ECE 469
Project 2 Part 2
Tamanna Ravi Rupani
665679988
Niranjani Venkatesan
675456032

INTRODUCTION

Following are the components that we have finished (Component 1 – Component 6)

Component 1: LFSR Processor

Understanding the working of the system and designing the SystemVerilog code for the LFSR Processor	3 days
Execution of System Verilog code and debugging errors	2 hours
Completing the table with machine codes	1 hour
Writing new assembly code and machine code	1 hour
Quartus part	30 minutes
Sketch of the processor	1 hour
Making the report for this part	2-3 hours

Component 2: Software Companion

Understanding the working of the system and designing the SystemVerilog to add to the LFSR processor code	1 hour
Converting the logic into software code	5-6 hours
Completing the table with machine codes	1 hour
Making the report for this part	1 hour

Component 3: HD Count

Understanding the working of the system and designing the SystemVerilog to add HD count to the LFSR processor code	1 hour
Execution of System Verilog code and debugging errors	1 hour
Completing the table with machine codes	1 hour
Writing new assembly code and machine code	1 hour
Quartus part	30 minutes
Sketch of the processor	30 minutes
Making the report for this part	2-3 hours

Component 4: Avg HD

Understanding the working of the system and designing the SystemVerilog to add average HD to the LFSR processor code	1 hour
Execution of System Verilog code and debugging errors	1 hour
Completing the table with machine codes	1 hour
Writing new assembly code and machine code	1 hour
Quartus part	30 minutes

Sketch of the processor	30 minutes
Making the report for this part	2 hours

Component 5: Multi-Cycle Run

Understanding the working of the system and designing the SystemVerilog to add Multi-cycle to the LFSR processor code	1 hour
Execution of System Verilog code and debugging errors	1 hour
Completing the table with machine codes	1 hour
Writing new assembly code and machine code	1 hour
Quartus part	30 minutes
Sketch of the processor	30 minutes
Making the report for this part	2-3 hours

Component 6: Batch Mem Storage

Understanding the working of the system and designing the SystemVerilog to add storage to the LFSR processor code	1 hour
Execution of System Verilog code and debugging errors	1 hours
Completing the table with machine codes	1 hour
Writing new assembly code and machine code	1-2 hours
Quartus part	30 minutes
Sketch of the processor	30 minutes
Making the report for this part	2 hours

Component 1: LFSR Processor

i. Provide your instruction encoding scheme (for machine code) for the table above.

Instruction	Functionality	Machine code Encoding	Opcode
config_L tap	setting the tap locations of the 8-bit LFSR according to the 7-bit binary tap	[opcode 31:26] [25:18] [17:10] [9:7] [tap location 6:0]	000001
init_L seed	initializing the LFSR with the 8-bit seed	[opcode 31:26] [25:18] [seed 17:10] [9:7] [6:0]	000010
run_L	let the LFSR run and update to the next pattern	[opcode 31:26] [25:18] [17:10] [9:7] [6:0]	000011
st_M_L	store the current 8-bit LFSR content into M[r_addr]	[opcode 31:26] [25:18] [17:10] [9:7] [6:0]	000100
ld_M_L	load the byte from M[r_addr] as the seed of the LFSR	[opcode 31:26] [25:18] [17:10] [9:7] [6:0]	000101
init_addr loc	initialize r_addr to the 8-bit binary number of loc	[opcode 31:26] [loc 25:18] [17:10] [9:7] [6:0]	000110
add_addr num	allowing pointer to move around: r_addr = r_addr + num 8-bit num can be negative or positive	[opcode 31:26] [25:18] [num 17:10] [9:7] [6:0]	000111
halt	terminate the program (PC no longer increase)	[opcode 31:26] [25:18] [17:10] [9:7] [6:0]	000000

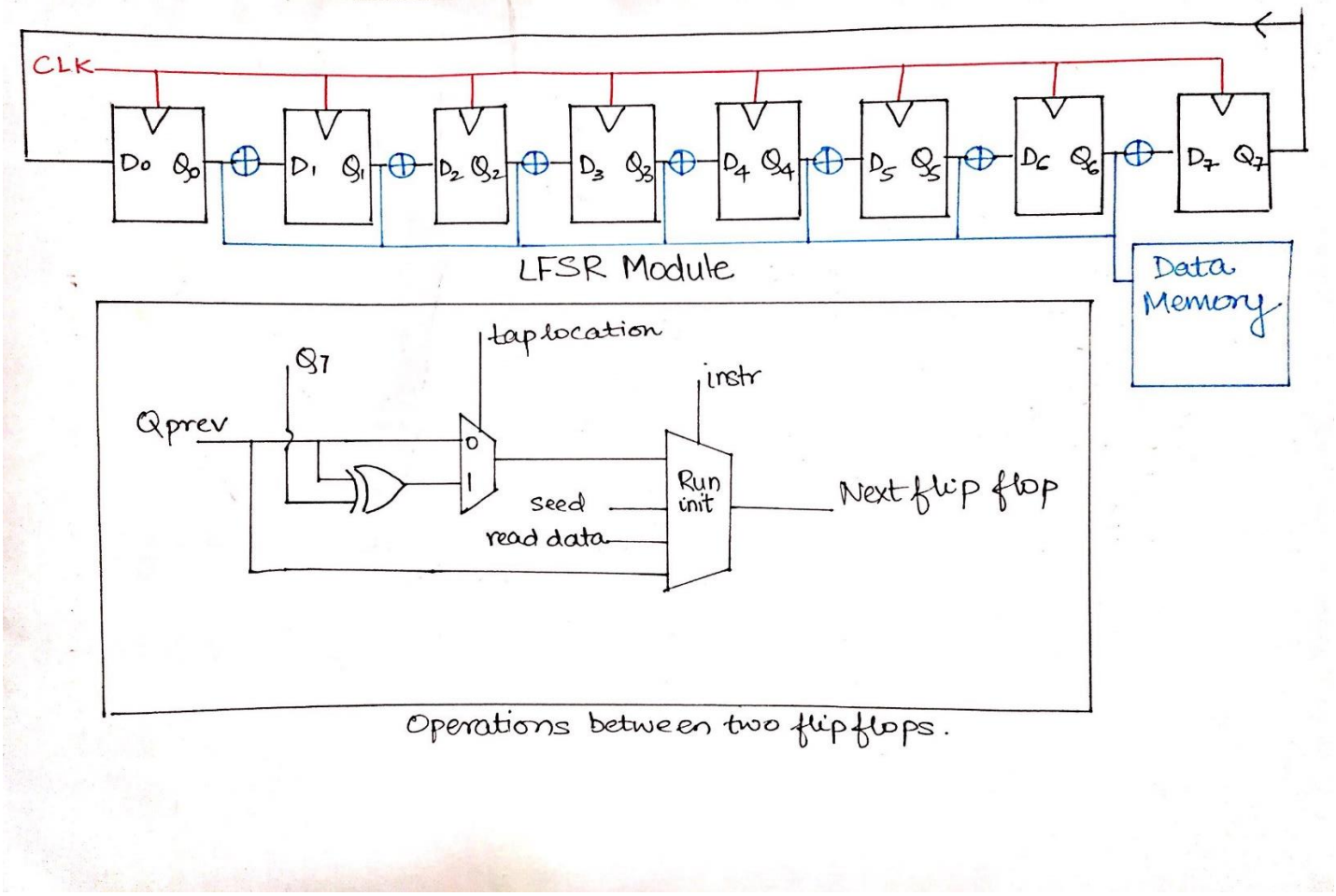
ii. Provide machine code (according to your instruction encoding design) for test program

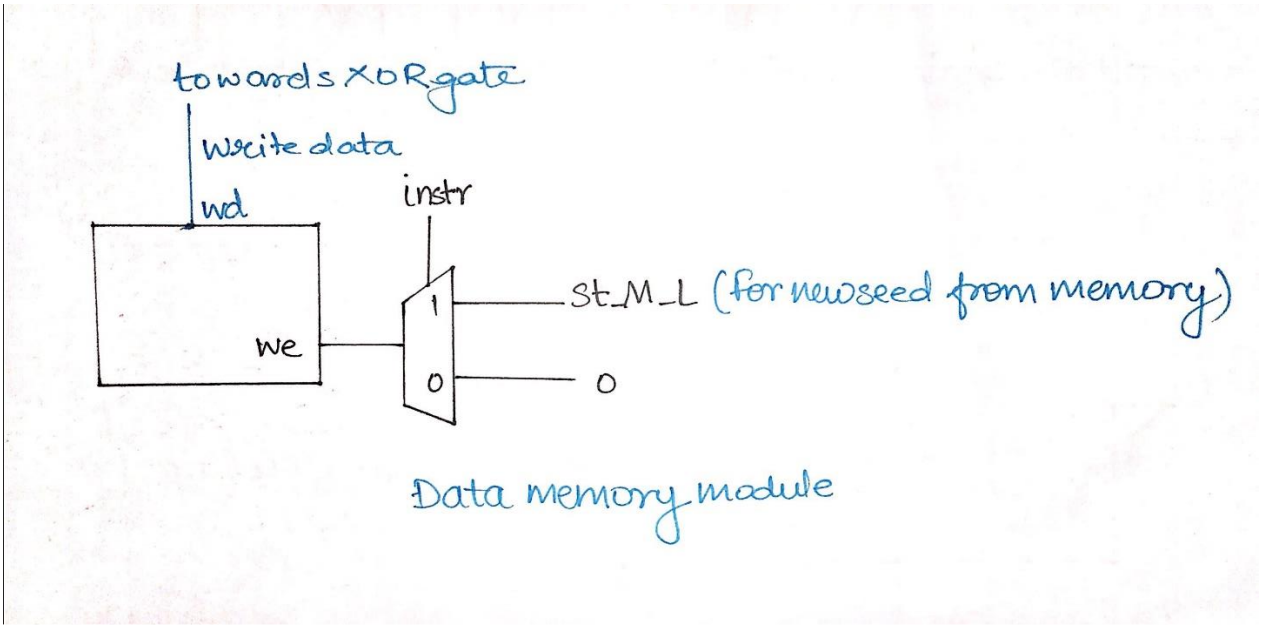
Line #	Assembly Language (lfsr_t1)	Machine Code
1	config_L 0100101;	04000025
2	init_L 11111111;	0803FC00
3	run_L;	0C000000
4	run_L;	0C000000
5	init_addr 00001001;	18240000
6	run_L;	0C000000
7	st_M_L;	10000000
8	run_L;	0C000000
9	add_addr 11111110	1C03F800
10	st_M_L;	10000000
11	add_addr 00000010;	1C000800
12	ld_M_L;	14000000
13	run_L;	0C000000
14	halt;	00000000

iii. Provide another testing program lfsr_t2 of your own design in a table format – use more rows if needed:

Line #	Assembly Language (lfsr_t2)	Machine Code	Comments
1	config_L 1001010;	0400004A	Configures LFSR as shown in the circuit
2	init_L 11111111;	0803FC00	seeds LFSR with 11111111
3	run_L;	0C000000	LFSR = 10110101
4	init_addr 00000100;	18100000	r_addr = 4
5	run_L;	0C000000	LFSR = 10010000
6	run_L;	0C000000	LFSR = 01001000
7	st_M_L;	10000000	M [4] = 01001000
8	run_L;	0C000000	LFSR =00100100
9	add_addr 00001000;	1C002000	r_addr = 4 + 8 = 12
10	st_M_L;	10000000	M [12] = 00100100
11	add_addr 11111000;	1C03E000	r_addr = 12 - 8 = 4
12	ld_M_L;	14000000	LFSR = M [4] = 01001000
13	run_L;	0C000000	LFSR = 00100100
14	halt;	00000000	Stops the program

iv. Provide a sketch of your processor design.



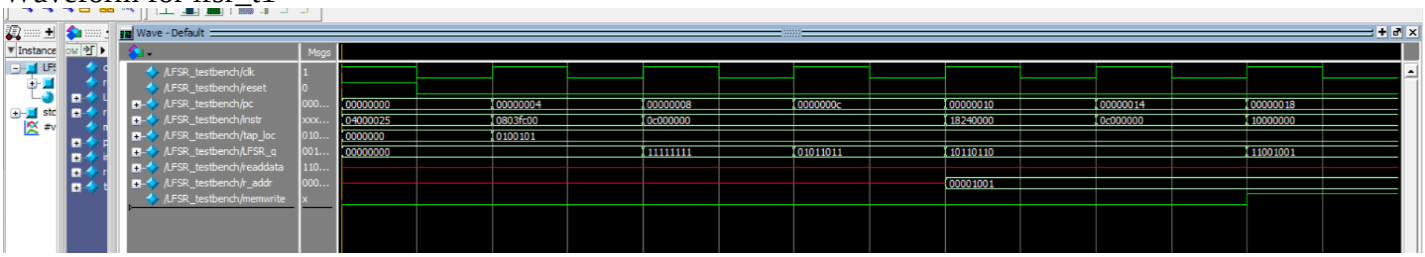


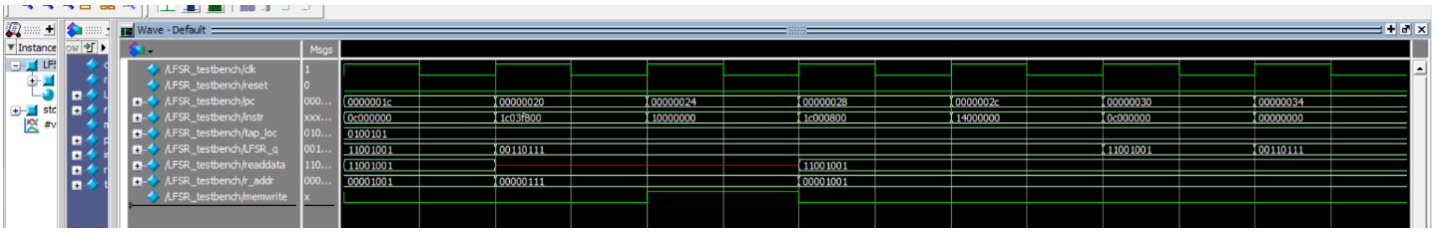
Here we have taken a few modules from single cycle MIPS processor, and added our module for LFSR working. Since there are a few redundant modules which aren't required for this LFSR processors, those modules from MIPS are excluded. The major change is in the datapath, where the LFSR operation takes place.

v. Implement your processor with System Verilog code (put all your code in the appendix part of the report).
(Code added in the APPENDIX 1)

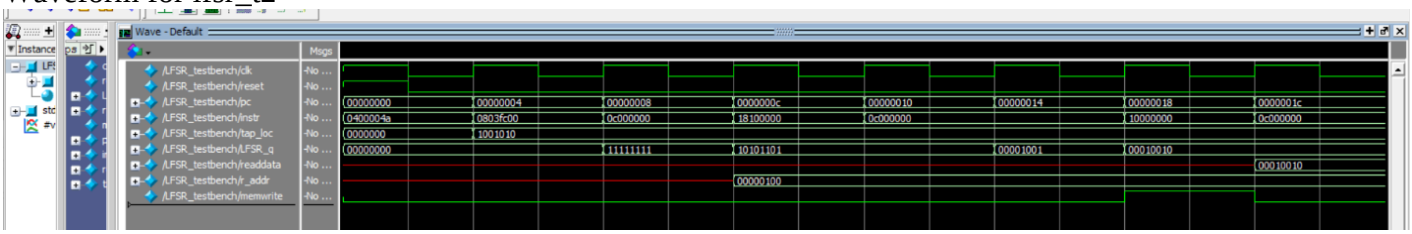
vi. Run both testing programs with ModelSim to verify that your design works correctly. Show the waveform results.

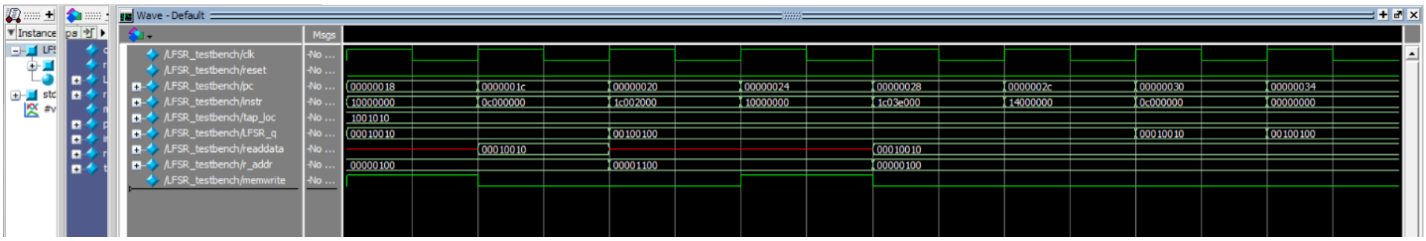
Waveform for lfsr_t1



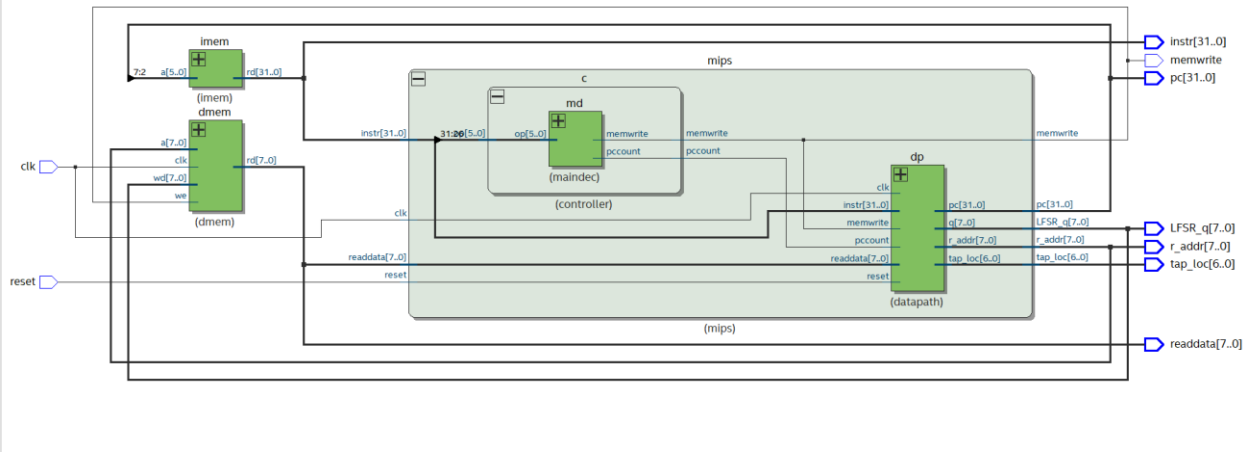


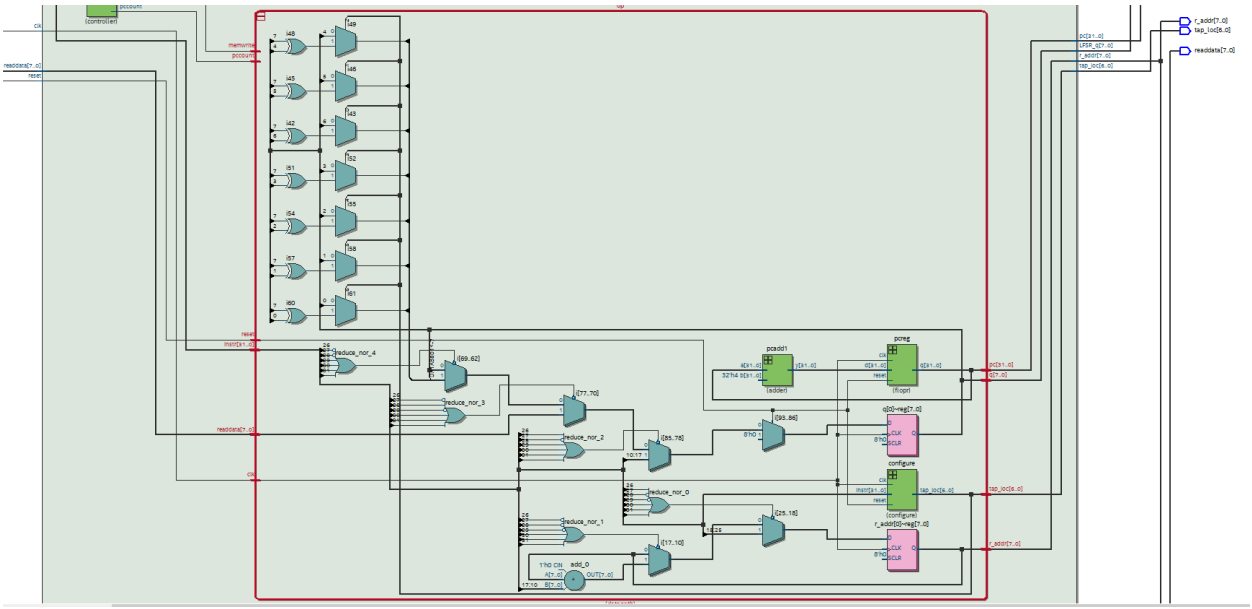
Waveform for lfsr_t2





RTL View




r_add[7..0]
tap_loc[6..0]
readdata[7..0]
pc[31..0]
LPSR_EQ7..0]
r_add[7..0]
tap_loc[6..0]
(controller)
clk
reset
readdata[7..0]
memwrite
pccount
inst[31..0]
reduce_nor_4
reduce_nor_3
reduce_nor_2
reduce_nor_1
reduce_nor_0
[69..62]
[77..70]
[85..78]
[93..86]
[9..0]
[25..18]
[17..10]
add_0
pcadd1
(adder)
pcreg
(flcopr)
q[0]~reg[7..0]
configure
(configure)
r_add[0]~reg[7..0]

Component 2: Software Companion

i. Provide an introduction to your software companion's main features.

