13주차 결과보고서

전공: 컴퓨터공학 학년: 2학년 학번: 20191629 이름: 이주헌

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

시프트 레지스터는 여러 플립플롭이 연결된 형태로서, 하나의 입력을 받아 그 입력을 바로 하위 또는 상위의 비트로 계속해서 전달하는 회로이다. 따라서, 여기에서는 하나의 입력을 받아 다음 클럭 신호가 올 때까지 입력된 신호를 유지하는 플립플롭인 D 플립플롭을 사용하여 구현하였다. 다음은 시프트 레지스터의 Verilog 소스코드이다.

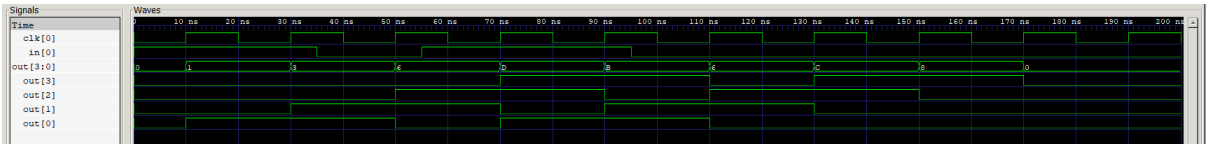
```
`timescale 1ns / 1ps

module four_bit_shift_register(clk, in, out);
	input clk, in;
	output [3:0] out;


	d_flip_flop d1(.d(in), .clk(clk), .q(out[0]), .nq());
	d_flip_flop d2(.d(out[0]), .clk(clk), .q(out[1]), .nq());
	d_flip_flop d3(.d(out[1]), .clk(clk), .q(out[2]), .nq());
	d_flip_flop d4(.d(out[2]), .clk(clk), .q(out[3]), .nq());
endmodule
```

여기서 `d_flip_flop`은 11주차에 구현했던 D 플립플롭 모듈이다. 시프트 레지스터는 한 방향으로 값이 흘러가는 것이 중요하지만, 흘러가는 방향 자체는 중요하지 않기 때문에 방향은 임의로 결정하였다. 이 코드에서는 최하위 비트에서 최상위 비트 방향으로 흘러가도록 구현하였다. 이 신호에 일정한 클럭 신호를 주고, 임의의 장소에서 입력 핀을 잠시 HIGH로 설정하면 매 클럭 신호마다 비트가 상위 비트 방향으로 흐르는 것을 볼 수 있어야 한다. 이러한 동작을 시뮬레이션하는 테스트벤치 코드는 다음과 같다.

```
`timescale 1ns / 1ps

module four_bit_shift_register_tb;
	reg clk, in;
	wire [3:0] out;
```

```
        four_bit_shift_register DUT(clk, in, out);

        initial begin
        $dumpfile("four_bit_shift_register.vcd");
        $dumpvars(1, four_bit_shift_register_tb);

        clk = 0;
        in = 1;

        #35 in = 0;

        #20 in = 1;

        #40 in = 0;

        #200 $finish;
        end

        always clk = #10 ~clk;

endmodule
```

이 코드에서는 총 두 번 입력 핀을 1으로 설정하였다. 또한, 4번의 클럭 사이클이 완료되기 전에 또 다른 입력을 주었으므로, 출력 4비트 중 하나 이상의 1이 존재할 가능성도 있다. 이 코드를 시뮬레이션하면 다음과 같은 결과를 얻을 수 있다.



시뮬레이션 결과가 예상했던 결과와 같은 것을 알 수 있다.

2. 4-bit ring counter의 결과 및 simulation 과정에 대해서 설명하시오.

4비트 원형 계수기는 1000 비트 패턴을 계속해서 시프트하며 반복하는 계수기이다. 이 회로에는 클럭 신호 말고는 입력 신호가 존재하지 않기 때문에 회로의 디자인을 변경하지 않고 비트 패턴을 변경하는 방법은 존재하지 않는다. 같은 패턴을 시프트하면서 반복한다는 점에서 위에서 실험한 시프트 레지스터와 구조가 비슷하다.

원형 계수기의 경우, 시프트 레지스터의 최상위 비트 출력이 다시 최하위 비트 입력으로 들어가는 형식으로 구현되는데, 이 작업만 하게 되면 시뮬레이션 시 초깃값을 설정할 수 없기 때문에 올바르지 않은 값이 도출될 가능성이 있다. 따라서, 미리 초깃값을 알맞게 설정해주는 작업이 중요하다. 이 작업을 거친 뒤의 원형 계수기 Verilog 소스코드는 다음과 같다.

