13주차 결과보고서

전공: 신문방송학과 학년: 4학년 학번: 20191150 이름: 전현길

1. 4-bit shift register의 결과 및 simulation 과정에 대해서 설명하시오.

Number Clock Transitions	outputs				
	x	L1	L2	L3	L4
1	0	0	0	0	0
2	1	1	0	0	0
3	1	1	1	0	0
4	1	1	1	1	0
5	0	0	1	1	1
6	0	0	0	1	1
7	1	1	0	0	1

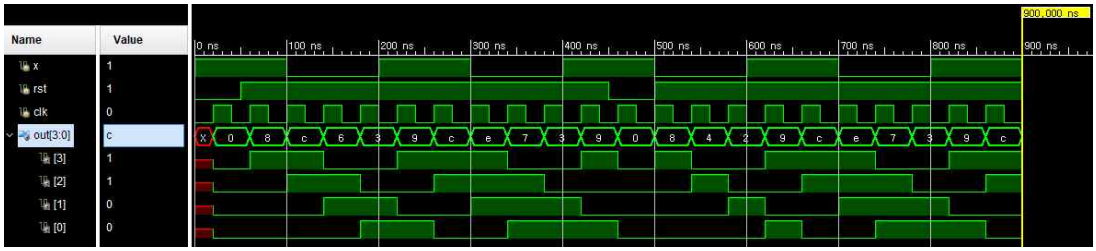
source code

```
`timescale 1ns / 1ps
module shift_register(
    input x, rst, clk,
    output reg [3:0] out
);
    always@ (posedge clk) begin
        if (!rst) out <= 4'b0000;
        else if (x) begin
            out[3] <= 1'b1;
            out[2] <= out[3];
            out[1] <= out[2];
            out[0] <= out[1];
        end
        else begin
            out[3] <= 1'b0;
            out[2] <= out[3];
            out[1] <= out[2];
            out[0] <= out[1];
        end
    end
endmodule
```

testbench code

```
`timescale 1ns / 1ps

module shift_register_tb;

reg x, rst, clk;
wire [3:0] out;

shift_register u_test(
    .x (x), .rst (rst), .clk (clk),
    .out (out)
);

initial begin
    x = 1'b1; rst = 1'b0; clk = 1'b0;
    forever #20 clk = ~clk;
end

always@ (x) begin
    #100 x <= ~x;
end

initial begin
    #50 rst = 1'b1;
    #300
    #100 rst = 1'b0; x = 1'b1;
    #50 rst = 1'b1;
    #300
    #100
    $finish;
end

endmodule
```

4-bit shift register simultation



shift register는 데이터를 저장하고, 이동시킬 때 사용되는 회로이다. 일반적으로 **D형 data latch**로 구성되며, 구현 방식에 따라 **SISO, SIPO, PISO, PIPO**로 나눌 수 있다.

이번에 구현할 4-bit counter의 경우 직렬로 값을 입력받아 직렬로 값을 출력하는 SISO shift register이다.

shift register의 값을 담기 위해 reg 변수 out[3:0]을 선언하였고 always@(posedge clk) 문을 이용해 클럭 신호의 rising edge에서 값이 출력되도록 구현했다. 이 때 LSB는 입력값 x를 저장하고, 이외의 bit는 이전 플립플롭의 출력을 입력으로 받는다. 또, active low로 구현된 rst의 값이 0이 될 때마다 값이 리셋된다.

testbench code에서 shift register의 동작을 시험하기 위해 입력값 x 신호의 주기를 200ns 단위로 구현했고, 클럭 신호가 rising edge일 때 out[3]의 값이 x의 값을 반영하는 것을 볼 수 있다. 단, x가 처음으로 0이 될 때, x가 clock 신호의 hold time delay를 충족하지 못하고 즉시 0이 되고 있는데도 1로 반영되고 있는데, 이 시뮬레이션이 현실에서 동일한 결과값을 갖도록 하려면 x의 값이 클록 신호의 rising edge 이후로 좀 더 유지될 수 있도록 할 필요가 있다.

단, shift register의 output table과 동일한 구조로 동작하고 있으므로 구현에 문제는 없는 것을 확인할 수 있다.

