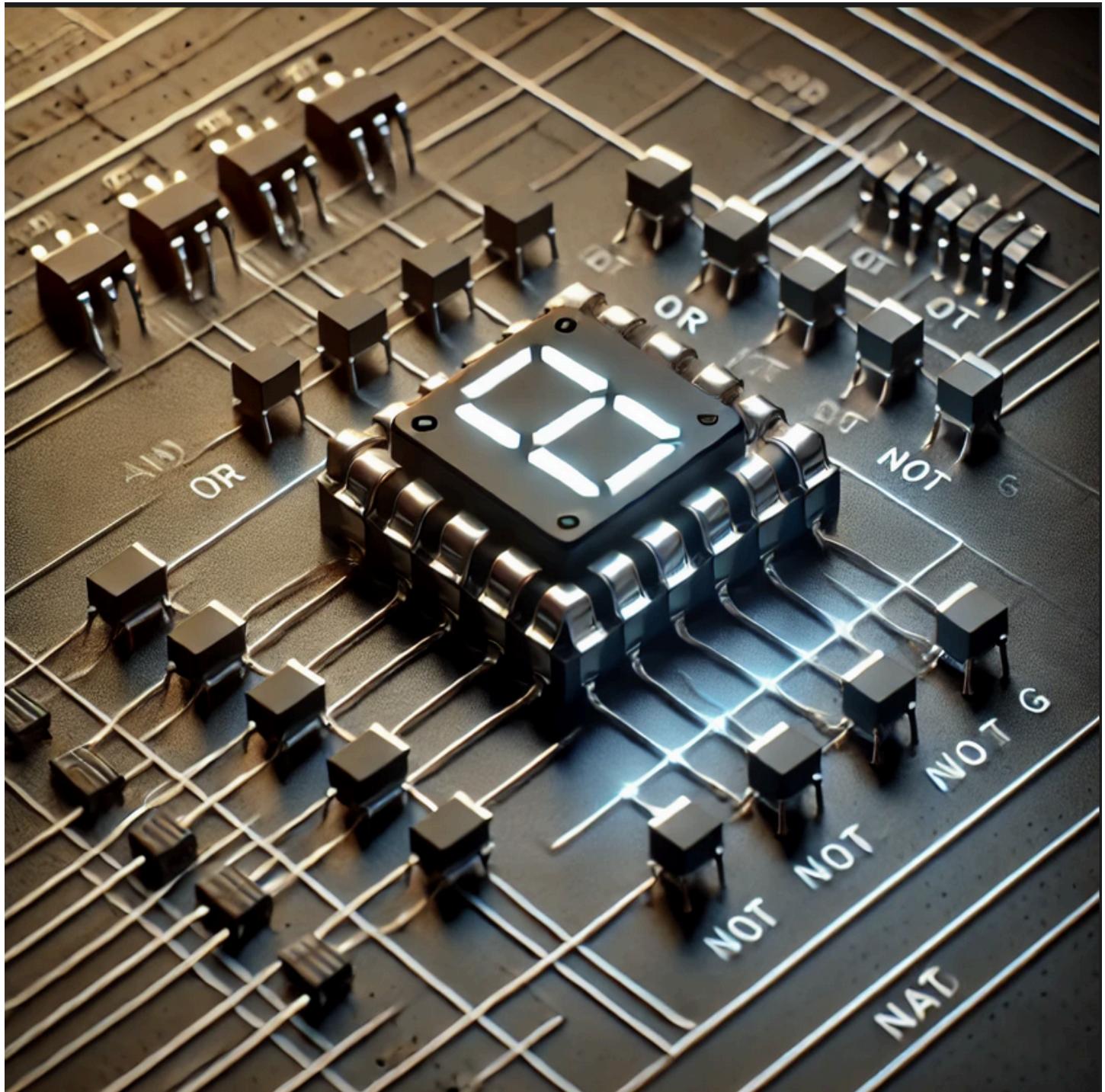


پروژه کامپیووتری سوم درس مدار منطقی

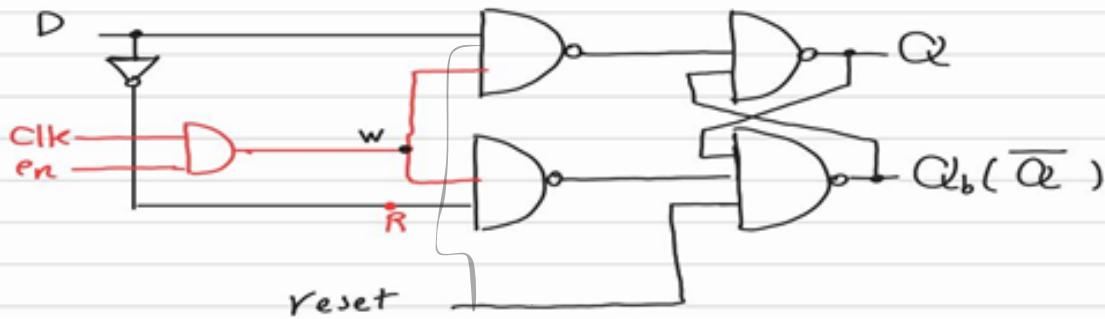
محمد امین رشید 810102454



D_Flip_Flop:

ابتدا باید d_latch را پیاده سازی کنیم:

D-Latch



که یک سیگنال سنکرون en و یک سیگنال آسنکرون reset دارد. البته در این مورد به سیگنال reset نیاز نداریم و هنگامی که یک فلیپ فlap را طراحی میکنیم این سیگنال اهمیت پیدا میکند.

سپس کد سطح گیت سیستم وریلاغ رو میزنیم و سپس تست بنچ مینویسیم:

```

D_Latch.sv
1 module D_latch (input D,input Clk,input en,input reset, output logic Q , output logic Qb);
2   wire P1,P2,R,w;
3   and(w,Clk,en);
4   nand(P1,D,w,reset);
5   not(R,D);
6   nand(P2,R,w);
7   nand(Q,P1,Qb);
8   nand(Qb,P2,Q,reset);
9
10 endmodule
11

```

در تست بنچ از چندتا initial block به صورت موازی استفاده میکنیم تا سیگنال کلاک و سایر سیگنال هارا جدا تغییر میدهیم. فرکانس کلاک را 20/1 در نظر میگیریم پس دوره تناوب برابر 20 نانو ثانیه است.

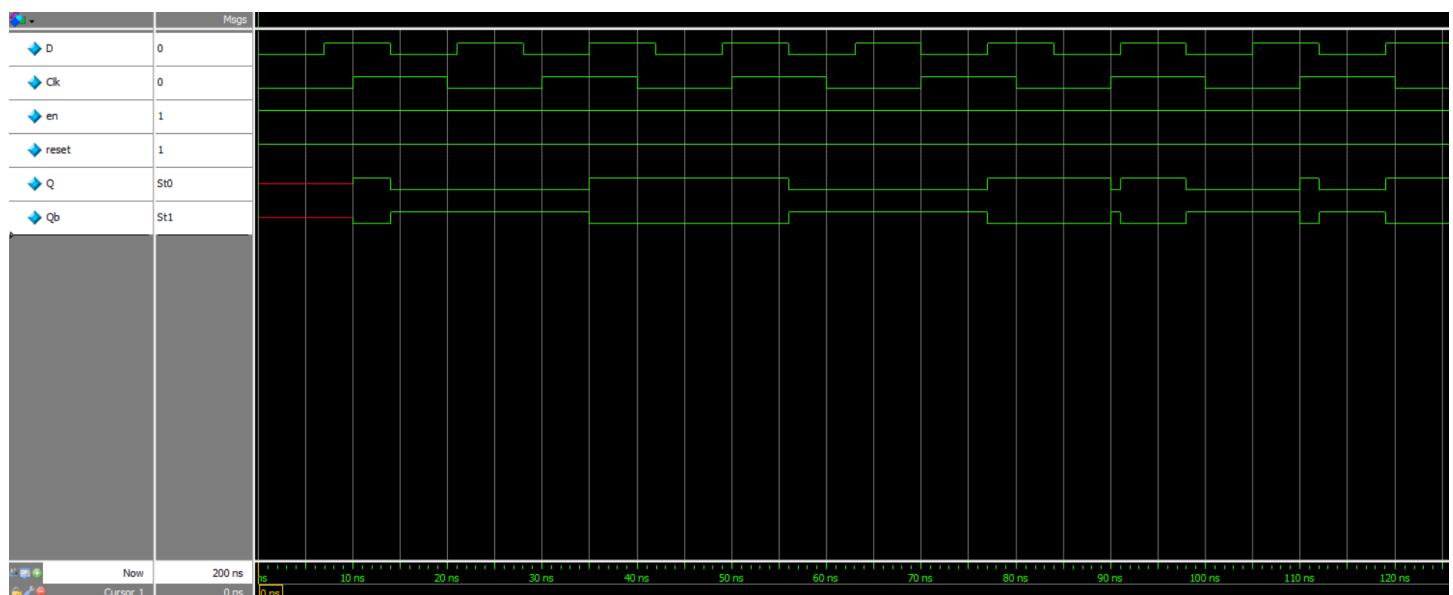
سیگنال en active high است ولی سیگنال reset active low است یعنی وقتی صفر میشود ریست اتفاق میفتند. سیگنال ورودی را هم هر 7 نانو ثانیه عوض میکنیم.

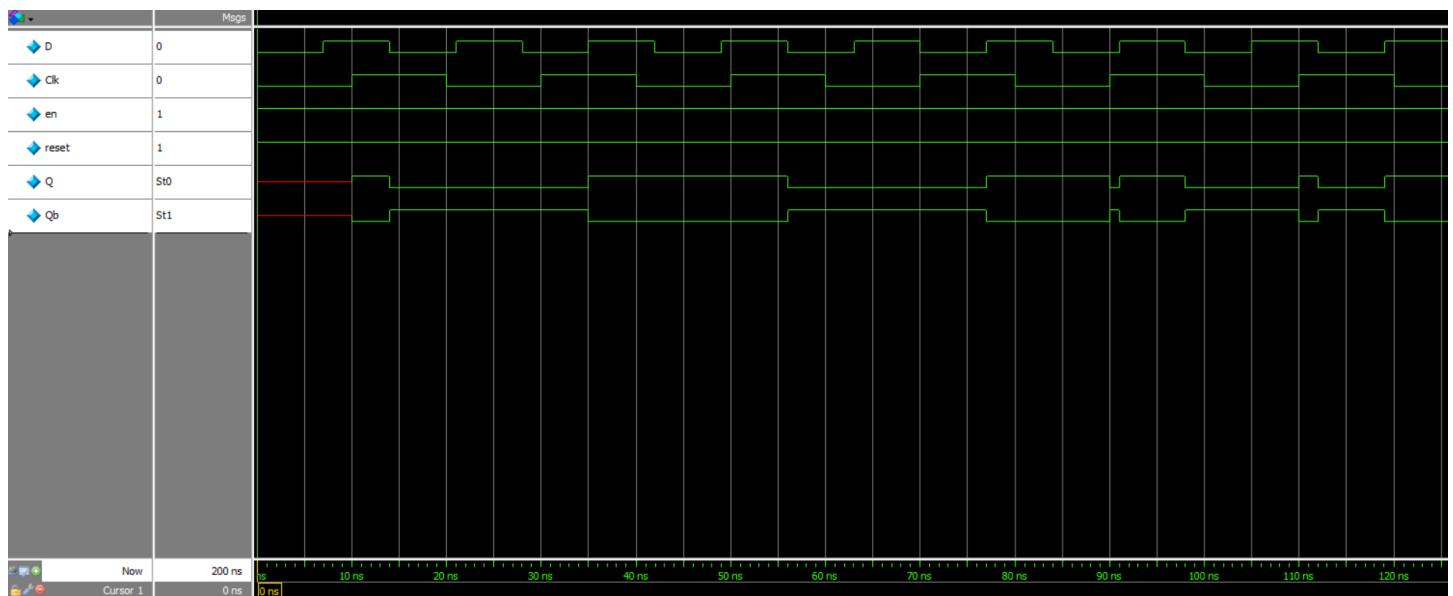
```

D_Latch_tb.sv
1 `timescale 1ns/1ns
2 module D_Latch_tb;
3
4
5     logic D=0;
6     logic Clk=0;
7     logic en=1;
8     logic reset=1;
9     wire Q;
10    wire Qb;
11    D_latch d_latch(D,Clk,en,reset,Q,Qb);
12    initial begin
13        repeat(20) #10 Clk=~Clk;
14    end
15
16    initial begin
17        repeat(20) #7 D=~D;
18
19    end
20
21    initial begin
22
23        #51
24        reset=0;
25        #1
26        reset=1;
27        #10
28        reset=0;
29        #1
30        reset=1;
31
32    end
33 endmodule
34

