嵌入式系统结构与设计课程实验

实验报告 5

Latch with MUX & More for Shifter

邱迪聪 / 11331262

2013/11/6

Content

Section 1: Latch with MUX

1. Target

Find out and understand the occasions when latch produced.

2. Normally design a 4-input MUX

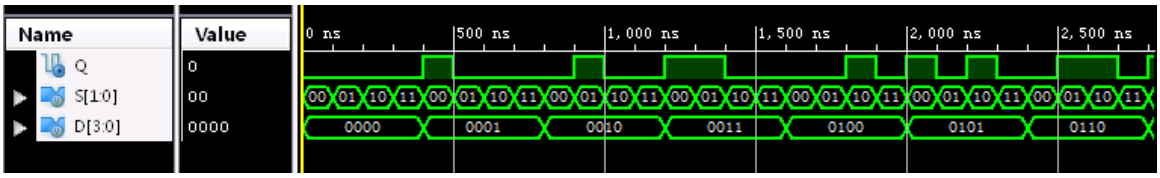
As we have done before at Lab 02, the 4-input MUX (Multi-Way Selector) can be easily designed with a `case` code segment. The primary code block is as followed.

```
case(S)
  2'b 00:  Q <= D[0];
  2'b 01:  Q <= D[1];
  2'b 10:  Q <= D[2];
  2'b 11:  Q <= D[3];
  default: Q <= D[0];
endcase
```

It is a completely designed `case` statement.

2.1 Simulation

A complete simulation testing is processed, and the partial result is as followed.



The simulation result shows that the 4-input multi-way selector functioned well, as expected. Among different cases, the output `Q` is always the bit within `D[3:0]`, which is selected by `S[1:0]`.

2.2 Synthesization, Implementation and Program Generation

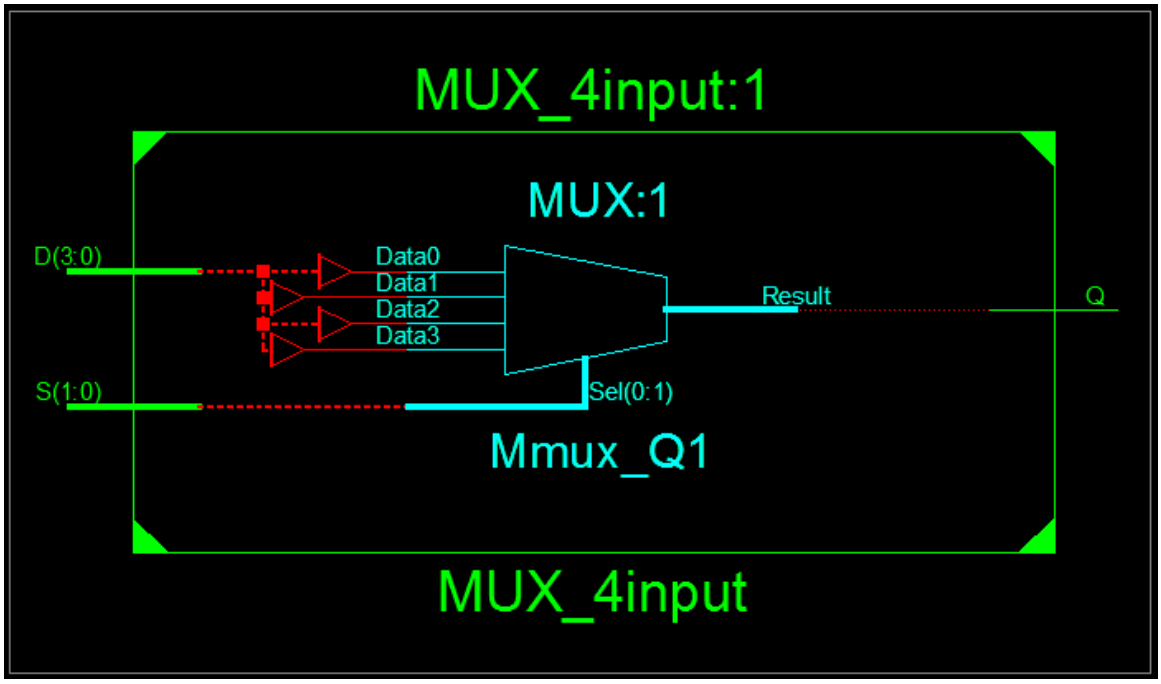
The implementation constraints file is as followed:

```
// Input: D[3:0]
NET "D[3]" LOC = "T5"; // left most button
NET "D[2]" LOC = "V8";
NET "D[1]" LOC = "U8";
NET "D[0]" LOC = "N8";

// Input: S[1:0]
NET "S[1]" LOC = "M8";
NET "S[0]" LOC = "V9";

// Output: Q
NET "Q" LOC = "T11"; // left most LED
```

