Free Response

Please answer the following questions.

1. Please explain, at a high level, how encryption using a LFSR works. (4 pts)

Word limit: 200 words

The LFSR is a linear feedback shift registers, which uses linked D-flip-flops in a chain. This example uses a 6 tap (or 6 base 2) example to cover all 2^6 – 1 cases possible out of 63. This series of bits is used to XOR with the original message. The LFSR is also capable of reverse engineering a decoded signal by using adjusted ASCII underscores as a codex. Using 6 bits in the shift register instead of the most significant bit allows us to port the LFSR state bit into an array for each clock cycle, bitwise XORing the current data at each clock cycle as a form of rolling code. The pseudo-random feature of the LSFR comes from the same sequence of bits occurring if everything was reset, unlike a true randomizer.

2. Please explain, in detail, the behavior of a LFSR. (4 pts)

Word limit: 200 words. Think about the input, output, timing, and behavior of the module.
TODO

The LFSR in this situation has 6 states and is a linear feedback shift register, using linked D-flip-flops in a chain. For each clock signal, the register's bits shift tom the right, as in 1>2, 2>3. This allows for a representation of six bits 63 different ways, of which we will use 6 combinations and XOR them with each byte of the original message, storing this value in the register 64 away, in order to store the encrypted message from 0-62 to 63-128 in a 1-to-1 way. Using maximal length signals in our scenario, we will be avoiding the all-zero possibility, which does nothing with XOR.

3. How did you manipulate your address pointers (both read and write) to implement your design? (4 pts)

Word limit: 200 words
TODO

waddr and raddr are two pointers used to access the addresses dat_mem[0] through dat_mem[127] in order to load from addresses 0-52 (preloaded through testbench) and manipulate the values in order to write them into addresses 64+pre_len, 65+pre_len…

In my design, the raddr from 0+X, XOR, and waddr into 64+X all happen per clock cycle when it comes to each byte. raddr also accesses the values 62-64 when accessing pre_len, taps, or start if that information is neeeded.

4. What is the purpose of making our messages have a preamble in our design? (3 pts)

Word limit: 200 words
TODO

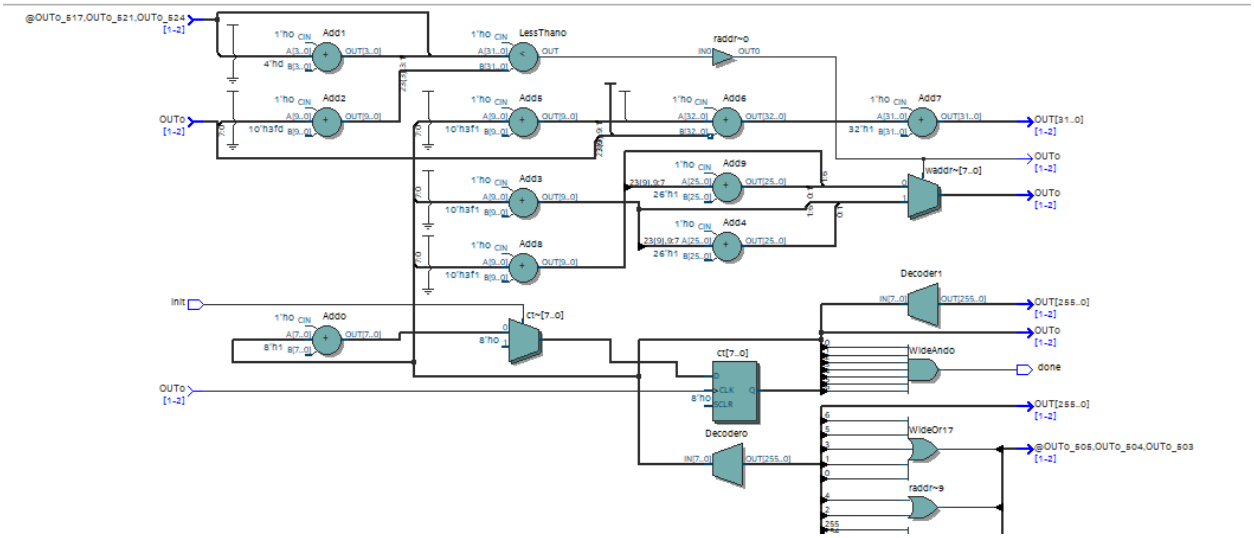
Having at least 6 underscores characters as a preamble when using a 6 tap LFSR allows us to encode 6 different encryptions for the same character, giving us a means of reverse engineering the different encryption algorithm for each symbol. Given this, the receiver is able to determine the starting state, which LFSR tap pattern (out of 6) was used, and before the clock cycle goes around more than 6 times.

