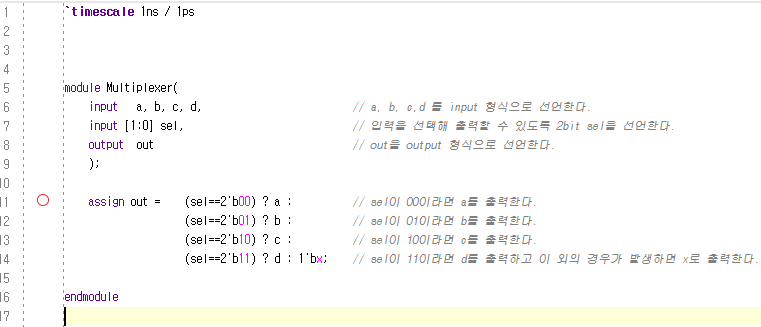
**Lab1.**

# **Homework1. 16to1 multiplexer**

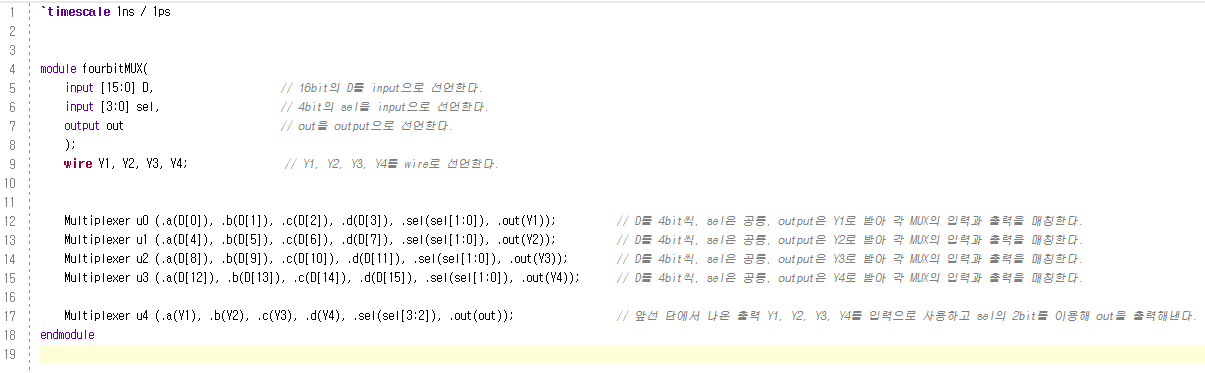
**-4to1 Mux 5개 INSTATIATION**

**Verilog Code / 주석**

Multiplexer

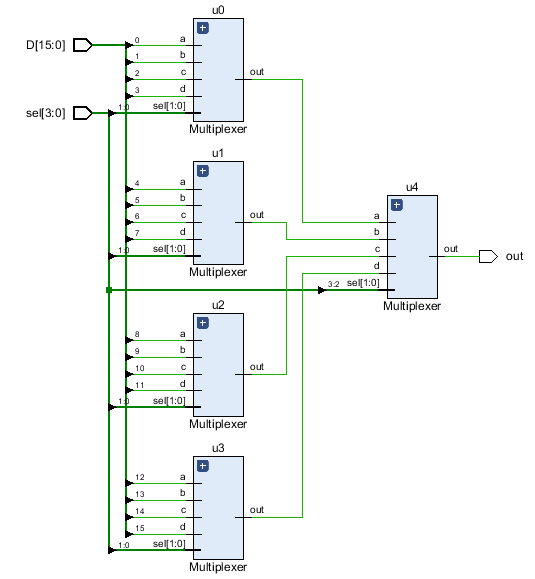


fourbitMUX

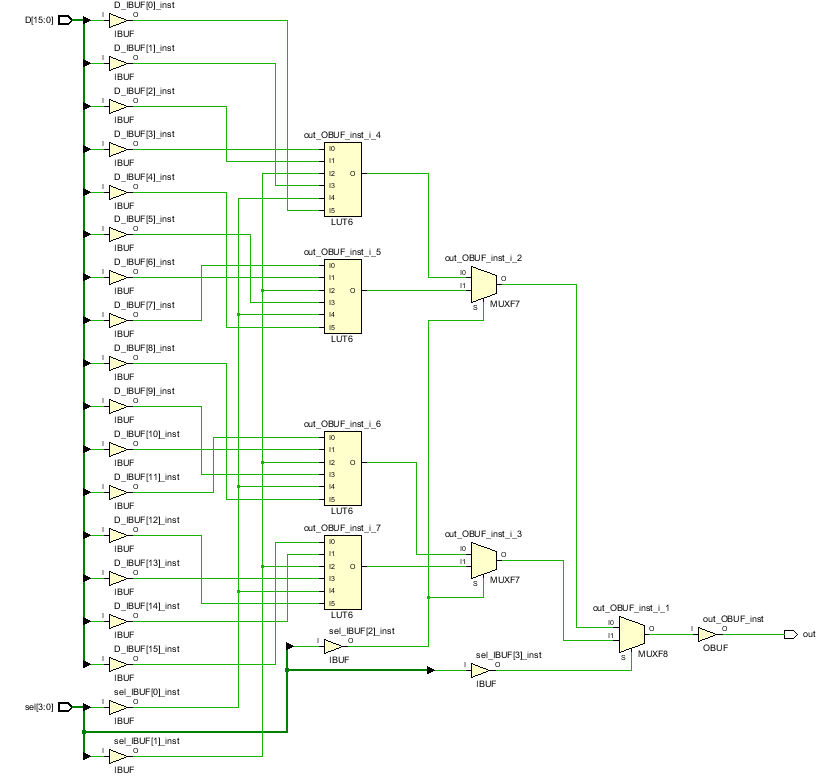


**Synthesis Report**

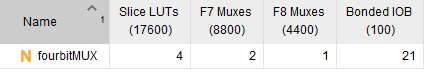
RTL Schematic

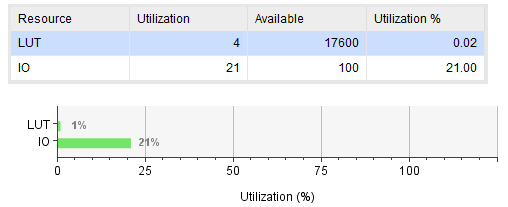


