13주차 결과보고서

전공: 컴퓨터공학과 학년: 2학년 학번: 20191619 이름: 이동석

**1. Shift Register**

텍스트이(가) 표시된 사진

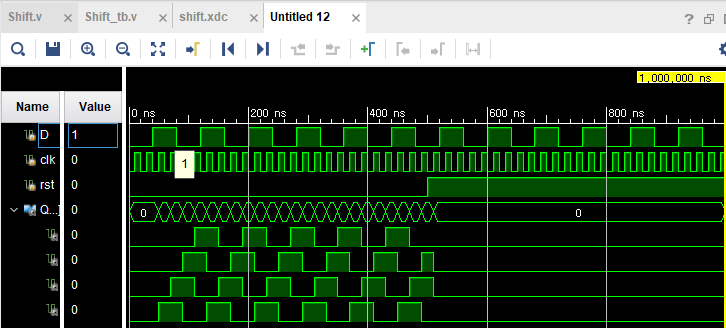
자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명

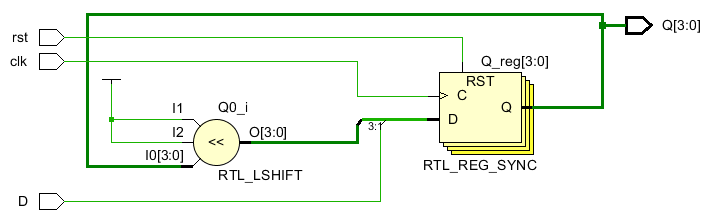
우선, Shift 레지스터는 왼쪽 혹은 오른쪽으로 쉬프트 시그널이 들어왔을 때 옮기는 레지스터이다. Shift 레지스터의 구현은 나름 어려울 것이라 생각한 것과는 달리 간단하게 구현이 가능했다. 이번 실습에서 Shift 시그널은 구현하지 않았다. 구조는 저번 실습과 유사하다. 처음에는 일일이 Q의 4개 변수를 한칸씩 옮겨주도록 코딩하였다. 하지만, 코드를 다시 수정하여 2주차에 배웠던 shift 연산자를 사용해 구현했다. 이번 실습에서 이동시키고자 하는 방향은 상관이 없었기 때문에 좌측 시프트 연산자 <<를 사용했다. Q의 좌측이라고 하면, Q[0]에서 Q[1]으로 Q[1]에서 Q[2]로 Q[2]에서 마지막 Q[3]로 이동시켰다. 또한, 입력을 받기 때문에, Q[0]은 항상 새로들어오는 입력값 D를 할당해주었다. 또한, 저번 실습과 비슷하게 reset을 의미하는 rst변수를 만들어 1이 될 경우 초기 값 0000으로 만들어주었다. 이제 쉬프트 레지스터 논리에 따라 강의자료에 있는 Output table을 다음과 같이 작성할 수 있다.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Shift Register OUTPUT TABLE | | | | | |
| Number Clock Transitions | OUTPUTS | | | | |
| ↑ | IN | L1 | L2 | L3 | L4 |
| 1 | 1 | 0 | 0 | 0 | 0 |
| 2 | 0 | 1 | 0 | 0 | 0 |
| 3 | 1 | 0 | 1 | 0 | 0 |
| 4 | 0 | 1 | 0 | 1 | 0 |
| 5 | 1 | 0 | 1 | 0 | 1 |
| 6 | 0 | 1 | 0 | 1 | 0 |
| 7 | 1 | 0 | 1 | 0 | 1 |

테이블의 값은 임의로 설정하여 채워넣었다. 붉은색으로 색칠된 1을 보면 각 클락이 지날 때 마다 오른쪽으로 쉬프트가 된다.



