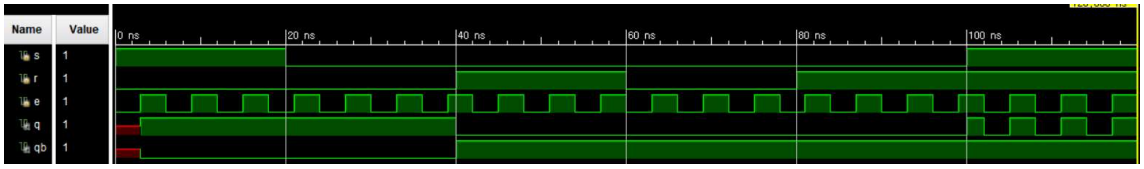
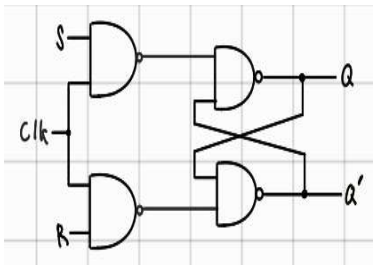
11주차 결과보고서

전공: 경영학과 학년: 4학년 학번: 20190808 이름: 방지혁

1.

RS Flip-Flop NAND





e	s	r	q
0	x	x	불변
1	0	0	불변
1	0	1	0
1	1	0	1
1	1	1	X

```
`timescale 1ns / 1ps

module srffnand(
  input e,
  input r,
  input s,
  output q,
  output qb
);

assign q  = ~((~(s&e))&qb);
assign qb = ~((~(r&e))&q);

endmodule
```

```
`timescale 1ns / 1ps

module srffnand_tb;
reg s, r, e;
wire q, qb;

srffnand test(
.s(s),
.r(r),
.e(e),
.q(q),
.qb(qb)
);

initial begin
s = 1'b0;
r = 1'b0;
e = 1'b0;
r = 1'b0; s = 1'b1; #20;
r = 1'b0; s = 1'b0; #20;
r = 1'b1; s = 1'b0; #20;
r = 1'b0; s = 1'b0; #20;
r = 1'b1; s = 1'b0; #20;
r = 1'b1; s = 1'b1; #20;
$finish;
end

always@(e) begin
e <= #3 ~e;
end

endmodule
```

e	Present State Q(t)	Q(t+1) Next State INPUT RS순			
		00	01	10	11
0	0	Q(t)			
	1				
1	0	Q(t)	1	0	X
	1				X

순서	R	S	Q	Q'
1	0	1	1	0
2	0	0	1	0
3	1	0	0	1
4	0	0	0	1
5	1	0	0	1
6	1	1	X	X

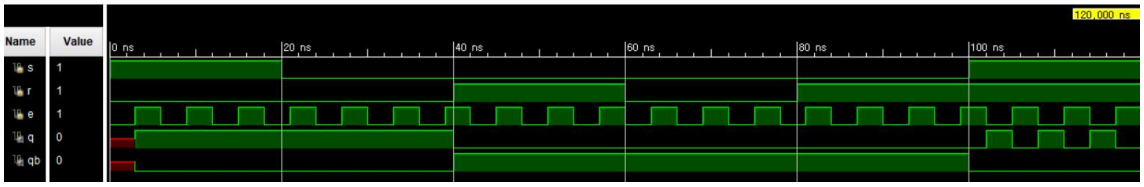
정확히 말하면 rs flip-flop이 아니라 nand게이트를 기반으로 만든 rs-latch입니다. 해당 회로는 clock level이 1일 경우에만 동작하도록 구현했습니다. 여기서 qb는 q의 반댓값입니다. nand 2개로 구현하였기에 첫 번째 좌측 상단 nand게이트의 output은 ~(s&e)이고, 이 값이 우측 상단의 nand게이트로 다시 들어가 output인 q = ~((~(s&e))&qb)입니다.

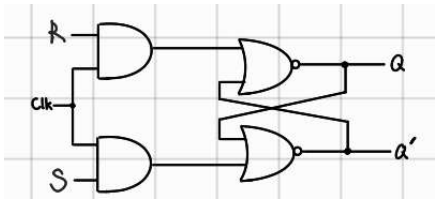
또한, 첫 번째 좌측 하단 nand게이트의 output은 ~(r&e)이고, 이 값이 우측 하단의 nand게이트로 다시 들어가 output인 qb = ~((~(r&e))&q)입니다.