- The software companion is written in Java. In software we have taken an input (machine codes) in a text file. The machine codes are in 32-bit binary encoding.
- We are basically stored instruction opcodes in different fields that are the instruction fields themselves. As and when the machine code is read from the file, it reads the bits and detects which instruction is to be run based on the opcode bits and its position in the code.
- We are using if-else functions to check through which instruction is to be run.
- In each of the if-else function, functionality of the instruction is defined and based on the opcode it detects from the machine code, functions will be accordingly called to get the desired outputs.

ii. Use your software companion's results to fill in the table above, and one more for your lfsr_t2 testing program.

lfsr_t1 assembly code	lfsr_t1 machine code	LFSR content	r_addr content	Other info
config_L 0100101;	04000025	00000000		
init_L 11111111;	0803FC00	11111111		
run_L;	0C000000	11011010		
run_L;	0C000000	01101101		
init_addr 00001001;	18240000	01101101	Address 9=00001001	M[9]
run_L;	0C000000	10010011	00001001	
st_M_L;	10000000	10010011	00001001	M[9]=10010011
run_L;	0C000000	11101100	00001001	
add_addr 11111110	1C03F800	11101100	9-2=7(00000111)	M[7]
st_M_L;	10000000	11101100	00000111	M[7]=11101100
add_addr 00000010;	1C000800	11101100	7+2=9	M[9]
ld_M_L;	14000000	10010011	00001001	LFSR=M[9]
run_L;	0C000000	11101100	00001001	
halt;	00000000	11101100		

lfsr_t2 assembly code	lfsr_t2 machine code	LFSR content	r_addr content	Other info
config_L 1001010;	0400004A	00000000		
init_L 11111111;	0803FC00	11111111		
run_L;	0C000000	10110101		
init_addr 00000100;	18100000	10110101	Address 4=00000100	M[4]
run_L;	0C000000	10010000	00000100	
run_L;	0C000000	01001000	00000100	
st_M_L;	10000000	01001000	00000100	M[4]=01001000
run_L;	0C000000	00100100	00000100	
add_addr 00001000;	1C002000	00100100	4+8=12(00001100)	M[12]
st_M_L;	10000000	00100100	00001100	M[12]=00100100
add_addr 11111000;	1C03E000	00100100	12-8 = 4 (00000100)	M[4]
ld_M_L;	14000000	01001000	00000100	LFSR=M[4]
run_L;	0C000000	00100100	00000100	
halt;	00000000	00100100		

lfsr_t1

```
LFSR  - [0, 0, 0, 0, 0, 0, 0, 0]
LFSR  - [1, 1, 1, 1, 1, 1, 1, 1]
LFSR  - [1, 0, 1, 1, 0, 1, 0, 1]
LFSR  - [1, 0, 1, 1, 0, 1, 0, 1]
LFSR  - [1, 0, 0, 1, 0, 0, 0, 0]
LFSR  - [0, 1, 0, 0, 1, 0, 0, 0]
Memory Content:[0, 1, 0, 0, 1, 0, 0, 0]LFSR  - [0, 1, 0, 0, 1, 0, 0, 0]
LFSR  - [0, 0, 1, 0, 0, 1, 0, 0]
Memory - [0, 0, 1, 0, 0, 1, 0, 0]LFSR  - [0, 0, 1, 0, 0, 1, 0, 0]
Memory Content:[0, 0, 1, 0, 0, 1, 0, 0]LFSR  - [0, 0, 1, 0, 0, 1, 0, 0]
Memory - [0, 0, 1, 0, 0, 1, 0, 0]LFSR  - [0, 0, 1, 0, 0, 1, 0, 0]
LFSR  - [0, 0, 1, 0, 0, 1, 0, 0]
LFSR  - [0, 0, 0, 1, 0, 0, 1, 0]
LFSR  - [0, 0, 0, 1, 0, 0, 1, 0]
```

lfsr_t2

```
LFSR  - [0, 0, 0, 0, 0, 0, 0, 0]
LFSR  - [1, 1, 1, 1, 1, 1, 1, 1]
LFSR  - [1, 1, 0, 1, 1, 0, 1, 0]
LFSR  - [0, 1, 1, 0, 1, 1, 0, 1]
LFSR  - [0, 1, 1, 0, 1, 1, 0, 1]
LFSR  - [1, 0, 0, 1, 0, 0, 1, 1]
Memory Content:[1, 0, 0, 1, 0, 0, 1, 1]LFSR  - [1, 0, 0, 1, 0, 0, 1, 1]
LFSR  - [1, 1, 1, 0, 1, 1, 0, 0]
Memory - [1, 1, 1, 0, 1, 1, 0, 0]LFSR  - [1, 1, 1, 0, 1, 1, 0, 0]
Memory Content:[1, 1, 1, 0, 1, 1, 0, 0]LFSR  - [1, 1, 1, 0, 1, 1, 0, 0]
Memory - [1, 1, 1, 0, 1, 1, 0, 0]LFSR  - [1, 1, 1, 0, 1, 1, 0, 0]
LFSR  - [1, 1, 1, 0, 1, 1, 0, 0]
LFSR  - [0, 1, 1, 1, 0, 1, 1, 0]
LFSR  - [0, 1, 1, 1, 0, 1, 1, 0]
```

iii. Provide all the code of your software companion in the appendix part. (Code added in APPENDIX 2)

Component 3: HD Count

Instruction	Functionality	Machine code Encoding	Opcode
st_M_HD	For the current pattern P in LFSR, compute the Hamming Distance between P and P_next, and store the HD count into M[r_addr]	[opcode 31:26] [25:18] [17:10] [9:7] [6:0]	001000

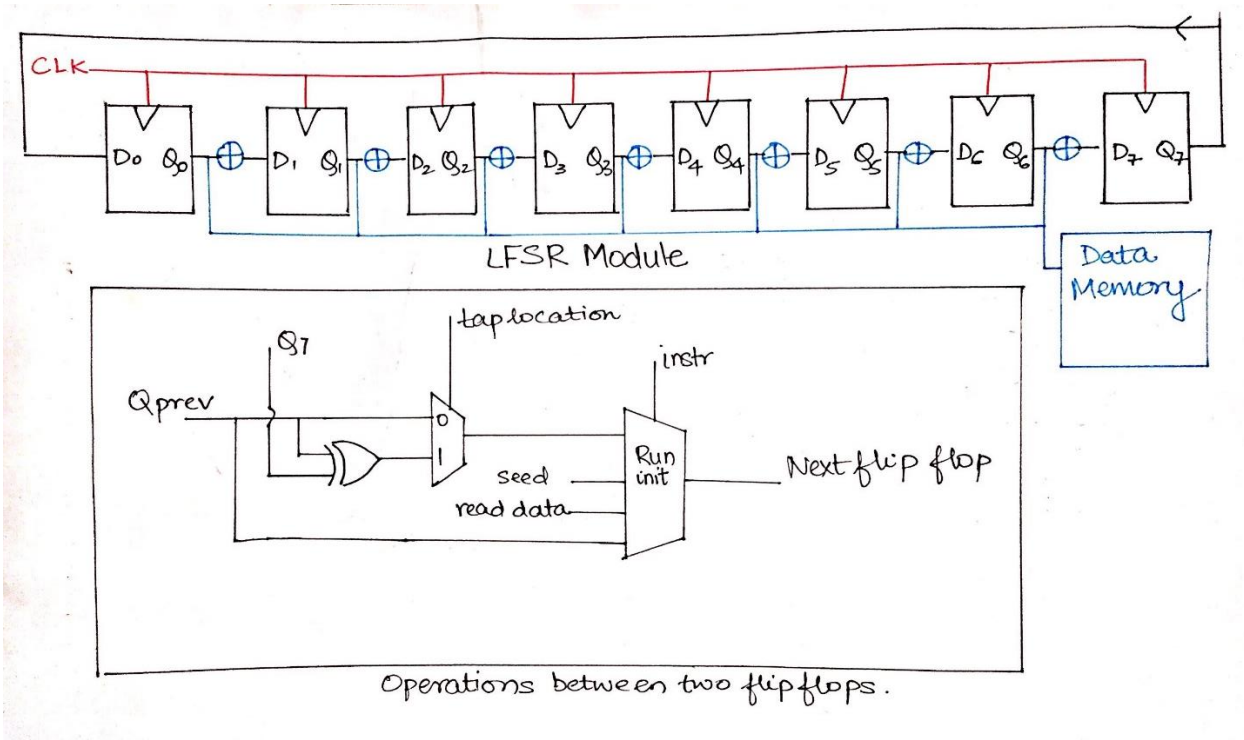
i. Provide the machine code and use your software companion to fill in the table for the following test program lfsr_t3a:

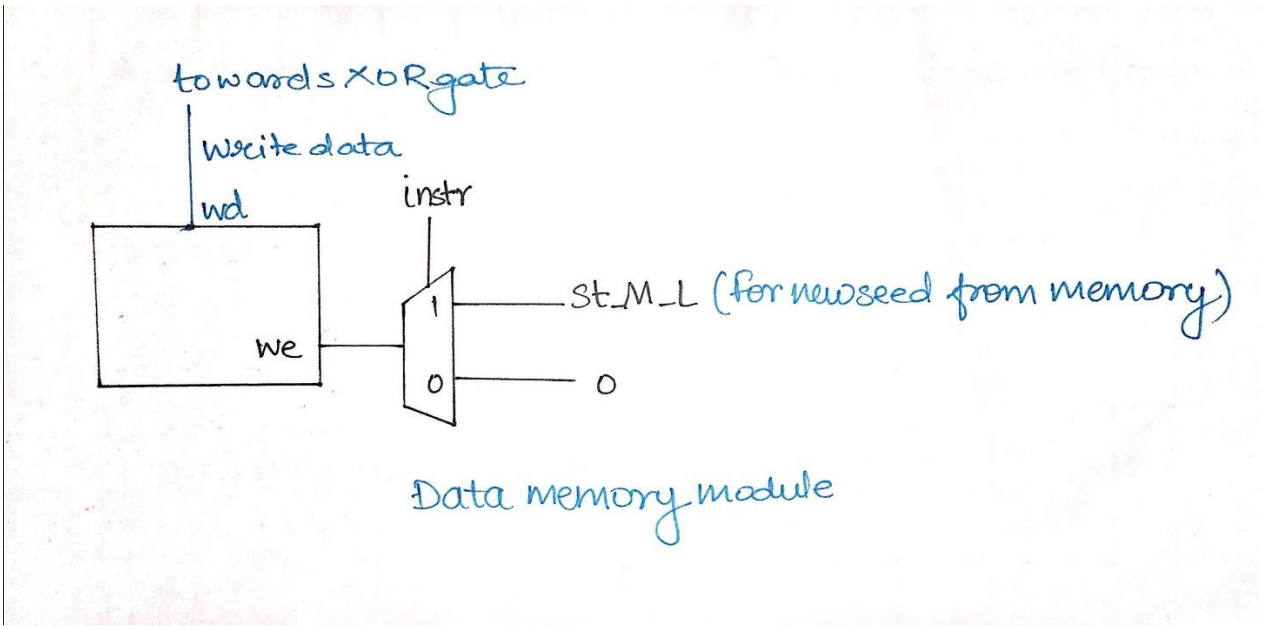
lfsr_t3a assembly code	lfsr_t3a machine code	LFSR content P	P_next	HD count	Other info
config_L 0100101;	04000025				
init_L 11111111;	0803FC00	11111111	11011010	3	
run_L;	0C000000	11011010	01101101	6	
run_L;	0C000000	01101101	10010011	7	
init_addr 00001001;	18240000	01101101	10010011	7	r_addr=9
run_L;	0C000000	10010011	11101100	7	
st_M_L;	10000000	10010011	11101100	7	M [9] =10010011
run_L;	0C000000	11101100	01110110	4	
add_addr 00000011;	1C000C00	11101100	01110110	4	r_addr=12
st_M_HD;	20000000	11101100	01110110	4	M [12] =00000100
run_L;	0C000000	01110110	00111011	5	
ld_M_L;	14000000	00000100	00000010	2	
run_L;	0C000000	00000010	00000001	2	
run_L;	0C000000	00000001	10100101	3	
st_M_HD;	20000000	00000001	10100101	3	M [12] =00000011
run_L;	0C000000	10100101	11110111	3	
st_M_L;	10000000	10100101	11110111	3	M [12] =10100101
run_L;	0C000000	11110111	11011110	3	
halt;	00000000	11110111	11110111	0	

ii. Provide another testing program lfsr_t3b of your own design to feature the new instruction of st_M_HD and use your software companion to fill in the table:

lfsr_t3b assembly code	lfsr_t3b machine code	LFSR content P	P_next	HD count	Other info
config_L 1001010;	0400004A				
init_L 11111111;	0803FC00	11111111	10110101	3	
run_L;	0C000000	10110101	10010000	3	
init_addr 0001010;	18280000	10110101	10010000	3	r_addr=10
run_L;	0C000000	10010000	01001000	4	
run_L;	0C000000	01001000	00100100	4	
st_M_L;	10000000	01001000	00100100	4	M [10] =01001000
run_L;	0C000000	00100100	00010010	4	
add_addr 0000010;	1C000800	00100100	00010010	4	r_addr=10+2=12
st_M_L;	10000000	00100100	00010010	4	M [12] =00100100
run_L;	0C000000	00010010	00001001	4	
run_L;	0C000000	00001001	11001110	5	
ld_M_L;	14000000	00100100	00010010	4	
st_M_HD;	20000000	00100100	00010010	4	M [12] =00000100
run_L;	0C000000	00010010	00001001	4	
st_M_L;	10000000	00010010	00001001	4	M [12] =00010010
run_L;	0C000000	00001001	11001110	5	
halt;	00000000	11001110	11001110	0	

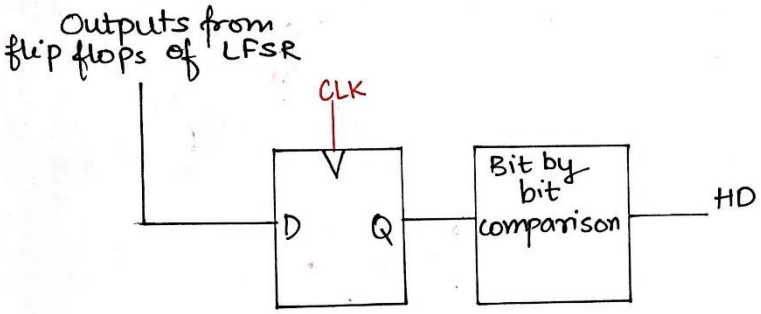
iii. Display your main modification to support this instruction upgrade in a sketch



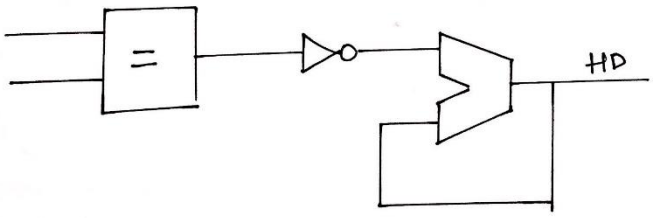


Data memory module

Hamming distance calculation



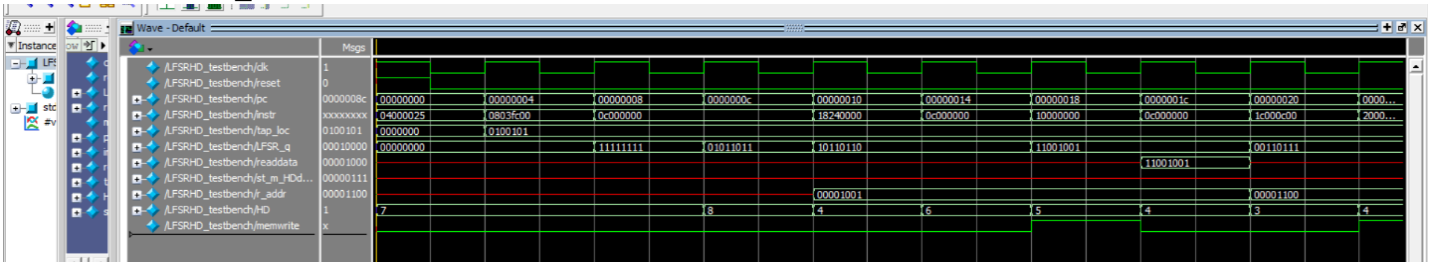
Comparison module

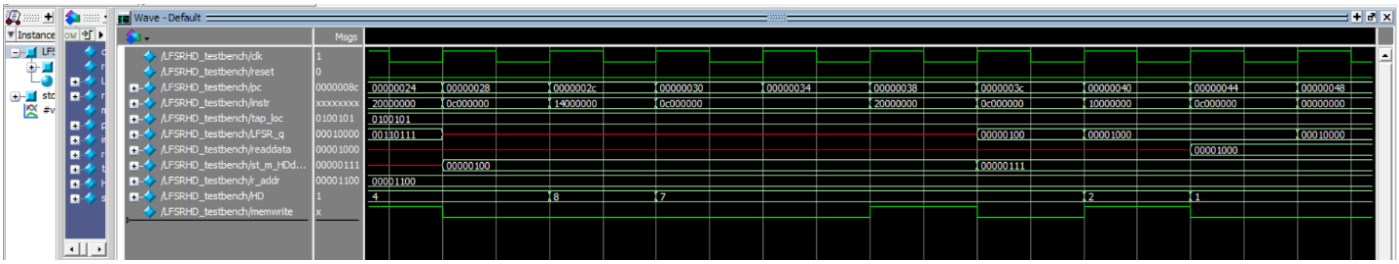


Along with the LFSR processor module above, we are adding a hamming distance calculation method where the previous LFSR content and current LFSR content is compared bit by bit, and any change in the buts is incremented in the counter and the distance is calculated.

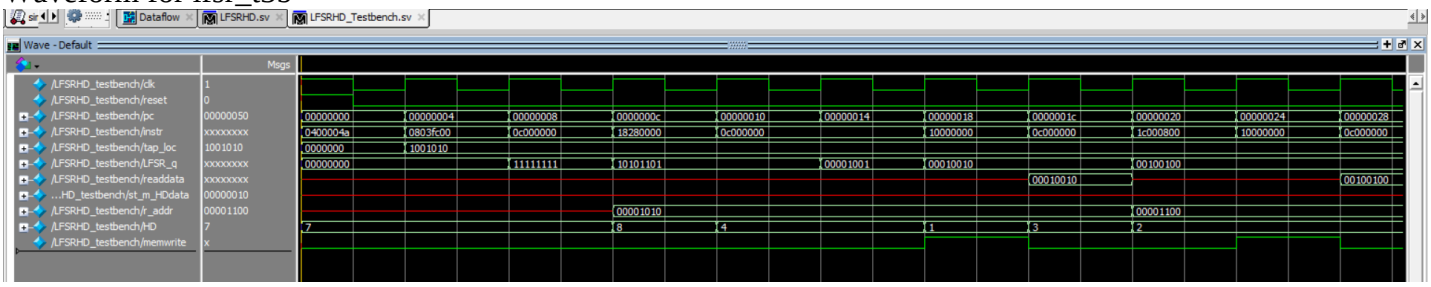
iv. Run both testing programs lfsr_t3a and lfsr_t3b with ModelSim to verify that your design works correctly. Show the waveform results.