우선 초기값을 0000으로 했다. 이후, D가1이 되고 clk의 상승에지를 만나는 시점에 Q[0](LSB)가 1이 되는 것을 볼 수 있다. 또한, 바로 다음 clk의 상승에지를 만나게 되면 Q[0]에 있던 1을 Q[1]으로 이동시킴과 동시에 D가 아직 1이므로 Q[0]에 1을 가지는 것을 볼 수 있다. 그 다음 상승에지 때는 D가 1이 아니므로, Q[1]과 Q[0]에 있던 1이 각각 Q[2]와 Q[1]으로 이동한다. 그리고 Q[0]은 D의 값 0을 할당받는다. 같은 방식으로 다음 clk에서 이제, Q[3]과 Q[2]가 1이고 남은 2개의 비트는 0을 가진다. 이제 다음 clk를 만나게 되면 기존 Q[3](MSB)가 가지고 있던 비트는 사라지고 Q[2]를 할당받는다. 그러나, 다시 D가 1이 되므로 Q[3]와 Q[0]이 각각 1의 값을 가지게 된다. 시뮬레이션 결과를 보면, 입력값 D가 clk의 상승에지를 만날 때 마다 Q[0]에 새로운 값이 들어오는 것을 볼 수 있고, 모든 비트들이 Q[0] -> Q[1] -> Q[2] -> Q[3]으로 Shift하는 것을 볼 수 있다. 마지막으로, rst가 1이 되고 상승에지를 만나자 마자 모든 비트들이 초기값 0000으로 돌아가는 것 역시 확인할 수 있다.



**2. Ring counter**

텍스트이(가) 표시된 사진

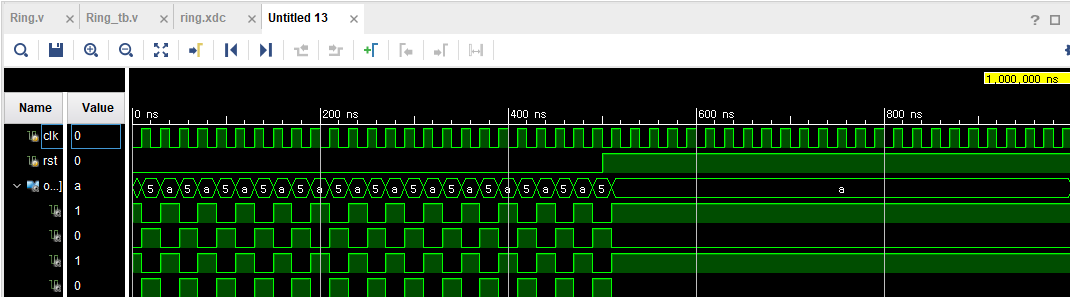
자동 생성된 설명 텍스트이(가) 표시된 사진

자동 생성된 설명

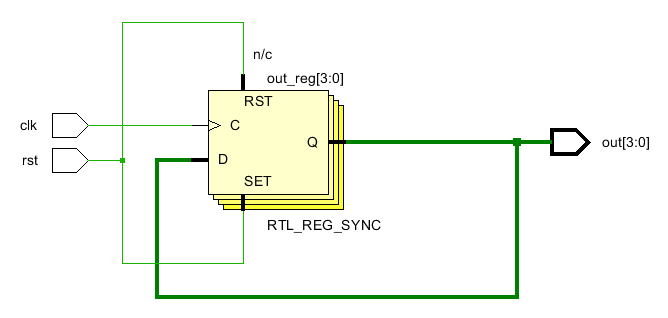
Shift 레지스터에서 마지막 출력이 다시 처음 플리플롭의 입력으로 가게되면 이를 시프트 레지스터 카운터 혹은 링 카운터라 부른다. 마찬가지로 전체적인 구조는 이전과 동일하다. 이번 코드역시 Shift 연산자를 사용해보려 했지만, 오히려 더 헷갈려서 일일이 값을 옮겨주는 방식으로 코드를 작성했다. 새로운 입력값이 주어지지 않고 초기에 할당한 값이 계속 움직이는 형태이므로 이번에는 초기값을 1010으로 설정했다. MSB인 Q[3]의 값을 임시로 저장할 변수 tmp에 할당한 후, 원하는 방향으로 비트를 한칸씩 옮겨준다. 이후 마지막 Q[0]에는 기존 Q[3] 값을 지니고 있는 tmp를 할당한다. 또한, rst도 역시 1이 될 경우 초기 값 1010으로 되돌려 준다. 이제 위 내용을 바탕으로 강의자료의 표를 작성하면 아래와 같다.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Ring Counter OUTPUT TABLE | | | | | |
| Number Clock Transitions | OUTPUTS | | | | |
| ↑ | L1 | L2 | L3 | L4 |
| 1 | 1 | 0 | 1 | 0 |
| 2 | 0 | 1 | 0 | 1 |
| 3 | 1 | 0 | 1 | 0 |
| 4 | 0 | 1 | 0 | 1 |
| 5 | 1 | 0 | 1 | 0 |
| 6 | 0 | 1 | 0 | 1 |
| 7 | 1 | 0 | 1 | 0 |

매 클락 마다 1010이 L1 -> L2 -> L3 -> L4 -> L1 으로 한칸씩 이동한다.



