2018년 2학기 컴퓨터공학실험Ⅱ
CSE3016-05반 13주차 결과 보고서

학번: 20171665
이름: 이 선 호

2018. 12. 14

목 차

Ⅰ 실험 결과

1. 4-Bit Shift Register

1) Verilog 코딩

inv.v

```
`timescale 1ns / 1ps

module inv(din, clk, reset, dout);

input din, clk, reset;
output [3:0]dout;
wire [2:0]s;

d_flip_flop u0 (.din(din),
.clk(clk),
.reset(reset),
.dout(s[0]));

assign dout[0] = s[0];

d_flip_flop u1 (.din(s[0]),
.clk(clk),
.reset(reset),
.dout(s[1]));

assign dout[1] = s[1];

d_flip_flop u2 (.din(s[1]),
.clk(clk),
.reset(reset),
.dout(s[2]));

assign dout[2] = s[2];

d_flip_flop u3 (.din(s[2]),
.clk(clk),
.reset(reset),
.dout(dout[3]));
```

```
endmodule
```

d_flip_flop.v

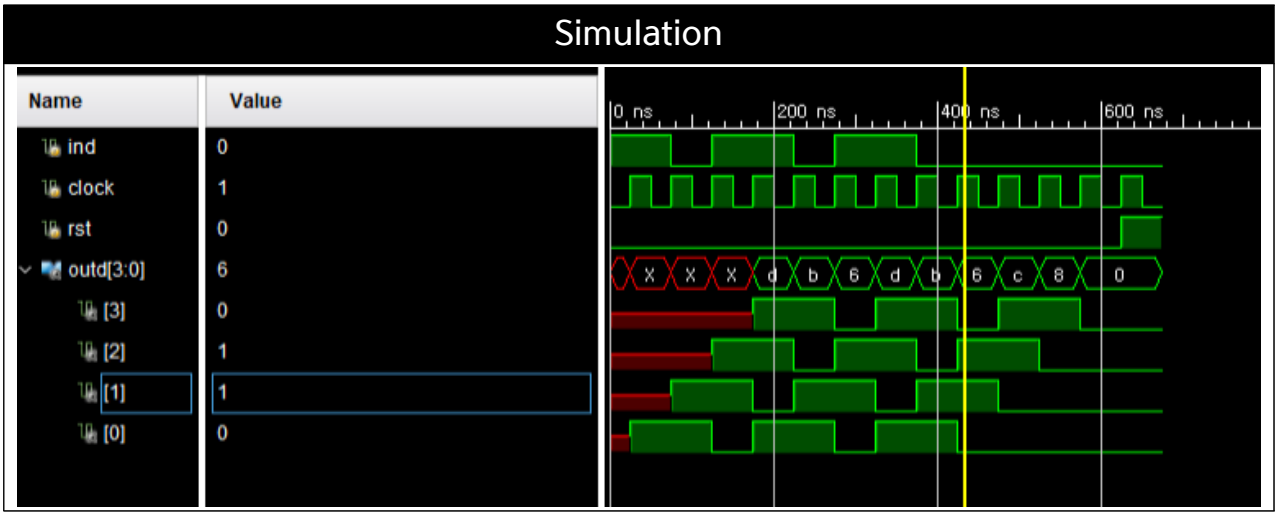
```
`timescale 1ns / 1ps

module d_flip_flop(din, clk, reset, dout);

output dout;
reg dout;

input din, clk, reset;

always @ (posedge clk)
begin
if(reset)
dout <= 0;
else
dout <= din;
end

endmodule
```

inv_tb.v

```
`timescale 1ns / 1ps

module inv_tb();

reg ind, clock, rst;
wire [3:0]outd;

inv shift_register(
.din(ind), .clk(clock), .reset(rst), .dout(outd));

initial
begin
ind = 1;
clock = 0;
rst = 0;
end
```

```
always
begin
#25 clock = ~clock;
end

initial
begin
#25 ind <= 1;
#50 ind <= 0;
#50 ind <= 1;
#50 ind <= 1;
#50 ind <= 0;
#50 ind <= 1;
#50 ind <= 1;
#50 ind <= 0;
#50 ind <= 0;

#200 rst <= 1;
#50;
$finish;

end

endmodule
```

