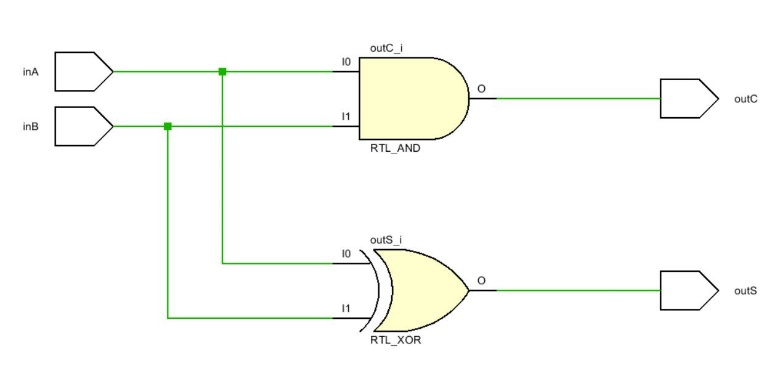
6주차 결과보고서

전공: 물리학과 학년: 3학년 학번: 20191286 이름: 김나현

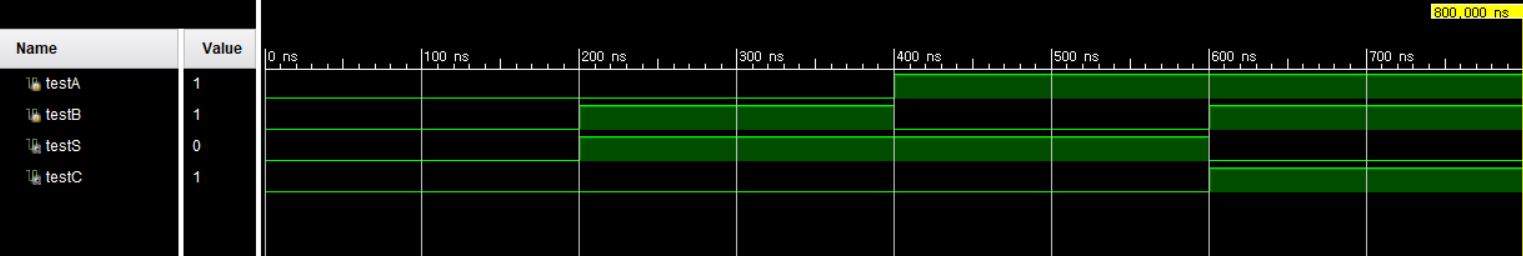
1. 실험 목적

여러 개의 논리 게이트로 만든 Adder, Subtractor, code converter를 Verilog 코드를 통해 구현해보고 test bench와 simulation 결과를 보고 전, 반가산기와 전, 반감산기, code converter의 동작을 확인해보는 것입니다.

1. Full Adder 및 Half Adder의 simulation 결과 및 과정에 대해서 설명하시오(진리표 포함).

<Figure 1> Half adder schematic diagram

위의 <Figure 1>은 반가산기를 구현하기 위해 Verilog 코드로, 입력 inA, inB에 대한 출력 outC, outS를 할당한 뒤 schematic을 확인한 결과입니다. outC를 얻기 위해서는 두 입력을 AND 연산하면 되고, outS를 얻기 위해서는 두 입력을 XOR 연산하면 된다는 사실을 알 수 있습니다. 이때, 반가산기에서 outC는 carry로, 이진 연산에서 올림수를 의미하고 outS는 해당 자릿수의 입력 합을 의미합니다. 이 Verilog 코드를 이용해서 simulation 결과를 확인하면 다음 <Figure 2>와 같습니다.