시뮬레이션을 확인해보면 위에서 작성한 테이플과 정확하게 일치하는 모습을 보여준다. 초기 값이 1010이기 때문에 상승에지를 만날 때 마다 2가지 패턴 1010과 이를 한칸 쉬프트한 0101이 반복해서 나타난다. 특히, Shift 레지스터와 다르게 MSB가 LSB가 되는 모습을 볼 수 있다. rst가 1일 경우 초기값 1010이 나타나는 것도 확인할 수 있다.

****

**3. Up/Down counter**

**테이블이(가) 표시된 사진

자동 생성된 설명** **텍스트이(가) 표시된 사진

자동 생성된 설명**

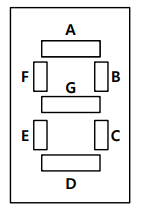
UP/DOWN 카운터는 UP과 DOWN 카운터들을 한번에 수행하는 카운터이다. 이번 실습에서는 시그널이 high이면 UP 카운터의 역할을 수행하고, 시그널이 low일 때 down 카운터를 수행하는 회로를 구현하였다. 또한, 7 Segment 디스플레이를 활용해 화면상으로 소문자 d와 대문자 U를 표시하도록 하였다. 이번에는 0~15까지 모두 사용하므로 오히려 저번 실습보다 간단하였다. 우선, up과 down 카운터를 결정할 시그널로 up을 사용했다. 코드에서는 up과 down 모두 있지만, 실제 실습시 up만을 사용했다. 다음으로는 위와 동일하게 clk와 rst를 입력으로 주었다. 출력으로는 현재 비트를 출력할 out과 디스플레이와 관련된 7비트 변수 dis를 선언했다. 여기서 중요한 점으로 FPGA보드에서 여러 개의 7 segement중 하나를 선택해야 하므로 digit이라는 변수를 선언했다.

이제, out을 초기값 0000으로 할당한다. 또한, digit은 항상 1이 되도록 하였다. 이제 상승에지를 만날 경우 3가지 경우로 나눌 수 있다.

1. rst == 1
2. up == 1
3. up == 0

가장 먼저, rst가 1이 되면 당연하게도 out은 초기값 0000을 준다. 또한 이때 display에 표시되는 것 역시 아무것도 없어야하므로 7개의 비트 모두 0으로 주어야 한다.

rst가 0이고 up이 1이 될 경우 상승에지를 만날 때 마다 out은 1비트씩 더해준 값을 출력해야한다. 또한, 디스플레이에 대문자 U를 출력해야하므로 0111110을 할당해주었다.

이는, 앞선 8장에서 했던 7-segment내용을 알고 있어야 가능한 코드이다. 왼쪽 그림에서 FPGA의 포트가 알파벳의 순서대로 할당된다. U를 만들기 위해서는 A와 G를 제외한 모든 디스플레이의 신호가 High가 되어야하므로, A = 0, B = 1, C = 1, D = 1, E = 1, F = 1, G = 0 이 되어야 하므로 dis 배열에 순서대로 0111110을 할당해주었다.

소문자 d의 경우도 동일하다. d가 되기 위해서는 B, C, D, E, G가 high 신호가 되야한다. 따라서, dis배열에는 0111101이 할당된다.

마지막은 rst가 0이고 up도 0인 경우이다. 이때는 down 카운터의 역할을 하므로, out은 1비트씩 빼준 결과를 출력해야한다. 또한 위에서 말했듯 dis에는 0111101을 할당한다.

