**13주차 결과보고서**

**전공: 경영학과 학년: 3학년 학번: 20190963 이름: 한다현**

**1.**

**이번 실험에서는 4개의 플립플롭으로 구성되어 있는 4비트 shift register를 설계하였다. 4비트 shift register는 clock pulse를 의미하는 Clk, reset을 의미하는 rst, 새로운 입력되는 데이터를 의미하는 inx가 입력된다. 또한 4개의 플립플롭에 저장될 데이터 reg[3:0]out이 필요한데, out의 초기 값은 0000으로 설정한다. 이들을 Verilog code로 작성하면 다음과 같다.**

**module shift(**

**input Clk, rst, inx,**

**output out**

**);**

**reg[3:0] out = 0;**

**이 후에 clock pulse가 0에서 1로 변화하는 positive edge에서 시프트 레지스터가 작동하도록 설계한다. Clock pulse가 positive edge일 때, 시프트 레지스터의 동작은 두 가지가 있다. reset이 1이 입력되는 경우와, reset이 0이 입력되는 경우이다. Reset이 1인 경우에는 out이 초기 값인 0000으로 돌아간다. Reset이 0일 때는 각 플립플롭에 있던 데이터들이 왼쪽으로 시프트하고 out[0]에는 새로운 데이터 inx가 입력된다. 이를 Verilog code로 작성하면 다음과 같다.**

