6주차 결과보고서

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

**1. 실험 목적**

본 실험의 주요 목적은 Verilog HDL을 활용하여 가산기(Adder)와 감산기(Subtractor)의 기본 개념을 탐구하고, 부호 변환기(Code Converter)의 작동 원리를 깊이 이해하는 데 있다. 이 과정을 통해 디지털 논리 설계의 핵심적인 이론을 실질적인 실습과 결합함으로써, 이론과 실습의 통합된 학습 경험을 강화하게 된다.

실험은 먼저 Verilog HDL을 사용하여 다양한 가산기와 감산기를 설계하고 구현하는 것으로 시작된다. 이를 통해 수치 데이터의 처리 및 연산에 있어서 핵심적인 디지털 회로의 설계 방법을 학습한다. 이어서, 다양한 부호 변환기의 설계와 구현을 통해 데이터의 표현 방식을 변환하는 데 필요한 논리 회로의 작동 원리에 대해 탐구한다.

이러한 디지털 회로의 설계 및 구현 과정을 거친 후, Vivado와 같은 시뮬레이션 툴을 사용하여 설계된 회로의 정확성과 효율성을 평가한다. 시뮬레이션을 통해 참가자들은 회로 설계에서 발생할 수 있는 잠재적 오류를 사전에 식별하고 수정할 수 있으며, 이를 통해 회로의 성능을 최적화할 수 있다.

마지막 단계에서는 FPGA를 사용하여 Verilog로 구현된 회로의 실제 동작을 확인한다. 이 과정에서 디지털 회로의 실제 구현 과정을 체험하고, 시뮬레이션에서 얻은 이론적 지식을 실제 하드웨어에 적용하는 방법을 배우게 된다.

**2. Full Adder 및 Half Adder 의 simulation 결과 및 과정에 대해서 설명하시오.**

**1) Half Adder**

**1)) Schematic**

**도표, 평면도이(가) 표시된 사진

자동 생성된 설명**

**2)) Design Source와 Testbench file**

**텍스트, 스크린샷, 폰트, 디스플레이이(가) 표시된 사진

자동 생성된 설명텍스트, 스크린샷, 번호, 폰트이(가) 표시된 사진

자동 생성된 설명**

**3)) Simulation 결과**

**스크린샷, 텍스트이(가) 표시된 사진

자동 생성된 설명**

**4)) 진리표**

|  |  |  |  |
| --- | --- | --- | --- |
| Input | | Output | |
| A | **B** | **S(sum)** | **C(Carry)** |
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |

**5)) 결과 및 구현 과정**

Half Adder의 설계는 두 입력 비트 a와 b에 대해 두가지 연산을 각각 수행하였다. 합(s)은 assign s = a^b, 캐리(c)는 assign c = a&&b를 통해 각각 XOR연산과 AND 연산을 수행하였다. 시뮬레이션을 위한 Testbench는 입력 비트 a와 b를 각각 100ns, 200ns마다 변화시켜 각각의 경우의 수 모두 살펴보았다. 그 결과 Sum은 a와 b 중 하나가 1인 경우 1이 나왔고, Carry는 a와 b 모두 1인 경우 1이 나왔다. 이러한 과정을 통해, 합과 캐리 출력이 입력에 따라 어떻게 변하는지 관찰할 수 있으며, Half Adder의 기본 동작에 맞게 수행되었음을 알 수 있다.

**2) Full Adder**

**1)) Schematic**

**도표, 라인, 평면도, 그래프이(가) 표시된 사진

자동 생성된 설명**

**2)) Design Source와 Testbench file**

**텍스트, 스크린샷, 디스플레이, 번호이(가) 표시된 사진

자동 생성된 설명텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명**

**3)) Simulation 결과**

**스크린샷, 텍스트, 사각형, 디스플레이이(가) 표시된 사진

자동 생성된 설명**

**4)) 진리표**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Input | | | Output | |
| A | **B** | **C\_in** | **S** | **C\_out** |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

