CPR E 281 - Digital Logic
FALL SEMESTER 2015

INSTRUCTOR: ALEXANDER STOYTCHEV

STUDENT: MATTHEW WALL
EMAIL: MWALL@IASTATE.EDU

06 December 2015

For the project below, I listed my concurrent thought process throughout. I'll separate the different parts of the project within their appropriate headers. I'll try to explain the different parts with as much detail and thoroughness as possible. Listed below are the components that I used:

- 8-bit seed
- 8-bit shifter that loads and shifts out the bits starting from the most significant bit
- Finite-State Machine
- 4-bit 4 wide Register File
- 4 7-segment decoders that display the Random Number on a Hex Display.

8-bit Seed

The 8-bit seed is used to feed into an 8 bit shifter through switches on the Altera DE2 board. I avoided using the first toggle switch because on most boards I found that this toggle causes a really bad toggle bounce and dealing with that is nothing less than annoying. The seed is loaded into the shifter when my LOAD input is set to 0 and CLOCK is ticked (represented by Key 0). You'll see below in the shifter diagram the 8 leading inputs are representing the 8-bit seed. With any Random Number Generator, there has to be a level of non-randomness. Because the user loads in a seed, the algorithm we use is generically pseudorandom. There's not much to say about this seed other than the fact that I tested it with 8 red LEDs and if the LED was lit up, it meant the seed value was 1, 0 otherwise.

8-bit Shifter

I'll start out by explaining a little bit about what the shift register does. Just as the name sounds, it's used to shift out, starting with the most significant bit, to the Finite-State Machine. This digit, either being 1 or 0, is translated through the Finite-State Machine, in which I'll get into in the next heading. On top of this, we were given an equation into which the random number would be generated.

$$f(0) = 0$$

$$f(t+1) = \begin{cases} 3f(t), & if\ S = 0 \\ f(t) - 3, & if\ S = 1 \end{cases}$$

This is stating that throughout the entire seed being shifted out of the shifter, it'll follow this equation. Initially, since $f(0) = 0,$ then the first selected bit will either be a 0 or a 7 in my random number generator. Then, for every following tick, it'll be dependant on whether $S = 0$ or $S = 1$, and follow the equation respectively. One thing to note on my project is that I have my LOAD inverted, so when LOAD is down, it will load the seed after one clock tick. As aforementioned, I tested this shifter with the 8 red LEDs to ensure that the values were actually being set to 1 and 0 if they were up and down respectively. On the next page, you'll find my shift register. One problem I ran into was after the 8-bits were shifted out it went into a continuous loop of random numbers, pattern was subtracting 3 or something of that sort. I couldn't quite get that to fix, but I didn't necessarily know if it was a problem or not. My idea to fix that was to implement a D-Flip Flop counter that stopped after 8 ticks. I didn't quite have enough time to implement this unfortunately, but I got the machine to work regardless for the first 8-bits. Another problem I ran into was the NOT gate on all the RESET inputs. I didn't necessarily need to have it inverted, but I always had it down and nothing happened so I was struggling a little bit until I toggled it up and everything started working. The only purpose of the shift register is to shift out a 1 or 0 depending on the seed passed in from the DE2 board starting at D7, the most significant bit.