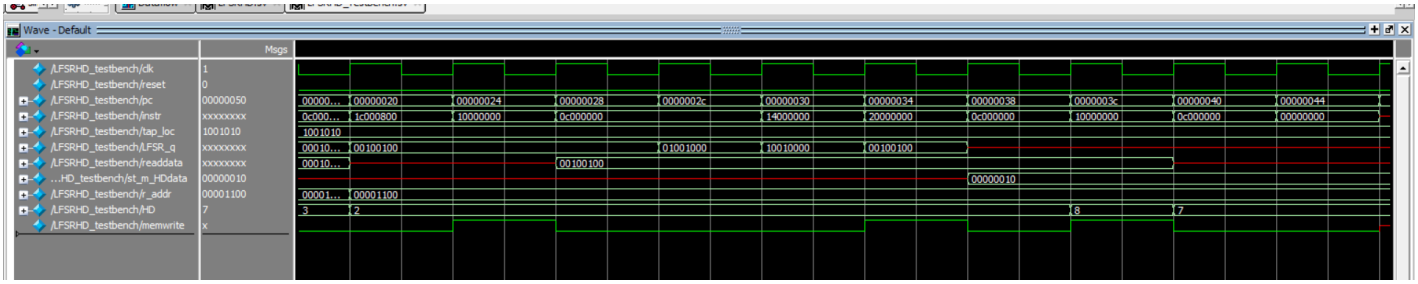
Waveform for lfsr_t3a



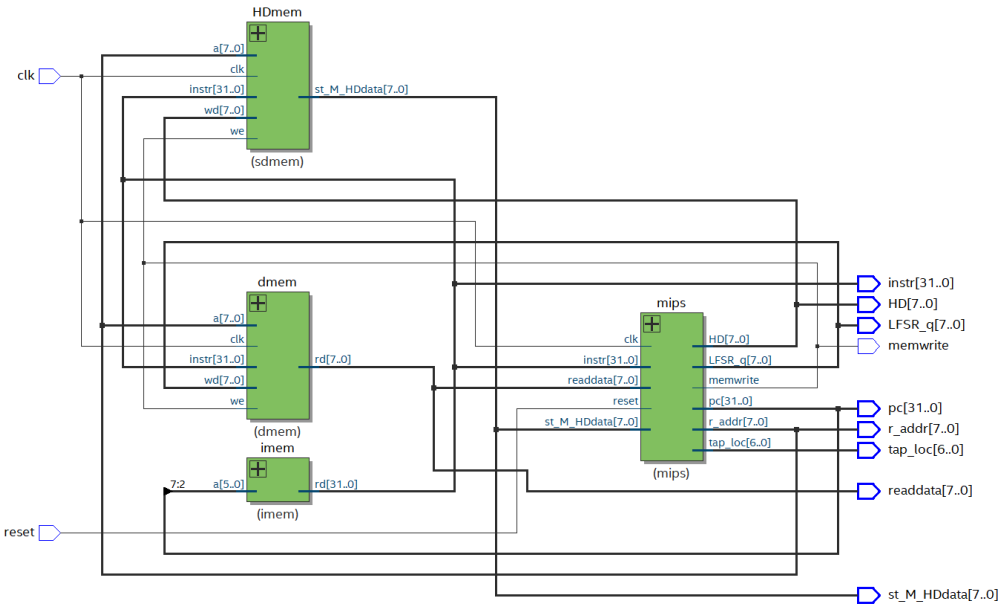


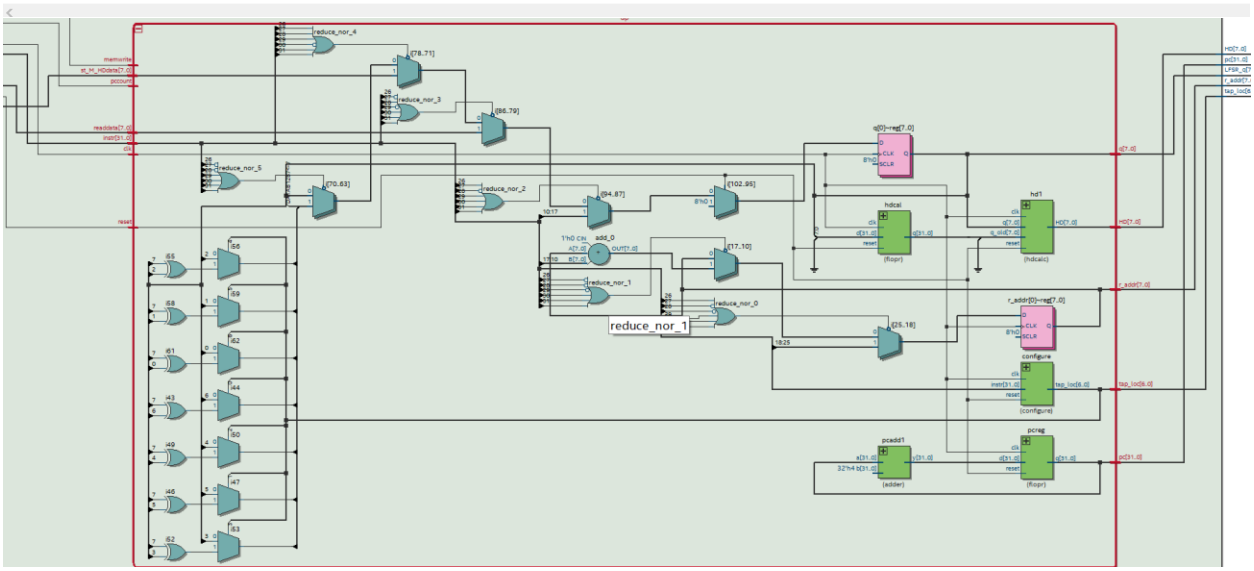
Waveform for lfsr_t3b

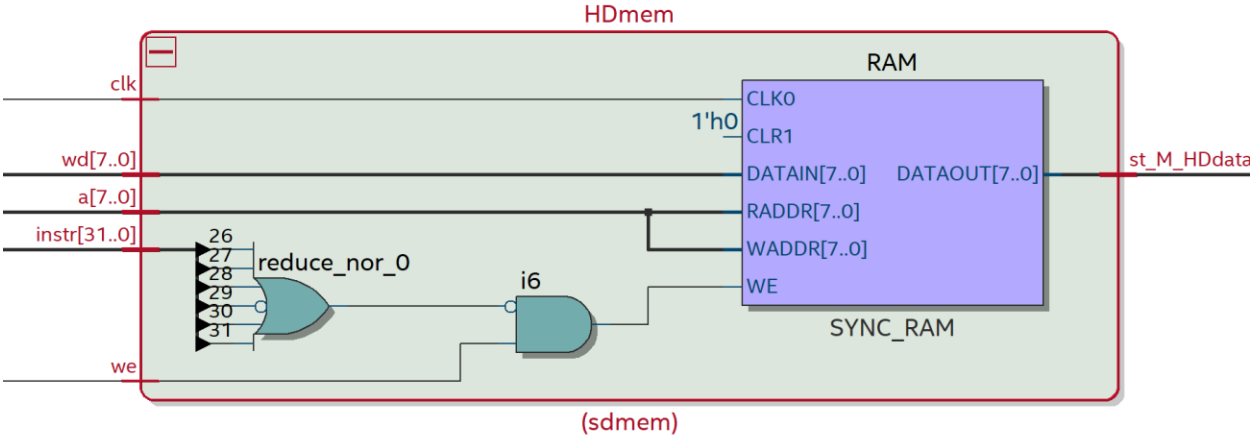


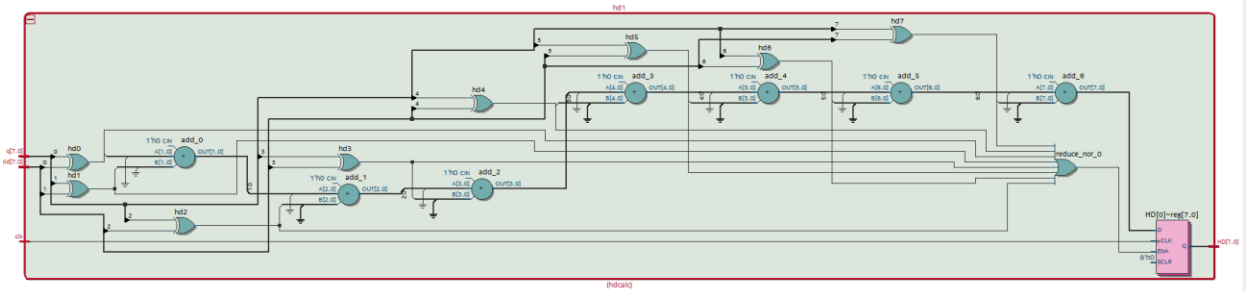


RTL View


HDmem
a[7..0]
clk
instr[31..0]
wd[7..0]
we
st_M_HDdata[7..0]
(sdmem)
clk
dmem
a[7..0]
clk
instr[31..0]
wd[7..0]
we
rd[7..0]
(dmem)
imem
7:2
a[5..0]
rd[31..0]
(imem)
reset
mips
clk
instr[31..0]
readdata[7..0]
reset
st_M_HDdata[7..0]
HD[7..0]
LFSR_q[7..0]
memwrite
pc[31..0]
r_addr[7..0]
tap_loc[6..0]
(mips)
instr[31..0]
HD[7..0]
LFSR_q[7..0]
memwrite
pc[31..0]
r_addr[7..0]
tap_loc[6..0]
readdata[7..0]
st_M_HDdata[7..0]


reduce_nor_1


HDmem
clk
wd[7..0]
a[7..0]
instr[31..0]
we
RAM
CLK0
1'h0
CLR1
DATAIN[7..0]
DATAOUT[7..0]
RADDR[7..0]
WADDR[7..0]
WE
SYNC_RAM
st_M_HDdata
26
27
28
29
30
31
reduce_nor_0
i6
(sdmem)

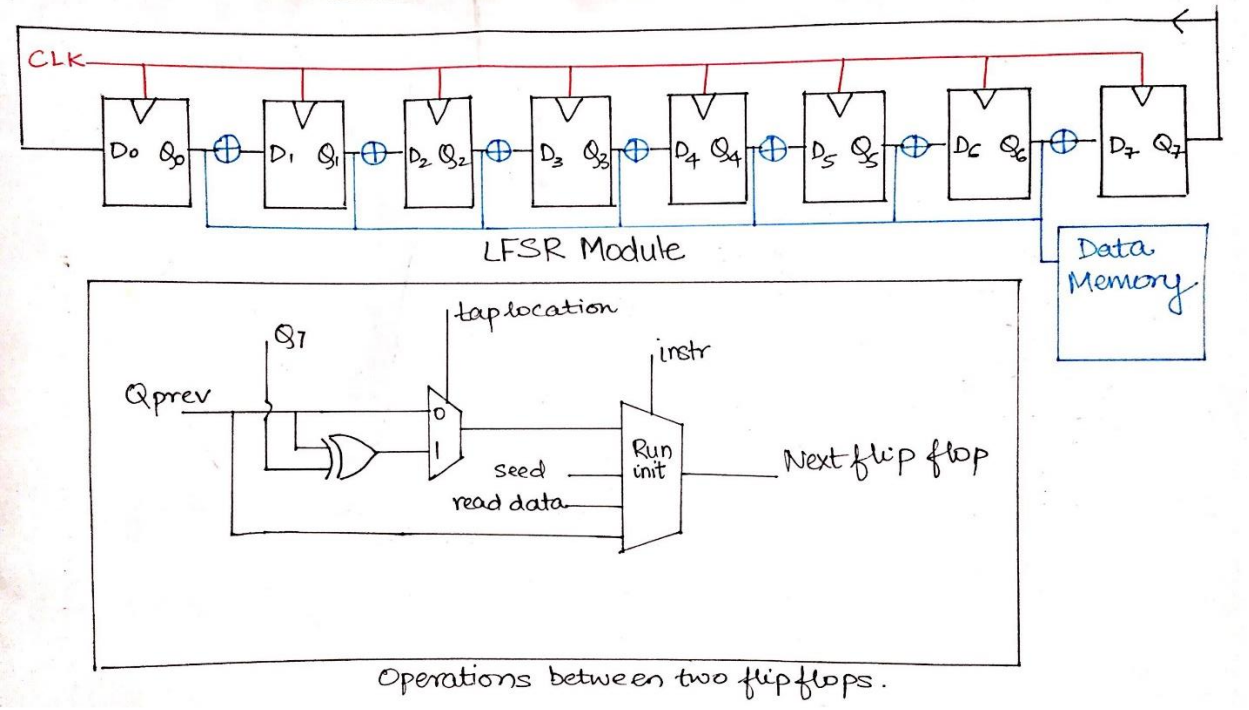

hd0
hd1
hd2
hd3
hd4
hd5
hd6
hd7
add_0
add_1
add_2
add_3
add_4
add_5
add_6
reduce_nor_0
HD[0]~reg[7..0]

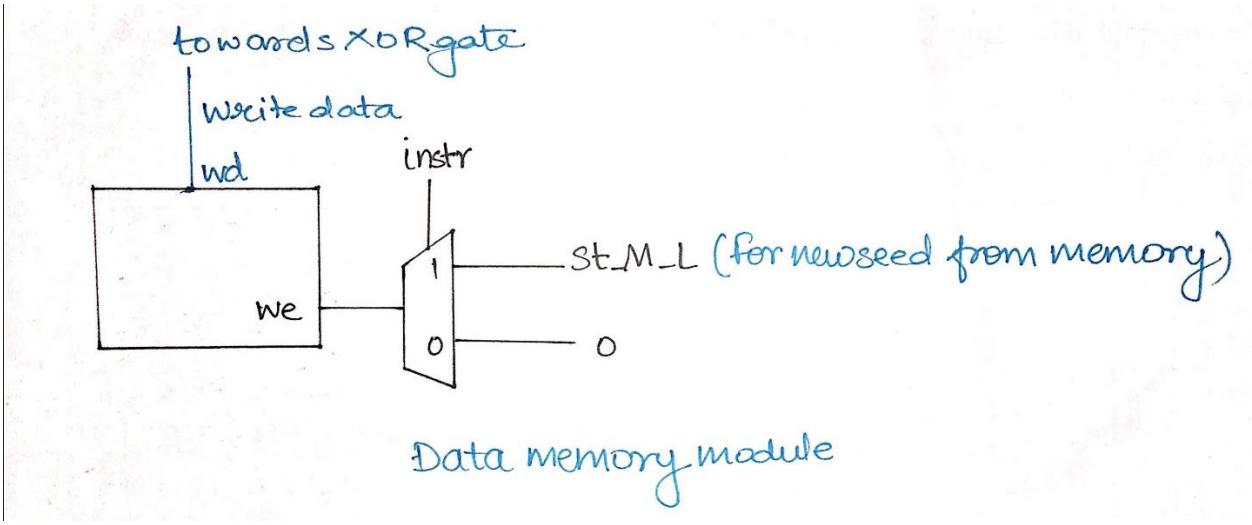
Component 4: Avg HD

i. Which new instruction(s) did you add to support such a feature? List out their assembly and machine code.

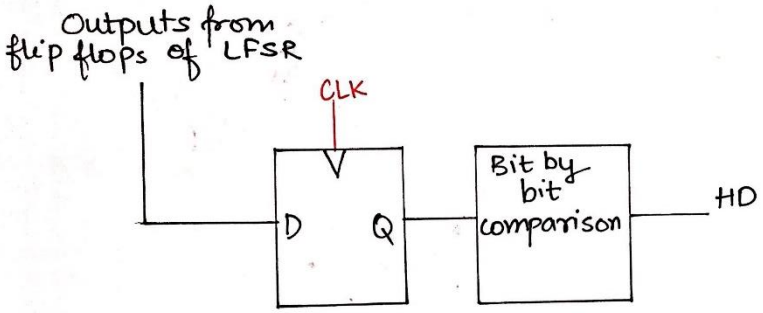
Instruction	Functionality	Machine code Encoding	Opcode
avg_M_HD	For the current pattern P in LFSR, compute the Hamming Distance between P and P_next, then find the average of the Hamming distance and store it in M [0]	[opcode 31:26] [25:18] [17:10] [9:7] [6:0]	001001

ii. Show the main modification to your processor in a sketch.

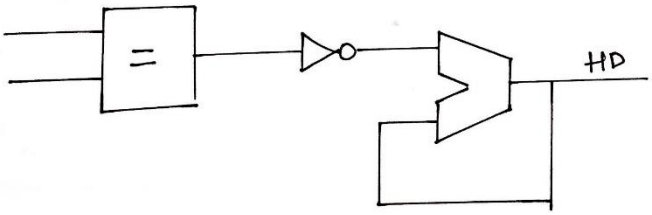




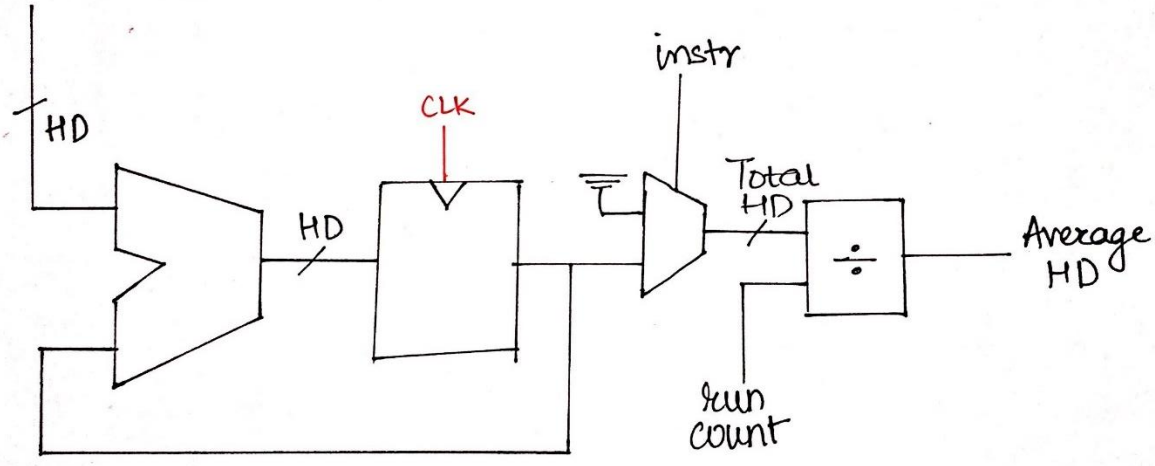
Hamming distance calculation



Comparison module



Hamming distance Average calculation



For this module, all the hamming distances are calculated, along with which, a run counter counts the number of times the LFSR has been run and has generated a value. This run count value is used for the average HD calculation.

iii. Provide a test program lfsr_t4a that achieves the following, and use your software companion to derive the result:

- Configure the LFSR to be 0100101
- Seed the LFSR with 11111111
- Let it run for 16 cycles
- Write the avg HD (of run of 16 cycles) in M [0]

Line #	Assembly Language	Machine Code	LFSR Content	LFSR next	HD
1	config_L 0100101;	04000025			
2	init_L 11111111;	0803FC00	11111111	11011010	3
3	run_L;	0C000000	11011010	01101101	6
4	run_L;	0C000000	01101101	10010011	7
5	run_L;	0C000000	10010011	11101100	7
6	run_L;	0C000000	11101100	01110110	4
7	run_L;	0C000000	01110110	00111011	3
8	run_L;	0C000000	00111011	10111000	4
9	run_L;	0C000000	10111000	01011100	4
10	run_L;	0C000000	01011100	00101110	4
11	run_L;	0C000000	00101110	00010111	4
12	run_L;	0C000000	00010111	10101110	5
13	run_L;	0C000000	10101110	01010111	6
14	run_L;	0C000000	01010111	10001110	5
15	run_L;	0C000000	10001110	01000111	4
16	run_L;	0C000000	01000111	10000110	3
17	run_L;	0C000000	10000110	01000011	4
18	run_L;	0C000000	01000011	10000100	5
19	avg_M_HD;	24000000	00000100		
20	halt;	00000000	01000011	01000011	0

Hamming Distance total = 78. Average = 78/16 = 4.875

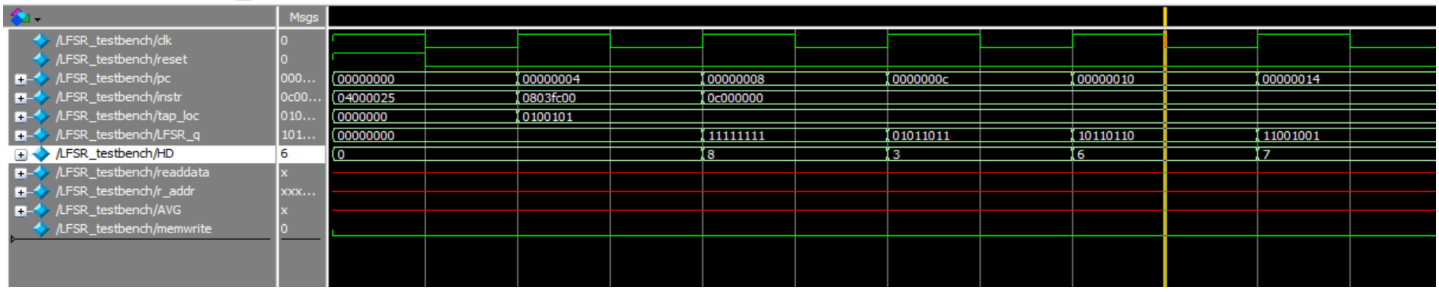
iv. Provide another testing program lfsr_t4b of your own design to feature the new instruction.

