7주차 결과보고서

전공: 컴퓨터공학과 학년: 4학년 학번: 20191583 이름: 김태곤

**1. Even Parity bit generator 및 checker 의 simulation 결과 및 과정에 대해서 설명하시오. (Truth table 작성 및 k-map 포함)**

**- Even Parity bit generator**

**1) Truth Table**

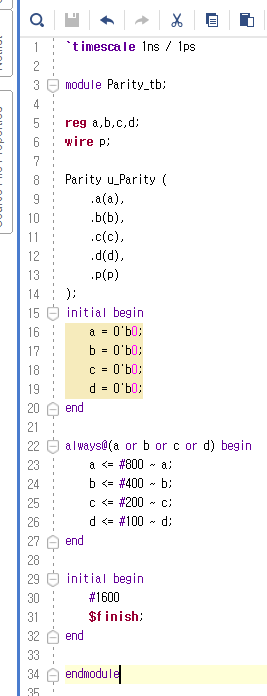
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Input** | | | | **Output** |
| **a** | **b** | **c** | **d** | **p** |
| **0** | **0** | **0** | **0** | **0** |
| **0** | **0** | **0** | **1** | **1** |
| **0** | **0** | **1** | **0** | **1** |
| **0** | **0** | **1** | **1** | **0** |
| **0** | **1** | **0** | **0** | **1** |
| **0** | **1** | **0** | **1** | **0** |
| **0** | **1** | **1** | **0** | **0** |
| **0** | **1** | **1** | **1** | **1** |
| **1** | **0** | **0** | **0** | **1** |
| **1** | **0** | **0** | **1** | **0** |
| **1** | **0** | **1** | **0** | **0** |
| **1** | **0** | **1** | **1** | **1** |
| **1** | **1** | **0** | **0** | **0** |
| **1** | **1** | **0** | **1** | **1** |
| **1** | **1** | **1** | **0** | **1** |
| **1** | **1** | **1** | **1** | **0** |

**2) K-map**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ab\cd** | **00** | **01** | **11** | **10** |
| **00** | **0** | **1** | **0** | **1** |
| **01** | **1** | **0** | **1** | **0** |
| **11** | **0** | **1** | **0** | **1** |
| **10** | **1** | **0** | **1** | **0** |

p = a’b’c’d+a’b’cd’+a’bc’d’+a’bcd+ab’c’d’+ab’cd+abc’d’+abcd’

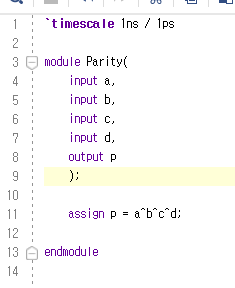
= a’b’(c’d+cd’)+a’b(c’d’+cd)+ab’(c’d’+cd)+ab(c’d+cd’)

****= (ab+a’b’)(c’d+cd’)+(ab’+a’b)(cd+c’d’)

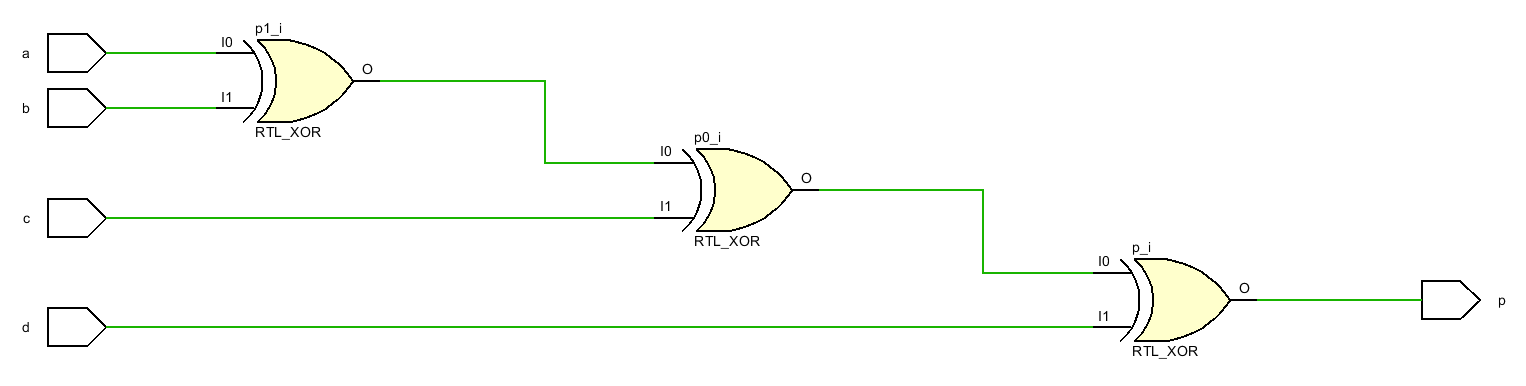
= (a⊕b)’(c⊕d)+(a⊕b)(c⊕d)’

