Lab06 Report

Park Chaehyun

1. Verilog implementation

(1) R-S latch

(i) Source code:

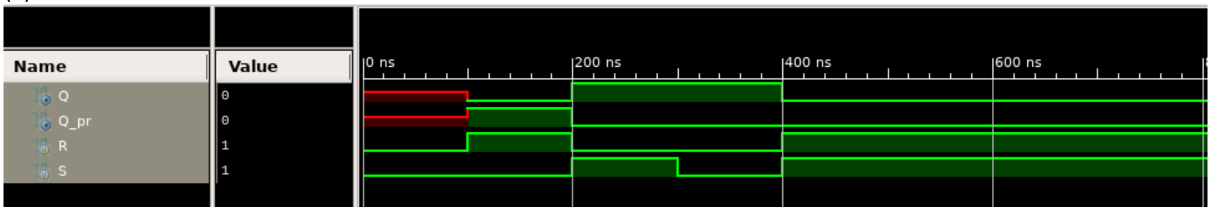
```
module RS_Latch(
	input R,
	input S,
	output Q,
	output Q_pr
	);

	nor u1(Q, R, Q_pr);
	nor u2(Q_pr, Q, S);

endmodule
```

The two inputs are R (reset) and S (set). The outputs are Q and Q_pr, where Q is the intended output and Q_pr is the inverse of Q.

(ii) Simulation result:



We can check that when R=0 and S=0 at the beginning, the output is undefined because there is no previous output to hold. And we check that R=1 and S=0 yields Q=0, R=0 and S=1 yields Q=1, and R=0 and S=0 holds the previous ouput, with Q_pr being the inverse of Q at both times. So this source code produces the intended output for 'Reset', 'Set' and 'Hold' function of the R-S latch.

(2) Gated R-S latch

(i) Source code:

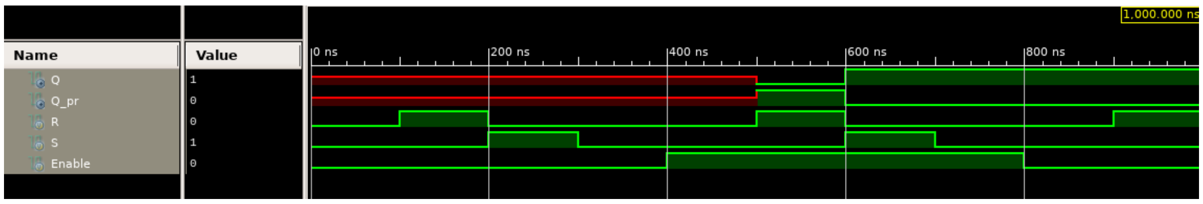
```
module Gated_RS_Latch(
	input R,
	input S,
	input Enable,
	output Q,
	output Q_pr
	);

	wire w1, w2;
	and a1(w1, R, Enable);
	and a2(w2, S, Enable);
	RS_Latch R1(w1, w2, Q, Q_pr);

endmodule
```

The three inputs are R (reset), S (set) and Enable. The outputs are Q and Q_pr, where Q is the intended output and Q_pr is the inverse of Q.

(ii) Simulation result:



We can check that when Enable=0 at the beginning, the output is undefined because there is no previous output to hold. When Enable=1, we can check that it produces correct output for each 'Reset', 'Set' and 'Hold' function, same as (1).

(3) Simple oscillator

(i) Source code:

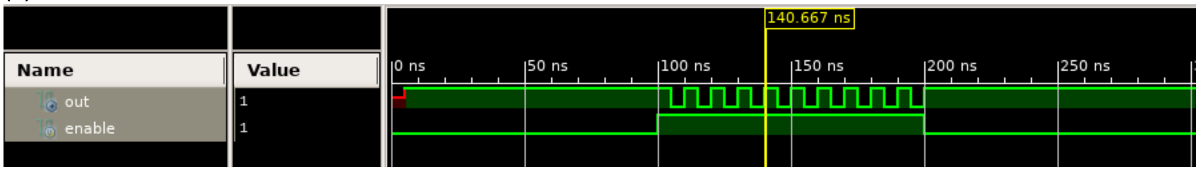
```
module Simple_Oscillator(
        input enable,
        output out
        );

        wire tmp;
        and t1(tmp, enable, out);
        not #5 t2(out, tmp);

endmodule
```

The input is 'enable' and the output is 'out'. and t1(tmp, enable, out) stores the AND of 'enable' and 'out' in 'tmp'. Then not #5 t2(out, tmp) stores the inverse of tmp in out, but with a 5ns delay.