SYNTHESIS Schematic

****

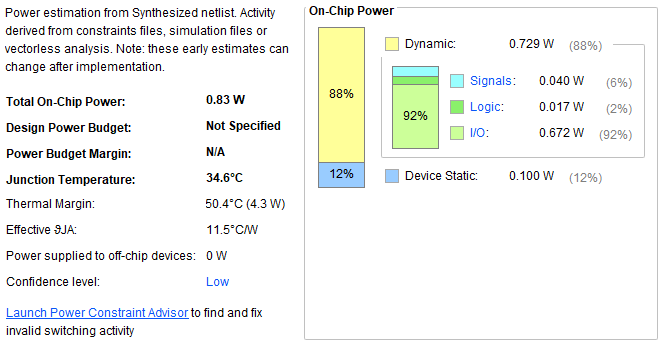
SYNTHESIS Utilization

****

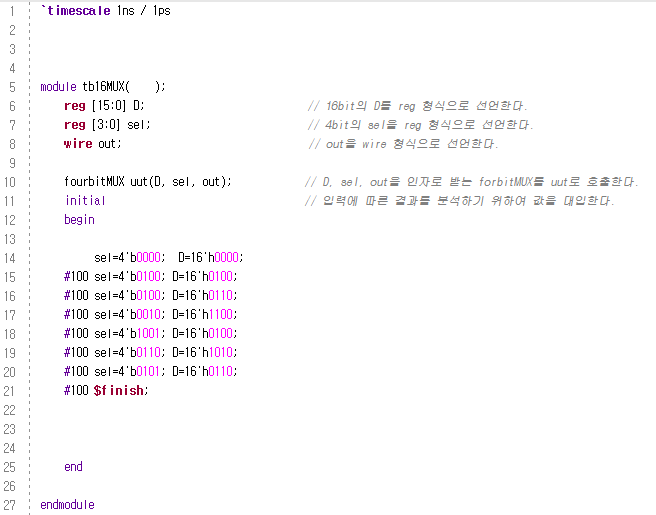
**테이블이(가) 표시된 사진

자동 생성된 설명**

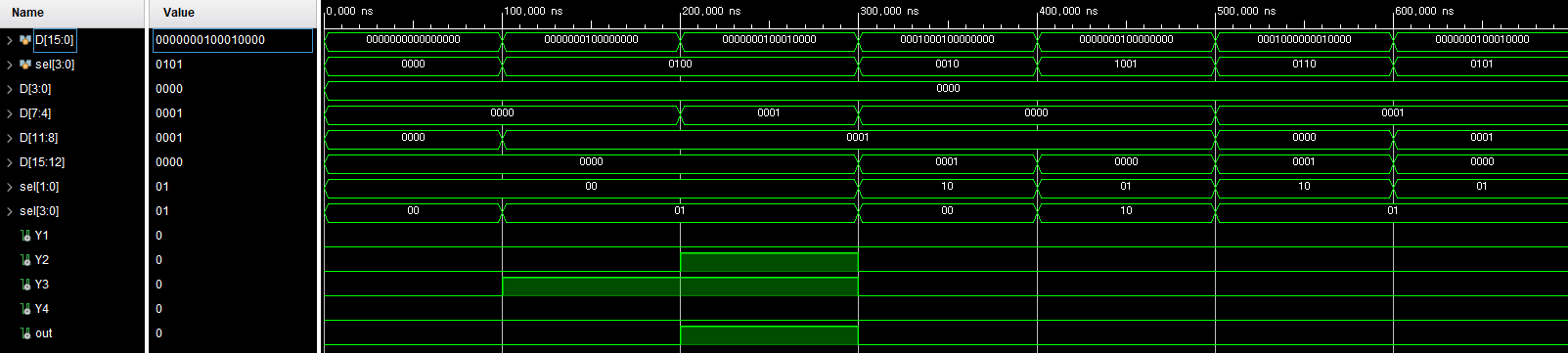
SYNTHESIS Power

****

**Test Bench Code**



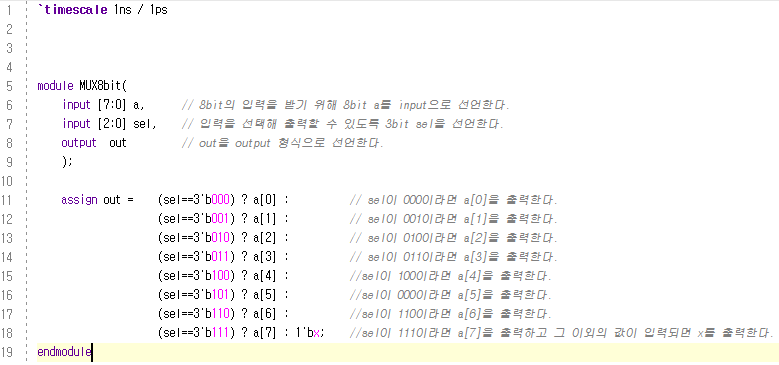
**Simulation Result**



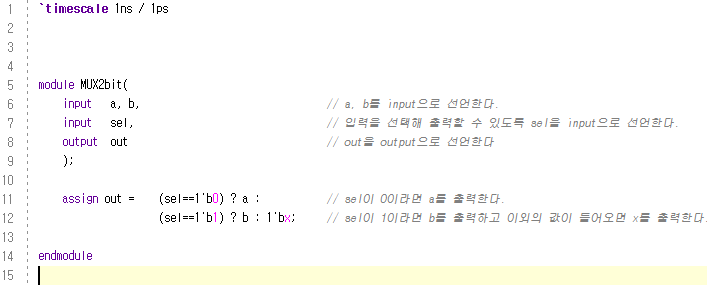
**-8to1 Mux 2개, 2to1 Mux 1개를 Instantiation**