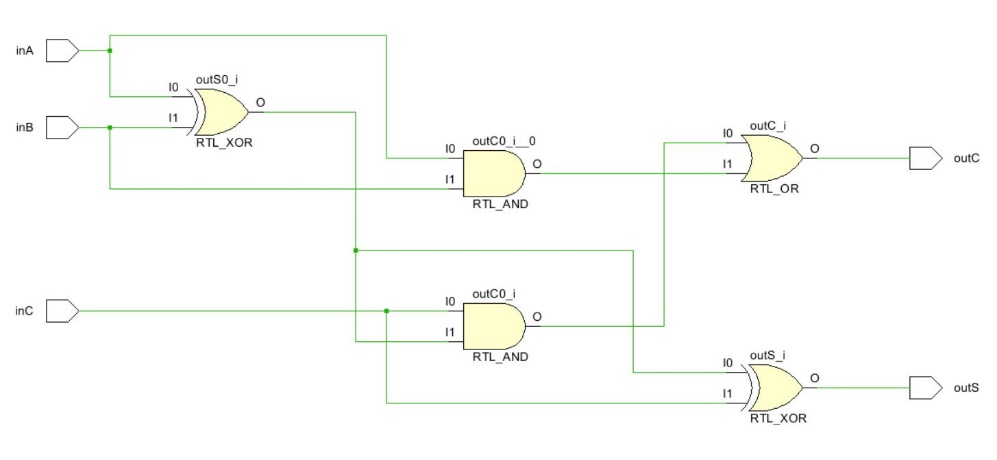
<Figure 2> Half adder simulation

이를 이용해 반가산기의 진리표를 작성해보면 다음 <Table 1>과 같습니다.

<Table 1> Half adder(반가산기)의 진리표

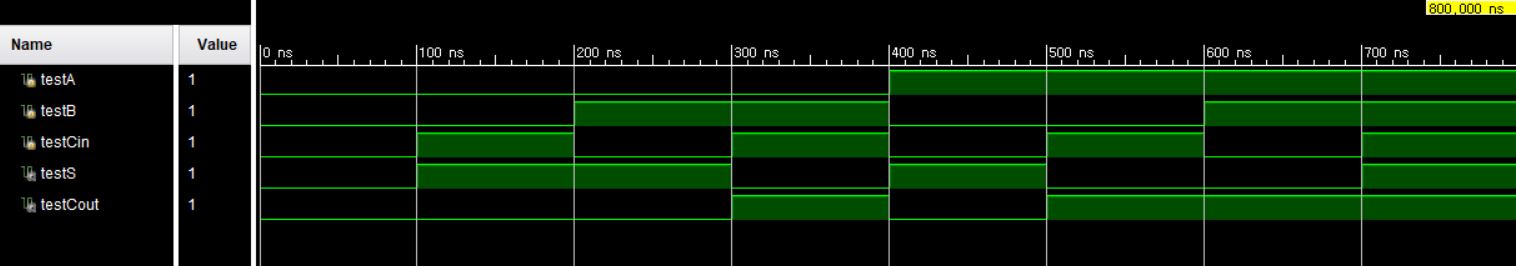
|  |  |  |  |
| --- | --- | --- | --- |
| A | B | S | C |
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |

위의 <Table 1>을 통해, 두 입력이 모두 1일 때 올림수, 즉 Carry를 의미하는 C가 1이 됨을 확인할 수 있고 두 입력의 합, 즉 S는 두 입력 중 하나만 1일 때, 즉 입력 중 참인 입력이 홀수 개일 때 1이 됨을 확인할 수 있습니다. 반가산기의 코드를 작성하기 위해서는 <Table 1>과 같은 진리표를 먼저 작성하고 SOP를 이용해 S는 A’B+AB’, C는 AB이라는 논리 함수로 나타낼 수 있다는 것을 통해 각각 XOR 연산과 AND 연산을 사용하여 구할 수 있다는 것을 알 수 있습니다.

하지만 반가산기는 하나의 비트를 연산하는 데에는 문제가 없지만 입력으로 하위 자릿수의 올림수가 없으므로 여러 개의 비트 연산에는 문제가 있습니다. 따라서 이러한 문제를 해결하기 위해 여러 비트 연산에는 반가산기의 문제점을 보완한 전가산기라는 것을 사용합니다. 다음 <Figure 3>는 전가산기의 schematic diagram입니다.

<Figure 3>Full adder schematic diagram

<Figure 3>, 전가산기의 schematic diagram을 보면 <Figure 1>과 다르게 입력이 하나 더 존재하는 것을 알 수 있습니다. 이는 앞서 말한 하위 자릿수의 올림수를 의미하고 이러한 하위 자릿수의 올림수가 입력으로 인가되기 때문에 전가산기는 여러 비트 연산이 가능합니다. 다음 <Figure 4>는 전가산기의 simulation 결과입니다.



