Lab Final: Crypto Cracking with Hardware

Report written by:
Micah McCombs: A20138748
Mike Arledge: A20317028

• Section 1: Introduction

After much work, we have arrived at the final lab for the course; and it doesn't disappoint. A few weeks ago, we worked on a Data Encryption Standard project (DES), which involved implementing DES with hardware descriptive language, such that the encryption would be running on a hardware level as opposed to software. In that project, we developed an exact copy of the DES algorithm, which is symmetric by the way, and could encrypt and decrypt a text using a key of a specific length; although it not need be a text, it could be any arbitrary data so long as it is expanded, formatted, and interpreted correctly. More or less, DES is a nice tool to have laying around, and is capable of quite a lot; even though we concluded in that report its various shortcomings. DES has been broken before, and stands replaced by more modern encryption algorithms like AES and Public key encryption methods for the ever growing wireless and digital era. However, it is in this lab that we will use the very hardware that enables our old security standard, to crack into it.

In the lab we will create a cracking device that will take a plaintext and ciphertext as its input, and will be responsible for finding the respective key that was used for encryption. The specific method of attack will be none other than brute force. As you may already know, before a parity check reduction, there are 2^64 possible combinations of keys to check, and even with parallelization and significant computing power, that is still an enormous number to try and tackle. For demonstration purposes, our key size will be much smaller and manageable.

• Section 2: Baseline Design

In our baseline design following figure 3, we started with the up input into the udl counter with an output of 56 bits. The counter in figure 4 takes an input of the clk, reset (rst), up, down, load, and 56 bits of input. The clk synchronizes the logic, the reset setting to the default 64 bits of 0 whereas load setting the output as the 56 bit input logic. The up and down acts as the actual counting for the counter module, decreasing or increasing the output by 1 bit. The baseline design looking at figure 3 and 5 also includes a basic flip flop which can be used for the registers later. This flip flop includes inputs clk, acting on the positive edge, and 2 bits d and an output of q with 2 bits. The Flip flop also takes in 8 bits in the parameter that we split from the inputs we use in logic we later talk about. After the output of 56 bits from the UDL counter, we take that into our parity generation in order to generate the correct number of bits for the key. This parity generation can be seen in figure 6 but will be talked about in section 3 since it is part of our design. The 64 bit counter with parity is placed into the DES in order to create the correct key per plain and ciphertext.

The DES, which is displayed in figure 7 and figure 3, takes in the 64 bit key which with the plaintext we can repeat these steps until we match the correct cipher text which is described more in section 3. The only other thing we would need are our registers which described in figure 8 take an input of 8 bits for the width parameter which can be changed later for larger inputs, described in section 3, is used for the fsm and acts like a reset and enable. The registers store information provided in the datapath, for which we store 64 bit inputs for which we use depending if our cipher text equals the generated cipher text along with the key. For when we store our key or ciphertext follows the fsm design described in figure 2. The input on the fsm called start basically works in the idle state which also is the state the fsm resets to. As soon as

the start is high the current state is moved to the next and begins searching for the key that matches with the ciphertext that is already given. The next state which will always occur no matter the input stores the ciphertext and key that the current state carries in and repeats the comparison function until the key is found. Once the key is found, the final state breaks the fsm from continuing to search and starts again if the start is 0. With all of this in mind we can bring this into our own design to take the width of 64 bits.

• Section 3: Design

Going into the design we start with our udl counter with input up which can be seen in figure 3. In order to generate the up when we need to versus when not to we need to use our fsm logic which can be seen in figure 9 and figure 10 with the module control. In order to determine up and other logic, we use our fsm from figure 3 to implementation in figure 10 with clock synchronization. inputs start, keyfound, reset and outputs en1, en2, and up, all creating the chain of states that we use for the repetition of our logic. With our fsm logic described earlier we use the control module to generate up, en1 and en2 in figure 9 which we directly use up in the next line with our counter. The counter we create needs to be 64 bits with the parity in order to line up the current ciphertext length, so we place the up logic into the UDL counter as seen in figure 10 and figure 11. With the 56 bit parameter we use our input logic clock, reset, up, load, down and in, while we use our output count. Load and down are immediately replaced with 0's and our 56 bit input is just the default 56 bit defaulted to just 0's, 56'h000_000_000_000_00. When our fsm logic starts up from the idle state the input start is set to high, increasing the input by 1 bit per cycle. The output count is then placed into a parity generator where we get the correct number of bits and parity, which can be described in figure 12. Instead of calling the baseline design for parity generation 8 times, we would create a separate but equal parity generator with 56 bit input