```

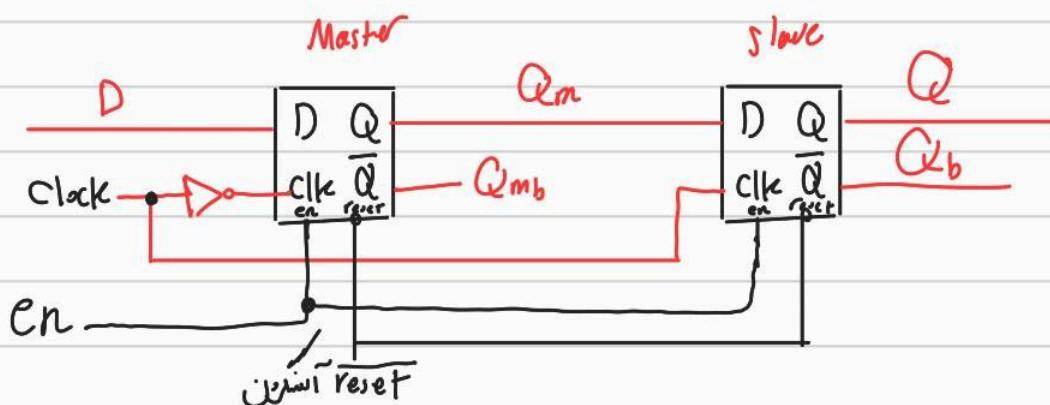
شبیه سازی را انجام میدهیم و به موج های زیر میرسیم و کاملا درست عمل میکند زیرا تنها زمانی D روی Q میرود که باشد در غیر صورت تا 1 شدن کلک صبر میکند.





سپس یک d_flip_flop master slave را طراحی میکنیم که به صورت زیر است:
این فلیپ فلاپ به صورت rising edge است.

Master_Slave D-FlipFlop :



کد سیستم وریلاگ+تست بنچ:

```
1 module D_FlipFlop(input D,input Clk,input en,input reset, output logic Q , output logic Qb);
2   wire Qm,Qmb;
3   wire not_Clk;
4   not(not_Clk,Clk);
5
6   D_latch dl1(D,not_Clk,en,reset,Qm,Qmb);
7   D_latch dl2(Qm,Clk,en,reset,Q,Qb);
8
9 endmodule
```

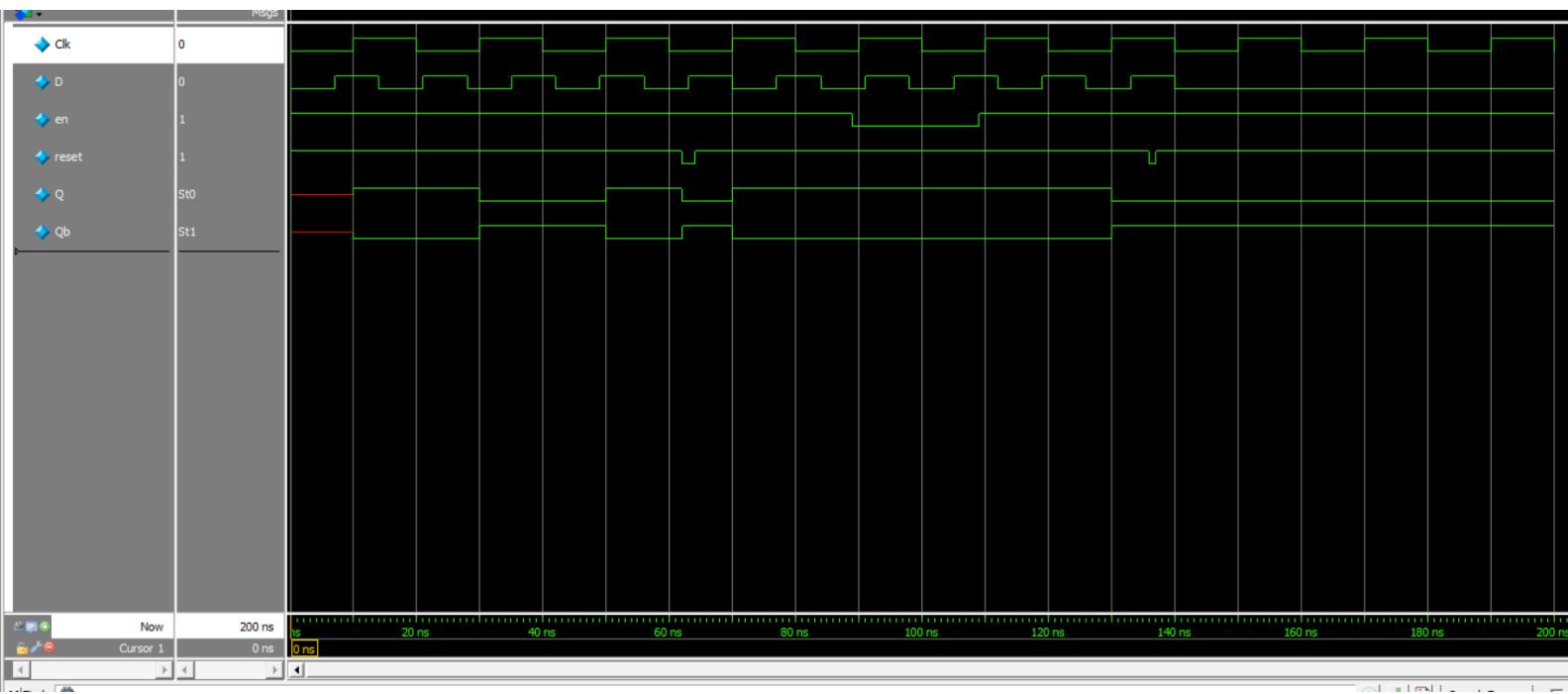
```

1  `timescale 1ns/1ns
2  module D_FlipFlop_tb;
3      logic D=0;
4      logic Clk=0;
5      logic en=1;
6      logic reset=1;
7      wire Q;
8      wire Qb;
9
10 D_FlipFlop DFF(D,Clk,en,reset,Q,Qb);
11 initial begin
12     repeat(20) #10 Clk=~Clk;
13 end
14
15 initial begin
16     repeat(20) #7 D=~D;
17
18 end
19 initial begin
20
21     #62 reset=0;
22     #2 reset=1;
23     //129 -> 149
24     #25 en=0;
25     #20 en=1;
26
27
28     #27
29     reset=0; //at 196 ns
30     #1 reset=1;
31     // #16
32     // reset=0;
33     // #1 reset=1;
34     end
35 endmodule
36

```

در تست بنج سیگنال های reset , en را هم چک میکنیم. در حالتی که en =0 است انگار کلک 0 است و در حالتی که reset یک است، Q صفر میشود و جدا از کلک عمل میکند زیرا آسنکرون است.

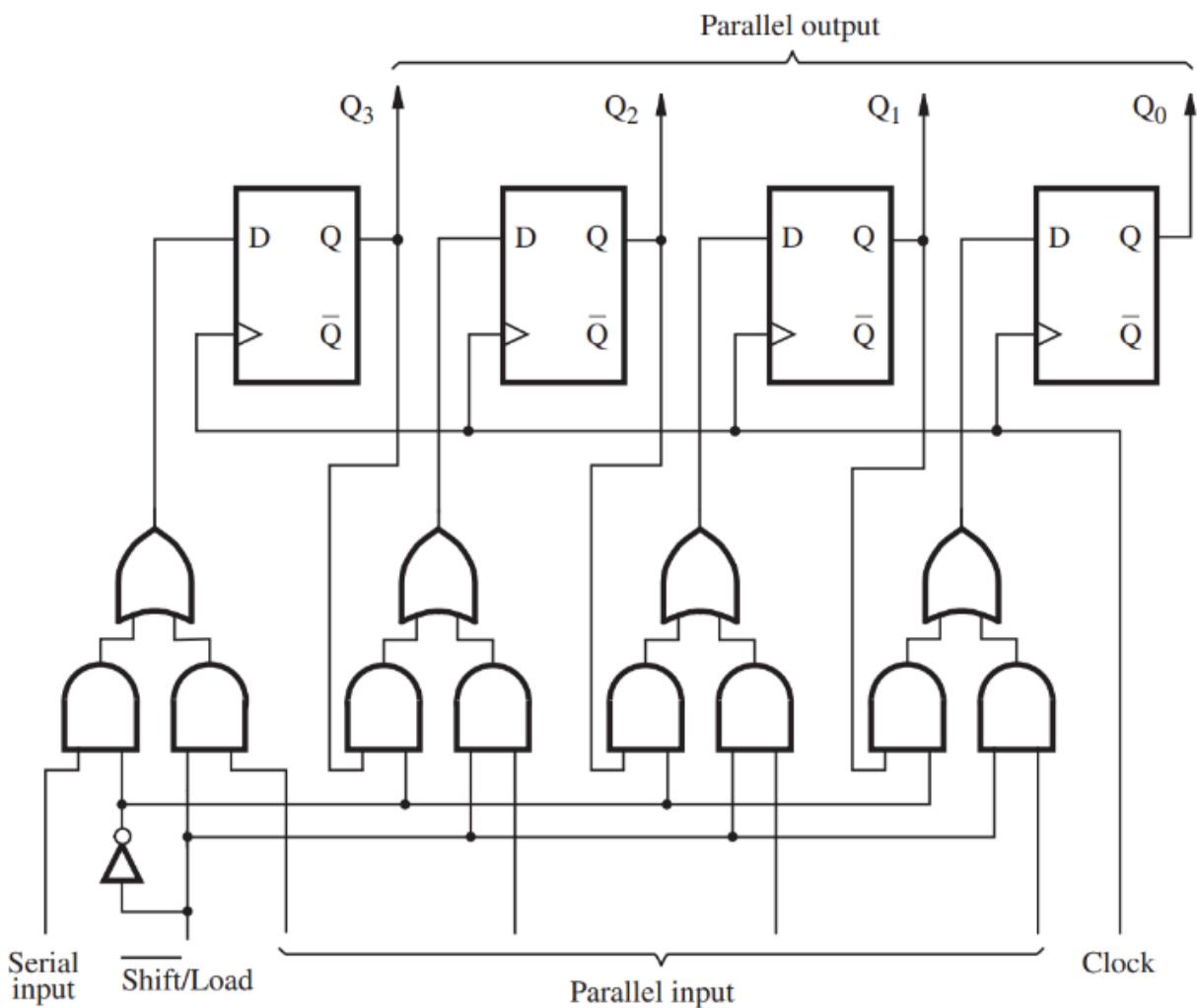
شبیه سازی:



کاملا مشخص است که فقط زمانی که کلک از 0 به 1 تغییر میکند D روی Q می رود و سیگنال $reset$ به درستی کار میکند.