= a⊕b⊕c⊕d

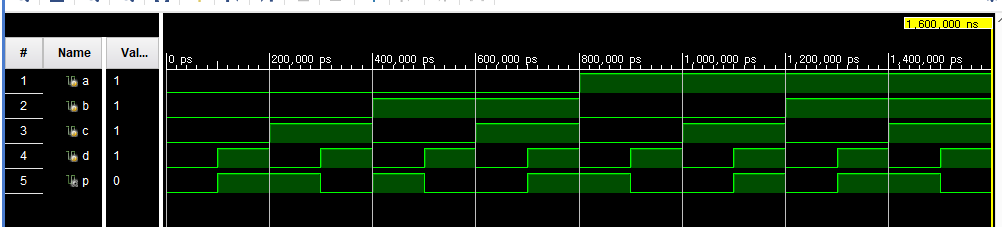
**3) Source Code**

****

**4) Schematic**

****

**5) Simulation**

****

**6) 결과 및 과정**

출력 parity는 입력 data의 모든 비트를 XOR 연산하는 것으로 정의된다. XOR 연산은 두 입력 비트가 다를 때 1을 출력한다. 결과적으로, 입력 비트 중 1의 개수가 홀수일 경우 패리티 비트 1, 짝수일 경우 0을 출력한다.

해당 Verilog 코드는 p = a^b^c^d로 작성하였으며, a,b,c,d는 각각 800, 400, 200, 100마다 bit가 전환되어 simulation 결과를 통해 진리표를 확인하기 쉽게 구현하였다.

simulation결과 예상했던 truth table과 동일한 결과를 얻을 수 있었다.

**- Even Parity bit checker**

**1) Truth Table**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Input** | | | | | **output** |
| **p** | **a** | **b** | **c** | **D** | **pec** |
| **0** | **0** | **0** | **0** | **0** | **0** |
| **0** | **0** | **0** | **0** | **1** | **1** |
| **0** | **0** | **0** | **1** | **0** | **1** |
| **0** | **0** | **0** | **1** | **1** | **0** |
| **0** | **0** | **1** | **0** | **0** | **1** |
| **0** | **0** | **1** | **0** | **1** | **0** |
| **0** | **0** | **1** | **1** | **0** | **0** |
| **0** | **0** | **1** | **1** | **1** | **1** |
| **0** | **1** | **0** | **0** | **0** | **1** |
| **0** | **1** | **0** | **0** | **1** | **0** |
| **0** | **1** | **0** | **1** | **0** | **0** |
| **0** | **1** | **0** | **1** | **1** | **1** |
| **0** | **1** | **1** | **0** | **0** | **0** |
| **0** | **1** | **1** | **0** | **1** | **1** |
| **0** | **1** | **1** | **1** | **0** | **1** |
| **0** | **1** | **1** | **1** | **1** | **0** |
| **1** | **0** | **0** | **0** | **0** | **1** |
| **1** | **0** | **0** | **0** | **1** | **0** |
| **1** | **0** | **0** | **1** | **0** | **0** |
| **1** | **0** | **0** | **1** | **1** | **1** |
| **1** | **0** | **1** | **0** | **0** | **0** |
| **1** | **0** | **1** | **0** | **1** | **1** |
| **1** | **0** | **1** | **1** | **0** | **1** |
| **1** | **0** | **1** | **1** | **1** | **0** |
| **1** | **1** | **0** | **0** | **0** | **0** |
| **1** | **1** | **0** | **0** | **1** | **1** |
| **1** | **1** | **0** | **1** | **0** | **1** |
| **1** | **1** | **0** | **1** | **1** | **0** |
| **1** | **1** | **1** | **0** | **0** | **1** |
| **1** | **1** | **1** | **0** | **1** | **0** |
| **1** | **1** | **1** | **1** | **0** | **0** |
| **1** | **1** | **1** | **1** | **1** | **1** |

**2) K-map**

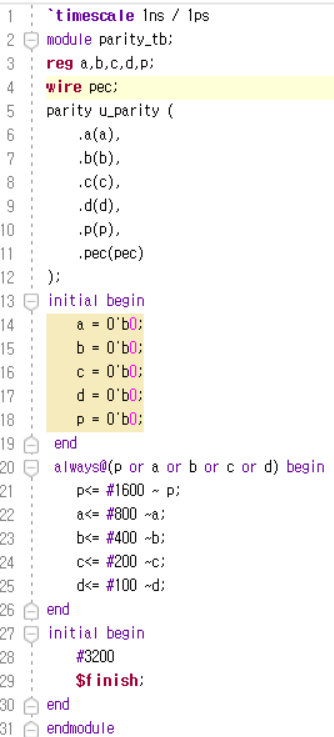
**P = 0**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ab\cd** | **00** | **01** | **11** | **10** |
| **00** | **0** | **1** | **0** | **1** |
| **01** | **1** | **0** | **1** | **0** |
| **11** | **0** | **1** | **0** | **1** |
| **10** | **1** | **0** | **1** | **0** |

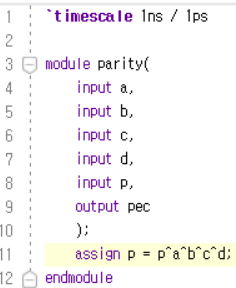
**P = 1**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ab\cd** | **00** | **01** | **11** | **10** |
| **00** | **1** | **0** | **1** | **0** |
| **01** | **0** | **1** | **0** | **1** |
| **11** | **1** | **0** | **1** | **0** |
| **10** | **0** | **1** | **0** | **1** |