and 64 bit output with parity instead of the 7 bit input and 8 bit output. This is instantiated once in figure 9 with count as our 56 bit input from the UDL counter and counter_64 being the 64 bit output with parity. The code from figure 12 uses a for loop to contain the 7 and 8 bit input calling the original parity generator 8 times, creating the output. From figure 9, with our 64 bit counter with parity that is increasing each clock edge which acts as the key, and the plain text given to us, we can instantiate our DES module to create our ciphertext test vectors. The DES described in figure 7, 9 and 13 creates the 64 bit ciphertext which we use as temptext. The temptext output then is placed in our first register storing the ciphertext each cycle, which is described in figure 14. In each cycle in order for us to compare the ciphertext we instantiate a comparator c which only compares the temptext to the ciphertext we generate with the key and our plaintext seen in figure 9. The comparator takes in the assigned ciphertext and the q1 which was the output of the register and compares them to output a high based on if they are both equal and a 0 if not as described in figure 15. The fsm from figure 9 is constantly cycling through different iterations comparing ciphertexts and q1 with the comparator until keyFound is high. The keys that the fsm cycles through from the counter are placed in storage in the key_register from figure 9. When the assigned cipher text is equal to q1 the next state is finally reached in figure 3 which concludes the brute force testing and outputs the key to a count.out file in figure 16. For our implementation we used a mux to display our key once it was found and used switches to cycle between 4 bits of the key, reset and start (Figure 17). We used binary encoding to switch between each 4 as seen in the video provided in the files.

• Section 4: Testing Strategy

In the presence of brute force, we are left with no choice but to design a testing strategy that guarantees a successful run. Unlike our other tests, we expect the running time to be

enormous. And also unlike other tests, we can't be entirely sure how long it will take to test the device since we don't know how long it will run in general. We can't afford to let loose a design that is incorrect, run for an hour or more just to have no hopes of ever returning the correct value. Instead we must come up with a way to manage our time efficiently and conduct smaller tests in an almost divide and conquer method. What is similar to before, is that we plan to use a combination of simulation first and device testing on hardware later; when we are absolutely certain it is correct.

However, it will be very inefficient to run the full length simulation everytime we wish to troubleshoot and debug. That is why we prebuilt a custom key that was only a few iterations from the start, such that the ciphertext would be found almost instantly. That way our testing would only last a few seconds, and kept the project manageable and under our control; as it would almost be impossible to complete the project without it.

We had the opportunity to try and parallelize our design, which we did try very early before most of the design was complete. By the end of the design it had been scrapped to maintain simplicity and functionality; it was also at this time that we had found the original key in only a matter of minutes, eliminating the need for any extra firepower. However, if we had pursued that bonus challenge it is quite possible that we would have come back to this, but alas.