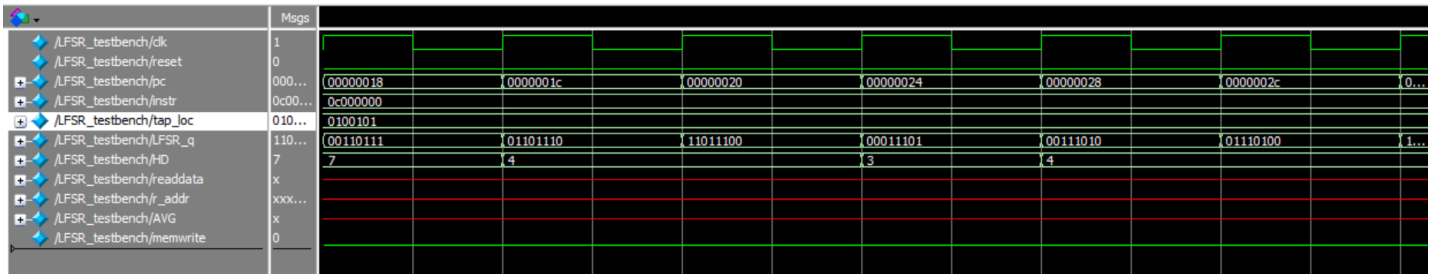
Line #	Assembly Language	Machine Code	LFSR Content	LFSR next	HD
1	config_L 1001010;	0400004A			
2	init_L 11111111;	0803FC00	11111111	10110101	3
3	run_L;	0C000000	10110101	10010000	3
4	run_L;	0C000000	10010000	01001000	4
5	run_L;	0C000000	01001000	00100100	4
6	run_L;	0C000000	00100100	00010010	4
7	run_L;	0C000000	00010010	00001001	4
8	run_L;	0C000000	00001001	11001110	5
9	run_L;	0C000000	11001110	01100111	4
10	run_L;	0C000000	01100111	11111001	5
11	run_L;	0C000000	11111001	10110110	5
12	avg_M_HD;	24000000	00000100		
13	halt;	00000000	01000011	01000011	0

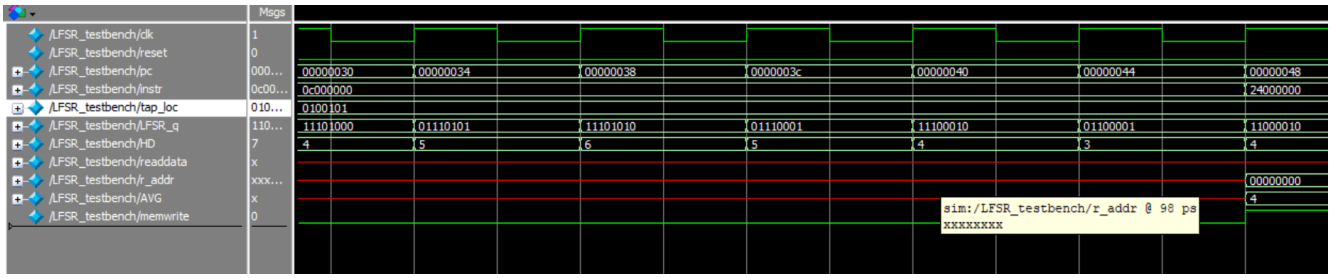
Hamming distance total = 36. Average = 36/9 =4

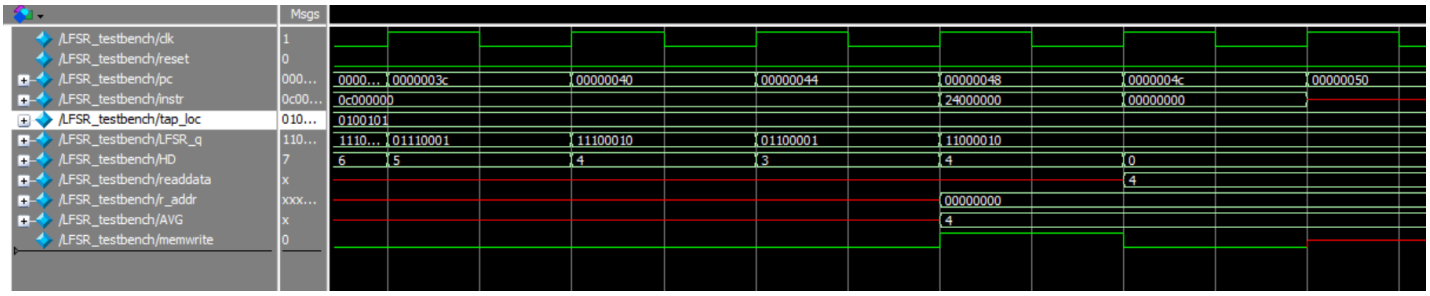
v. Run both testing programs lfsr_t4a and lfsr_t4b with ModelSim to verify that your design works correctly. Show the waveform results.

Waveform for lfsr_t4a

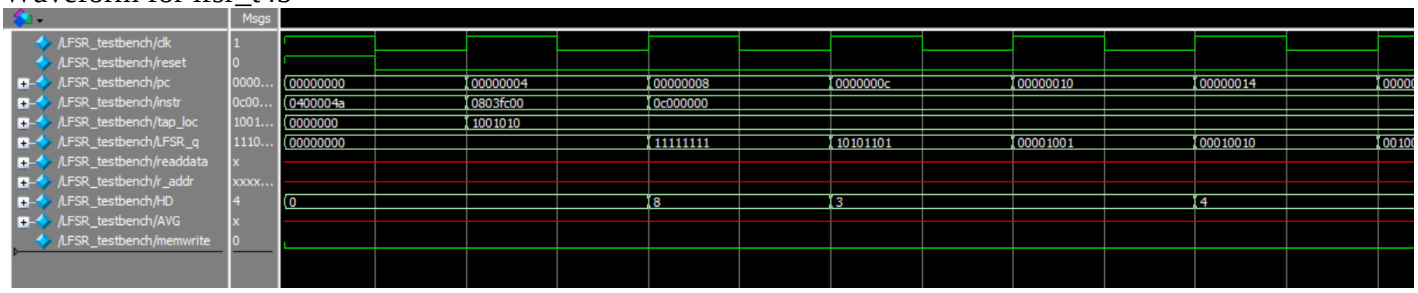


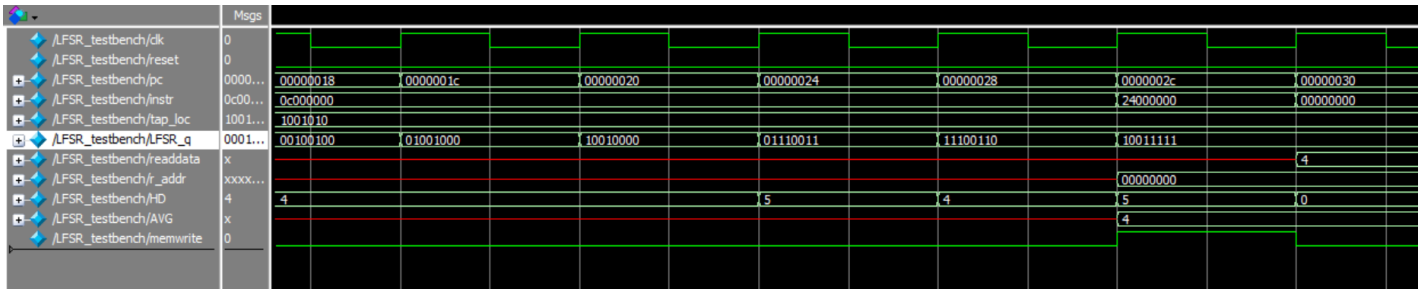




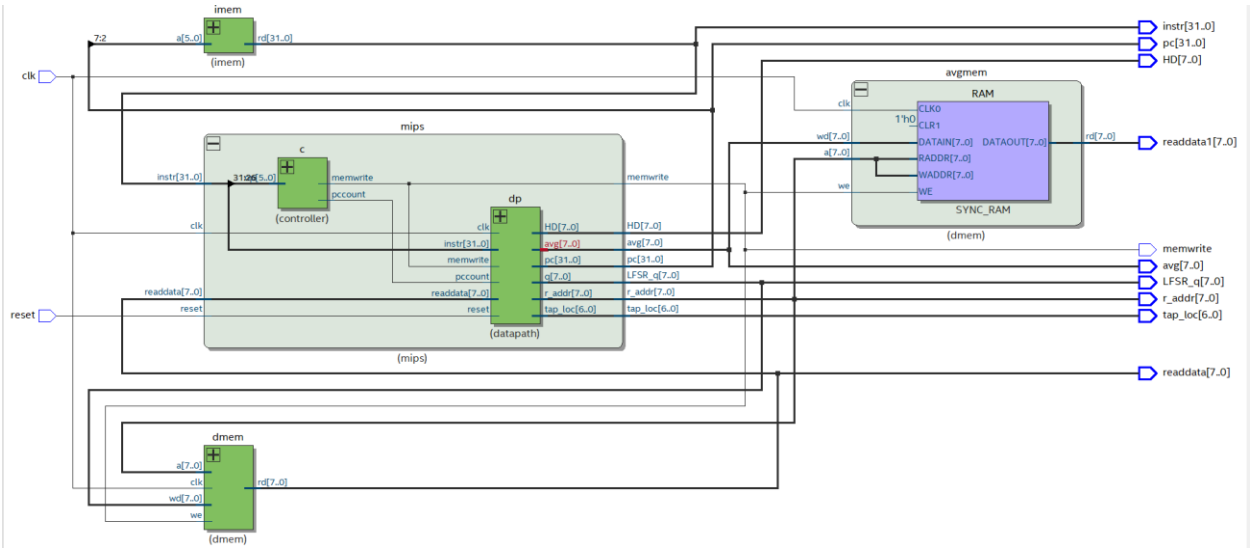


Waveform for lfsr_t4b





RTL View


imem
7:2
a[5..0]
rd[31..0]
(imem)
clk
mips
c
instr[31..0]
31:26
5..0]
memwrite
pccount
(controller)
dp
clk
instr[31..0]
memwrite
pccount
readdata[7..0]
reset
HD[7..0]
avg[7..0]
pc[31..0]
q[7..0]
r_addr[7..0]
tap_loc[6..0]
(datapath)
(mips)
reset
avgmem
RAM
clk
1'h0
wd[7..0]
a[7..0]
we
CLK0
CLR1
DATAIN[7..0]
DATAOUT[7..0]
RADDR[7..0]
WADDR[7..0]
WE
SYNC_RAM
(dmem)
rd[7..0]
dmem
a[7..0]
clk
wd[7..0]
we
rd[7..0]
(dmem)
instr[31..0]
pc[31..0]
HD[7..0]
readdata1[7..0]
memwrite
avg[7..0]
LFSR_q[7..0]
r_addr[7..0]
tap_loc[6..0]
readdata[7..0]

Component 5: Multi-Cycle Run

Instruction	Functionality	Machine code Encoding	Opcode
run_L cyl	Let the LFSR run for cyl number of cycles, where cyl is an 8-bit positive number	[opcode 31:26] [25:18] [17:10 cyl] [9:7] [6:0]	000011

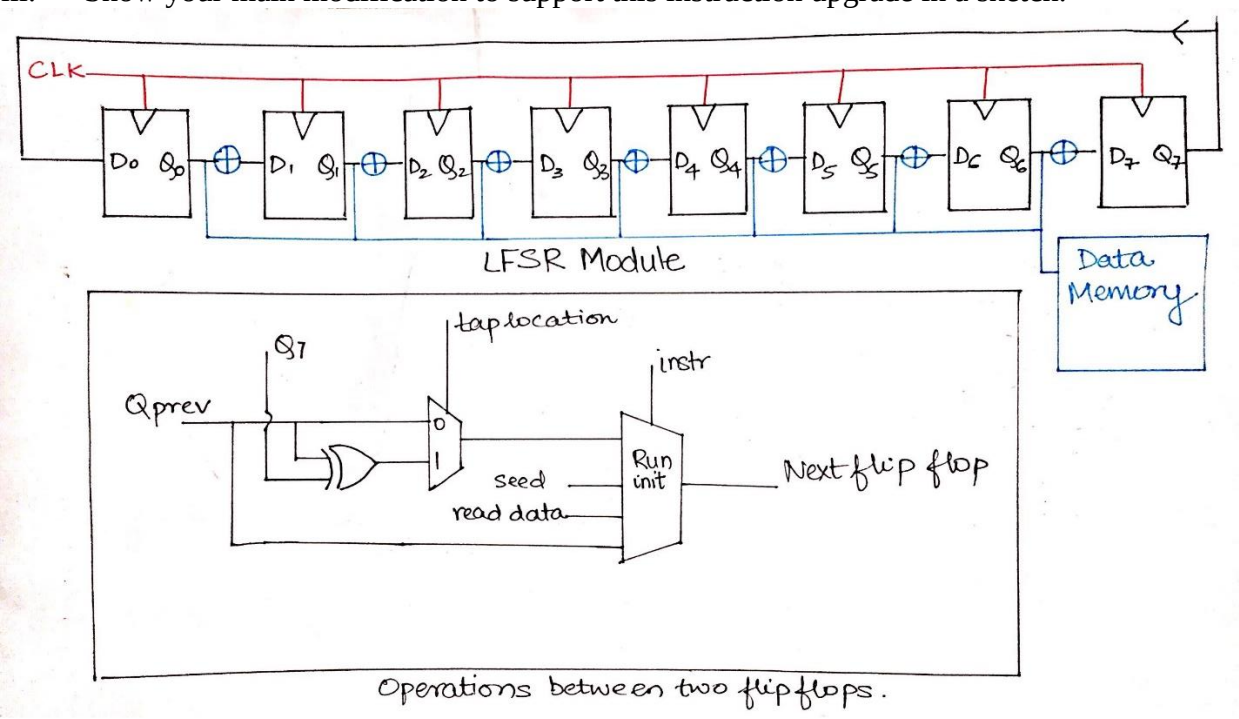
i. Provide the machine code and use your software companion to fill in the table for the following test program lfsr_t5a:

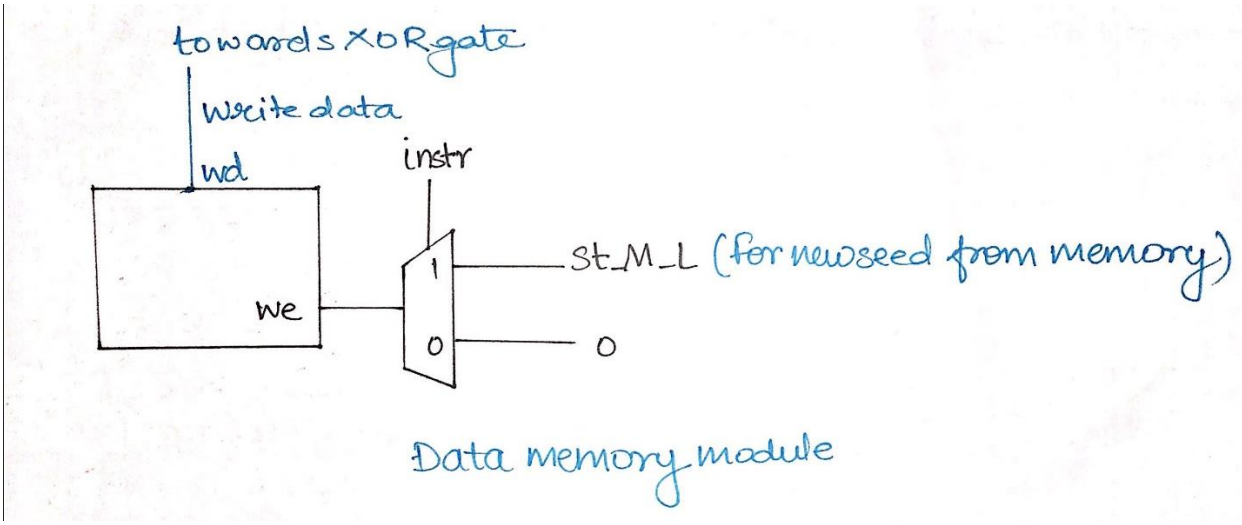
Assembly Language	Machine Code	LFSR Content	Other info
config_L 0100101;	04000025		
init_L 11111111;	0803FC00	11111111	
run_L 00000001;	0C000400	11011010	
run_L 00000000;	0C000000	11011010	
init_addr 00001001;	18240000	11011010	M[9]
run_L 00000101;	0C001400	00111011	
st_M_L;	10000000	00111011	M[9]=00111011
run_L 00001011;	0C002C00	10000100	
add_addr 11111110	1C03F800	10000100	9-2 = 7
st_M_L;	10000000	10000100	M[7]=10000100
add_addr 00000010;	1C000800	10000100	7+2=9
ld_M_L;	14000000	00111011	LFSR=M[9]
run_L 00011001;	0C006400	00010111	
add_addr 00000110;	1C001800	00010111	9+6=15
st_M_L;	10000000	00010111	M[15]=00010111
halt;	00000000		

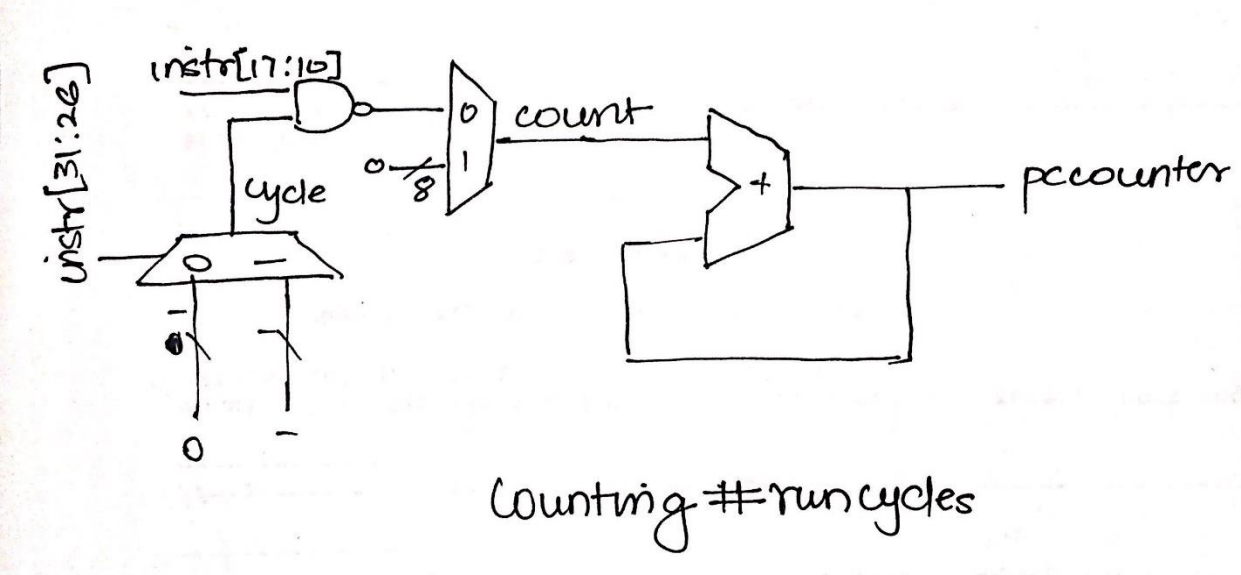
ii. Provide another testing program lfsr_t5b of your own design to feature the upgraded instruction of run_L cyl and use your software companion to fill in the table:

Assembly Language	Machine Code	LFSR Content	Other info
config_L 1001010;	0400004A		
init_L 11111111;	0803FC00	11111111	
run_L 00000010;	0C000800	10010000	
init_addr 00000100;	18100000	10010000	M[4]
run_L 00000111;	0C001C00	11111001	
st_M_L;	10000000	11111001	M[4]=11111001
add_addr 00001000;	1C002000	11111001	4+8 = 12
st_M_L;	10000000	11111001	M[12]=11111001
add_addr 11111000;	1C03E000	11111001	12 - 8 =4
ld_M_L;	14000000	11111001	LFSR=M[4]
run_L 0001001;	0C002400	00101001	
halt;	00000000		

iii. Show your main modification to support this instruction upgrade in a sketch.