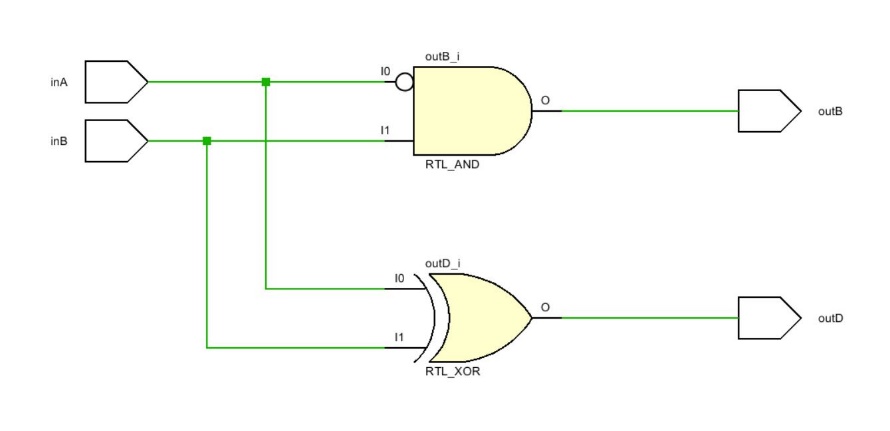
<Figure 4> Full adder simulation

이를 이용해 전가산기의 진리표를 작성해보면 다음 <Table 2>와 같습니다.

<Table 2> 1 bit 전가산기의 진리표

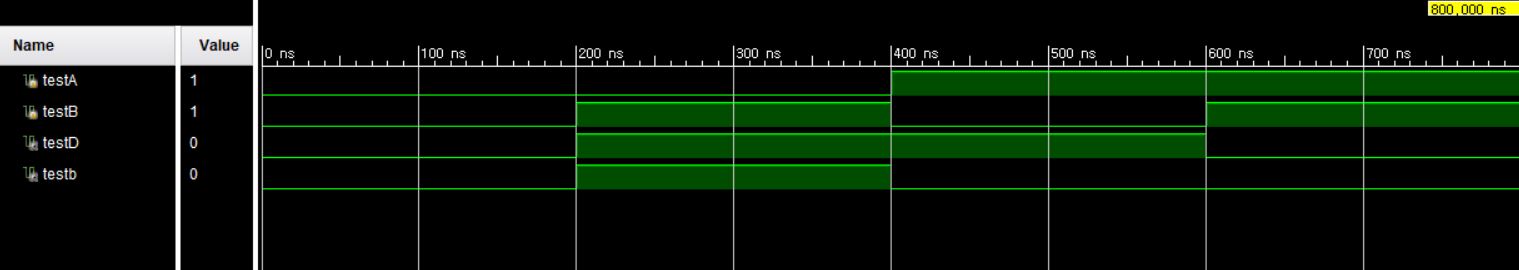
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| A | B | Cin | S | Cout |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

위의 <Table 2>를 통해, 세 개의 입력 중 참인 입력이 두 개 이상일 때 올림수, 즉 Carry를 의미하는 Cin가 1이 됨을 확인할 수 있고 세 입력의 합, 즉 S는 세 입력 중 참인 입력이 홀수 개일 때 1이 됨을 확인할 수 있습니다. 반가산기의 코드를 작성하기 위해서는 <Table 1>과 같은 진리표를 먼저 작성하고 SOP를 이용해 S는 A’B’Cin+A’BCin’+AB’Cin’+ABCin=Cin⊕A⊕B, Cout는 A’B’Cin’+AB’Cin+ABCin’+ABCin=AB+Cin(A⊕B)이라는 논리 함수로 나타낼 수 있다는 것을 통해 XOR 연산과 AND 연산을 사용하여 구할 수 있다는 것을 알 수 있습니다.

1. Full Subtracter 및 Half Subtractor의 simulation 결과 및 과정에 대해서 설명하시오(진리표 포함).

<Figure 5> Half subtractor schematic diagram

위의 <Figure 5>은 A비트에서 B비트를 빼는 반감산기를 구현하기 위해 Verilog 코드로, 입력 inA, inB에 대한 출력 outB, outD를 할당한 뒤 schematic을 확인한 결과입니다. outB를 얻기 위해서는 입력 inA에 NOT 연산을 한 후, inA’와 inB를 AND 연산하면 되고, outD를 얻기 위해서는 두 입력을 XOR 연산하면 된다는 사실을 알 수 있습니다. 이때, 반감산기에서 outB는 borrow로, 이진 연산에서 빌림수를 의미하고 outD는 해당 자릿수의 입력 차를 의미합니다. 이 Verilog 코드를 이용해서 simulation 결과를 확인하면 다음 <Figure 6>와 같습니다.