**Verilog Code / 주석**

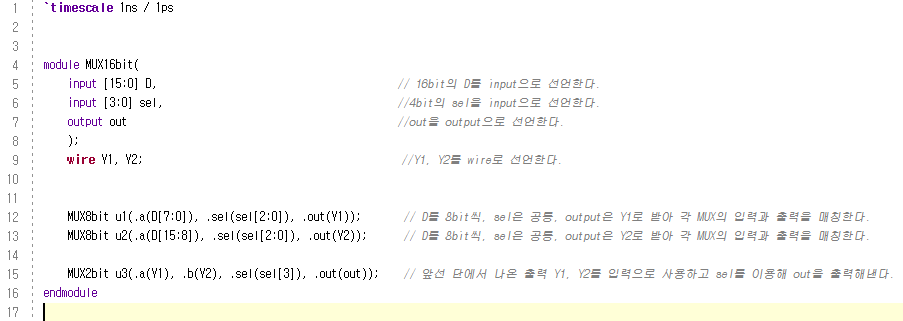
MUX8bit

****

MUX2bit

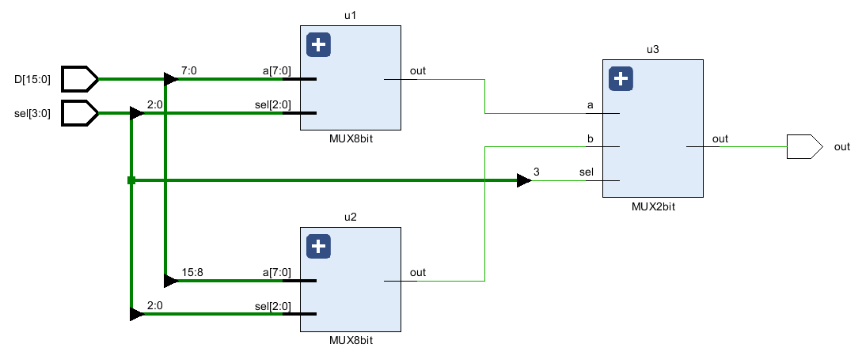
****

MUX16bit

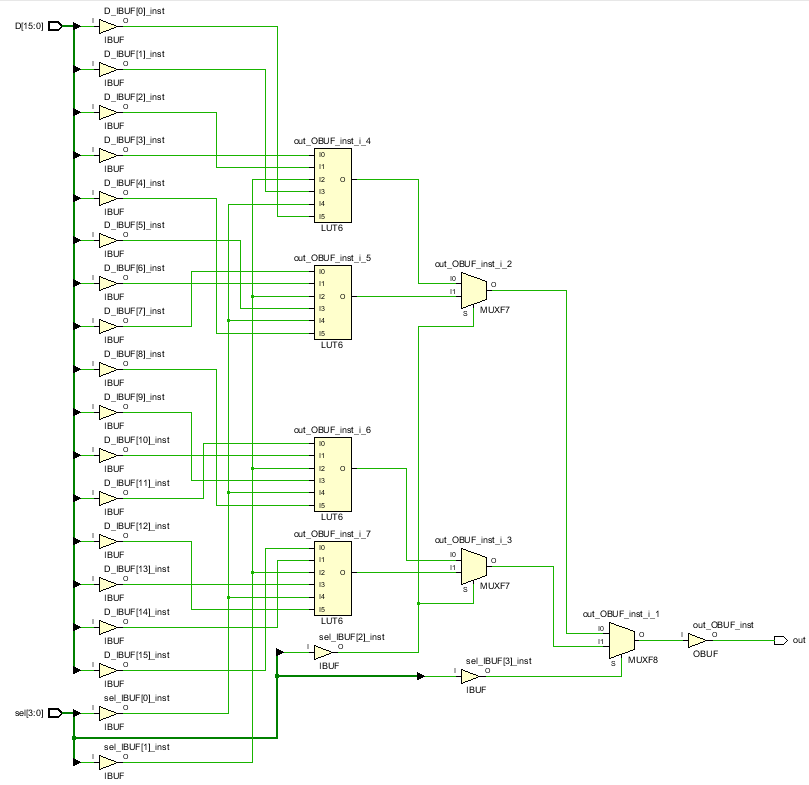
****

**Synthesis Report**

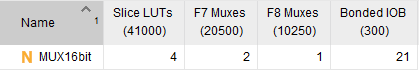
RTL Schematic

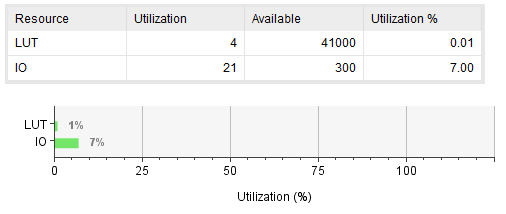
****

SYNTHESIS Schematic

****

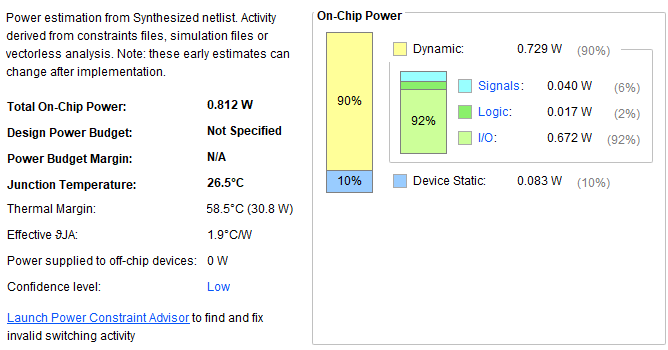
SYNTHESIS Utilization

****

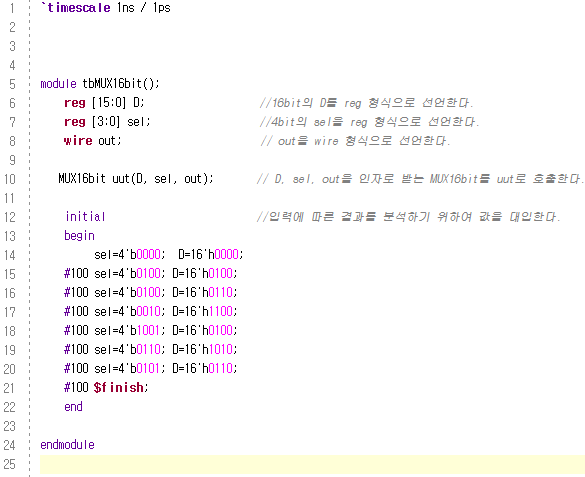
**테이블이(가) 표시된 사진

자동 생성된 설명**

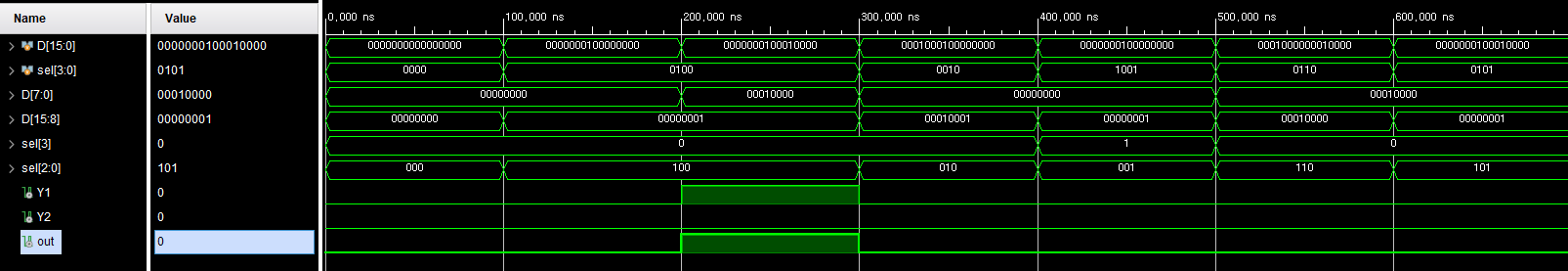
SYNTHESIS Power

****

**Test Bench Code**

****

**Simulation Result**



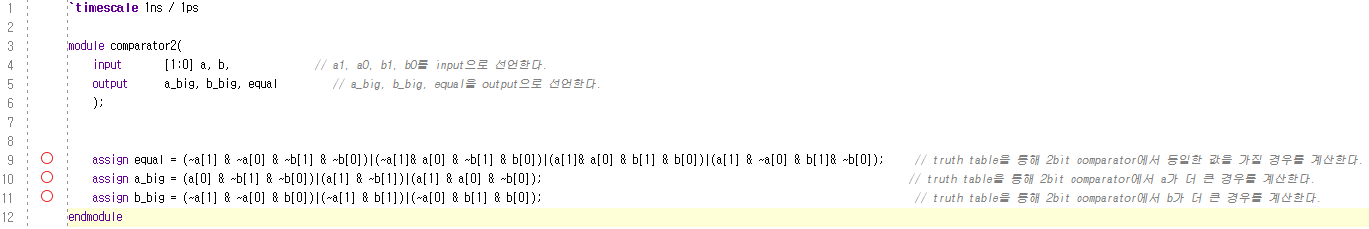
|  |
| --- |
| **Discussion** |
| . - Verilog Coding을 시작하기 전 작성한 Block Diagram    위 그림은 4to1 Mux5개를 Instantiation해서 만들기 전 예상한 Block Diagram이고 오른쪽은 8to1 Mux 2개, 2to1Mux1개를 Instantitation하여 만들기전 예상한 Block Diagram이다.  이 그림은 작성한 코드를 기반으로 하는 RTL Schemetic에서는 동일하게 나타났지만  SYNTHESIS schematic에서는 전혀 다르며 두 케이스가 같은 schematic을 보였다.  SYNTHESIS는 프로그램이 코드를 최적화하기 때문이다.  이 schematic에서는 MUX를 3개 사용하였고 LUT6을 이용하였다.  - 작성한 Verilog Module에 대한 설명  4to1 Mux5개를 Instantiation해서 만든 코드에서는 Multiplexer로 4to1의 Mux를 만들었고 fourbitMUX는 4개의4to1 Mux와 1개의 4to1 Mux를 만들어 서로 이어주는 module이다.  8to1 Mux 2개, 2to1Mux 1개를 Instantitation하여 만든 코드에서는 MUX8bit module에는 8to1 MUX를 만들었고 MUX2bit module에는 2to1 Mux를 만들었다.  MUX16bit module에는 8to1 Mux 2개와 2to1 Mux1개를 연결하여 16to1를 만들었다.  - 작성한 Code가 어떤 동작을 하는지? 왜 그런 동작을 하는지?  16bit의 D를 4개씩 나눠 하나의 MUX에 입력되도록 하였고 2bit의 sel동일하게 4개의 MUX에 입력하였다.  