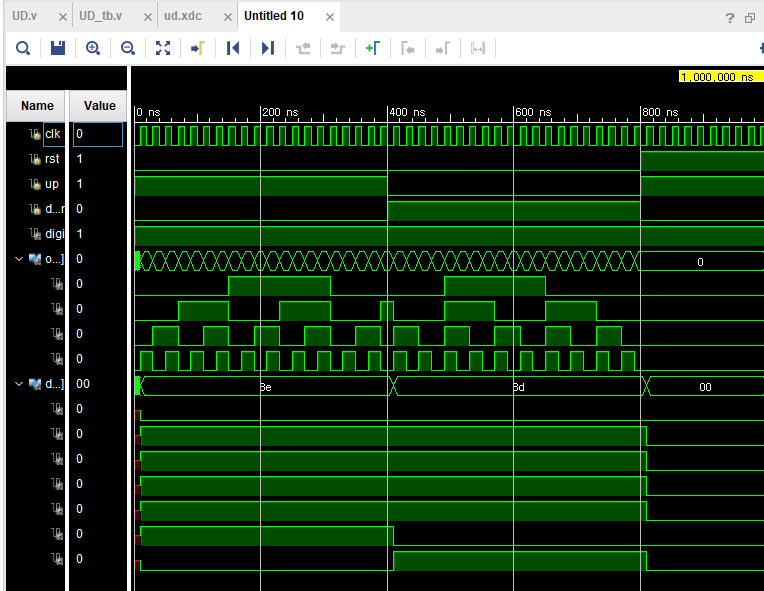
이제 위의 논리를 바탕으로 아래의 2가지 UP과 DOWN카운터의 표를 작성할 수 있다.

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

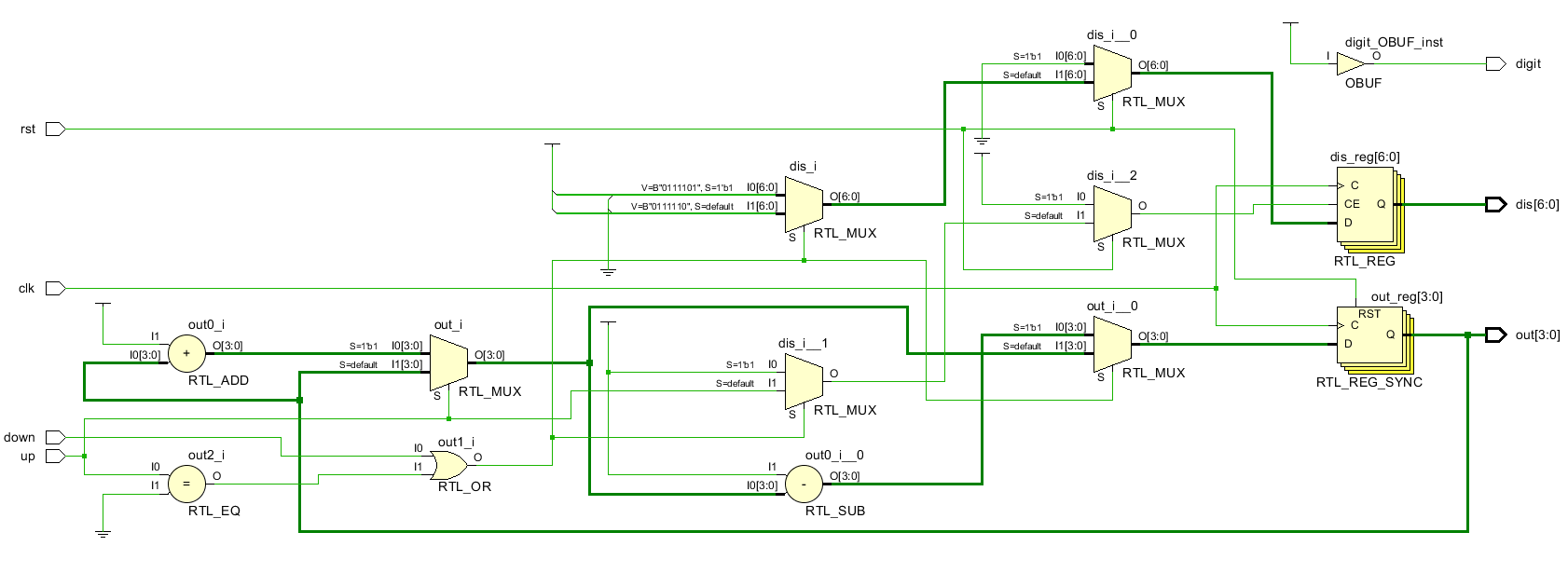
초기 0000에서 클락을 만날 때 마다 1비트씩 증가하며 화면에 U가 표시 된다.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| DOWN Counter OUTPUT TABLE | | | | | |
| Number Clock Transitions | OUTPUTS | | | | |
| ↑ | Display | L1 | L2 | L3 | L4 |
| 1 | d | 1 | 1 | 1 | 1 |
| 2 | d | 1 | 1 | 1 | 0 |
| 3 | d | 1 | 1 | 0 | 1 |
| 4 | d | 1 | 1 | 0 | 0 |
| 5 | d | 1 | 0 | 1 | 1 |
| 6 | d | 1 | 0 | 1 | 0 |
| 7 | d | 1 | 0 | 0 | 1 |

0000에서 1비트가 감소하면 1111이 된다. 화면에 소문자 d가 표시된다.

