12주차 결과보고서

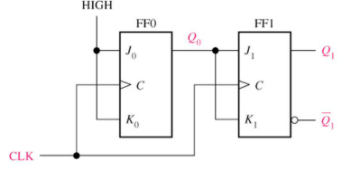
전공: 수학과 학년: 4학년 학번: 20171273 이름: 심현우

1. **2-bit counter의 결과 및 simulation 과정에 대해서 설명하시오. (Verilog source, 출력 예시, 과정 상세히 적을 것)**

2-bit counter는 출력을 반복하는 회로로 2bit를 이용하여 00부터 00, 01, 10, 11의 출력을 주기적으로 나타낸다. input으로 clock과 reset이 있는데, 여기서 clock은 회로의 동작에 관련된 변수로 clock의 pulse가 유지되는 동안 회로가 active하도록 한다. Reset은 출력 값을 reset해주는 역할을 하는 변수로 현재 어떤 값이 저장되어 있든, 그 값을 reset하여 00으로 만들어준다. 이를 표로 나타내면 다음과 같다.

|  |  |  |
| --- | --- | --- |
| **CP(clock pulse)** | **A** | **B** |
| **0** | **0** | **0** |
| **1** | **0** | **1** |
| **2** | **1** | **0** |
| **3** | **1** | **1** |

2-bit counter의 회로 구성은 flip-flop으로 한다. 2bit이므로 2개의 정보를 저장해야 하므로 flip-flop 두개를 연결하고, 동기식, 비동기식으로 구분하여 만들 수 있다. 이 때, 회로는 직렬로 연결하여 첫 flip-flop의 결과가 뒤의 flip-flop의 입력으로 들어가도록 한다. 이때, flip-flop은 원하는 것을 사용할 수 있으며 대체로 JK flip-flop을 사용한다. JK flip-flop으로 동기식 counter를 구성하는 것을 보자. 매 clock pulse마다 B의 값은 toggle된다. 따라서 첫 flip-flop의 JK의 값을 모두 1로 지정하여 첫 flip-flop의 출력 값인 B는 항상 toggle되도록 한다. 이후, B의 값을 뒤의 flip-flop의 input으로 JK모두에 넣는다. 이는 B의 값이 1일 때, A를 toggle하고, B의 값이 0일 때는 A가 no change하기 때문이다. 그림으로 확인하며 아래와 같다.



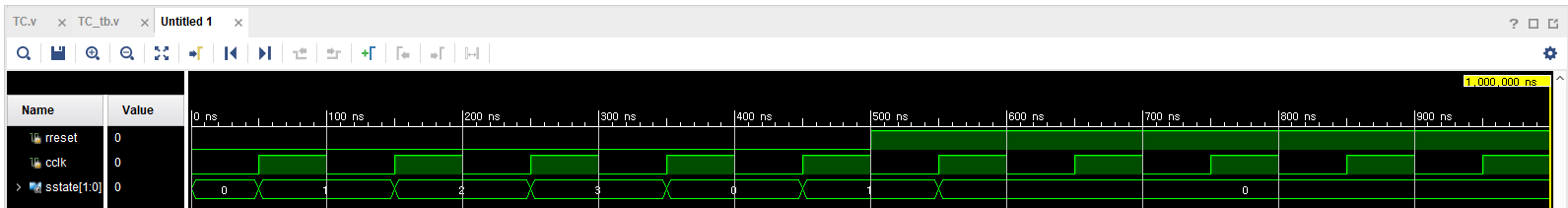
이번 실습에서는 위의 회로 대로 flip-flop을 직접 구성하지 않고, always를 사용했다. 정보를 저장하는 2-bit를 지정하고, if구문을 이용하여 source코드를 작성했다.