2) Simulation 결과



3) 과정

Shift Register는 일련의 플립플롭이 직렬로 연결되어서 한 플립플롭의 출력이 다음 플립플롭의 입력으로 가는 회로이다. 그래서 플립플롭의 메모리 상태는 Clock이 바뀔 때마다 우측으로 하나씩 밀려간다. 이번 실습에서는 D 플립플롭을 사용하여 4개의 비트를 갖는 Shift Register를 제작해야 했다. 따라서 Shift Register 회로를 구현하기 전에 이전 시간에 구현했던 D 플립플롭을 새로운 모듈로 제작하여 다시 구현했다.

D 플립플롭 모듈(d_flip_flop)의 포트는 입력 D(din)와 Clock(clk), Reset(reset), 그리고 플립플롭의 출력(dout)을 설정했다. Clock이 0에서 1로 설정될 때마다 만약 reset이 0이면 입력 받은 din을 dout에 non-blocking으로 할당한다. reset이 1이면 dout을 0으로 할당한다. Shift Register 모듈에서 4개의 D 플립플롭 모듈에 첫 번째 D 플립플롭을 제외한 나머지 D 플립플롭에서 이전 D 플립플롭의 출력과 같은 값을 갖는 wire s와 clk, reset을 입력받고, 각 플립플롭에 맞는 다음 출력 s에 연결한다. 각 wire s를 각각의 D 플립플롭 출력인 dout에 assign 구문을 사용하여 할당하면 Shift Register가 완성된다. Shift Register 모듈에서 dout을 output 포트로 선언했기 때문에 D 플립플롭 모듈의 input으로 연결하지 못하기 때문에 wire s를 사용하여 각 플립플롭의 출력과 입력을 연속적으로 연결하고 각 wire의 값을 output에 할당한 것이다.

Simulation 코드에서는 input인 ind, clk, rst를 레지스터로 선언하고, 각 D 플립플롭의 출력을 wire로 선언했다. 그리고 디자인 소스 코드가 저장된 inv 모듈에 각 포트에 맞게 연결했다. 이번 Shift Register에서는 Clock이 0에서 1로 설정될 때 output이 변화하기 때문에 Clock을 25ns 주기를 갖도록 설정하고 ind는 Clock이 0에서 1이 되기 바로 직전에 미리 입력하고자 하는 값으로 설정되어 있어야 하므로 처음에 25ns에 원하는 값을 처음으로 입력한 다음 그 이후에는 50ns 주기를 가지면서 입력하고자 하는 값을 변화시켰다. reset이 제대로 작동하는지 확인하기 위해 625ns가 지난 후에는 reset을 1로 변경했다.

2. 4-Bit Ring Counter

1) Verilog 코딩

inv.v

```
`timescale 1ns / 1ps

module inv(clk, reset, dout);

input clk, reset;
output reg [3:0]dout;
```

```
wire [3:0] s;

initial
begin
    dout[0] = 1'b1;
    dout[1] = 1'b0;
    dout[2] = 1'b0;
    dout[3] = 1'b0;
end

d_flip_flop u0 (.din(dout[3]),
.clk(clk),
.reset(reset),
.dout(s[0]));

d_flip_flop u1 (.din(dout[0]),
.clk(clk),
.reset(reset),
.dout(s[1]));

d_flip_flop u2 (.din(dout[1]),
.clk(clk),
.reset(reset),
.dout(s[2]));

d_flip_flop u3 (.din(dout[2]),
.clk(clk),
.reset(reset),
.dout(s[3]));

always @ (s[0] or s[1] or s[2] or s[3])
begin
    if(reset)
    begin
        dout[0] <= 1'b1;
        dout[1] <= 1'b0;
        dout[2] <= 1'b0;
        dout[3] <= 1'b0;
    end
    else if(~reset)
    begin
        dout[0] <= s[0];
```

```
        dout[1] <= s[1];
        dout[2] <= s[2];
        dout[3] <= s[3];
    end
end

endmodule
```