Shift Register:

در این بخش میخواهیم یک شیفت رجیستر N بسازیم که شکل 4 بیتی آن به شکل زیر است.



کد آن را به شکل زیر میزیم و از generate برای تولید فلیپ فلاب ها و سایر گیت ها استفاده میکنیم.

به جای استفاده از سیگنال shift/load از دو تا سیگنال مجزا استفاده میکنیم ولی باید در تست بنچ حواسمن باشد تا همیشه این دو سیگنال نقیض هم باشند.

```

1 module shift_register #(parameter N=8)(input Clk, ser_in,reset ,Par_load,shift_en,input [0:N-1] Par_in,output [0:N-1] Par_out,output ser_out);
2
3
4 wire D[0:N-1];
5 genvar i;
6 generate
7
8   for(i=0;i<N;i=i+1)begin
9     if(i==0)begin
10       assign D[i]=(ser_in& shift_en) | (Par_load & Par_in[i]);
11     end
12
13     else begin
14       assign D[i]=( Par_out[i-1]& shift_en) | (Par_load & Par_in[i]);
15     end
16
17   end
18
19   D_FlipFlop dff(D[i],Clk,1,reset,Par_out[i]);
20
21 end
22
23 assign ser_out = Par_out[N-1];
24 endgenerate
25
26 endmodule

```

تست بنچ:

در بلاک initial اول بحث این که لود بکند و یا شیفت دهد را هندل میکنیم و در بلاک دوم سریال ورودی و ریست را و همه حالات را چک میکنیم:

```

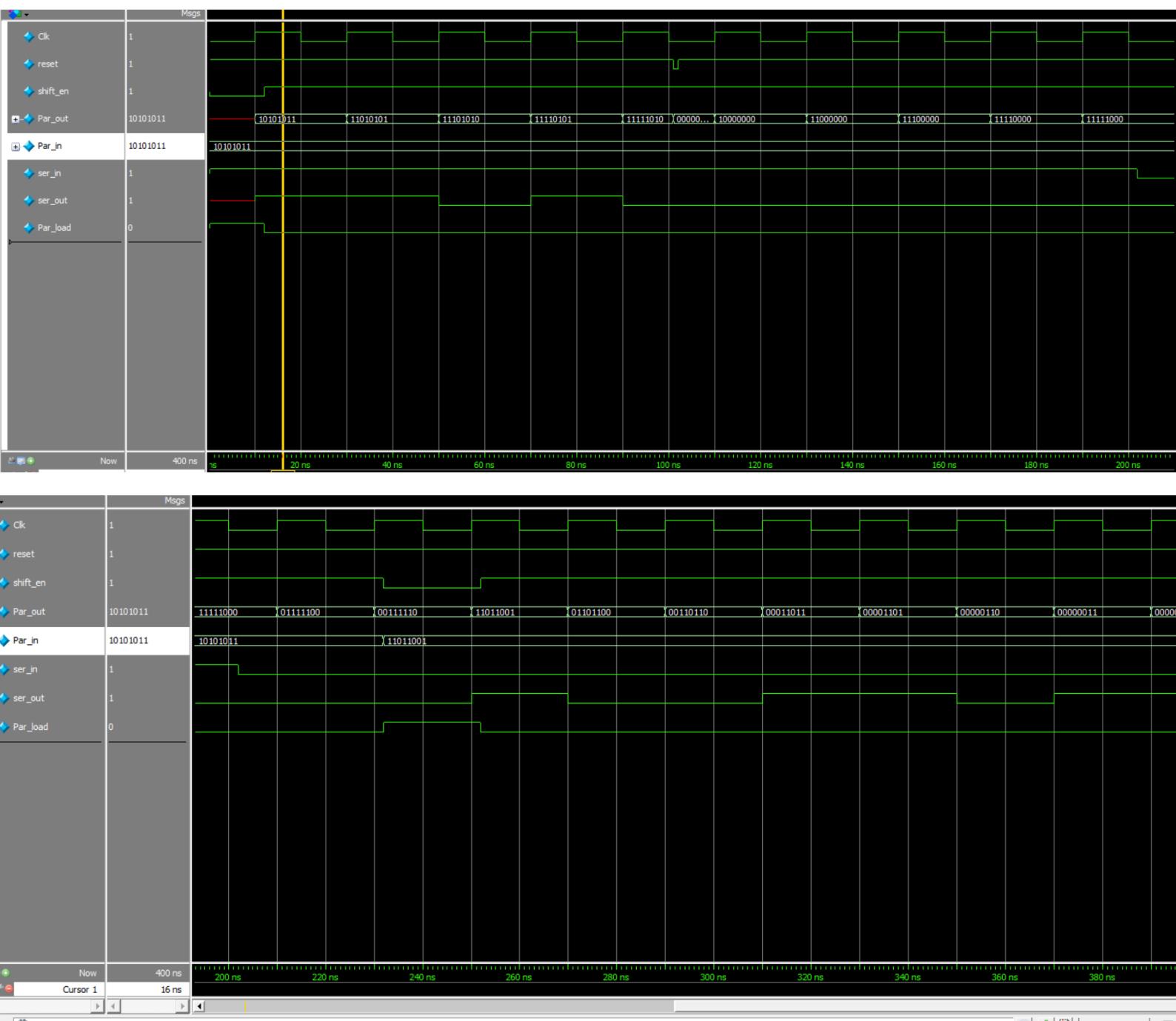
1 `timescale 1ns/1ns
2 module shift_register_tb;
3     logic Clk=0;
4     logic reset=1;
5     logic shift_en;
6     logic [0:7] Par_out;
7     logic [0:7] Par_in=8'b10101011;
8     logic ser_in;
9     logic ser_out;
10    logic Par_load;
11
12    shift_register sr8(Clk, ser_in,reset ,Par_load,shift_en,Par_in, Par_out,ser_out);
13
14
15    initial begin
16        repeat(40) #10 Clk=~Clk;
17    end
18    initial begin
19        shift_en=0;
20        Par_load=1;
21        #12
22        shift_en=1;
23        Par_load=0;
24        #220
25        shift_en=0;
26        Par_load=1;
27        Par_in=8'b11011001;
28        #20      //because change with sync clock
29        shift_en=1;
30        Par_load=0;
31
32    end
33    initial begin
34        ser_in=1;
35        #101
36        reset=0;
37        #1
38        reset=1;
39        #100
40        ser_in=0;
41    end
42
43 endmodule
44

```

موقع تغییر سیگنال لود و شیفت حواسمن است که زمان کافی را بین آن فاصله بگذاریم تا فرصت داشته باشد به سیکل بعد برسد و در آن جا در رجیستر ذخیره شود.

همیشه `ser_out` بیت آخر رجیستر است.

شبیه سازی را با $N=8$ چک می کنیم تا صحت عملکردش اطمینان پیدا کنیم.



از آنجایی که این ماژول را بر اساس پارامتر N تشکیل دادیم موقع اینستنس گرفتن انتخاب N دلخواه است و میتوان به راحتی شیفت رجیستر 24 و 80 بیتی ساخت. ولی در حالت دیفالت برابر 8 بیت است.

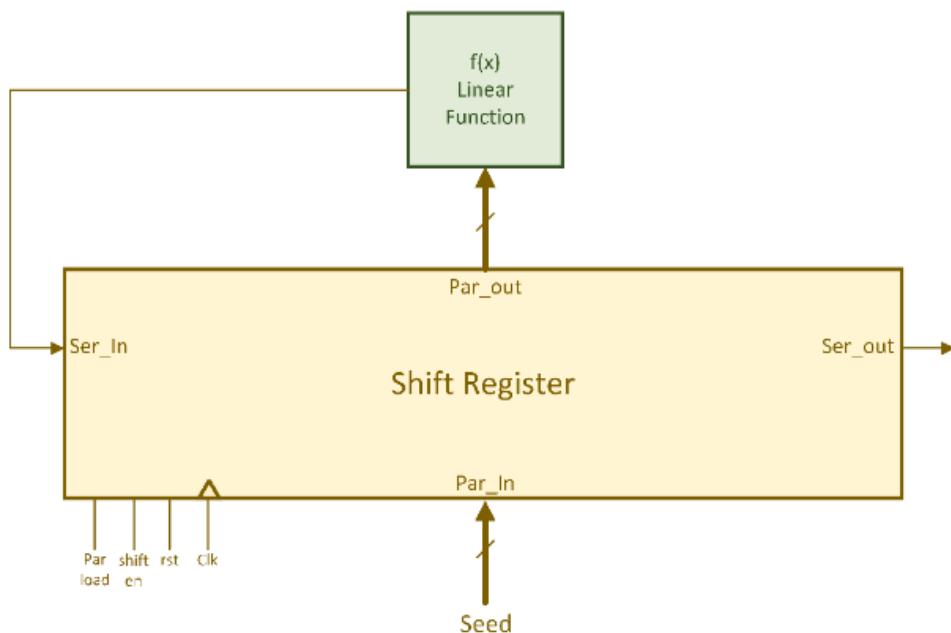
LFSR

در این بخش به کمک یک شیفت رجیستر 8 بیتی

این ماژول را طراحی می کنیم که به آن وارد می شود سریال ورودی است و براساس xor خطی زیر معلوم می شود:

LFSR

ماژول LFSR را مطابق شکل ۴ طراحی کنید. این ماژول از یک شیفت‌رジستر ۸۰ بیتی به همراه تابع خطی f تشکیل شده است:



شکل ۴- ماژول LFSR

$$f(X) = X_{62} \oplus X_{51} \oplus X_{38} \oplus X_{23} \oplus X_{13} \oplus X_0$$