**always @(posedge Clk) begin**

**if (rst == 1)**

**out <= 0;**

**else**

**begin**

**out = out << 1;**

**out[0] = inx;**

**end**

**end**

**endmodule**

**위 코드에서 ‘<<’ 연산자는 왼쪽 방향으로 시프트된다는 것을 의미한다. 이렇게 4비트 shift register의 디자인 코드를 작성한 후에는 시뮬레이션 코드를 작성한다. 시뮬레이션 코드는 다음과 같다.**

**module shift\_tb();**

**reg Clk, rst, inx;**

**wire[3:0] out;**

**shift shift\_tb(**

**.Clk(Clk),**

**.rst(rst),**

**.inx(inx),**

**.out(out));**

**always #10 Clk = ~Clk;**

**always #30 inx = ~inx;**

**initial begin**

**Clk <= 0;**

**rst <= 0;**

**inx <= 0;**

**#20 rst <= 0;**

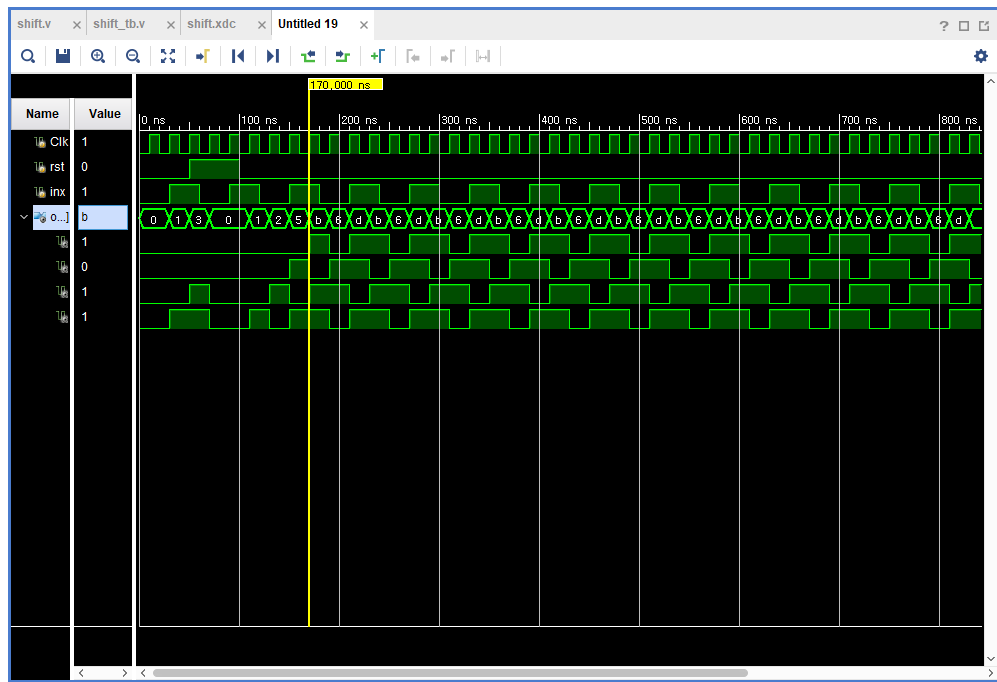
**#30 rst <= 1;**

**#50 rst <= 0;**

**end**

**endmodule**

**시뮬레이션 코드에서는 Clk, rst, inx, out을 선언한 후 각각을 디자인 코드의 변수와 연결한다. 변수들을 연결한 후에는 Clk와 inx의 값이 일정한 간격에 따라 변화하도록 설정한다. Clk는 10ns마다 값이 반전되고, inx는 20ns마다 값이 반전된다. 세 입력 Clk, rst, inx의 초기 값은 모두 0으로 설정한 후 Clk와 inx는 앞서 설계했던 것처럼 일정 간격으로 값이 반전되며, rst는 20ns 후에 rst 0이 되고, 다시 30ns 후에 1이 된다. 이 후 50ns 뒤에 다시 0이 되는데, 이는 50ns ~ 100ns 동안 rst의 값이 1이 된다는 것이다. 이렇게 작성한 시뮬레이션 코드를 실행하면 다음과 같은 시뮬레이션 결과를 볼 수 있다.**

****

**위에서 보여지는 출력 예시는 Clk가 positive edge, rst가 0이고, inx가 1인 경우이다. 이전의 시프트 레지스터에 저장되어 있는 데이터는 0101이다. Clk가 positive edge인 상황에서 rst가 0이기 때문에 데이터가 이동한다. 새로운 데이터 inx가 1이고, 0101이 왼쪽 방향으로 이동하기 때문에 이 때의 출력은 1011이 된다. 반면 rst가 1인 상황에서 Clk가 positive edge라면 시프트 레지스터에 저장되는 데이터는 초기 상태인 0000이 된다. 이 후 rst가 다시 0이 된 후 Clk가 positive edge이고, inx가 1이 되면 0001이 된다.**

**2.**

**링카운터는 시프트 레지스터를 활용한 카운터인데, clock pulse가 입력되면 레지스터에 저장된 데이터가 한쪽 방향으로 이동하고, 마지막 플립플롭의 출력이 처음 플립플롭에 입력된다. 이번 실험에서는 4비트 ring counter를 설계하였다. 링카운터는 새로운 데이터를 입력 받지 않고, 마지막 플립플롭의 출력이 다시 처음 플립플롭에 입력되기 때문에 앞서 설계한 shift register와 달리 inx라는 변수가 필요하지 않다. inx외에 Clk, rst은 동일하게 선언되고, out의 경우 초기 값이 0000이나 1111이라면 시프트가 발생하더라도 항상 0000이 저장되기 때문에 다른 값을 할당해야 하는데, 이번 실험에서는 out의 초기 값으로 1000을 사용하였다. 이를 Verilog code로 나타내면 다음과 같다.**

**module ring(**

**input Clk, rst,**

**output out**

**);**

**reg[3:0] out = 4'b1000;**

**이 후 시프트 레지스터를 설계할 때와 동일하게 Clk가 positive edge일 때 링 카운터가 작동하도록 설계한다. Clk가 positive edge일 때 rst의 값이 1이면 링 카운터는 초기 값으로 돌아간다. 링 카운터의 초기 값이 1000이므로 rst의 값이 1일 때 링 카운터의 상태는 1000이 된다. Clk가 positive edge일 때, rst의 값이 0이면 각 플립플롭에 저장되어 있는 데이터가 왼쪽으로 이동하고, 마지막 플립플롭의 출력이 처음 플립플롭에 저장된다. 이를 Verilog code로 작성하면 다음과 같다.**

