the lack of a gobal clock prohibited from implementing it.

3.7 PUF

The PUF implementation instantiates all the elementes needed to produce the 1-bit outputs of every challenge, and the logic to create the other 7 bits.

```
module puf( challenge, response, enable, ready
    );

input [7:0] challenge;
input enable;
output [7:0] response;
output ready;
// Control signals
reg [7:0] select;
wire CE;  // Control the counters
wire FCE;  // Control final register load
wire done;  // Signal that we obtained a result and stop the counters
reg next = 0; // Reserved to clear a flip flop for 'done'
```

It also implements the two MUXes and the RO sorting.

```
// Use challenge to select MUXes outputs
mux16 m0(.d0(w[0]), .d1(w[1]), .d2(w[2]), .d3(w[3]), .d4(w[4]), .d5(w[5]), .d6(w[6]), .d7(w[7]),
.d8(w[8]), .d9(w[9]), .d10(w[10]), .d11(w[11]), .d12(w[12]), .d13(w[13]), .d14(w[14]), .d15(w[15]),
.s3(select[3]), .s2(select[2]), .s1(select[1]), .s0(select[0]), .M(C0));
mux16 m1(.d0(w[0]), .d1(w[2]), .d2(w[1]), .d3(w[12]), .d4(w[7]), .d5(w[5]), .d6(w[4]), .d7(w[6]),
.d8(w[9]), .d9(w[14]), .d10(w[8]), .d11(w[15]), .d12(w[3]), .d13(w[13]), .d14(w[10]), .d15(w[11]),
.s3(select[7]), .s2(select[6]), .s1(select[5]), .s0(select[4]), .M(C1));
```

We can see how the second MUX has a somewhat random distribution of the RO inputs.

The PUF module uses the *enable* signal to clear all its flip flops and the *ready* signal to let the top module know that all eight bits of the *response* output are populated.The complete system diagram is shown in Figure 5.