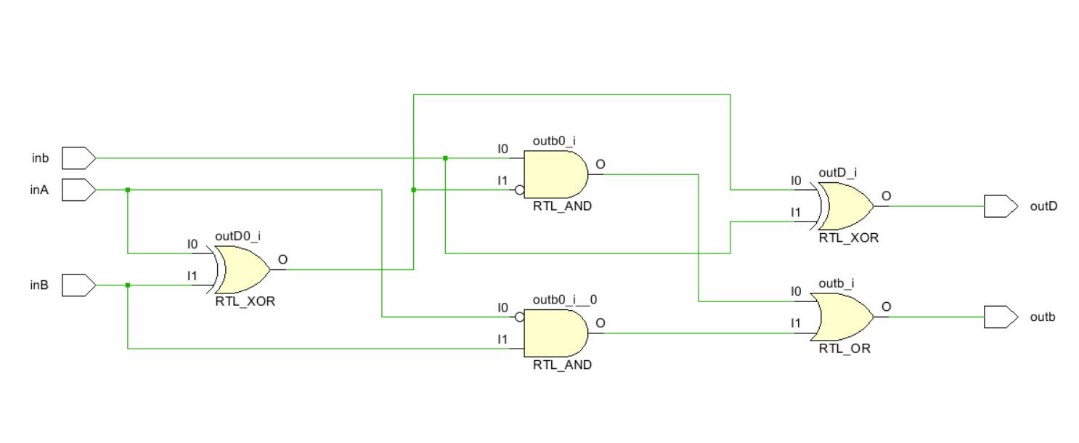
<Figure 6> Half subtractor simulation

이를 이용해 반감산기의 진리표를 작성해보면 다음 <Table 3>과 같습니다.

<Table 3> Half subtractor(반감산기)의 진리표

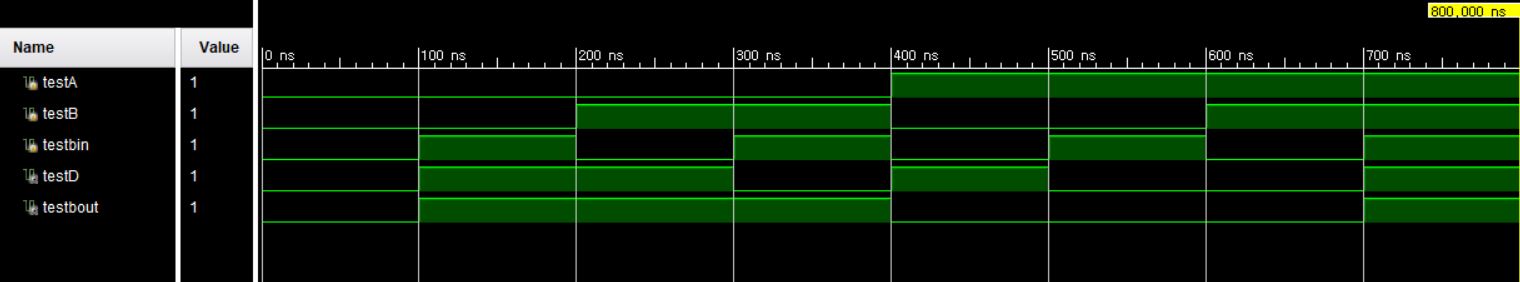
|  |  |  |  |
| --- | --- | --- | --- |
| X(A) | Y(B) | B | D |
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 0 | 0 |

위의 <Table 3>을 통해, X가 0이고 Y가 1일 때 빌림수, 즉 Borrow를 의미하는 B가 1이 됨을 확인할 수 있고 두 입력의 차, 즉 D는 두 입력 중 하나만 1일 때, 즉 입력 중 참인 입력이 홀수 개일 때 1이 됨을 확인할 수 있습니다. 반감산기의 코드를 작성하기 위해서는 <Table 3>과 같은 진리표를 먼저 작성하고 SOP를 이용해 B=X’Y, D는 X’Y+XY’이라는 논리 함수로 나타낼 수 있다는 것을 통해 NOT, AND 연산과 XOR 연산을 사용하여 구할 수 있다는 것을 알 수 있습니다.

