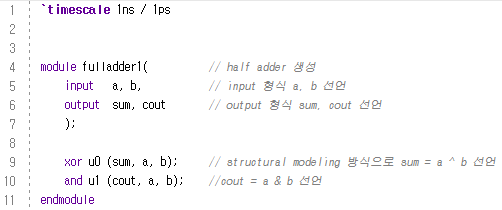
**Lab1.**

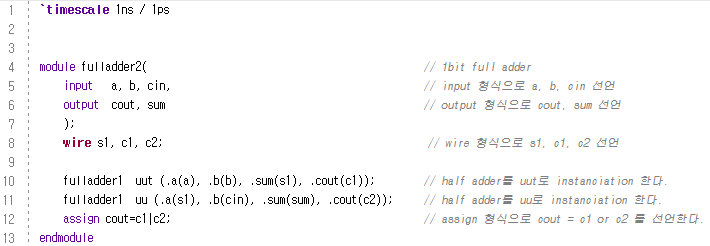
# **Homework1. 16bit ripple carry adder**

**Verilog Code / 주석**

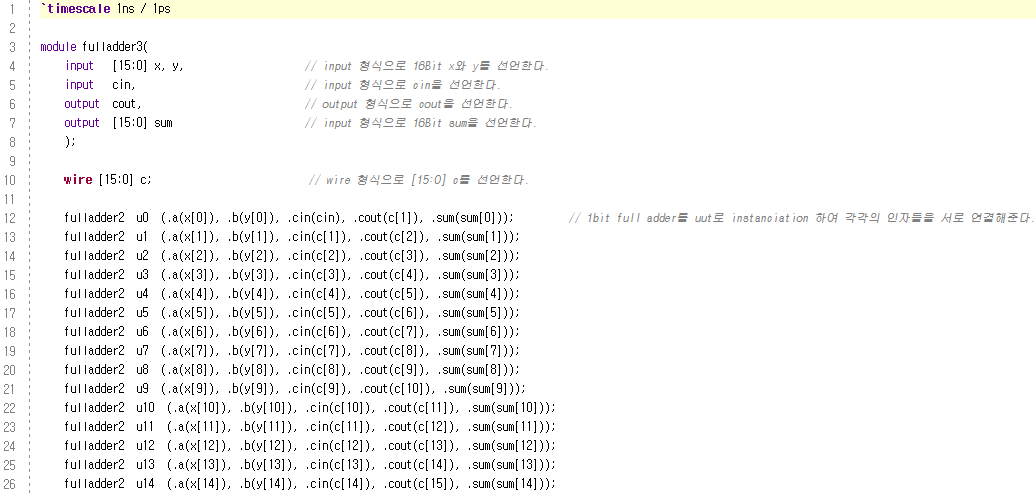
Module 이름이 fulladder1인 Half adder



Module 이름이 fulladder2인 1bit full adder

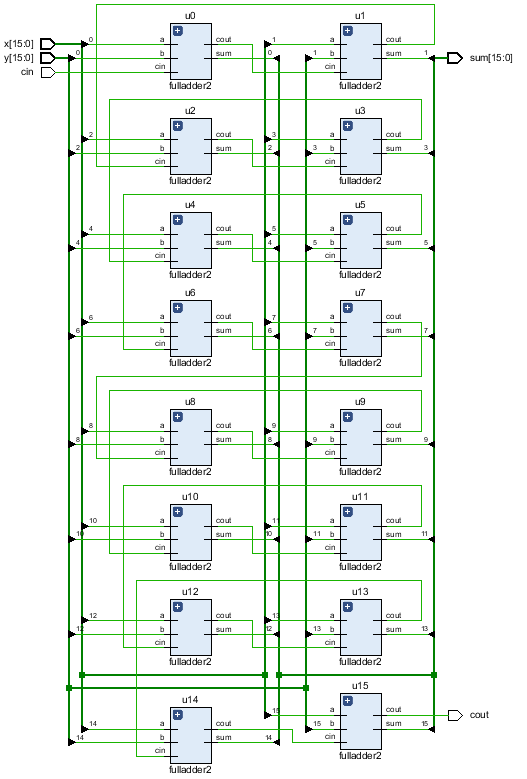


Module 이름이 fulladder3인 16bit full adder

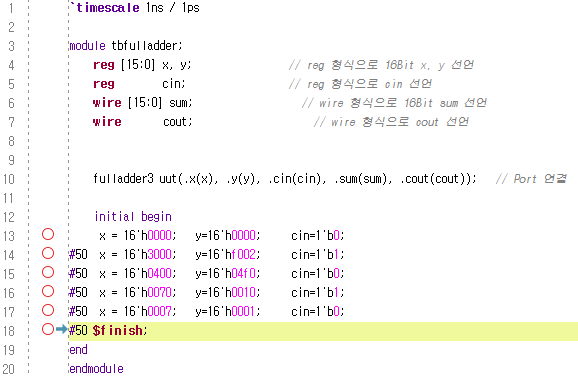




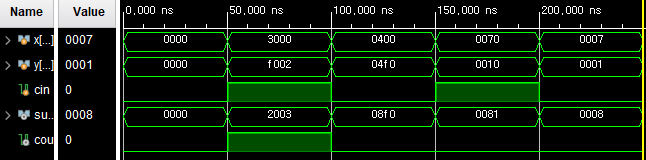
**Synthesis Report**

****

**Test Bench Code**



**Simulation Result**

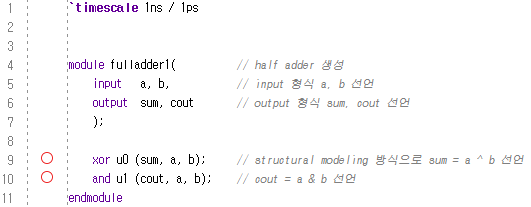


|  |
| --- |
| **Discussion** |
| . - Verilog Coding을 시작하기 전 작성한 Block Diagram    코딩 전 Block Diagram을 작성하였을 때 1bit Full Adder를 16개 직렬로 연결하여 회로를 구현하겠다고 생각했다.  각각의 1bit Full Adder에서x, y와 cin을 입력으로 받고 sum과 carry를 생성해 낸다.  이 carry는 다음 1 bit full adder의 입력으로 사용되고 이 과정이 반복되면서 cout과 sum이 최종적으로 나올 것이라고 생각했다.  실제로 Verilog로 coding하였을 때에도 동일한 모습으로 회로가 나타났다.  - 작성한 Verilog Module에 대한 설명  먼저 half adder를 작성하였다.    