کد سیستم وریلگ:

```
1 module LFSR (input Clk,reset ,Par_load,shift_en,input [79:0] SEED,output [79:0] X,output ser_out);
2
3 logic ser_in;
4 shift_register #(80) sr80(Clk, ser_in,reset ,Par_load,shift_en,SEED,X,ser_out);
5 xor(sr_in,X[62],X[51],X[38],X[23],X[13],X[0]);
6
7
8 endmodule
```

برای تست کردن از دوتا ورودی 80 بیتی باید استفاده کنیم. برای راحتی کار از کد hexadecimal آن استفاده میکنیم که به جای 80 رقم کافی است که 20 رقم قرار دهیم.

```
1 `timescale 1ns/1ns
2 module LFSR_tb;
3     logic Clk=0;
4     logic reset=1;
5     logic shift_en;
6
7     logic [79:0] X;
8     logic [79:0] SEED=80'h123456789ABCDEF12345;
9     logic ser_out;
10    logic Par_load;
11
12    LFSR lfsr80(Clk,reset ,Par_load,shift_en,SEED,X,ser_out);
13
14
15    initial begin
16        repeat(50) #10 Clk=~Clk;
17    end
18    initial begin
19        shift_en=0;
20        Par_load=1;
21        #12
22        shift_en=1;
23        Par_load=0;
24        #250
25        shift_en=0;
26        Par_load=1;
27        SEED=80'h114313ecba9118200465;
28        #12
29        shift_en=1;
30        Par_load=0;
31
32    end
33
34 endmodule
35
```

دو تا مقدار مختلف به عنوان SEED ابتدا 'h123456789ABCDEF12345' میدهیم

سپس: h114313ecba9118200465:

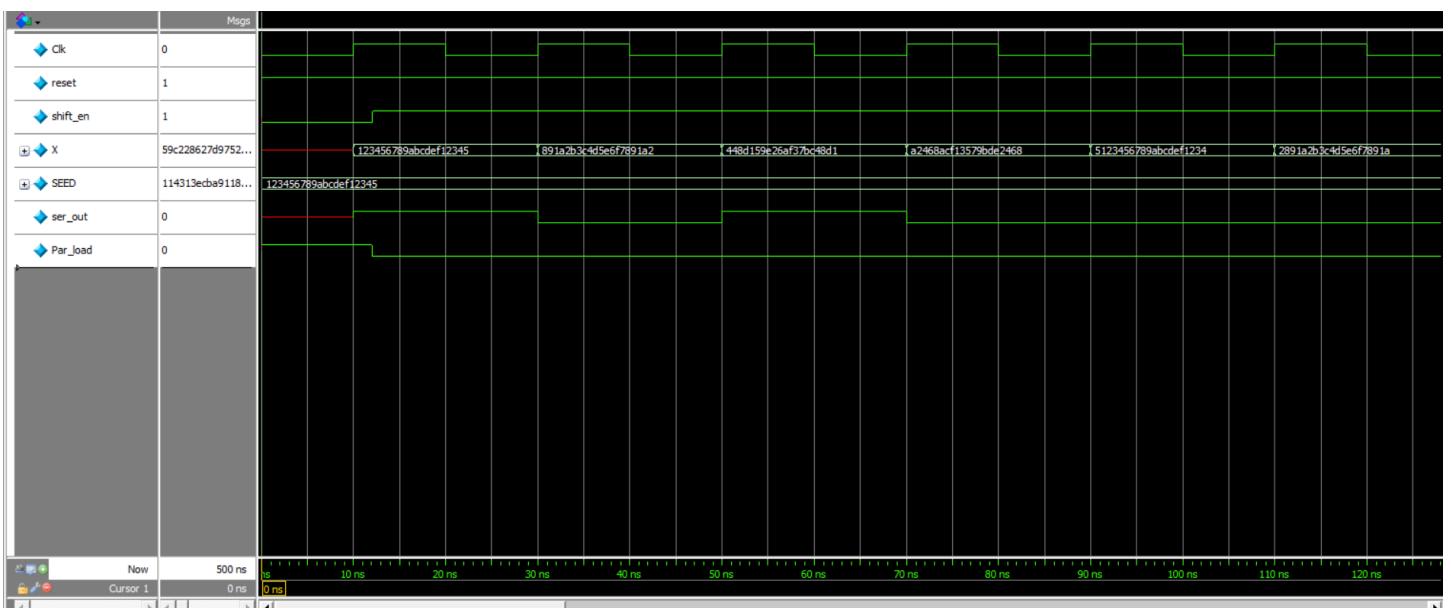
شبیه سازی میکنیم و سپس با کد محک چک میکنیم:

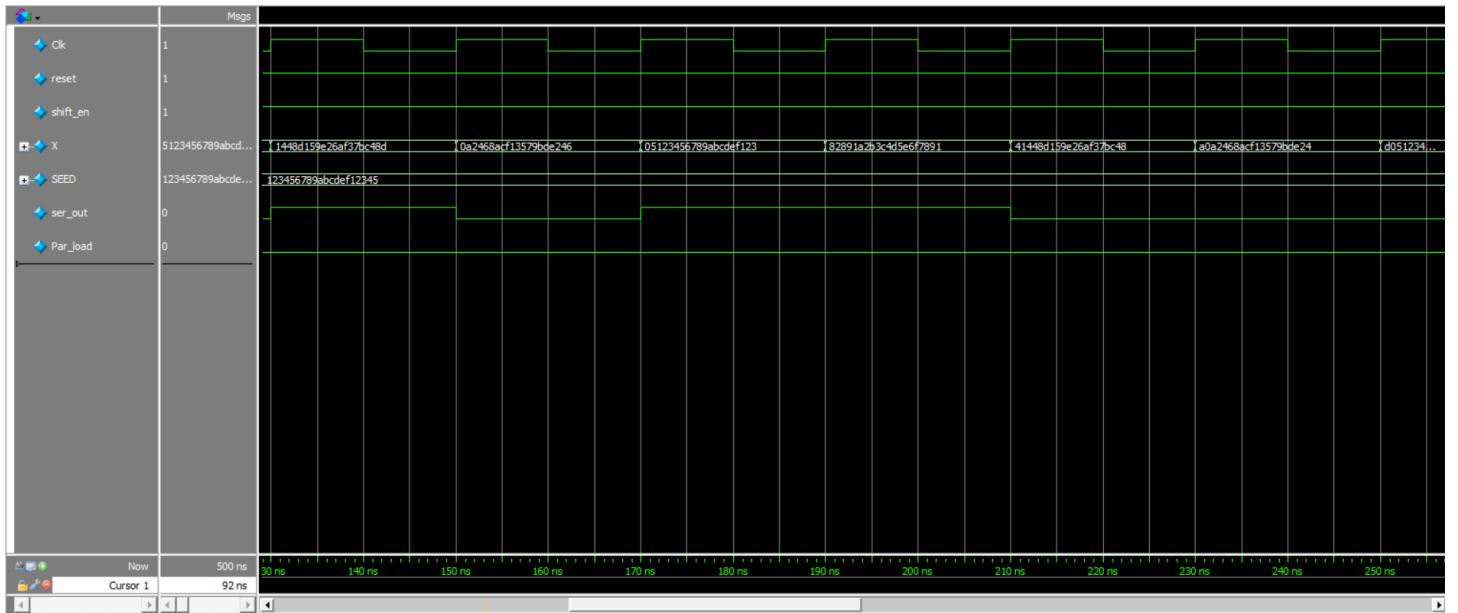
اول SEED :

```
1 , 0x123456789abcdef12345
0 , 0x891a2b3c4d5e6f7891a2
1 , 0x448d159e26af37bc48d1
0 , 0xa2468acf13579bde2468
0 , 0x5123456789abcdef1234
0 , 0x2891a2b3c4d5e6f7891a
1 , 0x1448d159e26af37bc48d
0 , 0x0a2468acf13579bde246
1 , 0x05123456789abcdef123
1 , 0x82891a2b3c4d5e6f7891
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS bash

```
sult : 1
sult : 0
sult : 0
sult : 1
sult : 1
sult : 0
sult : 0
sult : 1
sult : 0
sult : 1
mohammad-amin@mohammad-amin-1-2:~/Downloads/DLD-CA3-Grain-Simulation/DLD-CA3---Fall-2024---Grain-Simulation$ ./grain -l 123456789abcdef12345 10
```





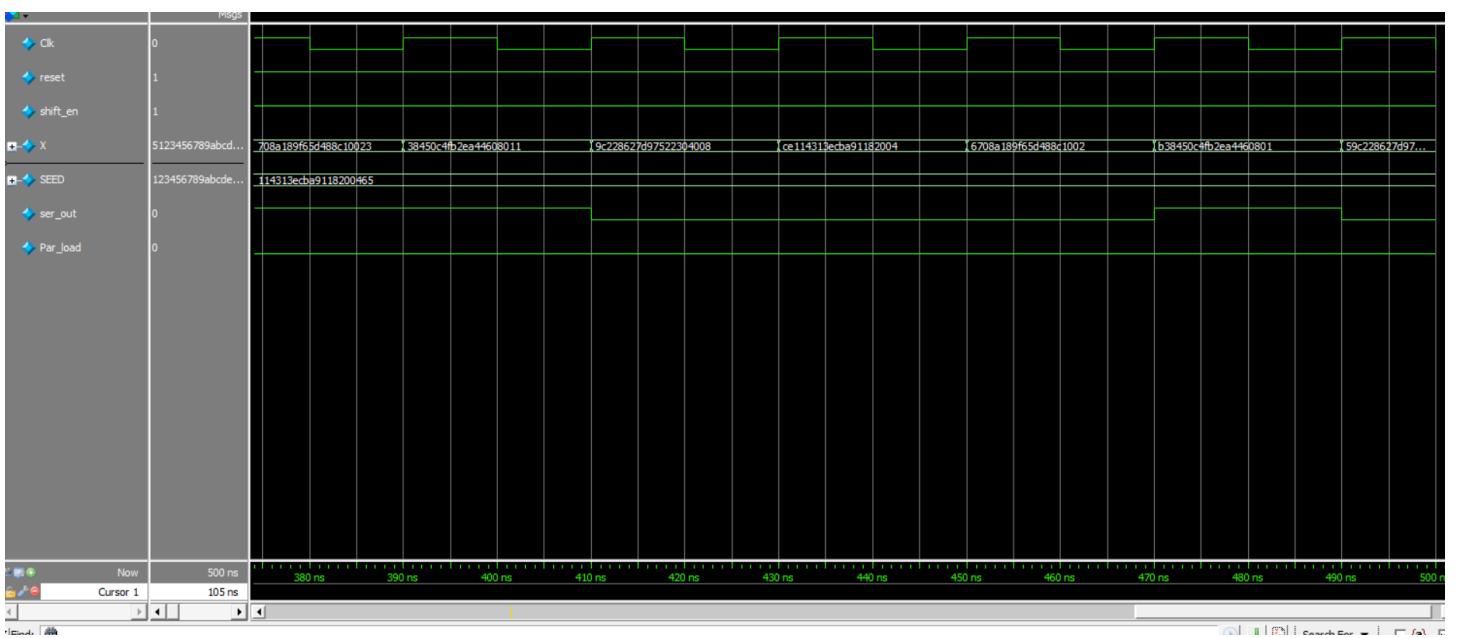
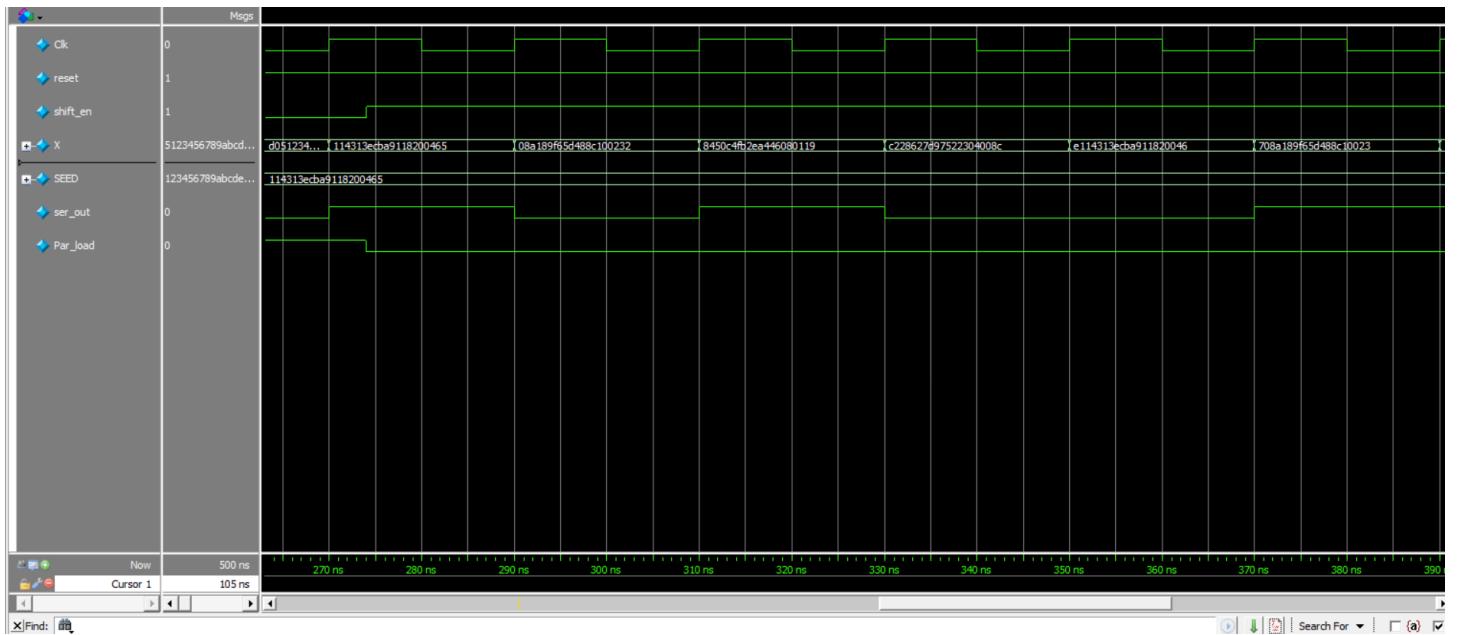
پروتکل SEED :