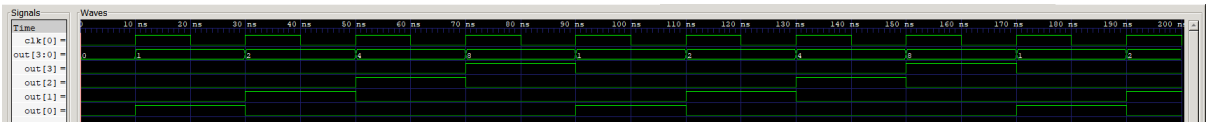
```
`timescale 1ns / 1ps

module four_bit_ring_counter(clk, out);
	input clk;
	output [3:0] out;

	wire [3:0] in;
	reg first;

	initial first = 1;

	always @(posedge clk)
	if (first)
		first = 0;

	assign in[0] = first | out[3];
	assign in[1] = 0 | out[0];
	assign in[2] = 0 | out[1];
	assign in[3] = 0 | out[2];

	d_flip_flop d1(.d(in[0]), .clk(clk), .q(out[0]), .nq());
	d_flip_flop d2(.d(in[1]), .clk(clk), .q(out[1]), .nq());
	d_flip_flop d3(.d(in[2]), .clk(clk), .q(out[2]), .nq());
	d_flip_flop d4(.d(in[3]), .clk(clk), .q(out[3]), .nq());
endmodule
```

원형 계수기의 경우, 클럭 신호를 제외한 입력 신호를 받지 않고, 미리 정해진 패턴을 반복하기 때문에 시뮬레이션을 한다면 한 번에 4개의 출력 비트 중 단 하나의 비트만 1이 되는 패턴을 반복한다고 예상할 수 있다. 일정한 클럭 신호를 주는 테스트벤치 코드는 다음과 같다.

```
`timescale 1ns / 1ps

module four_bit_ring_counter_tb;
	reg clk;
	wire [3:0] out;
```

```
        four_bit_ring_counter DUT(clk, out);

        initial begin
        $dumpfile("four_bit_ring_counter.vcd");
        $dumpvars(0, four_bit_ring_counter_tb);

        clk = 0;
        #200 $finish;
        end

        always clk = #10 ~clk;

endmodule
```

위 코드를 실행시키면 다음과 같은 시뮬레이션 결과를 볼 수 있다.



위에서 예상한 바와 같이 하나의 비트가 계속해서 시프트되는 패턴을 볼 수 있다.

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

4비트 상하향 계수기는 말 그대로 숫자를 1씩 더하거나 1씩 뺄 수 있는 계수기이다. 이전 실습에서 확인하였듯이, 계수기에는 동기 계수기와 비동기 계수기, 두 가지가 존재한다. 그러나 이전 실습에서 이미 동기식 상향 계수기를 구현하였기 때문에 여기서는 동기식 상향 계수기를 개량하여 상하향 계수기를 만들었다. 상향 계수기는 정출력을, 하향 계수기는 역출력을 이용한다는 점을 이용하여 상하향 선택 입력 핀이 사용할 출력 핀을 결정할 수 있도록 구현하였다.

```
`timescale 1ns / 1ps

module four_bit_up_down_counter(clk, rst, down, q);
        input clk, rst, down;
        output [3:0] q;

        wire [3:0] nq;

        jk_flip_flop jk0(
        .j(1),
```

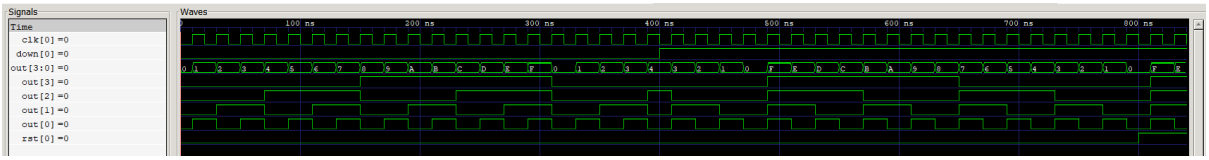
```
        .k(1),
        .clk(clk),
        .rst(rst_combined),
        .q(q[0]),
        .nq(nq[0])
        );
        jk_flip_flop jk1(
        .j((~down & q[0]) | (down & nq[0])),
        .k((~down & q[0]) | (down & nq[0])),
        .clk(clk),
        .rst(rst_combined),
        .q(q[1]),
        .nq(nq[1])
        );
        jk_flip_flop jk2(
        .j((~down & q[1] & q[0]) | (down & nq[1] & nq[0])),
        .k((~down & q[1] & q[0]) | (down & nq[1] & nq[0])),
        .clk(clk),
        .rst(rst_combined),
        .q(q[2]),
        .nq(nq[2])
        );
        jk_flip_flop jk3(
        .j((~down & q[2] & q[1] & q[0]) | (down & nq[2] & nq[1] & nq[0])),
        .k((~down & q[2] & q[1] & q[0]) | (down & nq[2] & nq[1] & nq[0])),
        .clk(clk),
        .rst(rst_combined),
        .q(q[3]),
        .nq(nq[3])
        );

endmodule
```