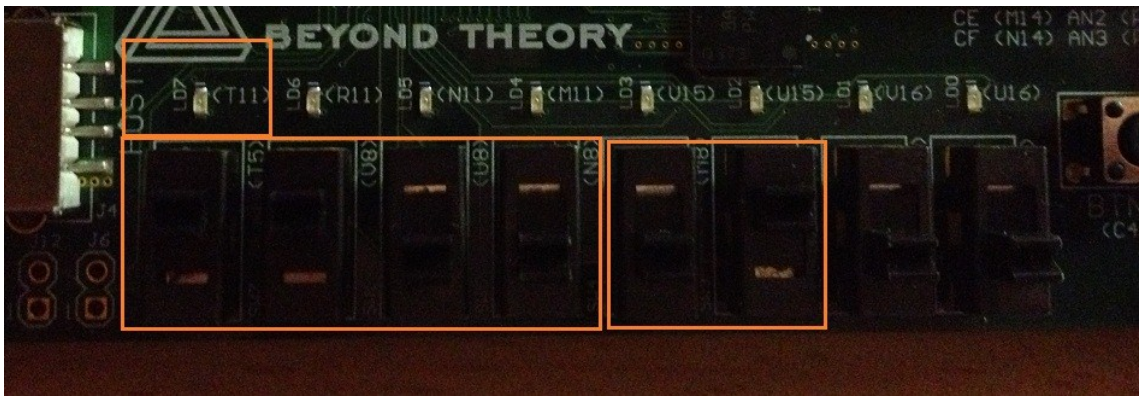
The RTL diagram generated from Synthesization is as below:



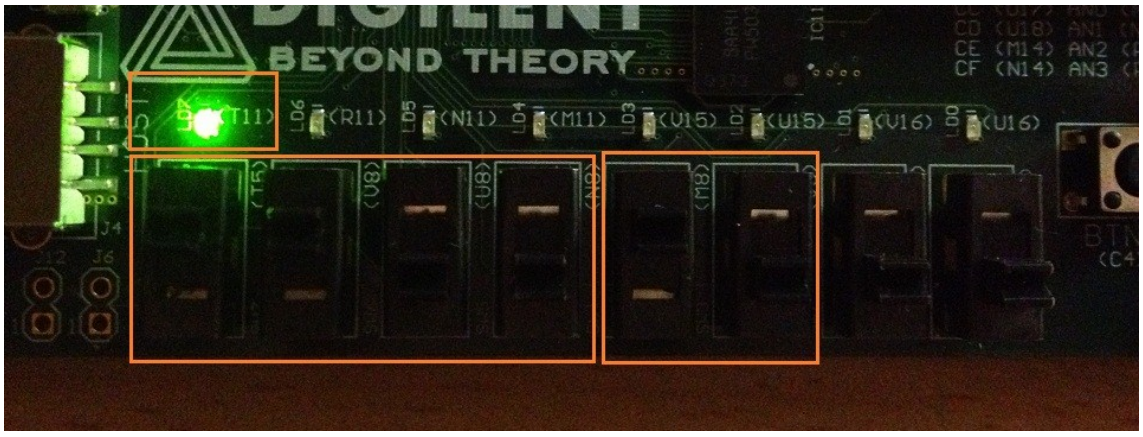
The following steps were successfully gone through and a .bit file was generated successfully.