이 4개에서 나온 출력을 다시 4to1 MUX로 연결하고 2bit의 sel값을 이용하여 원하는 값을 선택할 수 있는 MUX를 만들었다.  두번째 코드에서는 16bit의 D를 8개씩 나눠 하나의 MUX에 입력되도록 하였고 8개를 선택할 수 있도록 2비트 3개 즉 sel을 3bit로 설정하였다.  이렇게 나온 2개의 출력을 다시 2to1 MUX로 연결하고 2개중 하나를 선택할 수 있도록1bit의 sel 값을 입력할 수 있는 MUX를 만들었다.  위 두 케이스에서 볼 점은 sel을 4개를 사용하여 16bit의 입력중 하나의 출력을 결정할 수 있다는 점이다.  이는 2^4=16으로 4bit의 sel만 있으면 출력을 선택할 수 있기 때문이다.  첫번째 케이스에서는 2bit의 sel을 4개의 MUX에 공통적으로 입력하였다.  이는 원하는 값이 있지 않는 MUX에서 나온 출력은 다음 단의 MUX에 의해 무시되고 원하는 값 만을 출력할 수 있기 때문에 sel을 모두 하나로 묶어서 사용하여도 되기 때문이다.  두번쨰 케이스도 마찬가지로 8비트의 입력 중 하나를 선택할 수 있도록 3bit의 sel을 이용하여 원하는 출력을 결정하면 다음 단의 MUX를 통해 원하는 값을 출력할 수 있게 된다.  - 작성한 Test Bench Code가 어떤 동작을 하는지?  Test bench code는 16bit의 D와 4bit의 sel, 출력을 볼 수 있는 out을 선언하고 작성한 MUX를 호출하였다.  이 후 제대로 작동하고 있는지 알 수 있도록 값을 대입하는 코드를 작성하였다.  - Simulation 파형이 왜 그렇게 나온것인지?  시뮬레이션 파형은 두 경우 모두 정상적으로 출력되었으며 같은 값을 나타내었다.  Testbench에서 사용한 값과 원하는 출력은 아래와 같다.  sel=4'b0000; D=16'h0000 : D[0]=0  sel=4'b0100; D=16'h0100 : D[4]=0  sel=4'b0100; D=16'h0110 : D[4]=1  sel=4'b0010; D=16'h1100 : D[2]=0  sel=4'b1001; D=16'h0100 : D[9]=0  sel=4'b0110; D=16'h1010 : D[6]=0  sel=4'b0101; D=16'h0110 : D[5]=0  즉 원하는 값과 simulation값이 동일함을 확인할 수 있었다.  하지만 delay면에서 서로 다른 모습을 보였으며 같은 회로에서도 입력과 출력에 따라 다른 delay를 보였다.  먼저 같은 회로에도 다른 delay를 보이는 이유는 입력이 변화하는 속도가 입력마다 다르기 때문이다.  Rising time과 falling time이 다르고 0->0이나 1->1로 유지되는 경우는 시간적으로 다르게 나타난다.  4to1      8to1      위와 같이 회로에 따라 다른 delay를 보이는 이유는 회로가 생성된 위치가 다르기 때문이다.  아래 그림은 delay type의 이름과 위치를 보여준다.  같은 delay type이지만 다른 위치를 가지는 것을 볼 수 있고 이 때문에 delay가 다르게 나타나는 것으로 생각된다. |