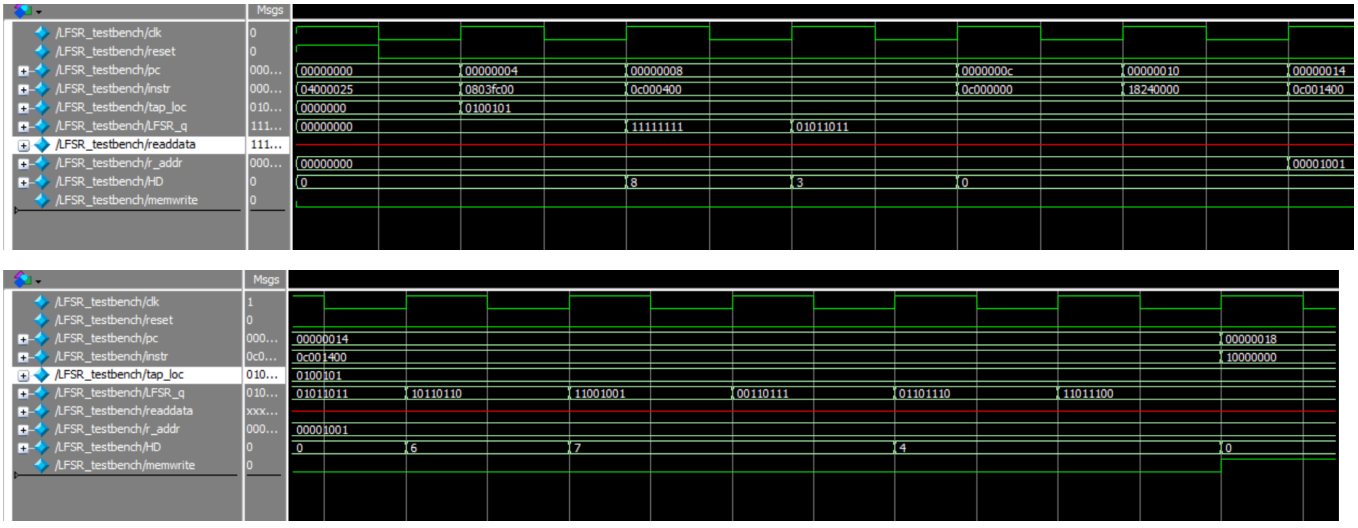


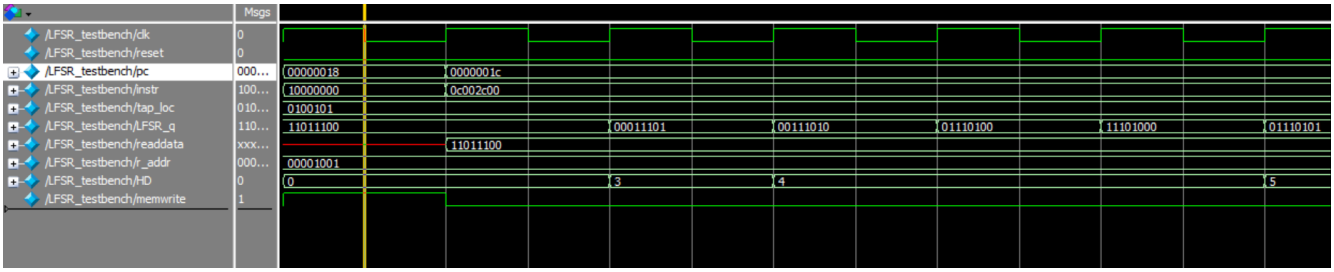


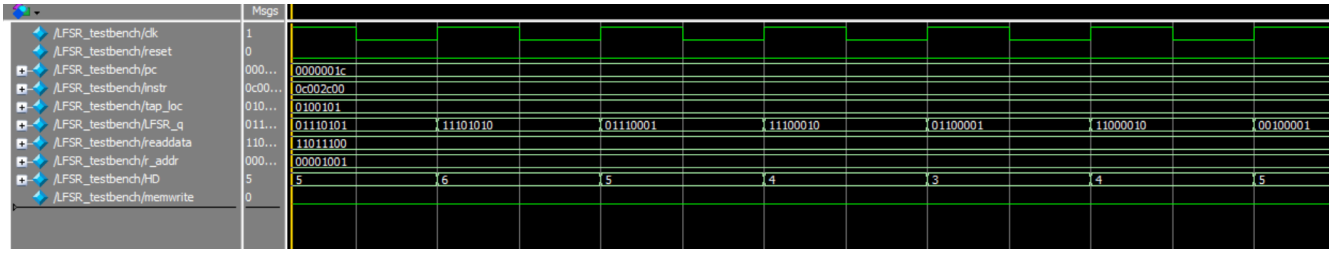
For this module, there is no new instruction, on the cycle bits are checked in the original run instruction, if there is a cycle bit instruction present, the count is set to 1 and for every time the run has to be performed, a pc counter keeps track of the count and the run is performed that many number of times.

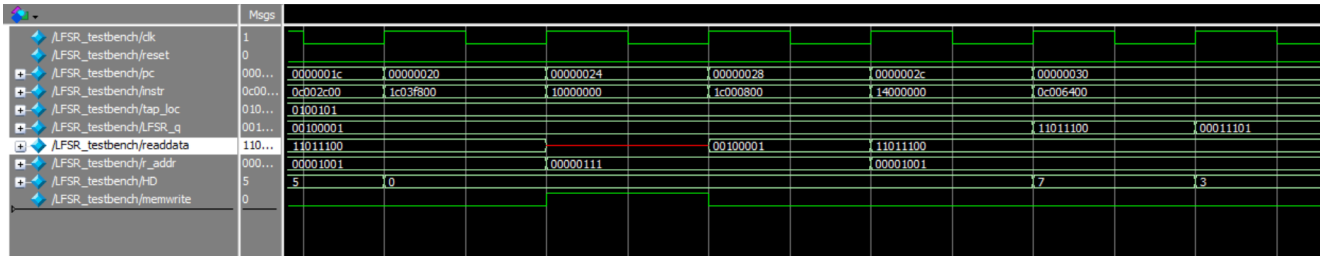
iv. Run both testing programs lfsr_t5a and lfsr_t5b with ModelSim to verify that your design works correctly. Show the waveform results.

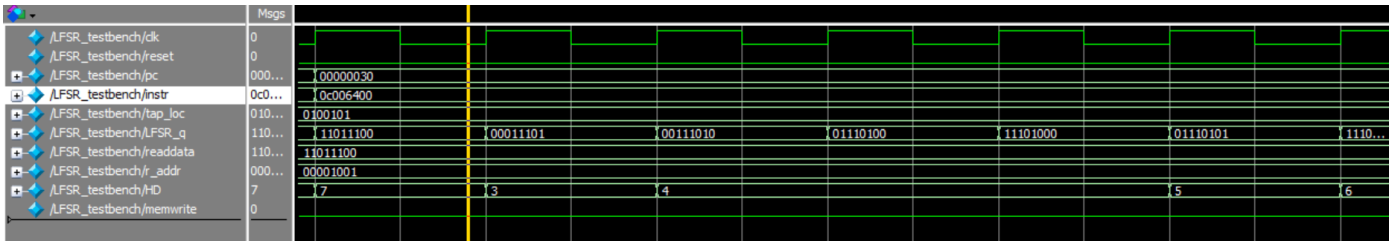
Waveform for lfsr_t5a



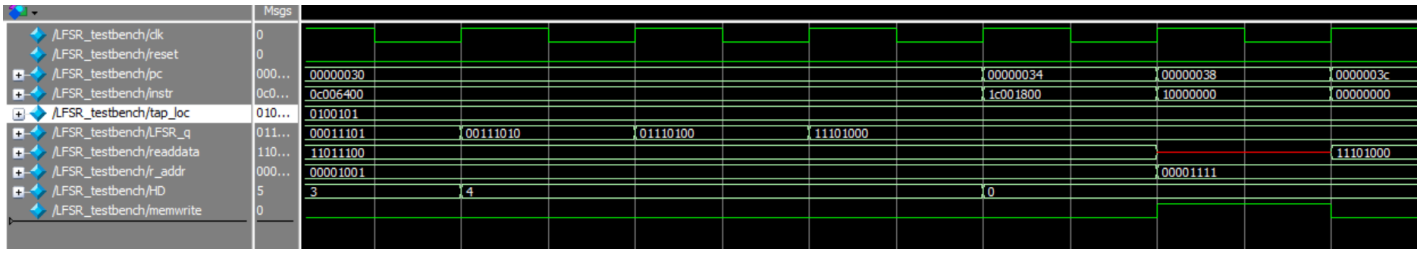




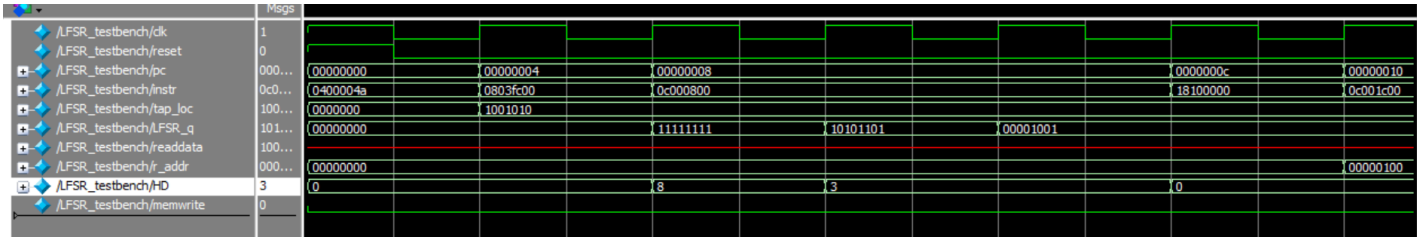


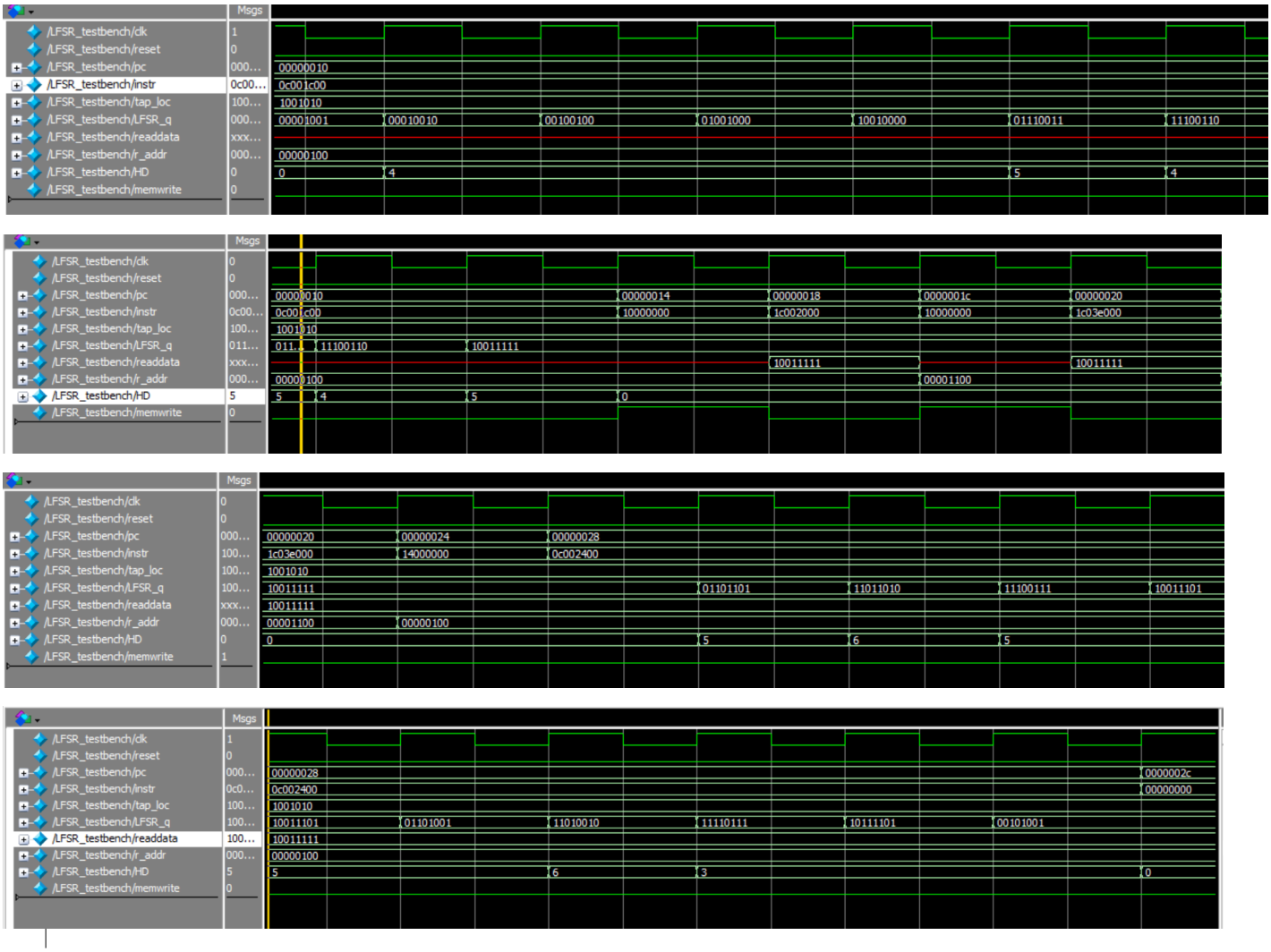


After 25 runs



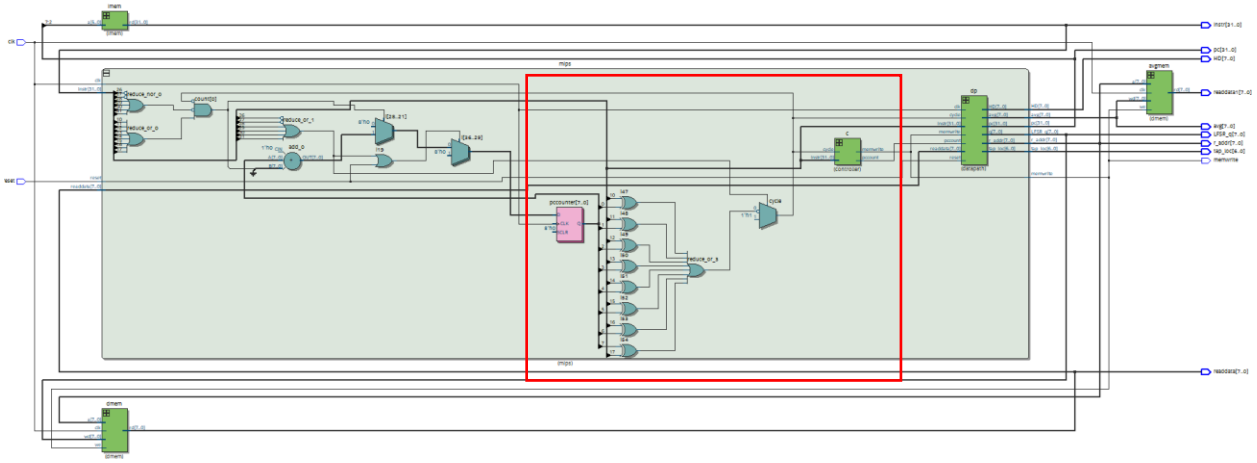
Waveform for lfsr_t5b





RTL View

Counter to halt program counter to run



Component 6: Batch Mem Storage

Instruction	Functionality	Machine code Encoding	Opcode
batch_run_st_ M_L cyl	store the next cyl number of LFSR patterns into a sequence of memory locations: M[r_addr], M[r_addr+1], … M[r_addr + cyl]. cyl is an 8-bit positive number	[opcode 31:26] [25:18] [17:10 cyl] [9:7] [6:0]	001010

i. Provide the machine code and use your software companion to fill in the table for the following test program lfsr_t6a:

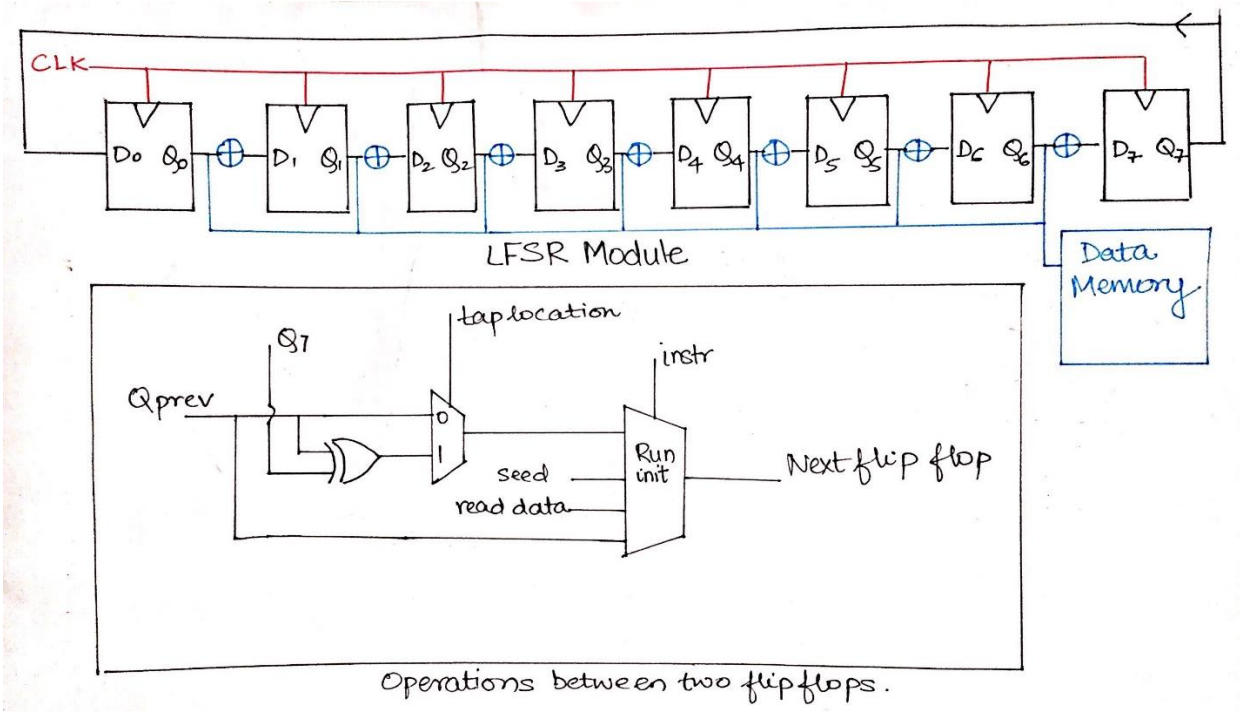
Assembly Language	Machine Code	LFSR Content or r_addr	Other info
config_L 0100101;	04000025		
init_L 11111111;	0803FC00	11111111	
init_addr 00001001;	18240000	r_addr = 9	M[9]
batch_run_st_M_L 00000110;	28001800	00111011	Continuously run LFSR for 6 cycles and write the patterns into M[9], M[10], … M[14]
init_addr 00001101;	18340000	r_addr = 13	M[13]
ld_M_L;	14000000	01110110	LFSR=M[13]
init_addr 00010000;	18400000	r_addr=16	M[16]
batch_run_st_M_L 00001011;	28002C00	01000011	Continuously run LFSR for 11 cycles and write the patterns into M[16], M[17], … M[26]
halt;	00000000		

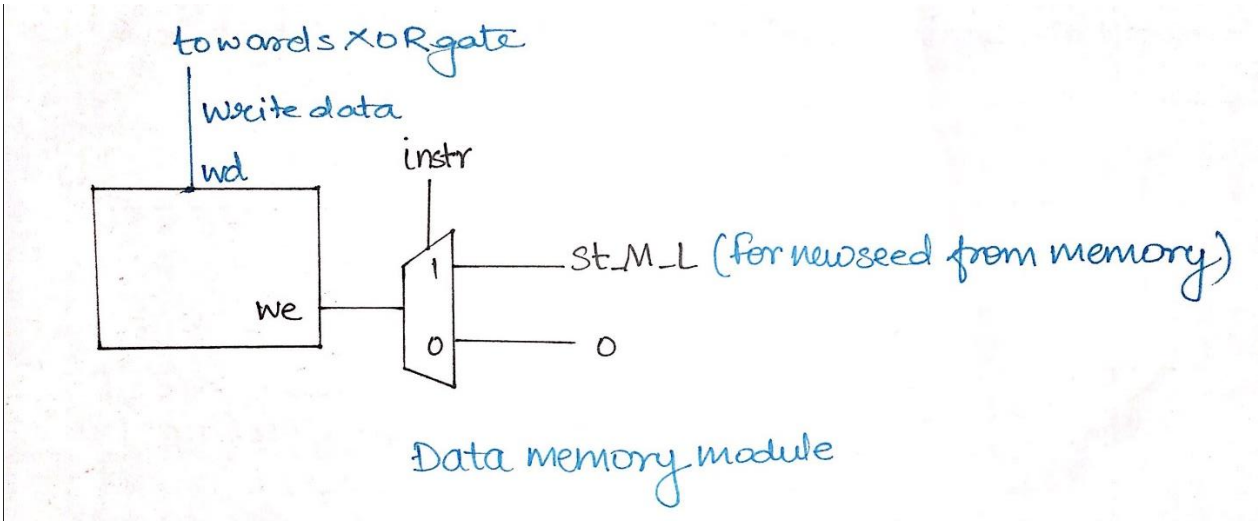
ii. Provide another testing program lfsr_t6b of your own design to feature the new instruction and use your software companion to fill in the table:

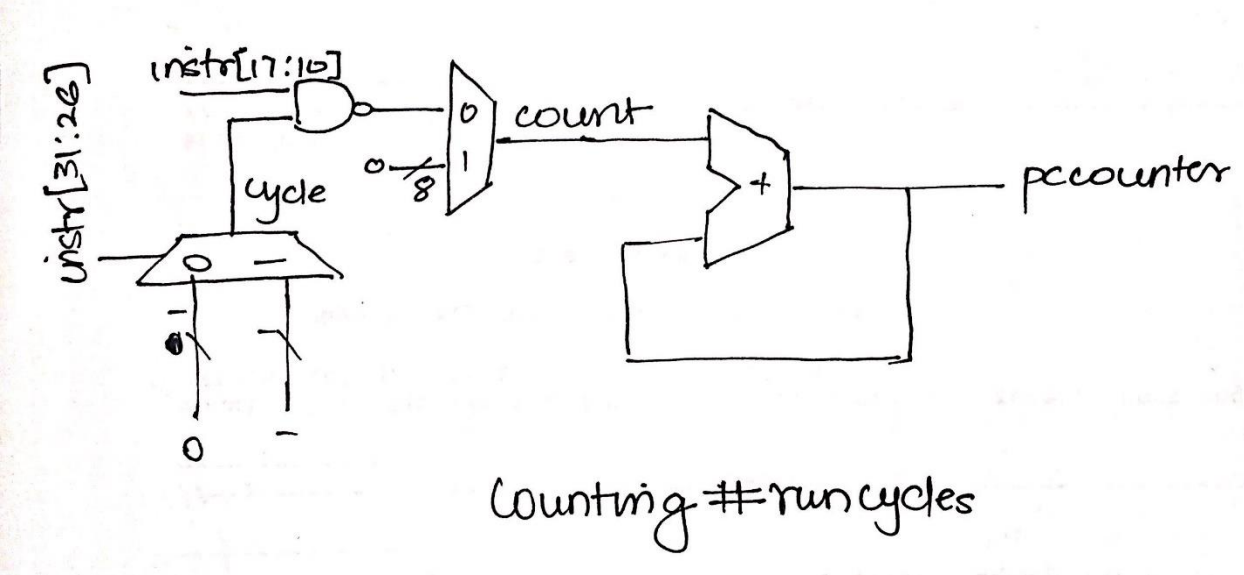
Assembly Language	Machine Code	LFSR Content or r_addr	Other info
config_L 1001010;	0400004A		
init_L 11111111;	0803FC00	11111111	
init_addr 00000100;	18100000	r_addr = 4	M[4]
batch_run_st_M_L 00000101;	28001400	00010010	Continuously run LFSR for 5 cycles and write the patterns into M[4], M[5], … M[9]
init_addr 00001000;	18200000	r_addr = 8	M[8]
ld_M_L;	14000000	00100100	LFSR=M[9]
halt;	00000000		

iii. Display your main modification to support this instruction upgrade in a sketch

iv.