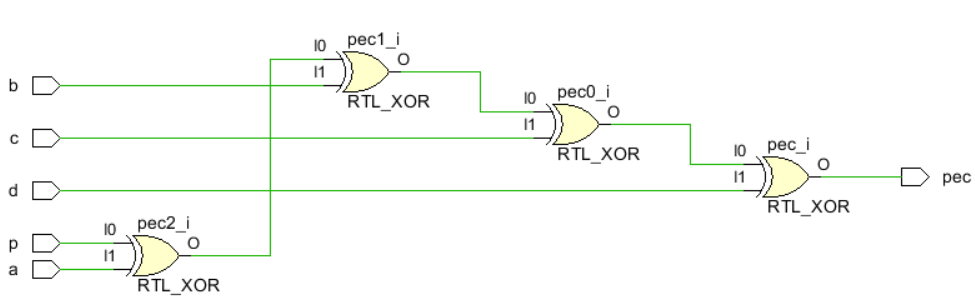
**pec = p’(**a⊕b⊕c⊕d**) + p(**a⊕b⊕c⊕d**)’ = p**⊕a⊕b⊕c⊕d

****

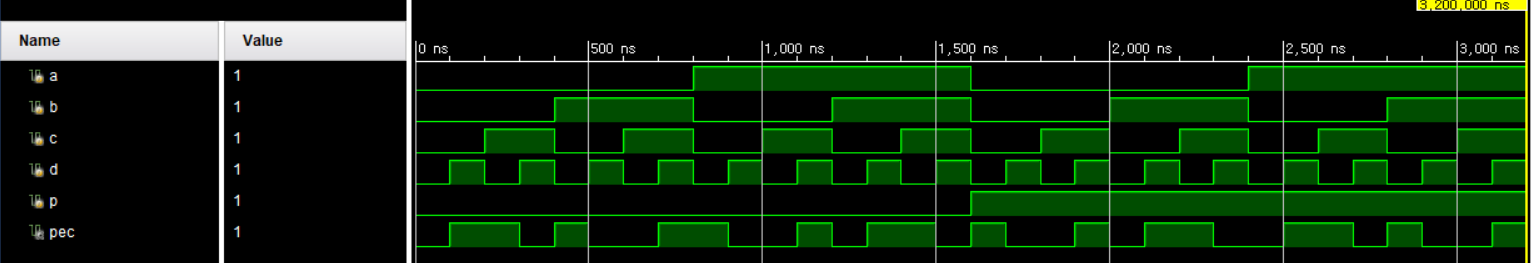
**3) Source Code**

****

**4) Schematic**

****

**5) Simulation**

****

**6) 결과 및 과정**

Pec는 입력 데이터의 모든 비트와 패리티 비트를 함께 XOR 연산하는 것으로 설정하였다. XOR 연산의 결과가 1이면, 전체 비트의 합이 홀수임을 의미하므로 오류가 있는 오류가 있는 것으로 간주한다. 결과가 0이면 모든 비트의 합이 짝수이므로 오류가 없는 것으로 간주한다.

해당 Verilog 코드는 pec = p^a^b^c^d로 작성하였으며, p,a,b,c,d는 각각 1600, 800, 400, 200, 100마다 bit가 전환되어 simulation 결과를 통해 진리표를 확인하기 쉽게 구현하였다.

simulation결과 예상했던 truth table과 동일한 결과를 얻을 수 있었다.

**2. Odd Parity bit generator 및 checker 의 simulation 결과 및 과정에 대해서 설명하시오. (Truth table 작성 및 k-map 포함)**

**- Odd Parity bit generator**

**1) Truth Table**

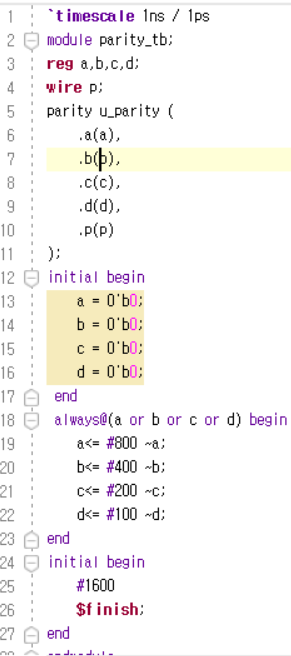
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Input** | | | | **Output** |
| **a** | **b** | **c** | **d** | **p** |
| **0** | **0** | **0** | **0** | **1** |
| **0** | **0** | **0** | **1** | **0** |
| **0** | **0** | **1** | **0** | **0** |
| **0** | **0** | **1** | **1** | **1** |
| **0** | **1** | **0** | **0** | **0** |
| **0** | **1** | **0** | **1** | **1** |
| **0** | **1** | **1** | **0** | **1** |
| **0** | **1** | **1** | **1** | **0** |
| **1** | **0** | **0** | **0** | **0** |
| **1** | **0** | **0** | **1** | **1** |
| **1** | **0** | **1** | **0** | **1** |
| **1** | **0** | **1** | **1** | **0** |
| **1** | **1** | **0** | **0** | **1** |
| **1** | **1** | **0** | **1** | **0** |
| **1** | **1** | **1** | **0** | **0** |
| **1** | **1** | **1** | **1** | **1** |