d_flip_flop.v

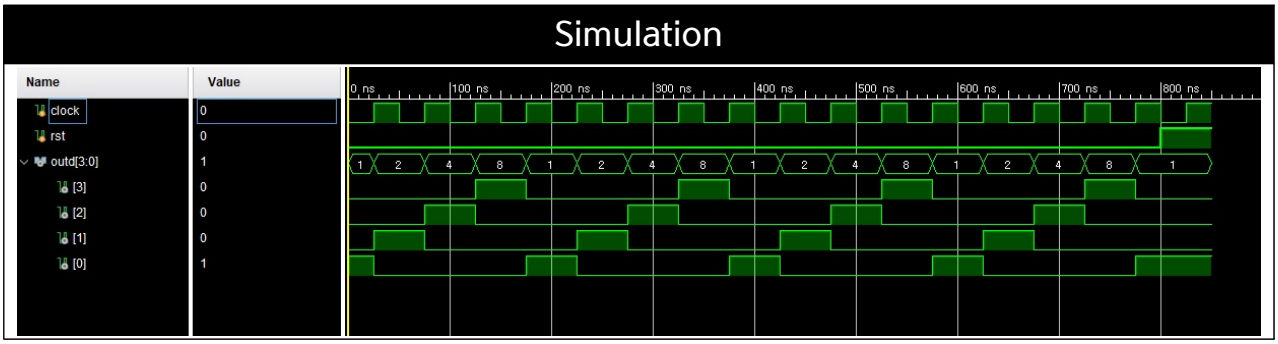
```
`timescale 1ns / 1ps

module d_flip_flop(din, clk, reset, dout);

output dout;
reg dout;

input din, clk, reset;

always @ (posedge clk)
begin
if(reset)
dout <= 0;
else
dout <= din;
end

endmodule
```

inv_tb.v

```
`timescale 1ns / 1ps

module inv_tb();

reg clock, rst;
wire [3:0]outd;

inv ring_counter(
.clk(clock), .reset(rst), .dout(outd));

initial
begin
clock = 0;
rst = 0;
```

```
end

always
begin
#25 clock = ~clock;
end

initial
begin
#800 rst <= 1;
#50;
$finish;

end

endmodule
```

3) Simulation 결과



4) 과정

Ring Counter는 일련의 플립플롭이 직렬로 연결되어서 한 플립플롭의 출력이 다음 플립플롭의 입력으로 가는 Shift Register와 유사하며, 차이점이 존재한다면 output을 비트열로 나타냈을 때 하나의 비트만 1이고 나머지는 0이어야 한다. 또한, 맨 마지막의 플립플롭의 출력이 처음에 위치한 플립플롭의 입력으로 들어가야 한다. 그래서 단순히 앞서 보았던 Shift Register에서 맨 마지막 D 플립플롭의 출력 dout[3] 또는 s[3]을 첫 번째 D 플립플롭 모듈의 input으로 설정할 수가 없었다. 이를 해결하기 위해 dout을 output에서 output register type으로 변경시키고, s값이 변화함에 따라 매번 dout의 값을 변화시키는 구문을 추가했다. 그리고 입력을 받는 input이 따로 존재하지 않기 때문에 초기 상태가 존재해야 한다. 따라서 initial 구문을 작성하여 구문 안에서 dout을 1000으로 설정했다. 또한 always 구문에서 s[0], s[1], s[2], s[3] 값 중에서 하나라도 변화가 일어나면, 만약 reset이 0일 경우 dout을 초기 상태로 되돌리고 그렇지

않을 경우 s의 값을 dout에 non-blocking으로 할당하는 과정을 작성했다.