하지만 반감산기도 반가산기와 마찬가지로 하위 자릿수의 빌림수를 알 수 없기 때문에 여러 비트의 연산에는 사용될 수 없다는 단점이 있습니다. 이와 같은 문제점을 보완한 감산기가 전감산기인데, 전감산기는 해당 자릿수의 비트, 즉 A와 B 뿐만 아니라 하위 자릿수의 빌림수 Bin을 입력으로 인가하여 출력을 얻을 수 있습니다. 다음 <Figure 7>은 전감산기의 schematic diagram을 나타낸 것입니다.

<Figure 7> Full subtractor schematic diagram

<Figure 7>, 전감산기의 schematic diagram을 보면 <Figure 5>과 다르게 입력이 하나 더 존재하는 것을 알 수 있습니다. 이는 앞서 말한 하위 자릿수의 빌림수를 의미하고 이러한 하위 자릿수의 빌림수가 입력으로 인가되기 때문에 전가산기는 여러 비트 연산이 가능합니다. 다음 <Figure 8>는 전감산기의 simulation 결과입니다.



<Figure 8> Full adder simulation

이를 이용해 전가산기의 진리표를 작성해보면 다음 <Table 4>와 같습니다.

<Table 4> 1 bit 전감산기의 진리표

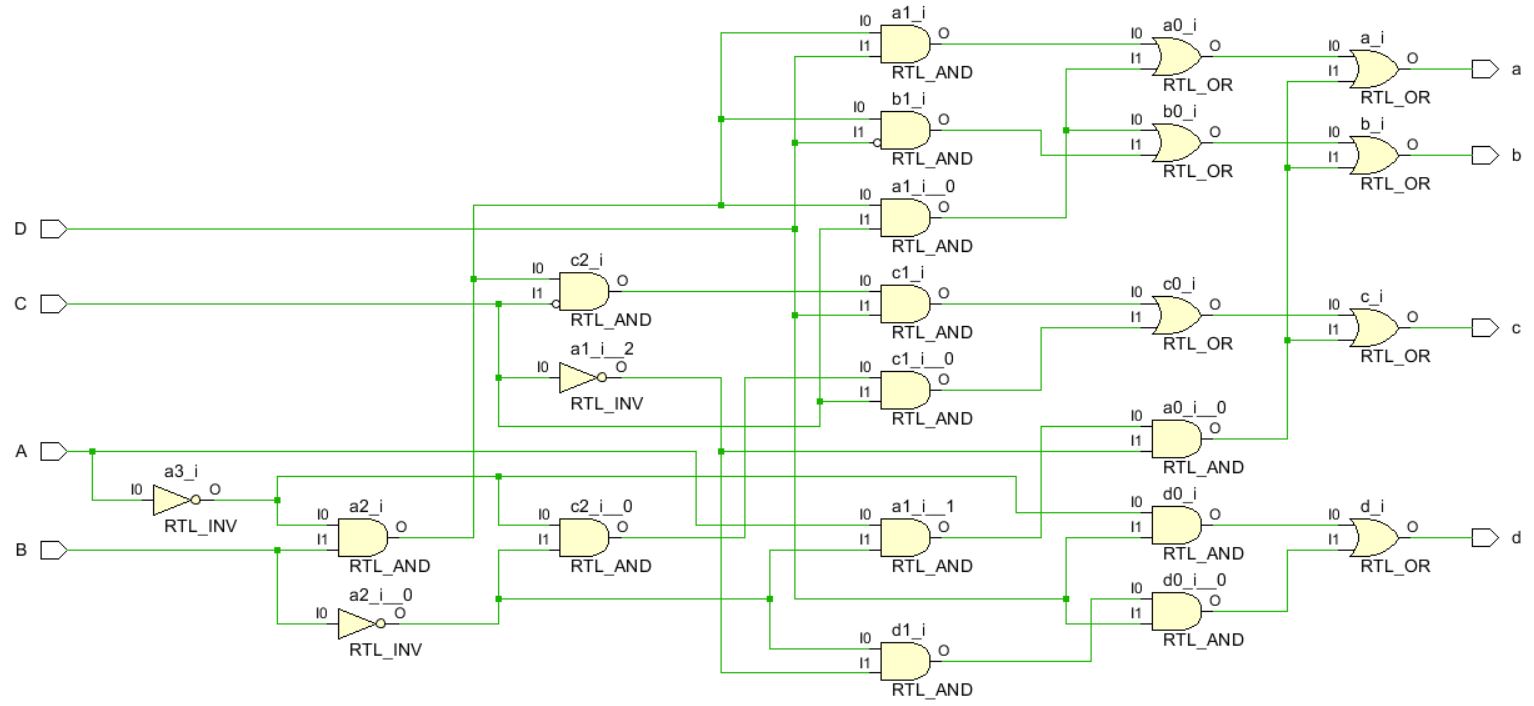
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| X(A) | Y(B) | Bin | Bout | D |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 1 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 |