**2) K-map**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ab\cd** | **00** | **01** | **11** | **10** |
| **00** | **1** | **0** | **1** | **0** |
| **01** | **0** | **1** | **0** | **1** |
| **11** | **1** | **0** | **1** | **0** |
| **10** | **0** | **1** | **0** | **1** |

p = a’b’c’d’+a’b’cd+a’bc’d+a’bcd’+abc’d’+abcd+ab’c’d+ab’cd’

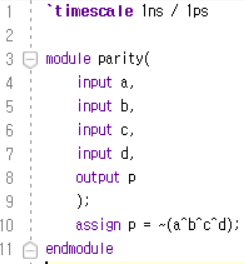
= a’b’(c’d’+cd)+a’b(c’d+cd’)+ab(c’d’+cd)+ab’(c’d+cd’)

****= (ab’+ab’)(c’d+cd’)+(ab+a’b’)(cd+c’d’)

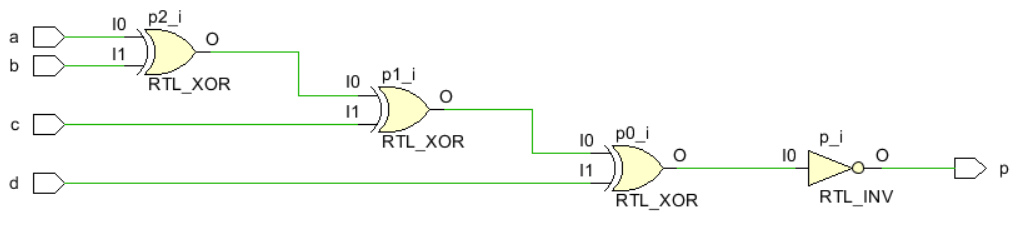
= (a⊕b)(c⊕d)+(a⊕b)’(c⊕d)’

= (a⊕b⊕c⊕d)’

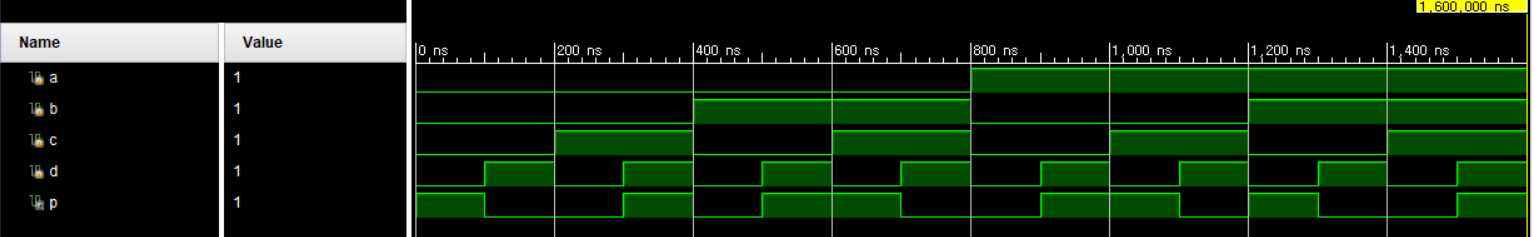
**3) Source Code**

****

**4) Schematic**

****

**5) Simulation**

****

**6) 결과 및 과정**

출력 parity는 입력 data의 모든 비트를 XOR 연산하는 것으로 정의되며, 최종 결과에 Not 연산을 추가하였다. 이렇게 함으로써, 입력 비트 중 1의 개수가 짝수일 경우 (원래 짝수 패리티에서는 0이 되는 상황), 패리티 비트가 1이 되어 전체 1의 개수를 홀수로 만든다.

해당 Verilog 코드는 p = (a^b^c^d)’로 작성하였으며, a,b,c,d는 각각 800, 400, 200, 100마다 bit가 전환되어 simulation 결과를 통해 진리표를 확인하기 쉽게 구현하였다.

simulation결과 예상했던 truth table과 동일한 결과를 얻을 수 있었다.