**5)) 결과 및 구현 과정**

Full Adder의 설계는 세 입력 비트 a와 b, Carry\_in에 대해 두가지 연산을 각각 수행하였다. 합(s)은 assign s = a^b, 캐리(Carry\_out)는 assign c\_out = (c\_in&&(a^b))|| (a&&b)을 통해 각각 XOR연산과, a와 b의 XOR 연산에 c in을 And한 값과 a와 b를 AND연산을 한 값의 OR 연산을 수행하였다. 시뮬레이션을 위한 Testbench는 입력 비트 a와 b를 각각 100ns, 200ns마다 변화시키고, Carry\_in을 400ns마다 변화시켜 각각의 경우의 수 모두 살펴보았다. 그 결과 Sum은 a와 b 중 하나가 1이고, carry가 없는 경우와 a와 b가 0이고 carry가 있는 경우, a와 b가 1이고 carry가 1인 경우 1이 나왔다. Carry\_out은 a, b, carry\_out 중 2개 이상 1인 경우 1이 나왔다.(진리표에서는 a는 400ns, b는 200ns, carry\_in은 100ns로 변화하는 모습을 좀 더 보기 좋게 나타냈었다.) 이러한 과정을 통해, 합과 캐리 출력이 입력에 따라 어떻게 변하는지 관찰할 수 있으며, Full Adder의 기본 동작에 맞게 수행되었음을 알 수 있다.

**3. Full Subtractor 및 Half Subtractor 의 simulation 결과 및 과정에 대해서 설명하시오.**

**1) Half Subtractor**

**1)) Schematic**

**도표, 평면도, 기술 도면, 라인이(가) 표시된 사진

자동 생성된 설명**

**2)) Design Source와 Testbench file**

**텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명**

**3)) Simulation 결과**

**텍스트, 스크린샷, 라인, 도표이(가) 표시된 사진

자동 생성된 설명**

**4)) 진리표**

|  |  |  |  |
| --- | --- | --- | --- |
| Input | | Output | |
| A | **B** | **B(Borrow)** | **D(Difference)** |
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 0 |

**5)) 결과 및 구현 과정**

Half Subtractor의 설계는 두 입력 비트 a와 b에 대해 두가지 연산을 각각 수행하였다. d(Difference)는 assign d = a^b, b(borrow)는 assign borrow = (~a)&&b를 통해 각각 XOR연산과 NOT, AND 연산을 수행하였다. 시뮬레이션을 위한 Testbench는 입력 비트 a와 b를 각각 100ns, 200ns마다 변화시켜 각각의 경우의 수 모두 살펴보았다. 그 결과 Difference는 a와 b 중 하나가 1인 경우 1이 나왔고, Carry는 a는 0, b는 1인 경우 1이 나왔다. 이러한 과정을 통해, Difference와 Borrow 출력이 입력에 따라 어떻게 변하는지 관찰할 수 있으며, Half Subtractor의 기본 동작에 맞게 수행되었음을 알 수 있다.

**2) Full Subtractor**

**1)) Schematic**

**도표, 평면도, 라인이(가) 표시된 사진

자동 생성된 설명**

**2)) Design Source와 Testbench file**

**텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명텍스트, 스크린샷, 폰트, 번호이(가) 표시된 사진

자동 생성된 설명**

**3)) Simulation 결과**

**스크린샷, 텍스트, 사각형이(가) 표시된 사진

자동 생성된 설명**

**4)) 진리표**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Input | | | Output | |
| A(n) | **B(n)** | **B(n-1)** | **b(n)** | **D(n)** |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 |

**5)) 결과 및 구현 과정**

Full Subtractor의 설계는 세 입력 비트 a와 b, borrow\_(n-1)에 대해 두가지 연산을 각각 수행하였다. Difference(d)는 assign dn = a^b^borrow\_(n-1), borrown(borrow\_n)은 assign borrown = (~(a^b)&&borrow\_(n-1))||(a&&b)를 통해 각각 XOR연산과, a와 b의 XOR 연산에 NOT을 한 값과 borrow\_(n-1)을 AND를 하고, a와 b를 AND연산을 한 값과 OR 연산을 통해 구현하였다. 시뮬레이션을 위한 Testbench는 입력 비트 a와 b를 각각 400ns, 200ns마다 변화시키고, borrow\_(n-1)을 100ns마다 변화시켜 각각의 경우의 수 모두 살펴보았다. 그 결과 difference는 세 입력 a, b, borrow\_(n-1) 중 홀수개의 1의 값을 받는 경우 1이 나왔다. borrow\_n 은 a가 0이고 b가 1인 경우, a와 b가 동일하지만 borrow\_(n-1)이 1인 경우 1이 나왔다. 이러한 과정을 통해, Difference와 Borrow 출력이 입력에 따라 어떻게 변하는지 관찰할 수 있으며, Full Subtractor의 기본 동작에 맞게 수행되었음을 알 수 있다.