• Section 5: Evaluation

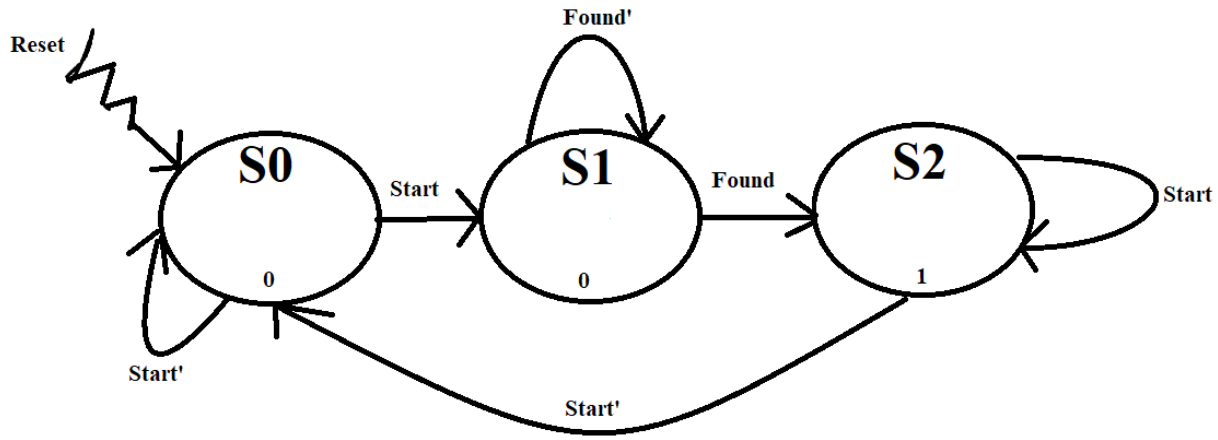
In our simulation we were able to find the key in a little over six minutes, which is incredibly quick, and when we ran the device on the fpga board, it came back with the result in only three minutes. To characterize this evaluation as anything other than successful would be improper and inadequate. Even so, had we more time to try and tackle the larger key, it might

have been a grander conclusion. Instead we must consider the reality of our results; at only an arm's reach away from the more challenging ciphertext, we have still come very far in compiling all that we have learned thus far, and to try and accomplish something much larger with it. We not only brought together registers and fsm design from the previous labs, but also DES, and together were able to brute force and break DES just like others did a long time ago. Perhaps the key was small, but the logic was all in the right place, and could theoretically keep going until it does eventually hit a larger key.

There are still optimizations that could be done to reduce the overall time that it would take to brute force a larger, or even the entire 64 bit key. As already discussed, parallelizing our design with more instances, filtering the bad parity keys, targeting other weak points, and even luck. Some of those we implemented, and others we did not. We shouldn't forget though, we need not necessarily check all of the keys, if we happen to get lucky and find it within only 25 percent total searched, we don’t need to continue from there. Just like trying the keys on a keychain, the first few may not be right, but once the correct one is found the process ends. Statistically speaking, the average case running time should be about 50% of the total keys, so unless you have poor luck, you shouldn't have to search all of them. In the case of sending other data like a message, we can exploit the fact that certain letters appear more than others, and even if certain algorithms are made to prevent this, there are still many things to consider to try and improve our attack. Although all of our data was arbitrary, this is still something to consider for other cryptography applications. Does this mean its possible? Yes of course, just not in the time frame we’ll live for. (Other Figures of Interest: 18 & 19)