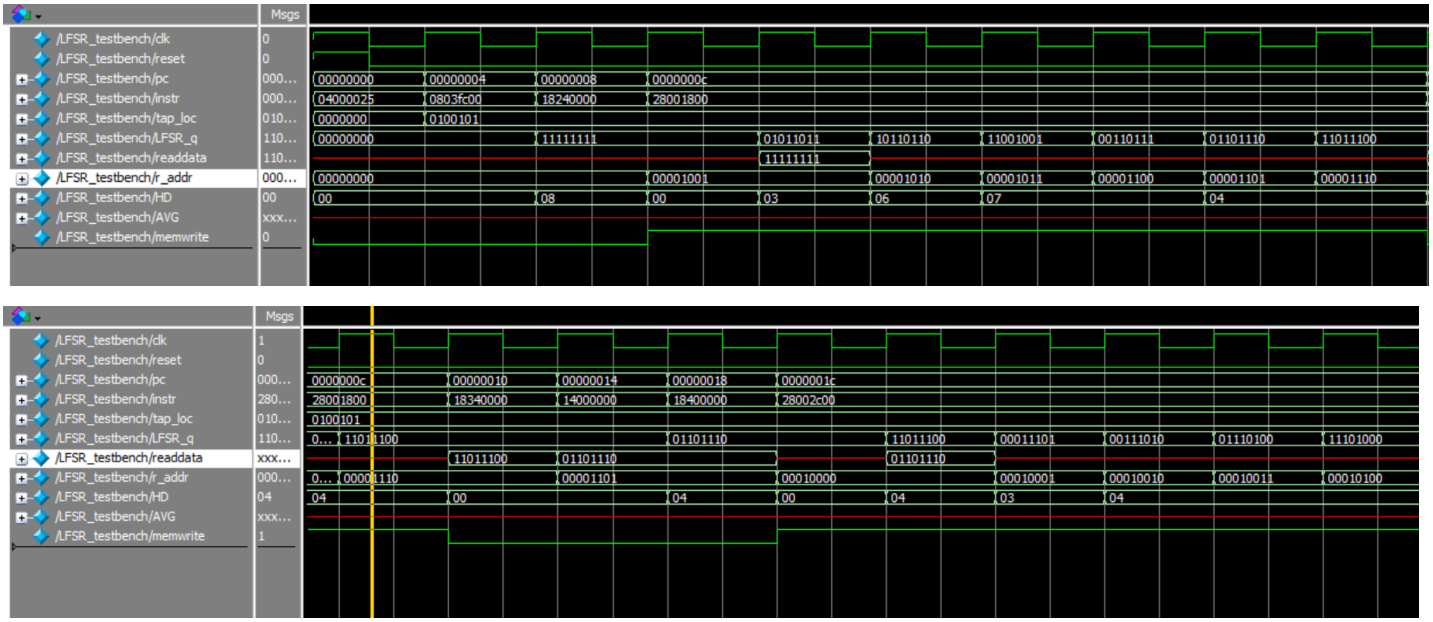


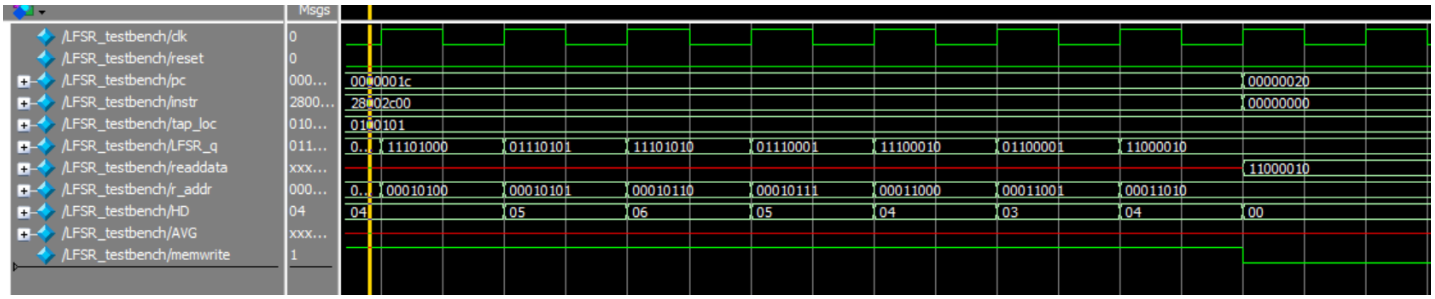


Along with the mechanism for multiple run cycles, the count here is set to one when the opcode instruction for storage is detected. And for every batch of run cycles, the LFSR content is simultaneously stored in memory address which also increments.

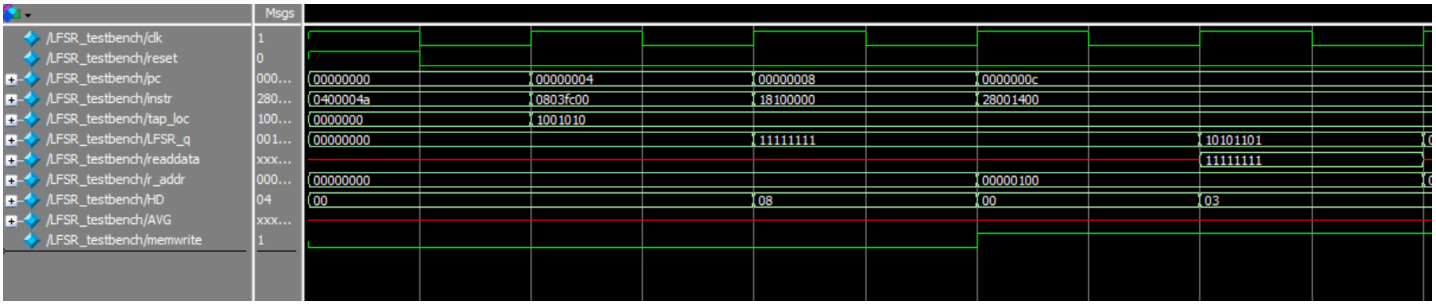
v. Run both testing programs lfsr_t6a and lfsr_t6b with ModelSim to verify that your design works correctly. Show the waveform results.

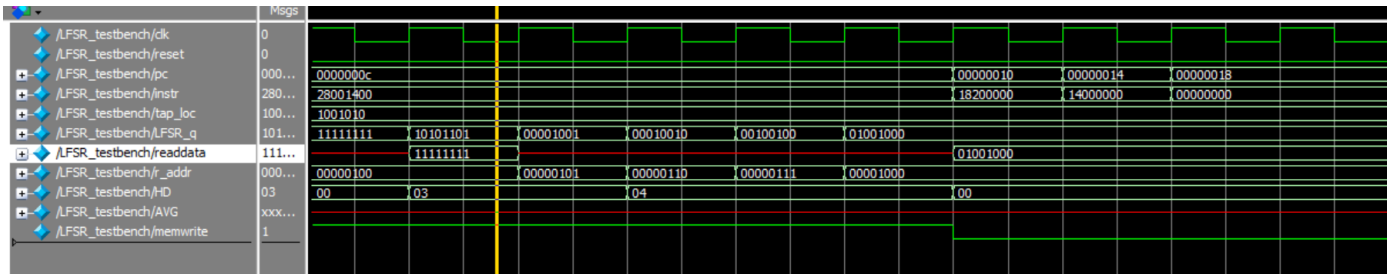
Waveform for lfsr_t6a



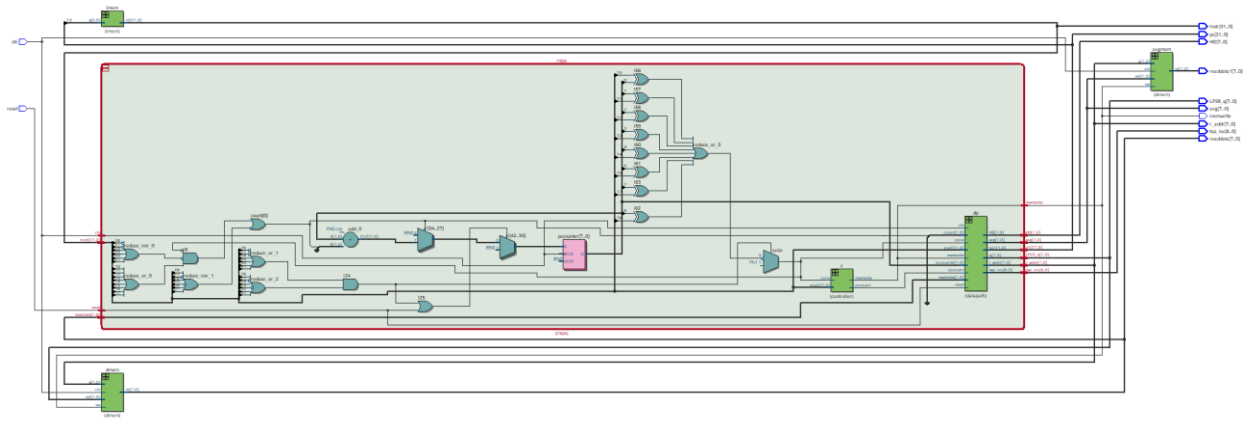


Waveform for lfsr_t6b





RTL View



APPENDIX 1

SystemVerilog Code

```
module lfsr(input logic clk, reset,
output logic [7:0] LFSR_q, r_addr,readdata,
output logic memwrite,
output logic [31:0] pc,instr,
output logic [6:0] tap_loc);



logic [7:0] aluout;

mips mips(clk,reset,pc,instr,memwrite,r_addr,LFSR_q,readdata,tap_loc);

imem imem(pc[7:2], instr);

dmem dmem(clk, memwrite, r_addr, LFSR_q, readdata);
endmodule
//-------------------------------------------------------------------------

//---------------------------------DATA MEMORY---------------------------

module dmem(input logic clk, we,
input logic [7:0] a, wd,
output logic [7:0] rd);

logic [7:0] RAM[255:0];

assign rd = RAM[a];

always_ff @(posedge clk)
if (we) RAM[a] <= wd;

endmodule

//----------------------------INSTRUCTION MEMORY-------------------------
module imem(input logic [5:0]  a, output logic [31:0] rd);

logic [31:0] RAM[63:0];

initial

$readmemh ("C:/intelFPGA/17.1/PROJALU/memfilelfsr.dat",RAM);

assign rd=RAM[a];
```

```
endmodule

//------------------------------------------------------------------------
//mips mips(clk,reset,pc,instr,memwrite,r_addr,LFSR_q,readdata,tap_loc,HD);
//----------------------------INSTRUCTION MEMORY-------------------------

module mips(input logic clk, reset,
output logic [31:0] pc,
input logic [31:0]instr,
output logic memwrite,
output logic [7:0] r_addr, LFSR_q,
input logic [7:0] readdata,
output logic [6:0] tap_loc);


controller c(instr[31:26],memwrite,pccount);

datapath dp(clk, reset, instr, readdata,memwrite,pccount, LFSR_q, pc,r_addr,tap_loc);
endmodule

//---------------------------------CONTROLLER---------------------------------------

module controller(input logic [5:0] op,
output logic memwrite,pccount);


maindec md(op,memwrite,pccount);

endmodule

//------------------------------------------------------------------------------------

//---------------------------------Main Decoder---------------------------------------

module maindec(input logic [5:0] op,
output logic memwrite,pccount);

logic [1:0] controls;

assign {memwrite, pccount} = controls;

always_comb
case(op)
6'b000000: controls <= 2'b01; // halt
6'b000001: controls <= 2'b00; // config_l
```

```
6'b000010: controls <= 2'b00; // init_L
6'b000011: controls <= 2'b00; // run_L
6'b000100: controls <= 2'b10; // st_M_L
6'b000101: controls <= 2'b00; // ld_M_L
6'b000110: controls <= 2'b00; // init_addr_loc
6'b000111: controls <= 2'b00; // add_addr_num
default: controls <= 2'bxx; // illegal op
endcase

endmodule
//------------------------------------------------------------------------------
//------------------------------------------------------------------------------
module datapath(input logic clk, reset,
input logic [31:0]instr,
input logic [7:0] readdata,
input logic memwrite,
input logic pccount, output logic[7:0] q,
output logic [31:0] pc,
output logic [7:0] r_addr,
output logic [6:0] tap_loc);



logic [31:0] pcnext, pcplus4;

chooseadd cadd (clk,reset,instr,r_addr);


logic temp;
assign temp = q[7];

always_ff @(posedge clk)
if(reset)
q <=0;
else if( instr [31:26] === 6'b000010)
q <= instr [17:10];
else if (instr [31:26] === 6'b000101)
q <= readdata;
else if (instr [31:26] === 6'b000011)
begin
if (tap_loc[0] === 1'b1)
q[7] <= q[6] ^ temp;
else
q[7] <= q[6];
if (tap_loc[1] === 1'b1)
q[6] <= q[5] ^ temp;
```

```
else
q[6] <= q[5];
if (tap_loc[2] === 1'b1)
q[5] <= q[4] ^ temp;
else
q[5] <= q[4];
if (tap_loc[3] === 1'b1)
q[4] <= q[3] ^ temp;
else
q[4] <= q[3];
if (tap_loc[4] === 1'b1)
q[3] <= q[2] ^ temp;
else
q[3] <= q[2];
if (tap_loc[5] === 1'b1)
q[2] <= q[1] ^ temp;
else
q[2] <= q[1];
if (tap_loc[6] === 1'b1)
q[1] <= q[0] ^ temp;
else
begin
q[1] <= q[0];
end
q[0] <= temp;
end
else
q <= q;

flopr #(32) pcreg(clk, reset, pcnext, pc);

adder pcadd1(pc, 32'b100, pcnext);

configure configure(clk,reset,instr, tap_loc);

endmodule

//------------------------------------------------------------------------------
//--------------------------------Config----------------------------------------
module configure (input logic clk, reset,input logic [31:0] instr, output logic [6:0] tap_loc);

always_ff @ (posedge clk)
if (reset)
tap_loc <= 0;

else
```

```
begin
case(instr[31:26])

6'b000001: tap_loc <= instr[6:0];
default: tap_loc <= tap_loc;

endcase
end
endmodule
//-------------------------------------------------------------------------------
//-------------------------Choose r_addr-----------------------------------------
module chooseadd(input logic clk,reset,input logic [31:0] instr, output logic [7:0] r_addr);

always_ff @ (posedge clk)
if (reset)
r_addr <= 0;

else if (instr [31:26] === 6'b000110)
r_addr <= instr[25:18];
else if ( instr [31:26] === 6'b000111)
r_addr <= r_addr + instr [17:10];
else
r_addr <= r_addr;

endmodule

//-------------------------------------------------------------------------------
//---------------------------------ADDER-----------------------------------------

module adder(input logic [31:0] a, b,  output logic [31:0] y);

assign y=a+b;

endmodule
//-------------------------------------------------------------------------------
//---------------------------------FLIP FLOP TO STORE----------------------------------

module flopr #(parameter WIDTH=8) (input logic clk, reset, input logic [WIDTH-1:0] d, output logic [WIDTH-
1:0] q);

always_ff @(posedge clk, posedge reset)

if (reset) q <= 0;

else q <= d;
```

```
endmodule
//-------------------------------------------------------------------------------
```

SystemVerilog Testbench Code

```
module LFSR_testbench();
logic clk;
logic reset;
logic [7:0] LFSR_q,r_addr;
logic memwrite;
logic [31:0] pc, instr;
logic [7:0] readdata;
logic [6:0] tap_loc;

lfsr dut (clk, reset,LFSR_q, r_addr,readdata,memwrite,pc,instr,tap_loc);

initial
begin
reset <= 1; # 5; reset <= 0;
end


always
begin
clk <= 1; # 5; clk <= 0; # 5;
end

endmodule
```

APPENDIX 2

```
package project2;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;

public class project2{
public static void main(String[] args) throws IOException {
ArrayList<Integer> code32bit = new ArrayList<Integer>();
ArrayList<Integer> operation = new ArrayList<Integer>();
ArrayList<Integer> register = new ArrayList<Integer>();
ArrayList<Integer> location = new ArrayList<Integer>();
ArrayList<Integer> tap = new ArrayList<Integer>();
ArrayList<Integer> seed = new ArrayList<Integer>();
ArrayList<Integer> num = new ArrayList<Integer>();
int decimalRegister = 0;
int decimalNum = 0;
int op = 0;
ArrayList<Integer> Q = new ArrayList<Integer>();
ArrayList<Integer> output = new ArrayList<Integer>();
ArrayList<Integer> previousQ = new ArrayList<Integer>();
ArrayList<ArrayList<Integer>> memory = new ArrayList<ArrayList<Integer>>();
String file="lfst_t1.txt";
String l = null;
FileReader f = new FileReader(file);
BufferedReader b = new BufferedReader(f);
char[] code32bitCharacterArray = null;
for(int i=0; i<8; i++){
previousQ.add(0);
seed.add(0);
output.add(0);
Q.add(0);
location.add(0);
register.add(0);
num.add(0);
}
for(int i=0; i<6; i++){
operation.add(0);
}
for(int i=0; i<7; i++){
tap.add(0);
}
for(int i=0; i<32; i++){
code32bit.add(0);
```

```
}
for(int i=0; i<1000; i++){
memory.add(Q);
}
while((l = b.readLine())!= null){
code32bitCharacterArray = l.toCharArray();
for(int i=0; i<code32bitCharacterArray.length; i++){
code32bit.set(i, code32bitCharacterArray[i] - '0');
}
for(int i = 0; i<6; i++){
operation.add(i,code32bit.get(i));
}
for(int i=0; i<6; i++){
op = (int) (op + operation.get(5-i)*Math.pow(2, i));
}
if(op==1){
for(int i=0; i<7; i++){
tap.set(i,code32bit.get(i+25));
}
}
else if(op==2){
for(int i=0; i<8; i++){
seed.set(i,code32bit.get(i+14));
output.set(i,seed.get(i));
previousQ.set(i,seed.get(i));
}
}
else if(op==3){
Q.set(0, previousQ.get(7));
for(int i=1; i<8; i++){
if(tap.get(i-1) == 1){

Q.set(i,((previousQ.get(i-1))^(previousQ.get(7))));
}
else{
Q.set(i,(previousQ.get(i-1)));
}
}
for(int i=0; i<8; i++){
output.set(i, Q.get(i));
previousQ.set(i, Q.get(i));
}
}
else if(op==6){
for(int i=0; i<8; i++){
location.add(i,code32bit.get(i+6));
```

```
register.set(i, location.get(i));
}
}
else if(op==4){
for(int i=0; i<8; i++){
decimalRegister = (int) (decimalRegister + register.get(7-i)*Math.pow(2, i));
}
memory.set(decimalRegister,output);
System.out.print("Memory Content:"+memory.get(decimalRegister));
}
else if(op==5){
ArrayList<Integer> x = new ArrayList<Integer>();
x = memory.get(decimalRegister);
for(int i=0; i<8; i++){
seed.set(i,(x.get(i)));
}
}
else if(op==7){
for(int i=0; i<8; i++){
num.set(i,code32bit.get(i+14));
}
for(int i=0; i<8; i++){
decimalRegister = (int) (decimalRegister + register.get(7-i)*Math.pow(2, i));
}
for(int i=0; i<8; i++){
decimalNum = (int) (decimalNum + num.get(i)*Math.pow(2, i));
}
decimalRegister  = decimalRegister + decimalNum;
System.out.print("Memory - "+memory.get(decimalRegister));
}
System.out.println("LFSR  - "+output);
operation.clear();
location.clear();
op = 0;
decimalNum = 0;
decimalRegister = 0;

}
b.close();
f.close();
}

}
```