**always @(posedge Clk) begin**

**if (rst == 1)**

**out = 4'b1000;**

**else**

**begin**

**out[0] <= out[3];**

**out[1] <= out[0];**

**out[2] <= out[1];**

**out[3] <= out[2];**

**end**

**end**

**endmodule**

**이렇게 4비트 링카운터의 디자인 코드를 작성한 후에는 시뮬레이션 코드를 작성해야 한다. 링카운터의 시뮬레이션 코드는 다음과 같다.**

**module ring\_tb();**

**reg Clk, rst;**

**wire[3:0] out;**

**ring ring\_tb(**

**.Clk(Clk),**

**.rst(rst),**

**.out(out));**

**always #10 Clk = ~Clk;**

**initial begin**

**Clk <= 0;**

**rst <= 0;**

**#20 rst <= 0;**

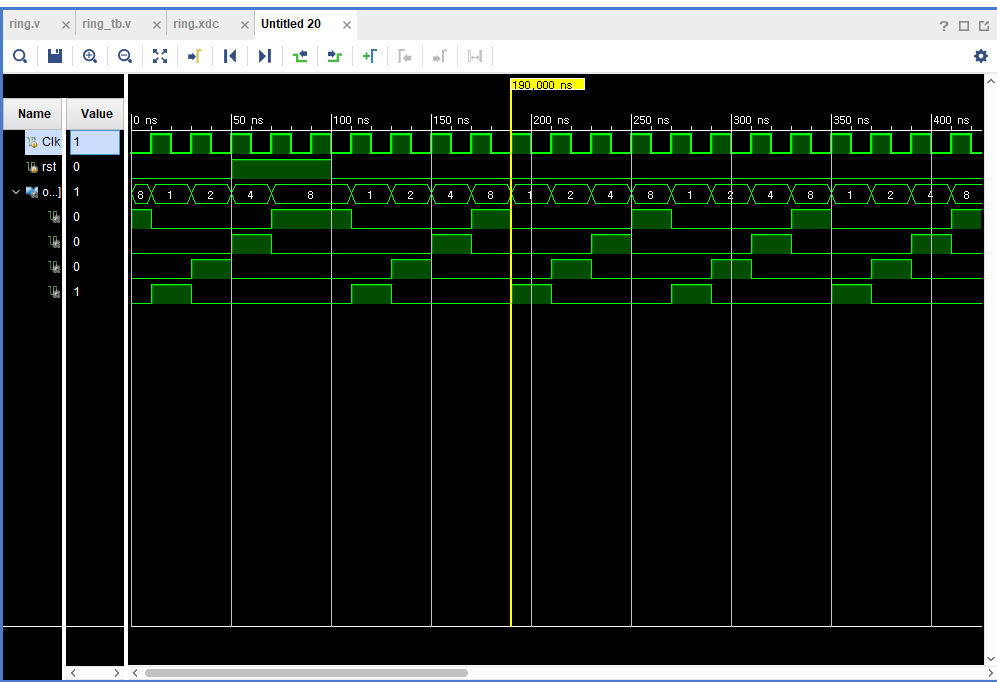
**#30 rst <= 1;**

**#50 rst <= 0;**

**end**

**endmodule**

**Clk, rst, out을 선언한 뒤 각각을 디자인 코드의 입력, 출력과 연결한다. Clk는 10ns마다 값이 반전되도록 설계한다. Clk와 rst의 초기 값을 모두 0으로 설정하고, Clk는 앞서 작성한 코드와 같이 일정 간격마다 값이 반전되고, rst는 20ns 뒤에 0, 다시 30ns 뒤에 1, 50ns 뒤에 다시 0이 되도록 설계한다. 즉, rst의 값은 50ns ~ 100ns 동안 1이 되고, 나머지 시간 동안에는 0이 된다. 이렇게 작성한 시뮬레이션 코드를 실행하면 다음과 같은 시뮬레이션 결과를 볼 수 있다.**

****

**위의 시뮬레이션 결과에서 보여지는 예시는 Clk가 positive edge이고, rst이 0이고 이전 상태가 1000인 경우이다. Clk가 positive edge이므로 링 카운터가 동작하고, rst가 0이기 때문에 초기 상태로 돌아가지 않는다. 1000에서 링카운터가 동작하면, 마지막 플립플롭의 출력인 1이 처음 플립플롭에 입력되고, 나머지 데이터들은 왼쪽으로 이동한다. 따라서 링 카운터의 상태는 0001이 된다. 이 후 다시 Clk가 positive edge이고 rst가 0이라면, 같은 원리로 링카운터의 상태는 0010이 된다. 반면 Reset이 1인 상태에서 Clk가 positive edge가 되면, 링 카운터가 리셋된다. 이번 실험에서 링 카운터의 초기 값은 1000이기 때문에, 링카운터의 상태는 1000이 된다.**