# **Homework2. 4-bit magnitude comparator**

**-2-bit comparator를 Instantiation화 하여 설계**

**Verilog Code / 주석**

Comparator2



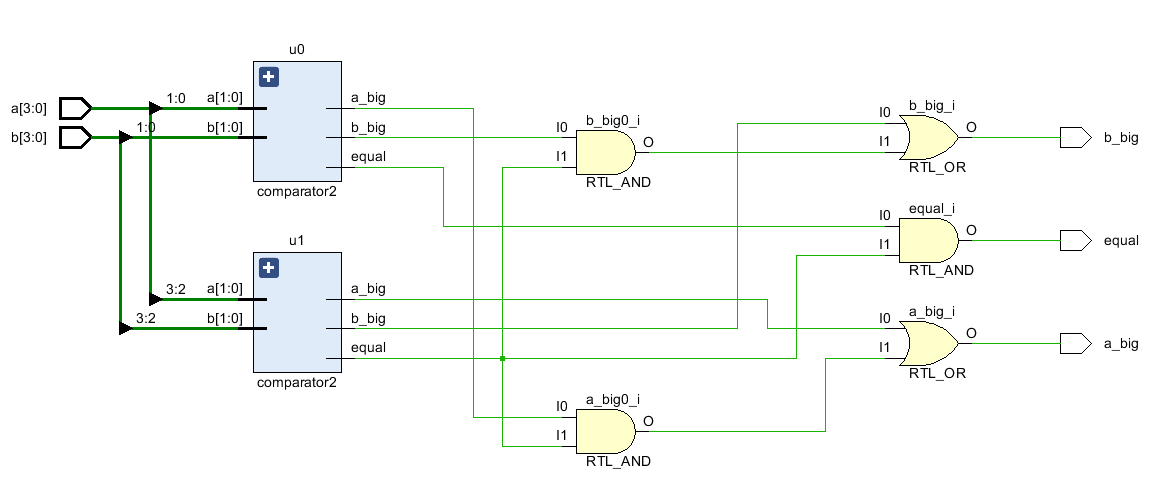
Comparator4bit1

텍스트이(가) 표시된 사진

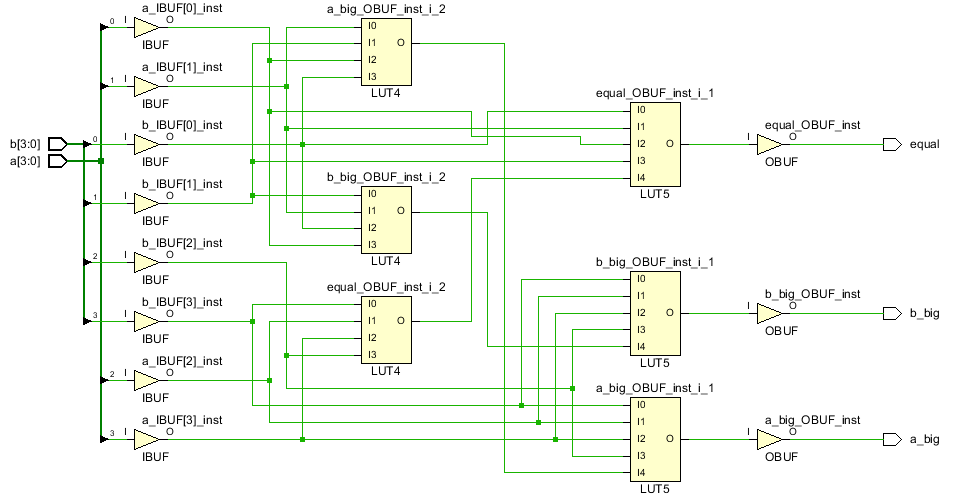
자동 생성된 설명

**Synthesis Report**

RTL Schematic

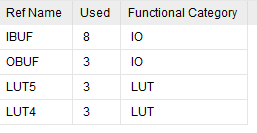
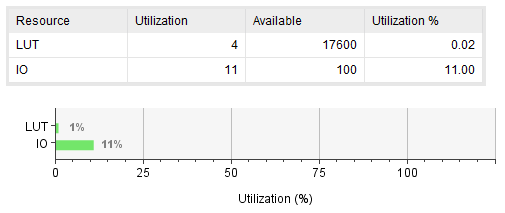
****