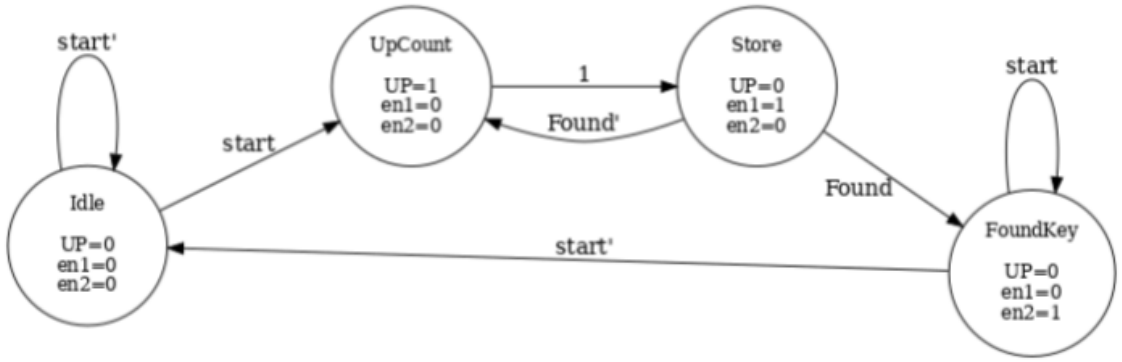
• Appendix

Figure 1



Initial FSM design

Figure 2



Final FSM design

Figure 3 - Design Diagram

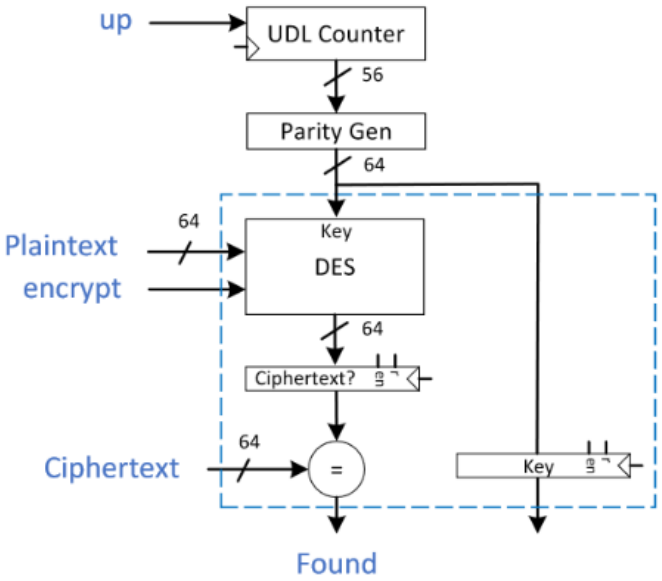


Figure 4 - Counter Logic

```
`timescale 1ns/1ps
module UDL_Count  #(parameter WIDTH=56)
   (clk, rst, up, down, load, in, out) ;

   input logic              clk;
   input logic              rst;
   input logic              up;
   input logic              down;
   input logic              load;
   input logic [55:0]  in;
   output logic [WIDTH-1:0] out;

   logic [WIDTH-1:0]        next;

   flop #(WIDTH) count(clk, next, out);

   always_comb begin
      if (rst)
      next = {WIDTH{64'b0}};
      else if (load)
      next = in;
      else if (up)
      next = out + 1'b1;
      else if (down)
      next = out - 1'b1;
      else
      next = out;
   end // always@ *
```

Figure 5 - Flip flop code

```
// ordinary flip-flop
module flop #(parameter WIDTH=8) (
  input  logic             clk,
  input  logic [WIDTH-1:0] d,
  output logic [WIDTH-1:0] q);

  always_ff @(posedge clk)
    q <= d;

endmodule
```

Figure 6 - Parity Generation module

```
module genParity(input logic [6:0] in, output logic [7:0] out);
  assign out[0] = ~^in;
  assign out[7:1] = in;
endmodule

module genParity8(input logic [55:0] in, output logic [63:0] out);
  genvar                                  index;
  for(index = 0; index < 8; index++) begin
      genParity genParity(.in(in[7*index +: 7]), .out(out[8*index +: 8]));
  end
endmodule
```