위의 <Table 4>를 통해, 세 개의 입력 중 참인 입력이 홀수 개일 때 입력의 차, 즉 Difference를 의미하는 D가 1이 됨을 확인할 수 있습니다. 따라서, 전감산기의 D는 D=X⊕Y⊕Bin라는 논리 함수를 이용해 구현할 수 있고, 전감산기의 Bout은 위의 전감산기 진리표 <Table 4>와 SOP을 이용하면 Bout=X’Y’Bin+X’YBin’+X’YBin+XYBin이라는 논리 함수를 이용해 구현할 수 있습니다. 이때, Bout을 구하는 논리 함수는 조금 복잡하므로 카르노맵의 방법을 이용해 논리 함수를 최적화하기 위해 다음 <Table 5>와 같은 카르노맵을 작성할 수 있습니다.

<Table 5> Bout을 구하기 위한 카르노맵

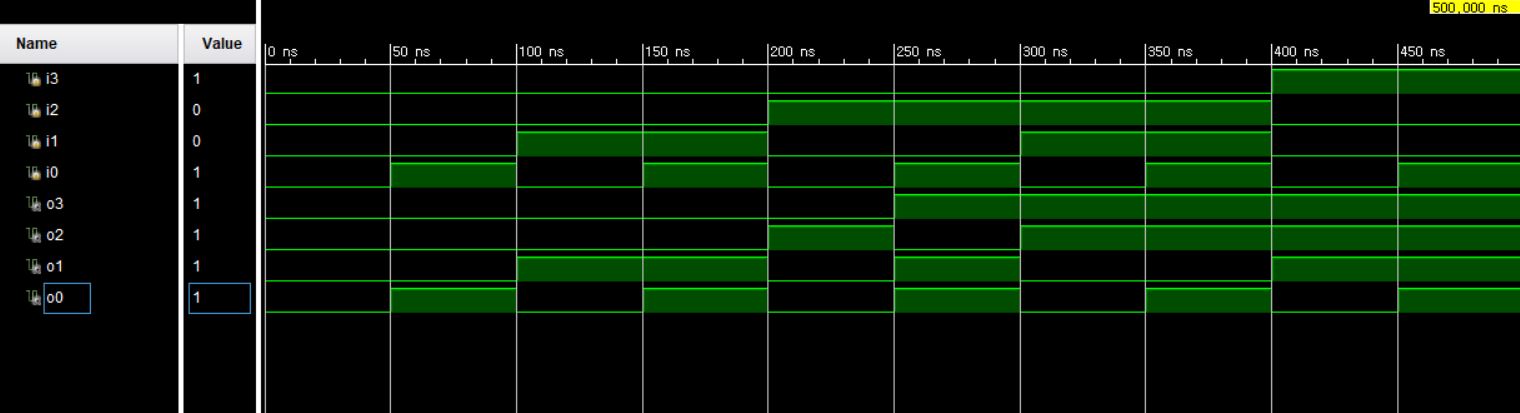
|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Bi\XY | 00 | 01 | 11 | 10 |
| 0 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 | 0 |

위의 <Figure 7>로부터 빌림수 Bout는 B=X’Bin+YBin+X’Y라고 정리할 수 있습니다. 이는 <Table 4>와 SOP를 이용해 구한 Bout=X’Y’Bin+X’YBin’+X’YBin+XYBin 논리 함수보다 훨씬 간단하고 논리 함수를 보고 Boolean Algebra를 이용해 최적화할 수 있지만 카르노맵을 사용하여 간편하게 논리 함수를 최적화하는 방법을 사용할 수도 있습니다.

1. 8421(BCD)-2421 Code converter simulation 결과 및 과정에 대해서 설명하시오(진리표 작성 및 카르노맵 SOP form, POS form 포함).