2.3 Online Testing

D = 1100, S = 01; Q = 0:



D = 1100, S = 10; Q = 1:



D = 1010, S = 01; Q = 1:

The test result matched expectation.

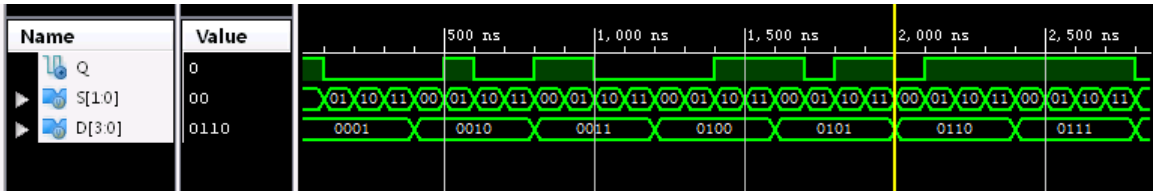
3. Abnormally design a 4-input MUX

To test what will happen if the `case` statement is incomplete, this experiment is to deliberately miss a critical case within the `case` statement.

```
case(S)
  2'b 00:  Q <= D[0];
  2'b 01:  Q <= D[1];
  2'b 10:  Q <= D[2];
  //2'b 11:  Q <= D[3];
  //default: Q <= D[0];
endcase
```

3.1 Simulation

A complete simulation testing is processed, and the partial result is as followed.



It is shown from the simulation result that every undeclared case derives the result from the previous output, which means if the case in the case statement is missed, the output for this case will keep the previous output state.

3.2 Synthesization, Implementation and Program Generation

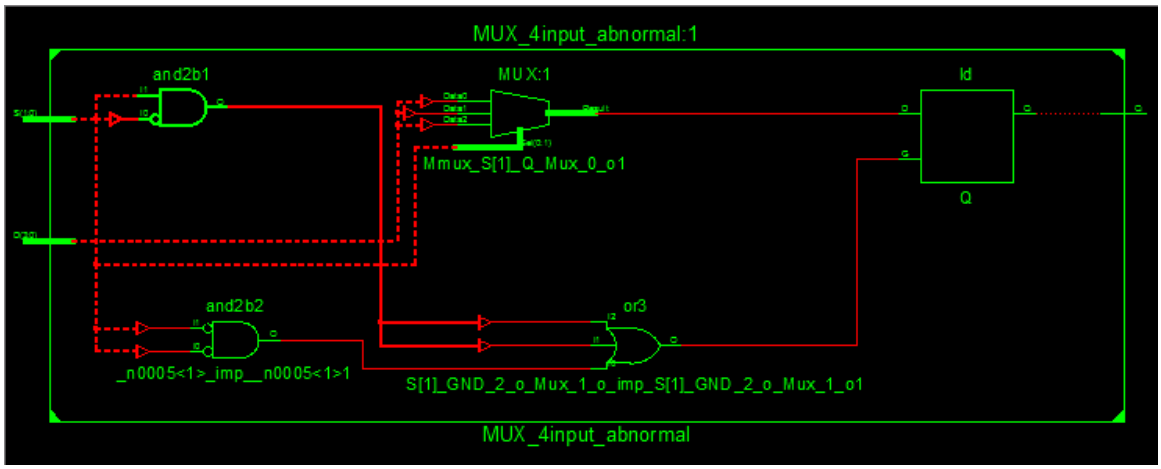
The implementation constraints file is as followed:

```
// Input: D[3:0]
NET "D[3]" LOC = "T5"; // left most button
NET "D[2]" LOC = "V8";
NET "D[1]" LOC = "U8";
NET "D[0]" LOC = "N8";

// Input: S[1:0]
NET "S[1]" LOC = "M8";
NET "S[0]" LOC = "V9";
```

```
// Output: Q
NET "Q" LOC = "T11"; // left most LED
```