텍스트이(가) 표시된 사진

자동 생성된 설명 텍스트이(가) 표시된 사진

자동 생성된 설명

source코드 먼저 확인해보자. input으로는 reset과 clock이 있고, output으로 reg [1:0] state가 있다. 여기서 output의 reg는 변수의 이전 상태를 저장할 수 있게 해준다. [1:0]은 변수의 범위를 설정하도록 한다. 따라서 output은 현재 2-bit의 크기를 가지며 이전 상태를 저장할 수 있도록 설정되었다. 다음 initial은 state의 초기 상태를 설정한다. 00부터 시작하기 위해 0으로 fix했다. Always 구문을 통해 counter의 동작을 정의했다. always@( posedge clk)을 통해 clock의 매 positive edge마다 아래의 코드가 수행되도록 했다. 하나씩 확인해보면 reset이 1일 경우 reset동작을 수행하기 위해, state를 0으로 초기화했다. Reset이 0인 경우, clk이 1이면 counter의 동작인 state의 값을 1증가시켜 순차적으로 출력을 만들고, clk=0인 경우 no change하도록 했다. Simulation을 보면 명확하게 코드의 동작을 확인할 수 있다.

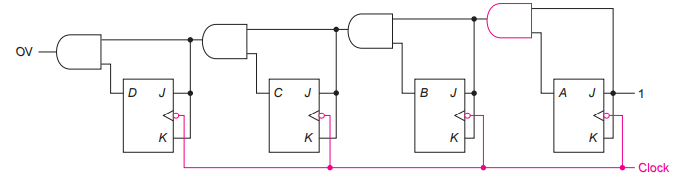


초기 state의 값은 0으로 초기화 되어 있다. Reset 또한 0이다. Clock이 발생함에 따라 reset이 0인 구간에서는 state가 하나씩 증가하여 0부터 3까지 증가한다. 이는 00, 01, 10, 11까지 출력이 나왔다는 것을 알 수 있다. 그 이후에는 다시 00으로 돌아가 이를 순차적으로 반복한다. Reset이 1인 경우 reset의 동작을 수행해 저장된 정보가 0으로 초기화되는 것을 확인할 수 있다. 이를 FPGA에 port하여 동작을 확인했을 때, counter의 기능대로 잘 동작이 되는 것도 확인할 수 있었다.

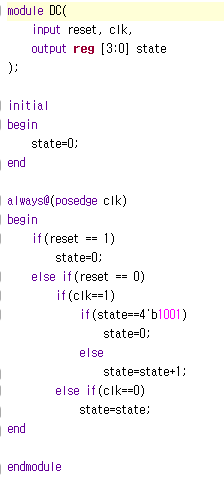
1. **4-bit decade counter의 결과 및 simulation 과정에 대해서 설명하시오. (Verilog source, 출력 예시, 과정 상세히 적을 것)**

4-bit decade counter는 2-bit counter를 확장하여 4-bit 정보를 저장하고 순차적인 출력을 만드는 회로이다. Decade counter이기 때문에 출력은 십진법을 사용한다. 4-bit BCD adder에서 구현했던 것처럼 출력은 0에서 9까지 10가지만 사용한다. 4-bit로는 총 16가지의 출력을 만들 수 있지만 이중에 0000부터 1001까지 10가지만 사용하고 나머지는 출력으로 나오지 않도록 한다. 위의 2-bit counter에서 구성한 것처럼 회로의 구성은 flip-flop을 직렬로 연결한다. 4-bit이기 때문에 4개의 flip-flop을 연결하고, JK또는 D flip-flop을 사용하여 구성할 수 있다. 회로와 표는 아래와 같다.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **CP** | **A** | **B** | **C** | **D** |
| **0** | **0** | **0** | **0** | **0** |
| **1** | **0** | **0** | **0** | **1** |
| **2** | **0** | **0** | **1** | **0** |
| **3** | **0** | **0** | **1** | **1** |
| **4** | **0** | **1** | **0** | **0** |
| **5** | **0** | **1** | **0** | **1** |
| **6** | **0** | **1** | **1** | **0** |
| **7** | **0** | **1** | **1** | **1** |
| **8** | **1** | **0** | **0** | **0** |
| **9** | **1** | **0** | **0** | **1** |