<Figure 9> 8421-2421 code converter schematic diagram

BCD(Binary Coded Decimal) 중 8421 code를 2421 code로 반전시키는 convert를 Verilog로 구현해보고 simulation 결과를 확인해보는 실험을 하였습니다. 예를 들어, 2421 code의 10진수 n을 얻기 위해서는 n을 8421 code로 나타낸 것의 1의 보수를 구하는 것이 아니라 9-n을 8421 code로 나타낸 것의 1의 보수를 구하는 것입니다. 따라서 십진수 0부터 4까지는 8421 code와 2421 code가 동일하지만 5부터 9까지는 두 code가 서로 전혀 다른데 2421 code로 5는 9-5로 4, 즉 0100의 1의 보수인 1011으로 나타내고 6은 9-6로 3, 즉 0011의 1의 보수인 1100, 7은 9-7로 2, 즉 0010의 1의 보수인 1101으로 나타내고 8은 9-8로 1, 즉 0001의 보수인 1110, 9는 1111으로 나타냅니다. 실제로 위의 <Figure 9>의 simulation 결과가 이러한 2421 code와 동일한지 확인하기 위해 simulation을 해보면 다음 <Figure 10>과 같은 결과를 얻을 수 있었습니다.



<Figure 10> 8421-2421 code converter simulation

이를 이용해 8421-2421 code의 진리표를 작성해보면 다음 <Table 6>과 같습니다.

<Table 6> 8421-2421 code converter의 진리표

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 십진수 | A | B | C | D | a | b | c | d |
| 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 |

<Table 6>은 앞서 언급한 0부터 8까지의 십진수에 따른 2421 code와 동일하므로 올바른 결과를 얻었음을 알 수 있습니다. 역으로 위의 진리표를 이용하여 8421-2421 code converter를 Verilog 코드를 통해 구현하기 위해 각 출력의 논리 함수를 구해보려면 카르노맵을 이용해야 합니다.

다음 표는 a에 대한 논리 함수를 구하기 위한 카르노맵입니다.

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

위의 카르노맵을 통해 SOP으로 a에 대한 논리 함수를 표현해보면 a=A’BD+A’BC+AB’C’이 되고, POS으로 a에 대한 논리 함수를 표현해보면 a=(A+B)(A+C+D)(B’+C+D)(A’+B’)(A’+C)가 되는데, 이는 a=(A+BC+BD)=(A’+B’C)(B’+C+D)=(AA’+AB’C+A’BC+BB’C+A’BD+BB’CD)(B’+C+D)=(AB’C+A’BC+ A’BD)(B’+C+D)=AB’C+AB’C+AB’CD+A’BC+A’BCD+A’BCD+A’BD=AB’C+AB’CD+A’BC+A’BCD+A’BCD+A’BD=AB’C+A’BC+A’BD가 되므로 SOP로 a를 표현한 것과 동일한 논리 함수임을 알 수 있습니다.

다음 표는 b에 대한 논리 함수를 구하기 위한 카르노맵입니다.

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

위의 카르노맵을 통해 SOP으로 b에 대한 논리 함수를 표현해보면 b=A’BC+A’BD’+AB’C’가 되고, POS으로 b에 대한 논리 함수를 표현해보면 b=(A+B)(A+C+D’)(A’+B’)(B’+C+D’)(A’+C’)=(A+BC+BD’) (B’+A’C+A’D’)(A’+C’)=(AB’+A’BC+A’BCD’+A’BCD’+A’BD’)(A’+C’)=AB’C’+A’BC+A’BCD’+A’BCD’+A’BD’+A’BC’D’= A’BC+AB’C’+A’BD’가 되므로 SOP로 b를 표현한 것과 동일한 논리 함수임을 알 수 있습니다.

다음 표는 c에 대한 논리 함수를 구하기 위한 카르노맵입니다.

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

위의 카르노맵을 통해 SOP으로 c에 대한 논리 함수를 표현해보면 c=A’BC’D+A’B’C+AB’C’가 되고, POS으로 c에 대한 논리 함수를 표현해보면 c=(A+B+C)(A’+B’)(A+C+D)(B’+C+D)(B’+C’)(A’+C’)= (A+C+BD)(A’+B’)(B’+C’)(A’+C’)=(A+C+BD)(A’+B’C’)(B’+C’)=(AB’C’+A’C+A’BD)(B’+C’)=AB’C’+A’B’C+A’BC’D가 되므로 SOP로 c를 표현한 것과 동일한 논리 함수임을 알 수 있습니다.

다음 표는 d에 대한 논리 함수를 구하기 위한 카르노맵입니다.