```

CA3---FALL-2024---GRAIN-SIMULATION  output > LFSR.log
Code
uid
utput
Grain.log
LFSR.log
LFSR.h
Grain
Grain.cpp
Grain.h
LFSR
LFSR.cpp
LFSR.h
NFSR
NFSR.cpp
NFSR.h
uint24
uint80
main.cpp
ignore
ry
ein
akefile
README.md

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
result : 1
result : 0
result : 0
result : 1
result : 1
result : 0
result : 0
result : 1
result : 0
result : 1
● mohammad-amin@mohammad-amin-1-2:~/Downloads/DLD-CA3-Grain-Simulation/DLD-CA3---Fall-2024---Grain-Simulation$ ./grain -l 114313ecba9118200465 10

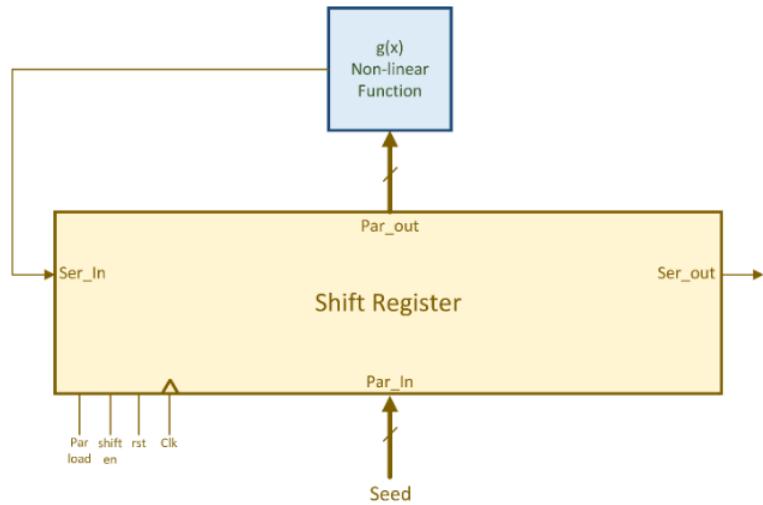
```



تمامی خروجی ها مطابقت دارند.

NFSR:

مشابه قسمت قبل پیاده سازی میشود با این تفاوت که تابع feedback غیرخطی میشود و شیفت رجیستر 24 بیتی میشود:



شکل ۵- مازول NFSR

$$g(X) = X_0 \oplus X_5 \oplus X_6 \oplus X_9 \oplus X_{17} \oplus X_{22} \oplus (X_4 \cdot X_{13}) \oplus (X_8 \cdot X_{16}) \oplus (X_5 \cdot X_{11} \cdot X_{14}) \\ \oplus (X_2 \cdot X_5 \cdot X_8 \cdot X_{10})$$