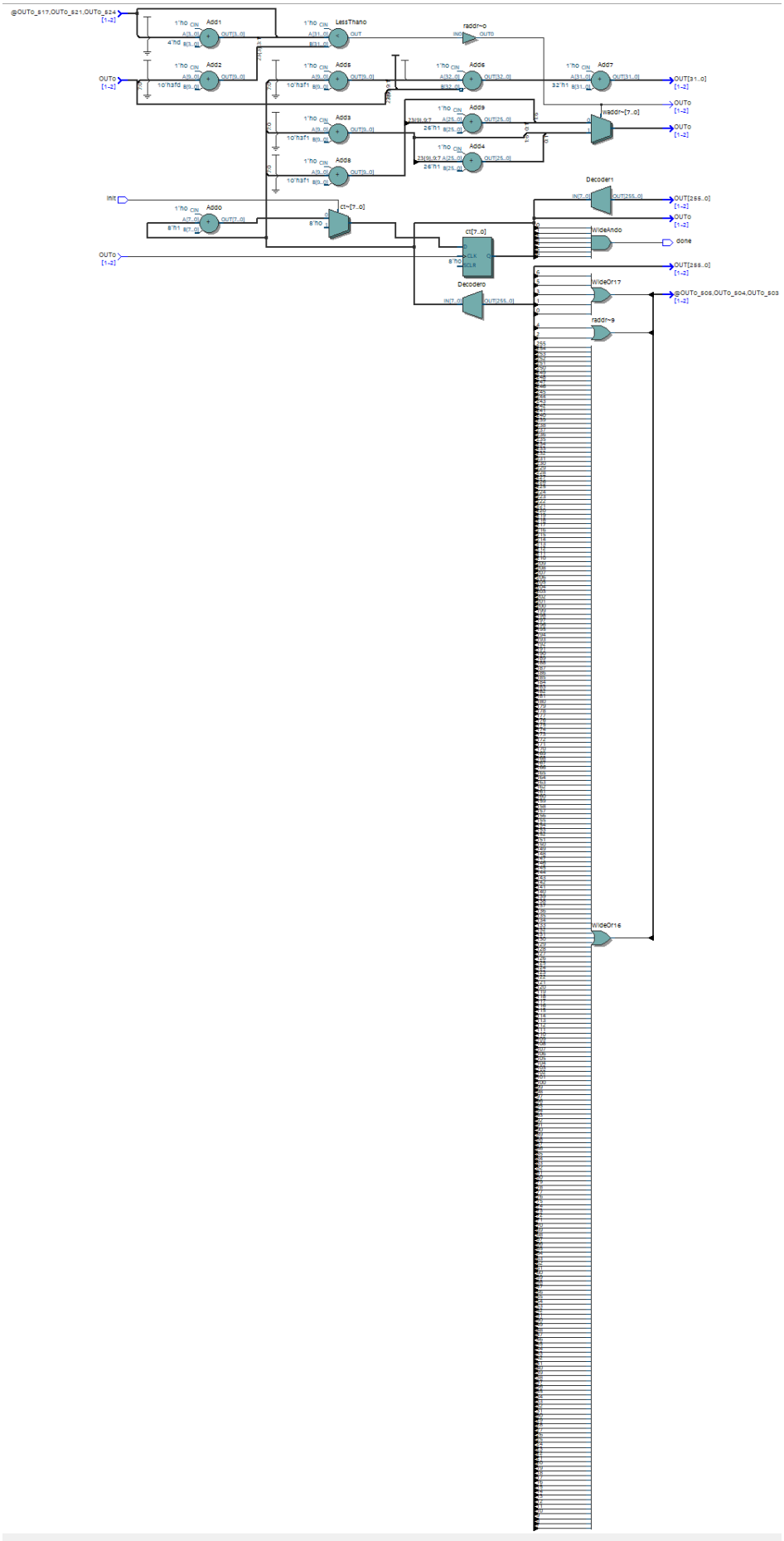
Screenshots

Screenshot of the RTL viewer top level schematic/block diagram in Quartus Or submit your Mentor Precision netlist file if using EDA Playground (5 pts)