**3.**

**업 카운터는 순차적으로 증가하는 카운터이고, 다운 카운터는 순차적으로 감소하는 카운터이다. 이번 실험에서는 ub라는 입력에 따라 업 카운터로 동작하기도 하고, 다운 카운터로 동작하기도 하는 4비트 업다운 카운터를 설계하였다. 이 카운터의 입력은 Clk, rst, ub이고 출력은 out이다. Ub는 이 카운터가 업 카운터로 동작할지 다운 카운터로 동작할지 결정하는 입력이다. Ub가 1이 되면 업 카운터로 동작하고, ub가 0이 되면 다운 카운터로 동작한다. 또한 업다운 카운터를 실행시킬 때, seven-segment display를 사용하였는데, ub의 값이 1이 되어 업 카운터로 동작하는 경우에는 seven-segment display에 U가 출력되고, ub의 값이 0이 되어 다운 카운터로 동작하는 경우에는 seven-segment display에 d가 출력된다. 따라서 output에 seven-segment를 나타내는 a, b, c, d, e, f, g와 여러 개의 seven-segment 중 결과가 출력될 seven-segment를 나타내는 digit이 필요하다. 이를 Verilog code로 작성하면 다음과 같다.**

**module updown(**

**input Clk, rst, ub,**

**output[3:0] out,**

**output a, b, c, d, e, f, g, digit**

**);**

**reg[3:0] out;**

**reg a, b, c, d, e, f, g, digit;**

**초기에는 카운터가 동작하지 않기 때문에 a~g와 digit의 값을 0으로 설정한다. 또한 이 카운터의 초기 상태를 0000으로 설정한다. 이후에 Clk가 positive edge일 때 카운터가 작동하도록 설계하고, ub가 1인 경우에는 이전 상태보다 1이 증가하고, seven-segment에서 U가 출력되도록 설계한다. 반대로, ub가 0인 경우에는 이전 상태보다 1이 감소하고, seven-segment에서 d가 출력되도록 설계한다. 이를 Verilog code로 작성하면 다음과 같다.**

**initial begin**

**out = 0;**

**a = 0;**

**b = 0;**

**c = 0;**

**d = 0;**

**e = 0;**

**f = 0;**

**g = 0;**

**digit = 0;**

**end**

**always @(posedge Clk) begin**

**if (rst == 1)**

**out = 0;**

**else**

**begin**

**if (ub == 1) begin**

**out <= out + 1;**

**a <= 0;**

**b <= 1;**

**c <= 1;**

**d <= 1;**

**e <= 1;**

**f <= 1;**

**g <= 0;**

**digit <= 1;**

**end**

**else begin**

**out <= out - 1;**

**a <= 0;**

**b <= 1;**

**c <= 1;**

**d <= 1;**

**e <= 1;**

**f <= 0;**

**g <= 1;**

**digit <= 1;**

**end**

**end**

**end**

**endmodule**

**if문을 사용하여 ub가 1인 경우와 ub가 0인 경우를 구분하여 설계하였다. 이렇게 업다운 카운터의 디자인 코드를 작성하였다면 다음으로 시뮬레이션 코드를 작성해야 한다. 업다운 카운터의 시뮬레이션 코드는 다음과 같다.**