```

NFSR.sv
1 module NFSR (input Clk,reset ,Par_load,shift_en, ser_out_lfsr,input [23:0] SEED,output [23:0] X,output ser_out);
2 //adding ser_out_lfsr for Grain Structure
3 logic ser_in;
4 logic w1,w2,w3,w4,g;
5 shift_register #(24) sr24(Clk, ser_in,reset ,Par_load,shift_en,SEED,X,ser_out);
6 xor(g,X[0],X[5],X[6],X[9],X[17],X[22],w1,w2,w3,w4);
7 and(w1,X[4],X[13]);
8 and(w2,X[8],X[16]);
9 and(w3,X[5],X[11],X[14]);
10 and(w4,X[2],X[5],X[8],X[10]);
11
12 xor(ser_in, g, ser_out_lfsr);
13
14 endmodule

```

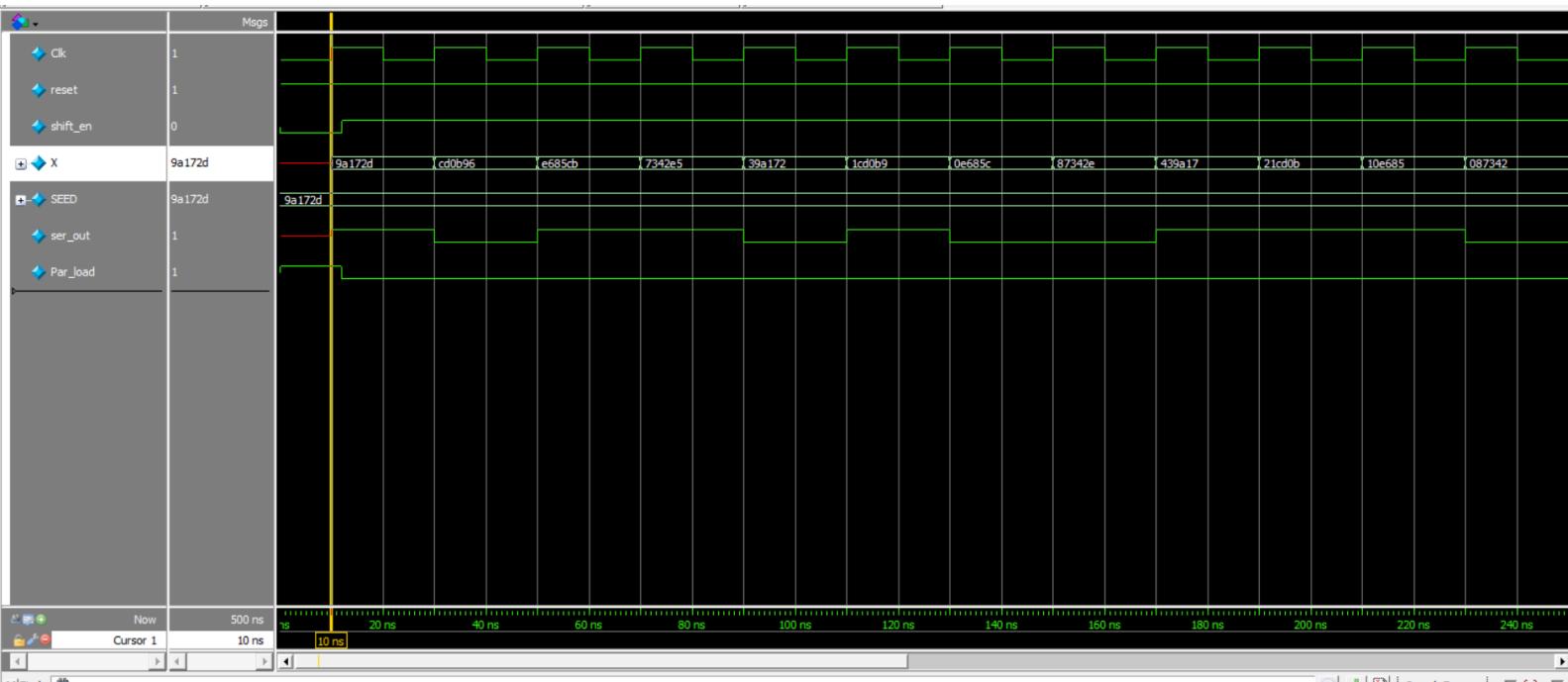
اینکه ser_out_lfsr را به این مازول وصل میکنم برای بخش پیاده سازی grain است که در ادامه خواهیم دید.

NFSR_tb.sv

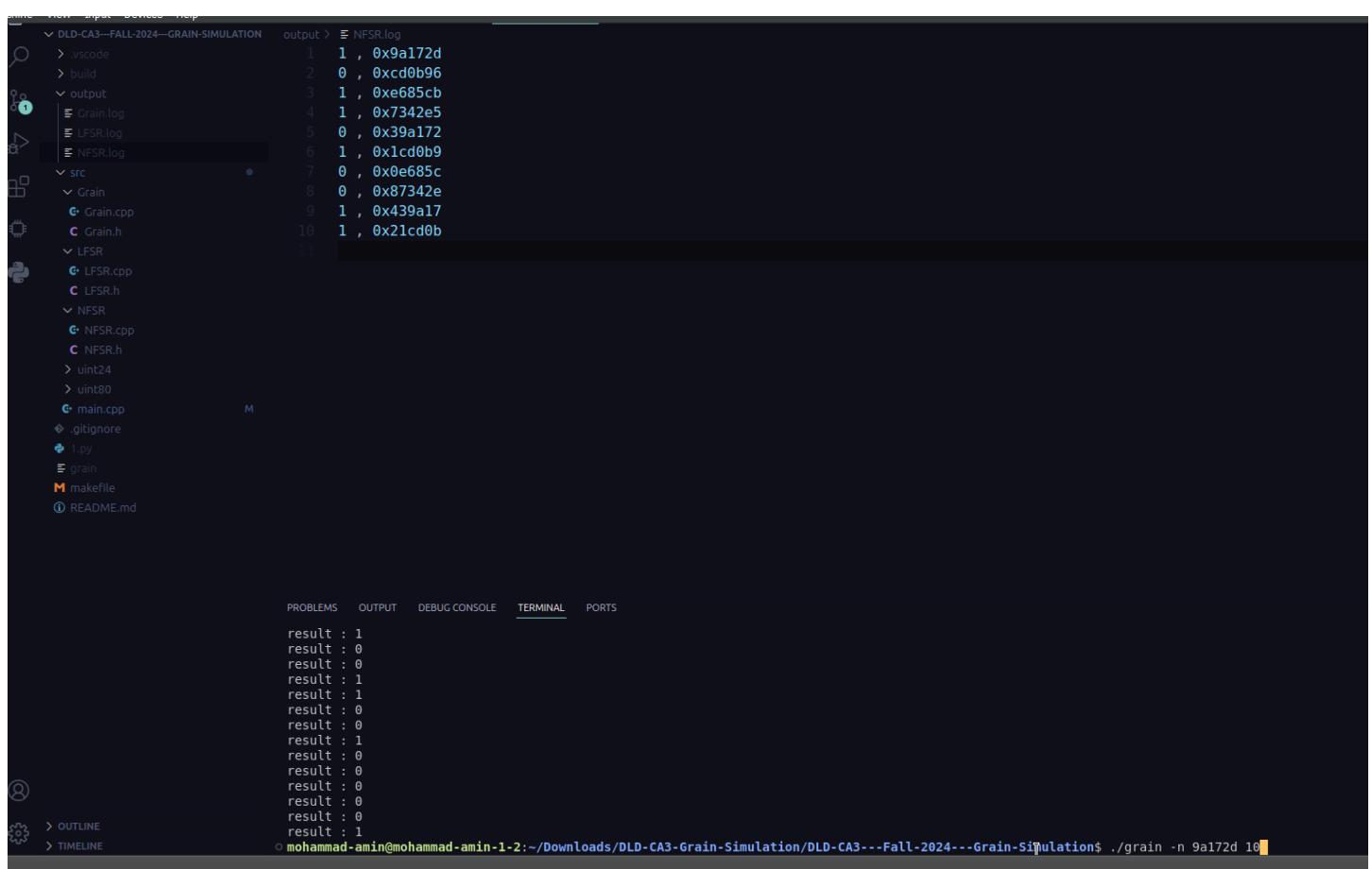
```
1 `timescale 1ns/1ns
2 module NFSR_tb;
3     logic Clk=0;
4     logic reset=1;
5     logic shift_en;
6
7     logic [23:0] X;
8     logic [23:0] SEED=24'h9a172d;
9     logic ser_out;
10    logic Par_load;
11
12    NFSR nfsr24 (Clk,reset ,Par_load,shift_en,0,SEED,X,ser_out);
13
14
15    initial begin
16        repeat(50) #10 Clk=~Clk;
17    end
18    initial begin
19        shift_en=0;
20        Par_load=1;
21        #12
22        shift_en=1;
23        Par_load=0;
24        #250
25        shift_en=0;
26        Par_load=1;
27        SEED=80'he6720b;
28        #12
29        shift_en=1;
30        Par_load=0;
31
32    end
33
34
35 endmodule
36
37
```

سید اول برابر هگزادسیمال: 9a172d

ابتدا شبیه سازی میکنیم.

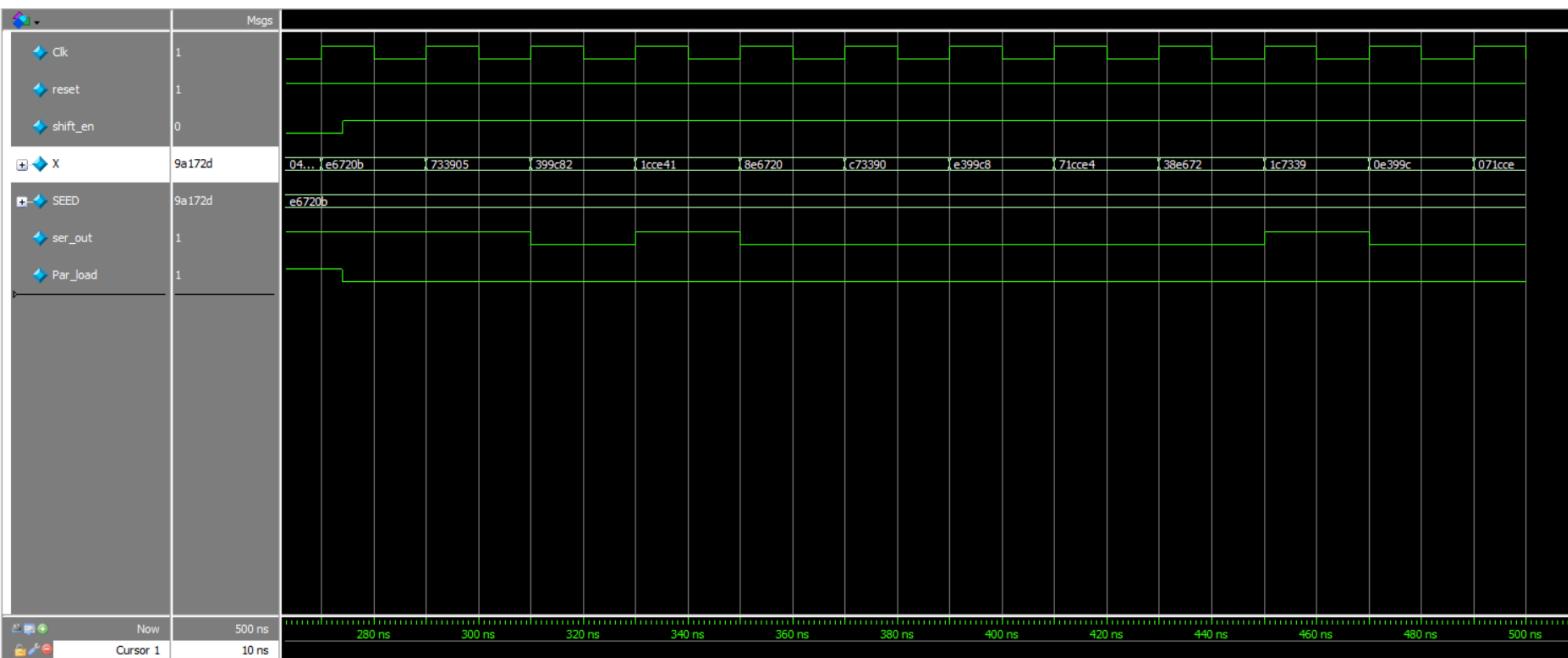


کد محک:



سید دوم برابر هگزادسیمال e6720b

ابتدا شبیه سازی میکنیم.



کدمحک:

```
GRAIN-SIMULATION output > NFSR.log
 1 1 , 0xe6720b
 2 1 , 0x733905
 3 0 , 0x399c82
 4 1 , 0x1cce41
 5 0 , 0x8e6720
 6 0 , 0xc73390
 7 0 , 0xe399c8
 8 0 , 0x71cce4
 9 0 , 0x38e672
10 1 , 0x1c7339
11

M

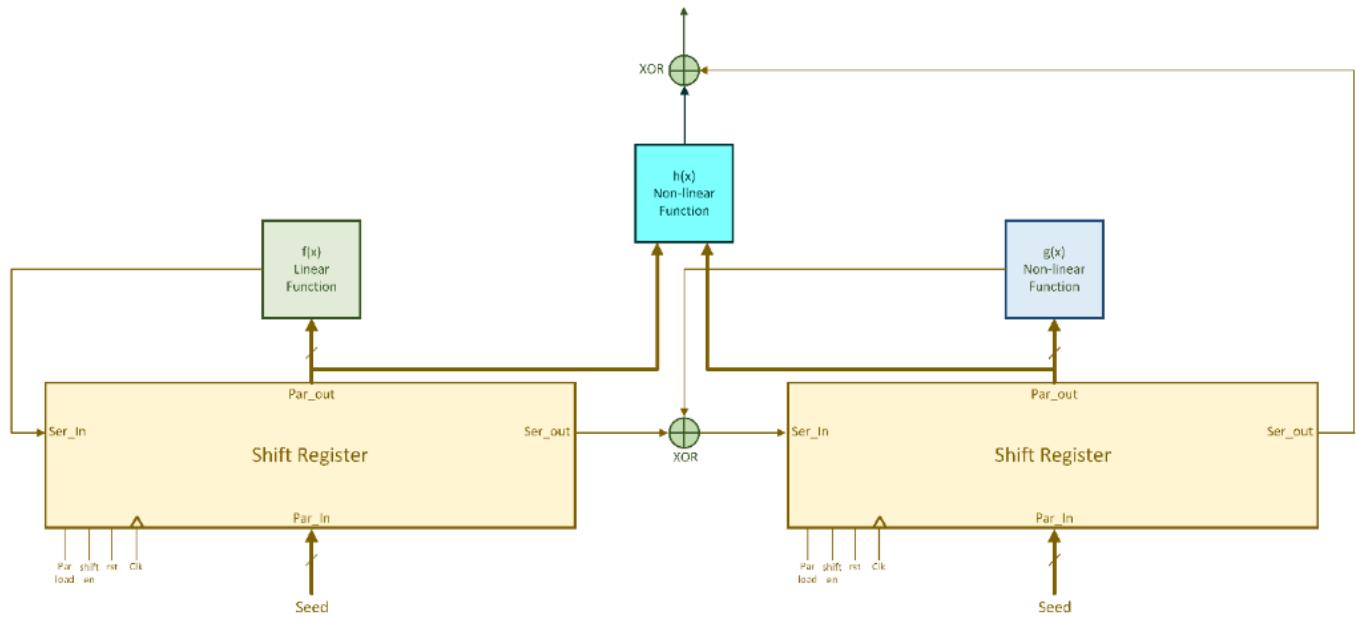
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS
result : 1
result : 0
result : 0
result : 1
result : 1
result : 0
result : 0
result : 1
result : 0
result : 1
mohammad-amin@mohammad-amin-1-2:~/Downloads/DLD-CA3-Grain-Simulation/DLD-CA3---Fall-2024---Grain-Simulation$ ./grain -n e6720b 10
```

با زهم کد های خروجی تطابق کامل دارند.

Grain:

در این بخش یک ماژول جدید تعریف میکنیم که ساختار ترکیبی زیر داشته باشد:

$$h(X_L, X_N) = X_{L0} \oplus X_{L3} \oplus (X_{L1} \cdot X_{L2}) \oplus X_{N0} \oplus (X_{N1} \cdot X_{L5}) \oplus (X_{N3} \cdot X_{L7}) \\ \oplus (X_{L8} \cdot X_{L13} \cdot X_{N5}) \oplus X_{N2}$$



شکل ۶- ماژول Grain

پس طراحی آن را به شکل زیر انجام میدهیم:

```
Grain.sv
1 module Grain(input Clk,reset ,Par_load,shift_en,input [79:0] SEED_1,input [23:0]SEED_n,output [79:0] X_1,output [23:0] X_n,output main_output);
2
3     wire ser_out_lfsr,ser_out_nfsr;
4     wire h,w1,w2,w3,w4;
5
6     LFSR lfsr80(Clk,reset ,Par_load,shift_en,SEED_1,X_1,ser_out_lfsr);
7     NFSR nfsr24 (Clk,reset ,Par_load,shift_en,ser_out_lfsr,SEED_n,X_n,ser_out_nfsr);
8
9
10
11    xor(h,X_1[0],X_1[3],X_n[0],X_n[2],w1,w2,w3,w4);
12    and(w1,X_1[1],X_1[2]);
13    and(w2,X_n[1],X_1[5]);
14    and(w3,X_n[3],X_1[7]);
15    and(w4,X_1[8],X_1[13],X_n[5]);
16
17    xor(main_output,h,ser_out_nfsr);
18
19
20
21 endmodule
```

که در آن از یک lfsr و nfsr اینستنس میگیریم و مدار ترکیبی را برای ایجاد تولید سیگنال $h(X_1, X_n)$

ایجاد می کنیم و در نهایت `h` را به سریال خروجی `nfsr` ایکس اور میکنیم و در `put` میریزیم.

ورودی `nfsr` و `lfsr` را به صورت مجزا به ماژول هایشان میدهیم.

اکنون تست بنچ مینویسیم:

```
Grain_tb.sv
1 `timescale 1ns/1ns
2 module Grain_tb;
3   logic Clk=0;
4   logic reset=1;
5   logic shift_en;
6   logic [23:0] X_n;
7   logic [23:0] SEED_n=24'h9a172d;
8   logic [79:0] X_1;
9   logic [79:0] SEED_l=80'h123456789ABCDEF12345;
10  logic main_output;
11  logic Par_load;
12
13  Grain grain(Clk,reset ,Par_load,shift_en, SEED_l , SEED_n , X_1 , X_n, main_output);
14
15 initial begin
16   repeat(40) #10 Clk=~Clk;
17 end
18 initial begin
19   shift_en=0;
20   Par_load=1;
21   #12
22   shift_en=1;
23   Par_load=0;
24 end
25
26 initial begin
27   #200
28   shift_en=0;
29   Par_load=1;
30   SEED_n=24'h313ec8;
31   SEED_l=80'h114313ecba9118200465;
32   #12
33   shift_en=1;
34   Par_load=0;
35 end
36 endmodule
37
38
39
```

شبیه سازی موج ها:

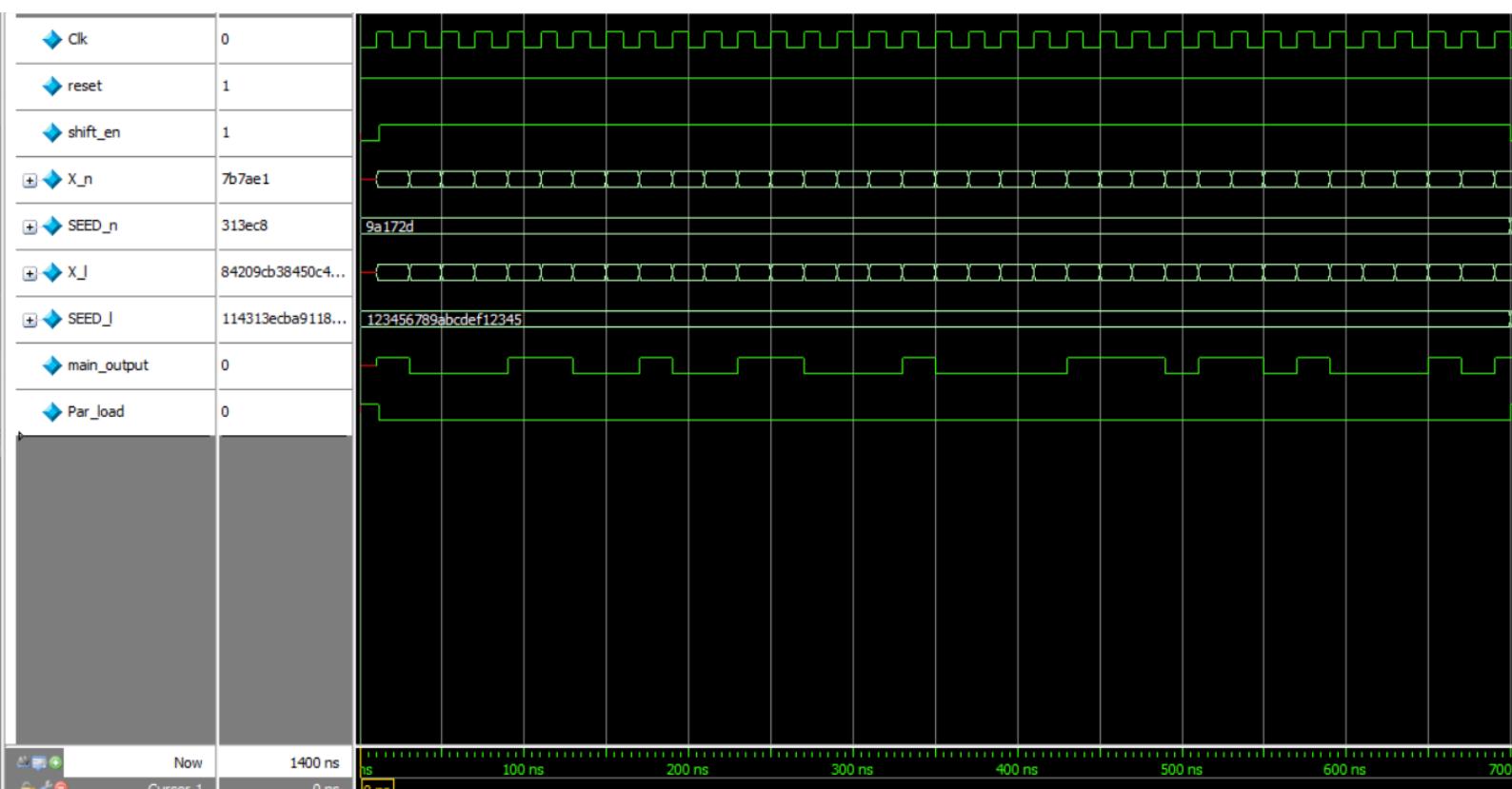
```

# Grain_tb.sv
1 `timescale 1ns/1ns
2 module Grain_tb;
3   logic Clk=0;
4   logic reset=1;
5   logic shift_en;
6   logic [23:0] X_n;
7   logic [23:0] SEED_n=24'h9a172d;
8   logic [79:0] X_l;
9   logic [79:0] SEED_l=80'h123456789ABCDEF12345;
10  logic main_output;
11  logic Par_load;
12
13  Grain grain(Clk,reset ,Par_load,shift_en, SEED_l , SEED_n , X_l , X_n, main_output);
14
15 initial begin
16   repeat(140) #10 Clk=~Clk;
17 end
18 initial begin
19   shift_en=0;
20   Par_load=1;
21   #12
22   shift_en=1;
23   Par_load=0;
24 end
25
26 initial begin
27
28   #700
29   shift_en=0;
30   Par_load=1;
31   SEED_n=24'h313ec8;
32   SEED_l=80'h114313ecba9118200465;
33   #12
34   shift_en=1;
35   Par_load=0;
36 end
37 endmodule
38

```

سید اول:

از 0 نا 700 نانو ثانیه با حداقل 32 خروجی



کد محک:

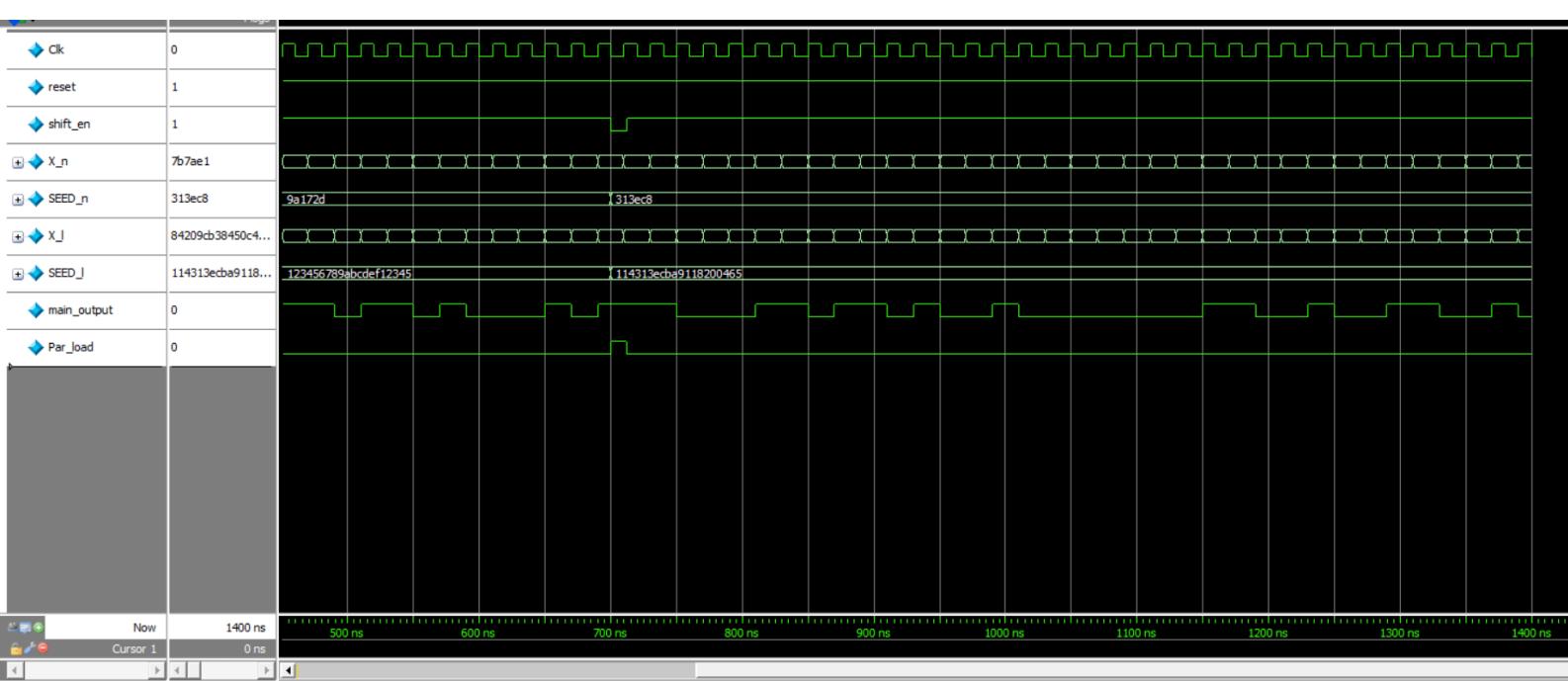
```
Run Terminal Help
LFSR.log main.cpp Grain.log 1.py
output > Grain.log
1 , 0x123456789abcdef12345 , 0x9a172d
2 , 0x891a2b3c4d5e6f7891a2 , 0x4d0b96
3 , 0 , 0x448d159e26af37bc48d1 , 0xa685cb
4 , 0 , 0xa2468acf13579bde2468 , 0x5342e5
5 , 1 , 0x5123456789abcdef1234 , 0x29a172
6 , 1 , 0x2891a2b3c4d5e6f7891a , 0x14d0b9
7 , 0 , 0x1448d159e26af37bc48d , 0x0a685c
8 , 0 , 0x0a2468acf13579bde246 , 0x05342e
9 , 1 , 0x05123456789abcdef123 , 0x829a17
10 , 0 , 0x82891a2b3c4d5e6f7891 , 0x414d0b
11 , 0 , 0x41448d159e26af37bc48 , 0x20a685
12 , 1 , 0xa0a2468acf13579bde24 , 0x105342
13 , 1 , 0xd05123456789abcdef12 , 0x0829a1
14 , 0 , 0x682891a2b3c4d5e6f789 , 0x0414d0
15 , 0 , 0x341448d159e26af37bc4 , 0x020a68
16 , 0 , 0x1a0a2468acf13579bde2 , 0x010534
17 , 1 , 0x8d05123456789abcdef1 , 0x80829a
18 , 0 , 0xcc682891a2b3c4d5e6f78 , 0x40414d
19 , 0 , 0x6341448d159e26af37bc , 0xa020a65
20 , 0 , 0x31a0a2468acf13579bde , 0xd01053
21 , 0 , 0x98d005123456789abcdef , 0xe80829
22 , 1 , 0x4c682891a2b3c4d5e6f7 , 0x740414
23 , 1 , 0xa6341448d159e26af37b , 0x3a020a
24 , 1 , 0x531a0a2468acf13579bd , 0x9d0105
25 , 0 , 0x298d05123456789abcde , 0xce8082
26 , 1 , 0x94c682891a2b3c4d5e6f , 0x674041
27 , 1 , 0x4a6341448d159e26af37 , 0xb3a020
28 , 0 , 0xa531a0a2468acf13579b , 0xd9d010
29 , 1 , 0xd298d05123456789abcd , 0x6ce808
30 , 0 , 0xe94c682891a2b3c4d5e6 , 0x367404
31 , 0 , 0xf4a6341448d159e26af3 , 0x9b3a02
32 , 0 , 0x7a531a0a2468acf13579 , 0xcd9d01
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