D 플립플롭 모듈의 포트는 입력 값인 din과 Clock(clk), Reset(reset), 그리고 플립플롭의 출력(dout)을 설정했다. Clock이 0에서 1로 설정될 때마다 만약 reset이 0이면 입력 받은 din을 dout에 non-blocking으로 할당한다. reset이 1이면 dout을 0으로 할당한다. Ring Counter 모듈에서 4개의 D 플립플롭 모듈에 첫 번째 D 플립플롭을 제외한 나머지 D 플립플롭에서 이전 D 플립플롭의 출력과 같은 값을 갖는 해당 dout element와 clk, reset을 입력받고, 각 플립플롭에 맞는 다음 출력 s에 연결한다. Ring Counter에서와 다르게 dout을 입력받을 수 이유는 dout을 앞에서 output register로 선언했기 때문이다. 각 wire s를 각각의 D 플립플롭 출력인 dout에 assign 구문을 사용하여 할당하면 Shift Register가 완성된다. Ring Counter 모듈에서 dout을 output 포트로 선언했기 때문에 D 플립플롭 모듈의 input으로 연결하지 못하기 때문에 wire s를 사용하여 각 플립플롭의 출력과 입력을 연속적으로 연결하고 각 wire의 값을 output에 할당한 것이다.

Simulation 코드에서는 input을 필요로 하지 않으므로 clk, rst만을 레지스터로 선언하고, 각 D 플립플롭의 출력을 wire로 선언했다. 그리고 디자인 소스 코드가 저장된 inv 모듈에 각 포트에 맞게 연결했다. 이번 Ring Counter에서는 Clock이 0에서 1로 설정될 때 output이 변화하기 때문에 Clock을 25ns 주기를 갖도록 설정했다. input이 존재하지 않기 때문에 시간에 따른 input의 변화는 구현하지 않았다. reset이 제대로 작동하는지 확인하기 위해 800ns가 지난 후에 reset을 1로 설정했다.

3. 4-bit Up/Down Counter

1) Verilog 코딩

inv.v

```
`timescale 1ns / 1ps

module inv (clk, L, updown, UD, seg);
input clk, updown;
output [3:0] L;
output UD;
output reg [7:0] seg;
reg [3:0] L;

initial
begin
```

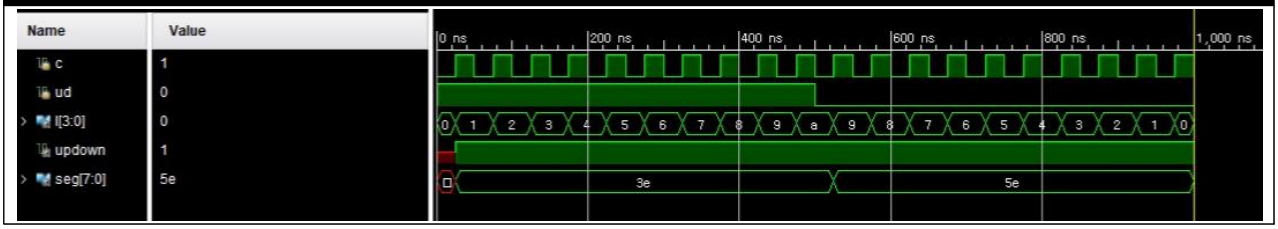
```
L <= 4'b0000;
end


always @(posedge clk)
begin
    if(updown == 1)
        begin
            seg[0] = 0;
            seg[1] = 1;
            seg[2] = 1;
            seg[3] = 1;
            seg[4] = 1;
            seg[5] = 1 && updown;
            seg[6] = 1 && ~updown;
            seg[7] = 0;
            if(updown == 15)
            L <= 0;
            else
            L <= L + 1;
        end
    else
        begin
            seg[0] = 0;
            seg[1] = 1;
            seg[2] = 1;
            seg[3] = 1;
            seg[4] = 1;
            seg[5] = 1 && updown;
            seg[6] = 1 && ~updown;
            seg[7] = 0;
            if(updown == 0)
            L <= 15;
            else
            L <= L - 1;
        end
end

assign UD = seg[0] || seg[1] || seg[2] || seg[3] || seg[4] || seg[5] || seg[6] || seg[7];

endmodule
```

inv_tb.v

```
`timescale 1ns / 1ps
module inv_tb;
reg c, ud;
wire [3:0] l;
wire updown;
wire [7:0] seg;

inv up_down_counter(
.clk(c), .L(l), .updown(ud), .UD(updown), .seg(seg)
);

always
begin
#25 c= ~c;
end

initial begin
    c <= 0;
    ud <= 1;
    #500 ud <= ~ud;
    #500 $finish;
end

endmodule
```