half adder의 회로도와 truth table은 위와 같다.  즉 sum=a^b이고 carry=a&b로 표현될 수 있다.  이half adder를 이용하여 1bit full adder를 만들었다.    1bit full adder의 회로와 truth table은 위와 같으며 오른쪽 그림과 같이 HA 2개와 1개의 or gate를 이용하여 구성할 수도 있다.  Verilog Module을 구성할 때 HA를 2개 사용하는 것으로 코드를 작성하였다.  이후 16bit ripple carry adder를 구현하기 위하여 bus를 16 bit로 만들었고 각각의 full adder가 서로 연결될 수 있도록 출력과 입력을 연결해주었다.  - 작성한 Code가 어떤 동작을 하는지? 왜 그런 동작을 하는지?  여러 bit를 사용하는 ripple carry adder는 앞선 full adder에서 나온 carry값을 cin으로 사용한다.  따라서 각각의 1bit full adder를 서로 wiring 해주면 여러 bit를 사용하는 ripple carry adder를 만들 수 있다.  - 작성한 Test Bench Code가 어떤 동작을 하는지?  작성한 test bench는 code에서 구현한 node를 모두 선언해주고 호출한 후 입력인 x, y, cin을 변경해가며 output을 측정할 수 있도록 만들었다.  X와 y는 16bit이므로 hexa 형식을 이용하여 16bit의 hexa type 0000을 입력하는 방법으로 16bit에 원하는 입력이 가해질 수 있도록 하였고 cin은 1bit이므로 1bit의 binary 0을 입력하는 방법으로 원하는 입력이 가해질 수 있도록 하였다.  - Simulation 파형이 왜 그렇게 나온것인지?  파형을 분석해 보면 모두 정상적인 결과를 보이는 것을 확인할 수 있었다.  