여기서 down 입력은 active-high 입력으로, 1일 때 숫자를 내려가면서 세도록 회로를 설정하는 역할을 한다.

down 핀을 0으로 설정하여 20회 수를 세고, 다시 down 핀을 1으로 설정하여 20회 수를 센 뒤, 1회 재설정하면 0부터 15까지 세고 0으로 돌아와 3까지 센 뒤 다시 반대로 세는 모양이 나와야 한다.

```
`timescale 1ns / 1ps

module four_bit_up_down_counter_tb;
```

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

        four_bit_up_down_counter DUT(clk, rst, down, out);

        initial begin
        $dumpfile("four_bit_up_down_counter.vcd");
        $dumpvars(1, four_bit_up_down_counter_tb);

        clk = 0;
        down = 0;
        rst = 0;

        // Count up for 20 cycles and set the down flag
        #400 down = ~down;

        // After 20 more cycles, pull reset pin high
        #400 rst = ~rst;

        #40 $finish;
        end

        always clk = #10 ~clk;

endmodule
```



시뮬레이션 결과에서, 설계한 회로가 제대로 동작하는 것을 확인할 수 있다.

4. 결과 검토 및 논의사항

이번 실습에서는 세 종류의 회로를 만들어 보았다. 가장 먼저 만든 것은 4비트 시프트 레지스터로, 여러 개의 D 플립플롭을 연결하여 만들었다. D 플립플롭은 다음 클럭 신호가 들어올 때까지 현재 출력을 유지하는 버퍼와 같은 역할을 하기 때문에, 매 클럭 신호마다 입력을 한 칸씩 옮기는 시프트 레지스터에 사용하기 좋은 모듈이다. 여기서는 이전 실험에서 구현했던 모듈을 완전히 재사용하여 비교적 간단하게 구현할 수 있었다.

그 다음으로 구현한 것은 4비트 원형 계수기이다. 원형 계수기는 단 하나의 1이 매 클럭 신호마다 순환하는 회로로, 시프트 레지스터와 거의 비슷한 동작을 하는 회로이다. 처음에는 시프트 레지스터를 완전히 재사용하려고 하였으나, 시프트 레지스터를 완전히 재사용하는 경우 초깃값이 제대로 설정되지 않아 올바른 결과로 시뮬레이션되지 않았다. 이를 해결하기 위해, 시프트 레지스터의 코드를 기반으로 1과 0을 OR 연산으로 연결하여 초깃값을 제대로 설정해 주었다.

마지막으로 4비트 상하향 계수기를 만들었다. 상하향 계수기는 지난 시간에 만든 계수기와 달리, 숫자를 1씩 더하면서 수를 셀 수도 있고, 1씩 빼면서 수를 셀 수 있는 식이다. 상향 계수기와 하향 계수기 모두 JK 플립플롭을 T플립플롭으로 사용하여 구현한다는 점은 똑같기 때문에, AND와 OR 연산을 적절히 조합하여 하향 계수를 원할 경우 다른 소스로부터 신호를 전달받을 수 있도록 구현하였다.

5. 추가 이론 조사 및 작성

지금까지의 실습에서는 기본적인 플립플롭을 처음부터 구현하여 여러 회로를 만들었지만, FPGA에 내장되어 있는 플립플롭인 레지스터를 사용하면 훨씬 더 간결하게 이러한 코드를 구현할 수 있다. 이러한 레지스터에는 사칙연산과 같은 기본적인 정수 연산이 정의되어 있기 때문에 계수기를 만드는 것도 간단하다. 이번에 실험한 세 개의 회로를 FPGA 내장 레지스터를 사용하여 구현하면 다음과 같다.

```
module four_bit_shift_register(clk, in, out);
      input clk, in;
      output reg [3:0] out;

      always @(posedge clk) begin
            out <= out << 1;
            out[0] <= in;
      end
endmodule
```

```
module four_bit_ring_counter(clk, out);
      input clk;
      output reg [3:0] out;

      initial out = 4'b0001;
      always @(posedge clk) begin
```

```
            out[3] <= out[0]
            for (int i = 0; i < 3; i = i + 1)
                  out[i] <= out[i + 1]
      end

endmodule
```

```
module four_bit_up_down_counter(clk, rst, down, q);
      input clk, rst, down;
      output reg [3:0] q;

      initial q = 4'b0000

      always @(posedge clk) begin
            if (down)
                  q <= q - 1;
            else
                  q <= q + 1;
      end

endmodule
```