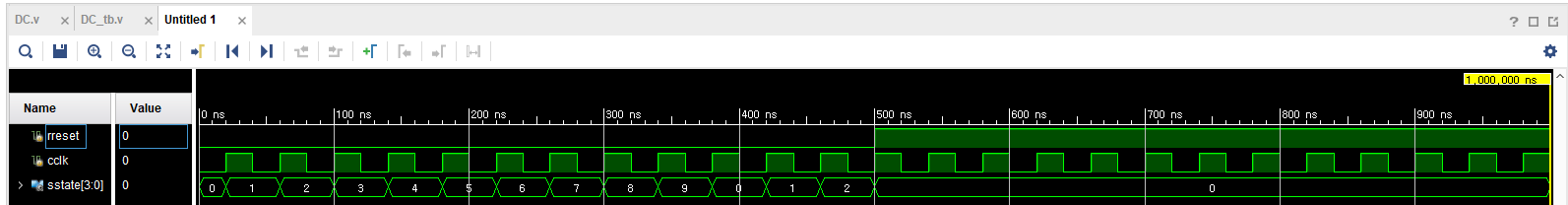


회로는 4-bit counter로 decade counter를 구성하기 위해 reset 변수를 도입하여 1001의 출력 값 이후, reset을 1로 설정하여 저장된 정보를 0000으로 초기화한다. 실습에서 역시 flip-flop을 직접 구현하지 않고 always 구문을 사용하여 동작만 확인했다.

텍스트이(가) 표시된 사진

자동 생성된 설명

source코드의 input, output의 구성은 2-bit counter와 동일하며 output의 범위를 0:3으로 하여 4-bit를 저장하도록 하였다. Initial value를 0으로 초기화하고 always구문을 작성했다. Reset 변수가 1일 때, state를 0으로 초기화한다. Reset이 0인 경우 clock에 따라 clock이 0이고 state가 1001보다 작으면, state를 1씩 증가시켜 counter의 동작을 수행했다. State값이 1001이면 state를 0으로 초기화하여 decade counter가 되도록 설정했다. Clk=0이면 no change를 수행했다. 이를 simulation을 통해 확인해보자.



simulation에서 확인할 수 있듯 reset이 0인 경우 0부터 9까지 state가 증가하는 것을 알 수 있다. 9이후 즉 1001이후 다시 0으로 초기화하여 decade counter가 되는 것도 확인할 수 있다. Reset이 1인 경우 state가 0으로 초기화되는 것도 확인할 수 있다. FPGA와 연결하여 동작이 잘 수행되는 것도 확인할 수 있었다.

1. **4-bit 2421 decade counter의 결과 및 simulation 과정에 대해서 설명하시오. (Verilog source, 출력 예시, 과정 상세히 적을 것)**

2421 decade counter는 8421BCD code를 2421BCD code로 변환한 counter이다. 8421를 2421로 변환한 내용을 표로 작성하면 다음과 같다.

|  |  |  |
| --- | --- | --- |
| **Decimal** | **8421** | **2421** |
| **0** | **0000** | **0000** |
| **1** | **0001** | **0001** |
| **2** | **0010** | **0010** |
| **3** | **0011** | **0011** |
| **4** | **0100** | **0100** |
| **5** | **0101** | **1011** |
| **6** | **0110** | **1100** |
| **7** | **0111** | **1101** |
| **8** | **1000** | **1110** |
| **9** | **1001** | **1111** |

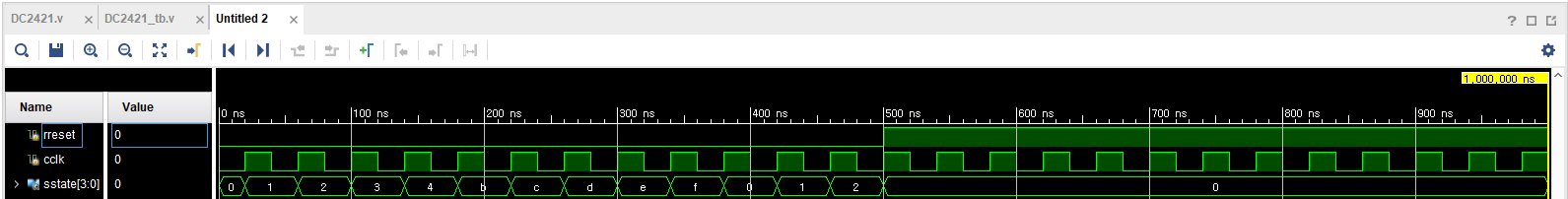
8421코드를 2421로 변환하기 위해서는 decimal값이 5이상일 때, 0110을 더해주면 된다. 코드를 변환하는 과정 이외에 회로 구성이나 코드 작성은 2번의 decade counter와 동일하다.