****

시뮬레이션 결과를 보면, 처음 clock을 만날 때 up이 1이고, rst가 0이므로, 초기값 0000에서 1비트가 올라간 모습을 볼 수 있다. 또한, dis의 값 역시 0111110으로 나타나는 것을 볼 수 있다. 매 상승에지를 만날 때 마다 1비트씩 상승하다가 up이 0이 되고 상승에지를 만났을 때 0100에서 0011로 감소한 것을 볼 수 있다. 마찬가지로, dis 역시 0111101으로 값이 바뀐 것을 볼 수 있다. 이제 계속 감소하다가 0000에서 1비트가 감소하자 다시 1111이 되는 것 역시 볼 수 있다. 이제 rst가 1이 되자, dis와 out 모두 0비트로 바뀌는 것 역시 확인할 수 있다.

****

**4. 결과 검토 및 논의**

쉬프트 레지터와 링 카운터는 비슷한 형태를 띄고 있어 구현하기 매우 간단했다. 처음으로 << 연산자를 사용했는데 코드의 방식이 깔끔하고 간편했다. 실습 도중에는 깨닫지 못했지만, 링 카운터를 쉬프트 연산자를 사용해 구현하면 다음과 같다. (쉬프트 연산자는 자동으로 비어있는 비트에 0을 채워 준다.)

tmp = Q[3];

Q = Q << 1;

Q[0] = tmp;

또한, tmp의 타입을 wire와 reg변수가 조금 헷갈렸지만, 이번 실습으로 조금 더 이해할 수 있게 되었다. wire는 주로 combinational logic을 묘사할 때 사용하며 reg는 combinational 뿐 아니라, sequential 에서도 사용할 수 있다. 때문에 특히, always 문안에서 사용되는 데이터 타입들은 reg로 선언한 타입이어야 한다.

UP/DOWN 카운터의 경우 +/- 연산자를 사용해 손쉽게 구현할 수 있었다. 상황에 맞게 if else문을 이용했다. 처음 FPGA로 실행했을 때 digit 변수를 선언해주지 않아 당황했었다. 8주차 강의자료를 다시 확인해 본 후 기억을 살려 다시 실습을 진행했다. 또한, 실행했을 때 조금 헷갈렸던 점이 있는데 바로 up시그널을 올리자마자 디스플레이가 바뀌지 않아 조금 당황했었다. 하지만, 이는 clk의 상승에지를 아직 만나지 않았기 때문에 바뀌지 않는 것이 옳다는 것을 깨달았다.

**5. 추가 이론 조사**

위에서 reg와 wire에 대해 짤막하게나마 설명했다. 조금 더 자세하게 찾아본 바로는 reg는 값을 저장할 수 있고, wire는 값을 저장하지 못하며 단순히 다른 element와의 연결을 해준다. 그렇기 때문에 assign을 활용한 연속적 할당문으로 구현할 때의 데이터타입으로 활용된다.

반면, reg는 다음 값이 할당되기 전까지 값을 저장한다. (assign이 불가능) 즉, 레지스터나 플리플롭의 역할을 한다. 하지만, 단순하게 레지스터로 생각하면 always문을 활용해 조합논리를 수행할 수 있게 하는 것은 말이 되지 않지만 reg는 wire과 비슷한 역할도 수행하기 때문에 둘 모두 가능하다. 때문에 저장이 필요한 순차회로에서 (플리플롭, 레지스터, 카운터 등) reg를 사용하는 것이다. 예로 앞선 코드에서 always문의 if문을 수행하기 위해서는 out의 이전 값을 계속해서 저장해놓아야 가능한 일이다. (if문을 만족하지 않을 때는 이전 out 값을 그대로 사용)

따라서, 변수의 타입을 사용할 때는 하드웨어 적으로 어떤 기능을 하는 지 이해하고 작성해야한다. 1주차 2주차때 공부했던 내용이 13주차와서야 이해를 할 수 있게 되었다. 이런면에서 C언어와의 차이점이 있다.