Figure 7 - DES Diagram

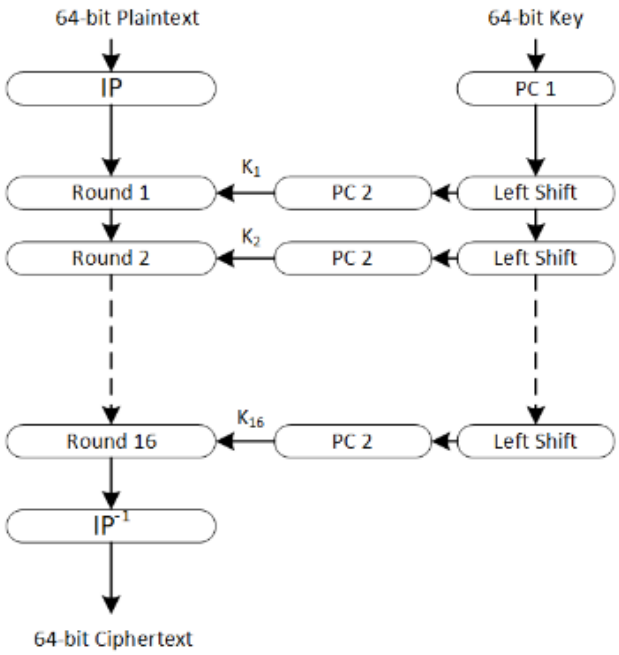


Figure 8 - Main Flip Flop

```
// enabled asynchronously resettable flip flop
module flopenr #(parameter WIDTH = 8)
  (input logic             clk,
   input logic         reset,
   input logic         en,
   input logic [WIDTH-1:0]  d,
   output logic [WIDTH-1:0] q);

  // asynchronous reset
  always_ff @(posedge clk, posedge reset)
    if      (reset) q <= {WIDTH{1'b0}};
    else if (en)    q <= d;
endmodule
```

Figure 9 - Top Main

```
module top (input logic     clk, Start, reset,
            input logic  [63:0] plaintext, ciphertext,
            output logic [55:0] count,
            output logic  [63:0]  Key,
            output logic        keyFound

      );
      logic [63:0] counter_64;
      logic [63:0] tempText;

      logic         en1;
      logic         en2;
      logic [63:0]  d1, d2, q1, q2;
      logic up;




                  //clk, reset, count, start, found, en1, en2, up
control control_1 (~clk, reset, Start, keyFound, en1, en2, up);

UDL_Count #(56) counter_1 (clk, reset, up, 1'b0, 1'b0, 56'h000_000_000_000_00, count);

genParity8 genParity8(count, counter_64);

DES des_1 (counter_64, plaintext, 1'b1, tempText);

flopenr ciphertext_Register (clk, reset, en1, tempText, q1);

comparator c (ciphertext,q1,keyFound);

flopenr key_Register (clk, reset, en2, counter_64, Key);

endmodule // top
```

Figure 10 - FSM / Control Logic

```
module control(clk, reset, start, keyFound, en1, en2, up);

input logic clk;
input logic reset;
input logic start;
input logic keyFound;
output logic en1, en2, up;


typedef enum logic [1:0] {S0,S1,S2,S3} statetype;
statetype state, nextstate;

  // state register
   always_ff @(posedge clk, posedge reset)
     if (reset) state <= S0;
     else       state <= nextstate;

    // next state + output logic
    always_comb
    case(state)

    S0: begin//idle
        en1 <= 1'b0;
        en2 <= 1'b0;
        up <= 1'b0;
        if(start == 1'b1) begin
            nextstate <= S1;
        end
        else nextstate <= S0;
    end

    S1: begin//upCount
        en1 <= 1'b0;
        en2 <= 1'b0;
        up <= 1'b1;
        nextstate <= S2;
```

```
    S1: begin//upCount
        en1 <= 1'b0;
        en2 <= 1'b0;
        up <= 1'b1;
        nextstate <= S2;

    end

    S2: begin//store
        en1 <= 1'b1;
        en2 <= 1'b0;
        up <= 1'b0;
        if (keyFound != 1'b1) begin
            nextstate <= S1;
        end
        else nextstate <= S3;
    end


    S3: begin//foundkey
        en1 <= 1'b0;
        en2 <= 1'b1;
        up <= 1'b0;

        if (start == 1'b1) begin
            nextstate <= S3;
        end
        else nextstate <= S0;

    end

    default: begin

        nextstate <= S0;
    end
endcase
```

Figure 11 - New UDL logic

```
`timescale 1ns/1ps
module UDL_Count  #(parameter WIDTH=56)
   (clk, rst, up, down, load, in, out) ;

   input logic              clk;
   input logic              rst;
   input logic              up;
   input logic              down;
   input logic              load;
   input logic [55:0]  in;
   output logic [WIDTH-1:0] out;

   logic [WIDTH-1:0]        next;

   flop #(WIDTH) count(clk, next, out);

   always_comb begin
      if (rst)
      next = {WIDTH{64'b0}};
      else if (load)
      next = in;
      else if (up)
      next = out + 1'b1;
      else if (down)
      next = out - 1'b1;
      else
      next = out;
   end // always@ *

endmodule
```