**- Odd Parity bit checker**

**1) Truth Table**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Input** | | | | | **output** |
| **p** | **a** | **b** | **c** | **D** | **pec** |
| **0** | **0** | **0** | **0** | **0** | **1** |
| **0** | **0** | **0** | **0** | **1** | **0** |
| **0** | **0** | **0** | **1** | **0** | **0** |
| **0** | **0** | **0** | **1** | **1** | **1** |
| **0** | **0** | **1** | **0** | **0** | **0** |
| **0** | **0** | **1** | **0** | **1** | **1** |
| **0** | **0** | **1** | **1** | **0** | **1** |
| **0** | **0** | **1** | **1** | **1** | **0** |
| **0** | **1** | **0** | **0** | **0** | **0** |
| **0** | **1** | **0** | **0** | **1** | **1** |
| **0** | **1** | **0** | **1** | **0** | **1** |
| **0** | **1** | **0** | **1** | **1** | **0** |
| **0** | **1** | **1** | **0** | **0** | **1** |
| **0** | **1** | **1** | **0** | **1** | **0** |
| **0** | **1** | **1** | **1** | **0** | **0** |
| **0** | **1** | **1** | **1** | **1** | **1** |
| **1** | **0** | **0** | **0** | **0** | **0** |
| **1** | **0** | **0** | **0** | **1** | **1** |
| **1** | **0** | **0** | **1** | **0** | **1** |
| **1** | **0** | **0** | **1** | **1** | **0** |
| **1** | **0** | **1** | **0** | **0** | **1** |
| **1** | **0** | **1** | **0** | **1** | **0** |
| **1** | **0** | **1** | **1** | **0** | **0** |
| **1** | **0** | **1** | **1** | **1** | **1** |
| **1** | **1** | **0** | **0** | **0** | **1** |
| **1** | **1** | **0** | **0** | **1** | **0** |
| **1** | **1** | **0** | **1** | **0** | **0** |
| **1** | **1** | **0** | **1** | **1** | **1** |
| **1** | **1** | **1** | **0** | **0** | **0** |
| **1** | **1** | **1** | **0** | **1** | **1** |
| **1** | **1** | **1** | **1** | **0** | **1** |
| **1** | **1** | **1** | **1** | **1** | **0** |

**2) K-map**

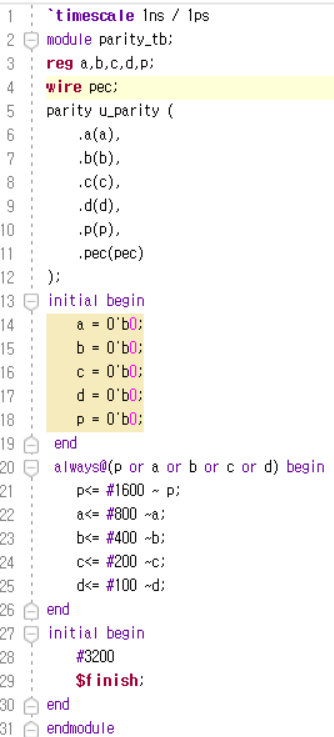
**P = 0**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ab\cd** | **00** | **01** | **11** | **10** |
| **00** | **1** | **0** | **1** | **0** |
| **01** | **0** | **1** | **0** | **1** |
| **11** | **1** | **0** | **1** | **0** |
| **10** | **0** | **1** | **0** | **1** |

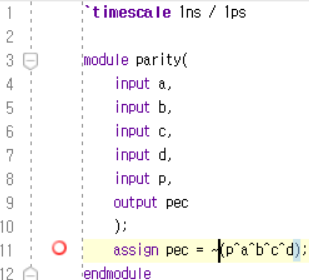
**P = 1**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **ab\cd** | **00** | **01** | **11** | **10** |
| **00** | **0** | **1** | **0** | **1** |
| **01** | **1** | **0** | **1** | **0** |
| **11** | **0** | **1** | **0** | **1** |
| **10** | **1** | **0** | **1** | **0** |

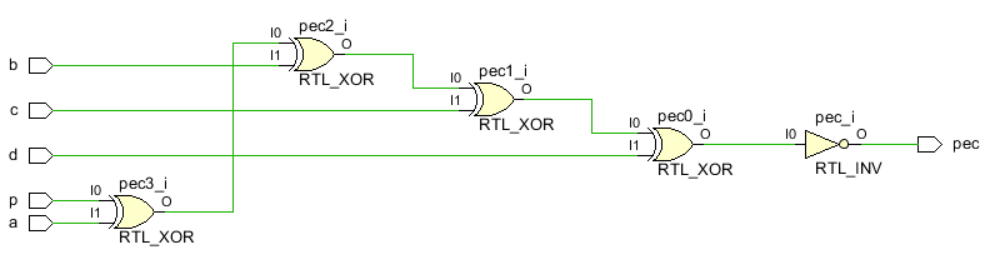
**pec = p’(**a⊕b⊕c⊕d**)’+ p(**a⊕b⊕c⊕d**) = (p**⊕a⊕b⊕c⊕d)’