**4. 8421(BCD)-2421 Code converter simulation 결과 및 과정에 대해서 설명하시오. (진리표 작성 및 카로노맵 SOP form, POS form 포함)**

**1) 진리표**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Decimal | 8421 BCD | | | | 2421 BCD | | | |
| A | B | C | D | W | X | Y | Z |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 |
| 2 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 |
| 3 | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 4 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| 5 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 1 |
| 6 | 0 | 1 | 1 | 0 | 1 | 1 | 0 | 0 |
| 7 | 0 | 1 | 1 | 1 | 1 | 1 | 0 | 1 |
| 8 | 1 | 0 | 0 | 0 | 1 | 1 | 1 | 0 |
| 9 | 1 | 0 | 0 | 1 | 1 | 1 | 1 | 1 |
| 10 | 1 | 0 | 1 | 0 | x | x | x | x |
| 11 | 1 | 0 | 1 | 1 | x | x | x | x |
| 12 | 1 | 1 | 0 | 0 | x | x | x | x |
| 13 | 1 | 1 | 0 | 1 | x | x | x | x |
| 14 | 1 | 1 | 1 | 0 | x | x | x | x |
| 15 | 1 | 1 | 1 | 1 | x | x | x | x |

**2) 카르노맵**

**1)) W의 카르노 맵**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| AB\CD | 00 | 01 | 11 | 10 |
| 00 | **0** | **0** | **0** | **0** |
| 01 | **0** | **1** | **1** | **1** |
| 11 | **x** | **x** | **x** | **x** |
| 10 | **1** | **1** | **x** | **x** |

**3)) Y의 카르노 맵**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| AB\CD | 00 | 01 | 11 | 10 |
| 00 | **0** | **0** | **1** | **1** |
| 01 | **0** | **1** | **0** | **0** |
| 11 | **x** | **x** | **x** | **x** |
| 10 | **1** | **1** | **x** | **x** |

**2)) X의 카르노 맵**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| AB\CD | 00 | 01 | 11 | 10 |
| 00 | **0** | **0** | **0** | **0** |
| 01 | **1** | **0** | **1** | **1** |
| 11 | **x** | **x** | **x** | **x** |
| 10 | **1** | **1** | **x** | **x** |