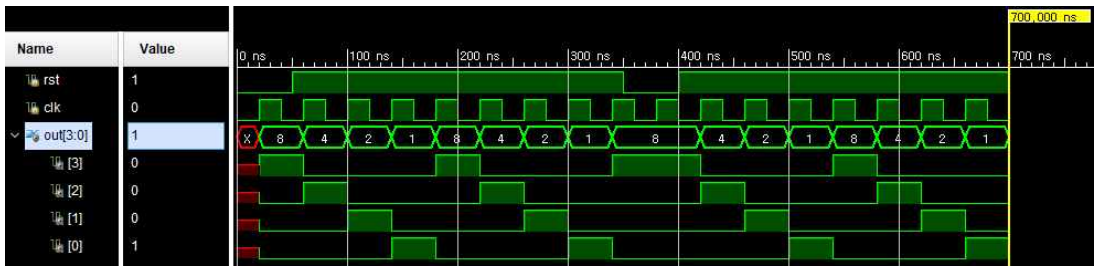
2. 4-bit up/down counter의 결과 및 simulation 과정에 대해서 설명하시오.

Number Clock Transitions	outputs			
	L1	L2	L3	L4
1	1	0	0	0
2	0	1	0	0
3	0	0	1	0
4	0	0	0	1
5	1	0	0	0
6	0	1	0	0
7	0	0	1	0

source code	testbench code

`timescale 1ns / 1ps
 	
module ring_counter(
input rst, clk,
output reg [3:0] out
);
 	
always@ (posedge clk) begin
if (!rst) out <= 4'b1000; // 초기값 reset
else begin
out[3] <= out[0];
out[2] <= out[3];
out[1] <= out[2];
out[0] <= out[1];
end
end
 	
endmodule
	`timescale 1ns / 1ps
	
	module ring_counter_tb;
	
	reg rst, clk;
	wire [3:0] out;
	
	ring_counter u_test(
	.rst (rst), .clk (clk),
	.out (out)
	);
	
	initial begin
	rst = 1'b0; clk = 1'b0; // 시작할 때 값을 reset
	forever #20 clk = ~clk;
	end
	
	initial begin
	#50 // rst
	rst = 1'b1; #300
	rst = 1'b0; #50 // rst
	rst = 1'b1; #300
	$finish;
	end
	
	endmodule

4-bit 8421 decade counter simultation



ring counter는 동일한 data bit를 끊임없이 순환하는 카운터로, shift register의 마지막 출력을 다시 첫 번째 플립 플롭에 입력해 구현할 수 있다. 값을 오른쪽으로 shift하는 임의의 4bit ring counter의 초기 bit가 1011이었다면,

1011 → 1101 → 1110 → 0111 → 1011의 순환을 끊임없이 반복하게 된다.

시뮬레이션 결과, rst 값이 0으로 set되었을 때 회로 상태가 '1000'으로 정상적으로 초기화되었고, 그 외에는 동일한 data bit sequence '1000 - 0100 - 0010 - 0001 - 1000'을 끊임없이 반복하는 것을 확인할 수 있다. 회로가 output table과 동일하게 동작하므로 성공적으로 구현되었음을 알 수 있다.

3. 4-bit up/down counter의 결과 및 simulation 과정에 대해 설명하시오.

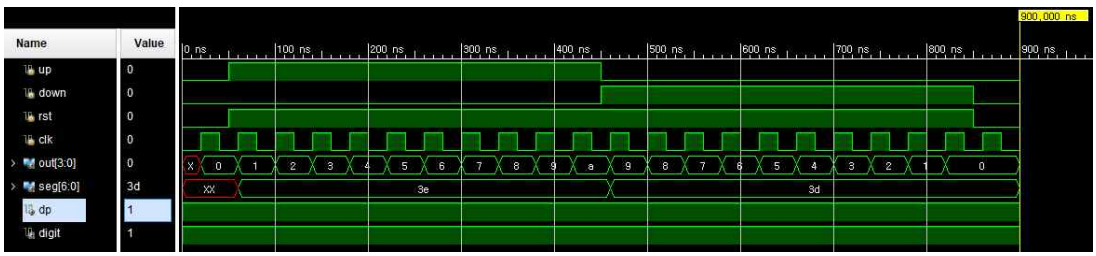
up counter output table					
Number Clock Transitions	outputs				
	L1	L2	L3	L4	DISPLAY
1	0	0	0	0	U
2	0	0	0	1	U
3	0	0	1	0	U
4	0	0	1	1	U
5	0	1	0	0	U
6	0	1	0	1	U
7	0	1	1	0	U

down counter output table					
Number Clock Transitions	outputs				
	L1	L2	L3	L4	DISPLAY
1	0	1	1	0	D
2	0	1	0	1	D
3	0	1	0	0	D
4	0	0	1	1	D
5	0	0	1	0	D
6	0	0	0	1	D
7	0	0	0	0	D