정리하자면, e가 0이거나 e의 신호가 1인 상태에서 s와 r값으로 둘 다 0이 들어오면 값은 q

는 이전 상태를 유지합니다. s의 값만 1인 경우 q의 값은 1이 되고, r의 값만 1인 경우 q의 값은 0이 됩니다. 둘 다 1인 경우는 원래 허용되지 않지만, 이에 대한 예외 처리는 배우지 않았기에 코드 상에서 따로 설정하지 않았고, 시뮬레이션 결과에도 그렇게 나타나지 않습니다. 이전 q, q' 상태에 대한 정의가 없기 때문에 시뮬레이션 결과상 빨갛게 X로 나타납니다.

RS Flip-Flop NOR






```
`timescale 1ns / 1ps

module srffnor(
  input e,
  input r,
  input s,
  output q,
  output qb
);

assign q  = ~((r&e)|qb);
assign qb = ~((s&e)|q);

endmodule
```

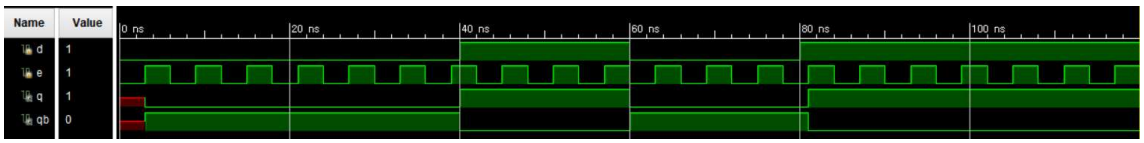
```
`timescale 1ns / 1ps

module srffnor_tb;
reg s, r, e;
wire q, qb;

srffnor test(
.s(s),
.r(r),
.e(e),
.q(q),
.qb(qb)
);

initial begin
s = 1'b0;
r = 1'b0;
e = 1'b0;
r = 1'b0; s = 1'b1; #20;
r = 1'b0; s = 1'b0; #20;
r = 1'b1; s = 1'b0; #20;
r = 1'b0; s = 1'b0; #20;
r = 1'b1; s = 1'b0; #20;
r = 1'b1; s = 1'b1; #20;
$finish;
end

always@(e) begin
e <= #3 ~e;
end

endmodule
```

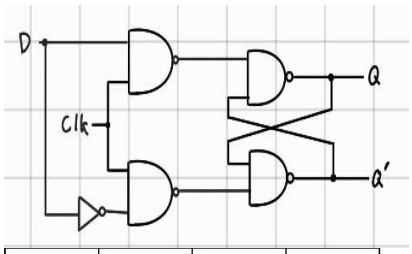
순서	R	S	Q	Q'
1	0	1	1	0
2	0	0	1	0
3	1	0	0	1
4	0	0	0	1
5	1	0	0	1
6	1	1	X	X

마찬가지로 rs flip-flop이 아니라 nand게이트를 기반으로 만든 level trigger인 rs-latch입니다. 해당 회로는 clock level이 1일 경우에만 동작하도록 구현했습니다. 여기서 qb는 q의 반댓값입니다. and 게이트 2개, nor 게이트 2개로 구현했습니다.
첫 번째 좌측 상단 and게이트의 output은 r&e이고, 이 값이 우측 상단의 nor게이트로 다시 들어가 output인 q = ~((r&e)|qb)입니다.
또한, 첫 번째 좌측 하단 and게이트의 output은 s&e이고, 이 값이 우측 하단의 nor게이트로 다시 들어가 output인 qb = ~((s&e)|q)입니다.
정리하자면, e가 0이거나 e의 신호가 1인 상태에서 s와 r값으로 둘 다 0이 들어오면 값은 q는 이전 상태를 유지합니다. s의 값만 1인 경우 q의 값은 1이 되고, r의 값만 1인 경우 q의 값은 0이 됩니다. 둘 다 1인 경우는 원래 허용되지 않지만, 이에 대한 예외 처리는 배우지 않았기에 코드 상에서 따로 설정하지 않았고, 시뮬레이션 결과에도 그렇게 나타나지 않습니다. 마찬가지로 이전 q, q' 상태에 대한 정의가 없기 때문에 시뮬레이션 결과상 빨갛게 X로 나타납니다.

다음 장에 이어서

2.