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

위의 카르노맵을 통해 SOP으로 d에 대한 논리 함수를 표현해보면 d=A’D+B’C’D가 되고, POS으로 d에 대한 논리 함수를 표현해보면 d=(C+D)(A’+B’)(C’+D)(A’+C’)=(A’+B’C’)D=A’D+B’C’D가 되므로 SOP로 d를 표현한 것과 동일한 논리 함수임을 알 수 있습니다.

1. 결과 검토 및 논의사항

전, 반가산기, 전, 반감산기, 8421-2421 code converter가 어떤 일을 하는 논리 회로인지를 알기 때문에 입력에 대한 출력을 예측할 수 있었고, 그 예측을 진리표로 작성하여 카르노맵을 통해 각 출력에 대한 논리 함수로 정리하여 Verilog로 코딩할 수 있었습니다. 본 실험에서 다룬 가산기, 감산기, code converter 이외에도 다양한 논리 회로에 대해 입력에 따른 출력을 예상하거나 이미 알고 있다면 어렵지 않게 코딩할 수 있고, test bench를 이용해 simulation함으로써 실제로 각 출력에 대한 논리 함수를 알맞게 구하였는지 검증할 수 있습니다. 하지만 입력의 개수가 7개 이상으로 늘어나면 실질적으로 카르노맵을 그리는 데에 무리가 있어 더 이상 카르노맵을 통해 canonical form으로 논리 함수를 구할 수 없고 5주차 예비보고서에서 다룬 Quine-McCluskey 최소화 알고리즘을 이용하여야 합니다.

1. 추가 이론 조사 및 작성

카르노맵을 통해 canonical form으로 논리 함수를 표현하는 것은 주로 SOP 형태로 표현되기 때문에 SOP 형태의 논리 함수를 구하는 것은 익숙하지만 POS 형태의 논리 함수를 구하는 것은 익숙하지 않을 수 있습니다. 따라서, 추가 이론 조사 및 작성에서는 카르노맵을 통해 POS 형태로 논리 함수를 표현하는 방법에 대해 설명해보겠습니다.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| A | B | C | F | F’ |
| 0 | 0 | 0 | 0 | 1 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 0 |

위와 같은 진리표가 있을 때, SOP(sum of product)로는 출력 F에 대한 논리 함수를 나타내기 위해 출력 F를 1로 만드는 입력들에 대한 곱의 합으로 표현하면 되므로 F=A’B’C+A’BC+AB’C+ABC’+AB C와 같이 나타내면 되고 POS(product of sum)로는 출력 F를 0으로 만드는, 즉 F’을 1로 만드는 입력들에 대한 합의 곱으로 표현하면 되므로 F=(A+B+C)(A+B’+C)(A’+B+C)와 같이 나타내면 됩니다. 이때, POS의 형태로 F를 나타내기 위해서는 F’의 SOP을 구하여 그 논리 함수에 보수를 취해주면 됩니다.

이러한 SOP와 POS의 형태로 논리 함수를 구하는 기본적인 방법을 응용하여 다음은 아래와 같은 진리표를 보고 SOP, POS 형태로 논리 함수를 구해보겠습니다.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| A | B | C | D | F |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 |
| 0 | 1 | 1 | 0 | 0 |
| 0 | 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 0 | 0 |
| 1 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 | 1 |

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

위의 표는 위의 진리표를 카르노맵으로 표현한 것으로, 익숙한 SOP 형태로 F에 대한 논리 함수로 나타내보면 F=BD라고 표현할 수 있습니다. 하지만 POS 형태로 F를 표현하기 위해서는 F를 0으로 만드는 입력의 합의 곱으로 나타내면 되므로 F=(C+D)(A+B)(A’+B)(C’+D)(=BD)라고 나타내도 되고, F’에 대한 카르노맵을 다음과 같이 표현한 후, SOP를 구한 후 그 논리 함수에 보수를 취해도 됩니다.

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

위의 카르노맵을 통해 F’=A’B’+C’D’+CD’+AB’을 구할 수 있고, F’에 보수를 취해 F’의 보수인 F=(F’)’=(A’B’+C’D’+CD’+ AB’)’=(A+B)(C+D)(C’+D)(A’+B)를 구할 수 있습니다. 위의 두 방법으로 구한 F에 대한 논리 함수는 서로 동일하고 SOP의 형태로 구한 논리 함수와 POS의 형태로 구한 논리 함수가 서로 다르지 않음을 알 수 있습니다.