3) Simulation 결과

Simulation



4) 과정

4-Bit Up/Down Counter에서는 플립플롭을 사용하지 않고 inv 모듈에서 always 구문을 이용하여 구현했다. 이번 4-Bit Up/Down Counter에서는 Clock이 0에서 1로 올라갈 때 Counter의 현재 상태가 바뀌어야 되므로 always의 조건을 Clock을 의미하는 변수 clk가 posedge될 때로 설정했다. 그리고 Counter의 현재 상태를 저장하는 변수 L의 데이터 type을 output register로 선언하고 변수의 최상위 비트와 최하위 비트를 정하

기 위해 [3:0]으로 선언했다. 그리고 Counter에서 현재 상태를 상승시킬 때와 하강시킬 때의 두 가지 상태에 관한 input을 필요로 하므로 이를 입력받는 updown 변수를 선언한다. 또한, FPGA에서 해당 Counter가 잘 작동되는지 확인하기 위해 현재 updown이 어떠한 상태를 가지는지 확인할 수 있도록 LED로 정보를 표시해야 한다. 이를 7-segment display로 구현하기 위해 7개의 비트를 갖는 output register인 seg를 선언하고, 한 개의 LED에 seg의 데이터를 모두 표시하기 위해 output UD를 선언했다. L의 초기 상태를 0부터 시작하기 위해 initial 구문을 사용하여 non-blocking으로 L을 0000으로 할당했다.

Clock이 0에서 1로 바뀌었을 때, 만약 현재 updown 상태가 up인 1이라면 segment display에는 u를 표시해야 하므로 첫 번째부터 다섯 번째 segment가 모두 1이 되도록 한다. 그리고 L을 1만큼 증가시키는데, L이 현재 15라면 L을 0으로 설정한다. 만약 현재 updown 상태가 down인 0이라면 segement display에는 d를 표시해야 하므로 첫 번째부터 네 번째 그리고 여섯 번째의 segement가 1로 설정되도록 한다. 그리고 L을 1만큼 감소시키는데. L이 현재 0이라면 L을 15로 설정한다.

7-segment display를 LED에 최종적으로 표시하기 위해 output UD에 각 segment를 OR 연산한 값을 할당했다.

Simulation 코드에서는 ud, clk, rst를 레지스터로 선언하고, output에 해당하는 wire, l, updown을 wire로 선언했다. 그리고 디자인 소스 코드가 저장된 inv 모듈에 각 포트에 맞게 연결했다. 이번 Up/down Counter에서는 Clock이 0에서 1로 설정될 때 output이 변화하기 때문에 Clock을 25ns 주기를 갖도록 설정했다. 처음에는 updown 상태를 1로 설정하여 증가하는 Counter를 시험하고, 그 이후에 감소하는 Counter를 시행해보기 위해 처음에 1로 설정한 ud를 500ns가 지난 후에 0으로 바뀌게끔 했다.

Ⅱ 논의

1. 결과 검토 및 논의사항

실험 결과에서 예비보고서에서 조사한 내용과 일치한다는 사실을 알 수 있다. 이번 실습의 simulation 결과를 truth table로 정리하면 다음과 같다.