source code	testbench code

`timescale 1ns / 1ps
 	
module updown_counter(
input up, down, rst, clk,
output reg [3:0] out,
output reg [6:0] seg,
output dp, digit
);
assign dp = 1; // 실행 중일 땐 항상 1
assign digit = 1; // 첫 번째 보드에 출력
 	
always@ (posedge clk) begin
if (!rst) out <= 4'b0000;
else if (up) begin
out <= out + 1;
seg[6:0] <= 7'b0111110; // U
end
else if (down) begin
out <= out - 1;
seg[6:0] <= 7'b0111101; // D
end
end
endmodule
	`timescale 1ns / 1ps
	
	module updown_counter_tb;
	
	reg up, down, rst, clk;
	wire [3:0] out;
	wire [6:0] seg;
	wire dp, digit;
	
	updown_counter u_test(
	.up (up), .down (down), .rst (rst), .clk (clk),
	.out (out), .seg (seg), .dp (dp), .digit (digit)
	);
	
	initial begin
	up = 1'b0; down = 1'b0; rst = 1'b0; clk = 1'b0;
	forever #20 clk = ~clk;
	end
	
	initial begin
	#50
	rst = 1'b1; up = 1'b1; #400 // up
	up = 1'b0; down = 1'b1; #400 // down
	down = 1'b0; rst = 1'b0; #50 // rst
	$finish;
	end
	
	endmodule

4-bit up-down counter simultation



up-down counter는 값의 증가, 감소 모두가 가능한 카운터이다. 이번 실험에서는 U, D의 두 가지 입력 bit를 설정하여, U = D = 0일 때는 값을 유지하고, U = 1일 경우 값을 증가시키고, D = 1일 경우엔 값을 감소시키는 카운터를 구현했다. 또 7-segment를 연결하여 7-segment의 첫 번째 display에 U = 1일 땐 U, D = 1일 땐 d를 출력하도록 했다.

reg 변수 out[3:0]을 선언하고, always@ (posedge clk) 문을 이용해 클럭 신호의 rising edge에서 상태가 변화하도록 했다. 추가로 up, down 입력에 따라 조건문을 설정하여 값을 증가시키거나 감소시킨다. 카운터의 구현 외에는 7-segment에서 출력할 값을 담기 위한 seg[6:0] 변수와 출력할 보드를 결정하기 위한 digit 입력이 추가된다. 이번에 구현한 up-down counter의 경우, U, D를 7-segment의 첫 번째 보드가 아닌 다른 보드에 출력시킬 필요가 없으므로 digit을 1로 고정했다.

시뮬레이션 결과, up = 1일 때 값이 증가하고, down = 1일 때는 값이 감소하고 있는 모습을 볼 수 있다. 또 rst = 0일 때 회로가 0000으로 초기화되며, rst = 1일 때는 값이 변화하는 것을 볼 수 있었다. 시뮬레이션은 드러나 있지 않지만 이번에 구현한 up-down counter의 경우 up = down = 0일 때 값이 유지되지만, 이 동작은 구현 시에 요구된 조건은 아니므로 시뮬레이션에서 보이는 것은 생략했다. 결과적으로 up-down counter가 정상적으로 동작하고 있는 것을 확인했다.

4. 결과 검토 및 논의 사항

이번 실험에서는 behavior level의 verilog source code를 이용하여 shift register, ring counter, up-down counter를 구현해 보았으며, FPGA 보드를 활용하여 실제 출력 결과를 확인해 보았다. 실험 결과, output table과 회로의 논리적 동작에 따라 기대되었던 동작이 정상적으로 이루어졌음을 확인할 수 있었다.

5. 추가 이론 조사 및 작성

이번 실험의 경우 behavior-level modeling 방식과 always문을 이용하여 순차 회로를 설계했다. 하지만 behavior-level modeling 방식으로 회로를 설계하는 것이 항상 좋은 것은 아닌데, 회로를 짜는 것은 gate-level modeling에 비해 추상화 수준이 높아 편하지만 내부 회로를 최적화된 방식으로 짜기는 어려워지기 때문이다.

예를 들어, always@ ()문을 사용할 경우, always문 안에서는 reg 변수만을 사용할 수 있기 때문에 예외 경우를 명확히 정의하지 않으면 값을 hold하게 된다. 이 경우 verilog source code가 compile될 때 hold된 값도 회로의 동작에 필요한 값으로 판단하게 되는데, 그 결과 실제 필요하지 않은 latch가 추가되는 등 회로가 복잡해지거나 delay가 늘어날 수 있다.