APPENDIX 3

SystemVerilog Code

```
module lfsrhd(input logic clk, reset,
output logic [7:0] LFSR_q, r_addr,readdata,
output logic memwrite,
output logic [31:0] pc,instr,
output logic [6:0] tap_loc,
output logic [7:0] HD,st_M_HDdata);


logic [7:0] aluout;

mips mips(clk,reset,pc,instr,memwrite,r_addr,LFSR_q,readdata,st_M_HDdata,tap_loc,HD);

imem imem(pc[7:2], instr);

dmem dmem(clk, memwrite, instr,r_addr, LFSR_q, readdata);

sdmem HDmem(clk,memwrite,instr,r_addr, HD,st_M_HDdata);
endmodule
//------------------------------------------------------------------------

//---------------------------------DATA MEMORY---------------------------

module dmem(input logic clk, we, input logic [31:0] instr,
input logic [7:0] a, wd,
output logic [7:0] rd);

logic [7:0] RAM[255:0];

assign rd = RAM[a];

always_ff @(posedge clk)
if (we & instr[31:26]=== 6'b000100) RAM[a] <= wd;

endmodule

//---------------------------INSTRUCTION MEMORY--------------------------
module imem(input logic [5:0]  a, output logic [31:0] rd);

logic [31:0] RAM[63:0];

initial
```

```
$readmemh ("C:/intelFPGA/17.1/PROJALU/memfilelfsr.dat",RAM);

assign rd=RAM[a];

endmodule

//-------------------------------------------------------------------------
//mips mips(clk,reset,pc,instr,memwrite,r_addr,LFSR_q,readdata,tap_loc,HD);
//----------------------------INSTRUCTION MEMORY-------------------------

module mips(input logic clk, reset,
output logic [31:0] pc,
input logic [31:0]instr,
output logic memwrite,
output logic [7:0] r_addr, LFSR_q,
input logic [7:0] readdata,st_M_HDdata,
output logic [6:0] tap_loc,
output logic [7:0] HD);


//logic [2:0] alucontrol;

controller c(instr[31:26],memwrite,pccount);

datapath dp(clk, reset, instr, readdata,st_M_HDdata,memwrite,pccount, LFSR_q, pc,HD,r_addr,tap_loc);
endmodule

//---------------------------------CONTROLLER---------------------------------------

module controller(input logic [5:0] op,
output logic memwrite,pccount);


maindec md(op,memwrite,pccount);


endmodule

//----------------------------------------------------------------------------------

//---------------------------------Main Decoder---------------------------------------

module maindec(input logic [5:0] op,
output logic memwrite,pccount);

logic [1:0] controls;
```

```
assign {memwrite, pccount} = controls;

always_comb
case(op)
6'b000000: controls <= 2'b01; // halt
6'b000001: controls <= 2'b00; // config_l
6'b000010: controls <= 2'b00; // init_L
6'b000011: controls <= 2'b00; // run_L
6'b000100: controls <= 2'b10; // st_M_L
6'b000101: controls <= 2'b00; // ld_M_L
6'b000110: controls <= 2'b00; // init_addr_loc
6'b000111: controls <= 2'b00; // add_addr_num
6'b001000: controls <= 2'b10; // st_M_HD
default: controls <= 2'bxx; // illegal op
endcase

endmodule
//------------------------------------------------------------------------------
//------------------------------------------------------------------------------
module datapath(input logic clk, reset,
input logic [31:0]instr,
input logic [7:0] readdata,st_M_HDdata,
input logic memwrite,
input logic pccount, output logic[7:0] q,
output logic [31:0] pc,
output logic [7:0] HD,r_addr,
output logic [6:0] tap_loc );



logic [31:0] pcnext, pcplus4;

chooseadd cadd (clk,reset,instr,r_addr);



logic temp;
assign temp = q[7];

always_ff @(posedge clk)
if(reset)
q <=0;
else if( instr [31:26] === 6'b000010)
q <= instr [17:10];
else if (instr [31:26] === 6'b000101)
```

```
q <= readdata;
else if (instr [31:26] === 6'b001000)
q <= st_M_HDdata;
else if (instr [31:26] === 6'b000011)
begin
if (tap_loc[0] === 1'b1)
q[7] <= q[6] ^ temp;
else
q[7] <= q[6];
if (tap_loc[1] === 1'b1)
q[6] <= q[5] ^ temp;
else
q[6] <= q[5];
if (tap_loc[2] === 1'b1)
q[5] <= q[4] ^ temp;
else
q[5] <= q[4];
if (tap_loc[3] === 1'b1)
q[4] <= q[3] ^ temp;
else
q[4] <= q[3];
if (tap_loc[4] === 1'b1)
q[3] <= q[2] ^ temp;
else
q[3] <= q[2];
if (tap_loc[5] === 1'b1)
q[2] <= q[1] ^ temp;
else
q[2] <= q[1];
if (tap_loc[6] === 1'b1)
q[1] <= q[0] ^ temp;
else
begin
q[1] <= q[0];
end
q[0] <= temp;
end
else
q <= q;

flopr #(32) pcreg(clk, reset, pcnext, pc);

adder pcadd1(pc, 32'b100, pcnext);

configure configure(clk,reset,instr, tap_loc);
```

```
logic [7:0] tmpry;


flopr #(32) hdcal(clk, reset, q, q_old);

assign tmpry = q_old;

hdcalc hd1 (clk,reset,q,tmpry,HD);

endmodule

//-------------------------------------------------------------------------------
//--------------------------------Config-----------------------------------------
module configure (input logic clk, reset,input logic [31:0] instr, output logic [6:0] tap_loc);

always_ff @ (posedge clk)
if (reset)
tap_loc <= 0;

else
begin
case(instr[31:26])

6'b000001: tap_loc <= instr[6:0];
default: tap_loc <= tap_loc;

endcase
end
endmodule
//-------------------------------------------------------------------------------
//--------------------------Choose r_addr----------------------------------------
module chooseadd(input logic clk,reset,input logic [31:0] instr, output logic [7:0] r_addr);

always_ff @ (posedge clk)
if (reset)
r_addr <= 0;

else if (instr [31:26] === 6'b000110)
r_addr <= instr[25:18];
else if ( instr [31:26] === 6'b000111)
r_addr <= r_addr + instr [17:10];
else if (instr [31:26] === 6'b001001)
r_addr <= 8'b00000000;
else
r_addr <= r_addr;
```

```
endmodule

//-------------------------------------------------------------------------------
//--------------------------------ADDER-----------------------------------------

module adder(input logic [31:0] a, b,  output logic [31:0] y);

assign y=a+b;

endmodule
//-------------------------------------------------------------------------------
//--------------------------------FLIP FLOP TO STORE----------------------------------

module flopr #(parameter WIDTH=8) (input logic clk, reset, input logic [WIDTH-1:0] d, output logic [WIDTH-1:0] q);

always_ff @(posedge clk, posedge reset)

if (reset) q <= 0;

else q <= d;

endmodule
//-------------------------------------------------------------------------------
//--------------------------------HAMMING DISTANCE MEMORY----------------------------

module sdmem(input logic clk, we,input logic [31:0] instr, input logic [7:0] a, wd,output logic [7:0] st_M_HDdata);

logic [7:0] RAM[255:0];

assign st_M_HDdata = RAM[a];


always_ff @(posedge clk)

if (we & instr[31:26] === 6'b001000) RAM[a] <= wd;

endmodule

//-------------------------------------------------------------------------

//------------------------HAMMING DISTANCE CALCULATOR-----------------------------------
module hdcalc (inout logic clk,reset,input logic [7:0] q,q_old, output logic [7:0] HD);

logic hd0,hd1,hd2,hd3,hd4,hd5,hd6,hd7,hd8;
```

```
logic tmp;

always_comb
if( q_old === q )
tmp=1'b1;
else tmp = 1'b0;

always_comb
if ( q_old[0] === q[0])
hd0 = 1'b0;
else hd0 = 1'b1;

always_comb
if ( q_old[1] === q[1])
hd1 = 1'b0;
else hd1 = 1'b1;

always_comb
if ( q_old[2] === q[2])
hd2 = 1'b0;
else hd2 = 1'b1;

always_comb
if ( q_old[3] === q[3])
hd3 = 1'b0;
else hd3 = 1'b1;

always_comb
if ( q_old[4] === q[4])
hd4 = 1'b0;
else hd4 = 1'b1;

always_comb
if ( q_old[5] === q[5])
hd5 = 1'b0;
else hd5 = 1'b1;

always_comb
if ( q_old[6] === q[6])
hd6 = 1'b0;
else hd6 = 1'b1;

always_comb
if ( q_old[7] === q[7])
```

```
hd7 = 1'b0;
else hd7 = 1'b1;



always_ff @(posedge clk)
if (tmp )
HD <= HD;
else
HD <= hd0+hd1+hd2+hd3+hd4+hd5+hd6+hd7;

endmodule

//------------------------------------------------------------------------------
```

SystemVerilog Testbench Code

```
module LFSRhd_testbench();
logic clk;
logic reset;
logic [7:0] LFSR_q,r_addr;
logic memwrite;
logic [31:0] pc, instr;
logic [7:0] readdata;
logic [6:0] tap_loc;
logic [7:0] HD,st_m_HDdata;

lfsrhd dut (clk, reset,LFSR_q, r_addr,readdata,memwrite,pc,instr,tap_loc,HD,st_m_HDdata);

initial
begin
reset <= 1; # 5; reset <= 0;
end


always
begin
clk <= 1; # 5; clk <= 0; # 5;
end

endmodule
```

APPENDIX 4
SystemVerilog Code

```
module lfsravg(input logic clk, reset,
output logic [7:0] LFSR_q, r_addr,readdata,
output logic memwrite,
output logic [31:0] pc,instr,
output logic [6:0] tap_loc,
output logic [7:0] HD,avg);


logic [7:0] aluout;

mips mips(clk,reset,pc,instr,memwrite,r_addr,LFSR_q,readdata,tap_loc,HD,avg);

imem imem(pc[7:2], instr);

dmem dmem(clk, memwrite, r_addr, LFSR_q, readdata);

dmem avgmem (clk,memwrite, r_addr , avg, readdata);
endmodule
//-------------------------------------------------------------------------

//---------------------------------DATA MEMORY---------------------------

module dmem(input logic clk, we,
input logic [7:0] a, wd,
output logic [7:0] rd);

logic [7:0] RAM[255:0];

assign rd = RAM[a];

always_ff @(posedge clk)
if (we) RAM[a] <= wd;

endmodule

//----------------------------INSTRUCTION MEMORY--------------------------
module imem(input logic [5:0]  a, output logic [31:0] rd);

logic [31:0] RAM[63:0];

initial

$readmemh ("C:/intelFPGA/17.1/PROJALU/memfilelfsr.dat",RAM);
```

```
assign rd=RAM[a];

endmodule

//-----------------------------------------------------------------------

//---------------------------INSTRUCTION MEMORY-------------------------

module mips(input logic clk, reset,
output logic [31:0] pc,
input logic [31:0]instr,
output logic memwrite,
output logic [7:0] r_addr, LFSR_q,
input logic [7:0] readdata,
output logic [6:0] tap_loc,
output logic [7:0] HD,avg);


controller c(instr[31:26],memwrite,pccount);

datapath dp(clk, reset, instr, readdata,memwrite,pccount, LFSR_q, pc,HD,r_addr,avg,tap_loc);
endmodule

//---------------------------------CONTROLLER---------------------------------------

module controller(input logic [5:0] op,
output logic memwrite,pccount);


maindec md(op,memwrite,pccount);


endmodule

//-----------------------------------------------------------------------------------

//---------------------------------Main Decoder---------------------------------------

module maindec(input logic [5:0] op,
output logic memwrite,pccount);

logic [1:0] controls;

assign {memwrite, pccount} = controls;
```

```
always_comb
case(op)
6'b000000: controls <= 2'b01; // halt
6'b000001: controls <= 2'b00; // config_l
6'b000010: controls <= 2'b00; // init_L
6'b000011: controls <= 2'b00; // run_L
6'b000100: controls <= 2'b10; // st_M_L
6'b000101: controls <= 2'b00; // ld_M_L
6'b000110: controls <= 2'b00; // init_addr_loc
6'b000111: controls <= 2'b00; // add_addr_num
6'b001000: controls <= 2'b10; // st_M_HD
6'b001001: controls <= 2'b10; //avg_M_HD
default: controls <= 2'bxx; // illegal op
endcase

endmodule
//------------------------------------------------------------------------------
//------------------------------------------------------------------------------
module datapath(input logic clk, reset,
input logic [31:0]instr,
input logic [7:0] readdata,
input logic memwrite,
input logic pccount, output logic[7:0] q,
output logic [31:0] pc,
output logic [7:0] HD,r_addr,avg,
output logic [6:0] tap_loc);



logic [31:0] pcnext, pcplus4;

chooseadd cadd (clk,reset,instr,r_addr);

logic temp;
assign temp = q[7];

always_ff @(posedge clk)
if(reset)
q <=0;
else if( instr [31:26] === 6'b000010)
q <= instr [17:10];
else if (instr [31:26] === 6'b000101)
q <= readdata;
else if (instr [31:26] === 6'b000011)
begin
if (tap_loc[0] === 1'b1)
```

```
q[7] <= q[6] ^ temp;
else
q[7] <= q[6];
if (tap_loc[1] === 1'b1)
q[6] <= q[5] ^ temp;
else
q[6] <= q[5];
if (tap_loc[2] === 1'b1)
q[5] <= q[4] ^ temp;
else
q[5] <= q[4];
if (tap_loc[3] === 1'b1)
q[4] <= q[3] ^ temp;
else
q[4] <= q[3];
if (tap_loc[4] === 1'b1)
q[3] <= q[2] ^ temp;
else
q[3] <= q[2];
if (tap_loc[5] === 1'b1)
q[2] <= q[1] ^ temp;
else
q[2] <= q[1];
if (tap_loc[6] === 1'b1)
q[1] <= q[0] ^ temp;
else
begin
q[1] <= q[0];
end
q[0] <= temp;
end
else
q <= q;

//------------------------HAMMING DISTANCE CALCULATOR-----------------------------------
logic [7:0] q_old;
logic hd0,hd1,hd2,hd3,hd4,hd5,hd6,hd7,hd8;

flopr #(32) hdcal(clk, reset, q, q_old);


always_comb
if (instr[31:26] != 6'b000010)
begin
if ( q_old[0] === q[0])
hd0 = 1'b0;
```

```
else hd0 = 1'b1;

if ( q_old[1] === q[1])
hd1 = 1'b0;
else hd1 = 1'b1;


if ( q_old[2] === q[2])
hd2 = 1'b0;
else hd2 = 1'b1;


if ( q_old[3] === q[3])
hd3 = 1'b0;
else hd3 = 1'b1;


if ( q_old[4] === q[4])
hd4 = 1'b0;
else hd4 = 1'b1;


if ( q_old[5] === q[5])
hd5 = 1'b0;
else hd5 = 1'b1;


if ( q_old[6] === q[6])
hd6 = 1'b0;
else hd6 = 1'b1;


if ( q_old[7] === q[7])
hd7 = 1'b0;
else hd7 = 1'b1;

end

assign HD = hd0+hd1+hd2+hd3+hd4+hd5+hd6+hd7;

avgcal ac1(clk,reset,instr,HD,avg);


//---------------------------------------------------------------------------

flopr #(32) pcreg(clk, reset, pcnext, pc);
```

```
adder pcadd1(pc, 32'b100, pcnext);

configure configure(clk,reset,instr, tap_loc);

endmodule

//-------------------------------------------------------------------------------
//--------------------------Choose r_addr-----------------------------------------
module chooseadd(input logic clk,reset,input logic [31:0] instr, output logic [7:0] r_addr);

always_ff @ (posedge clk)
if (reset)
r_addr <= 0;

else if (instr [31:26] === 6'b000110)
r_addr <= instr[25:18];
else if ( instr [31:26] === 6'b000111)
r_addr <= r_addr + instr [17:10];
else if (instr [31:26] === 6'b001001)
r_addr <= 8'b00000000;
else
r_addr <= r_addr;

endmodule

//-------------------------------------------------------------------------------
//-------------------------------------AVERAGE CALCULATOR----------------------------
module avgcal(input logic clk,reset,input logic [31:0] instr,input logic [7:0] HD, output logic [7:0] avg);

logic [7:0] rc,runcount,HD_TOTAL;

always_comb
if(instr[31:26] === 6'b000011)
rc = 8'b00000001;
else rc = 8'b00000000;

always_ff @(posedge clk)
if (reset)
begin
HD_TOTAL <= 0;
runcount <= 0;
end
else if (rc === 8'b00000001)
begin
HD_TOTAL <= HD_TOTAL + HD;
```

```
runcount <= runcount + rc;
end
else
begin
HD_TOTAL <= HD_TOTAL;
runcount <= runcount;
end

always_comb
if (instr [31:26] === 6'b001001)
avg = HD_TOTAL / runcount;

endmodule
//--------------------------------ADDER-----------------------------------------

module adder(input logic [31:0] a, b,  output logic [31:0] y);

assign y=a+b;
endmodule
//-------------------------------------------------------------------------------------
//--------------------------------Config-----------------------------------------
module configure (input logic clk, reset,input logic [31:0] instr, output logic [6:0] tap_loc);

always_ff @ (posedge clk)
if (reset)
tap_loc <= 0;

else
begin
case(instr[31:26])

6'b000001: tap_loc <= instr[6:0];
default: tap_loc <= tap_loc;

endcase
end
endmodule
//-------------------------------------------------------------------------------
//--------------------------------FLIP FLOP TO STORE----------------------------------

module flopr #(parameter WIDTH=8) (input logic clk, reset, input logic [WIDTH-1:0] d, output logic [WIDTH-
1:0] q);

always_ff @(posedge clk, posedge reset)

if (reset) q <= 0;
```

```
else q <= d;

endmodule
//-------------------------------------------------------------------------------
//---------------------------------HAMMING DISTANCE MEMORY---------------------------

module sdmem(input logic clk, we, input logic [7:0] a, wd);

logic [7:0] RAM[255:0];

always_ff @(posedge clk)

if (we) RAM[a] <= wd;
endmodule

//------------------------------------------------------------------------
```

SystemVerilog Testbench Code

```
module LFSRavg_testbench();
logic clk;
logic reset;
logic [7:0] LFSR_q,r_addr;
logic memwrite;
logic [31:0] pc, instr;
logic [7:0] readdata;
logic [6:0] tap_loc;
logic [7:0] HD,AVG;

lfsravg dut (clk, reset,LFSR_q, r_addr,readdata,memwrite,pc,instr,tap_loc,HD,AVG);

initial
begin
reset <= 1; # 5; reset <= 0;
end

always
begin
clk <= 1; # 5; clk <= 0; # 5;
end

endmodule
```