위 실험에서 사용한 입력x, y, cin과 결과 sum을 순서대로 나열하면 아래와 같다.  000016, 000016, 02, 000016  300016, f00216, 12, 200316  040016, 04f016, 02 , 08f016  007016, 001016, 12, 008116  000716, 000116, 02, 000816  또한 두번째에서는 네번째 bit가 초과되므로 cout이 발생할 것을 예측할 수 있고 실제 simulation에서도 같은 결과를 나타냈다.  즉 정상적인 16bit ripple carry adder로 동작하고 있음을 확인할 수 있었다.  - 작성한 Code가 잘 동작하지 않는다면 어디서 문제가 발생한 것인지?    위 그림은 timing simulation이다.  정상적인 회로의 동작에서는 2003이후 바로 08f0로 결과가 출력되어야 한다.  하지만 위와 같이 중간에 변화가 존재하게 되어 이상한 값이 측정된다.  이는 다수의 bit를 사용하는 full adder와 비동기식으로 동작하는 위 코드의 조합으로 인하여 발생한 것으로 생각된다.  Full adder를 직렬로 연결하여 회로를 완성하였다.  이렇게 작성한 코드의 특성상 앞선 출력으로 발생한 c[x]를 뒷 단에서 입력으로 사용하기 위하여 앞선 회로의 동작을 기다려야 하는 상황이 발생한다.  즉 MSB쪽과 LSB쪽의 결과가 같은 결과를 출력하는 경우가 아닐 수 있게 된다.  이러한 상황에 회로가 비동기식으로 동작하기 때문에 중간에 값이 바뀌게 되면 원하는 결과가 아니더라도 출력을 하게 되고 위 그림과 같은 결과를 보이게 되는 것으로 생각된다.  실제로 많은 LUT가 존재함을 알 수 있고 이 사이에서 위와 같은 경우가 발생할 수 있다.  - 오류를 해결 방법  버퍼와 register를 사용하여 출력이 순차적으로 나오지 않도록 delay시키거나 clock을 사용하여 정보를 읽어오는 timing을 조절하는 방법이 생각된다.  또는 앞선 결과를 통해 추후의 결과를 예측할 수 있는 register를 사용하여 위와 같은 오류의 빈도를 줄일 수 있는 방법이 생각된다. |