**module updown\_tb();**

**reg Clk, rst, ub;**

**wire[3:0] out;**

**wire a, b, c, d, e, f, g, digit;**

**updown updown\_tb(**

**.Clk(Clk),**

**.rst(rst),**

**.ub(ub),**

**.out(out),**

**.a(a),**

**.b(b),**

**.c(c),**

**.d(d),**

**.e(e),**

**.f(f),**

**.g(g),**

**.digit(digit));**

**always #10 Clk = ~Clk;**

**always #100 ub = ~ub;**

**initial begin**

**Clk <= 0;**

**rst <= 0;**

**ub <= 1;**

**#20 rst <= 0;**

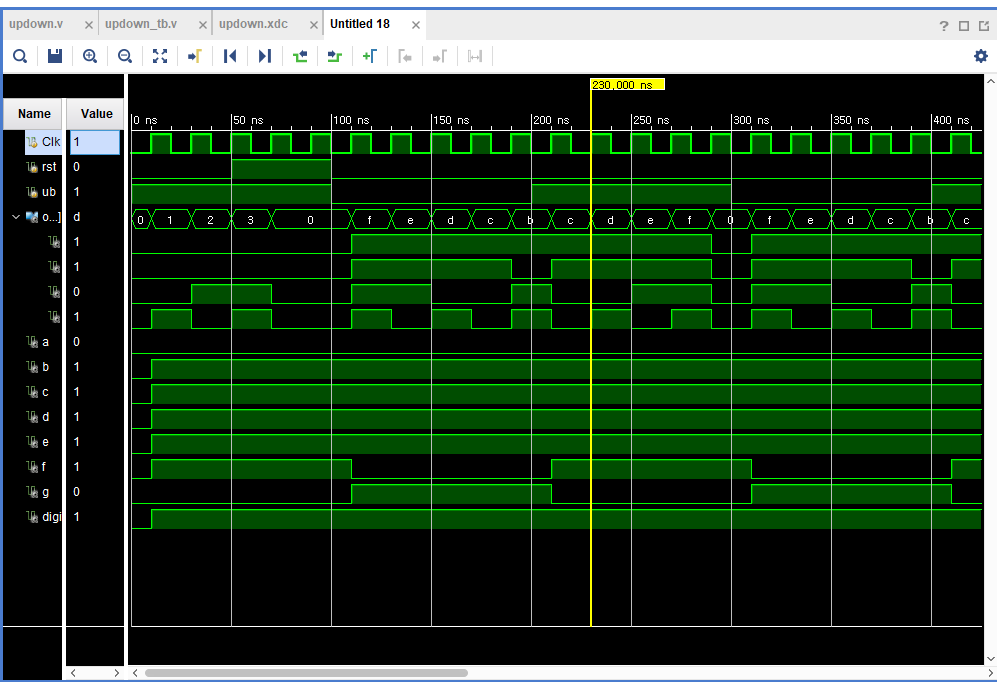
**#30 rst <= 1;**

**#50 rst <= 0;**

**end**

**endmodule**

**먼저, Clk, rst, ub, out, a~g, 그리고 digit을 선언한 뒤 이를 디자인 코드와 연결시킨다. Clk는 10ns마다 값이 반전되고 ub는 100ns마다 값이 반전되도록 설정하고, Clk, rst, ub의 초기 값을 모두 0으로 설정한다. Clk와 ub는 일정 간격마다 값이 반전되고, rst는 20ns 뒤에 0, 30ns 뒤에 1, 그리고 50ns 뒤에 다시 0이 되도록 설정한다. 즉, rst는 50ns ~ 100ns동안 1이 되고, 나머지 시간에는 0이 된다. 이러한 시뮬레이션 코드를 통해 시뮬레이션 결과를 확인하면 다음과 같다.**



**위에서 보여지는 출력 예시는 Clk가 positive edge, rst가 0, ub가 1, 이전 출력이 1100인 경우이다. Clk가 positive edge일 때 rst가 0이기 때문에 상태가 초기화 되지 않고 카운터가 동작한다. 이 때, ub가 1이기 때문에 업 카운터로 동작한다. 이전 출력이 1100인 상태에서 업 카운터가 동작하면 1이 증가하여 출력은 1101이 된다. 또한 ub가 1인 경우에는 seven-segment에 U가 출력되어야 하기 때문에 a와 g를 제외한 b~f의 값이 1이 되어 seven-segment에 U가 출력된다. 반대로 ub가 0인 경우에는 다운 카운터로 작동하는데, 이전 출력이 1101인 상태에서 ub와 rst가 0인 상태로 Clk가 positive edge가 되면, 1101에서 1이 감소한 1100이 출력된다. 또한 seven-segment에는 d가 출력되어야 하기 때문에 a와 f를 제외한 b~e와 g의 값이 1이 되어 seven-segment에 d가 출력된다.**