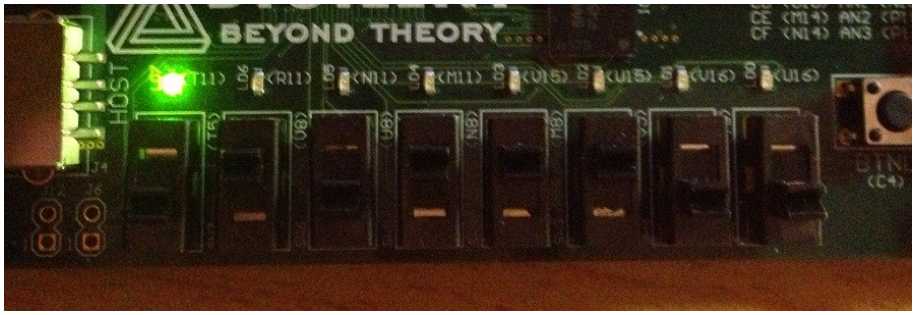
The RTL diagram generated from Synthesization is as below:



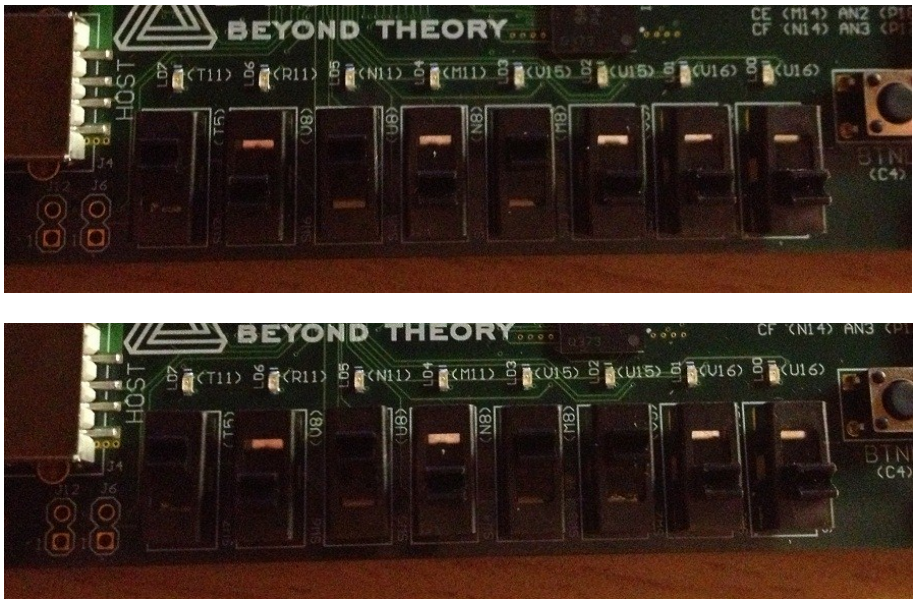
The RTL diagram also implies the same idea. Only the three declared cases within the case statement are connected to the MUX, and the missed one generates a activation signal to the Q flip-flop, which keeps the output if S = 11.

3.3 Online Testing

<u>D = 0101, S = 10 → D = 0101, S = 11:</u>



D = 1010, S = 10 → D = 1010, S = 11:



The test result matched expectation.

4. Compare

The synthesized circuits are different between the completely-designed MUX and incompletely-designed MUX. A latch will be automatically appended to the latter one in Synthesization will be automatically, in order to keep the output from the previous one if the current case is lack of declaration.

Section 2: More for Shifter

1. Target

- Understand shifter more deeply.
- Getting started to design your own Verilog, and evaluate in your board.