Shift Register OUTPUT TABLE					
Number Clock Transitions	OUTPUTS				
↑	IN	L1	L2	L3	L4
1	1	1	0	0	0

2	1	1	1	0	0
3	0	0	1	1	0
4	1	1	0	1	1
5	0	0	1	0	1
6	1	1	0	1	0
7	1	1	1	0	1

Ring Counter OUTPUT TABLE				
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

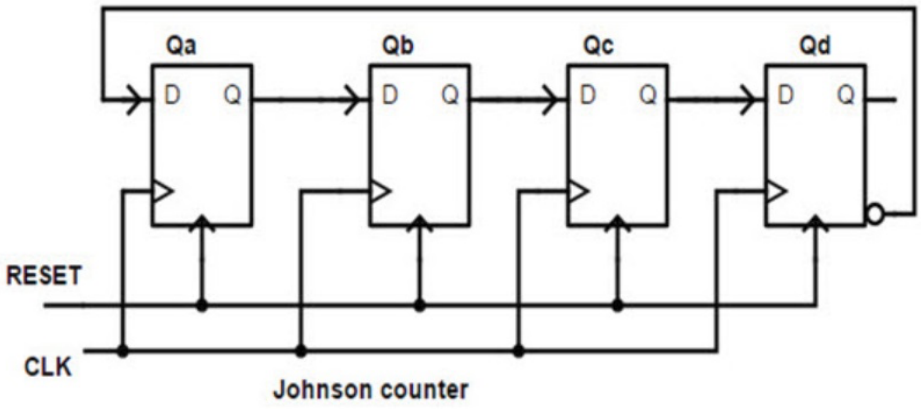
UP Counter OUTPUT TABLE					
Number Clock Transitions	OUTPUTS				
↑	L1	L2	L3	L4	DISPLAY
1	0	0	0	0	U
2	0	0	0	1	U
3	0	0	1	0	U
4	0	0	1	1	U
5	0	1	0	0	U
6	0	1	0	1	U
7	0	1	1	0	U

DOWN Counter OUTPUT TABLE					
Number Clock Transitions	OUTPUTS				
↑	L1	L2	L3	L4	DISPLAY
1	1	1	1	1	D
2	1	1	1	0	D
3	1	1	0	1	D
4	1	1	0	0	D
5	1	0	1	1	D
6	1	0	1	0	D
7	1	0	0	1	D

2. 추가 이론 조사 및 작성

Johnson Counter는 Ring Counter를 약간 변형한 것이며, 맨 마지막 플립플롭의 output을 첫 번째 플립플롭의 입력에 연결하는 Ring Counter와는 달리 맨 마지막 플립플롭의 output의 보수를 첫 번째 플립플롭의 입력에 연결한다. 회로도와 truth table을 그리면 아래와 같다.

Q_A	Q_B	Q_C	Q_D
0	0	0	0
1	0	0	0
1	1	0	0
1	1	1	0
1	1	1	1
0	1	1	1
0	0	1	1
0	0	0	1
repeat			



Johnson Counter를 Verilog 코드로 구현하여 실제로 simulation 해보았다. 해당 코드와 simulation 결과는 아래와 같다. 첨부하지 않은 코드는 Ring Counter와 일치한다.

inv.v

```
`timescale 1ns / 1ps

module inv(clk, reset, dout);

input clk, reset;
output reg [3:0]dout;
wire [3:0] s;

initial
begin
    dout[0] = 1'b0;
    dout[1] = 1'b0;
    dout[2] = 1'b0;
    dout[3] = 1'b0;
end

d_flip_flop u0 (.din(~dout[3]),
.clk(clk),
.reset(reset),
```

```
.dout(s[0]));

d_flip_flop u1 (.din(dout[0]),
.clk(clk),
.reset(reset),
.dout(s[1]));

d_flip_flop u2 (.din(dout[1]),
.clk(clk),
.reset(reset),
.dout(s[2]));

d_flip_flop u3 (.din(dout[2]),
.clk(clk),
.reset(reset),
.dout(s[3]));

always @ (s[0] or s[1] or s[2] or s[3])
begin
    if(reset)
    begin
        dout[0] <= 1'b1;
        dout[1] <= 1'b0;
        dout[2] <= 1'b0;
        dout[3] <= 1'b0;
    end
    else if(~reset)
    begin
        dout[0] <= s[0];
        dout[1] <= s[1];
        dout[2] <= s[2];
        dout[3] <= s[3];
    end
end

endmodule
```