TODO

see **netlist.sv** at end

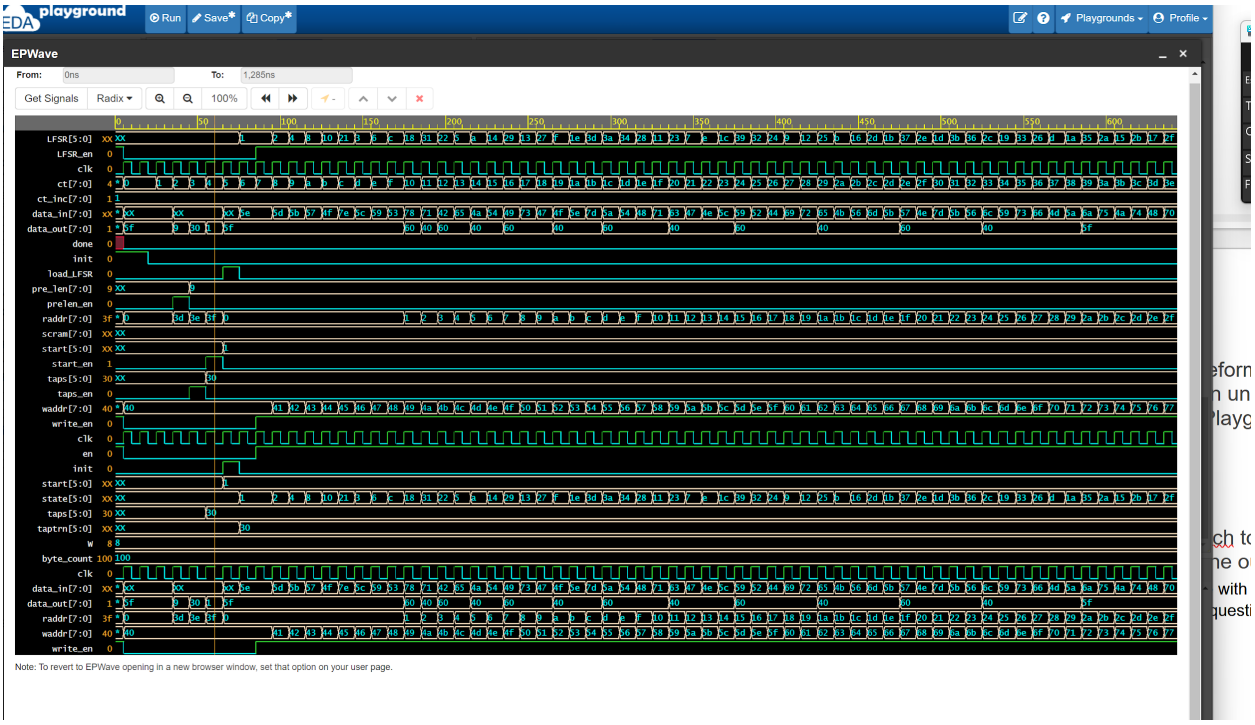



@OUTo_817,OUTo_821,OUTo_824
[1-2]
Add1
Add2
Add5
Add6
Add7
Add8
Add9
Add4
Add3
Addo
LessThano
raddr~o
waddr~[7..0]
OUTo
[1-2]
OUT[31..0]
OUT[255..0]
Decoder1
Decodero
WideAndo
WideOr17
WideOr16
raddr~9
done
ct~[7..0]
ct[7..0]
Init
@OUTo_805,OUTo_804,OUTo_803

Screenshot of your waveform viewer, including variables from both the testbench and the design under test. Or submit your Mentor Precision netlist file if using EDA Playground (5 pts)

TODO

see **netlist.sv** at end



Please edit your testbench to pipe the transcript to an output file in your submission, and name the output file “output.txt” (5 pts)