SYNTHESIS Schematic

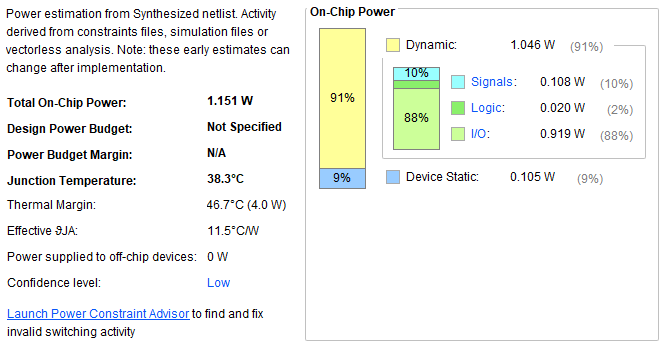
****

SYNTHESIS Utilization

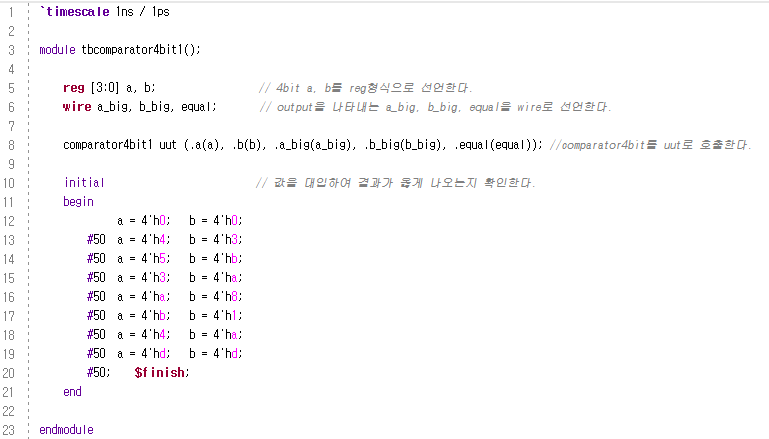
****

****

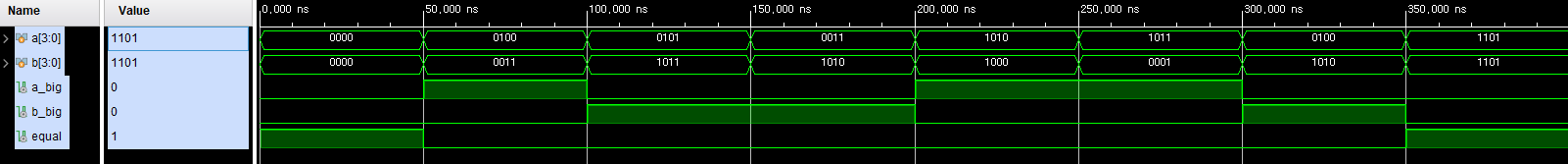
SYNTHESIS Power

****

**Test Bench Code**



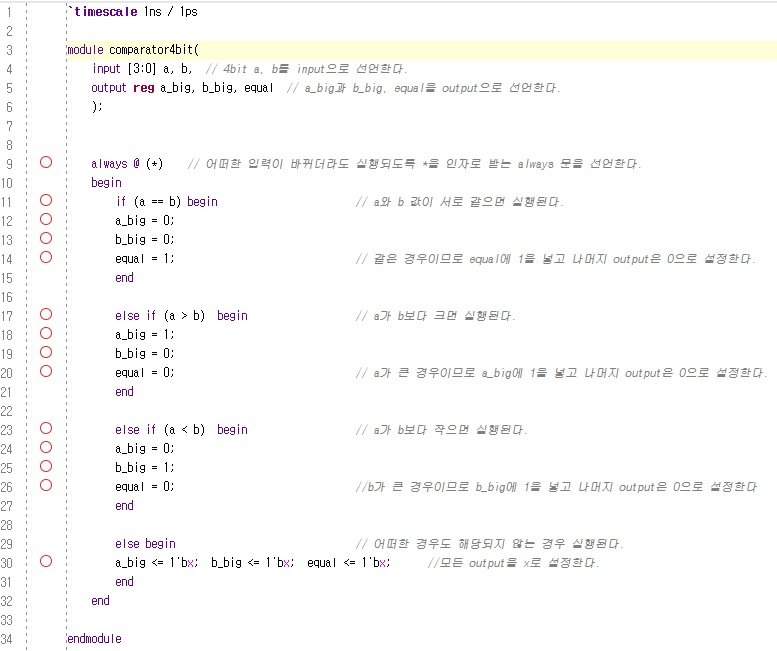