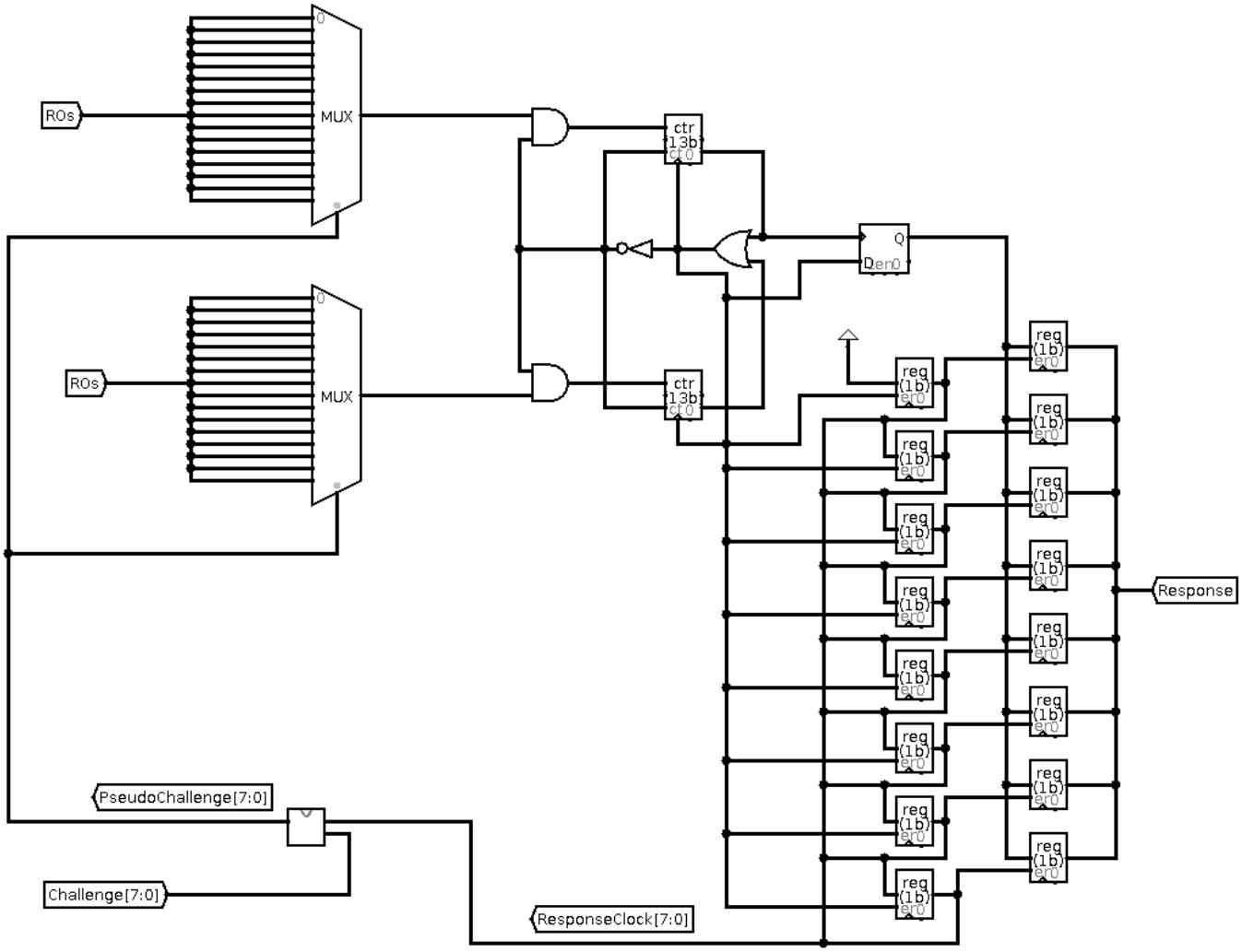

Figure 5: Complete 8-bit to 8-bit RO PUF implementation

4 Results

My simple 8:1 PUF performs quite nicely. It is able to output the response bit and includes a flip flop that holds the bit for the controller to load on the registers. Running a Post-Route simulation allows us to see the PUF operating as it produces the output bit. It certainly takes quite some time to create the ouput because the period of the oscillators is big. However, the control logic internal to the PUF works very well. Figure 6 shows a simulation of this simple PUF; it shows the counters being updated and the how it stops counting once the ouput is calculated. I was also able to program the Basys2 board with this PUF and see it work.

The simulations shows the signal *FCE* (coming out of the OR gate comparing *comp1* and *comp2*) appearing after the overflow and triggering the transitions. In this case, the challenge 9D caused the output to be a zero, however, we can see the signal *doneLED* turn on to mark the end of the operation. At this point, all it takes to create another output is change the challenge and restart by resetting *BTN*.