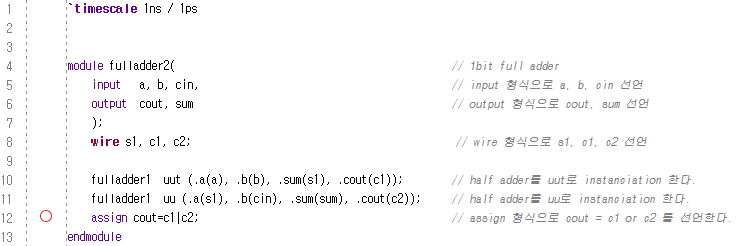
# **Homework2. 4bit adder & substractor**

**Verilog Code / 주석**

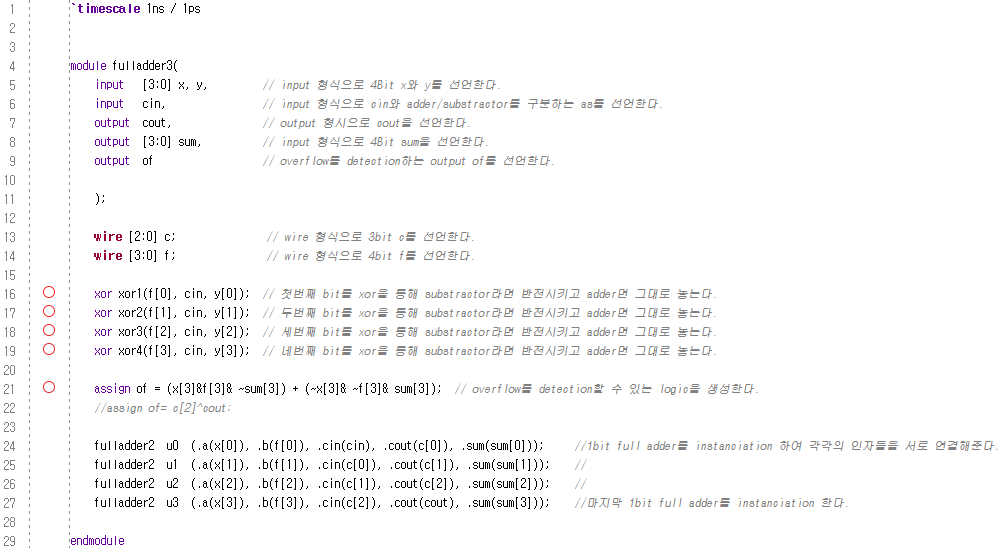
Module 이름이 fulladder1인 half adder



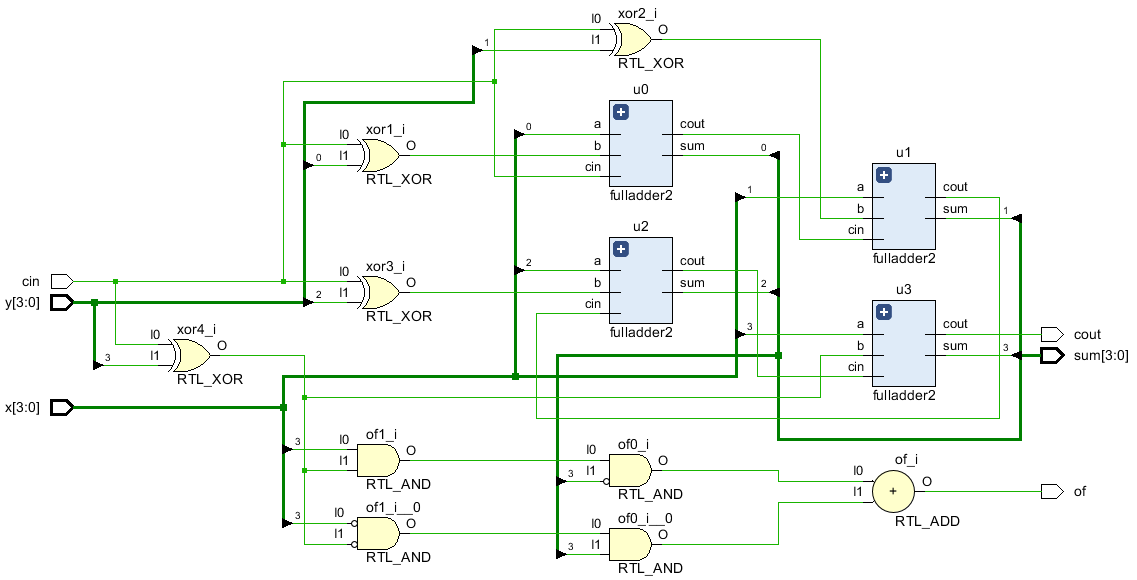
Module 이름이 fulladder2인 1bit full adder



Module 이름이 fulladder3인 4bit full adder/ substractor



**Synthesis Report**

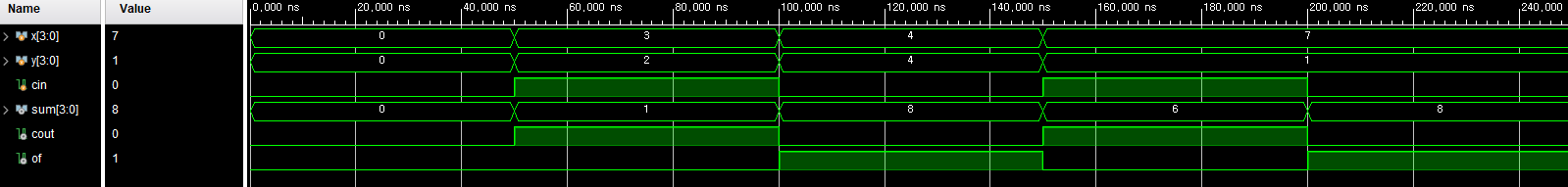
****

**Test Bench Code**

테이블이(가) 표시된 사진

자동 생성된 설명

**Simulation Result**



|  |
| --- |
| **Discussion** |
| . - Verilog Coding을 시작하기 전 작성한 Block Diagram    먼저 4개의 1bit full adder를연결하여 4bit adder를 구현하였다.  그 후 substractor와 adder를 판별하기 위하여 cin을 판별을 위한 기준으로 사용하였다.  먼저 cin 1일 때 substractor로 동작하고 0일 때 adder로 동작하도록 block diagram을 작성하였다.  Substractor를 동작시키기 위해서는 입력을 2의 보수로 변환해주는 과정이 필요하다.  2의 보수는 모든 bit를 반전시키고 최하단 bit에 +1을 해주는 것이다.  따라서 위 조건을 모두 만족시키는 방법은 xor을 사용해 cin과 비교하는 것이다.    위 사진은 xor의 truth table이다.  Cin을 B라고 한다면 cin이 0인 경우에는 부호를 반전하지 않고 그대로 출력하여 adder로 동작할 수 있도록 하고 cin이 1인 substractor에서는 A의 모든 bit의 부호가 반대로 출력되게 되게 한다.  이후 full adder의 동작에서 cin을 더해주게 되므로 2의 보수로 변환된다.  다음으로 overflow를 감지하는 회로를 of=x[3]&f[3]&sum[3]’+x[3]’f[3]’sum[3]으로 설계하였다.  Over flow는 양수 + 양수 혹은 음수 + 음수 중 일부의 경우에서 제한된 bit 수를 초과하여 원하지 않는 결과를 나타내는 경우를 말한다.  즉 x와 y가 양수인 경우 sum이 음수를 나타내는 1이거나 x와 y가 음수인 경우 sum이 양수를 나타내는 0인 경우이다.  이 내용을 truth table로 정리하면 아래와 같다.    Overflow를 감지하는 두번째 방법은 3번째 1bit full adder에서 발생한 carry와 cout을 xor하는 방법이다.  4bit의 adder에서 최상단의 bit는 부호를 나타내는 bit이므로 2의 보수법에 따르면 4bit에서 표현할 수 있는 수의 범위는 7~-8이다.  