2. Serial-Input-Parallel-Output Module

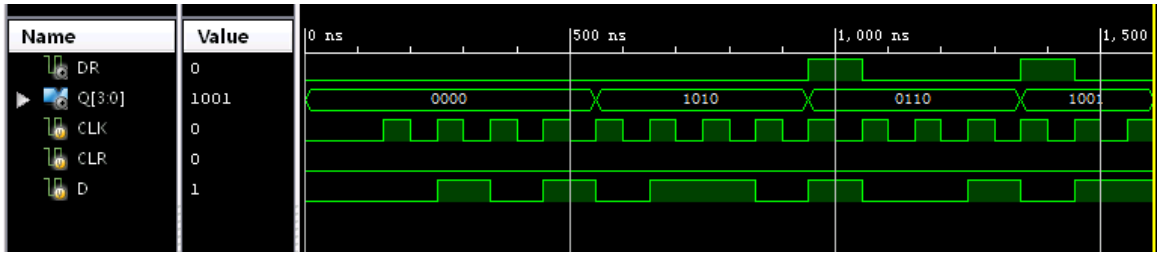
Using the 4-bit shifter module from our last week experiment is one of the available ways. But considering

the optimization, I decided to implement the codes all again, and so as to have a better optimization of the codes.

In the new developed code, 4 registers are used to keep the shifting data and the combination of these 4 registers can be considered as an inner 4-bit shifter. Another 4 registers are used to keep the output data, since the parallel output requires that the output changes only if 4 bits of the inner D flip-flops are filled all again. What's more, several registers are taken into consideration for the Data Ready signal and the update of the output.

2.1 Simulation

16 inputs are applied to test the functions of this converter. And the simulation result is as below:



The initial output is 0, and after one cycle, the output Q is set to be the 4 previous inputs; after the next loop, a DR (Data Ready) pulse comes to generate a signal to take the previous output.

2.2 Synthesization, Implementation and Program Generation

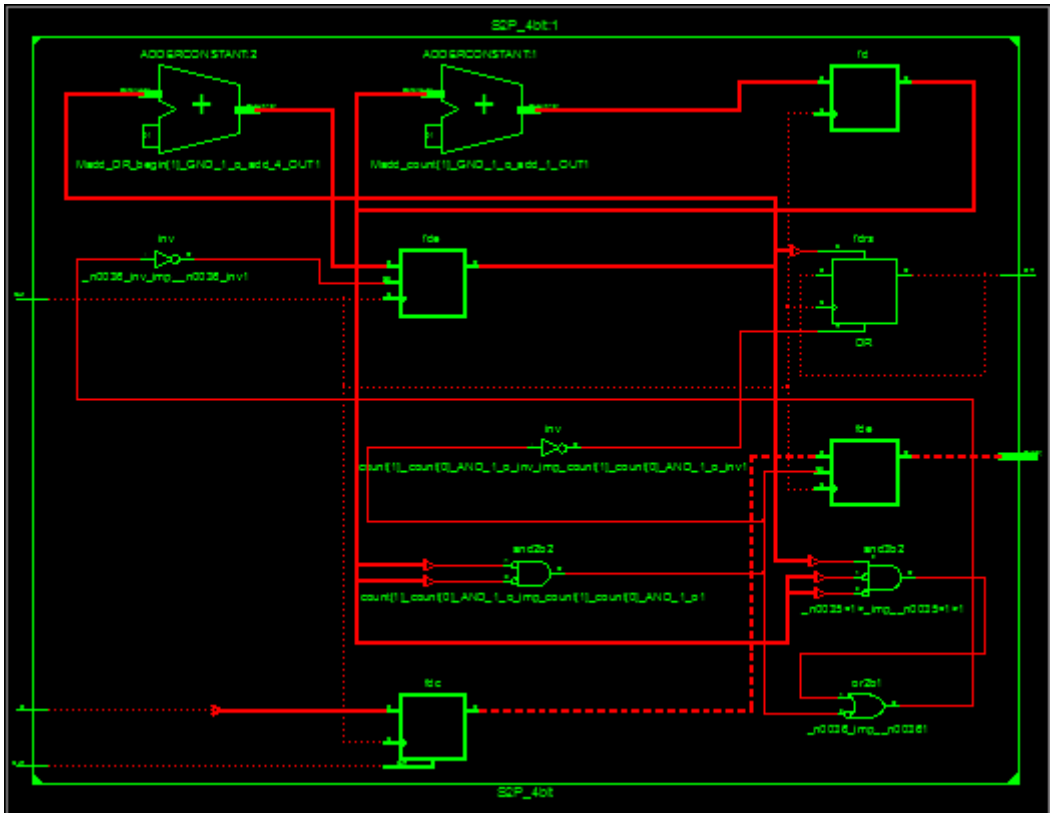
The implementation constraints file is as followed:

```
// Input: CLK, CLR, D
NET "CLK" CLOCK_DEDICATED_ROUTE = FALSE;
NET "CLK" LOC = "B8"; // central button
NET "CLR" LOC = "C9"; // down (BTND)
NET "D" LOC = "T5"; // left most button

// Output: Q[3:0]
NET "Q[3]" LOC = "T11"; // left most LED
NET "Q[2]" LOC = "R11";
NET "Q[1]" LOC = "N11";
NET "Q[0]" LOC = "M11";

// Output: DR
NET "DR" LOC = "U16"; // right most LED
```