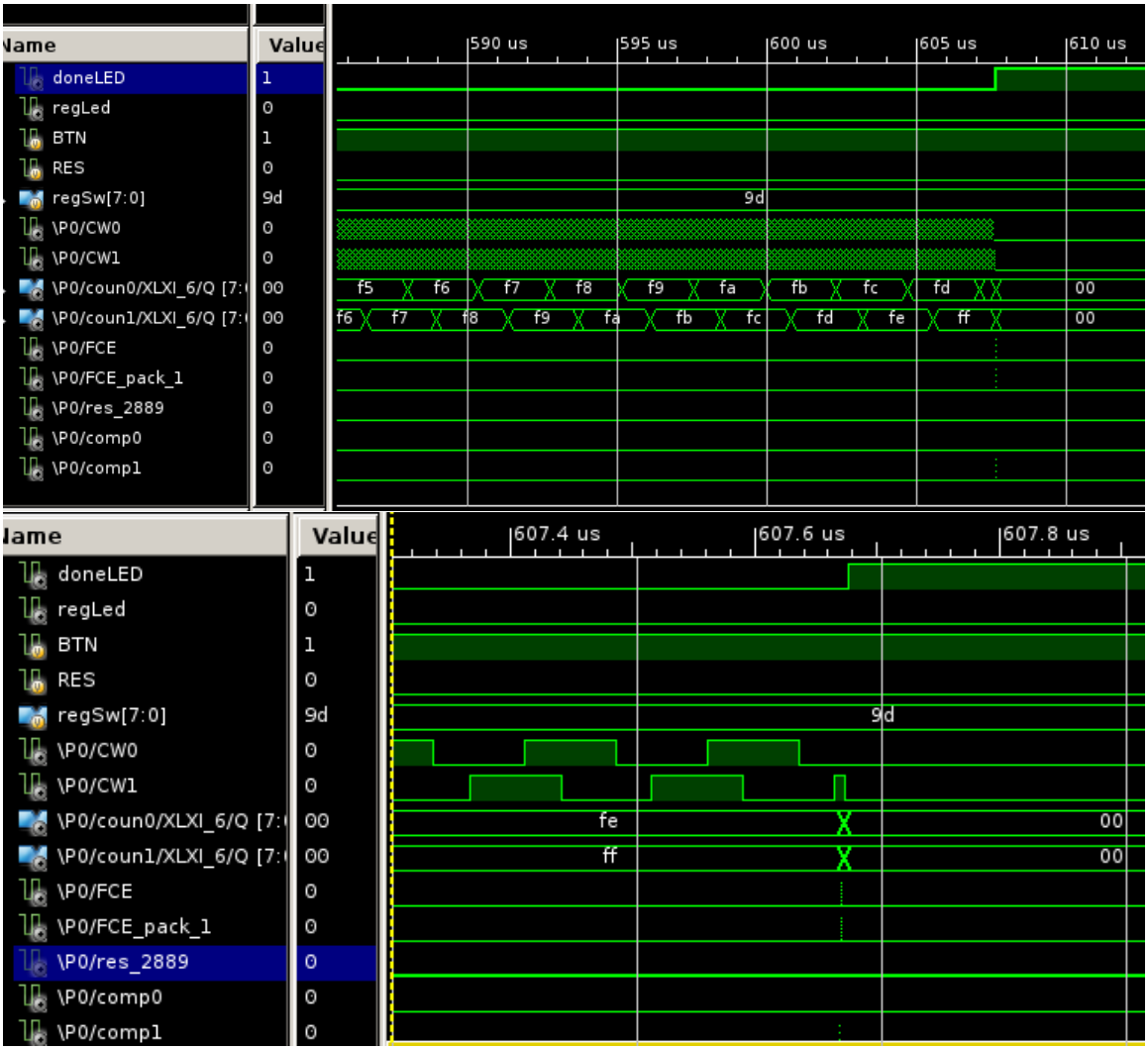

Figure 6: Simulation of the 8:1 PUF

However, the most difficult part was to insert the logic to create the 8-bit response. Including this logic and the flip flops was not a big deal, but the fact that I was lacking a clock made the control a little difficult. I had to find a way to control the clearing of the registers and signals in a non-blocking way in order for the challenge to be changed while the counters are cleared and ready to start again. I had problems with this and using a signal as quick as the *FCE* resulted in a system very sensitive to the transitions. As a result, I my counters transitions would trigger the controllers before the overflow. However, the control logic was working as expected. This can be seen in Figures 7 and 8.

In Figure 7 we can see how the control logic signals that one output bit has been created and the computation of the next one starts. We can clearly see how the new pseudo-challenge causes a different oscillator to be used. Figure 8 shows a closer look to the transitions. The transitions are successfully achieved between one state and another. For example, the counters are at F9 and EF initially; the EF counter goes through a quick transition where it is FF and causes its overflow signal to pop up (seen on top of F0), which triggers the *FCE* signal (on top of the F9 and 80) to try to stop the oscillations (interruption in the signal below the 9D) and to reset the counters (both ater back to 00). At the same time, the signal *ResponseCLK* (bottom) changes from 00 to 01 meaning that the first output register has been loaded (with a zero in this case). This later causes the pseudo-challenge to change from 9D to 37, which changes the ring oscillator choice (short 1 on time *576us*). In longer simulations I could see that the system successfully iterated through the different pseudo-challenges, but the faulty counters always caused an ouput of zero.