(ii) Simulation result:



When enable=0, tmp is 0 and out is 1. We can check that there is 5ns delay from tmp to out because out is undefined from t=0ns to t=5ns. Now when enable is changed from 0 to 1 at 100ns, tmp is 1 and out becomes 0 at 105ns. Then tmp becomes 0 again and makes out 0 at 110ns. This repeats as long as enable is 1. So 'out' will oscillate with a 10ns period, so it can be used as a clock with 10ns period.

(4) R-S Flip Flop (rising edge triggered)

(i) Source code:

```
module RS_Flip_Flop(
        input enable,
        input R,
        input S,
        output Q,
        output Q_pr,
        output clk
        );

        reg R_reg;
        reg S_reg;

        Simple_Oscillator S1(enable, clk);
```

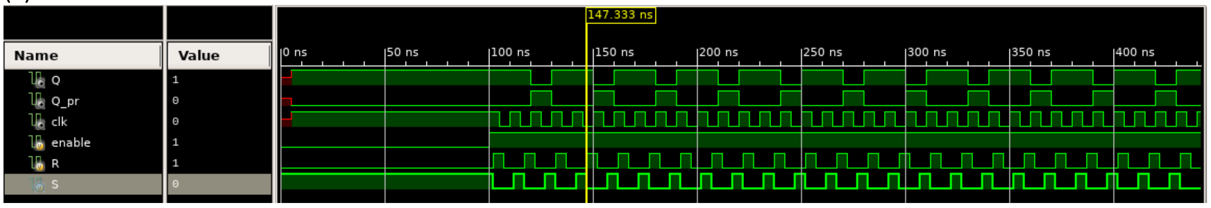
```
	Gated_RS_Latch R1(R_reg, S_reg, clk, Q, Q_pr);

	always@(posedge clk)
		begin
			R_reg = R;
			S_reg = S;
		end

endmodule
```

The three inputs are enable, R and S. The outputs are Q, Q_pr and clk where Q is the intended output, Q_pr is the inverse of Q and clk is the clock. An instance of (3) is created with the input enable and output clk. An instance of (2) is created with the inputs R_reg, S_reg and clk, and outputs Q and Q_pr. An always statement is used with the posedge of clk in the sensitivity list, and it assigns R and S to R_reg and S_reg respectively. So the inputs to the gated RS latch should only change at the rising edge of the clock.

(ii) Simulation result:



We can first check that changes to Q and Q_pr only occurs at the rising edge of the clock (t is a multiple of 10 starting from 110ns). At t=110ns, R=0 and S=0 so it holds the previous output. At t=120ns, R=1 and S=0 so Q becomes 0. At t=130ns, R=0 and S=1 so Q becomes 1. And changes to R and S do not affect the output outside the rising edge of the clock. So this is the correct behavior for the RS flip-flop.

(3) Discussion
In this week's lab session, we not only wrote code but programmed the written code on the FPGA board. This enabled us to give input via switches and check the output on the LEDs, instead of checking only the simulation result. Working with actual hardware enabled us to understand the purpose of writing and using hardware description languages such as Verilog more clearly.

2. Differences between flip-flops
(1) RS flip-flop
It has a set input (S) and a reset input (R). If S=0 and R=0, it holds the previous state. When S=0 and R=1, the next state is 0, performing 'reset'. When S=1 and R=0, the next state is 1. Performing 'set'. The output when S=1 and R=1 is not defined. So the truth table is as follows:

S	R	Q_{n+1}
0	0	Q_n
0	1	0
1	0	1
1	1	X

(2) JK flip-flop
JK flip-flop produces the same output as the RS flip-flop for the first 3 input combinations, but gives a defined output for the input S=1 and R=1, which is the inverse of the previous state. So the truth table is as follows:

S	R	Q_{n+1}
0	0	Q_n
0	1	0
1	0	1

1	1	$Q_n{}'$

(3) D flip-flop
D flip-flop produces the same output as the input. So the truth table is as follows:

D	Q_{n+1}
0	0
1	1

(4) T flip-flop
T flip-flop holds the previous state if T=0, and produces the inverse of the previous state if T=1. So it can perform 'toggle' function. The truth table is as follows:

T	Q_{n+1}
0	Q_n
1	$Q_n{}'$

Comparison: RS and JK flip-flops have two inputs, while D and T flip-flops have a single input. RS flip-flop can result in undefined state, unlike other flip-flops. JK flip-flop is an improvement on the RS flip-flop, overcoming the problem of the undefined state. For D flip-flop, the next state only depends on the input, while the other flip-flops' next states depend both on the input and the current state.

3. Discussion
We were able to design a sequential circuit, where the output not only depends on the input but also on the previous state. In this session, it was achieved by feeding the output back to the input. Also, we were able to make edge-triggered circuit, where the circuit's output only changes at the rising edge of the clock. This was achieved by using an always statement with the 'posedge clock' in the sensitivity list, and by having the inputs assigned inside the always statement. And we were able to make a clock signal by implementing gate delay, using the delay operator '#'.