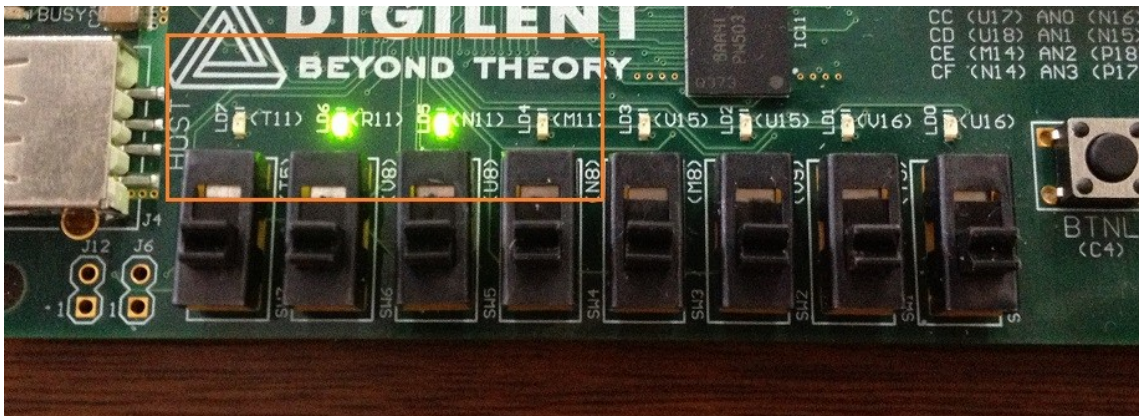
The RTL diagram generated from Synthesization is as below:



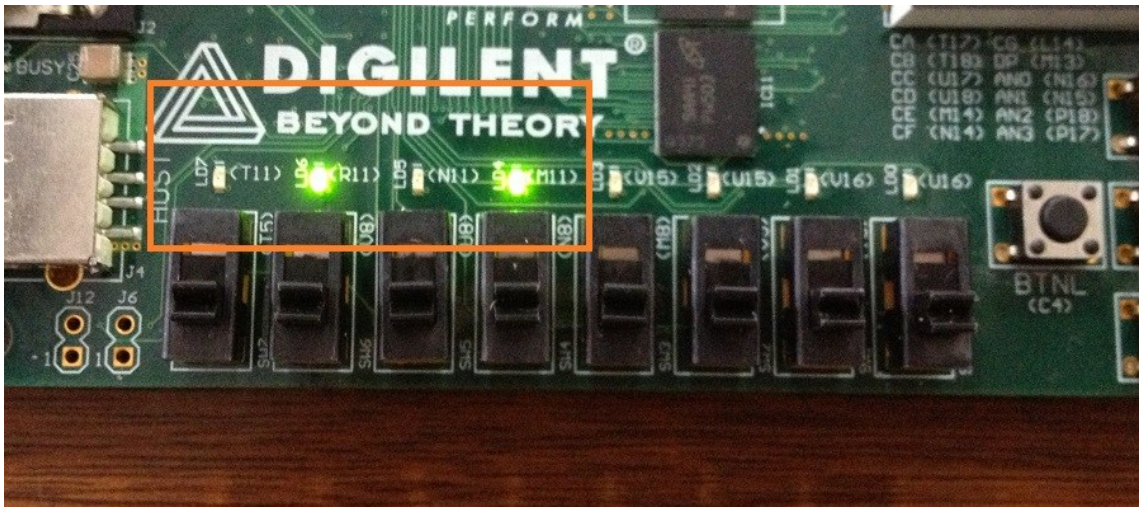
The following steps were successfully gone through and a .bit file was generated successfully.