즉 이 수를 벗어나는 4bit의 sum이 발생하게 된다면 overflow로 판별할 수 있다.  이 수를 벗어나는 경우를 생각해보면 두가지로 나눌 수 있다.  첫번째 경우는 양수와 양수의 합이 8이상이 되는 경우이다. (1xxx인 경우 8 이상이라고 했을 때)  0~7의 양수는 네번째 bit가 0이므로 cout은 발생하지 않는다.  overflow인 경우에 네번째 bit가 1이 되야 한다.  네번쨰 bit가 1이 되기 위해서는 c[3]이 발생하여 부호 bit를 1로 채워야 한다.  두번째 경우는 음수와 음수의 합이 양수가 되는 것이다.  0~-8의 음수는 네번쨰 bit가 1이므로 cout이 항상 발생하게 된다.  Overflow인 경우 네번째 bit가 0으로 양수가 되어야 한다.  이를 만족할 수 있는 방법은 c[3]가 발생하지 않아 부호 bit가 0이 되는 경우이다.  위 경우를 모두 정리하면 cout과 c[3]이 같은 값을 가지지 않는 경우 즉 cout^c[3]=1인 경우에 overflow가 발생한다.  예외의 경우를 생각해보면 양수+양수에서 cout은 항상 발생하지 않고 overflow가 아니라면 c[3]가 발생하지 않아 결과값이 항상 양수가 되도록 하므로 양수+양수에서는 예외의 경우가 없다.  음수+음수에서 cout은 항상 발생하고 overflow가 아니라면 c[3]는 항상 발생해 결과값이 항상 음수가 되도록 하므로 음수+음수의 경우에서 예외의 경우는 없다.  이번 실습에서는 overflow를 감지하는 방법 중 좀더 직관적인 첫번째 방법을 사용하였다.  - 작성한 Verilog Module에 대한 설명  위에서 작성한 diagram을 바탕으로 Verilog Module을 구성하였다.  HA, 1bit full adder를 작성하였고 새로운 f를 만들어 cin의 입력에 따라 y와 비교하여 adder와 substractor 중 원하는 동작을 할 수 있도록 하였다.  또한 full adder를 모두 wiring하여 앞 단의 출력을 입력으로 사용할 수 있도록 하였다.  또한 overflow을 detection할 수 있도록 회로를 추가하였다.  - 작성한 Code가 어떤 동작을 하는지? 왜 그런 동작을 하는지?  xor문은 각 bit가 반전되도록 하여 cin 더해져 2의 보수가 될 f를 만드는 과정이다.  Cin이 0이라면 xor문은 y의 값을 그대로 f에 넣으므로 adder로 동작할 수 있도록 한다.  또한 assign of = (x[3]&f[3]& ~sum[3]) + (~x[3]& ~f[3]& sum[3]);을 추가하여 overflow를 detection할 수 있도록 하였다.  - 작성한 Test Bench Code가 어떤 동작을 하는지?  작성한 test bench는 code에서 구현한 node를 모두 선언해주고 호출한 후 입력인 x, y, cin을 변경해가며 output을 측정할 수 있도록 만들었다.  