APPENDIX 5

SystemVerilog Code

```
module lfsrmulticycle(input logic clk, reset,
output logic [7:0] LFSR_q, r_addr,readdata,
output logic memwrite,
output logic [31:0] pc,instr,
output logic [6:0] tap_loc,
output logic [7:0] HD,avg);


logic [7:0] aluout;

mips mips(clk,reset,pc,instr,memwrite,r_addr,LFSR_q,readdata,tap_loc,HD,avg);

imem imem(pc[7:2], instr);

dmem dmem(clk, memwrite, r_addr, LFSR_q, readdata);

dmem avgmem (clk,memwrite, r_addr , avg, readdata);
endmodule
//------------------------------------------------------------------------

//---------------------------------DATA MEMORY---------------------------

module dmem(input logic clk, we,
input logic [7:0] a, wd,
output logic [7:0] rd);

logic [7:0] RAM[255:0];
```

```
assign rd = RAM[a];

always_ff @(posedge clk)
if (we) RAM[a] <= wd;

endmodule

//----------------------------INSTRUCTION MEMORY-------------------------
module imem(input logic [5:0]  a, output logic [31:0] rd);

logic [31:0] RAM[63:0];

initial

$readmemh ("C:/intelFPGA/17.1/PROJALU/memfilelfsr.dat",RAM);

assign rd=RAM[a];

endmodule

//-----------------------------------------------------------------------

//----------------------------INSTRUCTION MEMORY-------------------------

module mips(input logic clk, reset,
output logic [31:0] pc,
input logic [31:0]instr,
output logic memwrite,
output logic [7:0] r_addr, LFSR_q,
input logic [7:0] readdata,
```

```
output logic [6:0] tap_loc,
output logic [7:0] HD,avg);

logic [7:0] count,pccounter;
logic cycle;

always_comb
if( instr[31:26] === 6'b000011 &&  instr [17:10] !== 8'b0 && cycle !== 1'b1)
count = 8'b00000001;
else count = 8'b0;

always_ff @(posedge clk)
if(reset || instr [31:26] !== 6'b000011)
pccounter <=0;
else if(count === 8'b0000001)
pccounter <= pccounter + count;
else
pccounter <=0;

always_comb
if (instr[31:26] !== 6'b000011)
cycle = 1'b1;

else if(pccounter !== instr[17:10])
cycle = 1'b0;

else cycle = 1'b1;

controller c(instr,cycle,memwrite,pccount);
```

```
datapath dp(clk, reset, instr, readdata,memwrite,pccount, LFSR_q, pc,HD,r_addr,avg,tap_loc,cycle);
endmodule

//---------------------------------CONTROLLER---------------------------------------

module controller(input logic [31:0] instr,input logic cycle,
output logic memwrite,pccount);


maindec md(instr,cycle,memwrite,pccount);


endmodule

//----------------------------------------------------------------------------------

//---------------------------------Main Decoder---------------------------------------

module maindec( input logic [31:0] instr,input logic cycle,
output logic memwrite,pccount);

logic [1:0] controls;

assign {memwrite, pccount} = controls;

always_comb
case(instr[31:26])
6'b000000: controls <= 2'b01; // halt
6'b000001: controls <= 2'b00; // config_l
```

```
6'b000010: controls <= 2'b00; // init_L
6'b000011: begin
if( cycle === 1'b1)
controls = 2'b00;
else controls = 2'b01;
end
6'b000100: controls <= 2'b10; // st_M_L
6'b000101: controls <= 2'b00; // ld_M_L
6'b000110: controls <= 2'b00; // init_addr_loc
6'b000111: controls <= 2'b00; // add_addr_num
6'b001000: controls <= 2'b10; // st_M_HD
6'b001001: controls <= 2'b10; //avg_M_HD
default: controls <= 2'bxx; // illegal op
endcase

endmodule
//------------------------------------------------------------------------------
//------------------------------------------------------------------------------
module datapath(input logic clk, reset,
input logic [31:0]instr,
input logic [7:0] readdata,
input logic memwrite,
input logic pccount, output logic[7:0] q,
output logic [31:0] pc,
output logic [7:0] HD,r_addr,avg,
output logic [6:0] tap_loc,input logic cycle);



logic [31:0] pcnext, pcplus4;
```

```
chooseadd cadd (clk,reset,instr,r_addr);

logic temp;
assign temp = q[7];

always_ff @(posedge clk)
if(reset)
q <=0;
else if( instr [31:26] === 6'b000010)
q <= instr [17:10];
else if (instr [31:26] === 6'b000101)
q <= readdata;
else if (instr [31:26] === 6'b000011 && cycle !== 1'b1)
begin
if (tap_loc[0] === 1'b1)
q[7] <= q[6] ^ temp;
else
q[7] <= q[6];
if (tap_loc[1] === 1'b1)
q[6] <= q[5] ^ temp;
else
q[6] <= q[5];
if (tap_loc[2] === 1'b1)
q[5] <= q[4] ^ temp;
else
q[5] <= q[4];
if (tap_loc[3] === 1'b1)
q[4] <= q[3] ^ temp;
else
```

```
q[4] <= q[3];
if (tap_loc[4] === 1'b1)
q[3] <= q[2] ^ temp;
else
q[3] <= q[2];
if (tap_loc[5] === 1'b1)
q[2] <= q[1] ^ temp;
else
q[2] <= q[1];
if (tap_loc[6] === 1'b1)
q[1] <= q[0] ^ temp;
else
begin
q[1] <= q[0];
end
q[0] <= temp;
end
else
q <= q;

//------------------------HAMMING DISTANCE CALCULATOR-----------------------------------
logic [7:0] q_old;
logic hd0,hd1,hd2,hd3,hd4,hd5,hd6,hd7,hd8;

flopr #(32) hdcal(clk, reset, q, q_old);


always_comb
if (instr[31:26] != 6'b000010)
begin
```

```
if ( q_old[0] === q[0])
hd0 = 1'b0;
else hd0 = 1'b1;

if ( q_old[1] === q[1])
hd1 = 1'b0;
else hd1 = 1'b1;


if ( q_old[2] === q[2])
hd2 = 1'b0;
else hd2 = 1'b1;


if ( q_old[3] === q[3])
hd3 = 1'b0;
else hd3 = 1'b1;


if ( q_old[4] === q[4])
hd4 = 1'b0;
else hd4 = 1'b1;


if ( q_old[5] === q[5])
hd5 = 1'b0;
else hd5 = 1'b1;


if ( q_old[6] === q[6])
```

```
hd6 = 1'b0;
else hd6 = 1'b1;


if ( q_old[7] === q[7])
hd7 = 1'b0;
else hd7 = 1'b1;

end

assign HD = hd0+hd1+hd2+hd3+hd4+hd5+hd6+hd7;

avgcal ac1(clk,reset,instr,HD,avg);


//------------------------------------------------------------------------------

flopr #(32) pcreg(clk, reset, pcnext, pc);

adder pcadd1(pc, 32'b100, pcplus4);

configure configure(clk,reset,instr, tap_loc);

mux2 #(32) pcmux(pcplus4, pc, pccount, pcnext);

endmodule

//------------------------------------------------------------------------------
//--------------------------------MUX to choose pc-----------------------------------
module mux2 #(parameter WIDTH=8) (input logic [WIDTH-1:0] d0, d1,
```

```
input logic s,
output logic [WIDTH-1:0] y);

assign y=s ? d1 : d0;

endmodule
//-------------------------------------------------------------------------------

//-------------------------Choose r_addr---------------------------------------
module chooseadd(input logic clk,reset,input logic [31:0] instr, output logic [7:0] r_addr);

always_ff @ (posedge clk)
if (reset)
r_addr <= 0;

else if (instr [31:26] === 6'b000110)
r_addr <= instr[25:18];
else if ( instr [31:26] === 6'b000111)
r_addr <= r_addr + instr [17:10];
else if (instr [31:26] === 6'b001001)
r_addr <= 8'b00000000;
else
r_addr <= r_addr;

endmodule

//-------------------------------------------------------------------------------
//-----------------------------------AVERAGE CALCULATOR---------------------------
module avgcal(input logic clk,reset,input logic [31:0] instr,input logic [7:0] HD, output logic [7:0] avg);
```

```
logic [7:0] rc,runcount,HD_TOTAL;

always_comb
if(instr[31:26] === 6'b000011)
rc = 8'b00000001;
else rc = 8'b00000000;

always_ff @(posedge clk)
if (reset)
begin
HD_TOTAL <= 0;
runcount <= 0;
end
else if (rc === 8'b00000001)
begin
HD_TOTAL <= HD_TOTAL + HD;
runcount <= runcount + rc;
end
else
begin
HD_TOTAL <= HD_TOTAL;
runcount <= runcount;
end

always_comb
if (instr [31:26] === 6'b001001)
avg = HD_TOTAL / runcount;

endmodule
//--------------------------------ADDER----------------------------------------
```

```
module adder(input logic [31:0] a, b,  output logic [31:0] y);

assign y=a+b;
endmodule
//------------------------------------------------------------------------------
//--------------------------------Config----------------------------------------
module configure (input logic clk, reset,input logic [31:0] instr, output logic [6:0] tap_loc);

always_ff @ (posedge clk)
if (reset)
tap_loc <= 0;

else
begin
case(instr[31:26])

6'b000001: tap_loc <= instr[6:0];
default: tap_loc <= tap_loc;

endcase
end
endmodule
//------------------------------------------------------------------------------
//--------------------------------FLIP FLOP TO STORE----------------------------------

module flopr #(parameter WIDTH=8) (input logic clk, reset, input logic [WIDTH-1:0] d, output logic [WIDTH-1:0] q);

always_ff @(posedge clk, posedge reset)
```

```
if (reset) q <= 0;

else q <= d;

endmodule
//-------------------------------------------------------------------------------
//---------------------------------HAMMING DISTANCE MEMORY---------------------------

module sdmem(input logic clk, we, input logic [7:0] a, wd);

logic [7:0] RAM[255:0];

always_ff @(posedge clk)

if (we) RAM[a] <= wd;

endmodule

//------------------------------------------------------------------------
SystemVerilog Testbench Code
module LFSRmulticycle_testbench();
logic clk;
logic reset;
logic [7:0] LFSR_q,r_addr;
logic memwrite;
logic [31:0] pc, instr;
logic [7:0] readdata;
logic [6:0] tap_loc;
logic [7:0] HD,AVG;
```

```
lfsrmulticycle dut (clk, reset,LFSR_q, r_addr,readdata,memwrite,pc,instr,tap_loc,HD,AVG);

initial
begin
reset <= 1; # 5; reset <= 0;
end

always
begin
clk <= 1; # 5; clk <= 0; # 5;
end
endmodule
```

APPENDIX 6

SystemVerilog Code

```
module lfsrbatch(input logic clk, reset,
                 output logic [7:0] LFSR_q, r_addr,readdata,
                 output logic memwrite,
                 output logic [31:0] pc,instr,
                 output logic [6:0] tap_loc,
                 output logic [7:0] HD,avg);



logic [7:0] aluout;

          mips mips(clk,reset,pc,instr,memwrite,r_addr,LFSR_q,readdata,tap_loc,HD,avg);

          imem imem(pc[7:2], instr);

          dmem dmem(clk, memwrite, r_addr, LFSR_q, readdata);

        dmem avgmem (clk,memwrite, r_addr , avg, readdata);
endmodule
//------------------------------------------------------------------------

//---------------------------------DATA MEMORY---------------------------

module dmem(input logic clk, we,
          input logic [7:0] a, wd,
          output logic [7:0] rd);

logic [7:0] RAM[255:0];
```

```
assign rd = RAM[a];

always_ff @(posedge clk)
	if (we) RAM[a] <= wd;

endmodule

//----------------------------INSTRUCTION MEMORY-------------------------
module imem(input logic [5:0]  a, output logic [31:0] rd);

	logic [31:0] RAM[63:0];

	initial

		$readmemh ("C:/intelFPGA/17.1/PROJALU/memfilelfsr.dat",RAM);

		assign rd=RAM[a];

endmodule

//---------------------------------------------------------------------

//----------------------------INSTRUCTION MEMORY-------------------------

module mips(input logic clk, reset,
	output logic [31:0] pc,
	input logic [31:0]instr,
	output logic memwrite,
	output logic [7:0] r_addr, LFSR_q,
```

```
        input logic [7:0] readdata,
        output logic [6:0] tap_loc,
        output logic [7:0] HD,avg);

logic [7:0] count,pccounter;
logic cycle;

always_comb
if(( instr[31:26] === 6'b000011 &&  instr [17:10] !== 8'b00000000 && cycle !== 1'b1) || instr [31:26] === 6'b001010)
count = 8'b00000001;
else count = 8'b0;

always_ff @(posedge clk)
if( reset || (instr [31:26] !== 6'b000011 && instr [31:26] !== 6'b001010) )
    pccounter <= 0;
else if(count === 8'b0000001)
    pccounter <= pccounter + count;
else
    pccounter <=0;

always_comb
if (instr[31:26] !== 6'b000011 && instr [31:26] !== 6'b001010)
        cycle = 1'b1;

else if(pccounter !== instr[17:10])
       cycle = 1'b0;

else cycle = 1'b1;
```

```
controller c(instr,cycle,memwrite,pccount);

datapath dp(clk, reset, instr, readdata,memwrite,pccount, LFSR_q, pc,HD,r_addr,avg,tap_loc,cycle,pccounter,count);
endmodule

//---------------------------------CONTROLLER---------------------------------------

module controller(input logic [31:0] instr,input logic cycle,
                  output logic memwrite,pccount);

maindec md(instr,cycle,memwrite,pccount);

endmodule

//----------------------------------------------------------------------------------

//---------------------------------Main Decoder---------------------------------------

module maindec( input logic [31:0] instr,input logic cycle,
                output logic memwrite,pccount);

logic [1:0] controls;

assign {memwrite, pccount} = controls;

always_comb
```

```
        case(instr[31:26])
            6'b000000: controls <= 2'b01; // halt
        6'b000001: controls <= 2'b00; // config_l
            6'b000010: controls <= 2'b00; // init_L
            6'b000011: begin            //run_L
           if( cycle === 1'b1)
             controls = 2'b00;
           else controls = 2'b01;
           end
            6'b000100: controls <= 2'b10; // st_M_L
            6'b000101: controls <= 2'b00; // ld_M_L
            6'b000110: controls <= 2'b00; // init_addr_loc
            6'b000111: controls <= 2'b00; // add_addr_num
            6'b001000: controls <= 2'b10; // st_M_HD
        6'b001001: controls <= 2'b10; // avg_M_HD
        6'b001010: begin            // batch_run_st_M_L_cycle
            if( cycle === 1'b1)
              controls = 2'b10;
            else controls = 2'b11;
            end
        default: controls <= 2'bxx; // illegal op
        endcase

endmodule
//------------------------------------------------------------------------------
//------------------------------------------------------------------------------
module datapath(input logic clk, reset,
            input logic [31:0]instr,
            input logic [7:0] readdata,
            input logic memwrite,
```

```
            input logic pccount, output logic[7:0] q,
            output logic [31:0] pc,
        output logic [7:0] HD,r_addr,avg,
        output logic [6:0] tap_loc,input logic cycle,input logic [7:0] pccounter,count);



logic [31:0] pcnext, pcplus4;

chooseadd cadd (clk,reset,instr,pccounter,count,cycle,r_addr);

logic temp;
assign temp = q[7];

always_ff @(posedge clk)
if(reset)
  q <=0;
else if( instr [31:26] === 6'b000010)
  q <= instr [17:10];
else if (instr [31:26] === 6'b000101)
  q <= readdata;
else if ((instr [31:26] === 6'b000011 && cycle !== 1'b1)||(instr [31:26] === 6'b001010 && cycle !== 1'b1))
  begin
                    if (tap_loc[0] === 1'b1)
                      q[7] <= q[6] ^ temp;
                    else
                      q[7] <= q[6];
                    if (tap_loc[1] === 1'b1)
                      q[6] <= q[5] ^ temp;
                    else
```

```
                    q[6] <= q[5];
                      if (tap_loc[2] === 1'b1)
                        q[5] <= q[4] ^ temp;
                      else
                        q[5] <= q[4];
                    if (tap_loc[3] === 1'b1)
                        q[4] <= q[3] ^ temp;
                      else
                        q[4] <= q[3];
                      if (tap_loc[4] === 1'b1)
                        q[3] <= q[2] ^ temp;
                      else
                        q[3] <= q[2];
              if (tap_loc[5] === 1'b1)
                        q[2] <= q[1] ^ temp;
                      else
                        q[2] <= q[1];
                    if (tap_loc[6] === 1'b1)
                        q[1] <= q[0] ^ temp;
                      else
                  begin
                          q[1] <= q[0];
                  end
                     q[0] <= temp;
   end
else
q <= q;

//-----------------------HAMMING DISTANCE CALCULATOR----------------------------------
logic [7:0] q_old;
```

```
logic hd0,hd1,hd2,hd3,hd4,hd5,hd6,hd7,hd8;

flopr #(32) hdcal(clk, reset, q, q_old);


always_comb
if (instr[31:26] != 6'b000010)
begin
if ( q_old[0] === q[0])
   hd0 = 1'b0;
else hd0 = 1'b1;

if ( q_old[1] === q[1])
   hd1 = 1'b0;
else hd1 = 1'b1;


if ( q_old[2] === q[2])
   hd2 = 1'b0;
else hd2 = 1'b1;


if ( q_old[3] === q[3])
   hd3 = 1'b0;
else hd3 = 1'b1;


if ( q_old[4] === q[4])
   hd4 = 1'b0;
else hd4 = 1'b1;
```

```
if ( q_old[5] === q[5])
   hd5 = 1'b0;
else hd5 = 1'b1;


if ( q_old[6] === q[6])
   hd6 = 1'b0;
else hd6 = 1'b1;


if ( q_old[7] === q[7])
   hd7 = 1'b0;
else hd7 = 1'b1;

end

assign HD = hd0+hd1+hd2+hd3+hd4+hd5+hd6+hd7;

avgcal ac1(clk,reset,instr,HD,avg);


//------------------------------------------------------------------------------

flopr #(32) pcreg(clk, reset, pcnext, pc);

adder pcadd1(pc, 32'b100, pcplus4);

configure configure(clk,reset,instr, tap_loc);
```

```
mux2 #(32) pcmux(pcplus4, pc, pccount, pcnext);

endmodule

//-------------------------------------------------------------------------------
//---------------------------------MUX to choose pc-------------------------------------
module mux2 #(parameter WIDTH=8) (input logic [WIDTH-1:0] d0, d1,
                                  input logic s,
                                  output logic [WIDTH-1:0] y);

assign y=s ? d1 : d0;

endmodule
//-------------------------------------------------------------------------------------

//-------------------------Choose r_addr----------------------------------------
module chooseadd(input logic clk,reset,input logic [31:0] instr,input logic[7:0] pccounter,count,input logic cycle, output logic [7:0] r_addr);

always_ff @ (posedge clk)
if (reset)
                    r_addr <= 0;

        else if (instr [31:26] === 6'b000110)
        r_addr <= instr[25:18];
      else if ( instr [31:26] === 6'b000111)
        r_addr <= r_addr + instr [17:10];
      else if (instr [31:26] === 6'b001001)
        r_addr <= 8'b00000000;
```

```
        else if ( instr [31:26] === 6'b001010 && pccounter === 8'b00000000 && count === 8'b00000001)
            r_addr <= r_addr;
        else if (instr[31:26] === 6'b001010 && cycle === 1'b0)
            r_addr <= r_addr + 1;
        else
            r_addr <= r_addr;

endmodule

//-------------------------------------------------------------------------------
//-------------------------------------AVERAGE CALCULATOR--------------------------
module avgcal(input logic clk,reset,input logic [31:0] instr,input logic [7:0] HD, output logic [7:0] avg);

logic [7:0] rc,runcount,HD_TOTAL;

always_comb
if(instr[31:26] === 6'b000011)
rc = 8'b00000001;
else rc = 8'b00000000;

always_ff @(posedge clk)
if (reset)
begin
   HD_TOTAL <= 0;
   runcount <= 0;
end
else if (rc === 8'b00000001)
begin
   HD_TOTAL <= HD_TOTAL + HD;
   runcount <= runcount + rc;
```

```
end
else
begin
  HD_TOTAL <= HD_TOTAL;
  runcount <= runcount;
end

always_comb
if (instr [31:26] === 6'b001001)
avg = HD_TOTAL / runcount;

endmodule
//--------------------------------ADDER-----------------------------------------

module adder(input logic [31:0] a, b,  output logic [31:0] y);

        assign y=a+b;
endmodule
//-------------------------------------------------------------------------------
//--------------------------------Config-----------------------------------------
module configure (input logic clk, reset,input logic [31:0] instr, output logic [6:0] tap_loc);

always_ff @ (posedge clk)
if (reset)
                tap_loc <= 0;

        else
        begin
                case(instr[31:26])
```

```
                6'b000001: tap_loc <= instr[6:0];
                        default: tap_loc <= tap_loc;

                endcase
        end
endmodule
//---------------------------------------------------------------------------
//--------------------------------FLIP FLOP TO STORE---------------------------------

module flopr #(parameter WIDTH=8) (input logic clk, reset, input logic [WIDTH-1:0] d, output logic [WIDTH-1:0] q);

        always_ff @(posedge clk, posedge reset)

                        if (reset) q <= 0;

                        else q <= d;

endmodule
//---------------------------------------------------------------------------
//--------------------------------HAMMING DISTANCE MEMORY---------------------------

module sdmem(input logic clk, we, input logic [7:0] a, wd);

        logic [7:0] RAM[255:0];

        always_ff @(posedge clk)

                if (we) RAM[a] <= wd;
```

```
endmodule

//----------------------------------------------------------------------
```

SystemVerilog Testbench Code

```
module LFSRbatch_testbench();
logic clk;
logic reset;
logic [7:0] LFSR_q,r_addr;
logic memwrite;
logic [31:0] pc, instr;
logic [7:0] readdata;
logic [6:0] tap_loc;
logic [7:0] HD,AVG;

lfsrbatch dut (clk, reset,LFSR_q, r_addr,readdata,memwrite,pc,instr,tap_loc,HD,AVG);

initial
begin
reset <= 1; # 5; reset <= 0;
end


always
begin
clk <= 1; # 5; clk <= 0; # 5;
end

endmodule
```