Figure 12 - Parity Generation From Count

```
module genParity(input logic [6:0] in, output logic [7:0] out);
  assign out[0] = ~^in;
  assign out[7:1] = in;
endmodule

module genParity8(input logic [55:0] in, output logic [63:0] out);
  genvar                                  index;
  for(index = 0; index < 8; index++) begin
      genParity genParity(.in(in[7*index +: 7]), .out(out[8*index +: 8]));
  end
endmodule
```

Figure 13 - DES Parameters

```
module DES (input logic [63:0] key, input logic [63:0] plaintext,
            input logic encrypt, output logic [63:0] ciphertext);
```

Figure 14 - Register

```
1   // enabled asynchronously resettable flip flop
2   module flopenr #(parameter WIDTH = 64)
3     (input logic              clk,
4      input logic        reset,
5      input logic        en,
6      input logic [WIDTH-1:0]  d,
7      output logic [WIDTH-1:0] q);
8
9     // asynchronous reset
10    always_ff @(posedge clk, posedge reset)
11      if      (reset) q <= {WIDTH{1'b0}};
12      else if (en)    q <= d;
13  endmodule
14
```

Figure 15 - Comparator

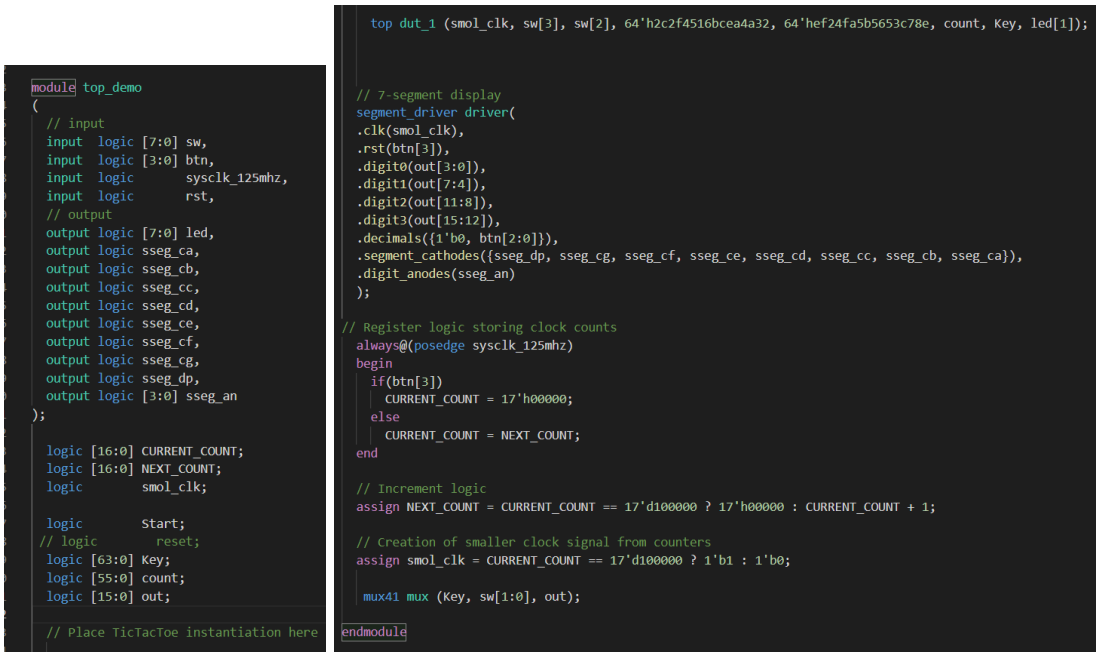
```
: > Users > soldi > Desktop > DLD > project >  comparator.sv
1   module comparator #(parameter WIDTH = 64)
2     (input logic [WIDTH-1:0] a, b,
3      output logic eq);
4
5     assign eq = (a == b);
6
7   endmodule // comparator
8
```