****

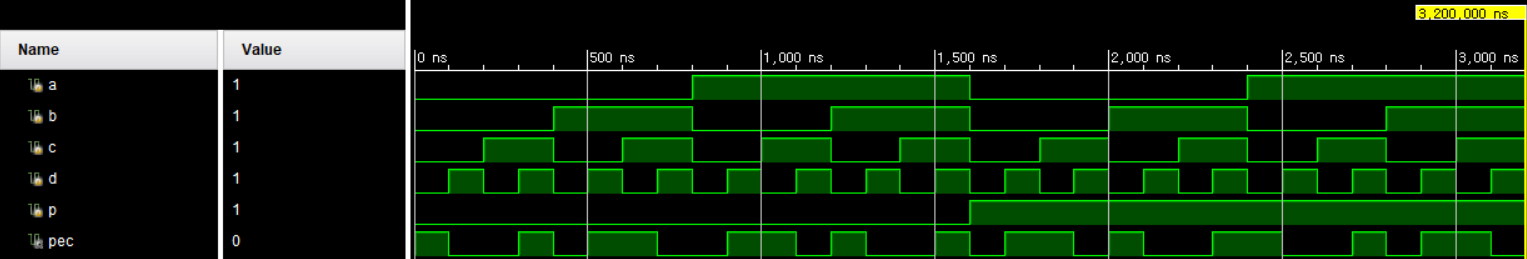
**3) Source Code**

****

**4) Schematic**

****

**5) Simulation**

****

**6) 결과 및 과정**

Pec는 입력 데이터의 모든 비트와 패리티 비트를 함께 XOR 연산하고, 최종 결과에 Not 연산을 추가하였다. XOR 연산의 결과가 1이면, 입력된 4개의 비트와 패리티 비트의 합이 짝수임을 의미하고, 에러가 없는 것을 의미한다. 연산 결과가 0이면, 입력된 4개의 비트와 패리티 비트의 합이 홀수임을 의미하고, 에러가 있음을 나타낸다.

해당 Verilog 코드는 pec = ~(p^a^b^c^d)로 작성하였으며, p,a,b,c,d는 각각 1600, 800, 400, 200, 100마다 bit가 전환되어 simulation 결과를 통해 진리표를 확인하기 쉽게 구현하였다.

simulation결과 예상했던 truth table과 동일한 결과를 얻을 수 있었다.

**3. 2-bit binary comparator simulation 결과 및 과정에 대해서 설명하시오. (Truth table 작성 및 k-map 포함, schematic 캡쳐 첨부)**

**1) Truth Table**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **input** | | | | **Output** | | |
| **A1** | **A2** | **B1** | **B2** | **F1(A<B)** | **F2(A=B)** | **F3(A>B)** |
| **0** | **0** | **0** | **0** | **0** | **1** | **0** |
| **0** | **0** | **0** | **1** | **1** | **0** | **0** |
| **0** | **0** | **1** | **0** | **1** | **0** | **0** |
| **0** | **0** | **1** | **1** | **1** | **0** | **0** |
| **0** | **1** | **0** | **0** | **0** | **0** | **1** |
| **0** | **1** | **0** | **1** | **0** | **1** | **0** |
| **0** | **1** | **1** | **0** | **1** | **0** | **0** |
| **0** | **1** | **1** | **1** | **1** | **0** | **0** |
| **1** | **0** | **0** | **0** | **0** | **0** | **1** |
| **1** | **0** | **0** | **1** | **0** | **0** | **1** |
| **1** | **0** | **1** | **0** | **0** | **1** | **0** |
| **1** | **0** | **1** | **1** | **1** | **0** | **0** |
| **1** | **1** | **0** | **0** | **0** | **0** | **1** |
| **1** | **1** | **0** | **1** | **0** | **0** | **1** |
| **1** | **1** | **1** | **0** | **0** | **0** | **1** |
| **1** | **1** | **1** | **1** | **0** | **1** | **0** |

**2) K-map**

**1)) A<B**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **A1A2\B1B2** | **00** | **01** | **11** | **10** |
| **00** | **0** | **1** | **1** | **1** |
| **01** | **0** | **0** | **1** | **1** |
| **11** | **0** | **0** | **0** | **0** |
| **10** | **0** | **0** | **1** | **0** |

F1 = A1’B1 + A2’B1B2 + A1’A2’B2

**2)) A=B**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **A1A2\B1B2** | **00** | **01** | **11** | **10** |
| **00** | **1** | **0** | **0** | **0** |
| **01** | **0** | **1** | **0** | **0** |
| **11** | **0** | **0** | **1** | **0** |
| **10** | **0** | **0** | **0** | **1** |

F2 = A1’A2’B1’B2’ + A1’A2B1’B2 + A1A2B1B2 + A1A2’B1B2’

=A1’B1’(A2’B2’+A2B2)+A1B1(A2B2+A2’B2’)

= A1B1(A2⊕B2)+A1’B1’(A2⊕B2)’

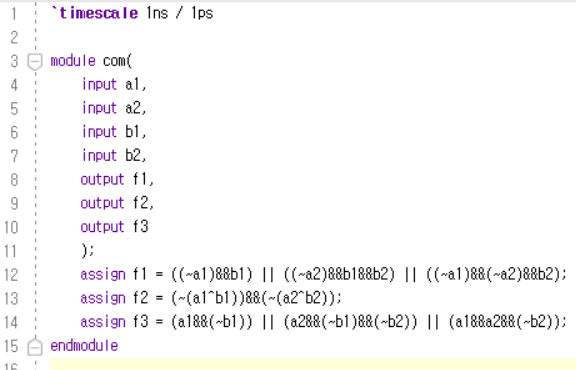
= (A1⊕B1)’(A2⊕B2)’