2.3 Online Testing

D: 0 → 1 → 1 → 0



D: 1 → 0 → 1 → 0



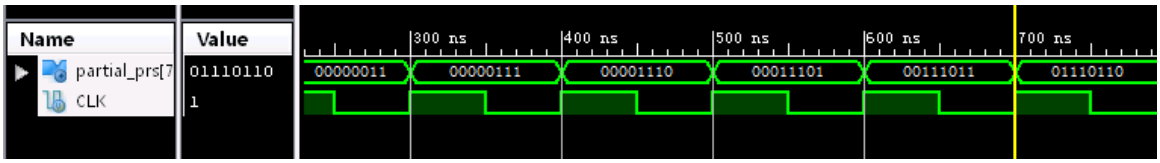
3. Pseudo-Random Sequence Generator

To generate a pseudo-random sequence, this design uses a LFSR method. The primary generator is a 4-bit sequence, and the 4th bit and the 3rd bit connect to a NXOR gate which feedback to the first input.

And the other 4 registers are needed to display the history records.

3.1 Simulation

Below is the simulation result of the pseudo-random sequence generator.



Each partial pseudo-random sequence output shifts one bit from the previous one, and it can be seen from the result that each partial sequence appears to be different in a specific rounds.

3.2 Synthesization, Implementation and Program Generation

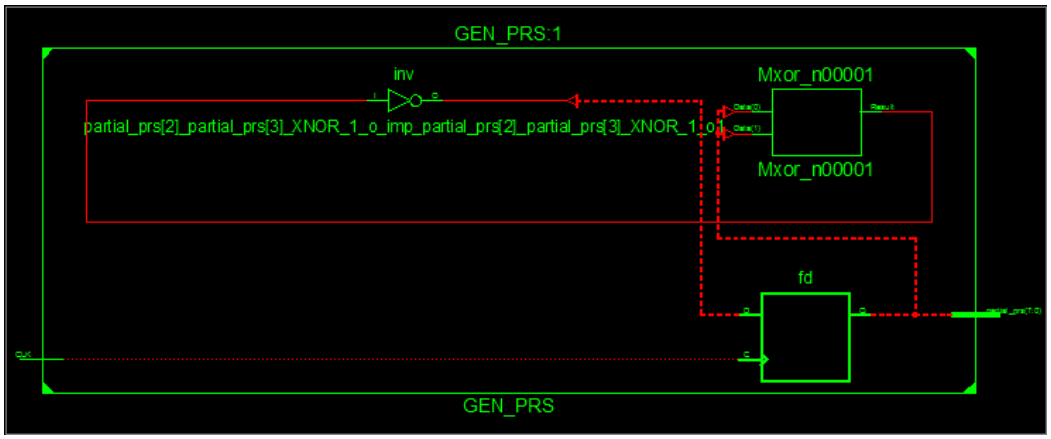
The implementation constraints file is as followed:

```
// Input: CLK
```

```
NET "CLK" CLOCK_DEDICATED_ROUTE = FALSE;
NET "CLK" LOC = "B8"; // central button

// Output: partial_prs[7:0]
NET "partial_prs[7]" LOC = "T11"; // left most LED
NET "partial_prs[6]" LOC = "R11";
NET "partial_prs[5]" LOC = "N11";
NET "partial_prs[4]" LOC = "M11";
NET "partial_prs[3]" LOC = "V15";
NET "partial_prs[2]" LOC = "U15";
NET "partial_prs[1]" LOC = "V16";
NET "partial_prs[0]" LOC = "U16"; // right most LED
```