We will be looking for a text file with that name specifically, so be sure to rename it. Nothing is required in the writeup for this question.

```
// netlist.sv
//
// Verilog description for cell top_level,
// Fri Aug 4 03:39:42 2023
//
// Precision RTL Synthesis, 64-bit 2021.2.0.8//


module top_level ( clk, init, done ) ;

input clk ;
input init ;
output done ;

wire [6:0]ct;
wire \inc_d(0) , nx8474z1, \inc_d(1) , nx8475z1, \inc_d(2) ,
nx51685z1,
\inc_d(3) , nx8477z1, \inc_d(4) , nx8478z1, \inc_d(5) , nx51682z1,
\inc_d(6) ;
wire clk_int;
wire init_int, nx8473z1, nx54262z1, nx26479z1, nx26479z2;



FDRE \reg_q(6) (.Q (ct[6]), .C (clk_int), .CE (nx54262z1), .D
(\inc_d(6) )
, .R (init_int)) ;
FDRE \reg_q(5) (.Q (ct[5]), .C (clk_int), .CE (nx54262z1), .D
(\inc_d(5) )
, .R (init_int)) ;
FDRE \reg_q(4) (.Q (ct[4]), .C (clk_int), .CE (nx54262z1), .D
(\inc_d(4) )
, .R (init_int)) ;
FDRE \reg_q(3) (.Q (ct[3]), .C (clk_int), .CE (nx54262z1), .D
(\inc_d(3) )
, .R (init_int)) ;
FDRE \reg_q(2) (.Q (ct[2]), .C (clk_int), .CE (nx54262z1), .D
(\inc_d(2) )
, .R (init_int)) ;
FDRE \reg_q(1) (.Q (ct[1]), .C (clk_int), .CE (nx54262z1), .D
(\inc_d(1) )
, .R (init_int)) ;
FDRE \reg_q(0) (.Q (ct[0]), .C (clk_int), .CE (nx54262z1), .D
(\inc_d(0) )
```

```
, .R (init_int)) ;
XORCY xorcy_0 (.O (\inc_d(0) ), .CI (nx54262z1), .LI (ct[0])) ;
XORCY xorcy_1 (.O (\inc_d(1) ), .CI (nx8474z1), .LI (ct[1])) ;
XORCY xorcy_2 (.O (\inc_d(2) ), .CI (nx8475z1), .LI (ct[2])) ;
XORCY xorcy_3 (.O (\inc_d(3) ), .CI (nx51685z1), .LI (ct[3])) ;
XORCY xorcy_4 (.O (\inc_d(4) ), .CI (nx8477z1), .LI (ct[4])) ;
XORCY xorcy_5 (.O (\inc_d(5) ), .CI (nx8478z1), .LI (ct[5])) ;
XORCY xorcy_6 (.O (\inc_d(6) ), .CI (nx51682z1), .LI (ct[6])) ;
OBUF done_obuf (.O (done), .I (nx26479z1)) ;
IBUF init_ibuf (.O (init_int), .I (init)) ;
GND ps_gnd (.G (nx8473z1)) ;
VCC ps_vcc (.P (nx54262z1)) ;
LUT4 ix26479z34082 (.O (nx26479z1), .I0 (nx26479z2), .I1 (ct[6]), .I2
(ct[5]
), .I3 (ct[4])) ;
defparam ix26479z34082.INIT = 16'h8000;
LUT4 ix26479z34083 (.O (nx26479z2), .I0 (ct[3]), .I1 (ct[2]), .I2
(ct[1]), .I3 (
ct[0])) ;
defparam ix26479z34083.INIT = 16'h8000;
BUFGP clk_ibuf (.O (clk_int), .I (clk)) ;
MUXCY muxcy_0 (.O (nx8474z1), .CI (nx54262z1), .DI (nx8473z1), .S
(ct[0])) ;
MUXCY muxcy_1 (.O (nx8475z1), .CI (nx8474z1), .DI (nx8473z1), .S
(ct[1])) ;
MUXCY muxcy_2 (.O (nx51685z1), .CI (nx8475z1), .DI (nx8473z1), .S
(ct[2])) ;
MUXCY muxcy_3 (.O (nx8477z1), .CI (nx51685z1), .DI (nx8473z1), .S
(ct[3])) ;
MUXCY muxcy_4 (.O (nx8478z1), .CI (nx8477z1), .DI (nx8473z1), .S
(ct[4])) ;
MUXCY muxcy_5 (.O (nx51682z1), .CI (nx8478z1), .DI (nx8473z1), .S
(ct[5])) ;
endmodule
```

Free Response

Please answer the following questions.