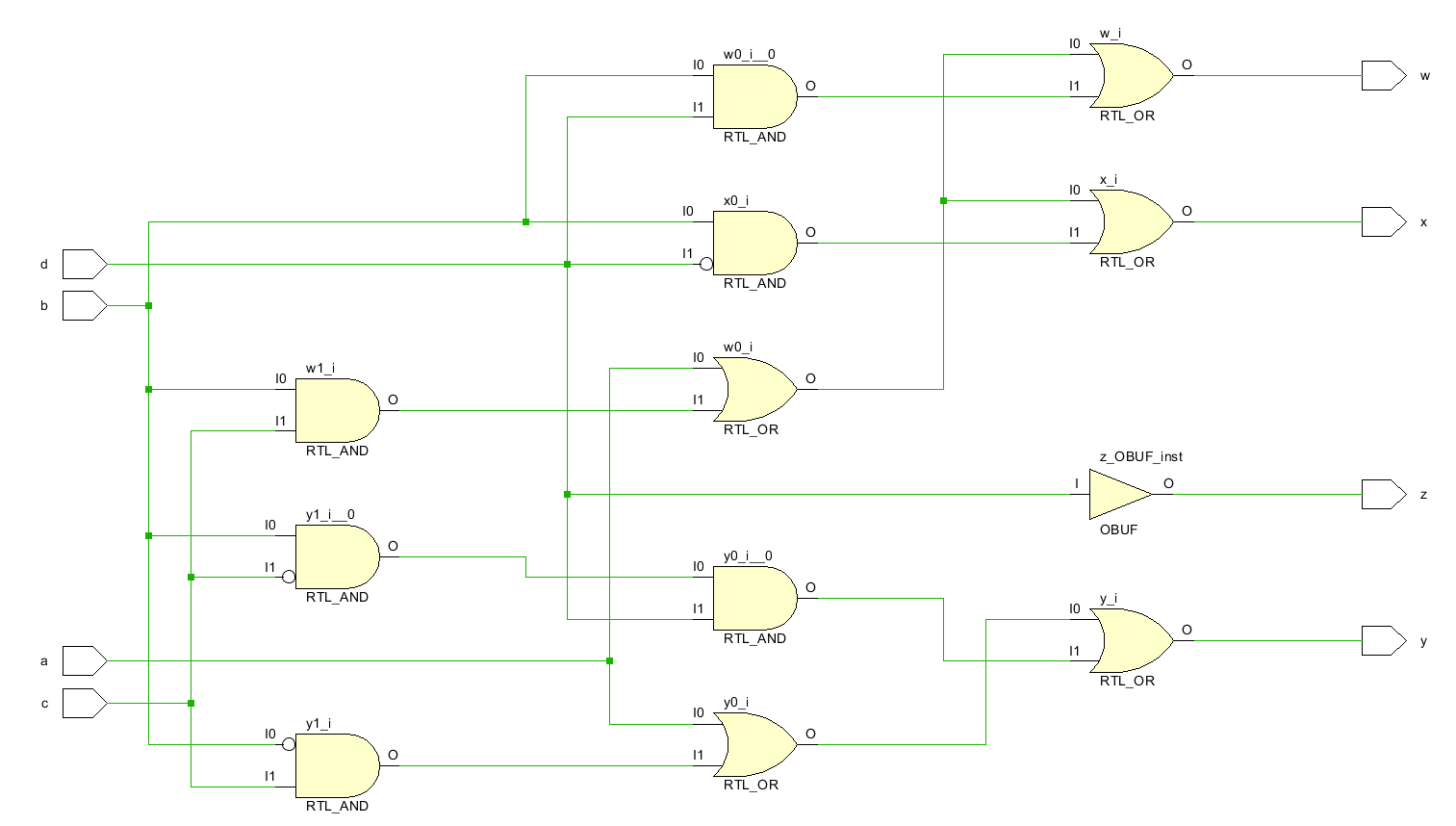
**3)) Z의 카르노 맵**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| AB\CD | 00 | 01 | 11 | 10 |
| 00 | **0** | **1** | **1** | **0** |
| 01 | **0** | **1** | **1** | **0** |
| 11 | **x** | **x** | **x** | **x** |
| 10 | **0** | **1** | **x** | **x** |

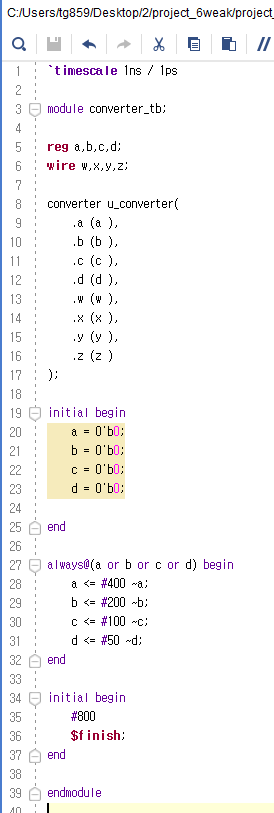
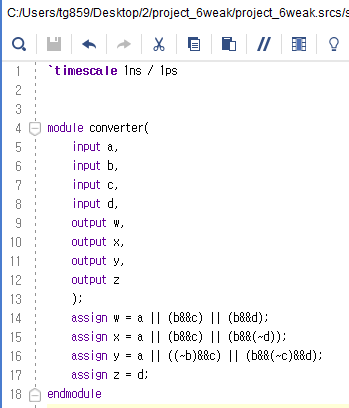
**5)) SOP Form & POS Form**

|  |  |  |
| --- | --- | --- |
|  | SOP Form | POS Form |
| W | **A + BC + BD** | **(A + B) (A + C + D)** |
| X | **A + BC + BD’** | **(A + B)(A + C + D’)** |
| Y | **A + B’C + BC’D** | **(A + B + C)(B’ + D)(B’ + C’)** |
| Z | **D** | **D** |