X와 y는 4bit이므로 decimal 형식을 이용하여 4bit의 decimal type 0을 입력하는 방법으로 4bit에 원하는 입력이 가해질 수 있도록 하였고 cin은 1bit이므로 1bit의 binary 0을 입력하는 방법으로 원하는 입력이 가해질 수 있도록 하였다.  또한 #50을 통하여 50ns의 delay가 생성될 수 있도록 하였다.  - Simulation 파형이 왜 그렇게 나온것인지?  x = 4'd0; y=4'd0; cin=1'b0;  #50 x = 4'd3; y=4'd2; cin=1'b1;  #50 x = 4'd4; y=4'd4; cin=1'b0;  #50 x = 4'd7; y=4'd1; cin=1'b1;  #50 x = 4'd7; y=4'd1; cin=1'b0;  Test bench에서 사용한 코드는 위와 같다.  첫 줄은 모든 입력이 0이므로 sum에 0이 출력된다.  두번째 줄은 3과 2의 뺄셈이므로 sum에 1이 출력된다.  세번째 줄은 4와 4의 덧셈으로 sum에 0이 출력되고 overflow가 발생하는 것으로 판별된다.  네번째 줄은 7 과 1의 뺄셈으로 sum에 6이 출력된다.  다섯번째 줄은 7과 1의 덧셈으로 sum에 0이 출력되고 overflow가 발생하는 것으로 판별된다.  위 simulation에서도 동일한 결과를 얻을 수 있었다.  - 작성한 Code가 잘 동작하지 않는다면 어디서 문제가 발생한 것인지?    위 그림은 timing simulation이다.  정상적인 회로의 동작에서는 0이후 바로 1로 결과가 출력되어야 한다.  하지만 위와 같이 중간에 변화가 존재하게 되어 이상한 값이 측정된다.  이는 다수의 bit를 사용하는 full adder와 비동기식으로 동작하는 위 코드의 조합으로 인하여 발생한 것으로 생각된다.  Full adder를 직렬로 연결하여 회로를 완성하였다.  이렇게 작성한 코드의 특성상 앞선 출력으로 발생한 c[x]를 뒷 단에서 입력으로 사용하기 위하여 앞선 회로의 동작을 기다려야 하는 상황이 발생한다.  즉 MSB쪽과 LSB쪽의 결과가 같은 결과를 출력하는 경우가 아닐 수 있게 된다.  이러한 상황에 회로가 비동기식으로 동작하기 때문에 중간에 값이 바뀌게 되면 원하는 결과가 아니더라도 출력을 하게 되고 위 그림과 같은 결과를 보이게 되는 것으로 생각된다.    실제로 많은 LUT가 존재함을 알 수 있고 이 사이에서 위와 같은 경우가 발생할 수 있다.  - 오류를 해결방법  버퍼와 register를 사용하여 출력이 순차적으로 나오지 않도록 delay시키거나 clock을 사용하여 정보를 읽어오는 timing을 조절하는 방법이 생각된다.  또는 앞선 결과를 통해 추후의 결과를 예측할 수 있는 register를 사용하여 위와 같은 오류의 빈도를 줄일 수 있는 방법이 생각된다.  Overflow를 계산하는 방법에서 nand와 nor gate를 사용하는 방법이 있다.  And gate는 delay가 길다.  다음 방법으로는 c[3]와 cin을 xor으로 비교하는 방법을 사용하는 것이다.  이 방법을 사용하면 gate level이 감소하여 delay가 감소할 것이라고 생각한다. |