1. Please describe, at a high level, how each stage of decryption using LFSR works. (4 pts)

Word limit: 200 words.
The stages we are looking for you to describe are the preamble stage, training stage, and decryption stage.
TODO

Using a pseudo-random code (a binary sequence in our case), the encrypter takes a start state that is necessary for the decoder to determine later. After shifting the bits and determining the selection of taps and their order, the LSFR XORs with the passcode, and then it continues for each new character or word, with steps being taken on both ends to pad the size for consistent manipulation
The preamble stage determines the key, as the sequence of same characters in a decipherable manner is the key to solving the puzzle. By going through six clock cycles, we are able to determine which of the LSFRs are being used in the decryption stage. The training stage is the aforementioned stage which involves deciphering the preamble characters, literally training their behaviors through deducing which sequence was used.

2. Why do we use 6 LFSRs in our top level module? (4 pts)

Word limit: 200 words.
TODO

The use of 6 LSFRs is a means of multi-core programming that allowed six operations to be performed per time cycle. This speeds up the runtime of the project by at least sixfold, and the comparison between the values is able to be performed on a single clock edge when it may take more for a novice. Typically, a codex is an intricate series of operations performed on a phrase or password, so the more registers, the easier it is to write and solve these puzzles. Having multiprocessor machines will help us design schematics in 141. Using multiple systems in a design can maximize the speed available via hardware.

3. Please explain how the testbench tests your design. (4 pts)

Word limit: 200 words.
You can briefly describe the major steps taken by the testbench.
TODO

The test bench takes a line, phrase, or sentence, chooses a preamble length, and encodes the values of the bytes into the registers 64-70 in a rolling code cryptograph format. Using a series of underscores XORed against the desired pattern, the testbench applies the code to a series of @@@@”” characters or a normal sentence that can be manipulated by the user. The TB chooses a random LSFR of the six given, and it also uses a random acceptable preamble length for the part 2 problem. It also performs its own, higher level version of the rolling code algorithm in order to test our implementation.

4. In our implementation, the preamble is an underscore character. Is it possible to decrypt messages with an unknown preamble character? How would you edit your decryption design to accommodate for different character preambles? (3 pts)

Word limit: 200 words.
Please answer both questions. For the second question, just explain at a high level what you would do. No need to give us extra code for this question. Unknown means that the receiver does not know what the preamble character is.
TODO

Given an unknown preamble character, it would be assumed that there would be a series of the same character as the preamble, similarly to the underscore in the lab. As it is simply a different starting ASCII code for a different symbol, using the encoded values and reverse engineering them should give the user a good idea of what the value could be. Given an unknown character, you can always double XOR it to return to the start, which I would try to implement in my design for the convenience factor. Obviously instead of relying on the ‘h5F for hardcoding the ASCII of the underline, I would have to use a generic means of finding the key, which may involve cycling through all the relevant ASCII characters.

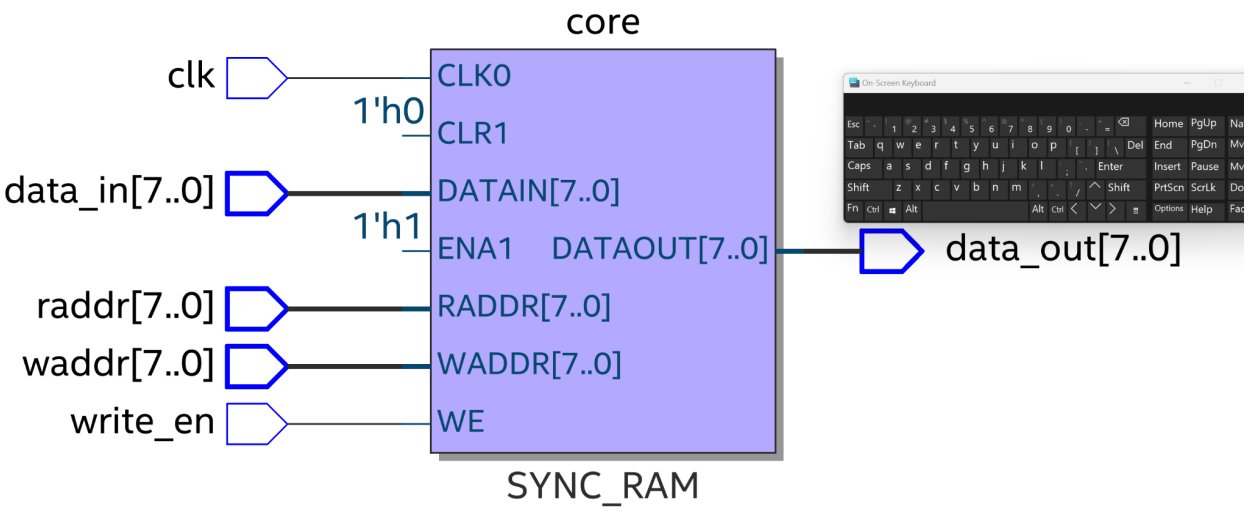
Screenshots

Screenshot of the RTL viewer top level schematic/block diagram in Quartus
Or submit your Mentor Precision netlist file if using EDA Playground (5 pts)