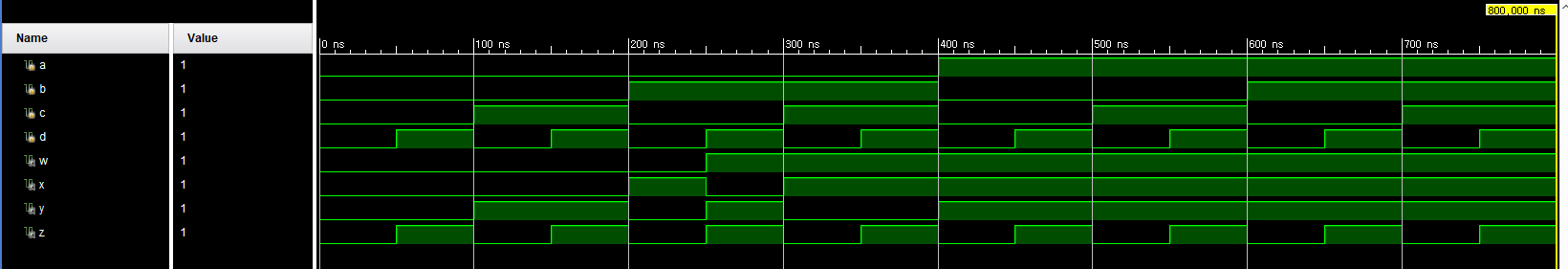
**3) Schematic**

****

**4) Design Source와 Testbench file**

****

**5) Simulation 결과**

****

**6) 결과 및 구현 과정**

8421 BCD 코드 입력(a, b, c, d)을 받아서 2421 BCD 코드 출력(w, x, y, z)으로 변환하였다. 8421 코드에서 가장 상위 비트인 a는 직접 2421 코드의 w로 사용된다. 또한, b가 1일 때 c 또는 d가 1이면 w도 1이 된다. x는 a 또는 b와 c의 AND 결과, 그리고 b와 d의 NOT의 AND 결과에 따라 결정된다. y는 a, ~b와 c의 AND 결과, 또는 b와 ~c 그리고 d의 AND 결과에 따라 결정된다. 2421 코드의 최하위 비트인 z는 8421 코드의 d 비트와 동일하다. 테스트는 초기 모든 입력비트 (a, b, c, d)를 0으로 설정하였다. 이후 입력 비트들에 대해 일정한 시간 간격(#50, #100, #200, #400)마다 값을 반전시키며, 이는 가장 빠른 d부터 시작하여 가장 느린 a까지 진행하였다. 시뮬레이션은 입력 비트가 모두 한번 이상 반전될 충분한 시간인 800ns 후에 종료하였다.

해당 구현을 통해 simulation 결과를 살펴본 결과 실습에서 제공된 8421-2421 BCD code 표에 맞게 결과가 나온 것을 확인할 수 있었다. 이를 통해 Truth table과 카르노 맵을 통한 SOP form 변환이 올바르게 이루어졌고, 실습 또한 성공적으로 진행하였음을 알 수 있다.

**5. 결과 검토 및 논의사항.**

해당 실습 과정에서 Half Adder, Full Adder, Half Subtractor, Full Subtractor, 그리고 8421-2421 BCD code converter의 구현을 통해 디지털 논리 회로의 기본 원리와 Verilog HDL을 활용한 실제 구현 방법을 학습하였다. Half Adder의 경우, XOR과 AND 연산을 통해 합과 캐리를 생성하며, 각 입력 조합에 대한 정확한 결과를 확인할 수 있었다. 이는 두 비트의 단순 덧셈에서 발생할 수 있는 모든 경우를 포괄한다. Full Adder는 추가적인 Carry 입력을 포함하여 보다 복잡한 덧셈 연산을 수행하고, 여러 비트의 덧셈을 체인 방식으로 확장하는 데 필요한 기반을 제공한다. 반면, Half Subtractor와 Full Subtractor는 감산 연산을 처리하며, Difference와 Borrow 출력이 각 입력에 따라 어떻게 변화하는지를 명확하게 보여주었다. 이들 회로는 감산 연산이 수행될 때 발생할 수 있는 모든 시나리오를 다루며, 디지털 계산기나 다른 컴퓨팅 시스템에서 중요한 역할을 한다. 8421-2421 BCD code converter의 구현은 다소 복잡한 논리 연산을 필요로 한다. 특히, 입력 조합에 따라 다양한 출력을 생성하는 데 필요한 조건부 논리가 포함된다. 이는 NAND, NOR 게이트만 활용하여 회로를 구성하여 비용을 훨씬 절감할 수 있다.