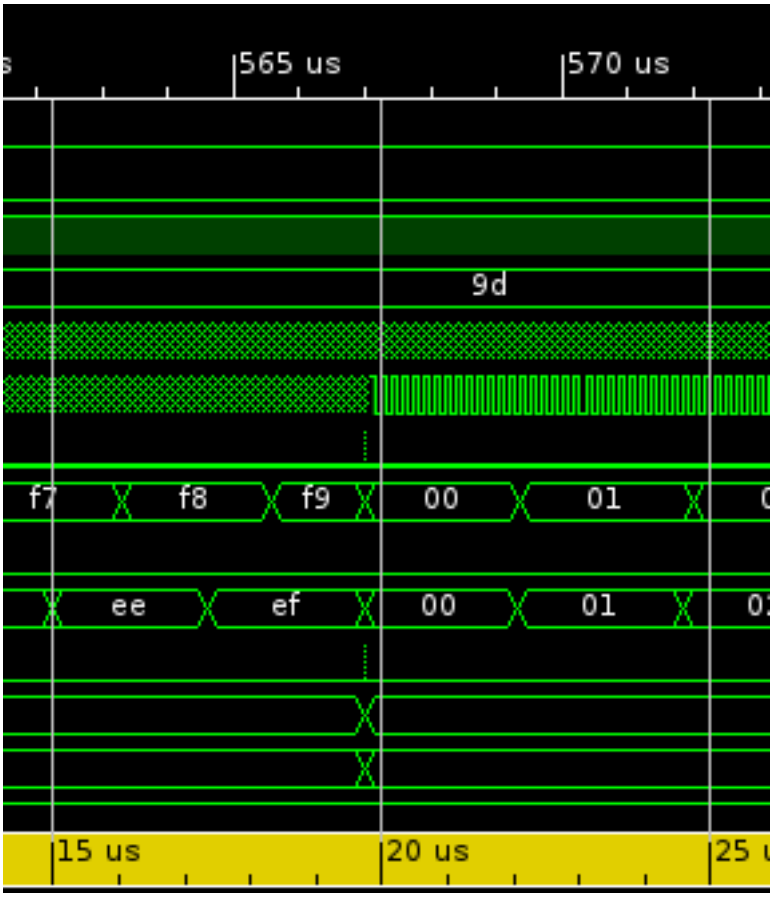

Figure 7: Control logic in action during a run of the 8:8 PUF

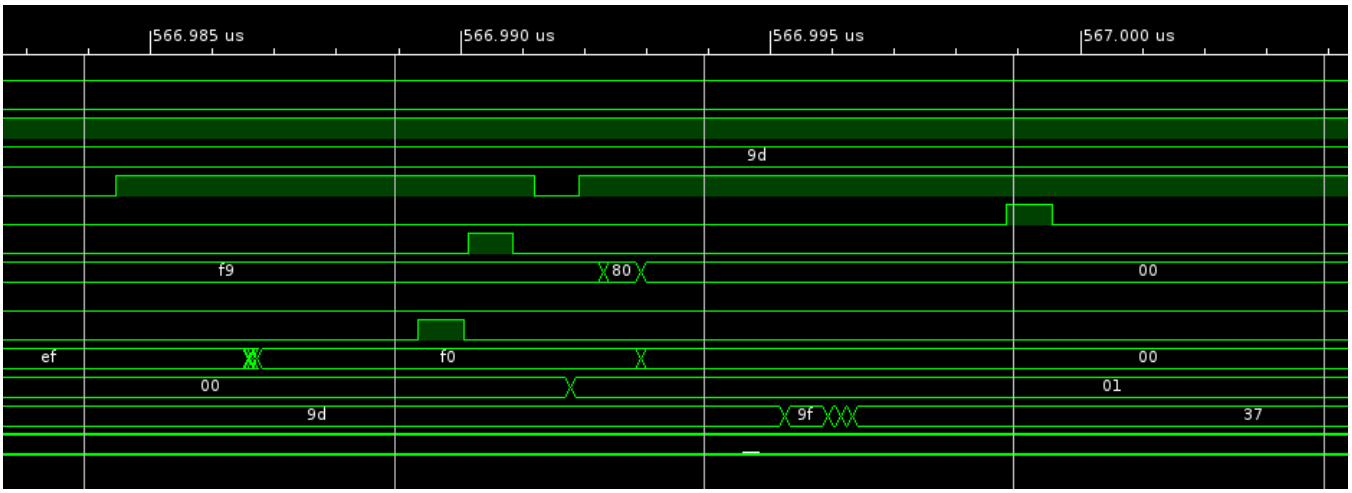

Figure 8: A closer look to the 8:8 transitions

These figures show that my logic is working correctly. Although I need to fix the timing issues with some of the control signals in order to ensure that the counters work correctly. In all my simulations of this system, the bottom counter would have a transition that causes the *FCE* signal to go HIGH for a short time and trigger the controller. My observations led me to believe that the Xilinx software made compiled the counters in a way that makes them more sensitive (I did

not have this problem running the 8:1 PUF). In this example, the faulty transition occured in the counter that triggers a response of zero.

5 Conclusions

The design of this project was not very complicated, but the implementation was a big challenge given the fact that the FPGA was running out of space quickly, and that the Xilinx software was difficult to work with. I decided to base my 8:1 PUF in logic gates and use as many flip flops as possible for the controller because of the limited space. It took me a long time to get the simple PUF to work both in simulations and the board. By the time I had this working, I decided to try and implement the controller so as to be able to eventually use the USB interface to create the challenge-response tables. However, the Xilinx software was very sensitive to any change in the code and I lost much time trying to test the controller logic. After this much time, I ended up implementing the system described in this report. I was able to program the board with it but since the result was always zero every time the *READY* signal triggered, I could not continue forth.

One thing I could try is to use the 8:1 PUF as implemented and bring the control logic to an upper level in which the counters and flip flops of the PUF are controlled by a single input of the PUF. Using this approach I should be able to have more flexibility and less sensitivity of the internal modules. Hopefully I will have some time to rework this. However, for my satisfaction, I was able to record a video of the 8:1 PUF working on the board and reacting to the different 8-bit challenges created using the board switches.