텍스트이(가) 표시된 사진

자동 생성된 설명텍스트이(가) 표시된 사진

자동 생성된 설명

source코드에서 reset이 0인 경우 clk=1일 때의 코드에서만 차이가 있다. decade에서는 1001까지 표현하고 그 이후는 0으로 초기화했다. 하지만 2421에서는 1000까지 표현되고 그 이후는 1011부터 1111까지 표현된다. 따라서 1000인 경우 7을 더해 1011로 출력한다. simulation으로 결과를 확인하자.



State의 값을 보면 4 이후 5부터 b로 출력되는 것을 확인할 수 있다. 이는 4-bit를 bcd로 변환했을 때, 1011의 값이 b가 출력되는 것을 확인할 수 있다. 0~9와 a~f까지 총 16가지 출력을 나타내는데 2421코드로 변환할 시 0~4까지 b~f까지 출력하여 총 10가지 출력값을 가지는 것을 알 수 있다. 따라서 reset=0일 때, 0부터 시작하여 f까지 잘 출력되는 것을 확인할 수 있고, reset=1일 때 0으로 초기화하는 동작도 잘 수행되는 것을 확인할 수 있다.

1. **결과 검토 및 논의 사항.**

이번 실습에서 직접 flip-flop을 구성하지 않고 always 구문을 이용해 정보의 변화만 출력으로 확인했다. 이전 실습들은 회로를 직접 구성하였는데 반해 이번 실습은 변화만 동작을 정의하여 변화만 확인했기 때문에 코드를 비교적 간단하게 작성했다. 이번 실습에서 reg와 [ : ]을 사용하여 변수를 작성했는데, 이는 변수의 정보 저장과 범위를 나타냈다. [ : ]을 통해 비트의 개수만큼 변수를 설정하고 사용했다. 이후 이를 FPGA의 출력으로 보여주기 위해 state[0], state[1], .. 등 각 비트를 따로 port하여 led로 출력했다. FPGA 작동도 의도한 counter처럼 잘 되었고, 결과도 simulation결과와 동일하게 나왔다.

1. **추가 이론 조사 및 작성.**

이전 실습까지는 assign을 통해 output에 대한 gate와 회로를 구성했다. 이번 실습에서는 always 구문과 if, else를 사용하여 동작을 구현했는데 이와 같이 verilog에서 사용할 수 있는 구문을 보자. 우선 이번에 사용한 always구문은 initial과 함께 쓰인다. Initial은 위에서 설명한 것처럼 초기 상태를 의미한다. Initial은 초기 동작을 제어하고, 그 이후 port변화에 따른 회로 동작은 always 구문을 통해 제어한다. Always@ ()안에 level또는 edge trigger를 통해 어떤 상태에서 always구문을 동작시킬 지 정의하고 동작 또한 정의한다. Level trigger는 그 변수 자체의 0또는 1의 값을 통해 제어하므로 변수 자체를 괄호안에 써서 제어하고, edge trigger의 경우 posedge나 negedge를 통해 제어한다. 그 외 if, else문, case문, 조건연산자, while, for문, repeat문 등을 자유롭게 사용하여 코드를 구성할 수 있다. 이 구문들은 c언어에서 사용하는 구문과 비슷하게 사용되며 시작과 끝을 begin과 end로 묶어준다. Case의 경우 endcase를 마지막에 적어 case문의 끝을 알린다.