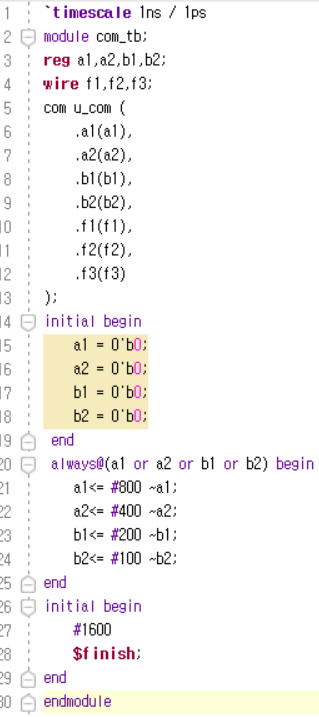
**3)) A>B**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **A1A2\B1B2** | **00** | **01** | **11** | **10** |
| **00** | **0** | **0** | **0** | **0** |
| **01** | **1** | **0** | **0** | **0** |
| **11** | **1** | **1** | **0** | **1** |
| **10** | **1** | **1** | **0** | **0** |

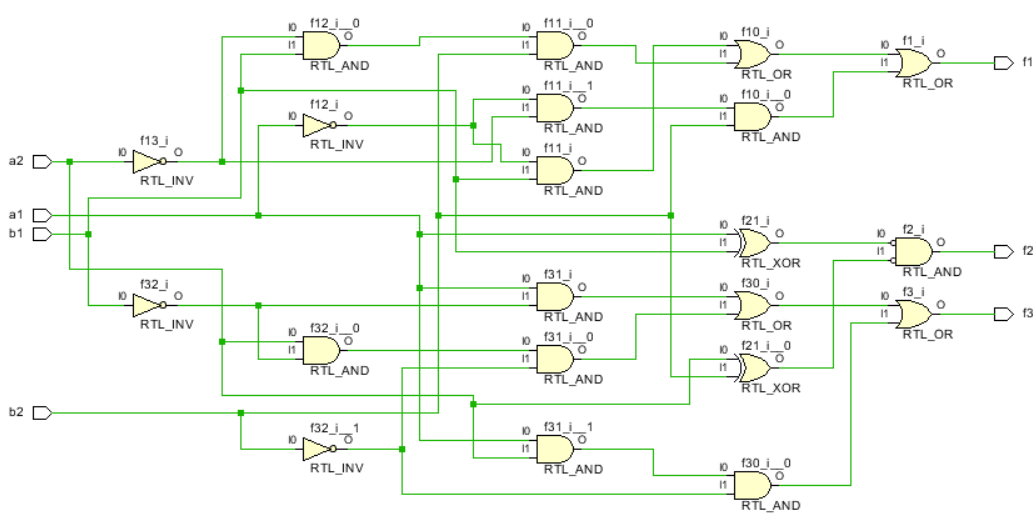
F3 = A1B1’ + A2B1’B2’ + A1A2B2’

**3) Source Code**

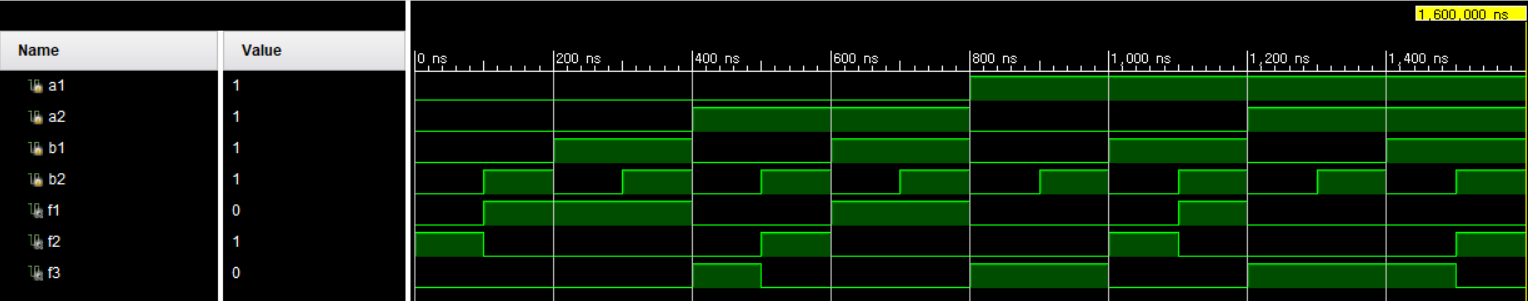
****

****

**4) Schematic**

****

**5) Simulation**

****

**6) 결과 및 과정**

두개의 이진수 A = a1a2, B = b1b2를 비교하여 세가지 가능한 결과를 작성하였다. F1,F2,F3는 각각 A<B, A=B, A>B이다. A<B일 때 여러 경우의 수를 나누어 생각하였다. ((~a1)&&b1),((~a2)&&b1&&b2),((~a1)&&(~a2)&&b2)은 각각 a1이 b1보다 작을 때, a2가 b1과 b2보다 작거나, a1과 a2 모두 작으면서 b2는 클 때를 나타낸다. A>B인 경우 A<B와 반대의 상황으로 고려해주었다. A=B의 경우 XNOR 연산과 AND 연산을 통해 a1과 b1이 같고, a2와 b2가 같은 경우를 고려하였다.

A1,A2,B1,B2는 각각 800, 400, 200, 100마다 bit가 전환되어 simulation 결과를 통해 진리표를 확인하기 쉽게 구현하였다. simulation결과 예상했던 truth table과 동일한 결과를 얻을 수 있었다.