**4.**

**이번 실험에서는 4 bit shift register, 4 bit ring counter, 4 bit updown counter를 설계하였다. 4비트 시프트 레지스터는 Clk가 positive edge일 때 동작하며, 이 때 reset이 1이면 초기 상태로 되돌아가고, reset의 값이 0이면 시프트 레지스터에 저장되어 있던 데이터가 왼쪽 방향으로 이동하였다. 또한 시프트 레지스터에는 새로 입력되는 데이터가 필요한데, 이를 inx로 선언하였다. Inx의 값이 1이면 처음 플립플롭에 1이 저장되며, inx의 값이 0이면 처음 플립플롭에 0이 저장되고, 나머지 데이터들은 왼쪽 방향으로 이동한다. 4 bit ring counter는 4개의 플립플롭으로 구성되어 있으며 shift register를 응용한 카운터이다. 링 카운터는 Clk가 positive edge일 때 작동하며, 이 때 reset이 1이면 초기 상태로 돌아가고, reset의 값이 1이면 링 카운터에 저장되어 있던 데이터가 왼쪽 방향으로 이동하며 이 때 마지막 플립플롭의 출력이 처음 플립플롭의 입력이 된다. 따라서 shift register와 달리 새로운 입력 inx가 필요하지 않고, Clk가 positive edge일 때, 새로운 입력 없이 동작한다. 또한 링 카운터의 초기 상태는 0000이나 1111이 되지 않아야 한다. 0000이나 1111이 초기 상태일 경우, Clk가 positive edge가 되어 왼쪽 방향으로 시프트가 일어나도 링 카운터의 상태는 여전히 0000이나 1111이기 때문이다. 이렇게 되면 계수를 할 수 없기 때문에 초기 상태는 1000, 0111 등으로 설정하여야 한다. 마지막으로 이번 실험에서 설계한 업다운 카운터는 ub라는 입력에 따라 업 카운터로 동작할지, 다운 카운터로 동작할지 결정되도록 설계하였다. 입력 ub가 1이면 업 카운터로 동작하고, 0이면 다운 카운터로 동작하도록 설정하였다. 업 카운터로 동작할 때, 이전 상태가 0000이라면 1이 증가한 0001이 출력되고, 다운 카운터로 동작할 때, 이전 상태가 1111이라면 1이 감소한 1110이 출력되었다. 또한 이번 실험에서 설계한 업다운 카운터는 seven-segment를 사용하였다. 업 카운터로 동작할 때는 seven-segment에 U가 출력되고, 다운 카운터로 동작할 때는 seven-segment에 d가 출력되도록 설계하였다. 이번 실험에서는 직렬입력-직렬출력 방식의 시프트 레지스터를 설계하였는데, 다른 종류의 시프트 레지스터 어떻게 설계되는지 고민해볼 필요가 있다고 생각한다.**

**5.**

**시프트 레지스터를 활용한 카운터에는 링 카운터 외에도 존슨 카운터가 있다. 존슨 카운터는 n개의 플립플롭으로 이루어져 있으며, 각 플립플롭의 출력은 다음 플립플롭의 입력이 된다. 그리고 마지막 플립플롭의 출력의 부정이 처음 플립플롭의 입력이 된다. 링 카운터는 마지막 플립플롭의 출력 Q가 처음 플립플롭에 입력된다. 반면 존슨 카운터는 마지막 플립플롭의 Q’이 처음 플립플롭에 입력된다. 4비트 존슨 카운터를 예로 들면, 0000인 상태에서 clock pulse가 입력되면 0의 부정인 1이 처음 플립플롭에 입력되어 존슨 카운터는 0001을 출력한다. 다시 clock pulse가 입력되면 마찬가지로 0의 부정인 1이 처음 플립플롭에 입력되어 0011이 된다. 이와 같은 과정을 반복하면 0111, 1111, 1110, 1100, 1000, 0000 … 순서로 출력된다. 즉, 4비트 존슨 카운터는 0000부터 1000까지 총 8가지의 수를 셀 수 있다. 따라서 n비트 존슨 카운터는 2n개의 수를 셀 수 있으며 n개의 수를 셀 수 있는 n비트 링 카운터보다 2배의 수를 셀 수 있다.**