순서	D	Q	Q'
1	0	0	1
2	0	0	1
3	1	1	0
4	0	0	1
5	1	1	0
6	1	1	0

```
`timescale 1ns / 1ps

module dff(
  input e,
  input d,
  output q,
  output qb
);

assign q  = ~((~(d&e))&qb);
assign qb = ~((~((~d)&e))&q);

endmodule
```

e	d	q
0	x	불변
1	0	0
1	1	1

```
`timescale 1ns / 1ps

module d_ff_tb;
reg d, e;
wire q, qb;

dff test(
.e(e),
.d(d),

.q(q),
.qb(qb)
);
initial begin
e = 1'b0;
d = 1'b0; #20
d = 1'b0; #20
d = 1'b1; #20
d = 1'b0; #20
d = 1'b1; #20
d = 1'b1; #20
$finish;
end

always@(e) begin
e <= #3 ~e;
end
endmodule
```

e	Present State Q(t)	Q(t+1) Next State	
		INPUT D	
		0	1
0	0	Q(t)	
	1		
1	0	0	1
	1		

마찬가지로 d flip-flop이 아니라 nand게이트를 기반으로 만든 level trigger인 d-latch입니다. 해당 회로는 clock level이 1일 경우에만 동작하도록 구현했습니다.
여기서 qb는 q의 반댓값입니다. nand 게이트 4개로 구현했습니다.
첫 번째 좌측 상단 nand게이트의 output은 ~(d&e)이고, 이 값이 우측 상단의 nand게이트로 다시 들어가 output인 q = ~((~(d&e))&qb)입니다. 또한, 첫 번째 좌측 하단 nand게이트의 output은 ~((~d)&e)이고, 이 값이 우측 하단의 nand게이트로 다시 들어가 output인 qb = ~((~((~d)&e))&q)입니다.
정리하자면, e가 0인 상태에서는 d 값과 상관없이 q는 이전 상태를 유지합니다. d의 값이 1인 경우 q의 값은 1이 되고, d의 값이 0인 경우 q의 값은 0이 됩니다. 마찬가지로 이전 q, q' 상태에 대한 정의가 없기 때문에 시뮬레이션 결과상 빨갛게 X로 나타납니다.

3.

RS와 D Latch의 개념을 습득할 뿐만 아니라 이를 NAND와 NOR 게이트 2가지 방식으로 구현할 수 있다는 것을 확인했습니다. 다만, 본 실험의 목적이였던 FF을 구현하지는 못하여 추후 edge trigger 방식이 되도록 이에 대한 코드 구현 방식을 습득할 필요가 있습니다. 시뮬레이션을 통해 결과를 확인한 후 FPGA 보드에 올려 직접 실행해보았습니다. 그러나 이 과정에서 실험자의 컴퓨터가 문제인지 bitstream을 만드는 과정에서 난항을 겪기도 했습니다. 무엇

보다도 이번 verilog design source에서 output이 다른 ouput으로 서로 들어가기 때문에 상호 의존성이 발생하고, 순환 연결을 허용하기 위해 XDC 파일에 새로운 코드를 추가했는데, 이 코드가 정확히 어떤 역할을 하는지 학습할 필요가 있어보입니다.

4.

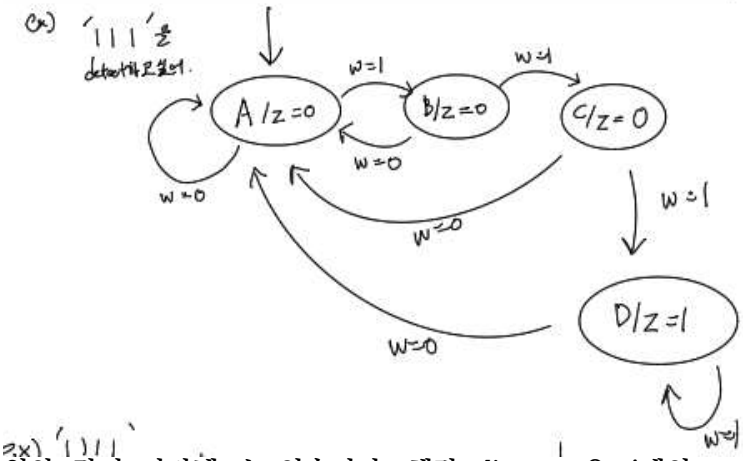
set_property ALLOW_COMBINATORIAL_LOOPS TRUE [get_nets q_OBUF]

우선 코드의 앞부분인 set_property는 특정 target에 대한 설정을 하겠다는 뜻입니다. 우리가 이번 실험에서 했던 코드의 순환 연결은 일반적으로 허용되지 않지만 이를 허용하기 위해 TRUE로 설정해주는 것입니다. 여기서 q_OBUF는 출력버퍼에 연결된 net을 말합니다. 이런 예외 설정을 통해 순환연결을 허용할 수 있게 됩니다.

State Diagram

상태를 원으로 표현하고 한 상태 간 변화를 화살표로 나타내는 다이어그램입니다. 화살표 옆에는 해당 전이가 발생하기 위한 조건을 적습니다.

ex) 우리가 예를 들어 111을 detect하고 싶다고 가정합니다.



위와 같이 나타낼 수 있습니다. 해당 diagram은 4개의 state를 필요로 하기 때문에 2개의 bit가 요구되며 2개의 flip flop이 요구됩니다.