**4. 결과 검토 및 논의 사항.**

이번 프로젝트에서는 even, odd parity generator 및 checker와 2 bit binary comparator를 설계하고 시뮬레이션 테스트를 통해 검증하였다.

Even parity generator는 입력 데이터의 모든 bit를 XOR 연산하여 생성하였고, odd parity generator는 모든 bit를 XOR 연산 후 NOT 연산을 통해 생성하였다. Even parity checker는 입력 데이터와 패리티 비트를 함께 XOR 연산하여 생성하였고, odd parity checker는 함께 XOR 연산한 후 NOT 연산을 통해 생성하였다. 2 bit binary comparator는 두 비트 A,B를 각각 A=a1a2, B=b1b2로 구분하고, 각각의 bit 상태에 따라 A<B, A=B, A>B의 결과를 비교하였다.

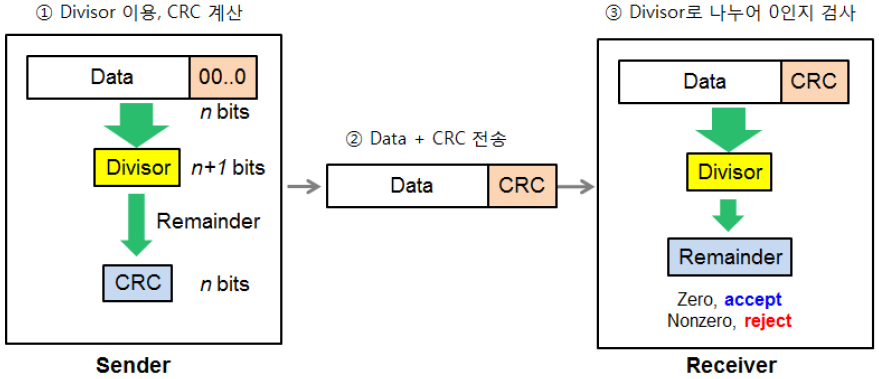
위의 구현을 simulation 통해 확인 결과, 입력 패턴의 비트 전환 주기를 통해 각 모듈의 동작을 체계적으로 검증할 수 있었다. 또한 진리표와 일치하는 시뮬레이션 결과는 설계가 의도한대로 정확하게 동작함을 의미한다. 이는 각 구성 요소의 기능이 올바르게 구현되었음을 보여준다. 이 프로젝트를 통해 복잡한 디지털 로직을 효과적으로 설계하고 검증할 수 있었다. 나아가 이를 바탕으로 더욱 복잡한 시스템으로 확장하는 기반이 될 수 있다.

**5. 추가 이론 조사 및 작성.**

이번 프로젝트에서 구현한 parity bit는 데이터 전송 오류를 감지하기 위해 사용되는 가장 간단한 형태의 오류 검출코드이다. 주로 싱글 비트 오류를 감지하는 데 사용되며, 간단한 통신 시스템이나 메모리 시스템에서 흔히 볼 수 있다. 그러나 복수의 오류는 감지하지 못하는 한계가 있다. 이 외에도 여러 오류 검출 방법이 존재한다.

**CRC(Cyclic Redundancy Cehck)**

CRC는 데이터의 무결성을 검사하기 위해 널리 사용되는 기술로, 특히 데이터 통신과 저장 매체에서 오류를 감지하는 데 효과적이다. CRC는 수학적인 알고리즘에 기반을 둔 더 복잡한 오류 검출 방식이며, 전송되는 데이터 블록에 대해 높은 수준의 오류 감지 능력을 제공한다. CRC의 기본 원리는 모든 데이터를 하나의 큰 이진 숫자로 취급하고, 이 숫자를 사전에 정의된 고정된 생성 다항식(generator polynomial)으로 나누는 것이다. 이 과정에서 나머지를 구하여 그 나머지를 오류 검사 코드로 사용한다. CRC는 오류 감지 능력이 강하고, 많은 통신 프로토콜과 데이터 저장 포맷에서 표준화되어 사용되고 있다. 하지만 오류 정정 능력의 부재와 계산 복잡성 등 단점이 존재한다.



**Checksum**

체크섬은 데이터의 무결성을 확인하는 간단하면서도 효율적인 방법으로, 데이터 전송이나 저장 과정에서 오류를 감지하는 데 사용된다. 체크섬은 일련의 데이터에 대한 간단한 계산을 통해 얻은 값으로, 데이터가 송수신되거나 저장될 때 계산되어, 그 데이터와 함께 전송되거나 저장된다. 데이터를 받는 시스템은 같은 체크섬 계산을 수행하여, 수신한 데이터의 체크섬과 비교함으로써 데이터의 무결성을 검증한다. Checksum은 모든 데이터 세그먼트의 값을 합산한다. 이때 오버플로우를 일으키는 경우 주로 합계에 다시 추가되거나 무시된다. 이후 최종 합계를 간단히 수정하여 최종 체크섬 값을 생성한다. 주로 보통 합계에 1의 보수를 취하는 과정을 통해 이루어진다. 이러한 체크섬은 간단하고 빠르며 구현이 용이하다는 장점이 있다. 하지만 제한된 오류 감지 능력과 오류 정정 능력의 부재라는 단점이 존재한다.