**Simulation Result**



**-2bit comparator를 사용하지 않고 설계**

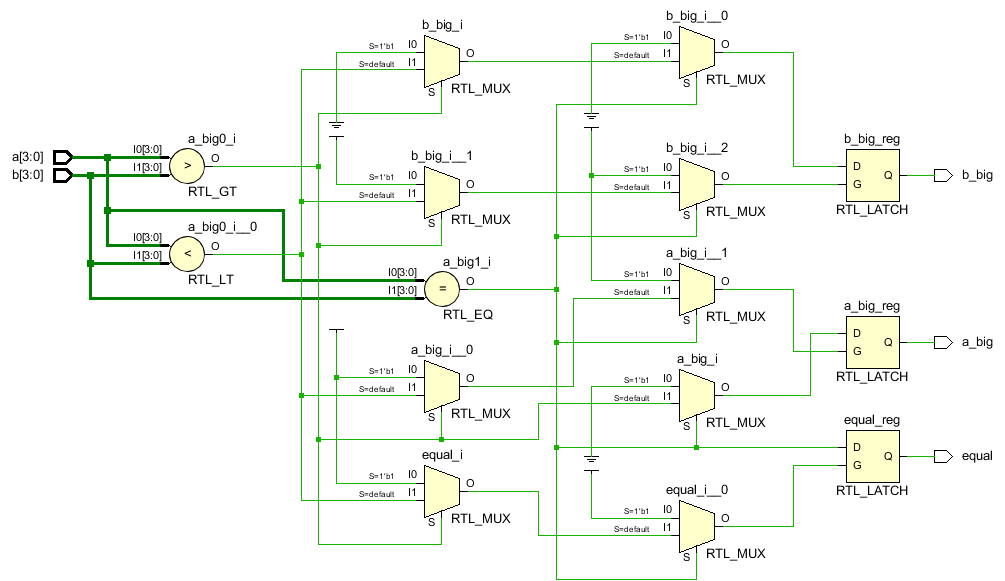
**Verilog Code / 주석**

comparator4bit

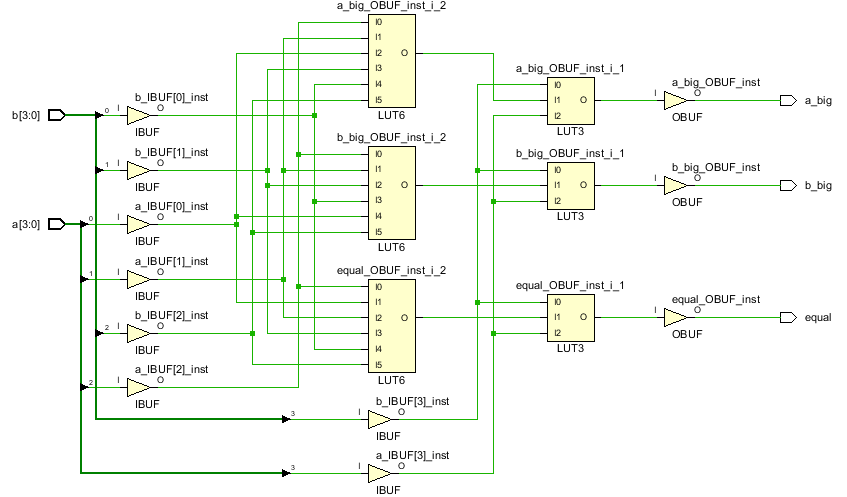


**Synthesis Report**

RTL Schematic

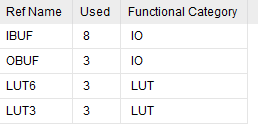
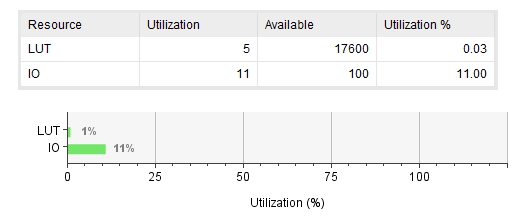