**6. 추가 이론 조사 및 작성.**

BCD(Binary-Coded Decimal) 코드는 각 10진수를 별도의 이진 숫자로 표현하는 여러 가지 방식을 포함한다. 8421과 2421 코드 외에도 여러 가지 다른 BCD 코드가 있으며, 각각은 특정한 용도와 특징을 가진다.

5421 BCD 코드는 각 10진수 숫자를 특정한 4비트 이진 코드로 표현하는 방식 중 하나이다. 이 코드의 명칭인 5421은 각 비트의 가중치를 나타낸다. 4비트 코드에서 가장 왼쪽 비트(가장 상위 비트)는 5의 가중치를, 그 다음 비트는 4의 가중치를, 그 다음은 2의 가중치를, 가장 오른쪽 비트(가장 하위 비트)는 1의 가중치를 가진다. 5421 BCD 코드에서 각 10진수는 그에 해당하는 가중치의 합으로 표현된다. 예를 들어, 10진수 7은 이진수 1101로 표현된다. 여기서 첫 번째 비트(1)는 5를, 두 번째 비트(1)는 4를, 세 번째 비트(0)는 2를 나타내지 않으며, 네 번째 비트(1)는 1을 나타낸다. 이 비트들을 합하면 5 + 4 + 0 + 1 = 10, 즉 7을 나타내게 된다. 이러한 원리로 인해 5421 BCD 코드는 각 비트의 가중치가 쉽게 인지될 수 있어 직관성이 좋고, 비교적 간단한 가중치 구조로 인해, 이진수에서 10진수로, 또는 그 반대로의 변환이 비교적 단순하다. 또한 10진수를 이진 코드로 변환하여 저장 및 처리할 때, 공간 및 처리 효율성을 제공한다.

그레이 코드(Gray Code) BCD는 그레이 코드의 특성을 이용하여 BCD(Binary-Coded Decimal) 형식으로 확장한 코드이다. 그레이 코드는 연속된 값들이 변할 때 하나의 비트만이 변화하는 방식으로 설계되어 있어, 일반적인 이진 코드와는 다르게 오류 감지 및 수정이 용이하다는 장점이 있다. 그레이 코드 BCD를 구성하는 방식은 일반적인 그레이 코드를 생성하는 알고리즘을 기반으로 한다. 그레이 코드는 이진 코드를 그레이 코드로 변환할 때 각 비트를 이전 비트와 XOR 연산하여 얻을 수 있다. 예를 들어, 표준 이진 코드를 그레이 코드로 변환하는 공식은 다음과 같다. Gray(i) = Binary(i) XOR Binary(i+1) 그레이 코드 BCD는 주로 오류 감지가 중요한 시스템에서 사용된다. 디지털 통신, 고정밀 측정 기기, 로터리 인코더 등에서 그레이 코드의 오류 감지 능력과 함께 사용된다. 데이터 전송 중 비트의 변화가 최소화되기 때문에, 오류 발생 시 오류의 원인을 빠르게 찾아낼 수 있으며, 시스템의 전반적인 신뢰성을 향상시킬 수 있다.

|  |  |  |
| --- | --- | --- |
| Decimal | 5421 BCD 코드 | 그레이 코드 BCD |
| 0 | 0000 | 0000 |
| 1 | 0001 | 0001 |
| 2 | 0010 | 0011 |
| 3 | 0011 | 0010 |
| 4 | 0100 | 0110 |
| 5 | 1000 | 0111 |
| 6 | 1001 | 0101 |
| 7 | 1010 | 0100 |
| 8 | 1011 | 1100 |
| 9 | 1100 | 1101 |