The RTL diagram generated from Synthesization is as below:



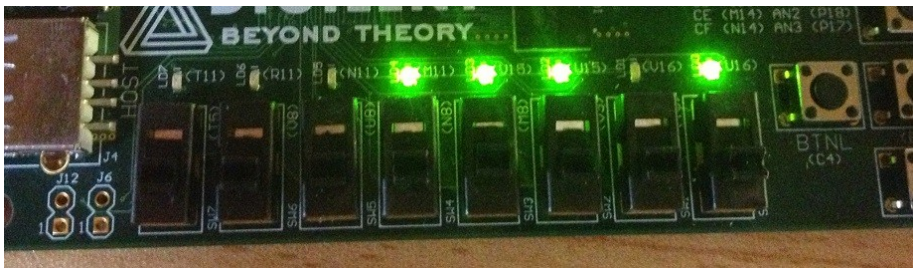
3.3 Maximal length of the PRS

The number of different combinations for a 4-bit pseudo-random generator, the total different cases are $2^4 = 16$, hence the maximal length in bits of the designed sequence is $16 \times 4 = 64$.

3.4 Online Testing

Below is a partial sequence shown with the LEDs, and the direction is right to left (←).


BEYOND THEORY
BTNL
(C4)


BEYOND THEORY
BTNL
(C4)


BEYOND THEORY
BTNL
(C4)


DIGILENT
BEYOND THEORY
BTNL
(C4)

Afterthought

I'm quite confused this time why our laboratory instructions turn to be in English, so I just have to change my report to English. I am kind of sorry for that.

This experiment is actually a combination of 4 small experiments, with each 2 experiments trying to deepen the applications of one former experiment. We are required to extend the applications of multi-way selector and shifter.

The one interesting thing I have thought before when I was doing the experiment of multi-way selector is that I hope to know what might happen if the case statement was incomplete. And this experiment is just an answer to my question. The design of latch for lack of case statement is a genius idea, and what's greater, they have implemented this idea. If we simply miss a key case for the case statement, the circuit design will be implemented automatically with latch to complete the missed case, and of course, a warning will be shown. So our design with bugs will not just crash down immediately, since the ISE software will try its best to make up our careless mistake.

Another thing made me delight is that I notice a parallel output should be provided with a clock signal. In the experiment 2.1, we are required to design a serial-to-parallel convertor, and the design will not be difficult if it is actually that easy as required. Actually, it is the experiment that cost me the most of time to design. I notice the order of signal output is critical for the converter. If the data-ready signal is sent before the initial step, the system will generate wrong information for the outer system, so an initial counter is especially required to skip the initial round. And after the initial round, looping rounds are quite challenging. My first idea is to delay a clock pulse after the serial signals are all inputted, but the design becomes so inefficient. Instead, my design is to delay a serial-to-parallel round before the data-ready signal for the previous round is outputted.

The pseudo-random sequence generator is quite amazing since I've never thought of such kind of thing, but for the implementation, it is easy if we follow the principles of LFSR. Hence, there is not too much to talk about.

I'm glad to do this experiment, and the deeper use means a lot of benefit to me.

David Qiu (邱迪聪)

2013.11.6

Appendix 1: Attachment List

1.1 MUX_4in

\ MUX_4in.v

\ MUX_4in_ctr.ucf

\ MUX_4in_test.v

1.2 MUX_4in_abnormal

\ MUX_4in_abnormal

\ MUX_4in_abnormal_ctr.ucf

\ MUX_4in_abnormal_test.v

2.1 S2P_4bit

\ S2P_4bit

\ S2P_4bit_ctr.ucf

\ S2P_4bit_test.v

2.2 GEN_PRS

\ GEN_PRS

\ GEN_PRS_ctr.ucf

\ GEN_PRS_test.v

X.v files are primary code files; X_ctr.ucf files are implementation constraints files; X_test.v files are test files.