SYNTHESIS Schematic

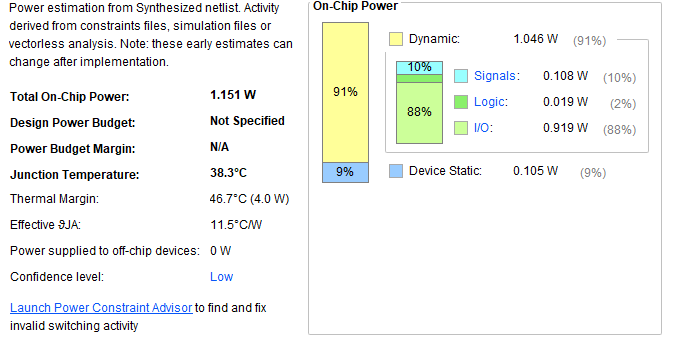


SYNTHESIS Utilization

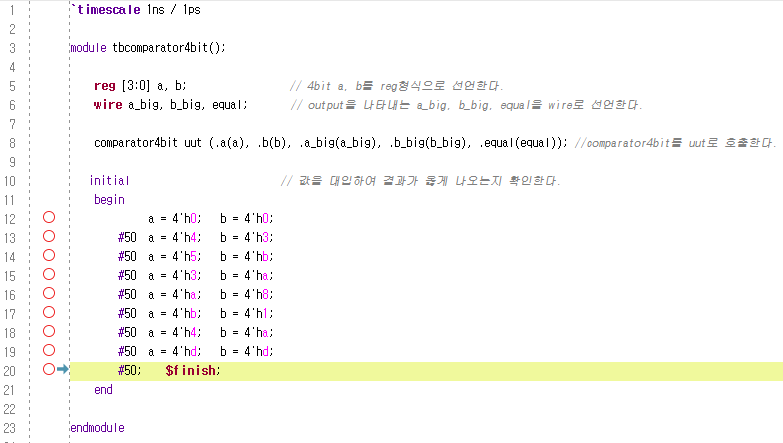




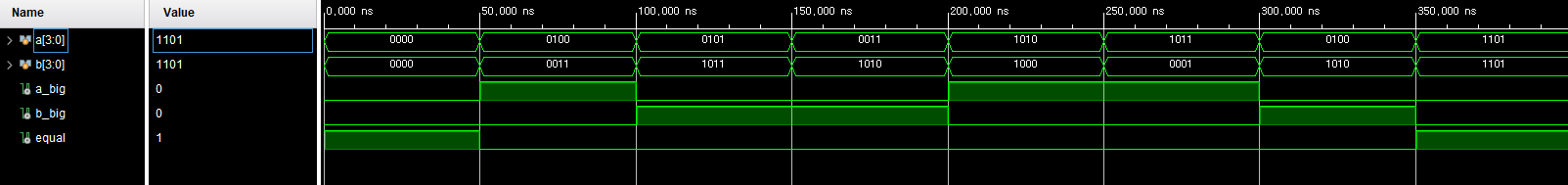
SYNTHESIS Power



**Test Bench Code**



**Simulation Result**



|  |
| --- |
| **Discussion** |
| . - Verilog Coding을 시작하기 전 작성한 Block Diagram    왼쪽 그림은 2bit comparator를 Instantiation으로 엮어서 만든 Block Diagram이고 오른쪽 그림은 4bit를 한번에 비교할 수 있는 comparator이다.  2bit comparator를 Instantiation해서 만든 코드의 schemetic은 위 그림과 동일하게 나왔지만 Instantation을 하지 않고 4bit를 한번에 비교할 수 있는 comparator를 만든 경우에는 다른 schemetic을 보였다.  이는 behavioral Modeling을 사용했기 때문으로 생각된다.  Structure modeling이나 dataflow modeling과 다르게 behavioral modeling은 원하는 작동을 코드로 입력하고 내부적인 회로는 tool에서 알아서 작성하게 된다.  따라서 원하는 회로와 다른 회로가 작성되고 최적화 되지 않은 회로가 나타날 수 있다.  위 경우에 이러한 이유로 다른 회로가 나타난 것으로 생각된다.  - 작성한 Verilog Module에 대한 설명    -a\_big -b\_big  Instantiation으로 설계한 회로에서 comparator2는 2bit comparator를 만들어주었고 comparator4bit1 module은 2bit comparator 2개를 Instantiation하여 4bit의 comparator로 동작하도록 만들었다.  한번에 4bit comparator를 설계한 회로는 comparator4bit module로 구성되어 있다.  이 module에는 always문과 if문을 통해 들어온 입력을 바로 비교할 수 있도록 구성하였다.  - 작성한 Code가 어떤 동작을 하는지? 왜 그런 동작을 하는지?  Instantiation으로 설계한 회로는 먼저 truth table을 분석하여 2bit comparator를 설계하였다.    위 truth table을 통해 아래 식을 얻을 수 있다.  equal = (~a[1] & ~a[0] & ~b[1] & ~b[0])|(~a[1]& a[0] & ~b[1] & b[0])|(a[1]& a[0] & b[1] & b[0])|(a[1] & ~a[0] & b[1]& ~b[0]);  a\_big = (a[0] & ~b[1] & ~b[0])|(a[1] & ~b[1])|(a[1] & a[0] & ~b[0])  b\_big = (~a[1] & a[0] & b[0])|(~a[0] & b[1])|(~a[0] & b[1] & b[0])  이후 comparator를 u0, u1으로 호출하여 2bit씩 계산한 후 비교를 통하여 어떠한 값이 큰지 판별하였다.  이 방법을 통해4bit로 만들었다.    2bit Comparator의 출력을 이용해 작성한 truth table은 위와 같다.  A1은 a\_bigger[1]이라는 뜻이고 A는 최종적으로 A가 B보다 더 크다는 뜻이다.  Equal은 AB의 결과에 따라 정해진다.  위 내용을 토대로 수식을 작성해보면  equal = eq[0] & eq[1];  a\_big = (a\_bigger[1] | (a\_bigger[0] & eq[1]));  b\_big = (b\_bigger[1] | (b\_bigger[0] & eq[1]));  으로 나타낼 수 있다.  Instantiation을 사용하지 않은 회로는 always와 if문을 이용하여 설계하였다.  if문을 통해 입력된 값을 비트로 비교하는 것이 아닌 수와 수의 크기를 비교하도록 하였다.  - 작성한 Test Bench Code가 어떤 동작을 하는지?  Test bench는 설계한 4bit comparator회로에 임의의 입력이 들어가 출력이 나올 수 있도록 설계하였다.  이 과정을 통해 회로가 정상적으로 작동하는지 확인할 수 있었다.  - Simulation 파형이 왜 그렇게 나온것인지?  시뮬레이션에서 output은 정상적으로 출력되었다.  testbench에서 사용한 입력과 원하는 출력은 아래와 같다.  a = 4'h0; b = 4'h0; : equal  a = 4'h4; b = 4'h3; : a\_big  a = 4'h5; b = 4'hb; : b\_big  a = 4'h3; b = 4'ha; : b\_big  a = 4'ha; b = 4'h8; : a\_big  a = 4'hb; b = 4'h1; : a\_big  a = 4'h4; b = 4'ha; : b\_big  a = 4'hd; b = 4'hd; : equal  실제 시뮬레이션 에서도 같은 값을 확인할 수 있었다.      하지만 implementation delay면에서 서로 다른 모습을 보였으며 hazard를 보이는 곳도 있었다.  또한 같은 회로에서도 입력과 출력에 따라 다른 delay를 보였다.  먼저 같은 회로에도 다른 delay를 보이는 이유는 입력이 변화하는 속도가 입력마다 다르기 때문이다.  Rising time과 falling time이 다르고 0->0이나 1->1로 유지되는 경우는 시간적으로 다르게 나타난다.    위와 같이 회로에 따라 다른 delay를 보이는 이유는 회로가 생성된 위치가 다르기 때문이다.  아래 그림은 delay type의 이름과 위치를 보여준다.  같은 delay type이지만 다른 위치를 가지는 것을 볼 수 있고 이 때문에 delay가 다르게 나타나는 것으로 생각된다.  Hazard가 발생하는 이유는 module을 작성할 때 2bit의 결과를 4bit의 결과로 표현하면서 최소한의 항으로 표현하다 보니 이어지지 않은 부분이 발생하였고 이 때문에 hazard가 발생한 것으로 추정된다. |