Finite-State Machine

Below is the Verilog code for the truth table behind my Finite-State Machine. I used modulo ten as my overflow fix. So for instance if the number was 3 and it got multiplied by 3 to equal 12, but since I modulo by base ten (since my RNG only goes from 0-9, or 10 integers) it would equal 2 to fix the overflow. My truth table for my FSM shows that the machine has 10 states since modulo based ten. I derived a state diagram and state table for this to demonstrate exactly how it works rather than explaining it.

```
module FSM (S,A,B,C,D,Z,Y,X,W);
    input S,Z,Y,X,W;
    output A,B,C,D;
    reg A,B,C,D;

    always@(S or Z or Y or X or W)
    begin
       case({S,Z,Y,X,W})
              5'b00000:{A,B,C,D}=4'b0000;
              5'b00001:{A,B,C,D}=4'b0011;
              5'b00010:{A,B,C,D}=4'b0110;
              5'b00011:{A,B,C,D}=4'b1001;
              5'b00100:{A,B,C,D}=4'b0010;
              5'b00101:{A,B,C,D}=4'b0101;
              5'b00110:{A,B,C,D}=4'b1000;
              5'b00111:{A,B,C,D}=4'b0001;
              5'b01000:{A,B,C,D}=4'b0100;
              5'b01001:{A,B,C,D}=4'b0111;
              5'b10000:{A,B,C,D}=4'b0111;
              5'b10001:{A,B,C,D}=4'b1000;
              5'b10010:{A,B,C,D}=4'b1001;
              5'b10011:{A,B,C,D}=4'b0000;
              5'b10100:{A,B,C,D}=4'b0001;
              5'b10101:{A,B,C,D}=4'b0010;
              5'b10110:{A,B,C,D}=4'b0011;
              5'b10111:{A,B,C,D}=4'b0100;
              5'b11000:{A,B,C,D}=4'b0101;
              5'b11001:{A,B,C,D}=4'b0110;
              default: {A,B,C,D}=4'b0000;
       endcase
    end
endmodule
```

I combine this with 4 D-Flip Flops and a bus line that feeds data output, while looping the outputs back into the inputs for the closure of the FSM. Throughout the FSM I have clock being controlled by a key since toggle bouncing was again a problem here with too with toggles. I assigned PRN and CLRN to a VCC as they don't really matter in a FSM, they should always be set to 1. The LD_DATA output that gets transferred into the register file is based solely on the seed value that gets pushed from the shift register. Because the

first value will only have one value, it'll be either a 7 or 0 as aforementioned. But after that the looped around values will take into account and your value will be changed. Here is the State Diagram, Truth Table, and State Table that helped me construct the FSM.

Truth Table for FSM

	A	B	C	D	S=0	S=1
0	0	0	0	0	0000	0111
1	0	0	0	1	0011	1000
2	0	0	1	0	0110	1001
3	0	0	1	1	1001	0000
4	0	1	0	0	0010	0001
5	0	1	0	1	0101	0010
6	0	1	1	0	1001	0011
7	0	1	1	1	0001	0100
8	1	0	0	0	0100	0101
9	1	0	0	1	0111	0110

State Table for FSM

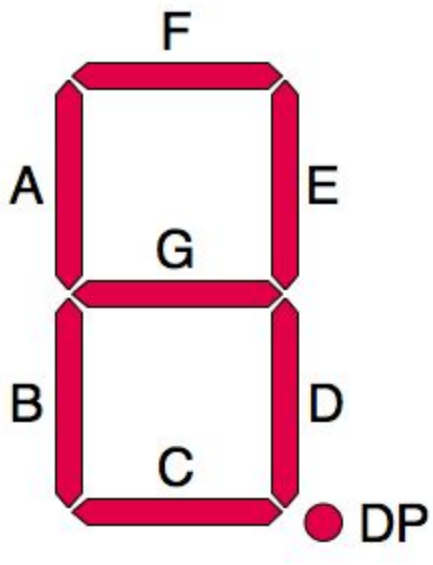
	Current State				Next State	
	A	**B**	**C**	**D**	**S=0**	**S=1**
A	0	0	0	0	A	H
B	0	0	0	1	D	I
C	0	0	1	0	G	J
D	0	0	1	1	J	A
E	0	1	0	0	C	B
F	0	1	0	1	F	C
G	0	1	1	0	J	D
H	0	1	1	1	B	E
I	1	0	0	0	E	F
J	1	0	0	1	H	G