Figure 16 - Count.out

```
01010101010db6f7 1 | 0000000001adfb
```

```
initial
  begin
  clk = 1'b0;
  forever #5 clk = ~clk;
  end

initial
  begin
  vectornum = 0;
  outputFilePointer = $fopen("count.out");
end


always @(posedge clk)
  begin
  if (keyFound == 1'b1) begin

  #50 $fdisplay(outputFilePointer, "%h %b | %h", Key, keyFound, count);
  $display("Found key!");
  $stop();
    end
end


initial
  begin
  // Initialize
  #0   Start = 1'b0;
  #0   reset = 1'b1;
  #101 reset = 1'b0;
  #32  Start = 1'b1;
```

Figure 17 - Implementation on Vivado



```
module top_demo
(
  // input
  input  logic [7:0] sw,
  input  logic [3:0] btn,
  input  logic       sysclk_125mhz,
  input  logic       rst,
  // output
  output logic [7:0] led,
  output logic sseg_ca,
  output logic sseg_cb,
  output logic sseg_cc,
  output logic sseg_cd,
  output logic sseg_ce,
  output logic sseg_cf,
  output logic sseg_cg,
  output logic sseg_dp,
  output logic [3:0] sseg_an
);

  logic [16:0] CURRENT_COUNT;
  logic [16:0] NEXT_COUNT;
  logic        smol_clk;

  logic        Start;
 // logic        reset;
  logic [63:0] Key;
  logic [55:0] count;
  logic [15:0] out;

  // Place TicTacToe instantiation here
    top dut_1 (smol_clk, sw[3], sw[2], 64'h2c2f4516bcea4a32, 64'hef24fa5b5653c78e, count, Key, led[1]);



  // 7-segment display
  segment_driver driver(
  .clk(smol_clk),
  .rst(btn[3]),
  .digit0(out[3:0]),
  .digit1(out[7:4]),
  .digit2(out[11:8]),
  .digit3(out[15:12]),
  .decimals({1'b0, btn[2:0]}),
  .segment_cathodes({sseg_dp, sseg_cg, sseg_cf, sseg_ce, sseg_cd, sseg_cc, sseg_cb, sseg_ca}),
  .digit_anodes(sseg_an)
  );

// Register logic storing clock counts
  always@(posedge sysclk_125mhz)
  begin
    if(btn[3])
      CURRENT_COUNT = 17'h00000;
    else
      CURRENT_COUNT = NEXT_COUNT;
  end

  // Increment logic
  assign NEXT_COUNT = CURRENT_COUNT == 17'd100000 ? 17'h00000 : CURRENT_COUNT + 1;

  // Creation of smaller clock signal from counters
  assign smol_clk = CURRENT_COUNT == 17'd100000 ? 1'b1 : 1'b0;

   mux41 mux (Key, sw[1:0], out);

endmodule
```

Figure 18 - Timing in transcript

```
376   # Loading work.S2_Box(fast)
377   # Loading work.S3_Box(fast)
378   # Loading work.S4_Box(fast)
379   # Loading work.S5_Box(fast)
380   # Loading work.S6_Box(fast)
381   # Loading work.S7_Box(fast)
382   # Loading work.S8_Box(fast)
383   # Loading work.SF(fast)
384   # Loading work.FP(fast)
385   # Loading work.flopenr(fast)
386   # Loading work.comparator(fast)
387   # Found key!
388   # ** Note: $stop    : tb_class.sv(56)
389   #    Time: 2201695 ns  Iteration: 0  Instance: /stimulus_class
390   # Break in Module stimulus_class at tb_class.sv line 56
391   # End time: 16:18:30 on Apr 28,2022, Elapsed time: 1:29:57
392   # Errors: 0, Warnings: 0
393
```

Figure 19 - Key Found