TODO

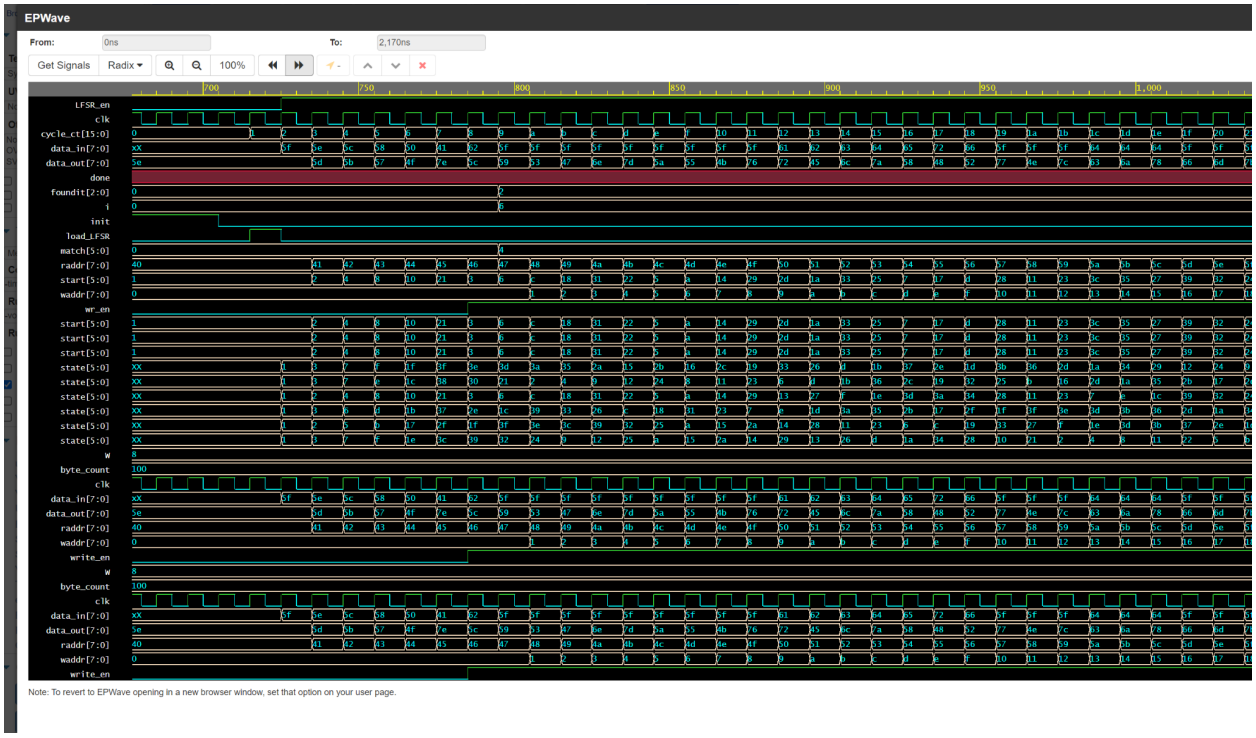
netlist.sv

```
//
// Verilog description for cell top_level_5b,
// Sun Aug 6 16:45:58 2023
//
// Precision RTL Synthesis, 64-bit 2021.2.0.8//


module top_level_5b ( clk, init, done ) ;

input clk ;
input init ;
output done ;

wire nx26479z1;



OBUF done_obuf (.O (done), .I (nx26479z1)) ;
VCC ps_vcc (.P (nx26479z1)) ;
endmodule
```



Screenshot of your waveform viewer, including variables from both the testbench and the design under test. Or submit your Mentor Precision netlist file if using EDA Playground (5 pts)

TODO



netlist.sv