Register File

The register file I split up into two parts. Step one consisted of making one 4-bit register that took in 4 data segments and outputted 4 data segments on the negative side of the LOAD data. I initially had forgot to negate my LOAD and that messed the whole program up by displaying to every Hex Display EXCEPT the one I wanted. I got it figured out once I realized my mistake. I had a line of four 2-1 multiplexers that fed the data into their respective D-Flip Flops based on the output from the mux. The output data was then fed into outputs and that sums up a single 4-bit register that I made a symbol file for ease in the next step. This is where things got out of hand. Initially, I have two inputs, WA0 and WA1 which decide the index to write random number to. Also, I initially made my 2-1 decoder without an enable and had to go back and add one. Anyway, WA0 and WA1 lead to A and B respectively and output if and only if enable was high. This is where the tricks came into play. Initially, I had the D0-D3 inputs leading into LOAD backwards, so it was causing my hex displays to display in the improper places. Anyway, from here I eventually switched them around and fed them in the correct LOADs. I negated RESET because I wanted it to toggled on if I wanted to reset the displays. I had a 4-bit bus line fed in from the FSM, so my DATA0-DATA3 inputs are for that, and conveniently I named them so they matched up. And once my data is fed in through each singular *register file* it outputs to four 4-bit bus lines: D[3..0], C[3..0], B[3..0], A[3..0]. This will come in handy when we connect our 7-segment displays to each bus line to display the data.

7-Segment Decoders

The seven segment decoder works as simple as taking in 4 inputs (in my case a bus line fed from the four wide register file) and converting that to display on a seven segment hex display much like the one seen below:



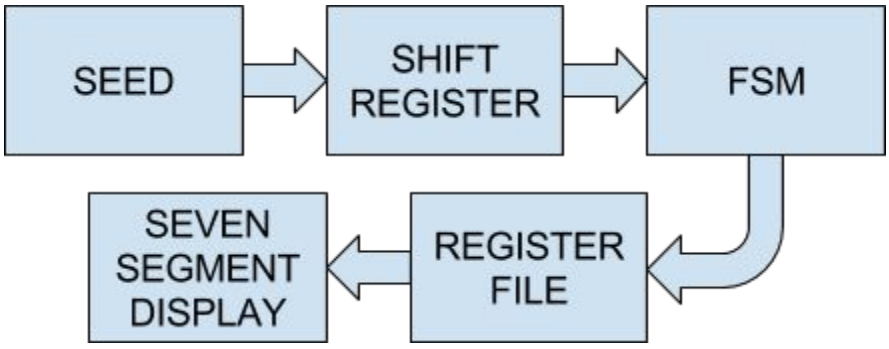
The outputs actually match up to figure displayed so in my case it would be 7'ABCDEFG; which for example if the input to the seven segment display was all zeroes it would result in every input being lit up other than G (which would look like a zero) due to the fact that when the variable is 1 it's actually toggled off. Here's my Verilog code I used to convert from a four bit bus to the seven segment display on the following page.

Seven Segment Display

```
module seven_seg_decoder (A,B,C,D,E,F,G,Z);
    input [3:0]Z;
    wire [3:0]Z;
    output A,B,C,D,E,F,G;
    reg A,B,C,D,E,F,G;

    always @(Z)
    begin
       case({Z})
              4'0000:{A,B,C,D}=7'b0000001;
              4'0001:{A,B,C,D}=7'b1001111;
              4'0010:{A,B,C,D}=7'b0010010;
              4'0011:{A,B,C,D}=7'b0000110;
              4'0100:{A,B,C,D}=7'b1001100;
              4'0101:{A,B,C,D}=7'b0100100;
              4'0110:{A,B,C,D}=7'b0100000;
              4'0111:{A,B,C,D}=7'b0001111;
              4'1000:{A,B,C,D}=7'b0000000;
              4'1001:{A,B,C,D}=7'b0000100;
              4'1010:{A,B,C,D}=7'b0001000;
              4'1011:{A,B,C,D}=7'b1100000;
              4'1100:{A,B,C,D}=7'b0110001;
              4'1101:{A,B,C,D}=7'b1000010;
              4'1110:{A,B,C,D}=7'b0110000;
              4'1111:{A,B,C,D}=7'b0111000;
       endcase
    end
endmodule
```

And finally when we put it all together we have the order listed below:



The last file I'll attach is my final project as a whole with each component put together and represented. The pins are clearly labelled and if you follow the project display as I described you can see that the design is merely controlled solely from the input seed given by the user. As mentioned before, nothing can be completely random. We have to pass in a seed which makes the Random Number Generator actually a Pseudo-Random Number Generator.