LFSR seed : 123456789abcdef12345
mohammad-amin@mohammad-amin-1-2:~/Downloads/DLD-CA3-Grain-Simulation/DLD-CA3---Fall-2024---Grain-Simulation\$./grain -g 9a172d123456789abcdef12345 32
NESR seed : 9a172d

سید دوم:

از 700 نانو ثانیه تا 1400 نانو ثانیه با حداقل 32 سیکل



```

output > Grain.log
1 , 0x114313ecba9118200465 , 0x313ec8
2 , 0x08a189f65d488c100232 , 0x989f64
3 , 0x8450c4fb2ea446080119 , 0x4c4fb2
4 , 0 , 0xc228627d97522304008c , 0xa627d9
5 , 0 , 0xe114313ecba911820046 , 0xd313ec
6 , 0 , 0x708a189f65d488c10023 , 0x6989f6
7 , 0 , 0x38450c4fb2ea44608011 , 0xb4c4fb
8 , 0 , 0x9c228627d97522304008 , 0x5a627d
9 , 1 , 0xce114313ecba91182004 , 0xad313e
10 , 1 , 0x6708a189f65d488c1002 , 0xd6989f
11 , 0 , 0xb38450c4fb2ea4460801 , 0xeb4c4f
12 , 1 , 0x59c228627d9752230400 , 0xf5a627
13 , 0 , 0x2ce114313ecba9118200 , 0x7ad313
14 , 0 , 0x96708a189f65d488c100 , 0x3d6989
15 , 1 , 0xcb38450c4fb2ea446080 , 0xleb4c4
16 , 0 , 0xe59c228627d975223040 , 0x0f5a62
17 , 0 , 0x72ce114313ecba911820 , 0x87ad31
18 , 0 , 0x396708a189f65d488c10 , 0xc3d698
19 , 0 , 0x9cb38450c4fb2ea44608 , 0xe1eb4c
20 , 0 , 0x4e59c228627d97522304 , 0x70f5a6
21 , 0 , 0x272ce114313ecba91182 , 0xb87ad3
22 , 0 , 0x1396708a189f65d488c1 , 0x5c3d69
23 , 1 , 0x09cb38450c4fb2ea4460 , 0xae1eb4
24 , 1 , 0x04e59c228627d9752230 , 0xd70f5a
25 , 0 , 0x8272ce114313ecba9118 , 0xeb87ad
26 , 0 , 0x41396708a189f65d488c , 0xf5c3d6
27 , 1 , 0x209cb38450c4fb2ea446 , 0x7ae1eb
28 , 0 , 0x104e59c228627d975223 , 0xbd70f5
29 , 0 , 0x08272ce114313ecba911 , 0xdeb87a
30 , 1 , 0x841396708a189f65d488 , 0xf5c3d
31 , 1 , 0x4209cb38450c4fb2ea44 , 0xb7ae1e
32 , 0 , 0x2104e59c228627d97522 , 0xdbd70f

```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```

● mohammad-amin@mohammad-amin-1-2:~/Downloads/DLD-CA3-Grain-Simulation/DLD-CA3---Fall-2024---Grain-Simulation$ ./grain -g 313ec8114313ecba9118200465 32
NFSR seed : 313ec8
LFSR seed : 114313ecba9118200465
○ mohammad-amin@mohammad-amin-1-2:~/Downloads/DLD-CA3-Grain-Simulation/DLD-CA3---Fall-2024---Grain-Simulation$ 

```

به ازای تمامی 32 مقادیر خروجی m اصلی صحیح است. هم چنین خروجی موازی $nfsr$ و $lfsr$ صحیح است.

بخش امتیازی:

در این بخش یک تست بنج جدید به صورت زیر مینویسیم و یک SEED را میخوانیم و سپس همانند قبل عمل میکنیم. برای نوشتن در فایل از `fwrite$` استفاده میکنیم. هم چنین برای شمردن تعداد 0ها `state` 1 ها 00 ها 10 ها و 11 ها از متغیر های `integer` برای ذخیره سازی آن ها و از یک متغیر `integer` استفاده میکنیم تا عدد قبلی را در آن ذخیره کینم تا بتوانیم دو بیت متوالی رابه درستی بشمریم.

از متغیر فلگ برای بررسی state اول استفاده میکنیم.

پرینت خروجی فایل ها در هر سیکل انجام میشود در حالی که پرینت count ها در آخر پس از 20000 نانوثانیه صورت میگیرد.

```
bonus_tb.sv
1  `timescale 1ns/1ns
2  module Grain_bonus_tb;
3      logic Clk=0;
4      logic reset=1;
5      logic shift_en;
6      logic [23:0] X_n;
7      logic [23:0] SEED_n=24'h9a172d;
8      logic [79:0] X_l;
9      logic [79:0] SEED_l=80'h123456789ABCDEF12345;
10     logic main_output;
11     logic Par_load;
12     integer file1;
13     integer file2;
14     integer flag=0;
15     integer count_0=0;
16     integer count_1=0;
17     integer count_00=0;
18     integer count_01=0;
19     integer count_10=0;
20     integer count_11=0;
21     logic state;
22     Grain grain(Clk,reset ,Par_load,shift_en, SEED_l , SEED_n , X_l , X_n, main_output);
23
24     initial begin
25         repeat(2000) #10 Clk=~Clk;
26     end
27     initial begin
28         shift_en=0;
29         Par_load=1;
30         file1 = $fopen("main_output.txt", "w");
31         file2 = $fopen("count.txt", "w");
32         #12
33         shift_en=1;
34         Par_load=0;
35     end
36
37     initial begin
38         #20000
39         $fwrite(file2,"count0 = %d\n",count_0);
40         $fwrite(file2,"count1 = %d\n",count_1);
41         $fwrite(file2,"count00 = %d\n",count_00);
```

bonus_tb.sv

```
36
37     initial begin
38         #20000
39         $fwrite(file2,"count0 = %d\n",count_0);
40         $fwrite(file2,"count1 = %d\n",count_1);
41         $fwrite(file2,"count00 = %d\n",count_00);
42         $fwrite(file2,"count01 = %d\n",count_01);
43         $fwrite(file2,"count10 = %d\n",count_10);
44         $fwrite(file2,"count11 = %d\n",count_11);
45     end
46
47     always @(posedge Clk) begin
48
49         $fwrite(file1,"%b\n",main_output);
50     end
51
52     always @(posedge Clk) begin
53         if(flag==0)begin
54             state=main_output;
55             flag=1;
56         end
57         if(main_output==1)begin
58             count_1=count_1+1;
59             if(state==0)begin
60                 count_01=count_01+1;
61
62             end
63             else begin
64                 count_11=count_11+1;
65
66             end
67             state=1;
68         end
69         else if(main_output==0)begin
70
71             count_0=count_0+1;
72             if(state==0)begin
73                 count_00=count_00+1;
74
75             end
76         end
77     end
78 
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS SERIAL MONITOR

```
bonus_tb.sv
52    always @(posedge Clk) begin
53        if(flag==0)begin
54            state=main_output;
55            flag=1;
56        end
57        if(main_output==1)begin
58            count_1=count_1+1;
59            if(state==0)begin
60                count_01=count_01+1;
61            end
62            else begin
63                count_11=count_11+1;
64            end
65        end
66        state=1;
67    end
68    else if(main_output==0)begin
69
70        count_0=count_0+1;
71        if(state==0)begin
72            count_00=count_00+1;
73
74        end
75        else begin
76            count_10=count_10+1;
77
78        end
79    end
80    state=0;
81    end
82
83    end
84    endmodule
85
86
```

خروجی فایل های txt درست و به صورت زیر است:

count.txt

```
1 count0 = 491
2 count1 = 509
3 count00 = 243
4 count01 = 248
5 count10 = 248
6 count11 = 261
7
```

main_output.txt

```
966 0
967 1
968 1
969 0
970 0
971 0
972 0
973 1
974 0
975 1
976 1
977 0
978 0
979 0
980 0
981 0
982 1
983 1
984 1
985 1
986 1
987 1
988 1
989 1
990 1
991 0
992 1
993 0
994 0
995 1
996 1
997 0
998 1
999 1
1000 1
1001
```

The end